External Sorting

Database System Technology

Niv Dayan

Midterm next week

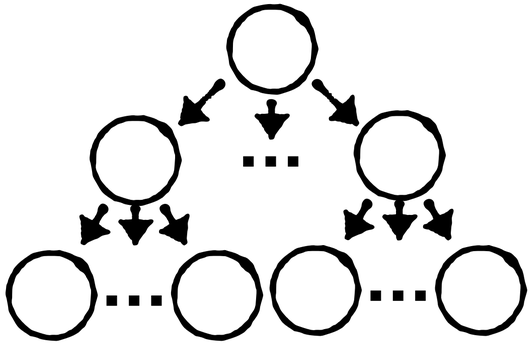
TA office hours after class

Extra office hours (TBD)

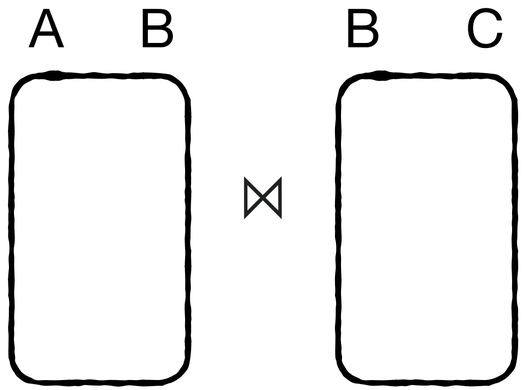
Why do databases need to Sort?

User asking for sorted output (Select…order by…)

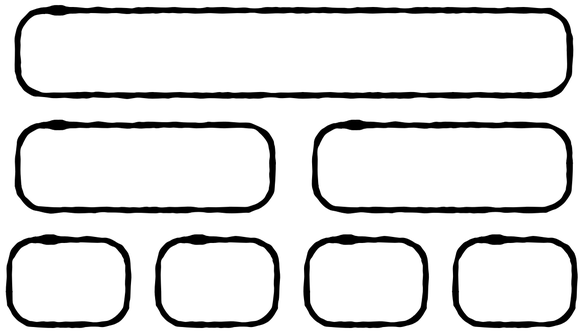
Creating an Index



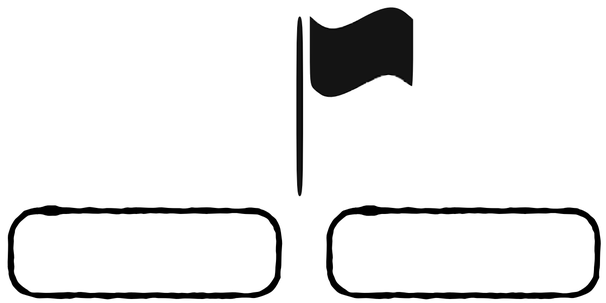
Joining Tables (More later)



If data fits in memory, traditional sorting algorithms work



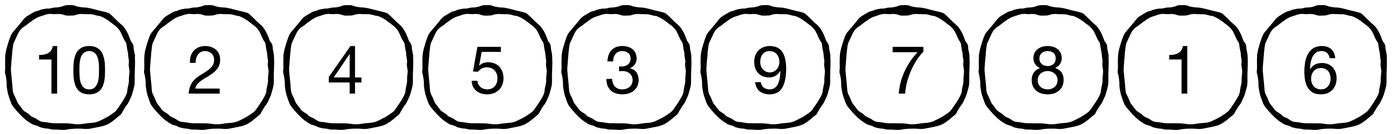
Merge-Sort



Quick-Sort

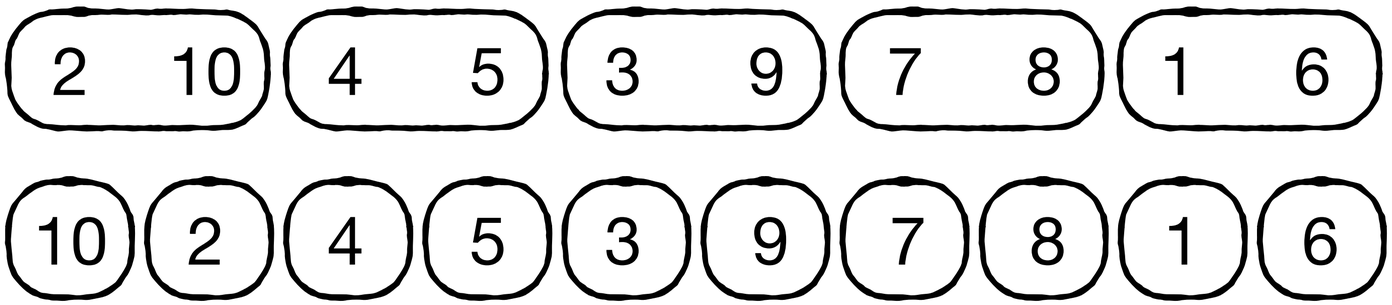
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.



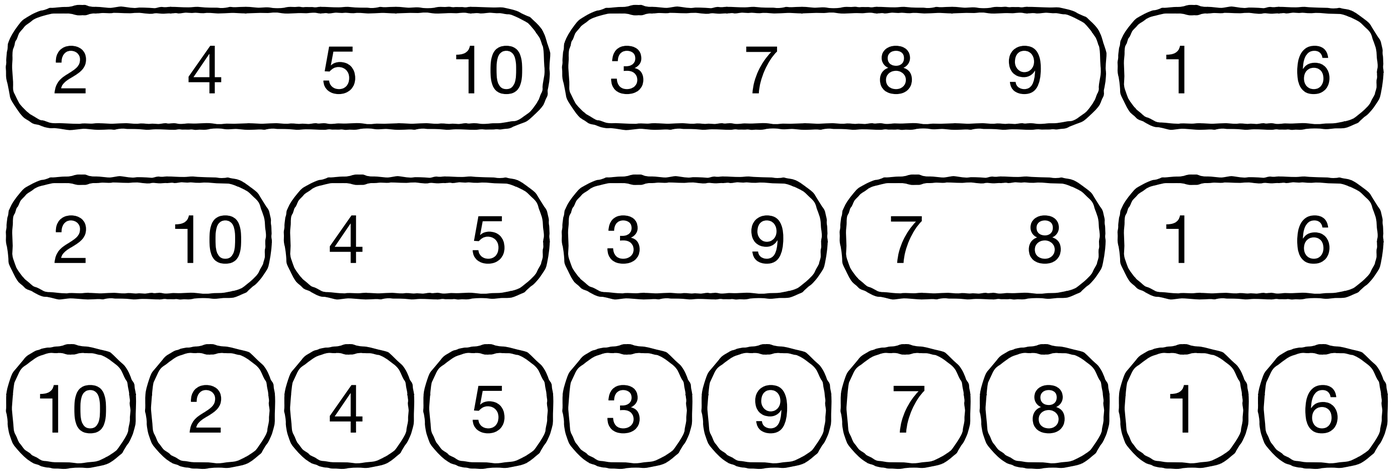
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left



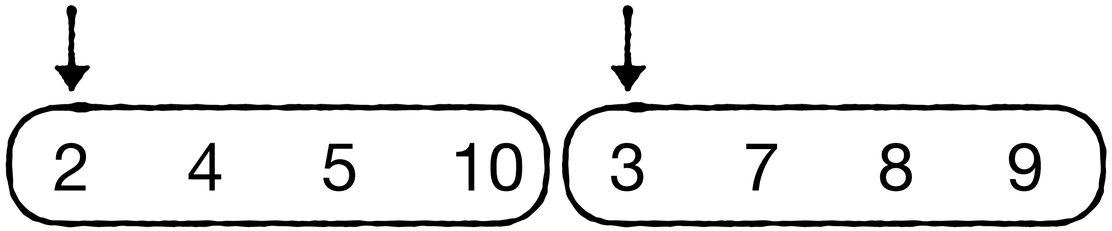
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left



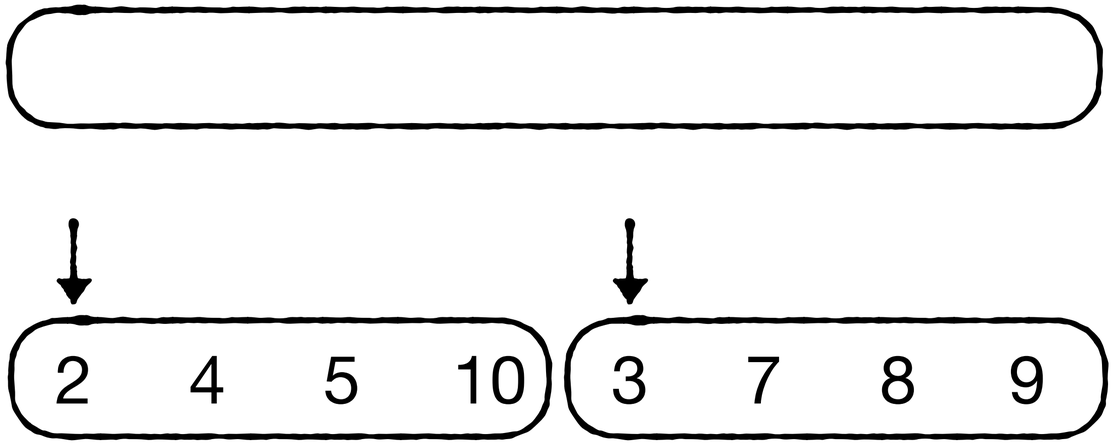
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left
3. Using cursors, draw the smallest element of the two lists each step



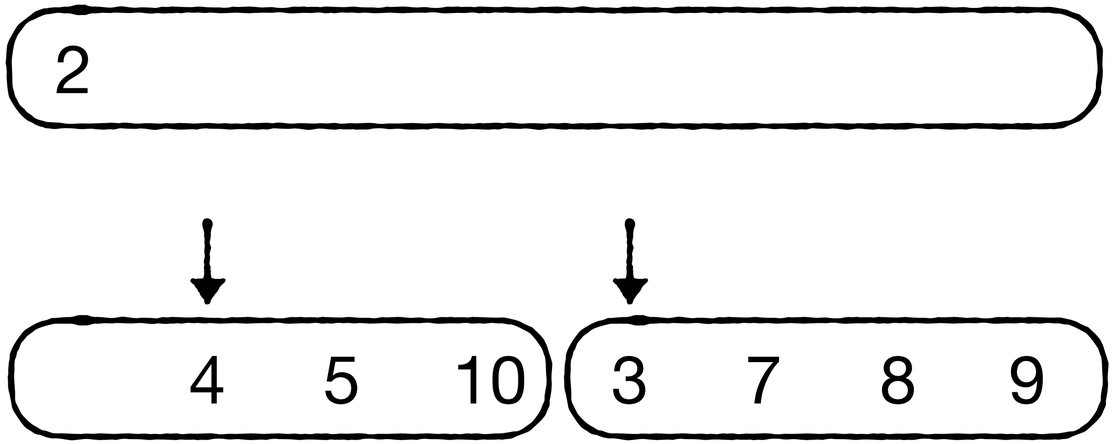
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left
3. Using cursors, draw the smallest element of the two lists each step



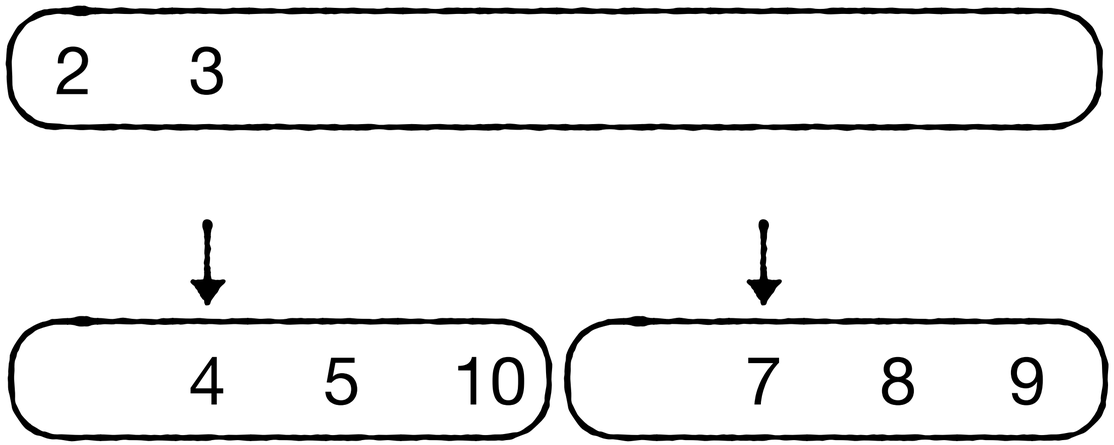
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left
3. Using cursors, draw the smallest element of the two lists each step



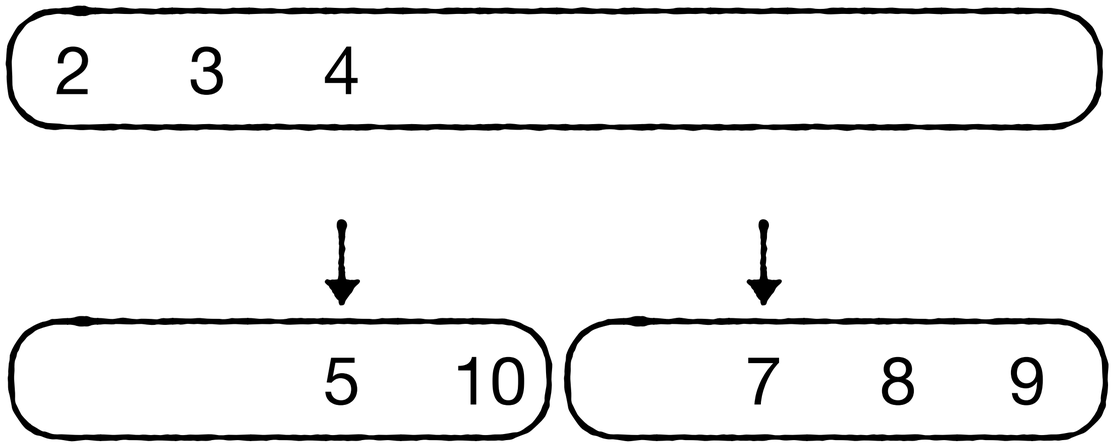
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left
3. Using cursors, draw the smallest element of the two lists each step



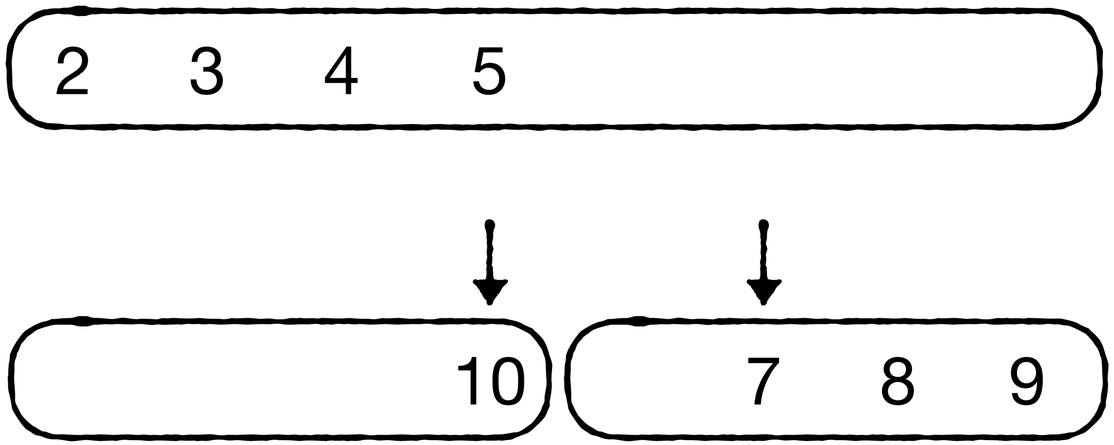
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left
3. Using cursors, draw the smallest element of the two lists each step



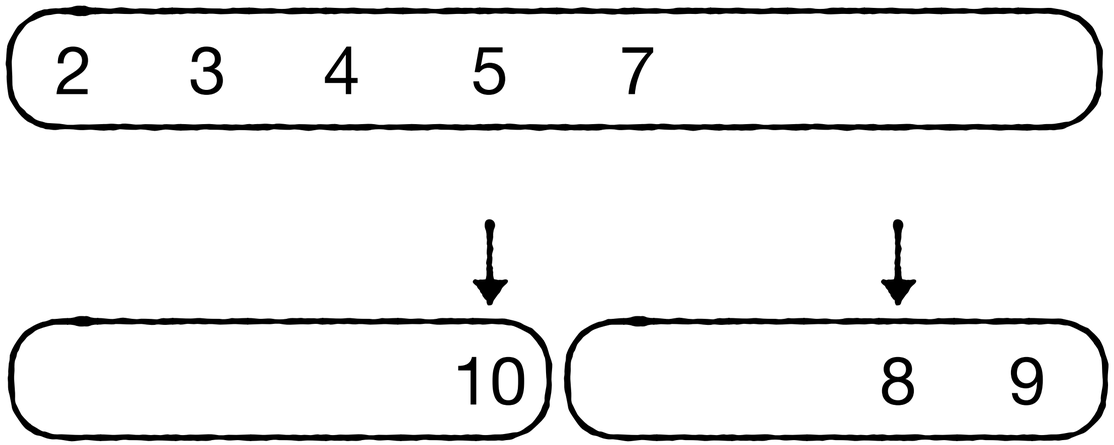
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left
3. Using cursors, draw the smallest element of the two lists each step



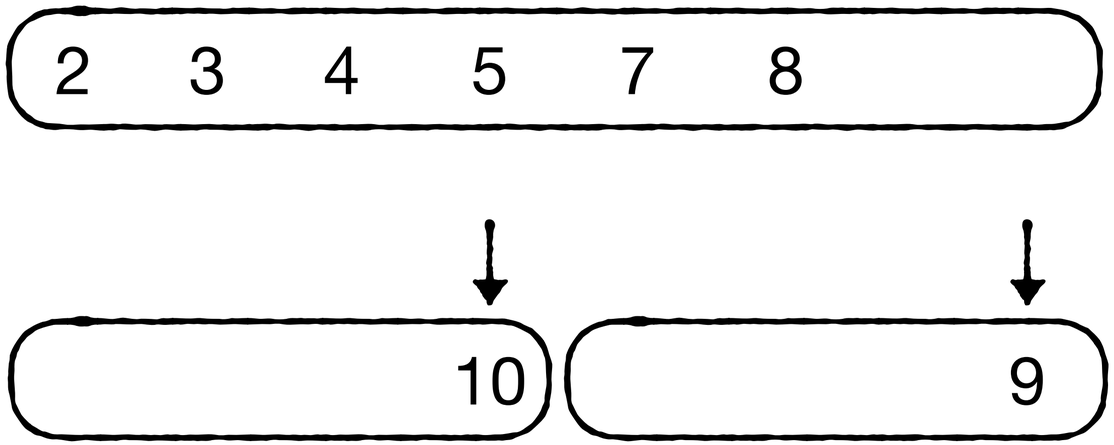
Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left
3. Using cursors, draw the smallest element of the two lists each step



Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left
3. Using cursors, draw the smallest element of the two lists each step

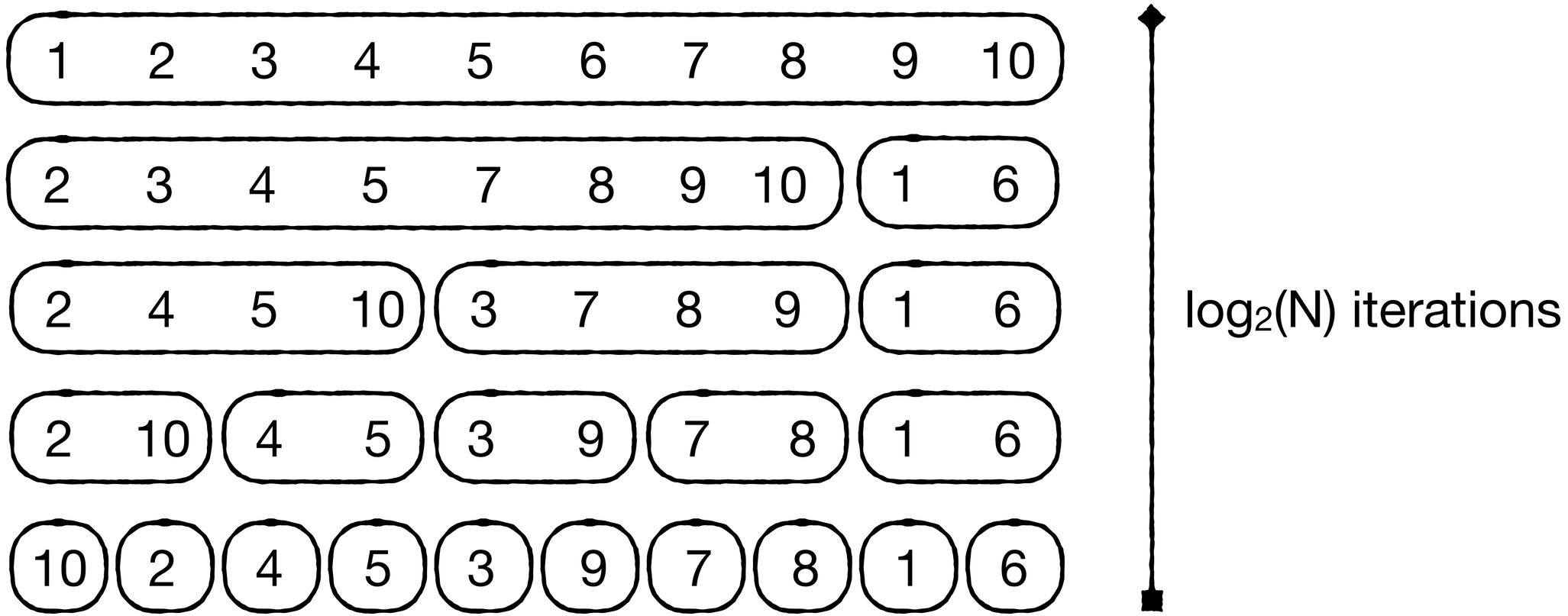


Merge-Sort

1. Given n entries, divide them into n sublists, each containing one element.
2. Merge two adjacent sublists at a time until there is one left
3. Using cursors, draw the smallest element of the two lists each step

Merge-Sort Analysis

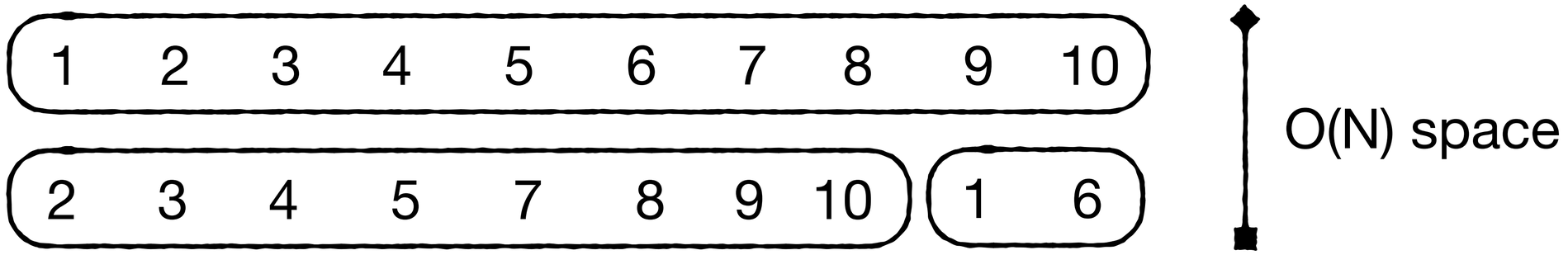
Log(N) iterations, each of which traverses all N elements: O(N $\log_2(N)$) CPU work



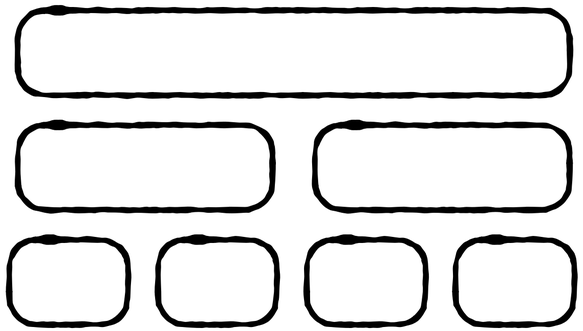
Merge-Sort Analysis

Log(N) iterations, each of which traverses all N elements: O(N $\log_2(N)$) CPU work

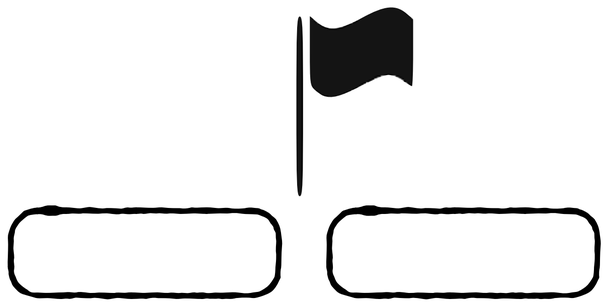
We need O(N) space by maintaining at most two lists at a time



If data fits in memory, traditional sorting algorithms work



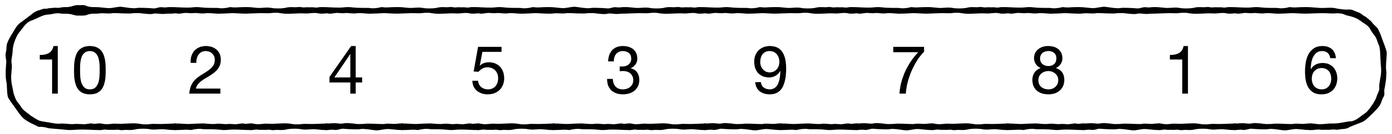
Merge-Sort



Quick-Sort

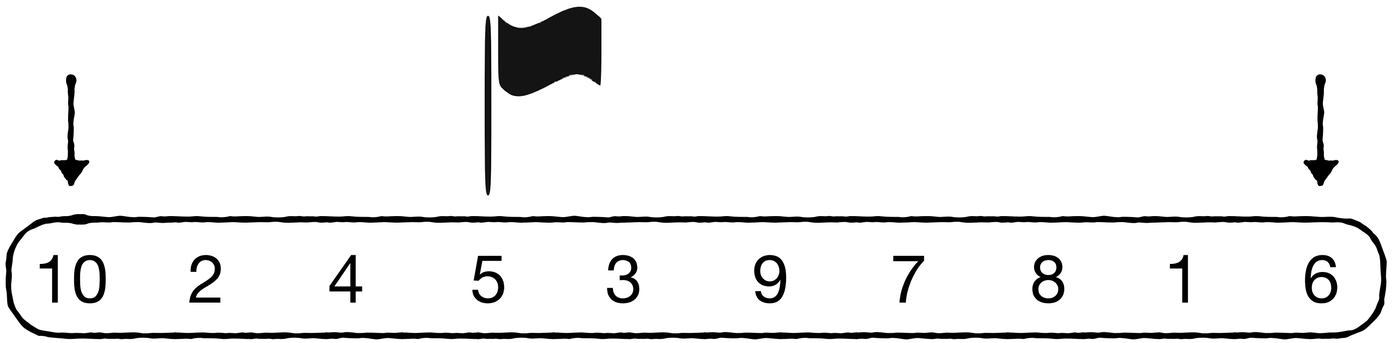
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends



Quick-Sort

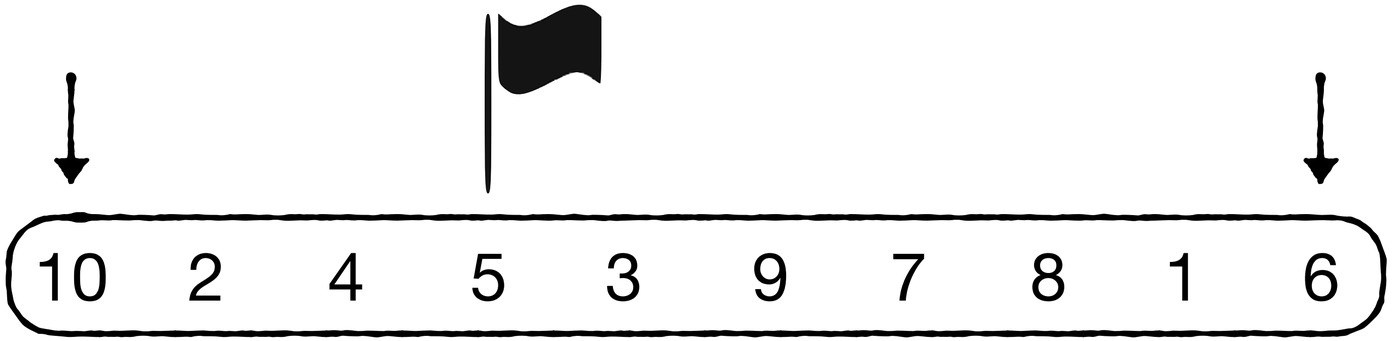
1. Pick random pivot point and initialize two pointers at ends



Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

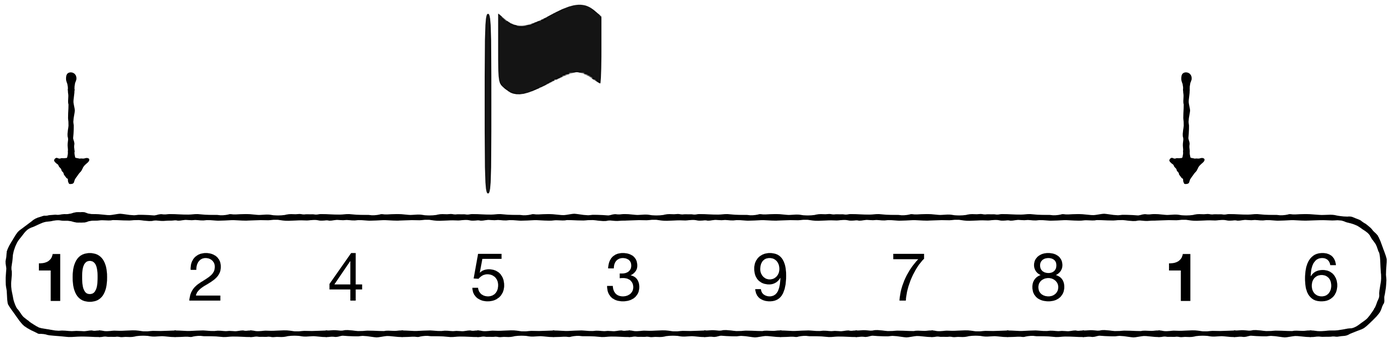
2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot



Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

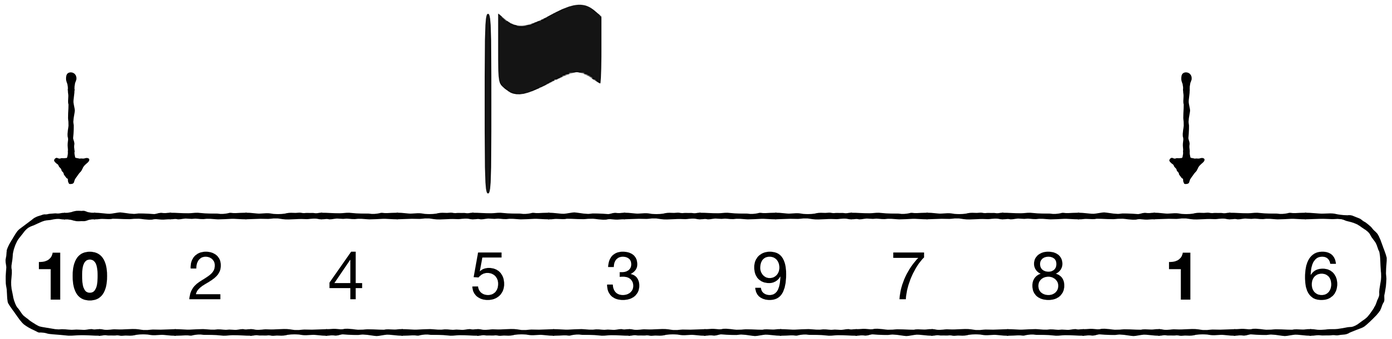


Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

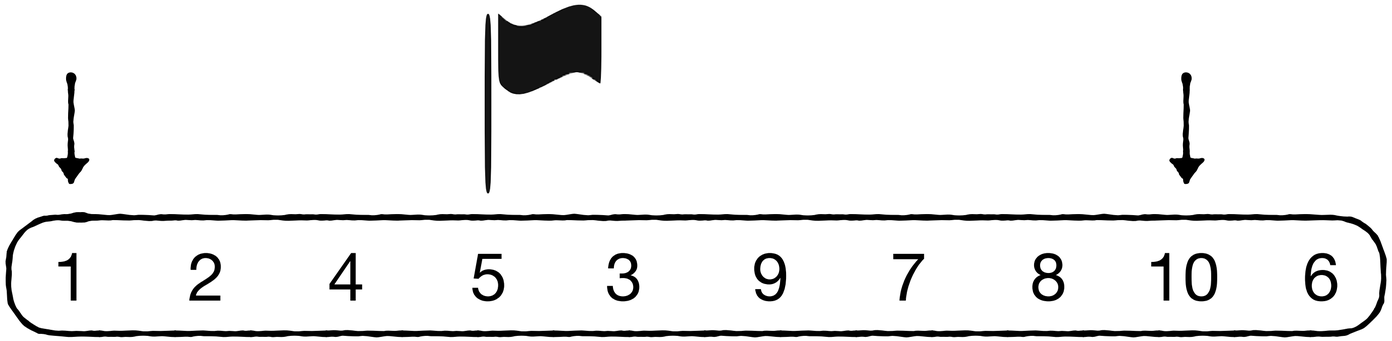


Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

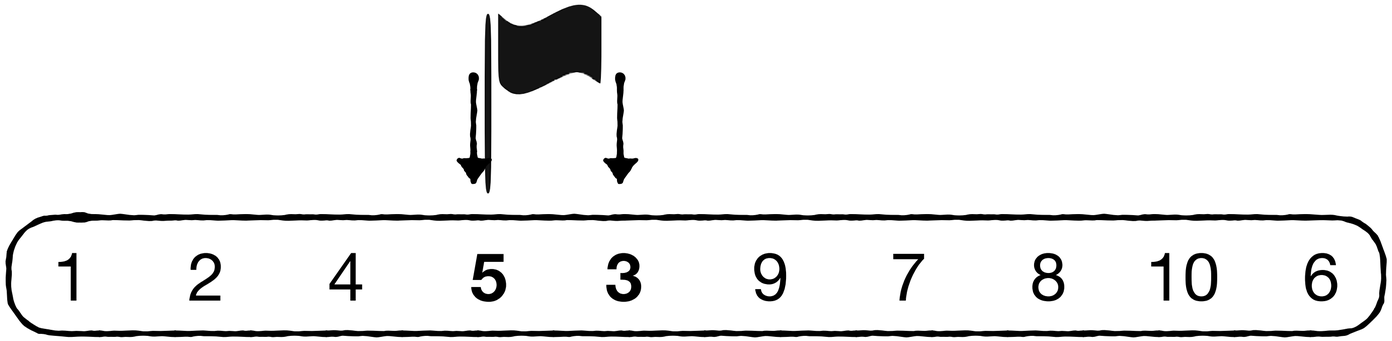


Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

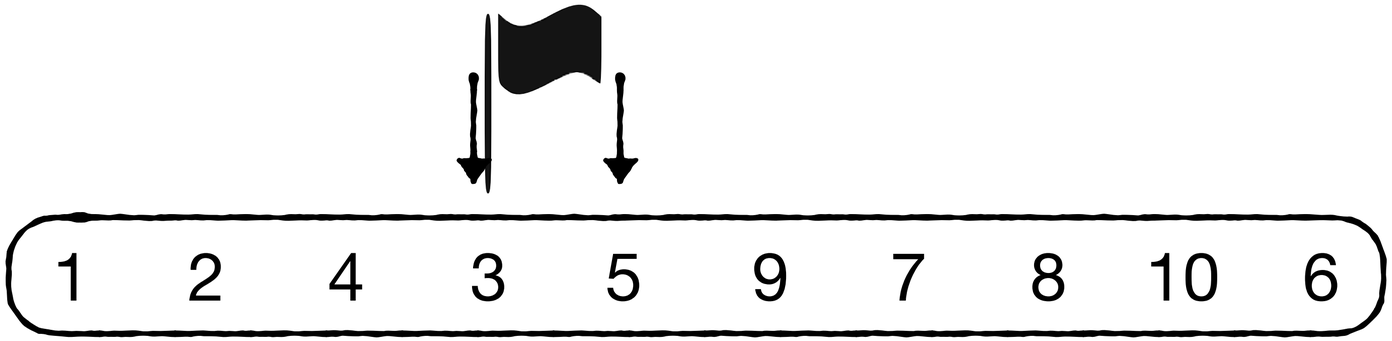


Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue



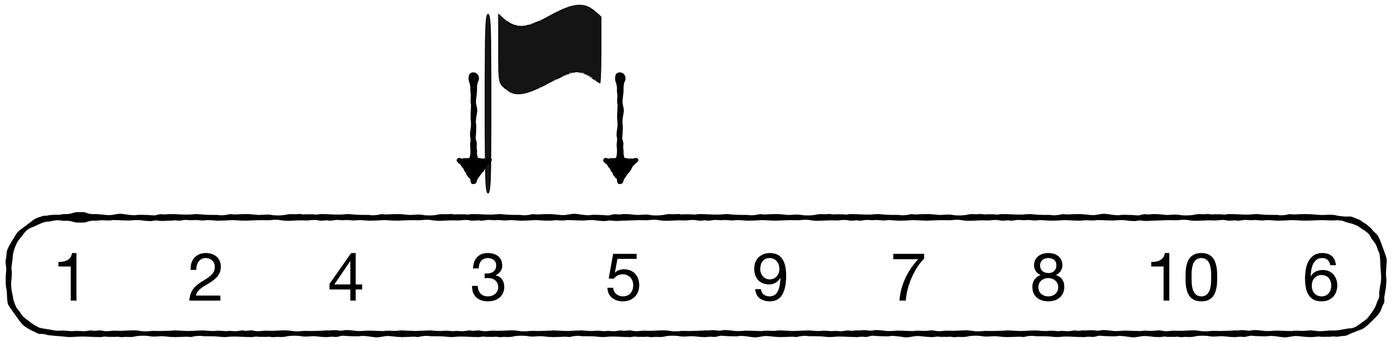
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

4. Continue recursively on both partitions around pivot.



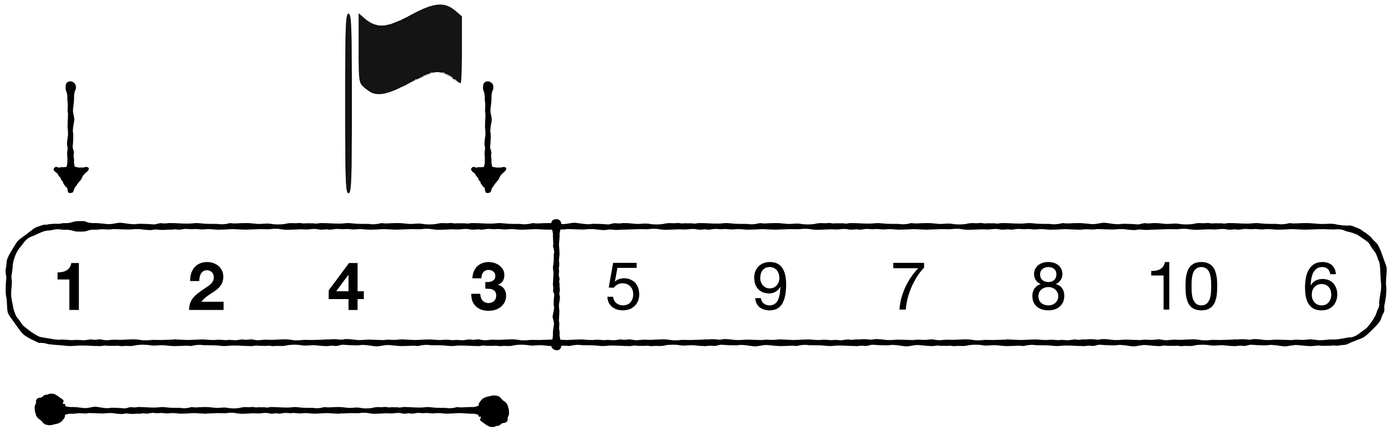
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

4. Continue recursively on both partitions around pivot.



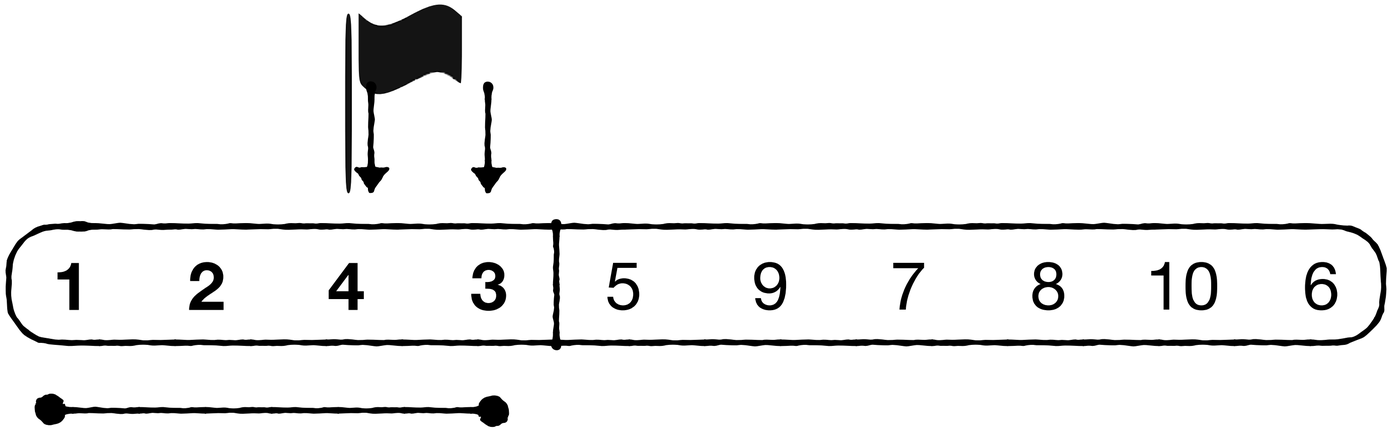
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

4. Continue recursively on both partitions around pivot.



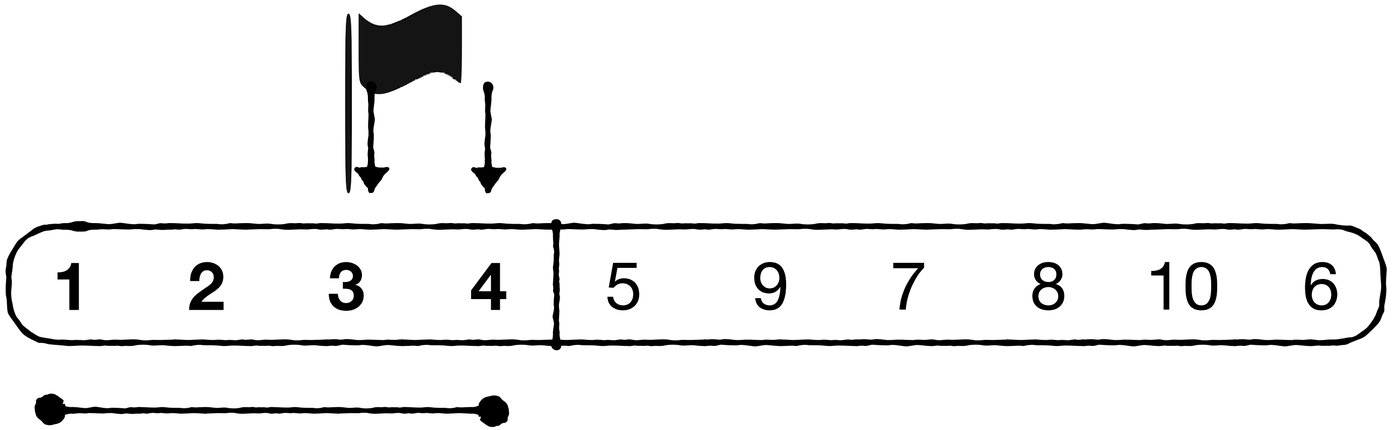
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

4. Continue recursively on both partitions around pivot.



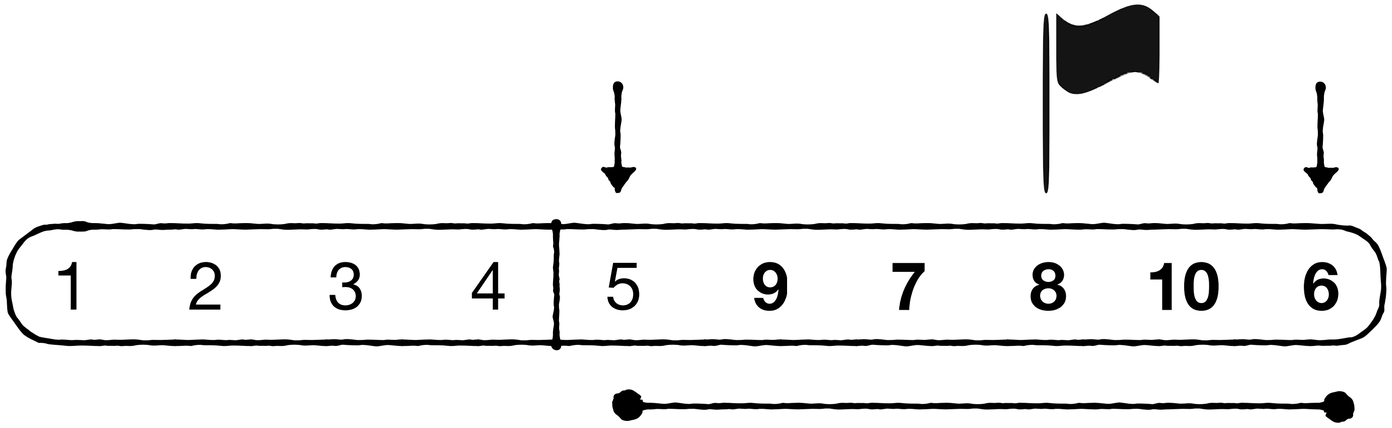
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

4. Continue recursively on both partitions around pivot.



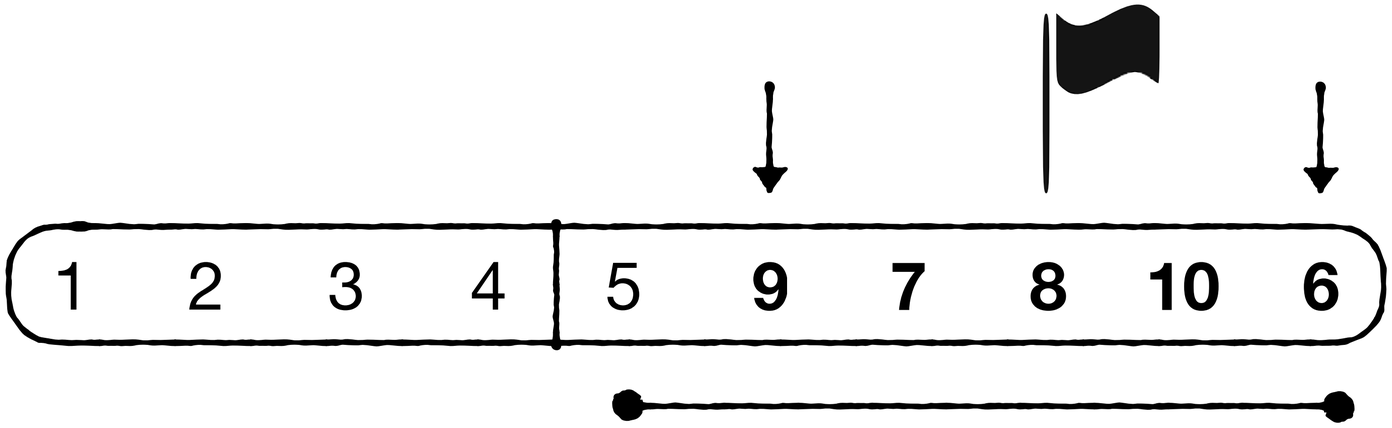
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

4. Continue recursively on both partitions around pivot.



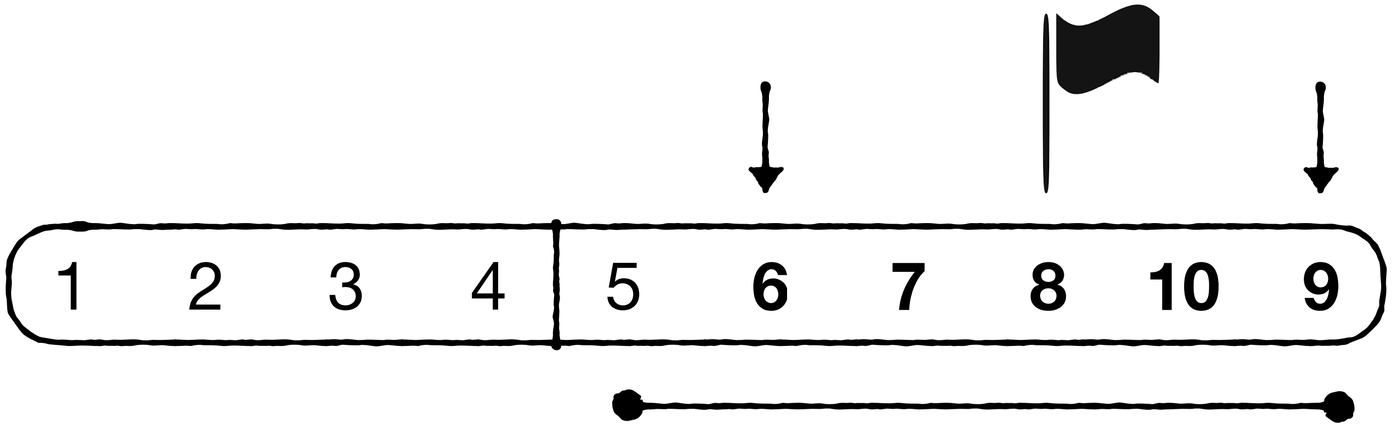
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

4. Continue recursively on both partitions around pivot.



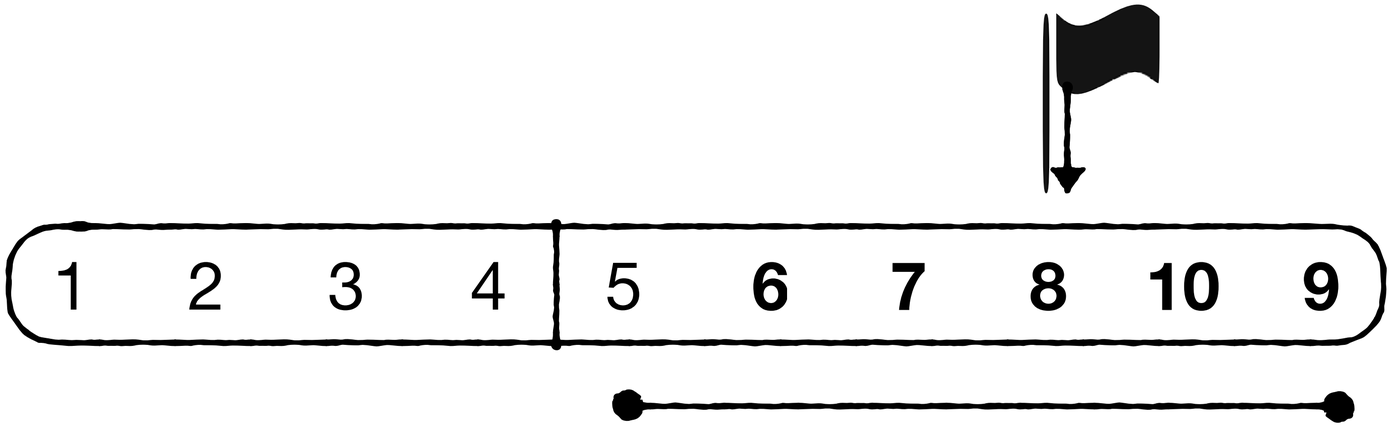
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

3. Swap pair of out-of-order entries and continue

4. Continue recursively on both partitions around pivot.



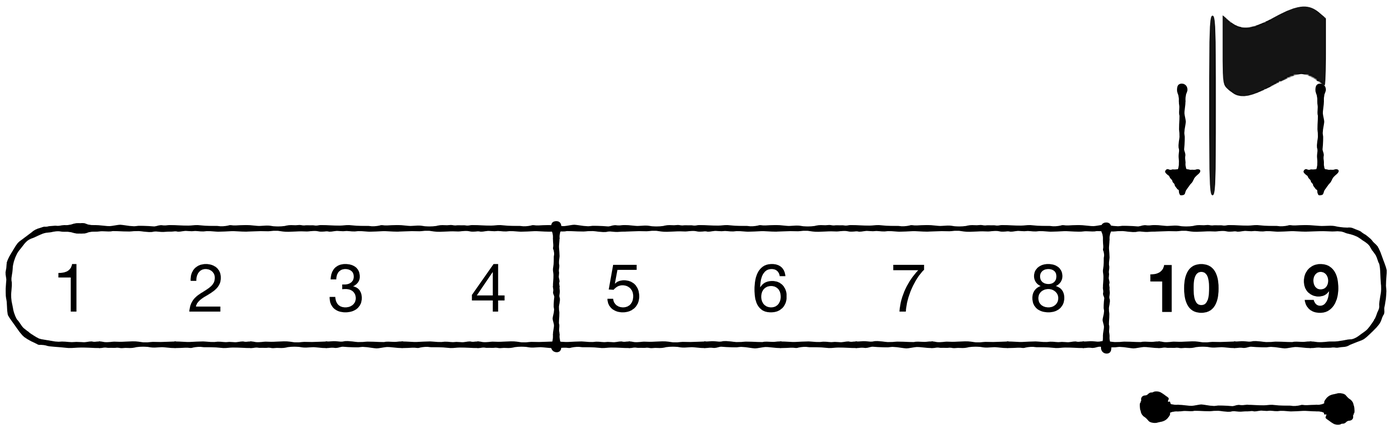
Quick-Sort

1. Pick random pivot point and initialize two pointers at ends

2. Move each pointer towards pivot, stopping at first out-of-order value respect to pivot

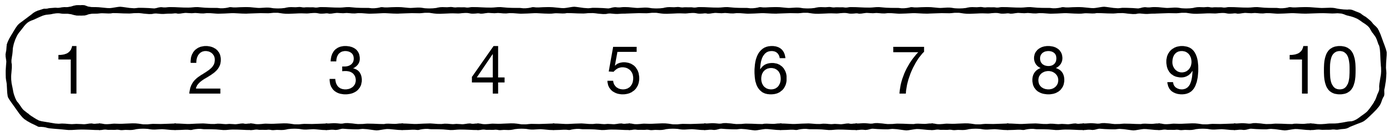
3. Swap pair of out-of-order entries and continue

4. Continue recursively on both partitions around pivot.



Quick-Sort

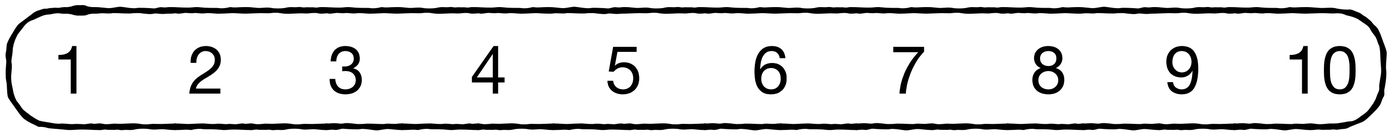
Properties: **Expected $O(N \log_2 N)$ worst-case**



Quick-Sort

Properties: **Expected O(N $\log_2$ N) worst-case**

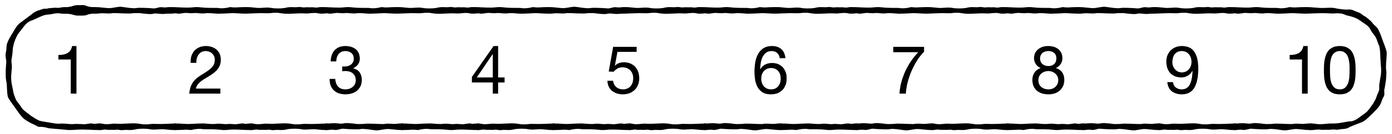
Can it be worse?



Quick-Sort

Properties: Expected $O(N \log_2 N)$ worst-case

But can be $O(N^2)$ with low probability

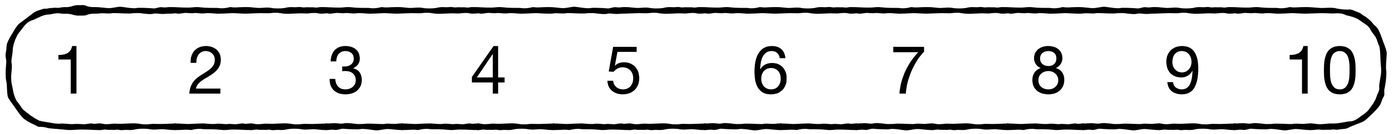


Quick-Sort

Properties: Expected $O(N \log_2 N)$ worst-case

But can be $O(N^2)$ with low probability

(e.g., always pick minimum key as pivot)



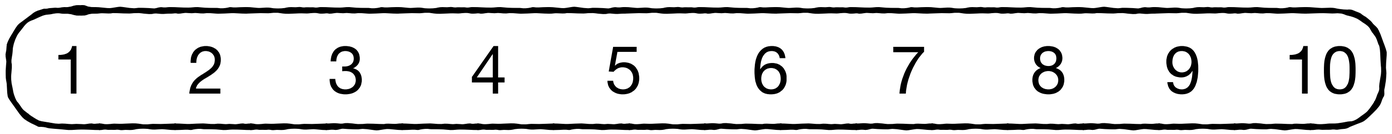
Quick-Sort

Properties: Expected $O(N \log_2 N)$ worst-case

But can be $O(N^2)$ with low probability

(e.g., always pick minimum key as pivot)

Also uses sequential access, which is fast



Quick-Sort

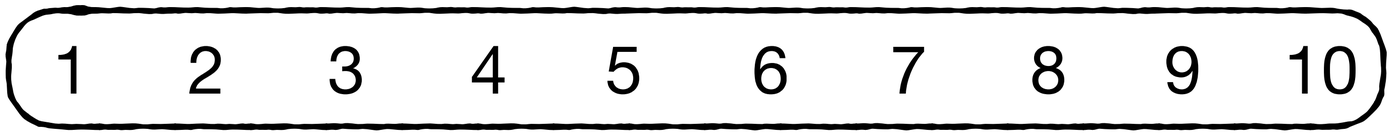
Properties: Expected $O(N \log_2 N)$ worst-case

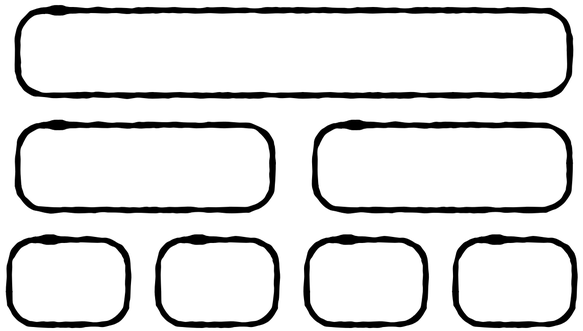
But can be $O(N^2)$ with low probability

(e.g., always pick minimum key as pivot)

Also uses sequential access, which is fast

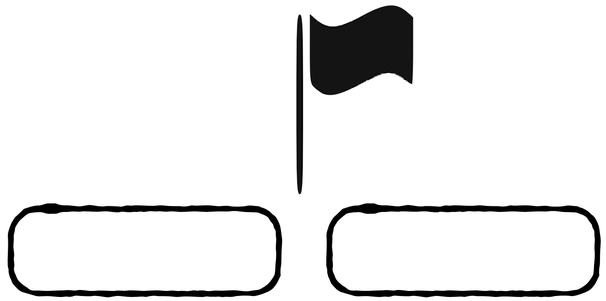
In-place algorithm: no need for x2 space like merge-sort





Merge-Sort

(More robust performance)



Quick-Sort

(Less space & faster on avg.)

But what if data does not fit in memory?

M/B pages

Unordered data (N/B pages)

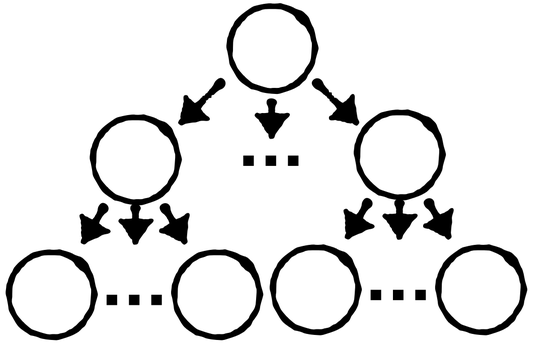
But what if data does not fit in memory?

M/B pages

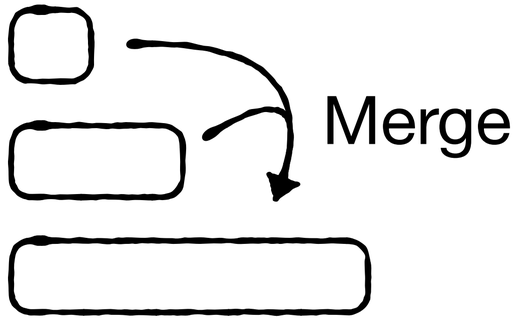
Unordered data (N/B pages)

how to sort based on things we've seen in class?

Baselines

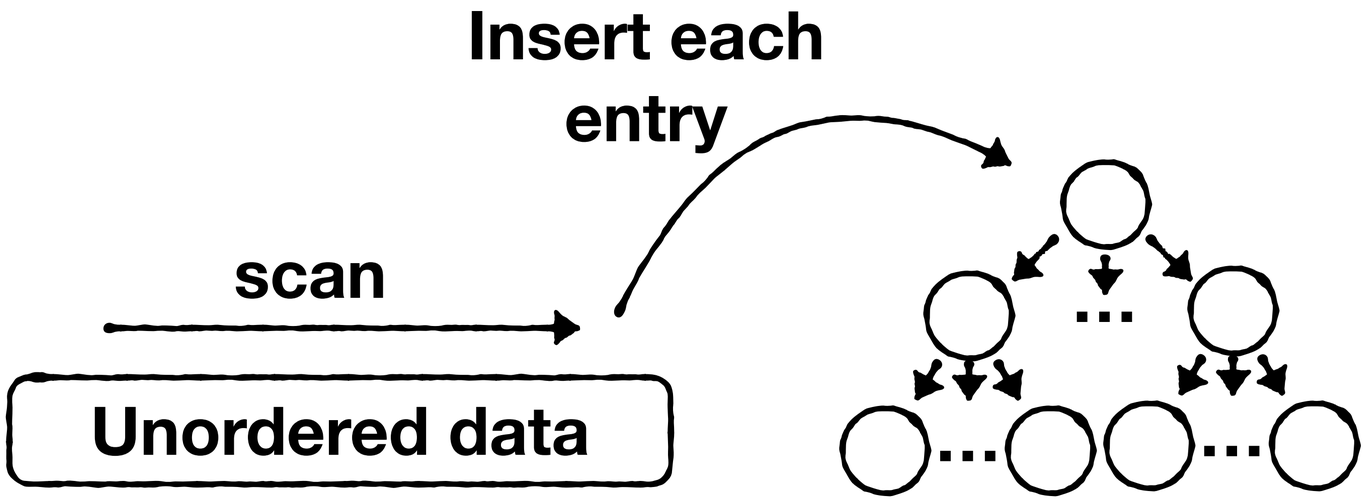


B-Tree

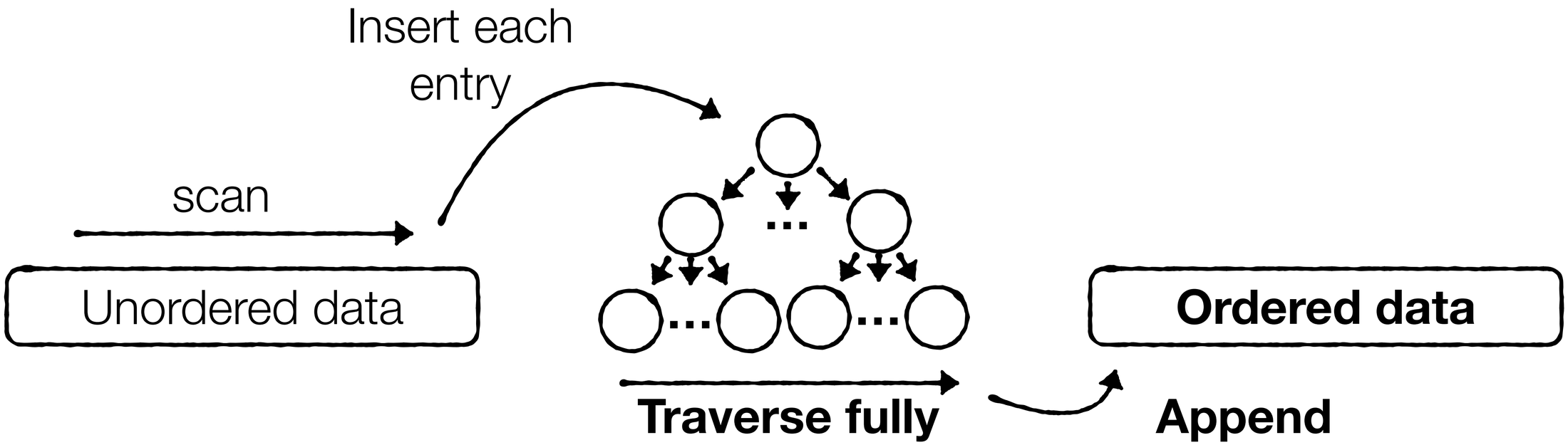


Log-Structured
Merge-Tree

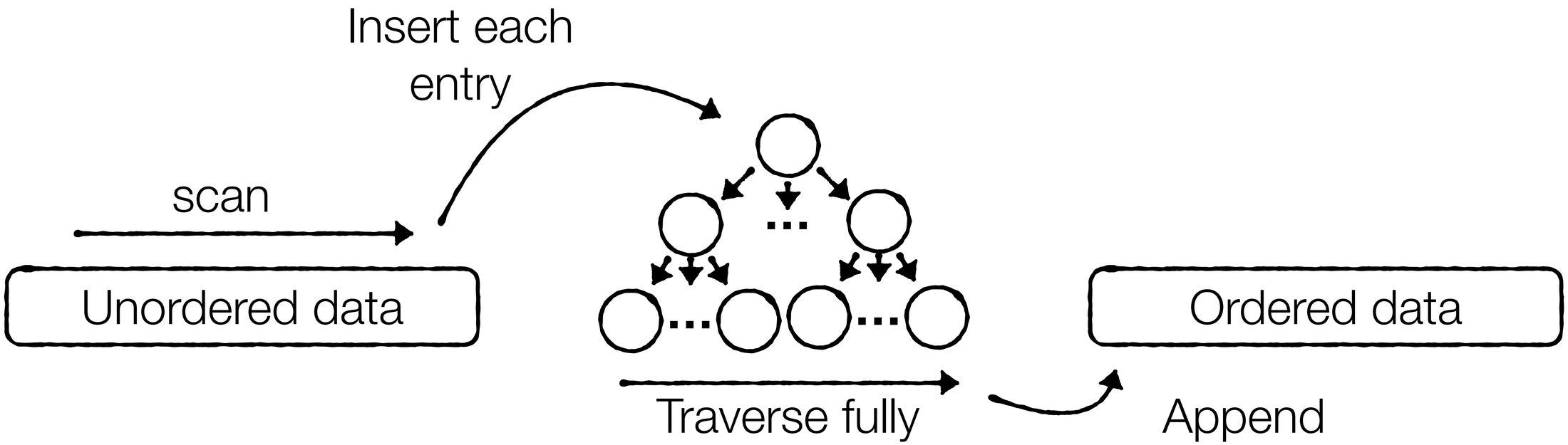
Insert data entries into a B-tree



Insert data entries into a B-tree

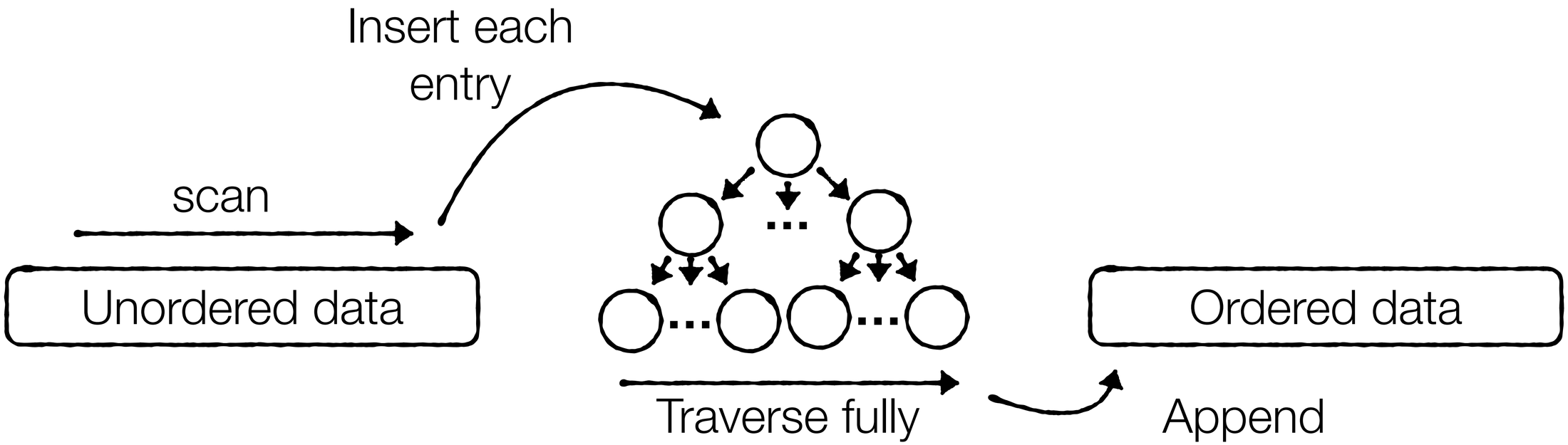


Insert data entries into a B-tree



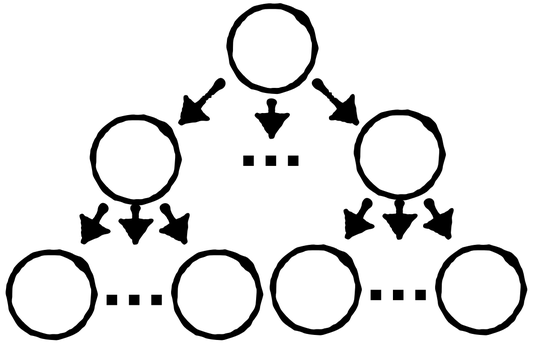
If internal nodes are in **storage**: **O(N $\log_B$ N) reads & O(N) writes**

Insert data entries into a B-tree

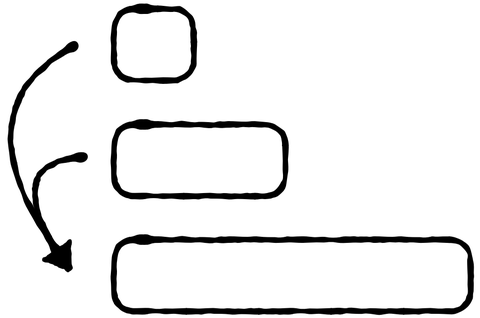


If internal nodes are in **memory**: **O(N) reads & writes**

Baselines

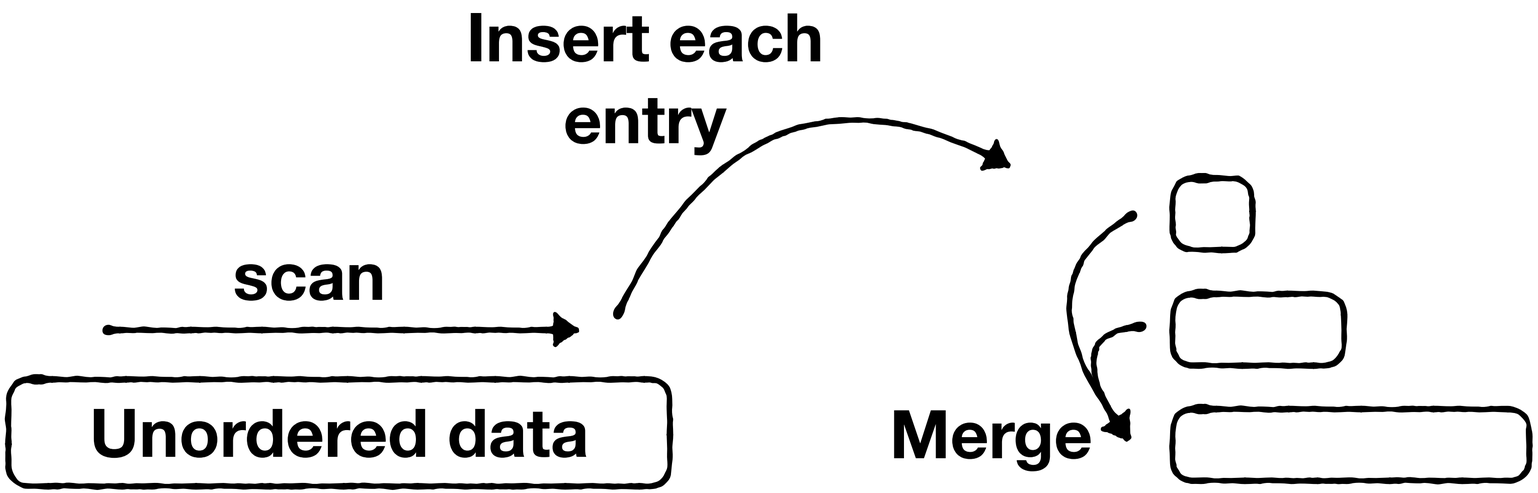


B-Tree

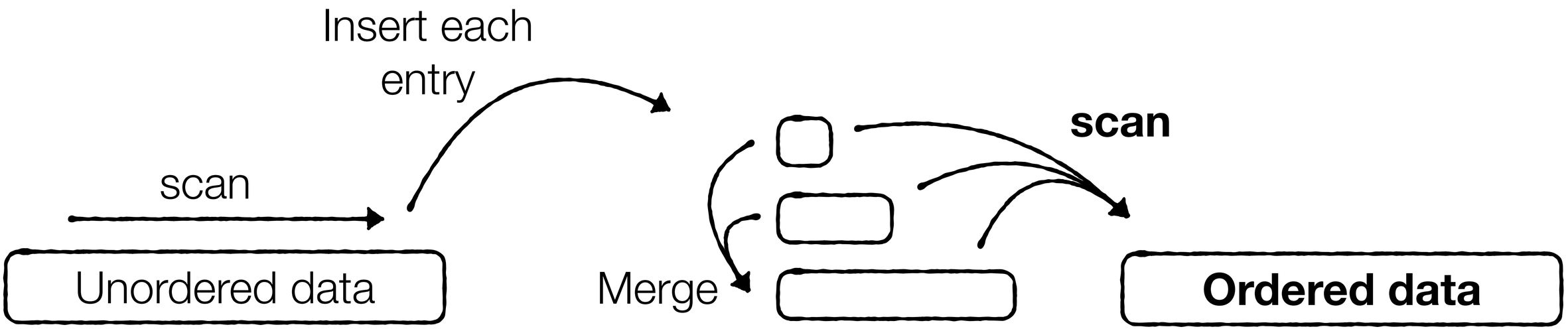


Log-Structured
Merge-Tree

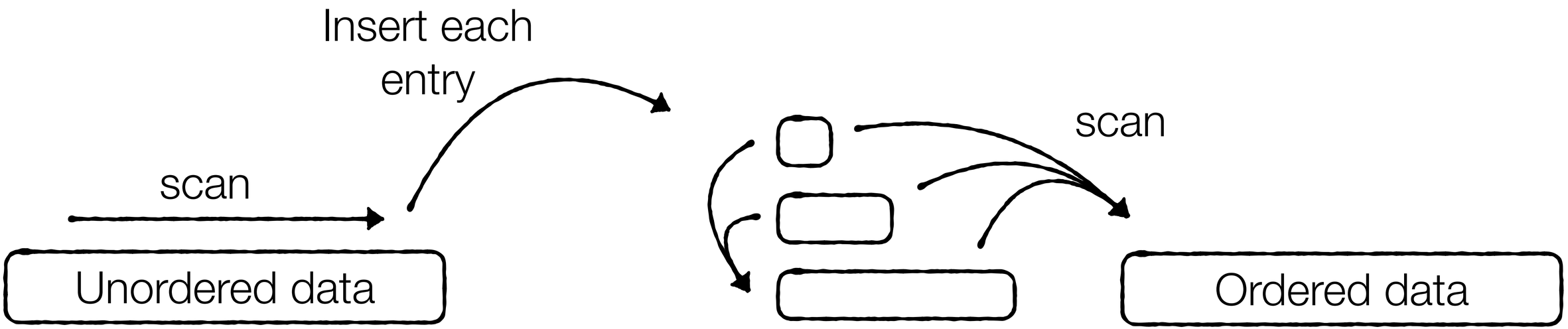
Insert entries into LSM-tree



Insert entries into LSM-tree



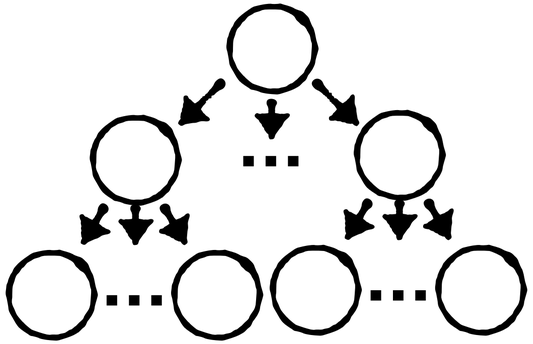
Insert entries into LSM-tree



With basic LSM-tree: O(N/B * $\log_2$(N/P)) reads & writes

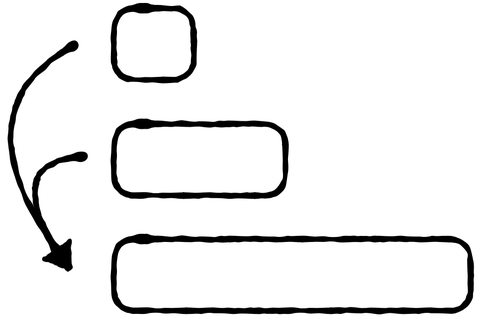
(Regardless of whether internal nodes fit in memory)

B-Tree



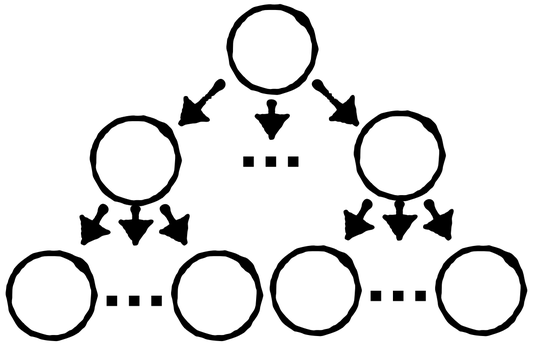
O(N)

LSM-Tree



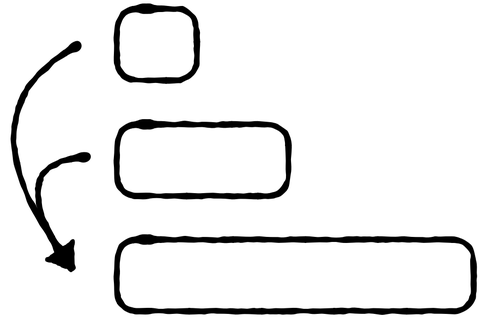
O(N/B * $\log_2$(N/P))

B-Tree



O(N)

LSM-Tree



O(N/B * $\log_2$(N/P))

Can we do better?

Ideally we want O(N/B)

How can we efficiently sort data that doesn’t fit in memory?

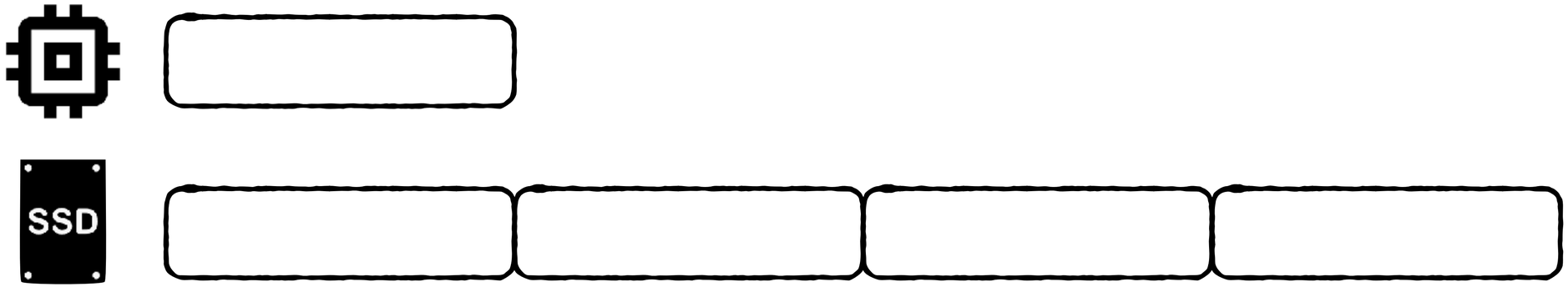
Think & then discuss with your neighbors

M/B pages

Unordered data (N/B pages)

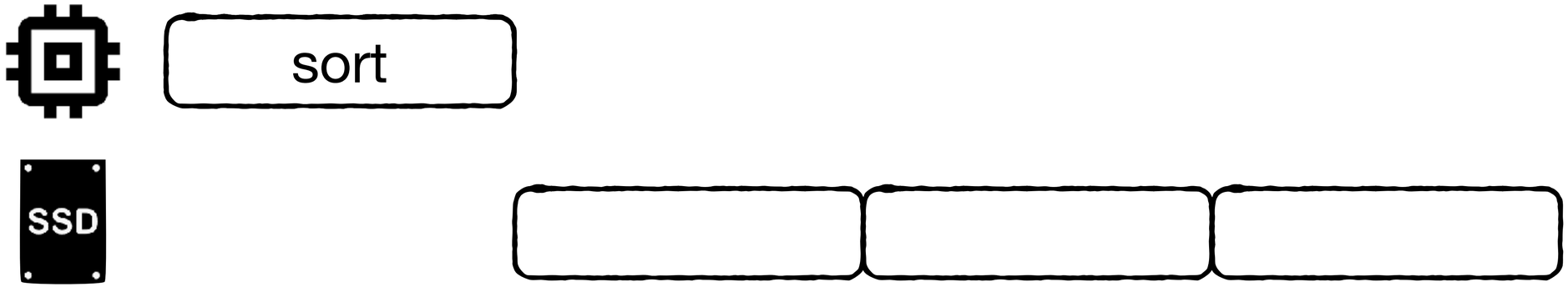
Multi-Way Merge-Sort

1. Sequentially read chunks that fit in memory, sort, and store back as temporary files



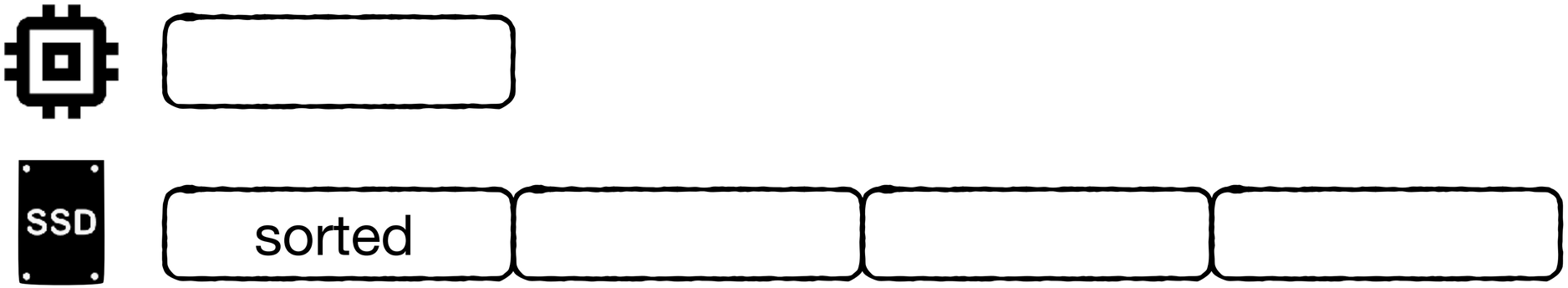
Multi-Way Merge-Sort

1. Sequentially read chunks that fit in memory, sort, and store back as temporary files

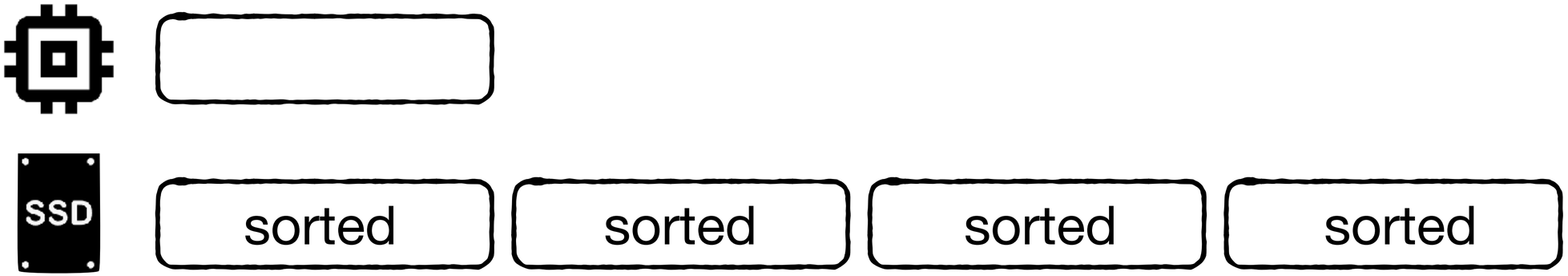


Multi-Way Merge-Sort

1. Sequentially read chunks that fit in memory, sort, and store back as temporary files

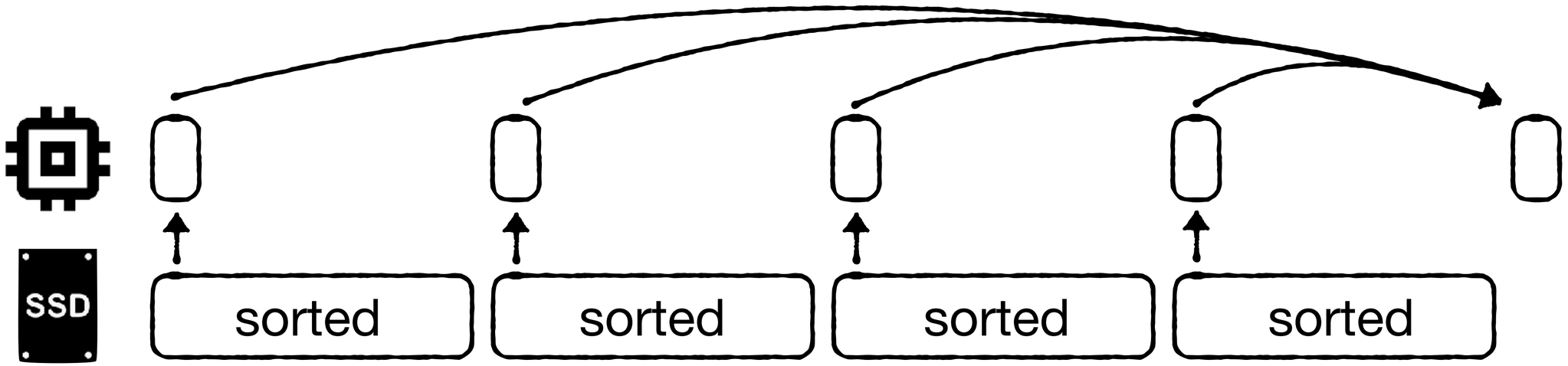


1. Sequentially read chunks that fit in memory, sort, and store back as temporary files



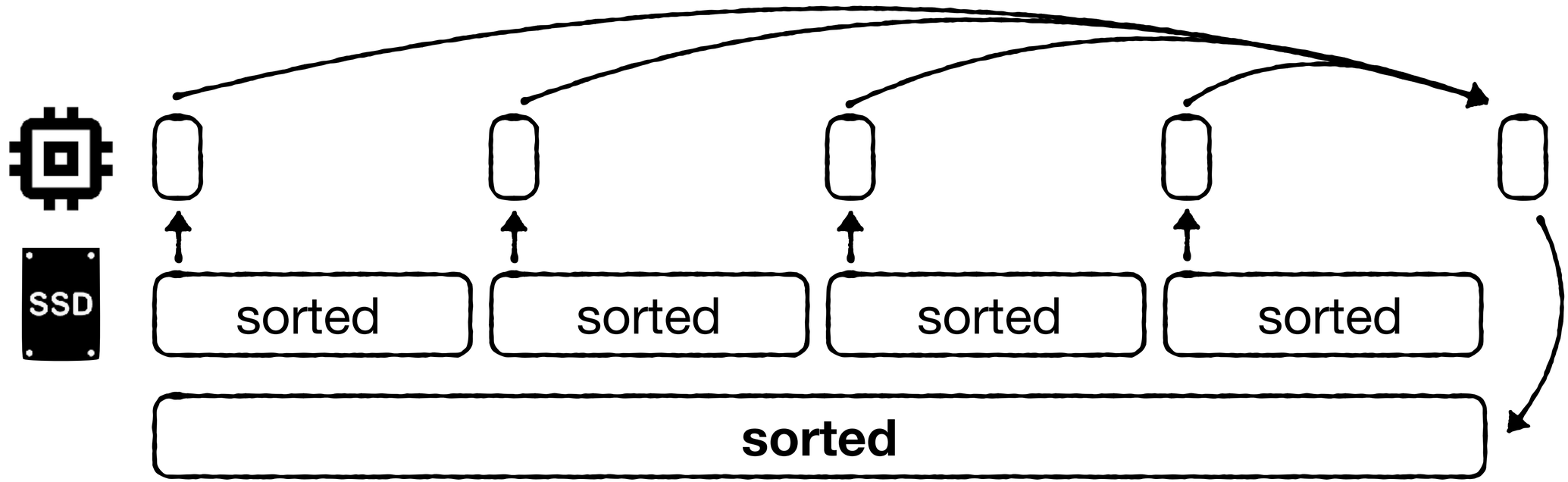
1. Sequentially read chunks that fit in memory, sort, and store back as temporary files

2. Allocate a buffer for each input file and merge into output stream.



1. Sequentially read chunks that fit in memory, sort, and store back as temporary files

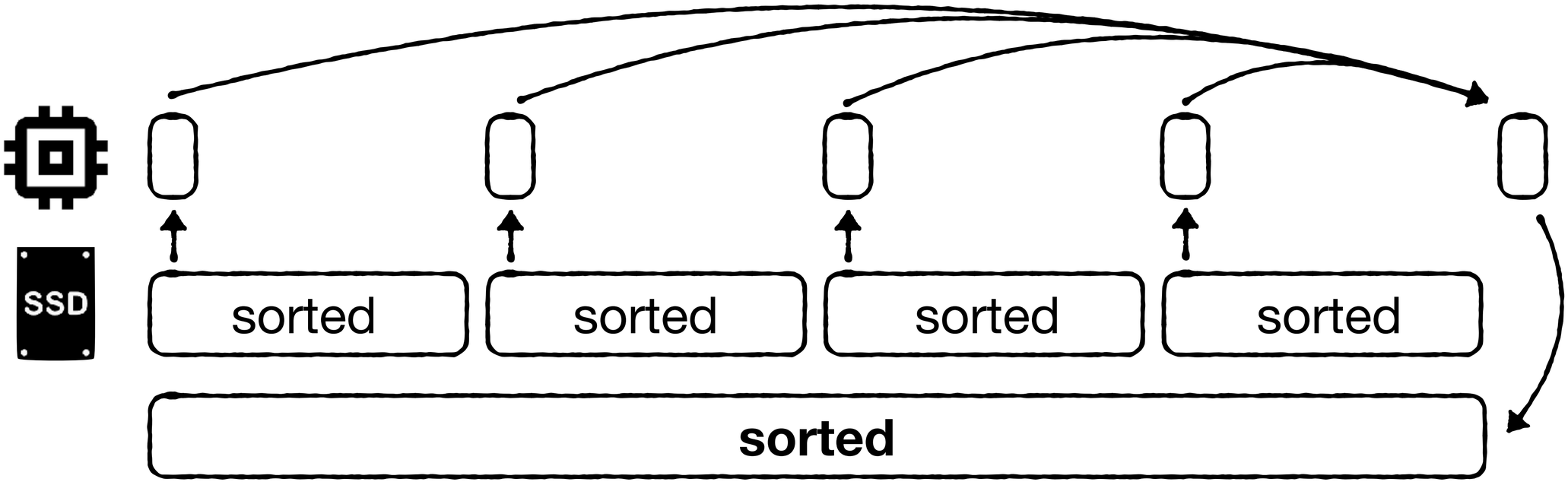
2. Allocate a buffer for each input file and merge into output stream.



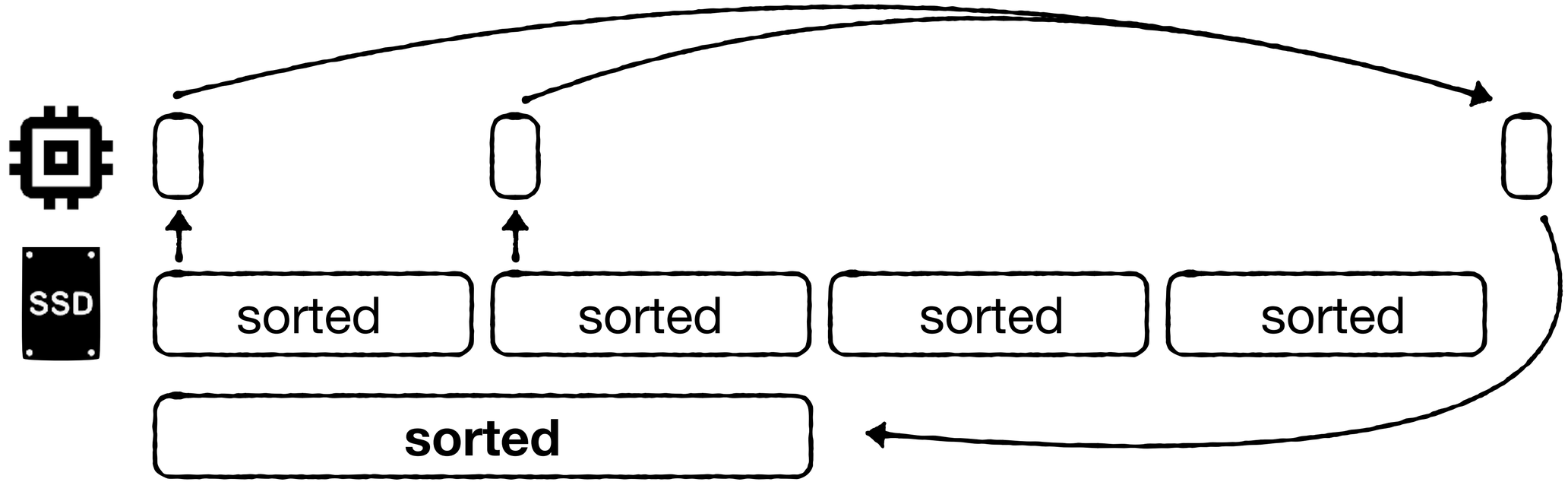
Problem?

1. Sequentially read chunks that fit in memory, sort, and store back as temporary files

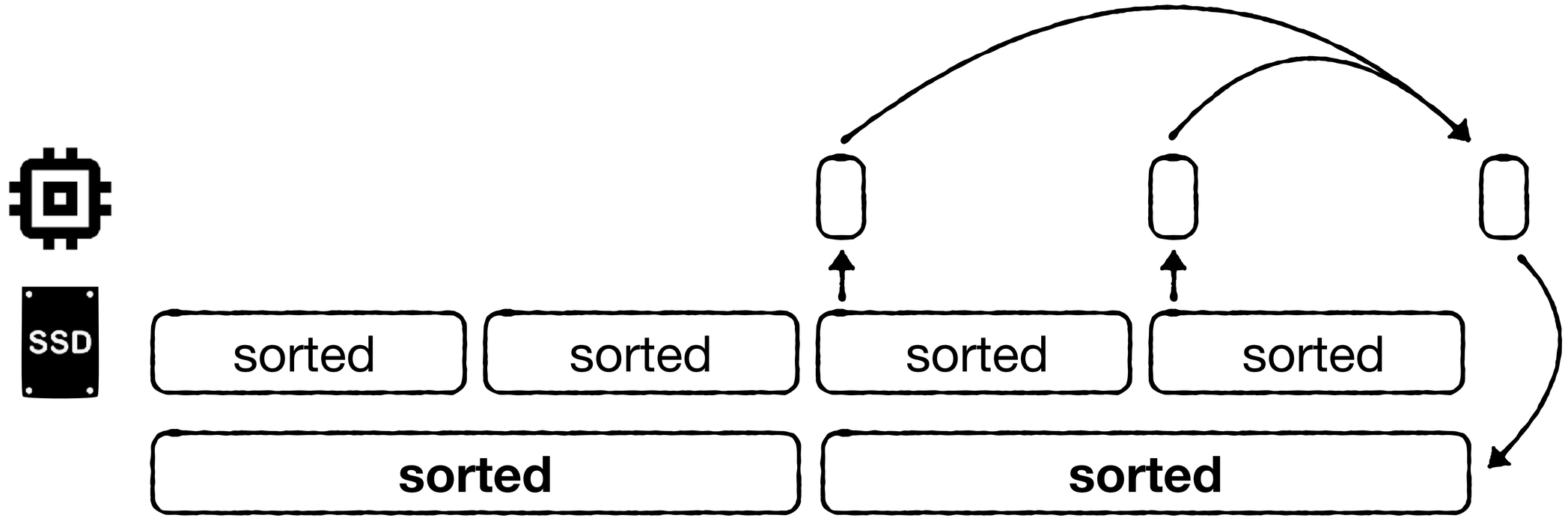
2. Allocate a buffer for each input file and merge into output stream.



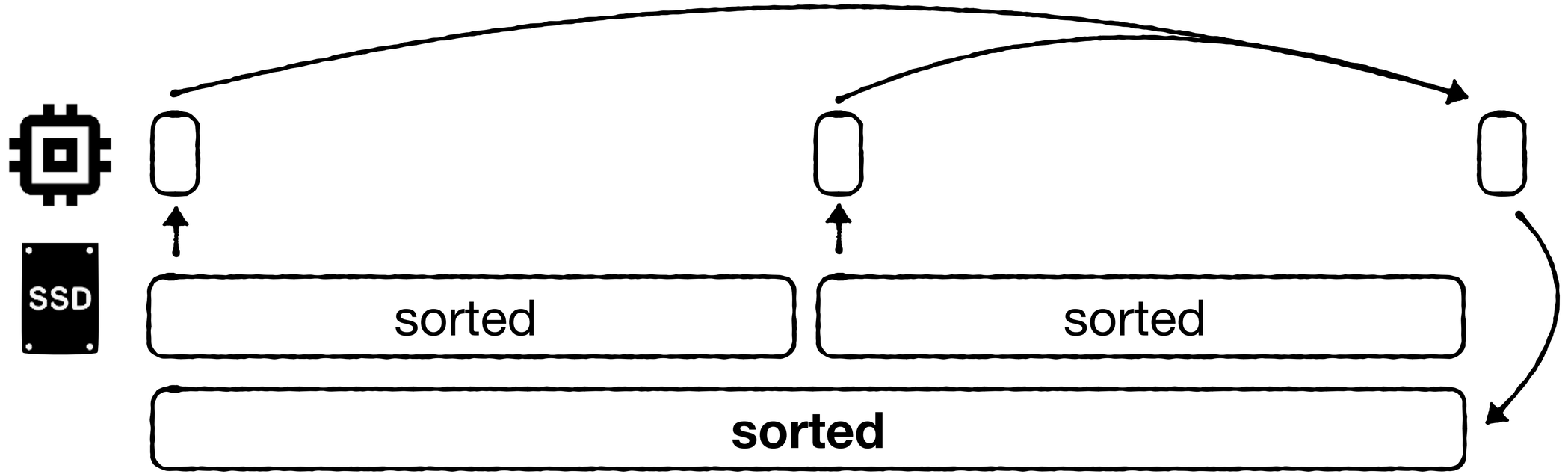
3. If we have little memory, may have to do multiple passes



3. If we have little memory, may have to do multiple passes



3. If we have little memory, may have to do multiple passes



Analysis

How many merging iterations (passes) must we do?

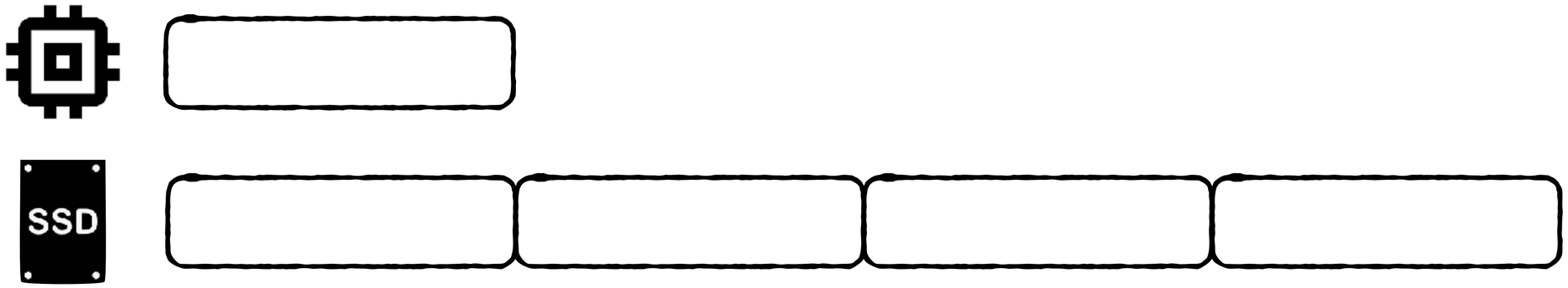
M/B pages

N/B pages

Analysis

How many merging iterations (passes) must we do?

Data is divided into N/M partitions



N/M partitions

Analysis

How many merging iterations (passes) must we do?

Data is divided into N/M partitions

We have M/B buffers, so each iteration merges M/B partitions

Analysis

How many merging iterations (passes) must we do?

Data is divided into N/M partitions

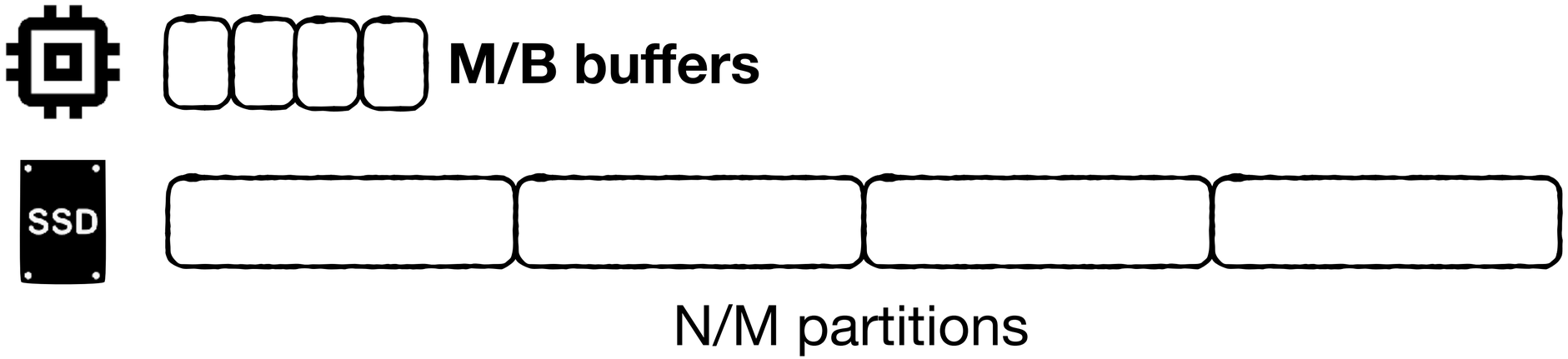
We have M/B buffers, so each iteration merges M/B partitions

Iterations = $\log_{M/B}(N/M)$

Analysis

Iterations = $\log_{M/B}(N/M)$

Each iteration does a full pass over the data costing O(N/B)

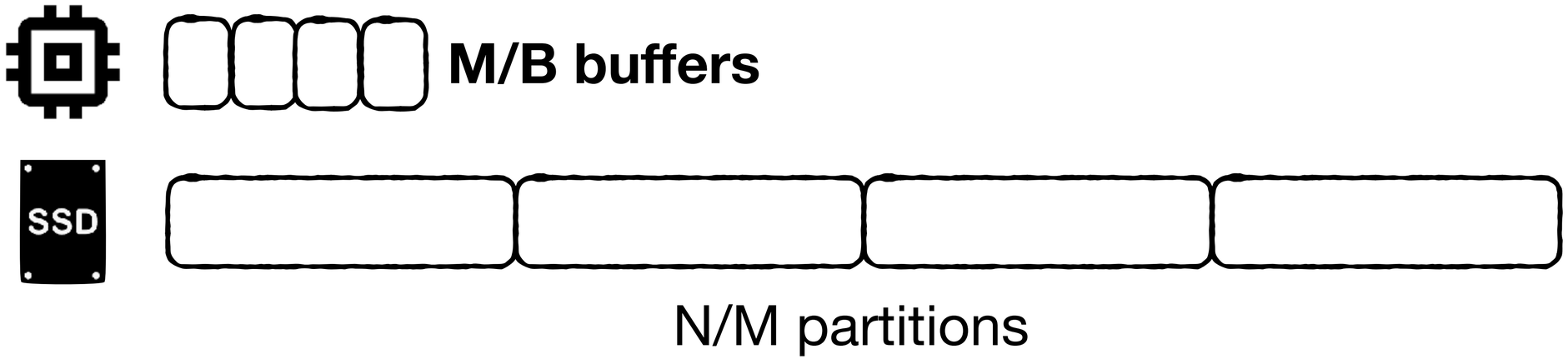


Analysis

Iterations = $\log_{M/B}(N/M)$

Each iteration does a full pass over the data costing O(N/B)

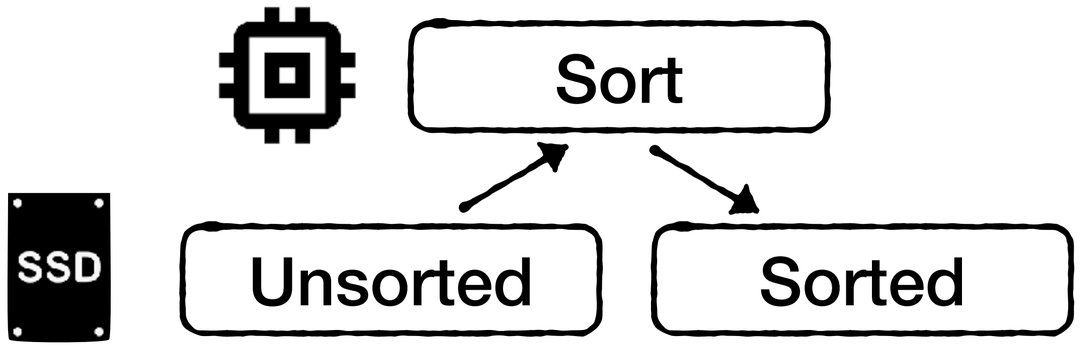
Cost of Merging Phase: $N/B \cdot \lceil \log_{M/B}(N/M) \rceil$



Analysis

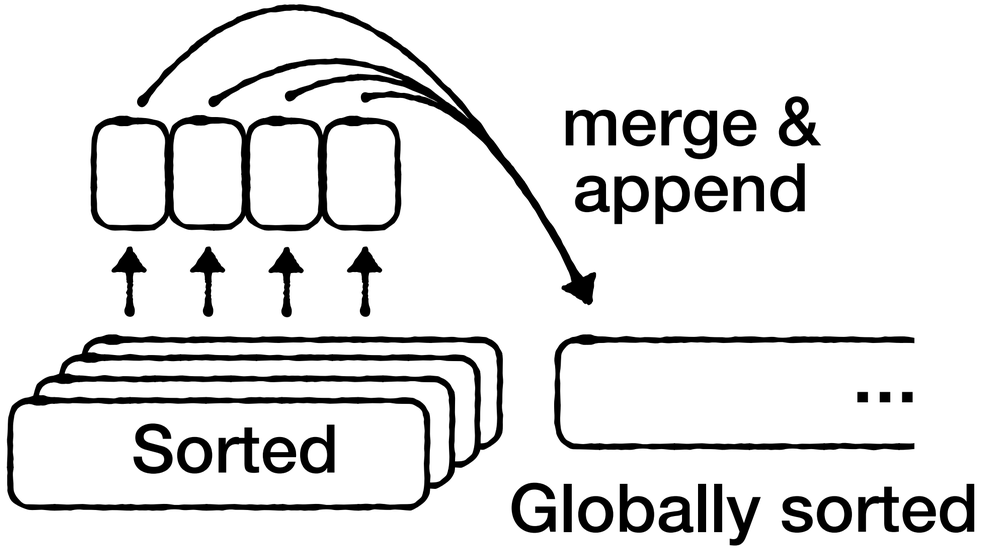
Partitioning Phase

N/B I/Os



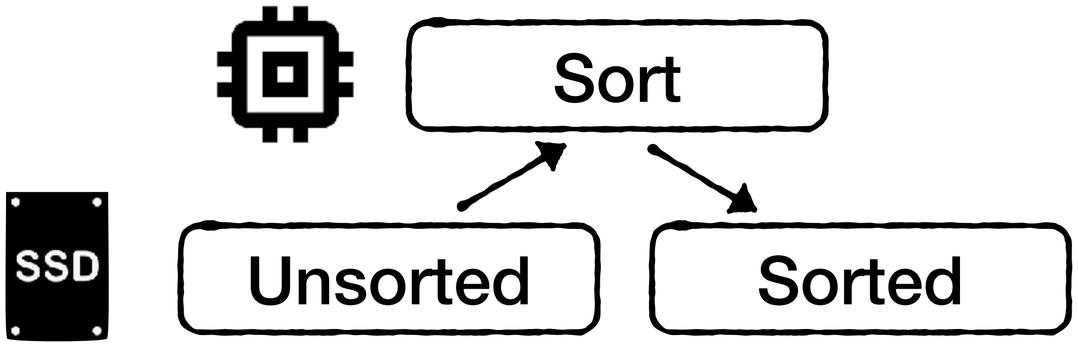
Merging Phase

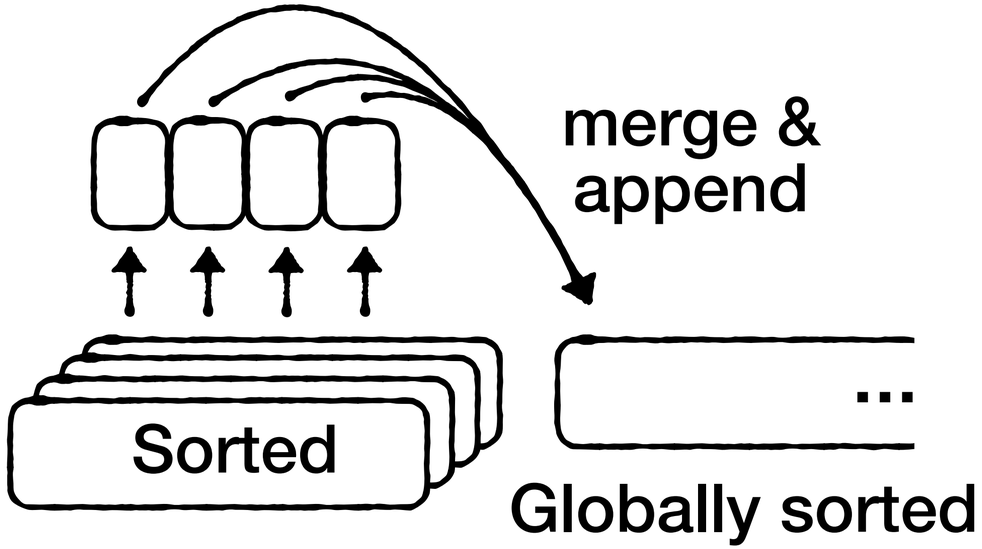
$N/B \cdot \lceil \log_{M/B}(N/M) \rceil$



Analyzing I/O for External Sort

Partitioning Phase		Merging Phase		**Total**
N/B I/Os	+	$N/B \cdot \lceil \log_{M/B}(N/M) \rceil$	=	**$O(N/B \cdot \log_{M/B}(N/B))$**





Impact of More Memory

$$O(N/B \cdot \log_{M/B}(N/B))$$

M/B pages

N/B pages

Impact of More Memory

$$O(N/B \cdot \log_{M/B}(N/B))$$

Merging more partitions per pass

M/B pages

N/B pages

How much memory to merge all partitions in one pass?

$$O(N/B \cdot \log_{M/B}(N/B))$$

M/B pages

N/B pages

How much memory to merge all partitions in one pass?

Let $\log_{M/B}(N/B) = 2$ and solve for M

M/B pages

N/B pages

How much memory to merge all partitions in one pass?

Let $\log_{M/B}(N/B) = 2$ and solve for M

We get: $M = \sqrt{N \cdot B}$ (Measured in entries)

M/B pages

N/B pages

How much memory to merge all partitions in one pass?

Let $\log_{M/B}(N/B) = 2$ and solve for M

We get: $M = \sqrt{N \cdot B}$ (Measured in entries)

Hence, memory can accommodate $\sqrt{N/B}$ buffers

M/B pages

N/B pages

Two-Pass Merge Sort Algorithm

Use at least $M = \sqrt{N \cdot B}$ memory to partition the data.

This creates at most $N/M = \sqrt{N/B}$ sorted partitons

M/B pages

N/B pages

Two-Pass Merge Sort Algorithm

Use at least $M = \sqrt{N \cdot B}$ memory to partition the data.

This creates at most $N/M = \sqrt{N/B}$ sorted partitons

Then merge in one pass using at most $\sqrt{N/B}$ input buffers

M/B pages

N/B pages

Two-Pass Merge Sort Algorithm

Use at least $M = \sqrt{N \cdot B}$ memory to partition the data.

This creates at most $N/M = \sqrt{N/B}$ sorted partitons

Then merge in one pass using at most $\sqrt{N/B}$ input buffers

Cost: O(N/B)

M/B pages

N/B pages

How much memory do we need in practice?

Assume 1 TB, 16 byte entries, and 4KB pages

N=2^{36} entries. And with 4KB pages, B=2^{8} entries.

We need $M = \sqrt{N \cdot B} = \sqrt{2^{44}} = 2^{22}$ entries in memory, or 64 MB

M/B pages

N/B pages

How much memory do we need in practice?

Assume 1 TB, 16 byte entries, and 4KB pages

N=2^{36} entries. And with 4KB pages, B=2^{8} entries.

We need $M = \sqrt{N \cdot B} = \sqrt{2^{44}} = 2^{22}$ entries in memory, or 64 MB

Hence, for all practical purposes, a 2 pass sorting algorithm is practical.

M/B pages

N/B pages

Achieved our goal of sorting using O(N/B) I/Os using little memory :)

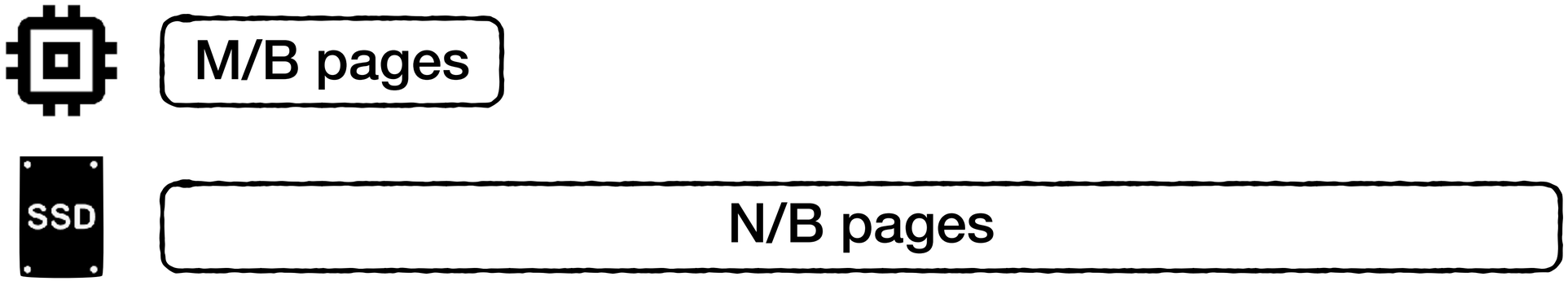
M/B pages

N/B pages

Achieved our goal of sorting using O(N/B) I/Os using little memory :)

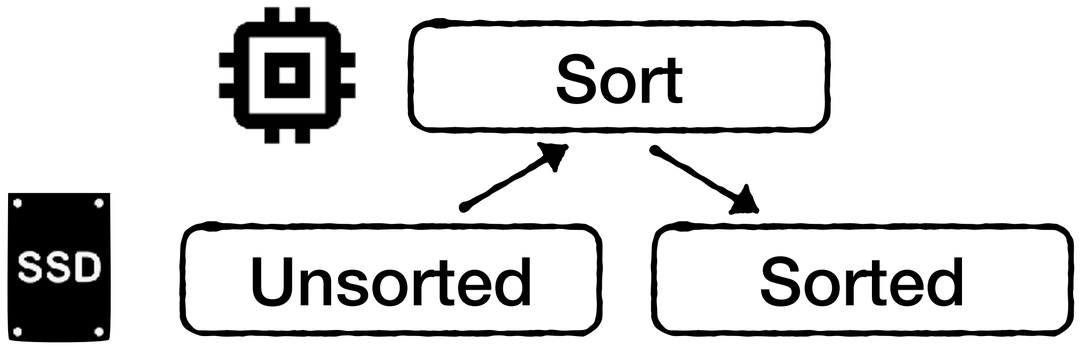
But how about CPU overheads?

We still expect $O(N \cdot \log_2 N)$. Do we achieve it?

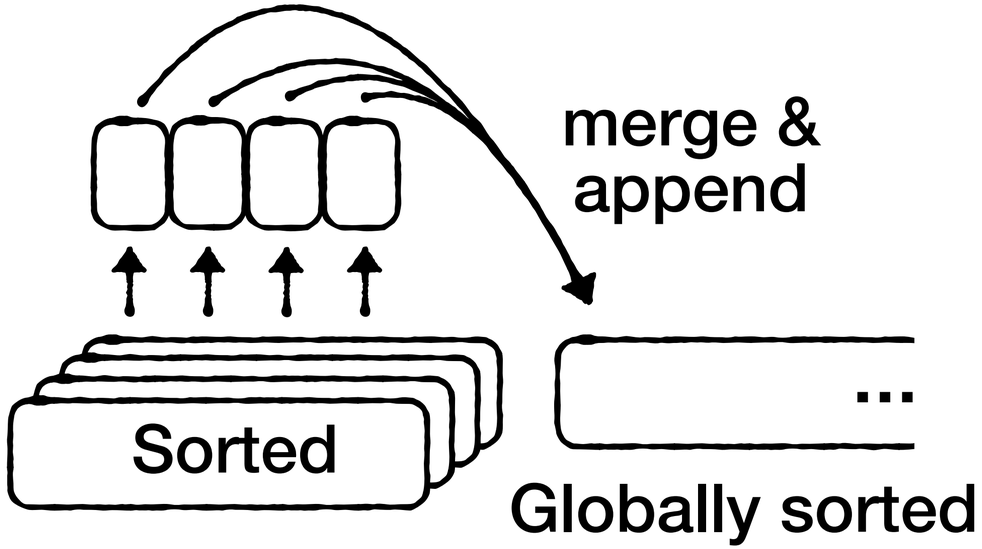


Analyzing CPU

Partitioning Phase



Merging Phase

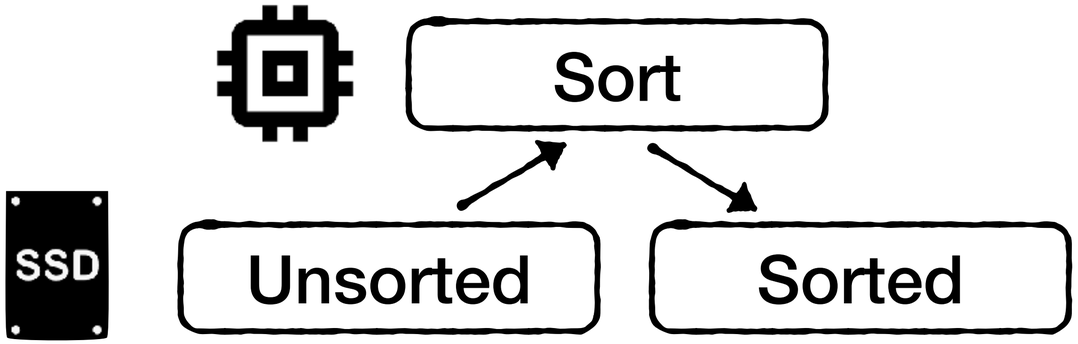


Partitioning Phase

Each chunk contains M entries

Need $O(M \cdot \log_2 M)$ CPU cycles to merge-sort it in-memory

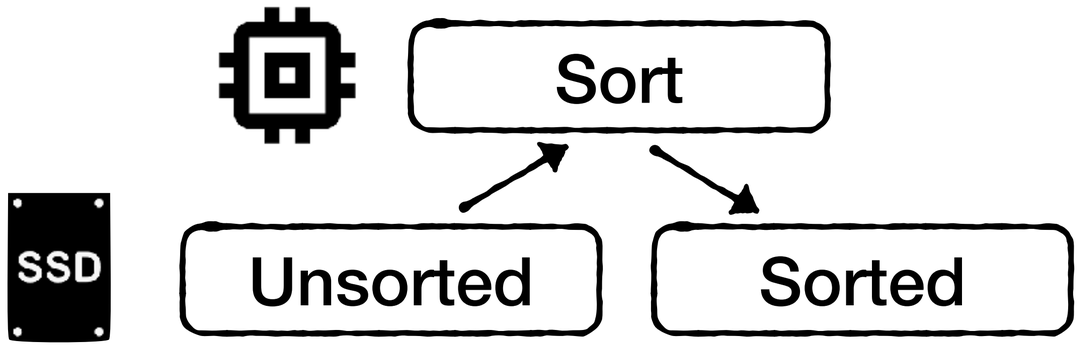
Doing this for all N/M chunk takes $O(N \cdot \log_2 M)$ CPU



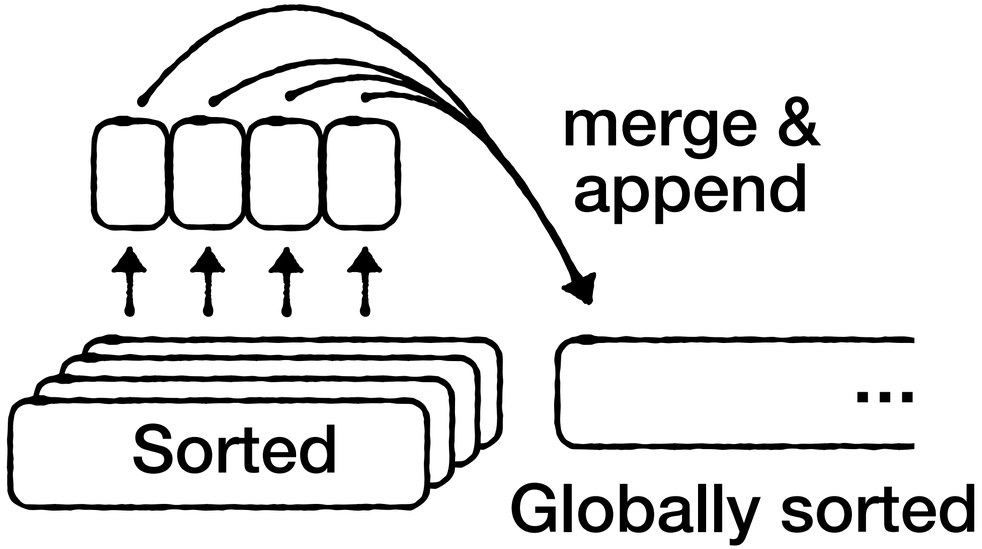
Analyzing CPU

Partitioning Phase

$O(N \cdot \log_2 M)$



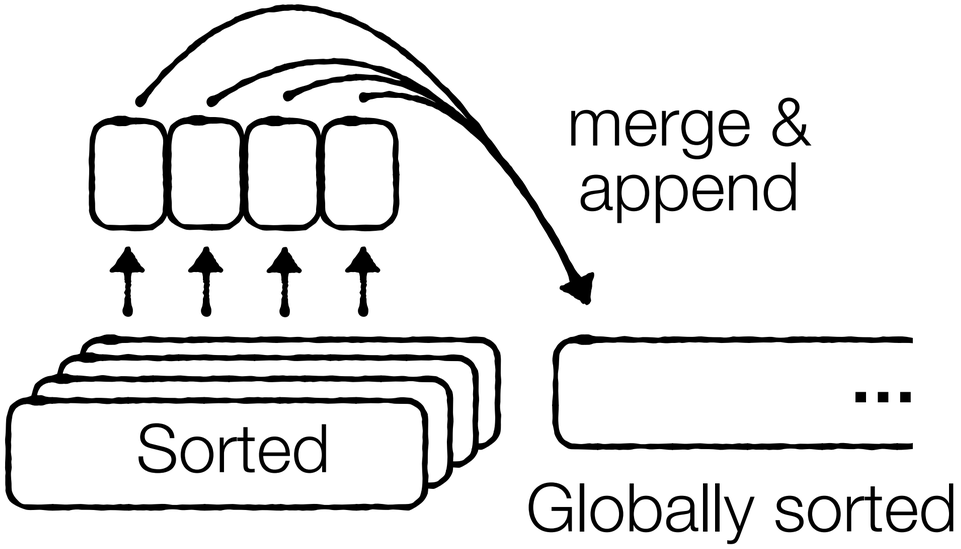
Merging Phase



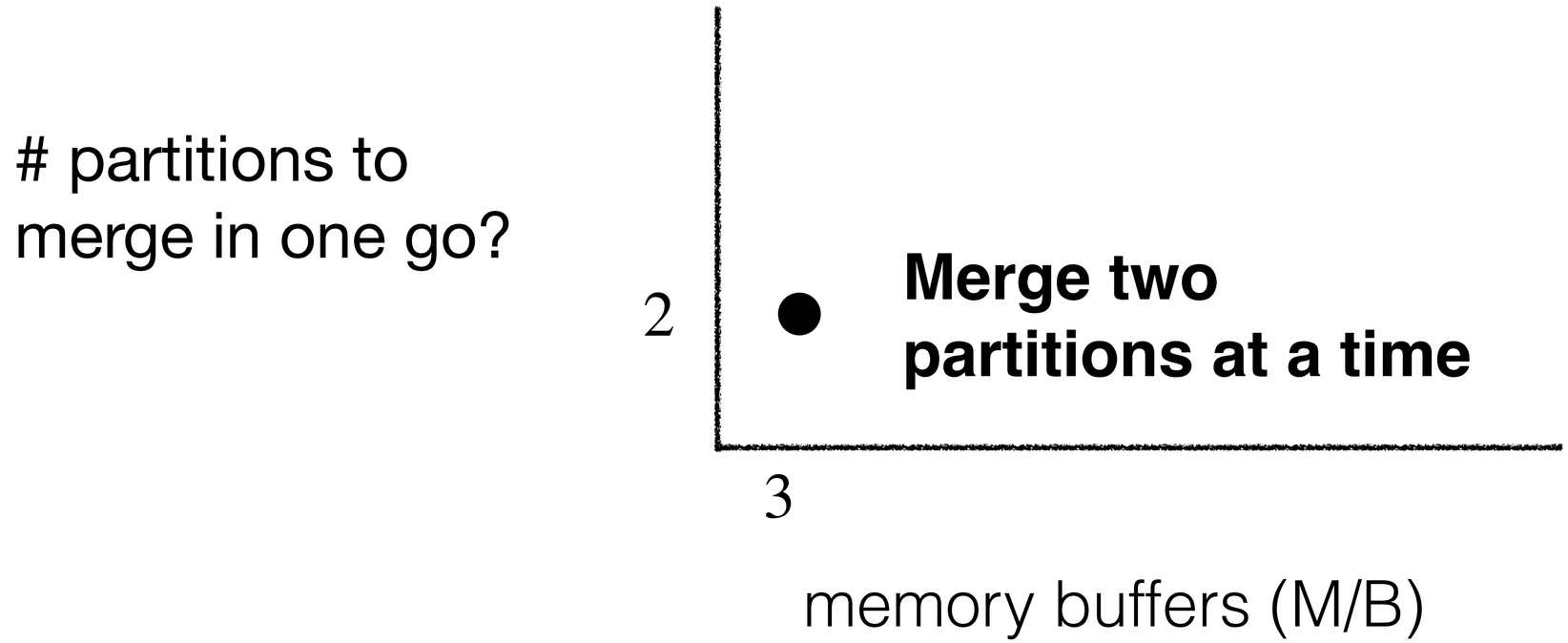
Merging Phase

maximum # partitions to merge in one go?

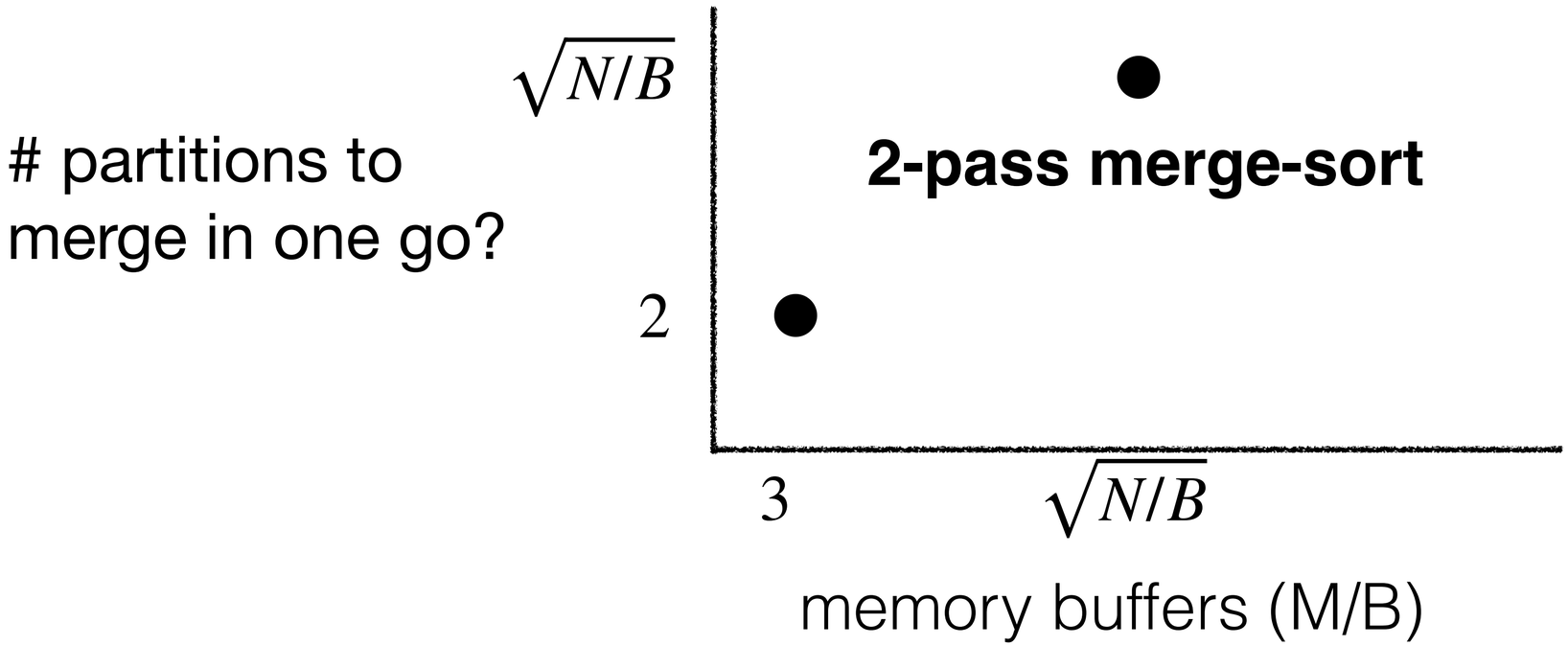
How do we merge them?



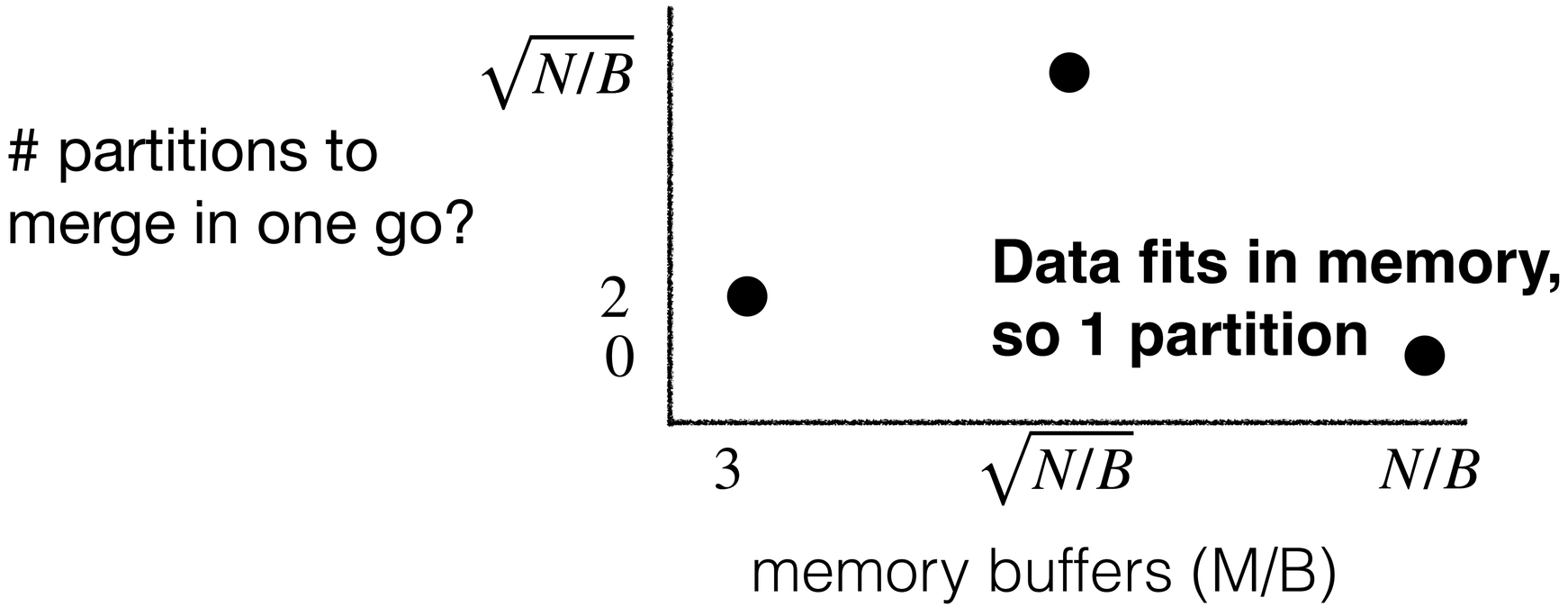
Merging Phase



Merging Phase

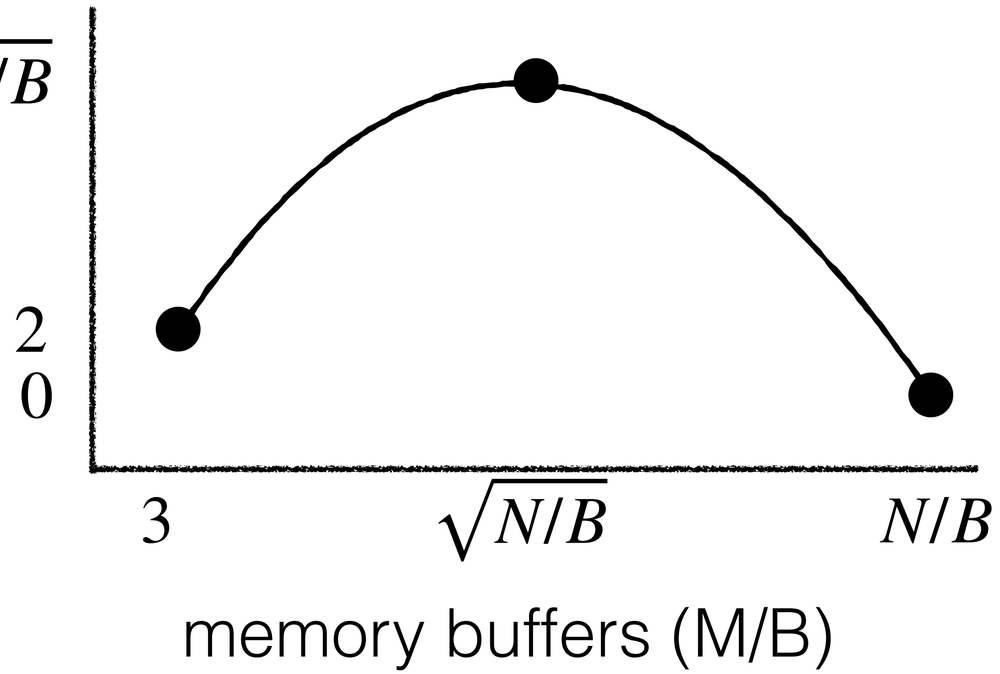


Merging Phase



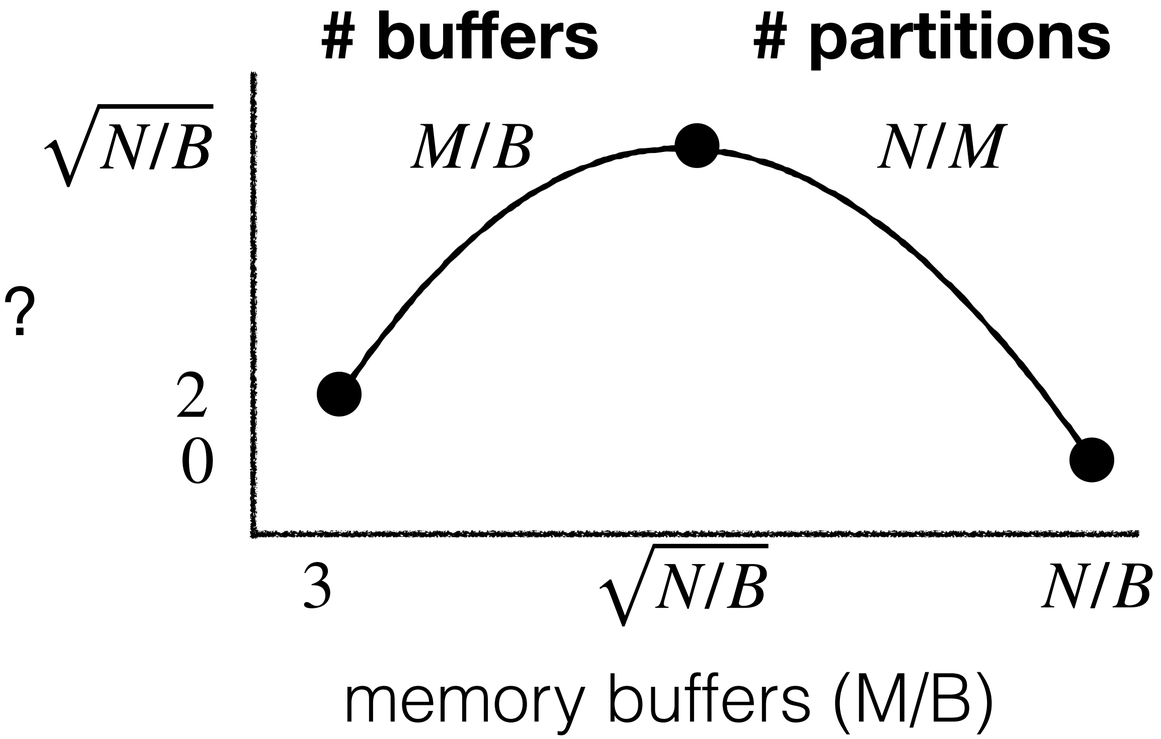
Merging Phase

partitions to merge in one go?



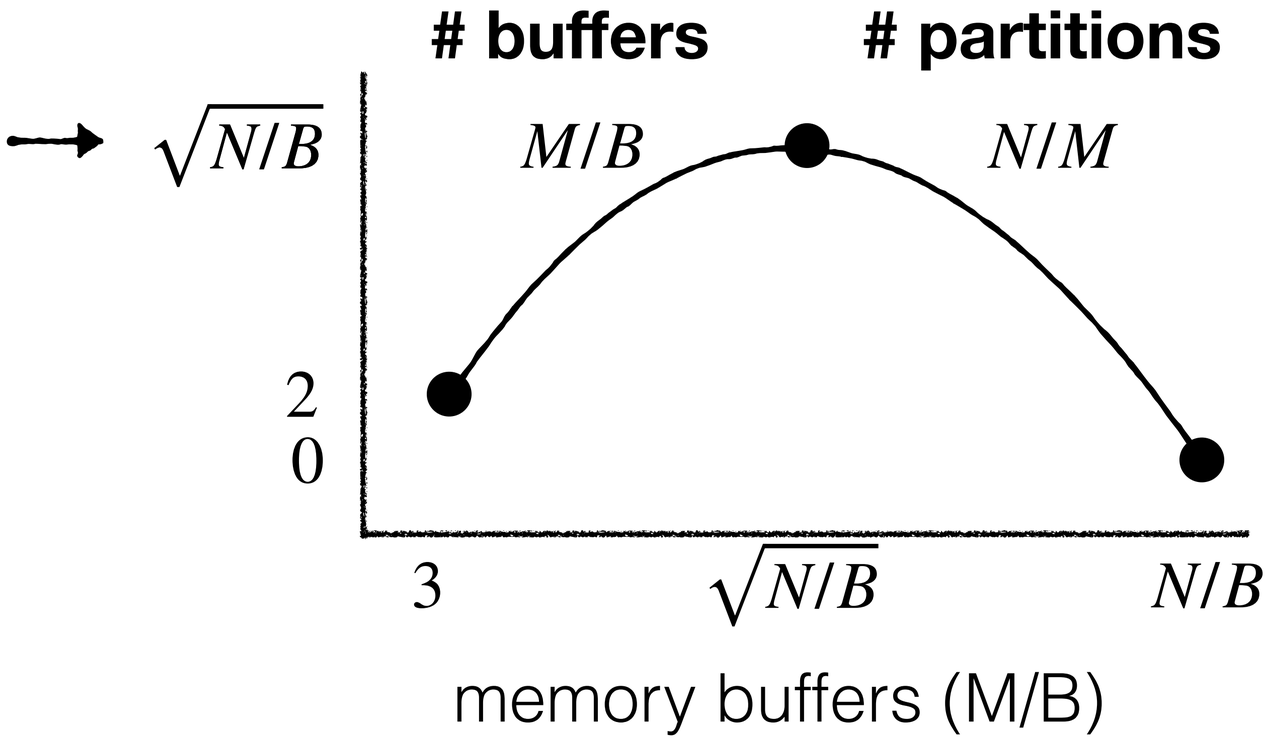
Merging Phase

partitions to merge in one go?



Merging Phase

Maximum # partitions to merge in one go

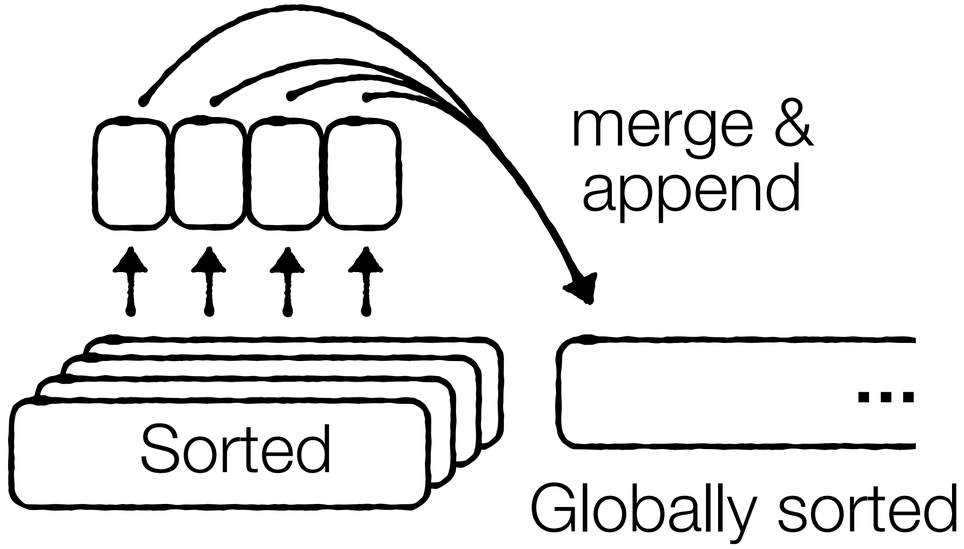


Merging Phase

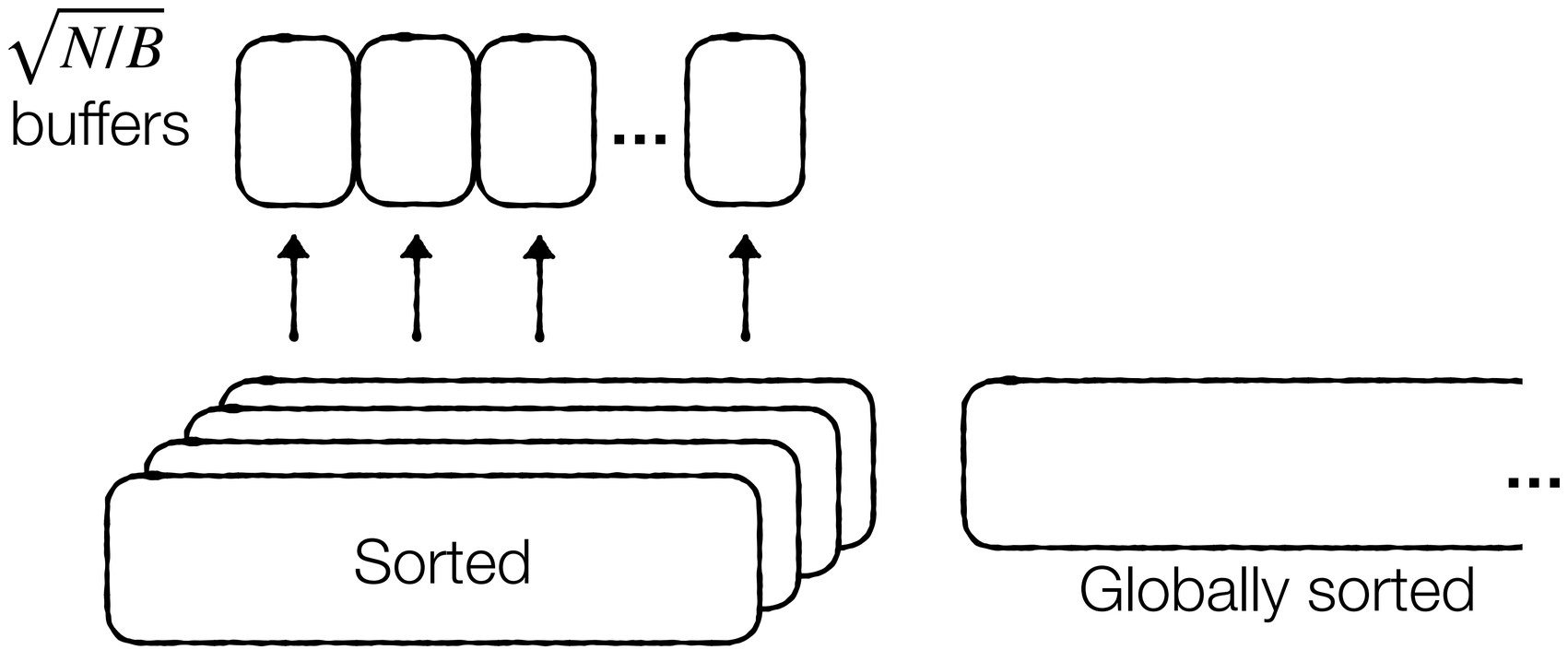
maximum # partitions
to merge in one go?

$$\sqrt{N/B}$$

How to merge partitions?

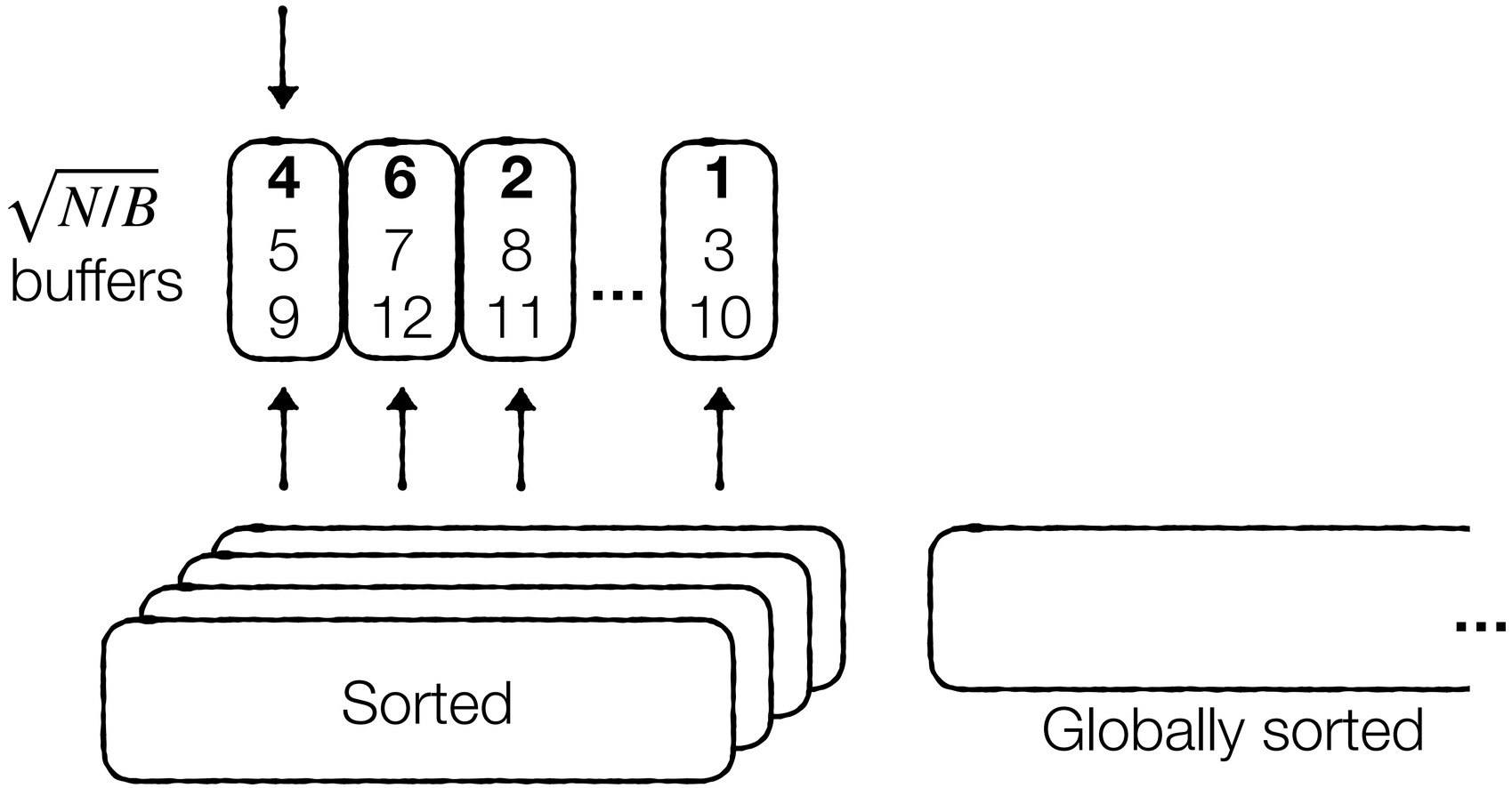


How to merge partitions?



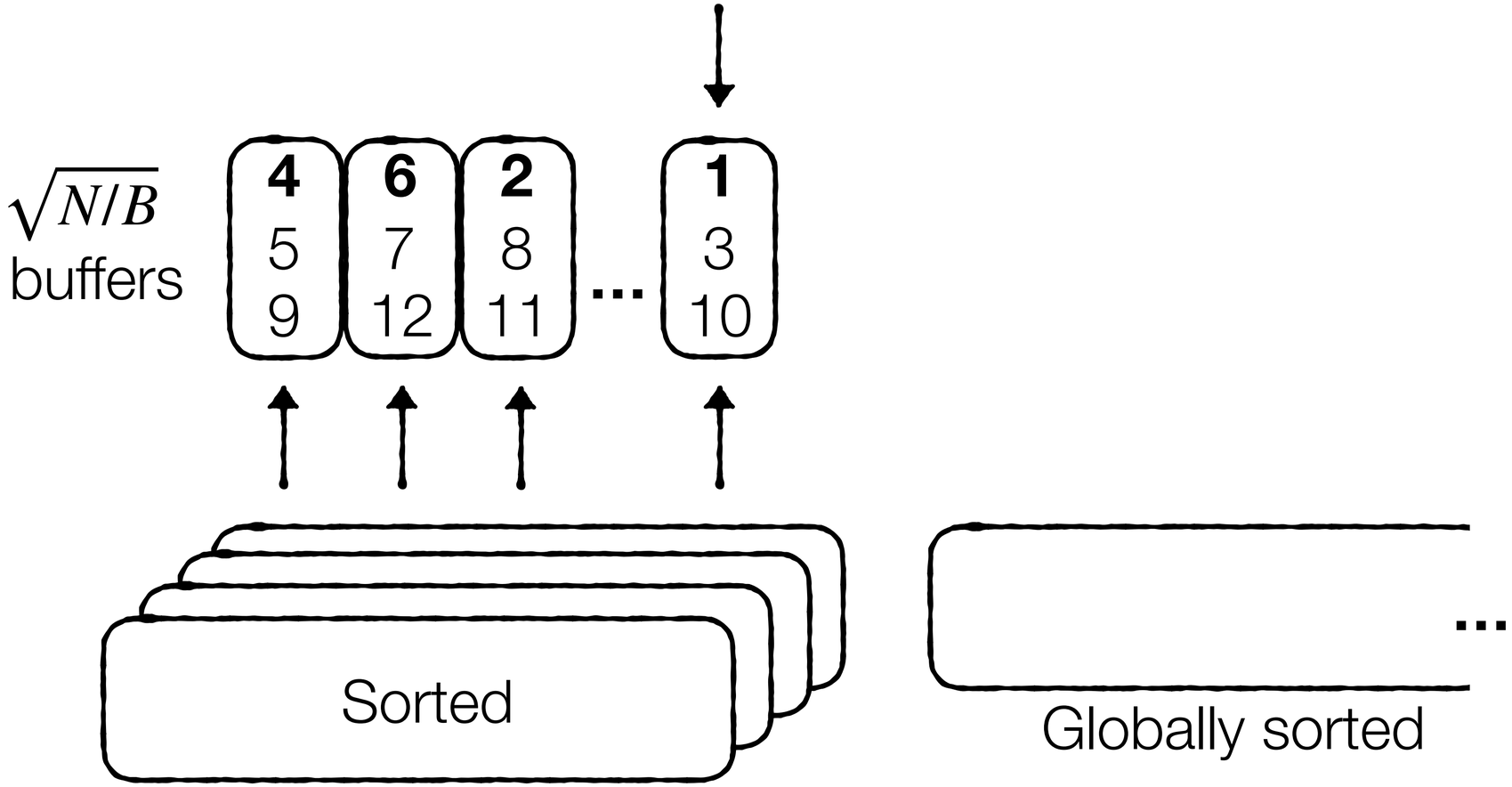
How to merge partitions?

Simple solution: Traverse all buffers and pick minimum



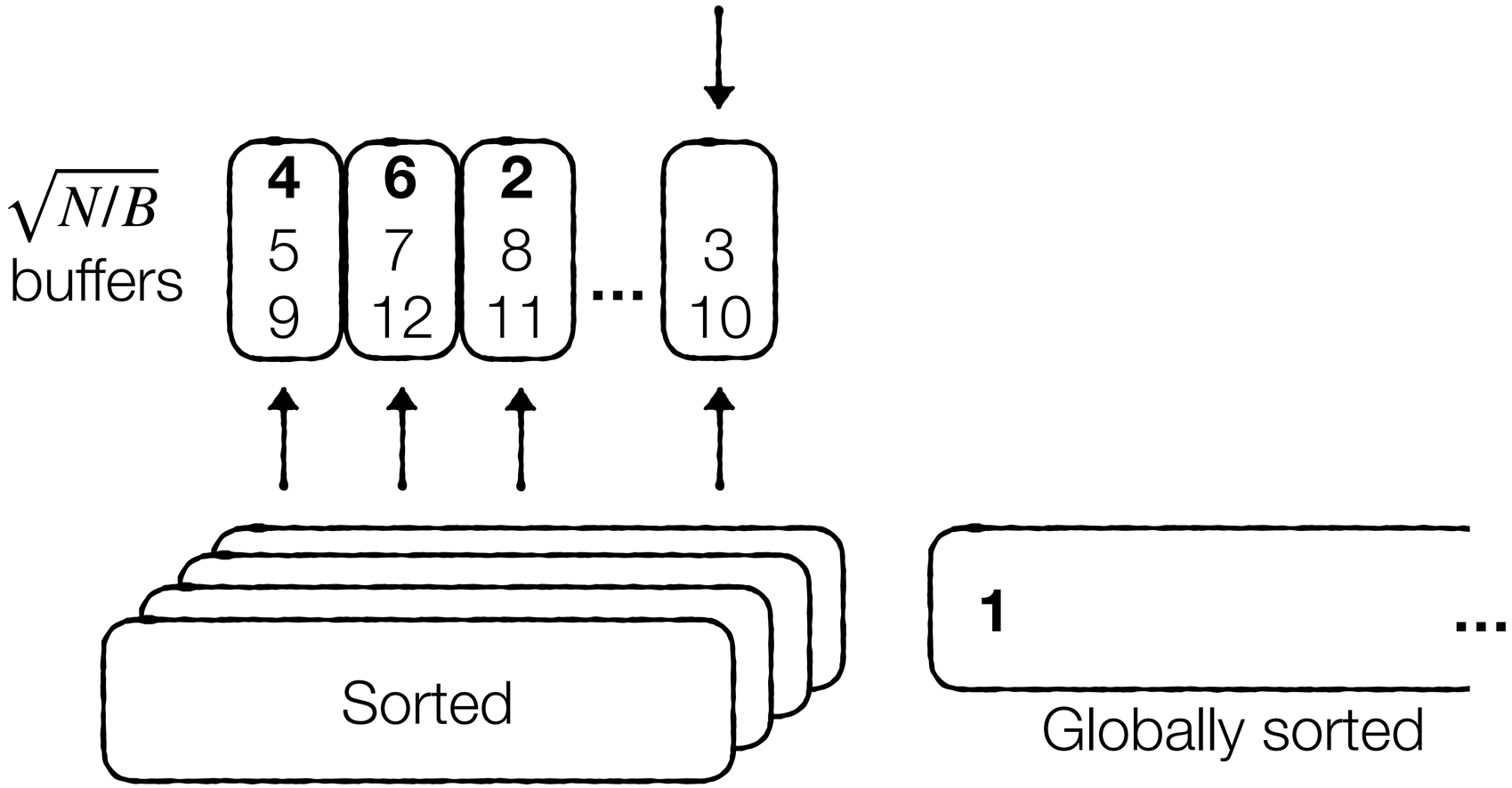
How to merge partitions?

Simple solution: Traverse all buffers and pick minimum



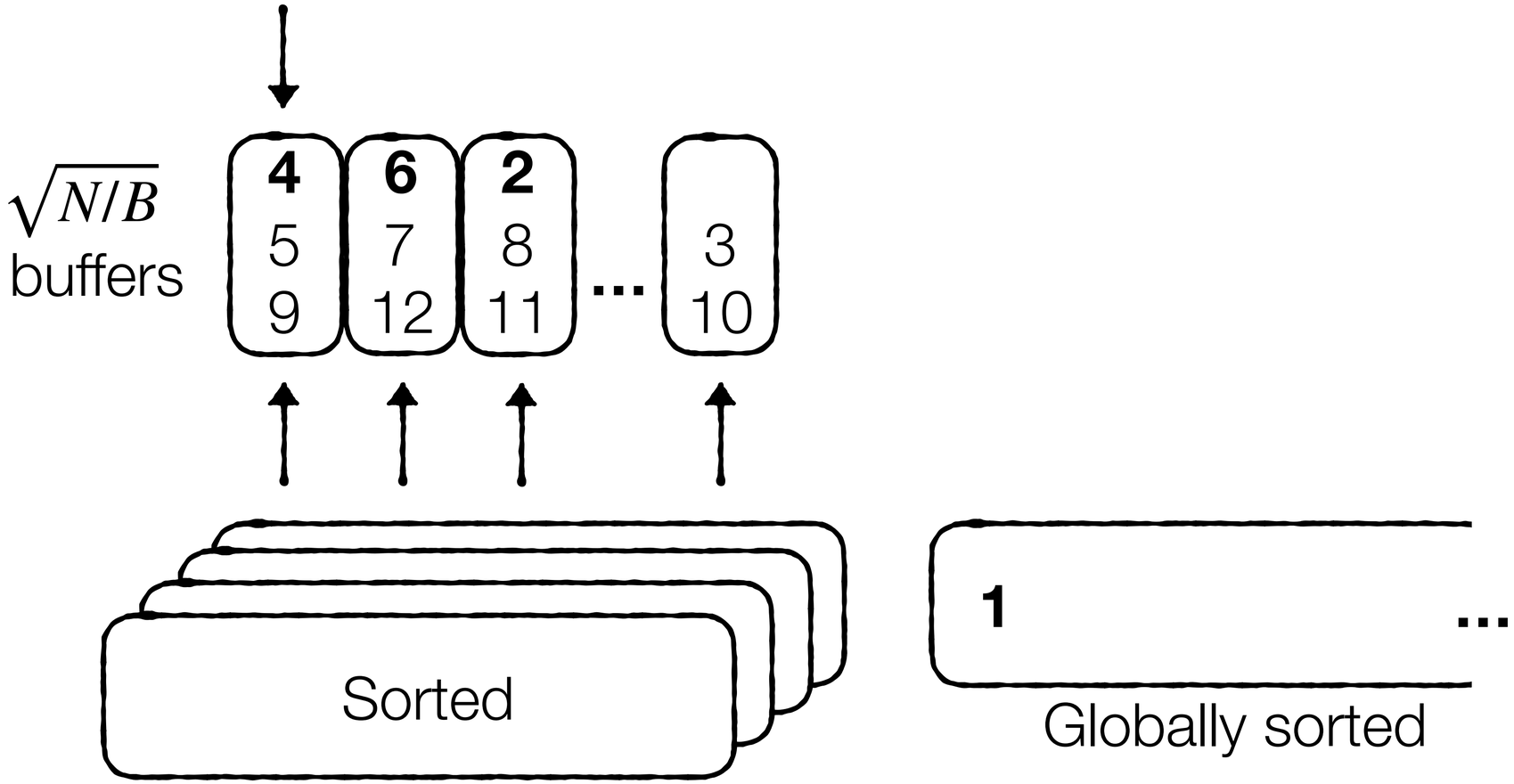
How to merge partitions?

Simple solution: Traverse all buffers and pick minimum



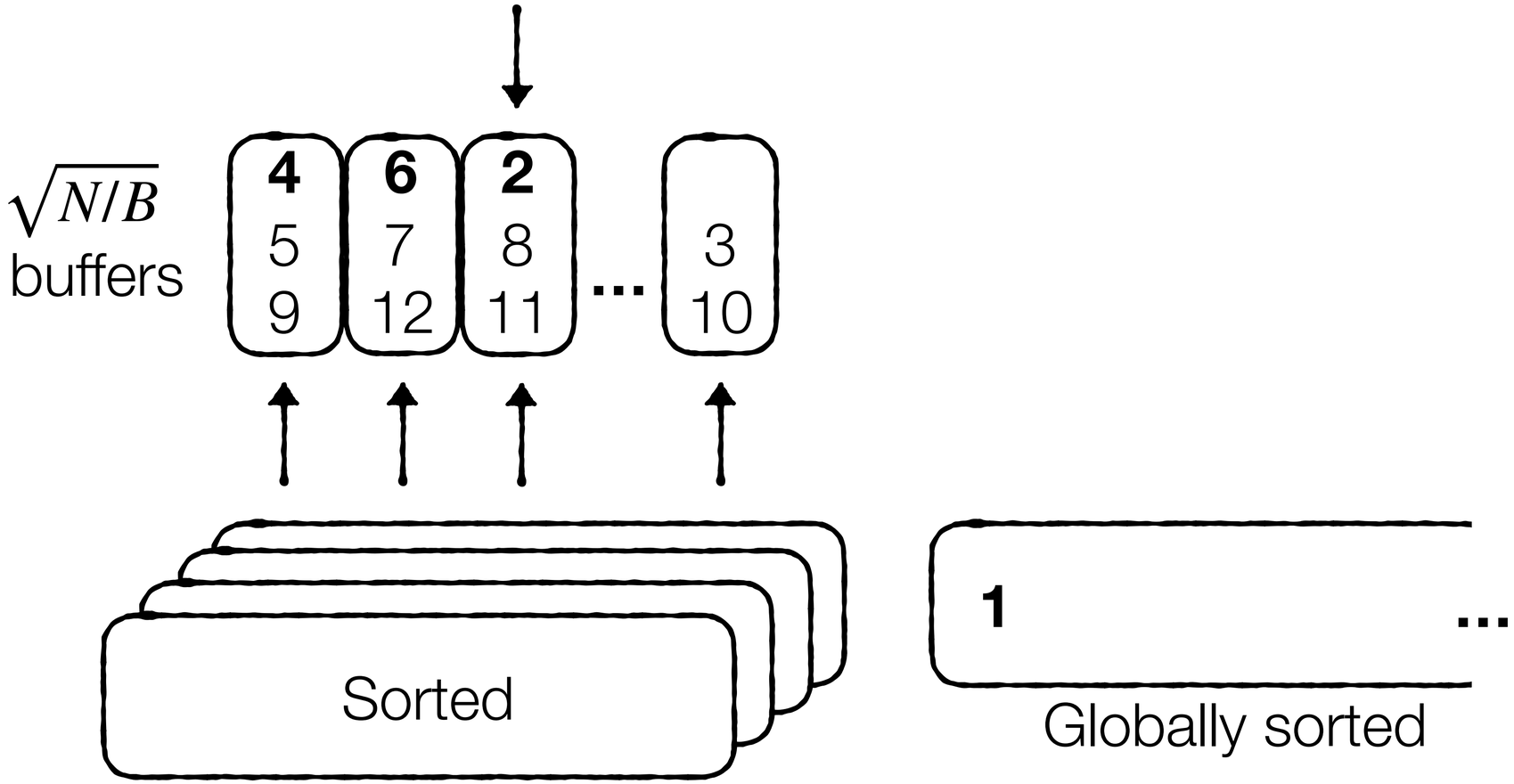
How to merge partitions?

Simple solution: Traverse all buffers and pick minimum



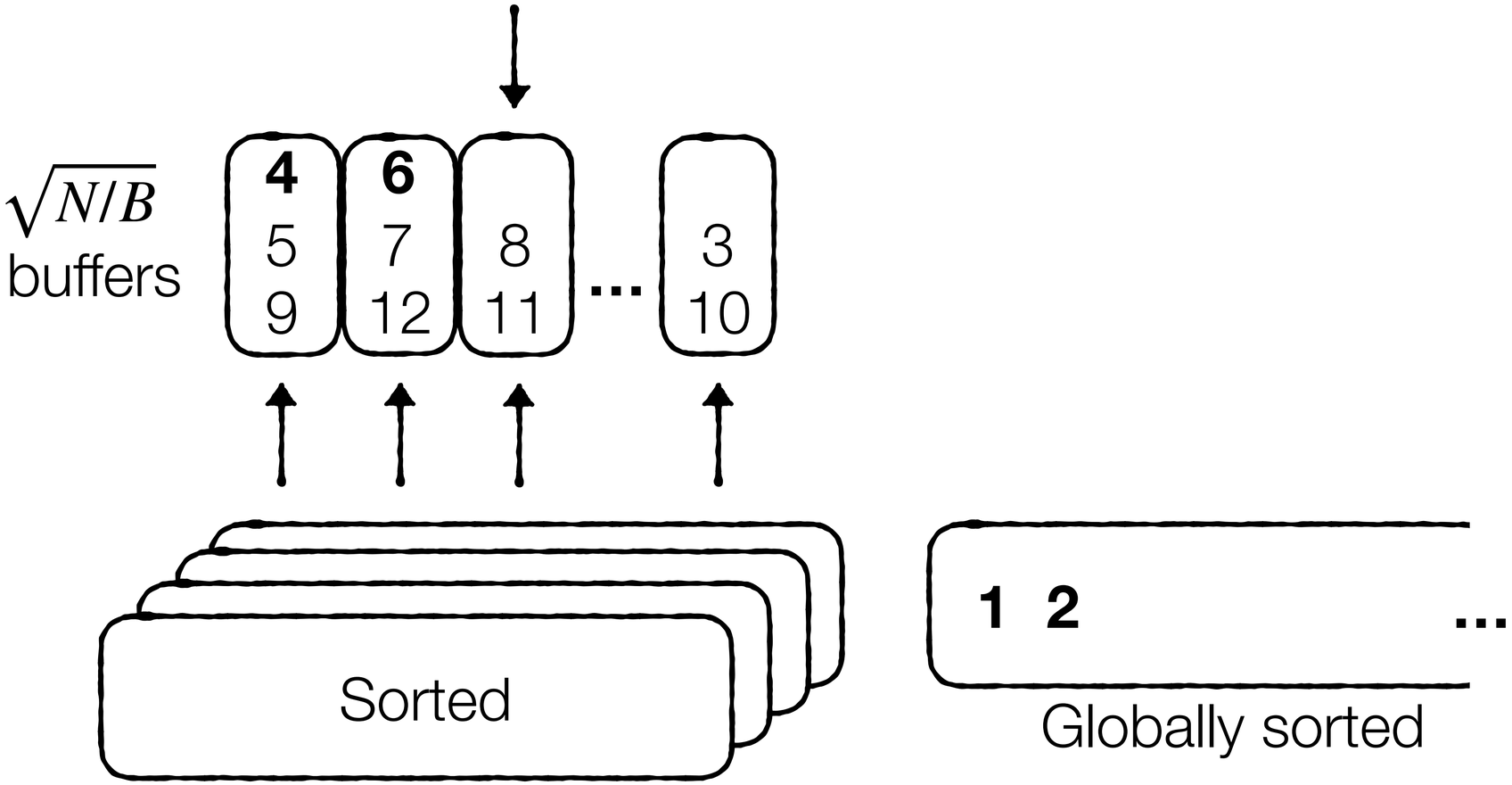
How to merge partitions?

Simple solution: Traverse all buffers and pick minimum



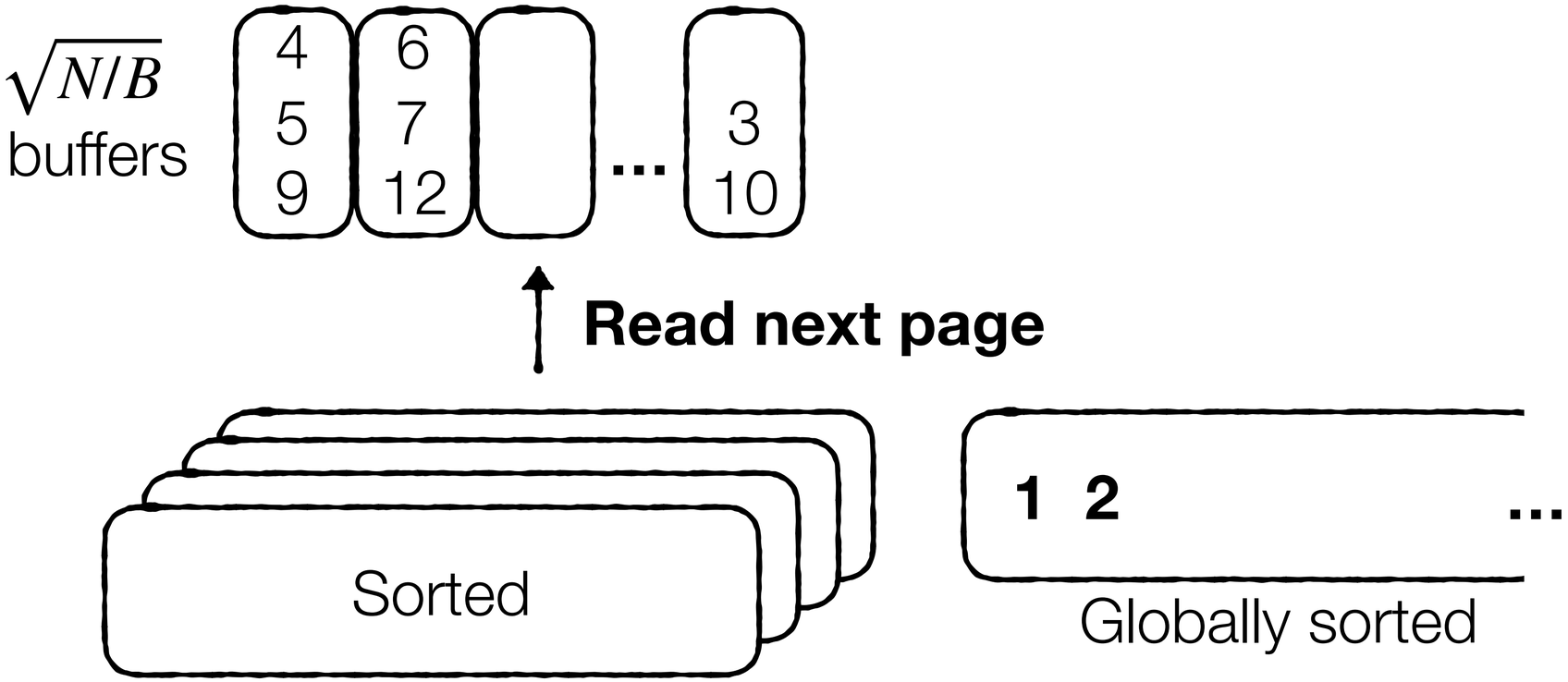
How to merge partitions?

Simple solution: Traverse all buffers and pick minimum



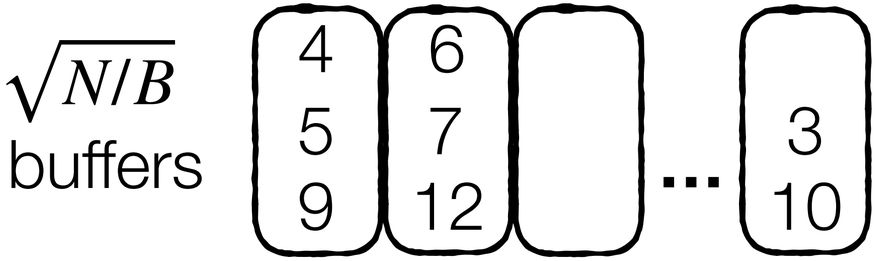
How to merge partitions?

Simple solution: Traverse all buffers and pick minimum

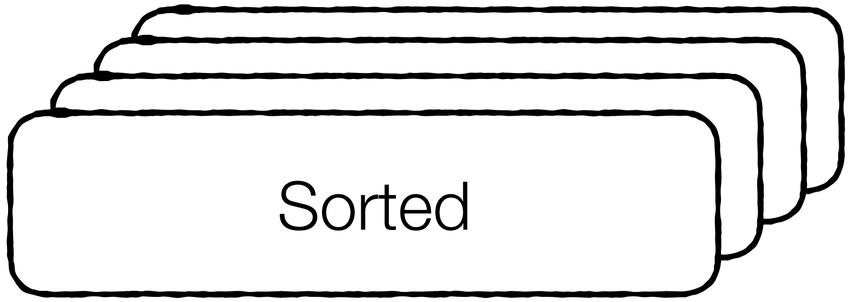


How to merge partitions?

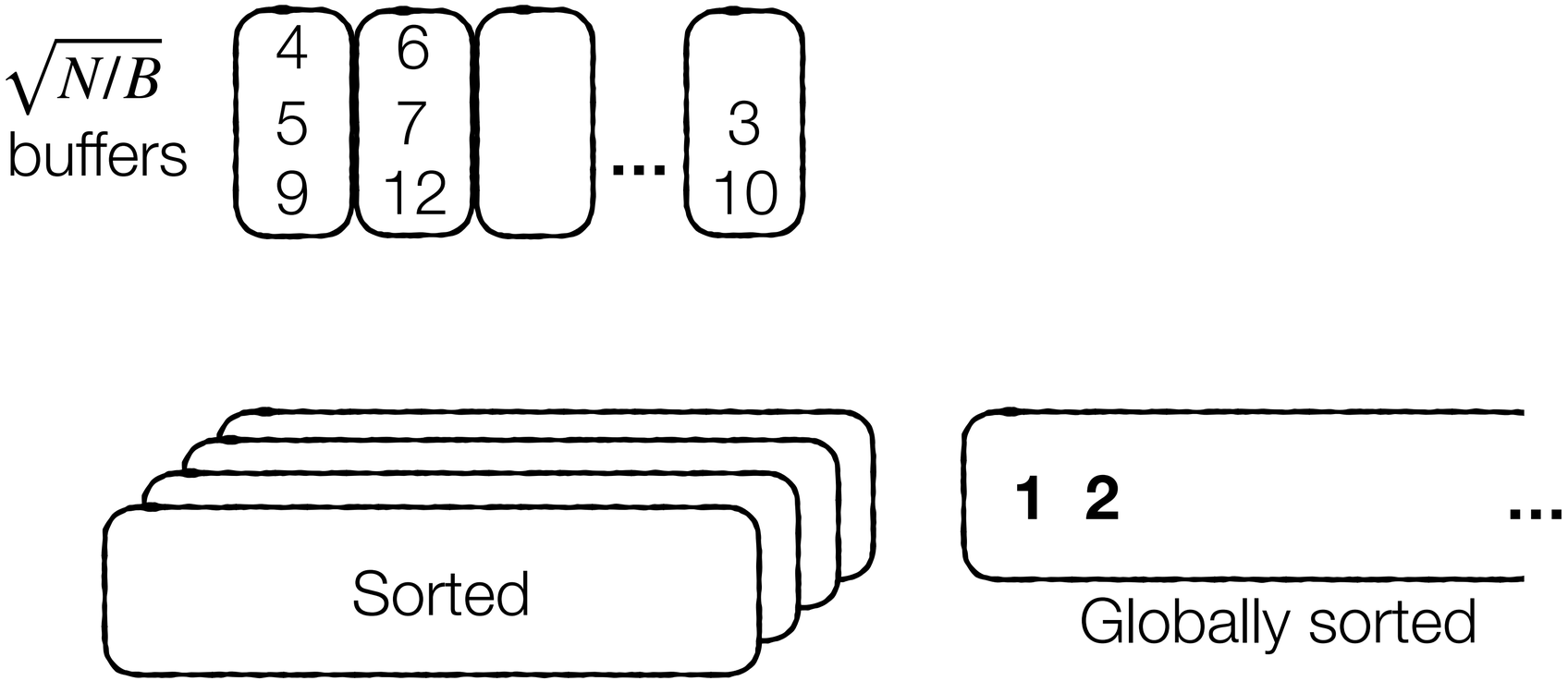
Simple solution: Traverse all buffers and pick minimum



$O(\sqrt{N/B} \cdot N)$ **comparisons**

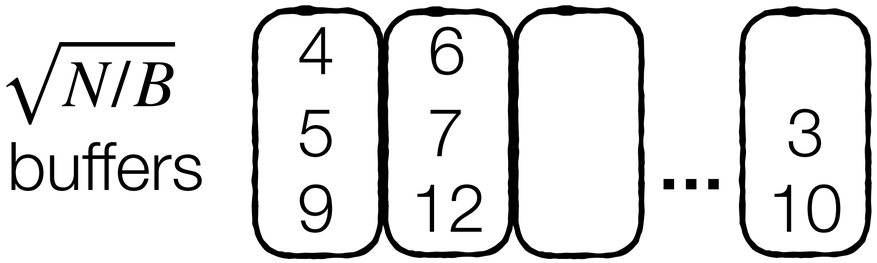


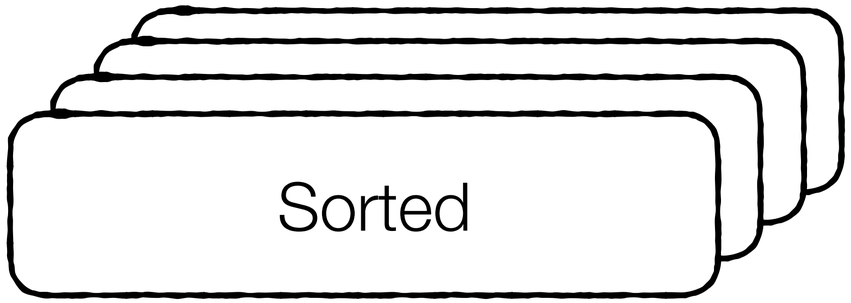
How to merge partitions more efficiently?



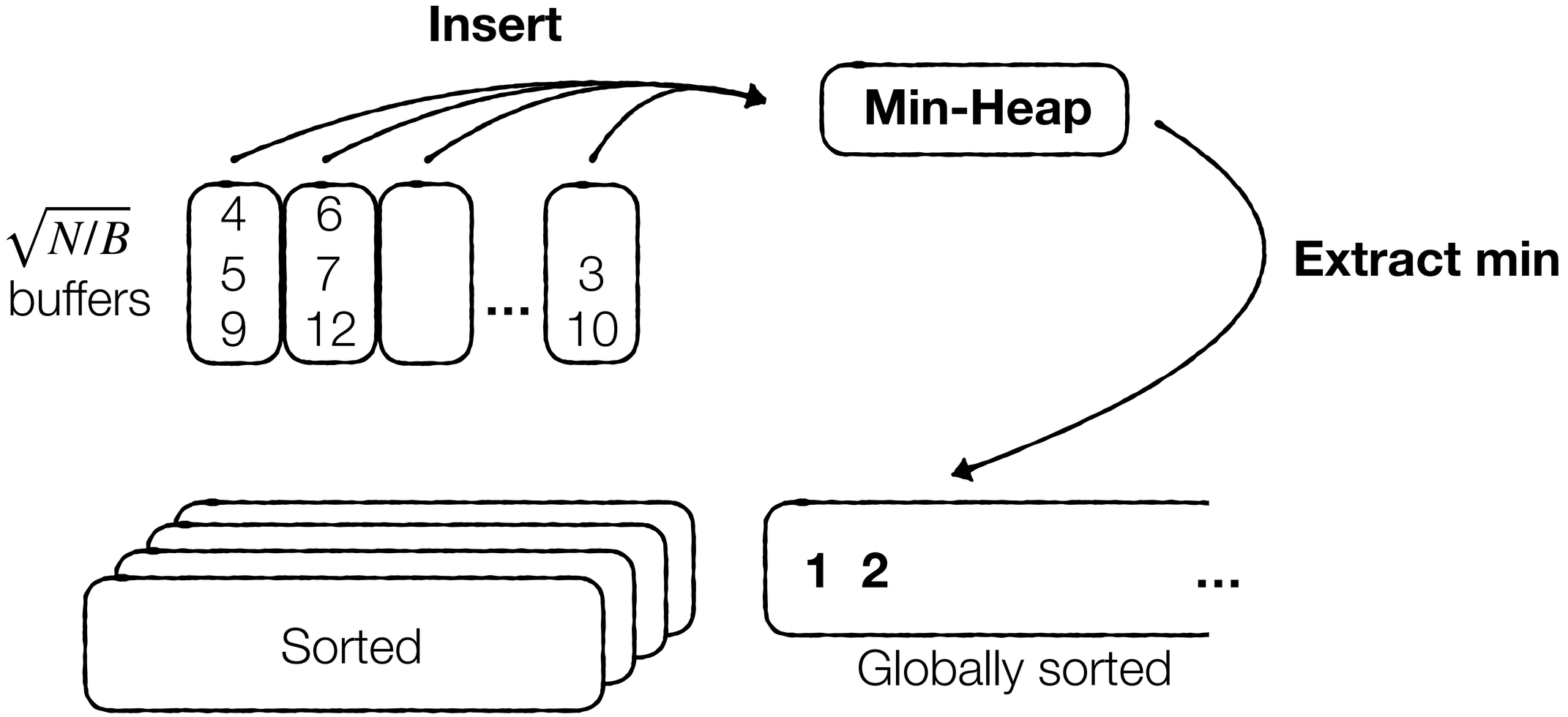
How to merge partitions more efficiently?

Discuss with your neighbors (2 min)





How to merge partitions more efficiently?



Binary Min-Heap

Well-known data structure that efficiently extracts the minimum value in a collection of data items

API	***Runtime***
Insert(key)	$O(\log_2 N)$
Key = extract_min()	$O(\log_2 N)$

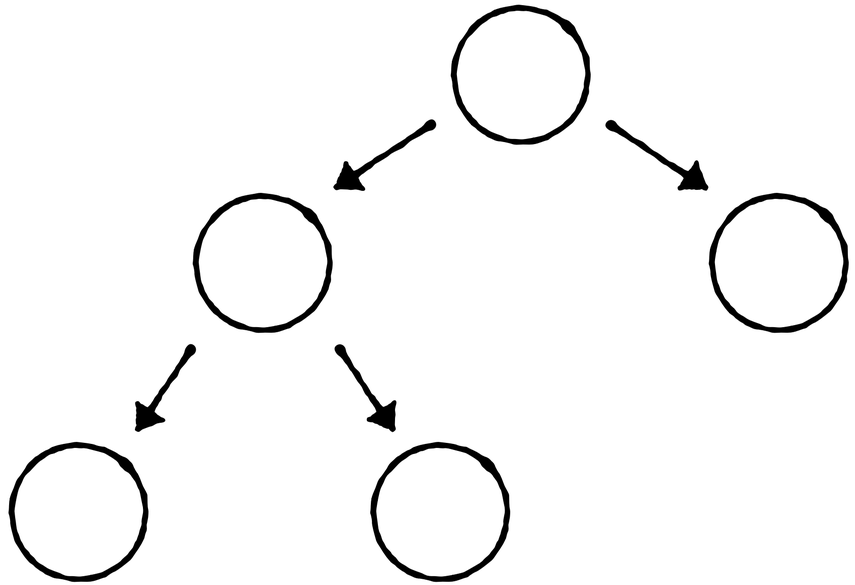
Binary Min-Heap

Well-known data structure that efficiently extracts the minimum value in a collection of data items

API	***Runtime***
Insert(key)	$O(\log_2 N)$
Key = extract_min()	$O(\log_2 N)$
min_key = insert_and_extract(new_key) (efficiently combines both operations)	$O(\log_2 N)$

Binary Min-Heap

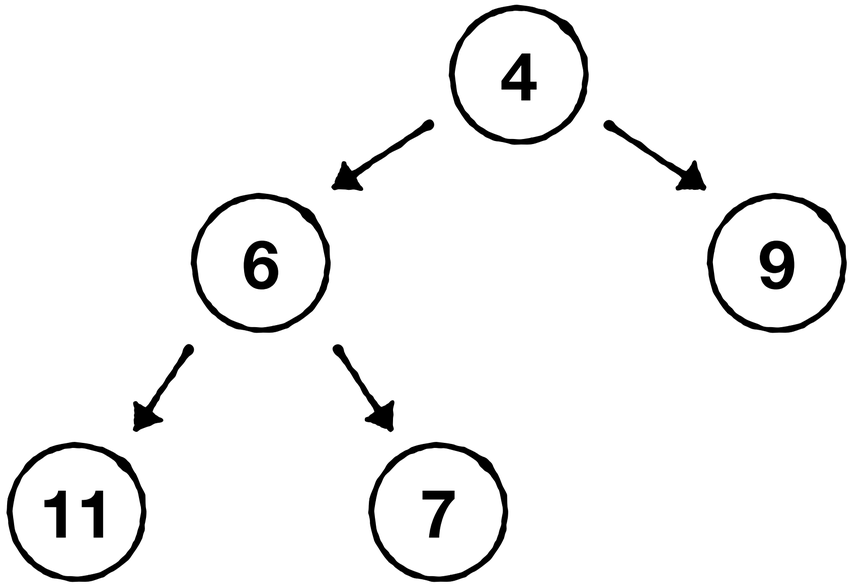
(1) **Complete binary tree**
(All levels are full & largest
level is full from left to right)



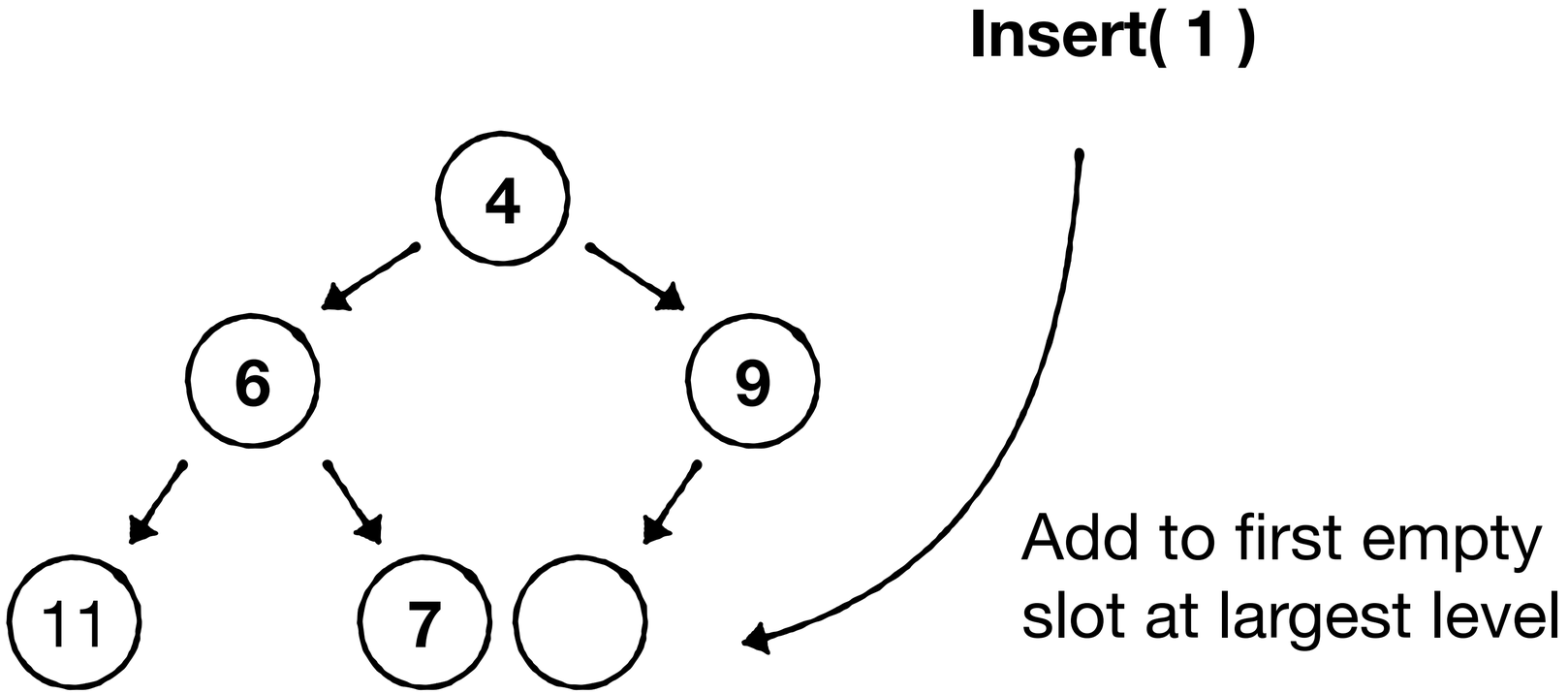
Binary Min-Heap

(1) **Complete binary tree**
(All levels are full & largest level is full from left to right)

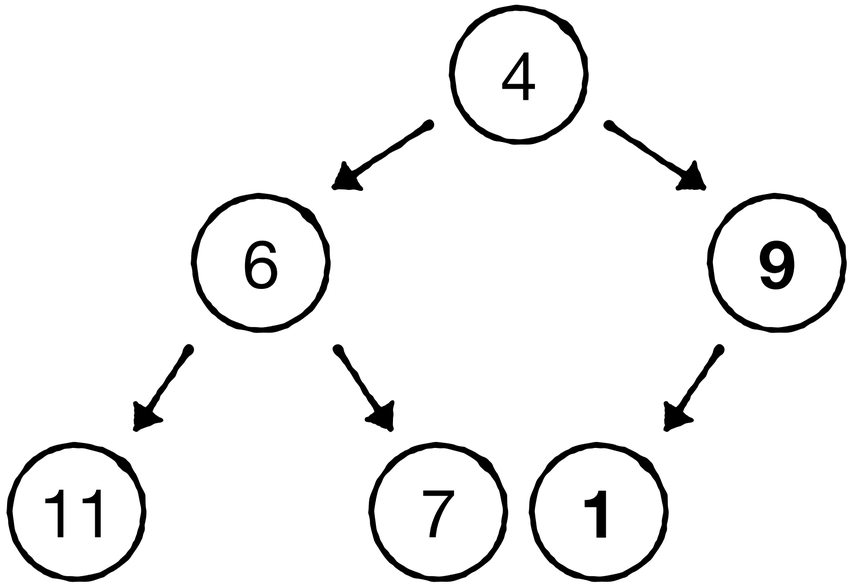
(2) Parent key always smaller than children's.



Binary Min-Heap Insertion

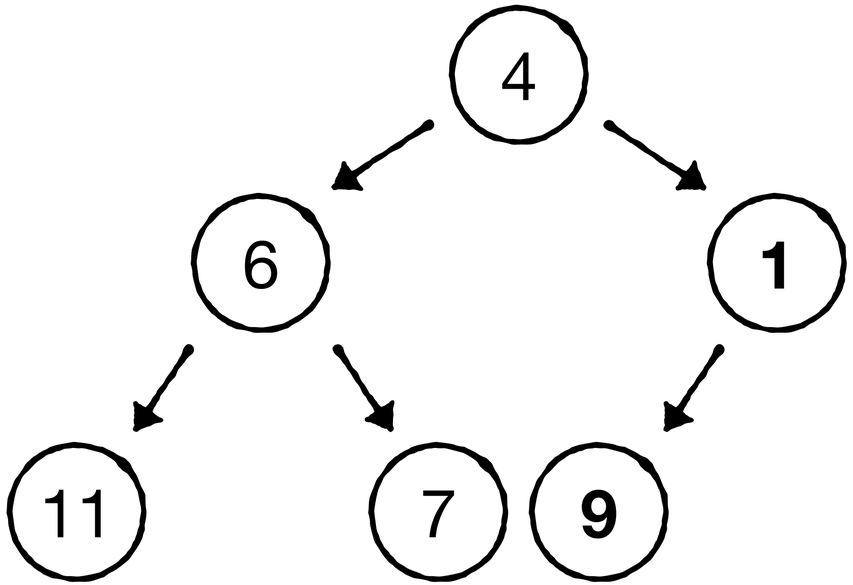


Binary Min-Heap Insertion



Swap until parent is always smaller

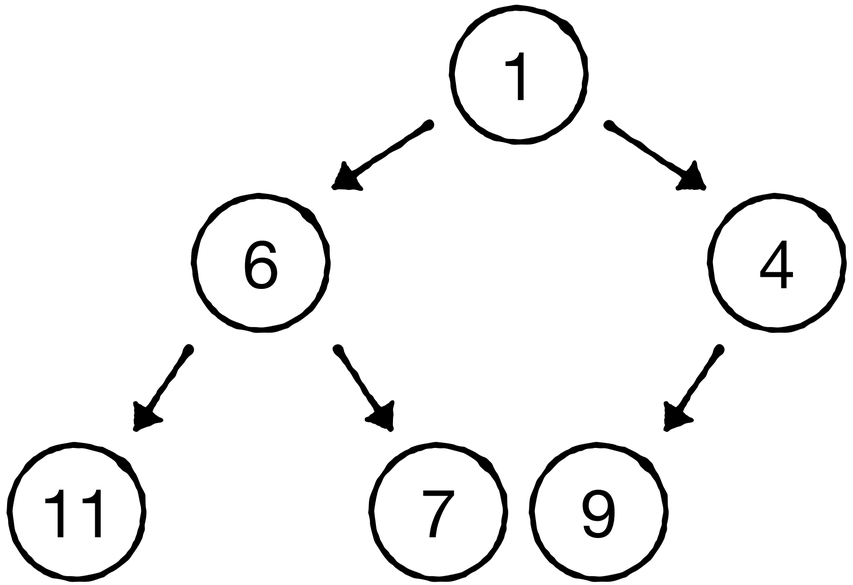
Binary Min-Heap Insertion



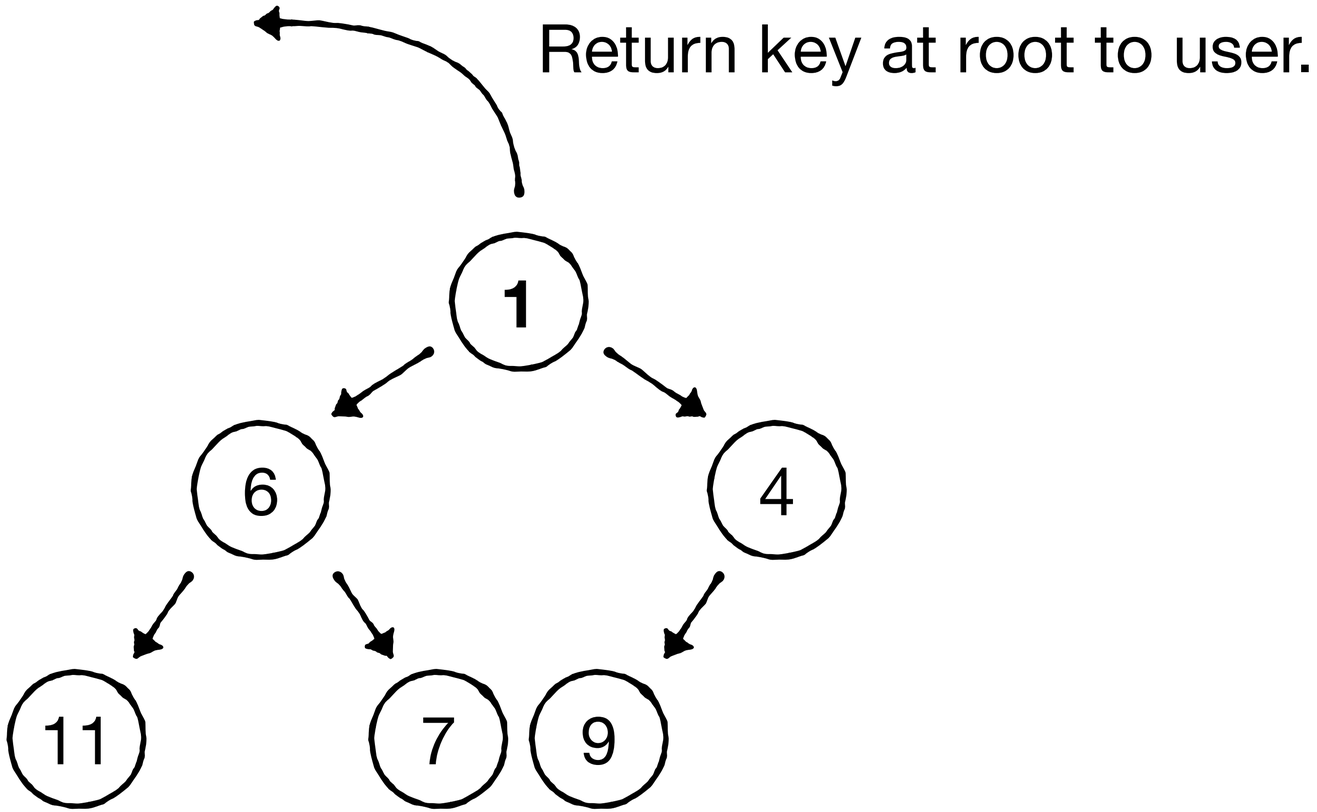
Swap until parent is always smaller

Binary Min-Heap Insertion

$O(\log_2 N)$ insertion cost

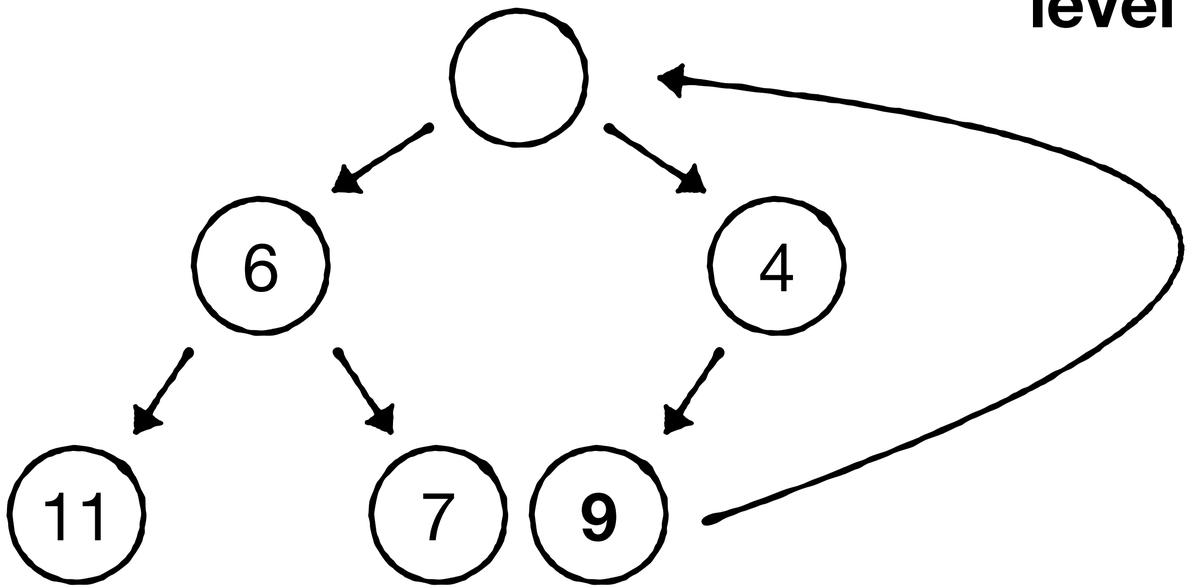


Binary Min-Heap Extract Minimum

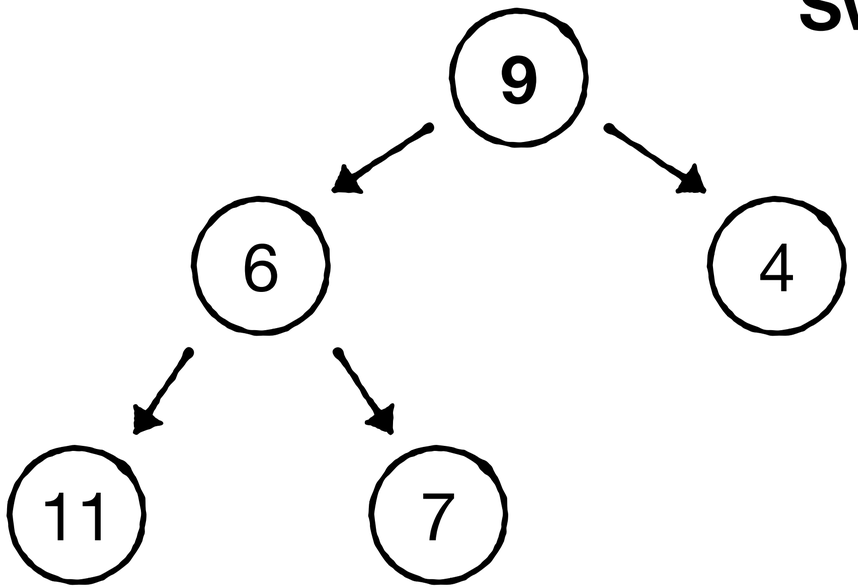


Binary Min-Heap Extract Minimum

Set last key at last level to be new root



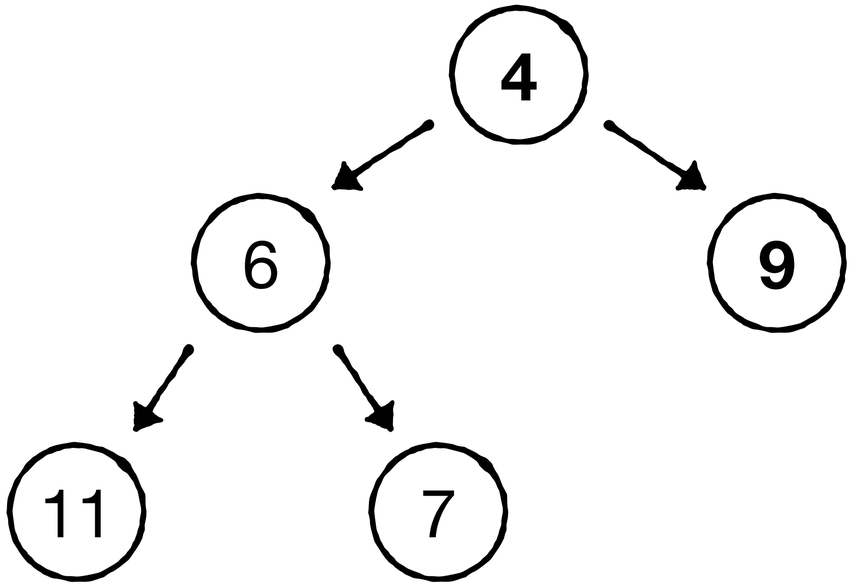
Binary Min-Heap Extract Minimum



Swap with smaller child

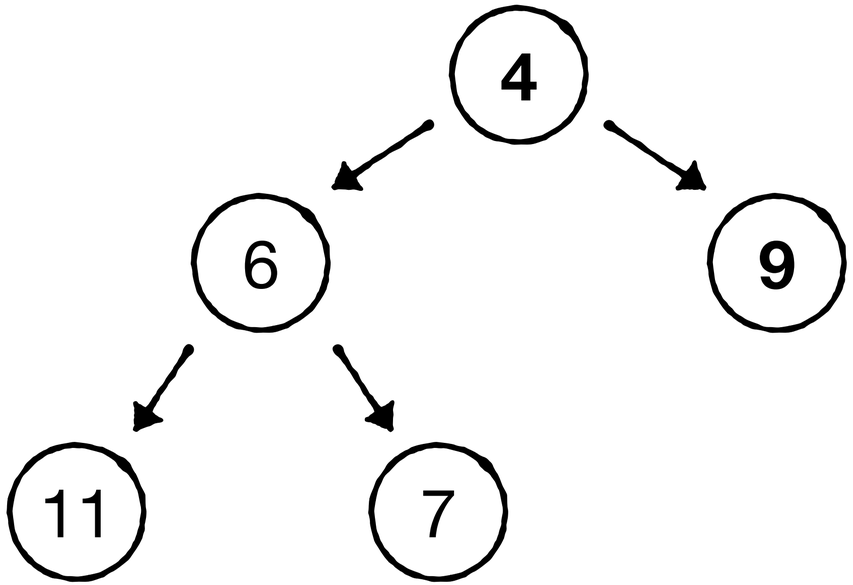
Binary Min-Heap Extract Minimum

$O(\log_2 N)$ min extraction cost

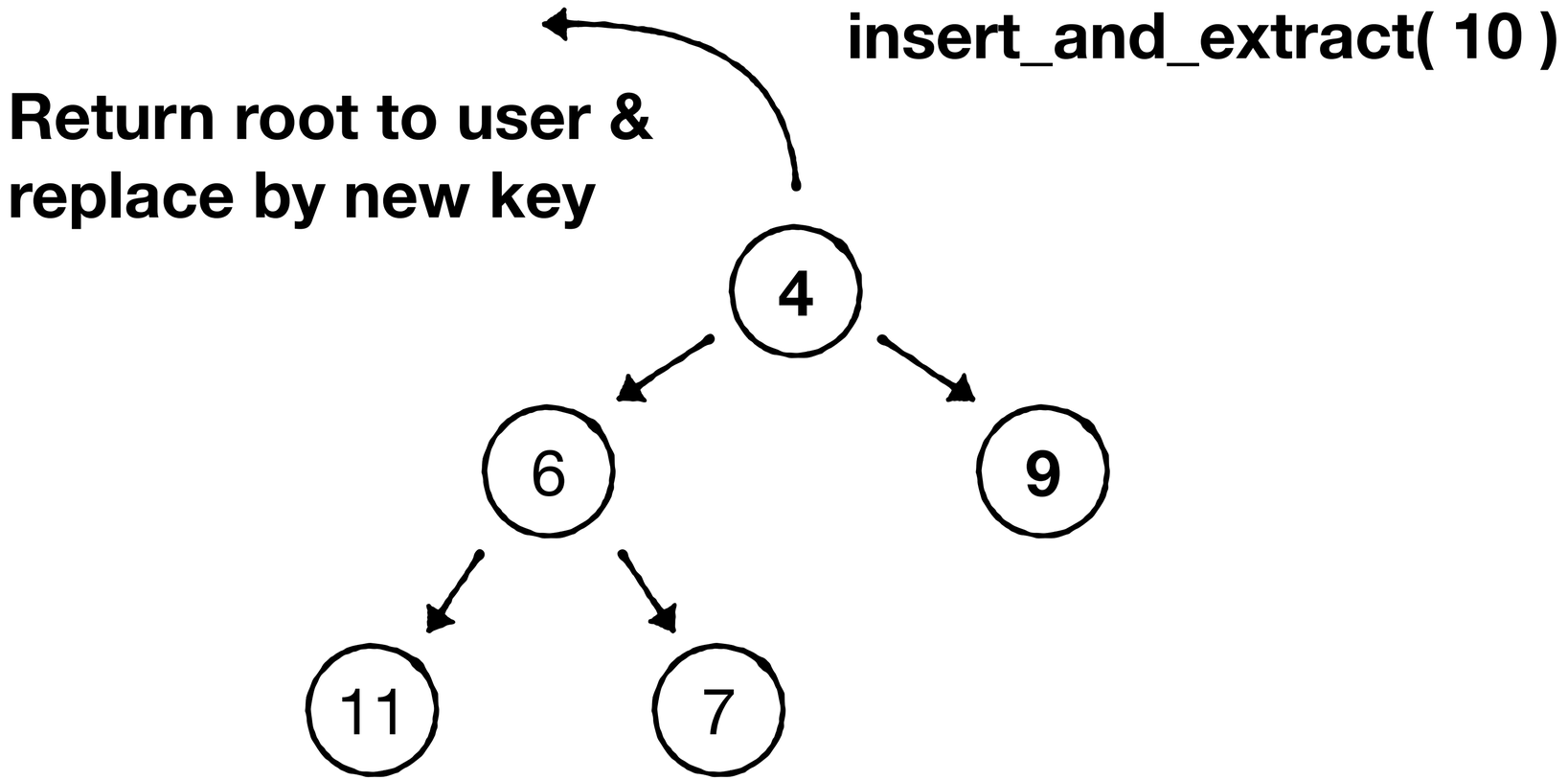


Binary Min-Heap Insert & extract

min_key = insert_and_extract(new_key)

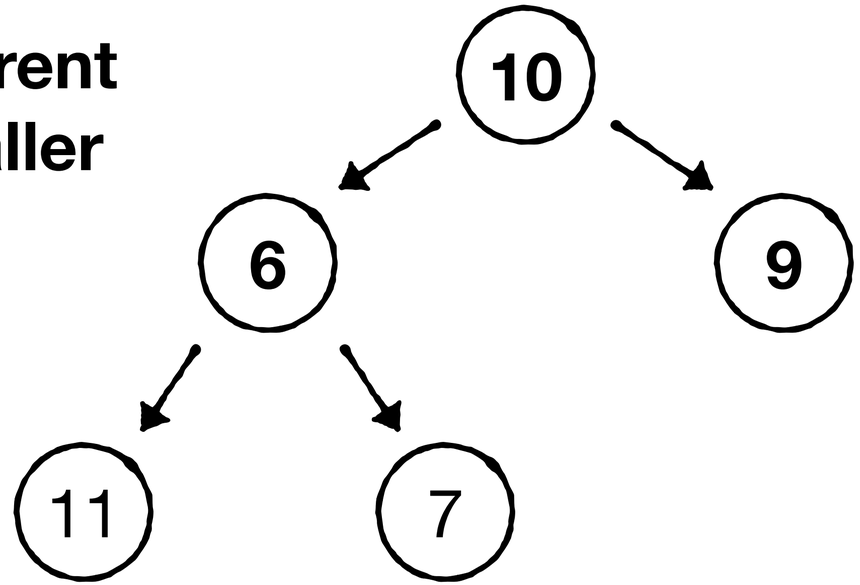


Binary Min-Heap Insert & extract



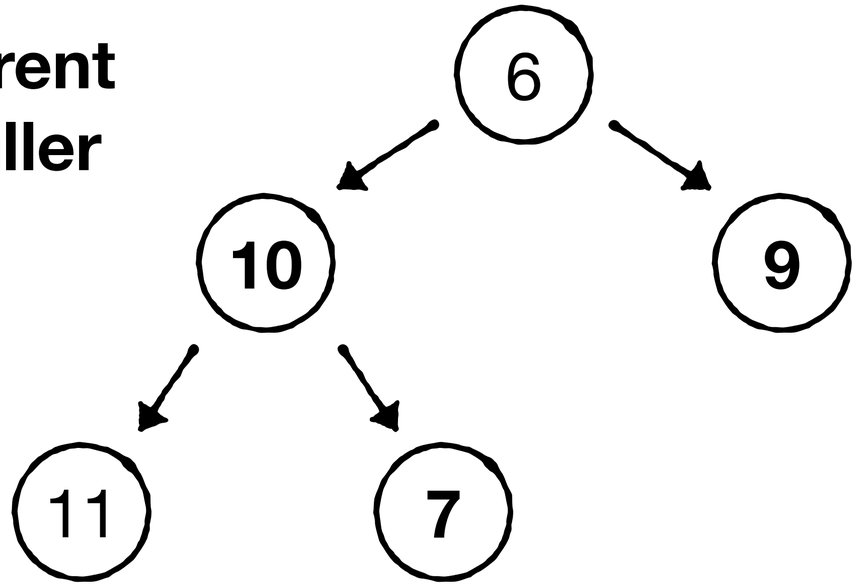
Binary Min-Heap Insert & extract

Swap until parent is always smaller



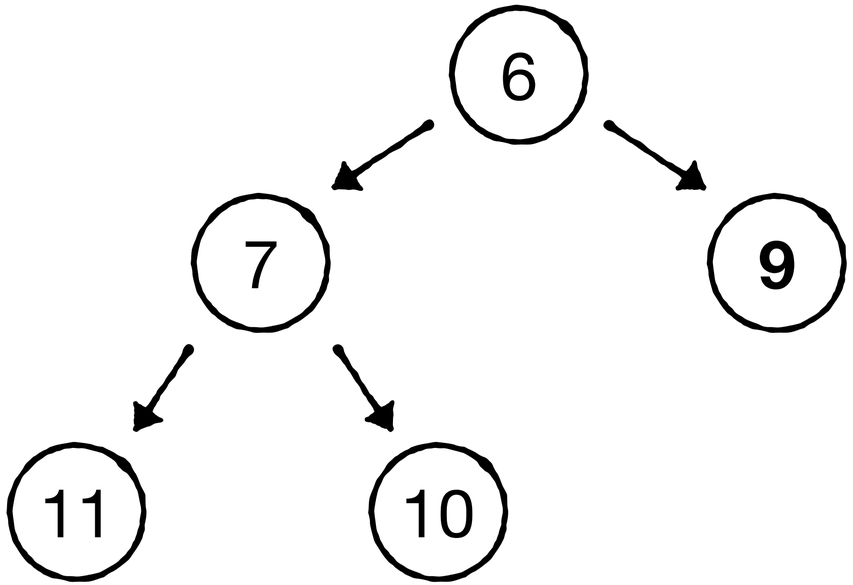
Binary Min-Heap Insert & extract

Swap until parent is always smaller



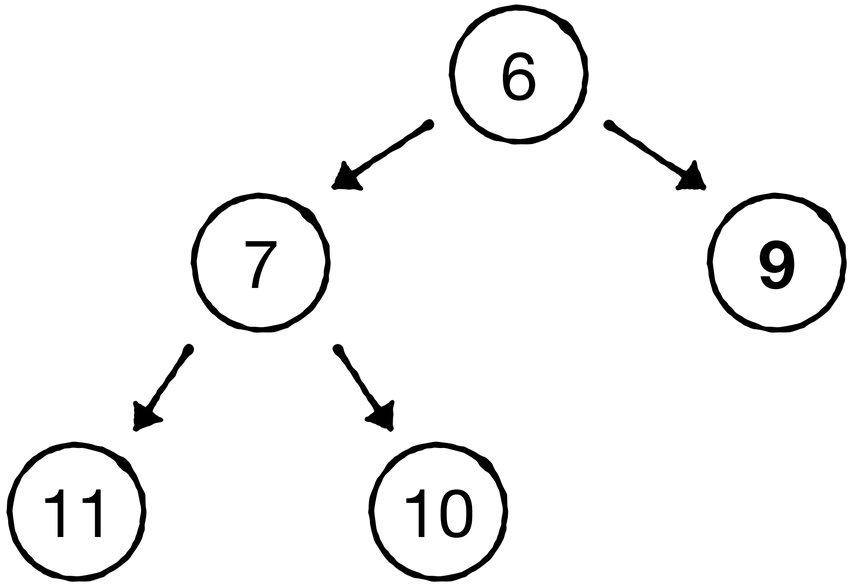
Binary Min-Heap Insert & extract

$O(\log_2 N)$ for insert_and_extract

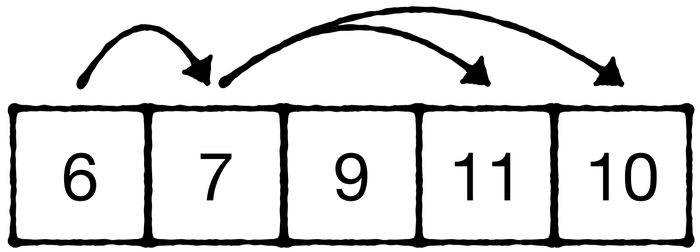


Binary Min-Heap Implementation

Tree with Pointers



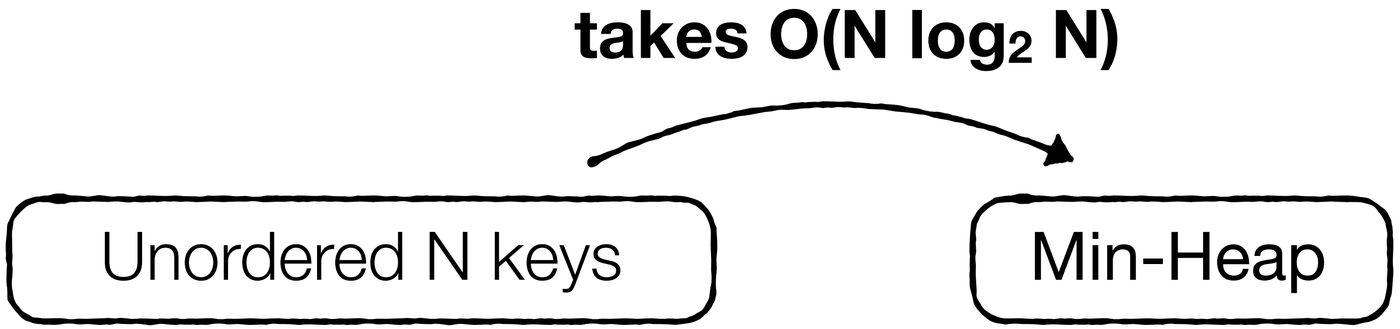
Array



Compact since the binary tree is complete. Avoids overhead of pointers.

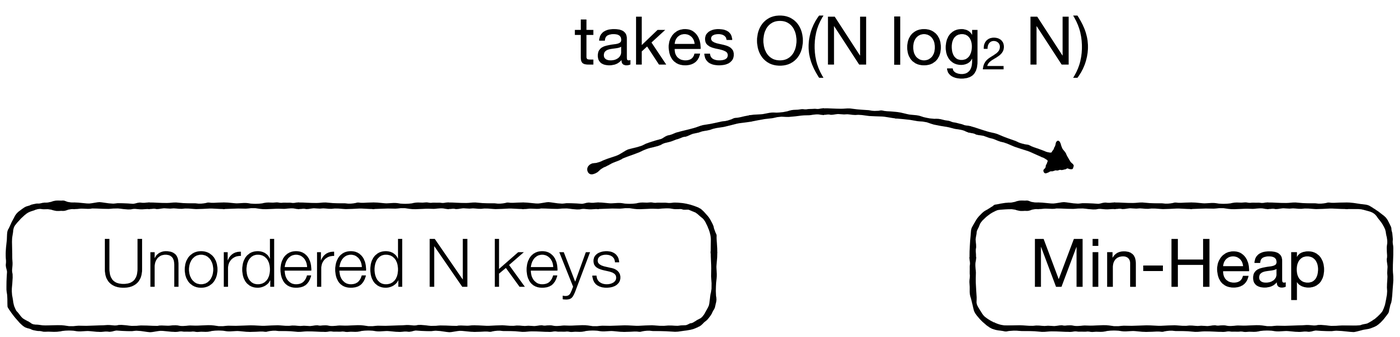
Binary Min-Heap Construction

We can construct a heap for N entries using normal insertions



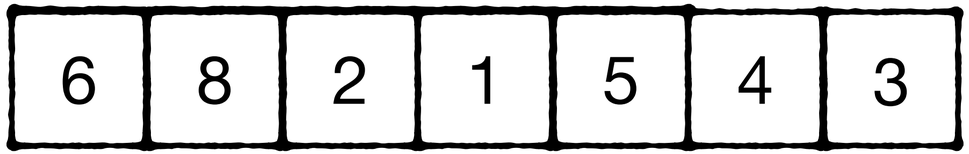
Binary Min-Heap Construction

We can construct a heap for N entries using normal insertions



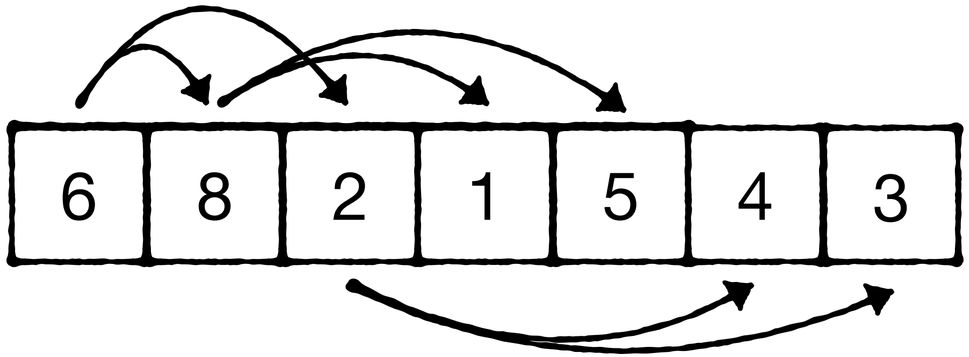
We can do better :)

Binary Min-Heap Efficient Array Construction

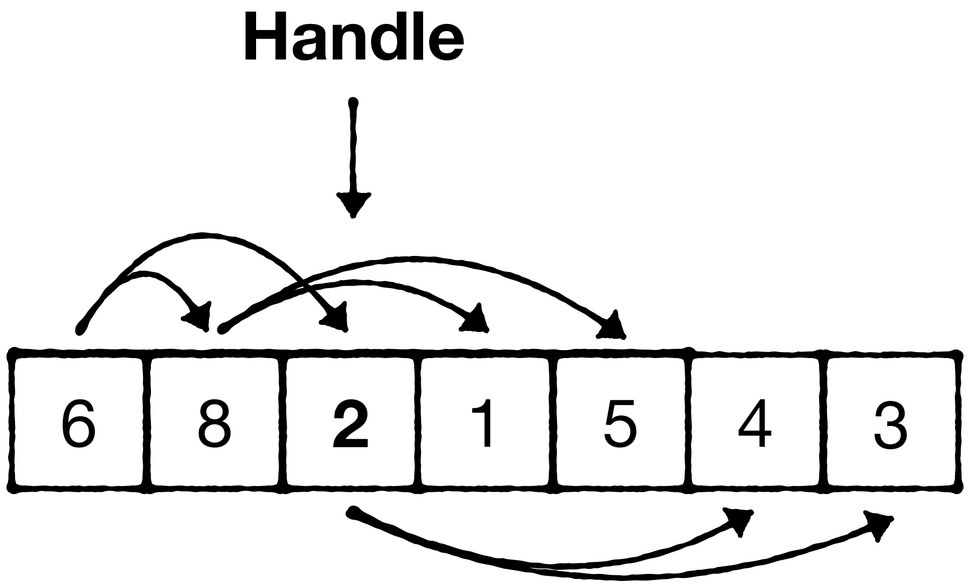


Start with Unordered N keys

Binary Min-Heap Efficient Array Construction

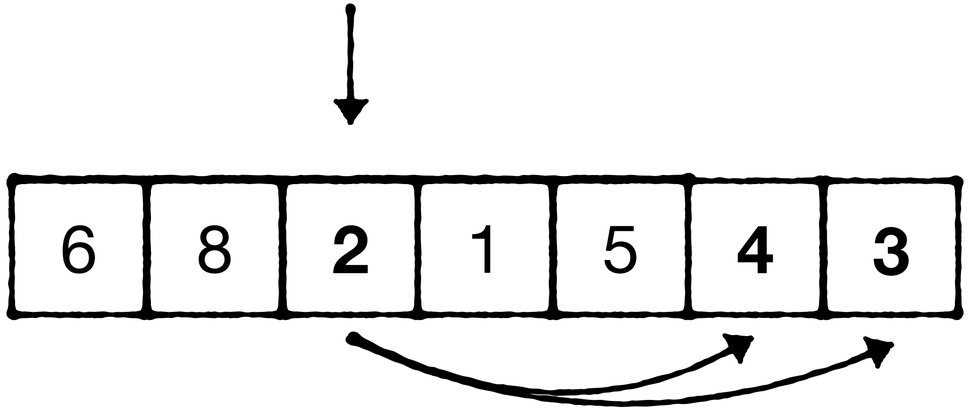


We can infer which slots should be the parent of which other slots.



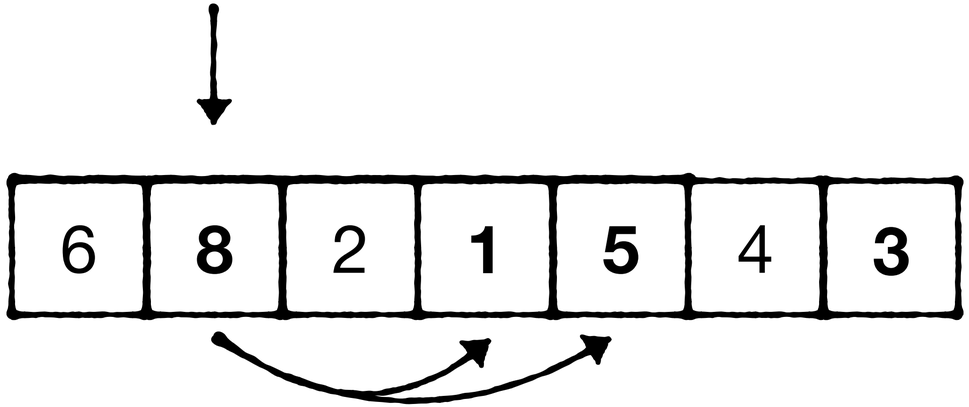
For each parent node starting from right and moving left, swap until every parent in sub-tree is smaller than its children

Handle (Do nothing)

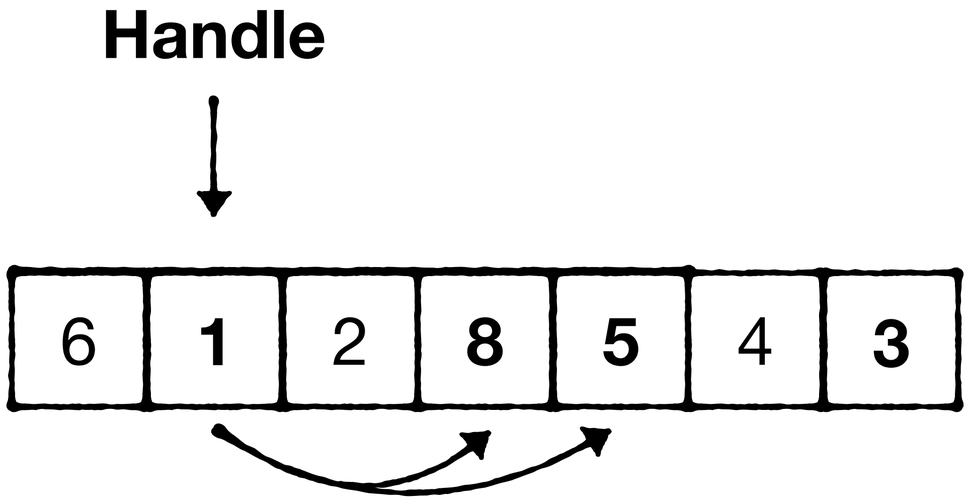


For each parent node starting from right and moving left, swap until every parent in sub-tree is smaller than its children

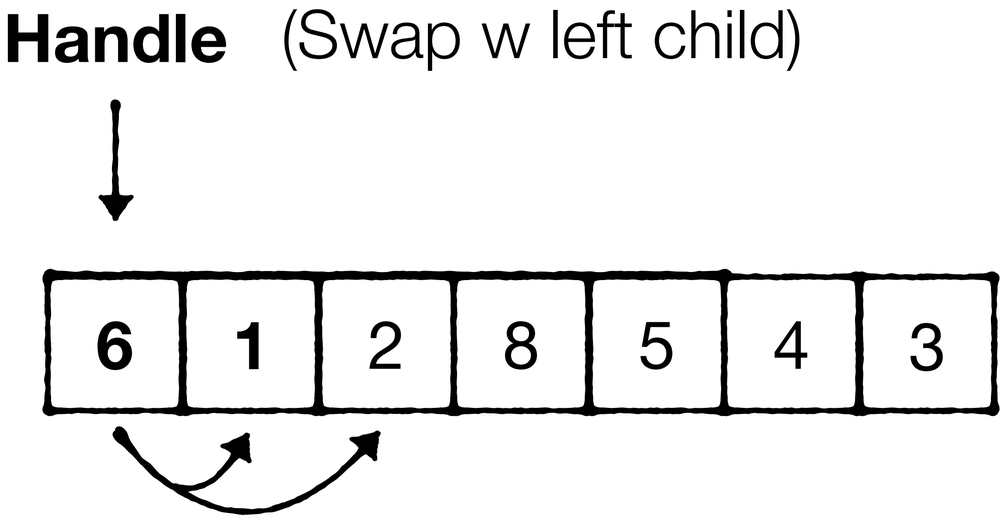
Handle (Swap w left child)



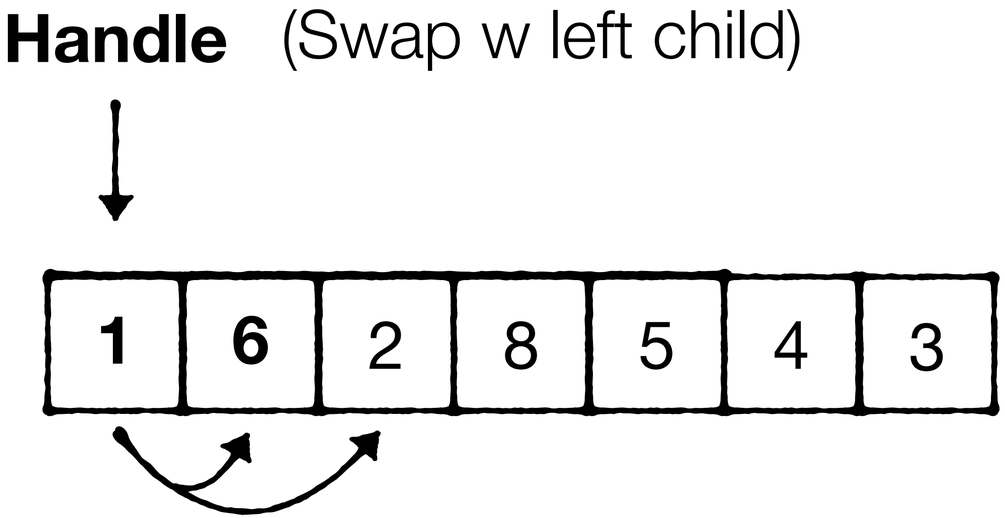
For each parent node starting from right and moving left, swap until every parent in sub-tree is smaller than its children



For each parent node starting from right and moving left, swap until every parent in sub-tree is smaller than its children

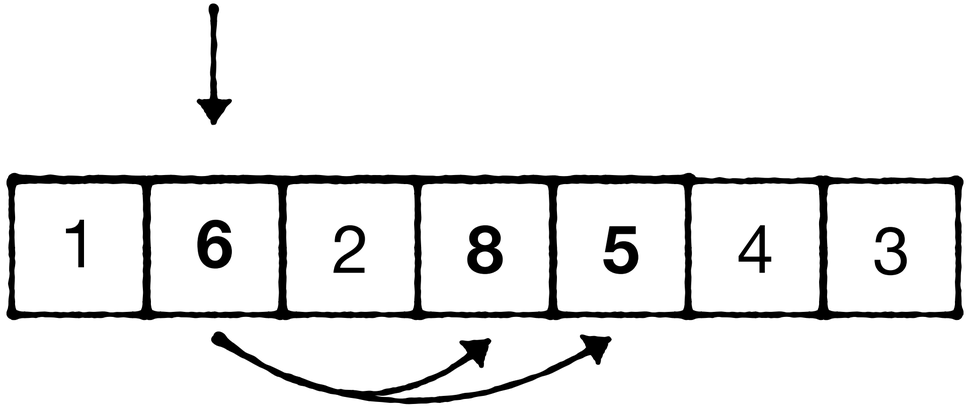


For each parent node starting from right and moving left, swap until every parent in sub-tree is smaller than its children

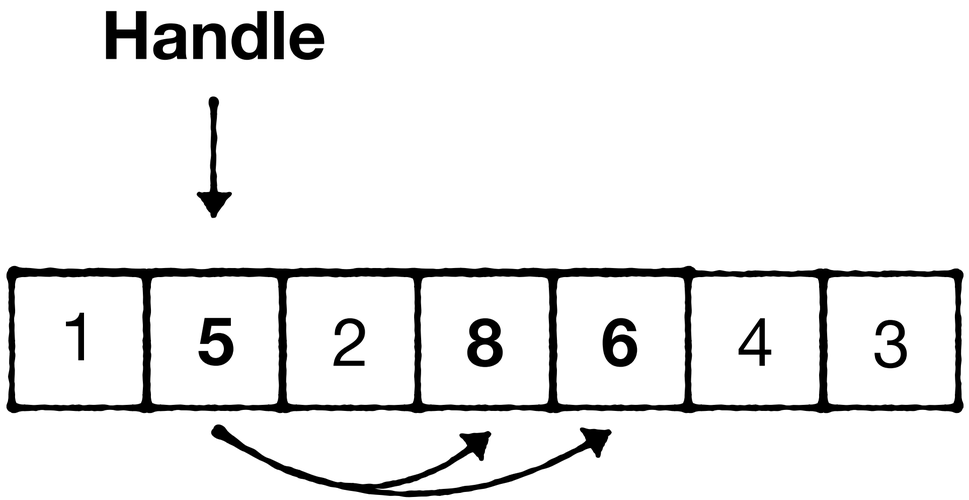


For each parent node starting from right and moving left, swap until every parent in sub-tree is smaller than its children

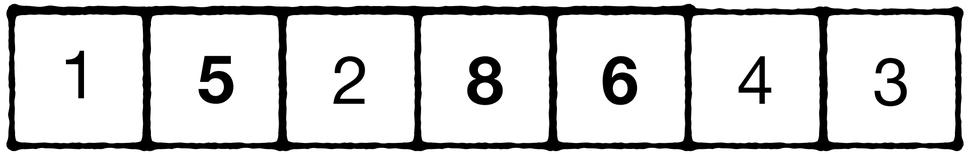
Handle (Swap w right child)



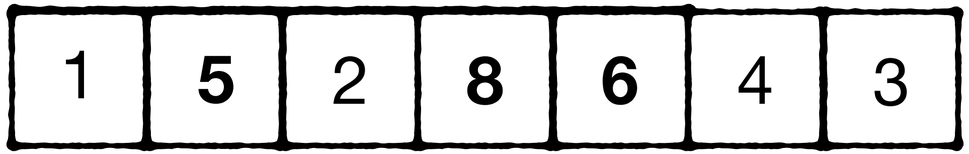
For each parent node starting from right and moving left, swap until every parent in sub-tree is smaller than its children



For each parent node starting from right and moving left, swap until every parent in sub-tree is smaller than its children

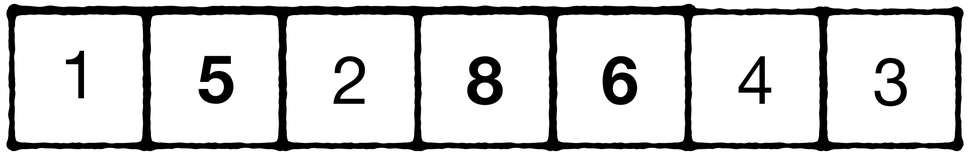


Max swaps: **2 1 1 0 0 0 0**



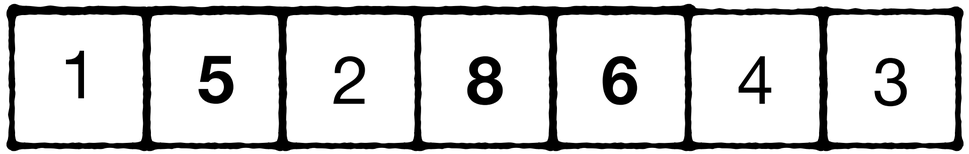
Max swaps: **2 1 1 0 0 0 0**

Max swaps for a sub-tree is equal to its depth



Max swaps: **2 + 1 + 1 + 0 + 0 + 0 + 0 = O(N)**

Max swaps for a sub-tree is equal to its depth

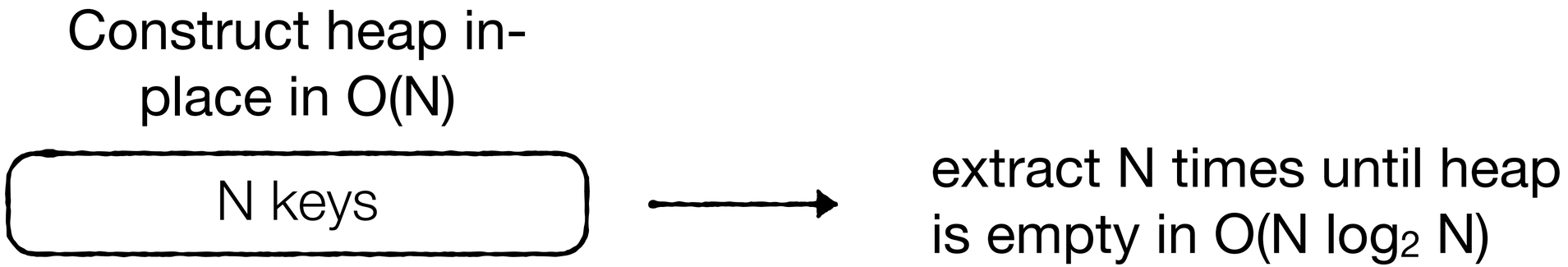


O(N) < O(N log$_2$ N)

from before with pure insertions

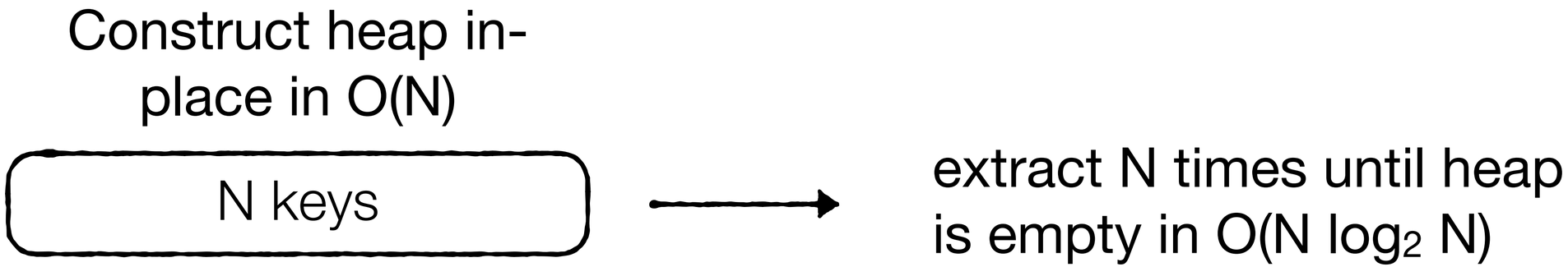
Side-Note on Heap-Sort

If data fits in memory, we can sort it using a heap

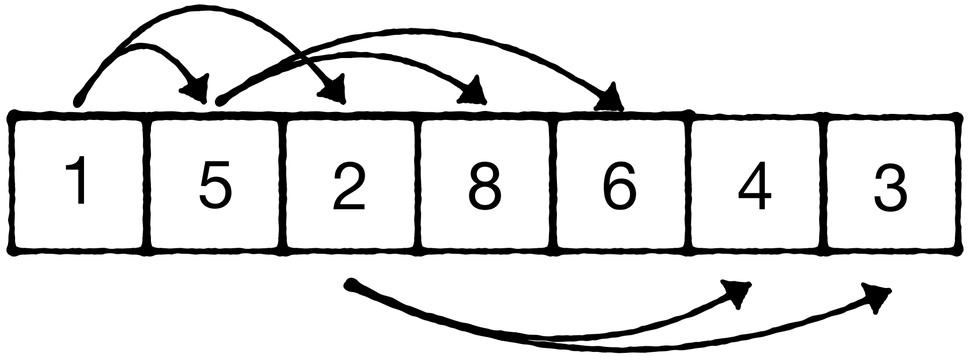


Side-Note on Heap-Sort

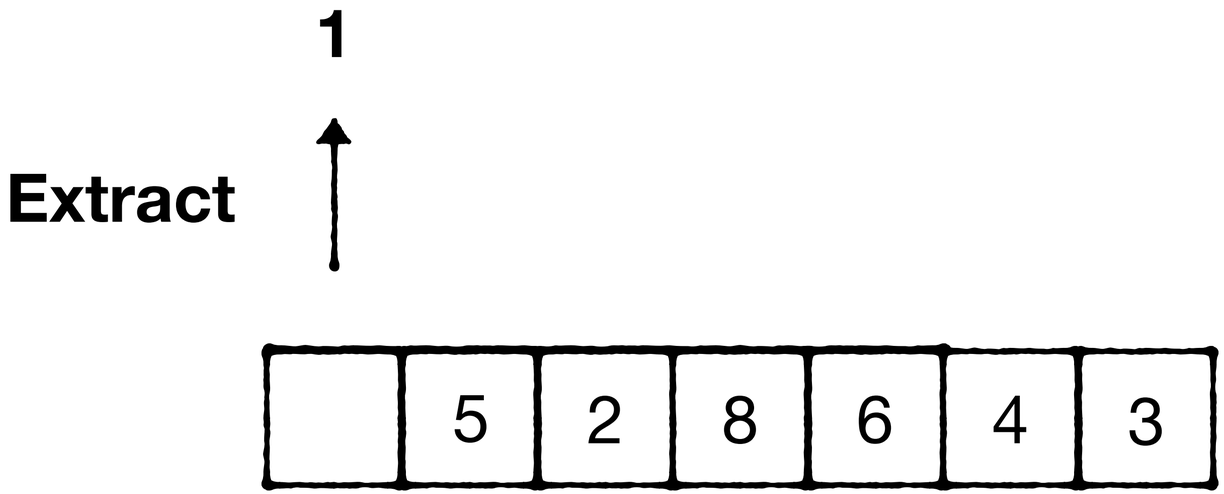
If data fits in memory, we can sort it using a heap

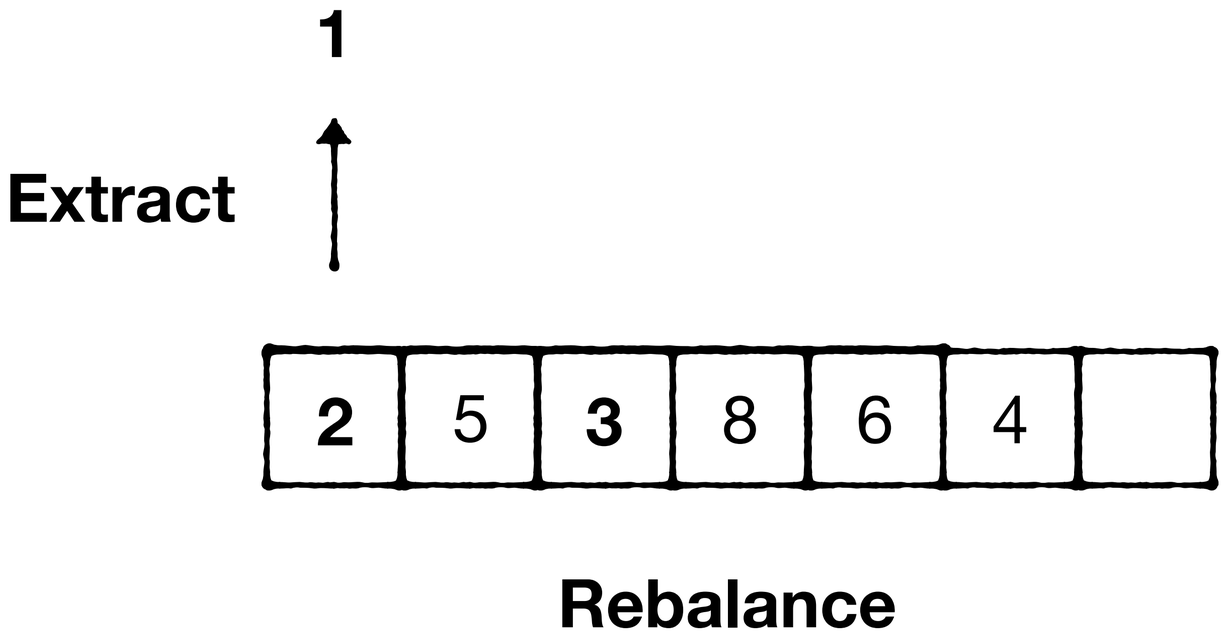


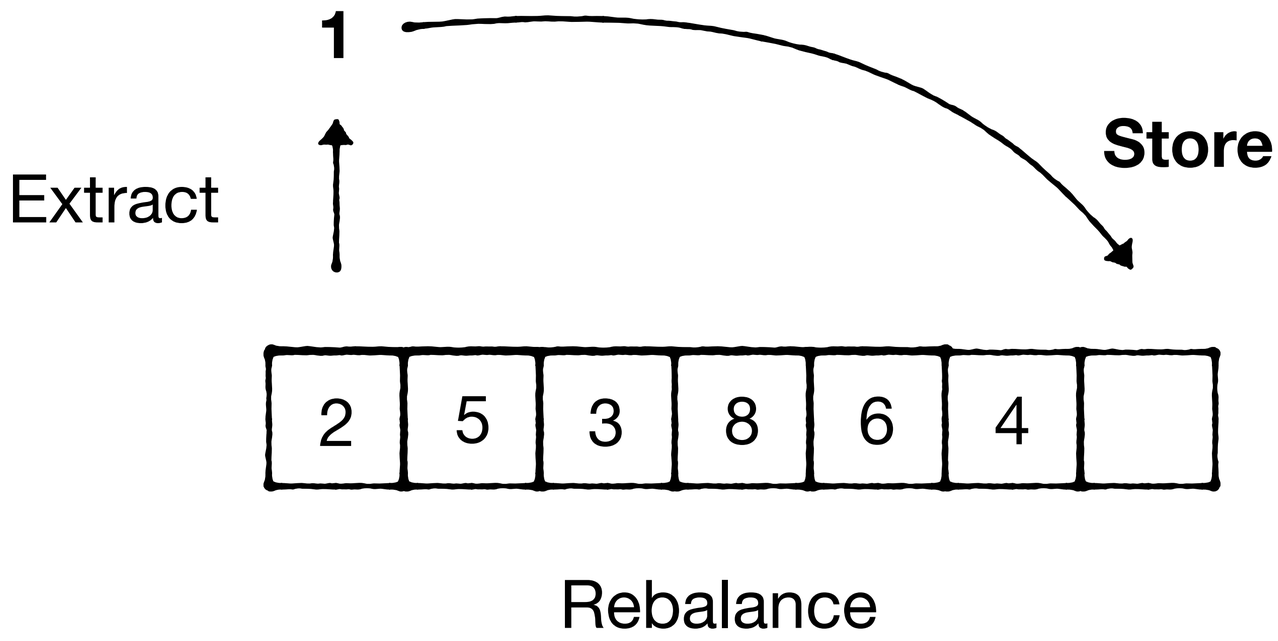
Can you do this in-place? :)

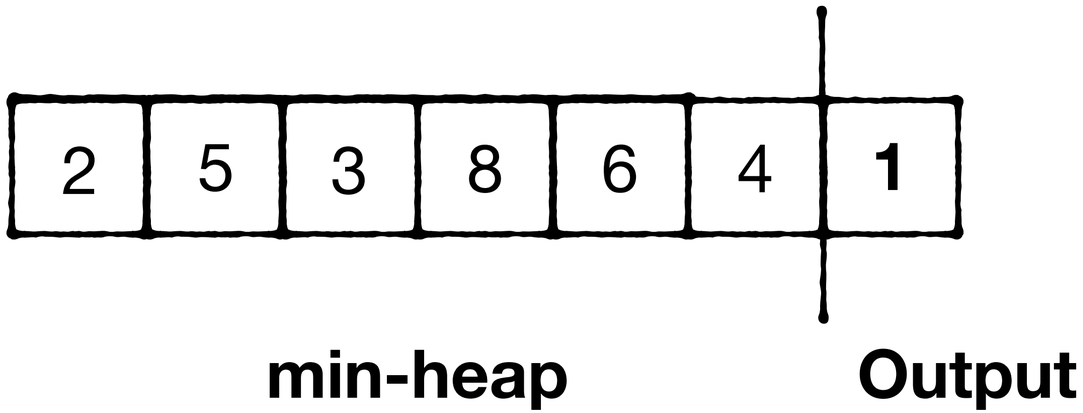


Recall min-heap can be stored as compact array

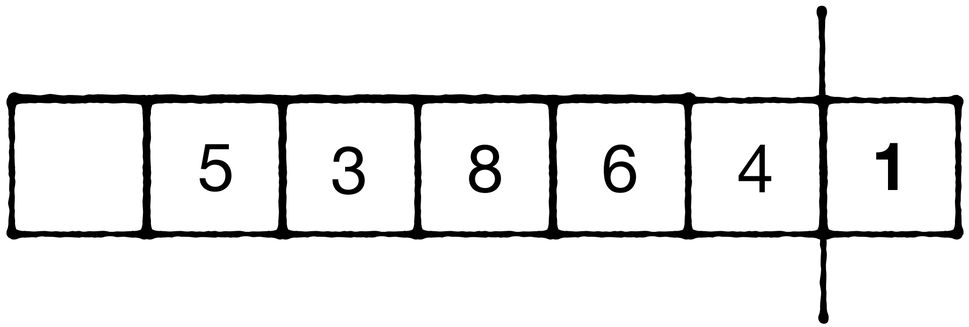

1
Extract
5
2
8
6
4
3


1
Extract
2
5
3
8
6
4
Rebalance

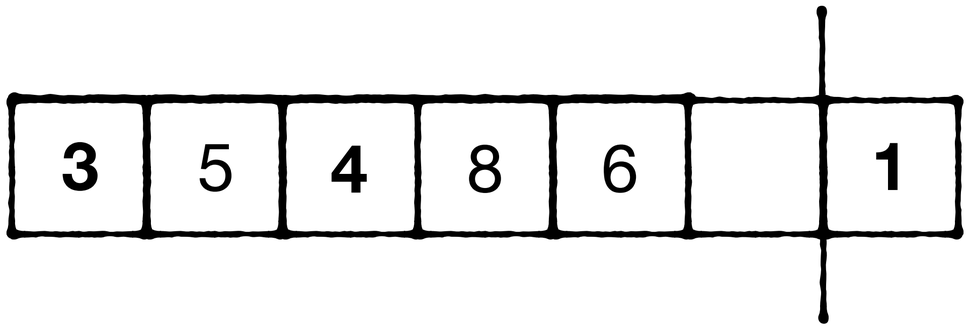

1
Store
Extract
2
5
3
8
6
4
Rebalance

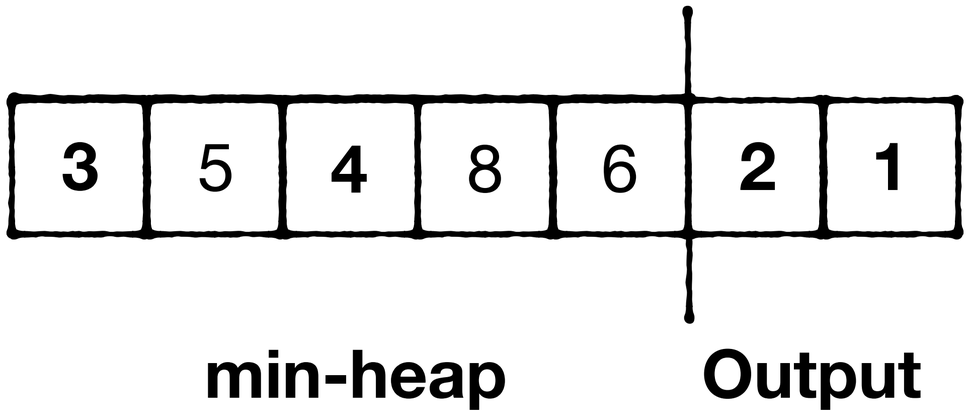

2
5
3
8
6
4
1
min-heap
Output

2

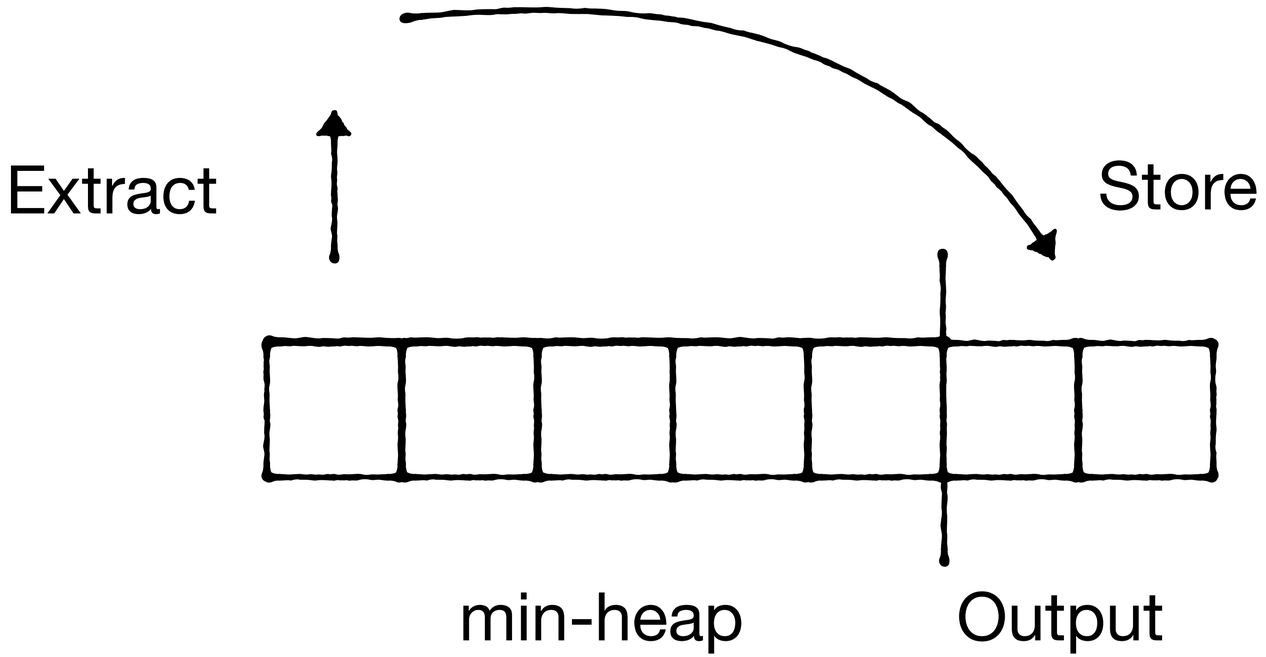


2

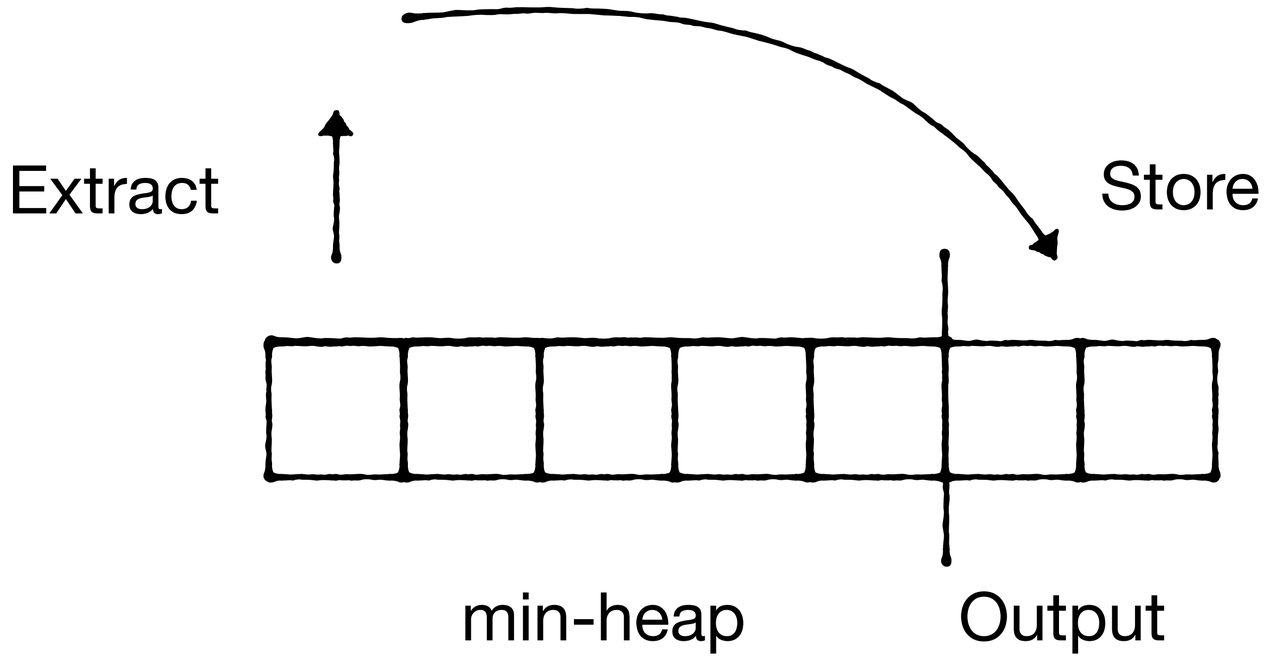



3
5
4
8
6
2
1
min-heap
Output

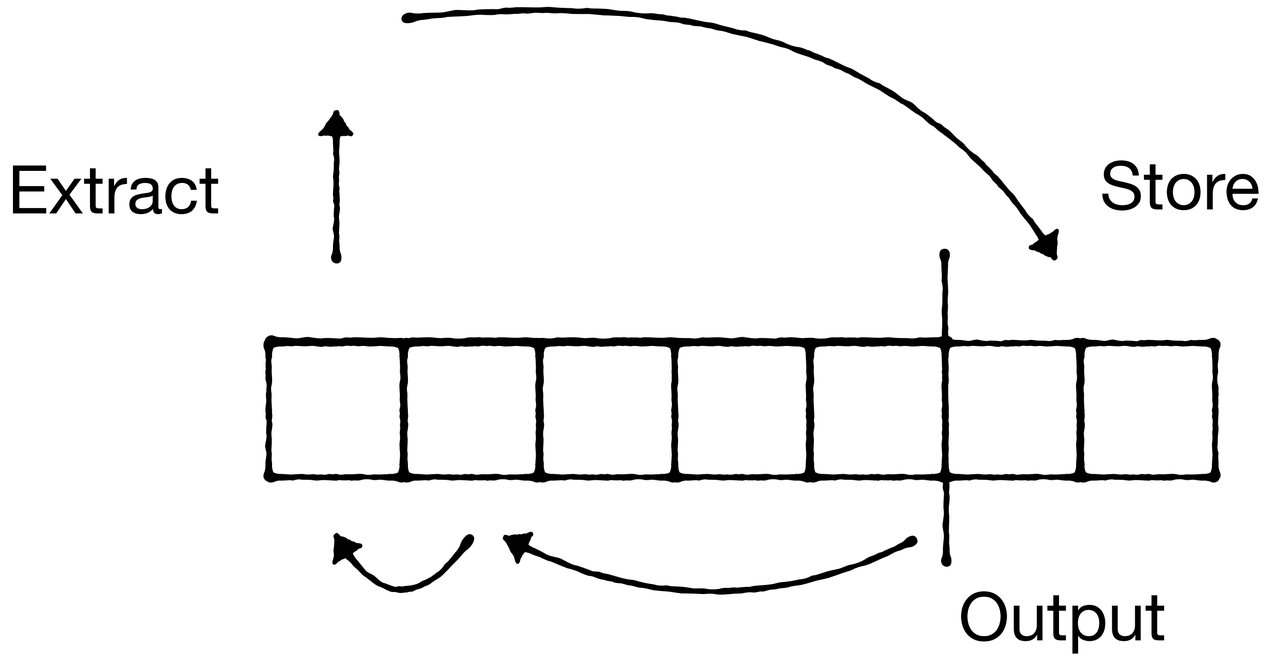
in-place algorithm :)



Downsides compared to quick-sort or merge-sort?



Downsides compared to quick-sort or merge-sort?



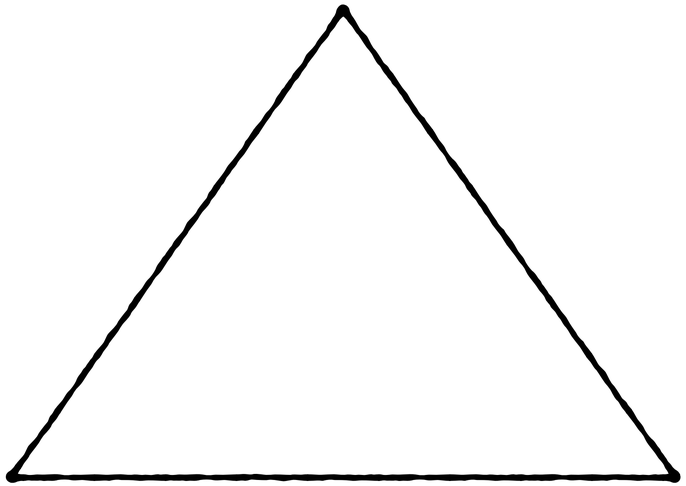
Random access to rebalance

Heap-Sort

Worst case O(N log N)

In-place

More random access



Merge-Sort

Worst case O(N log N)

2x more space

Sequential access

Quick-Sort

Avg. worst case O(N log N)

In-place

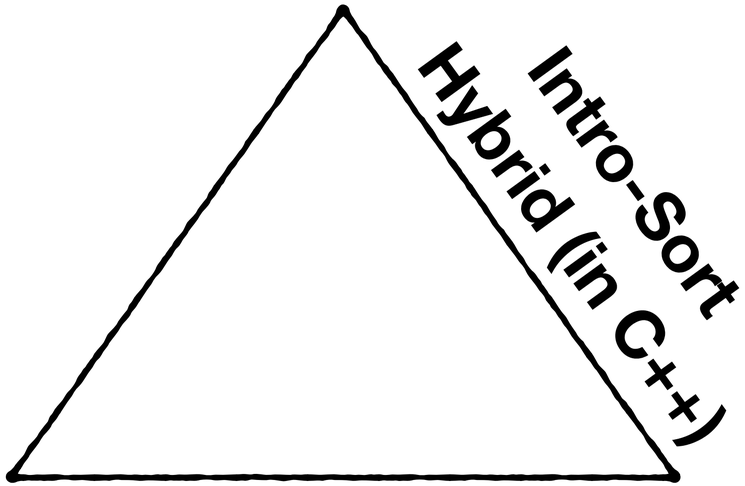
Sequential access

Heap-Sort

Worst case O(N log N)

In-place

More random access



Merge-Sort

Worst case O(N log N)

2x more space

Sequential access

Quick-Sort

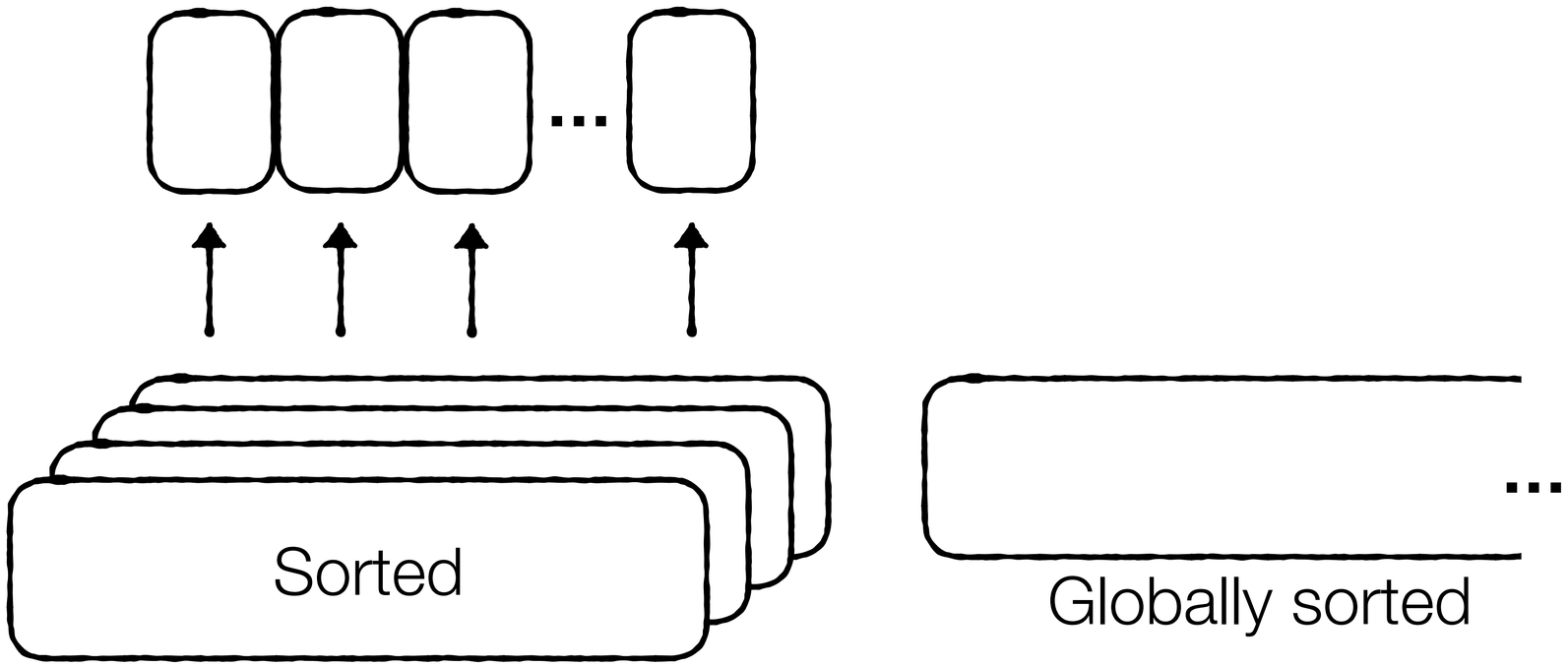
Avg. worst case O(N log N)

In-place

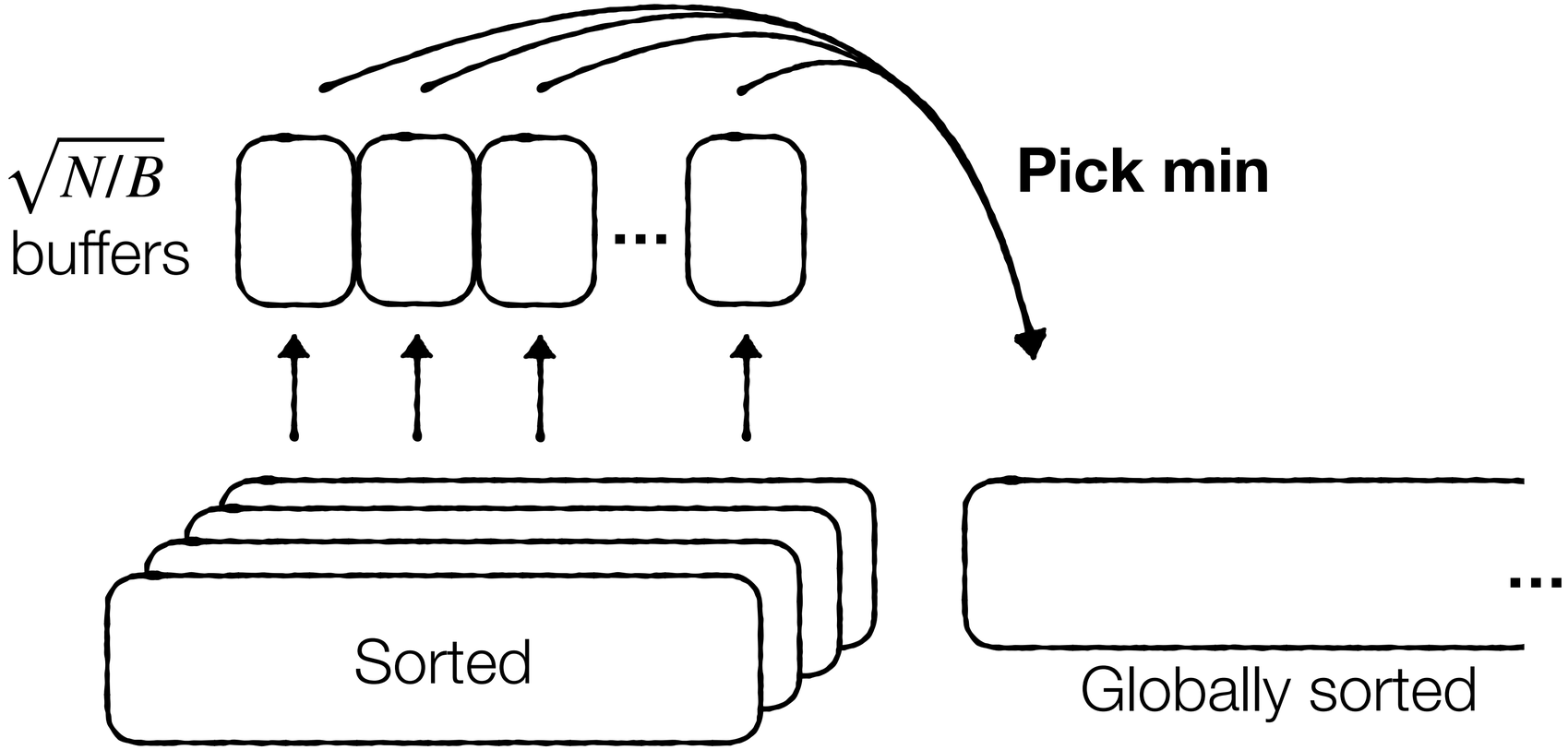
Sequential access

Now back to merging partitions in external merge-sort

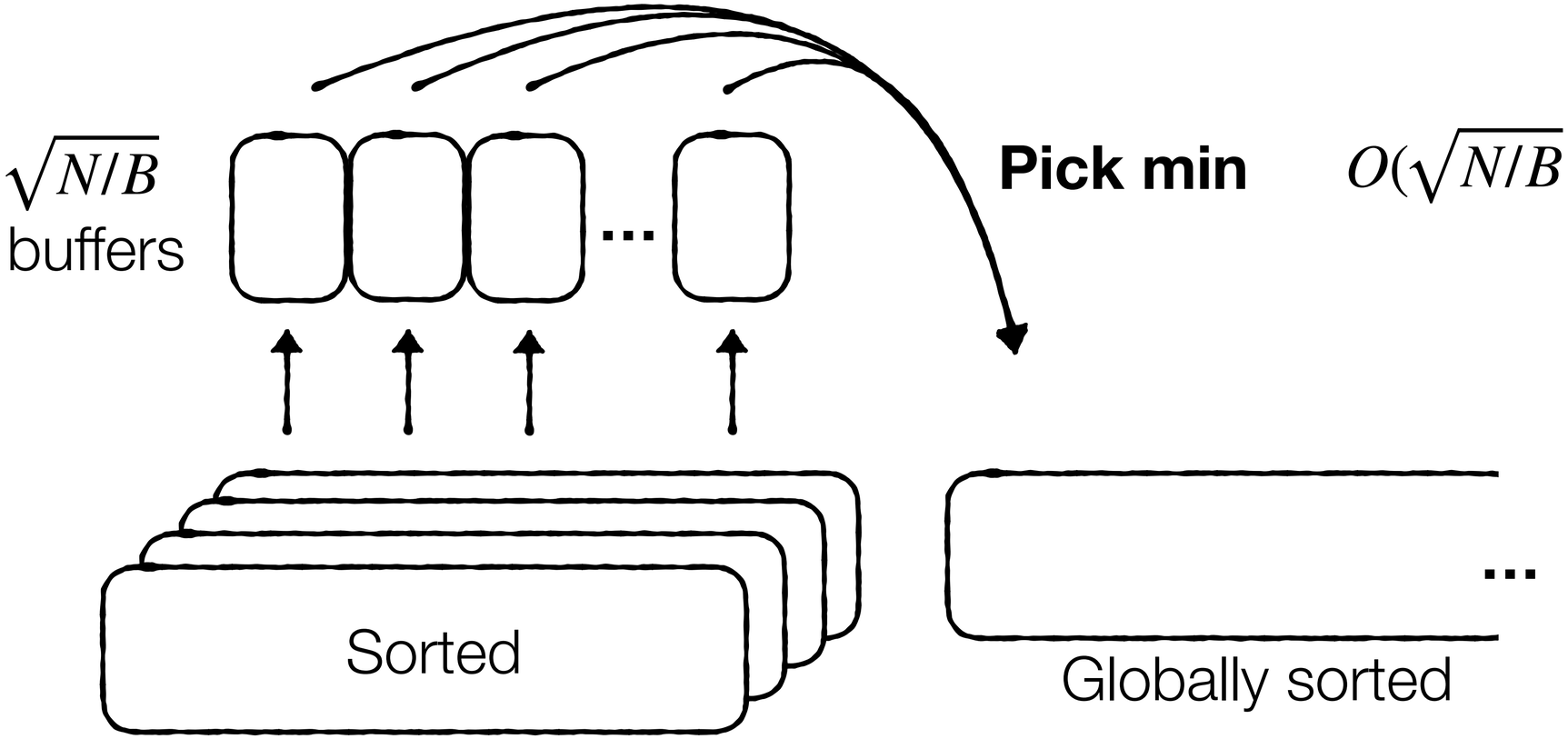
How to merge partitions?



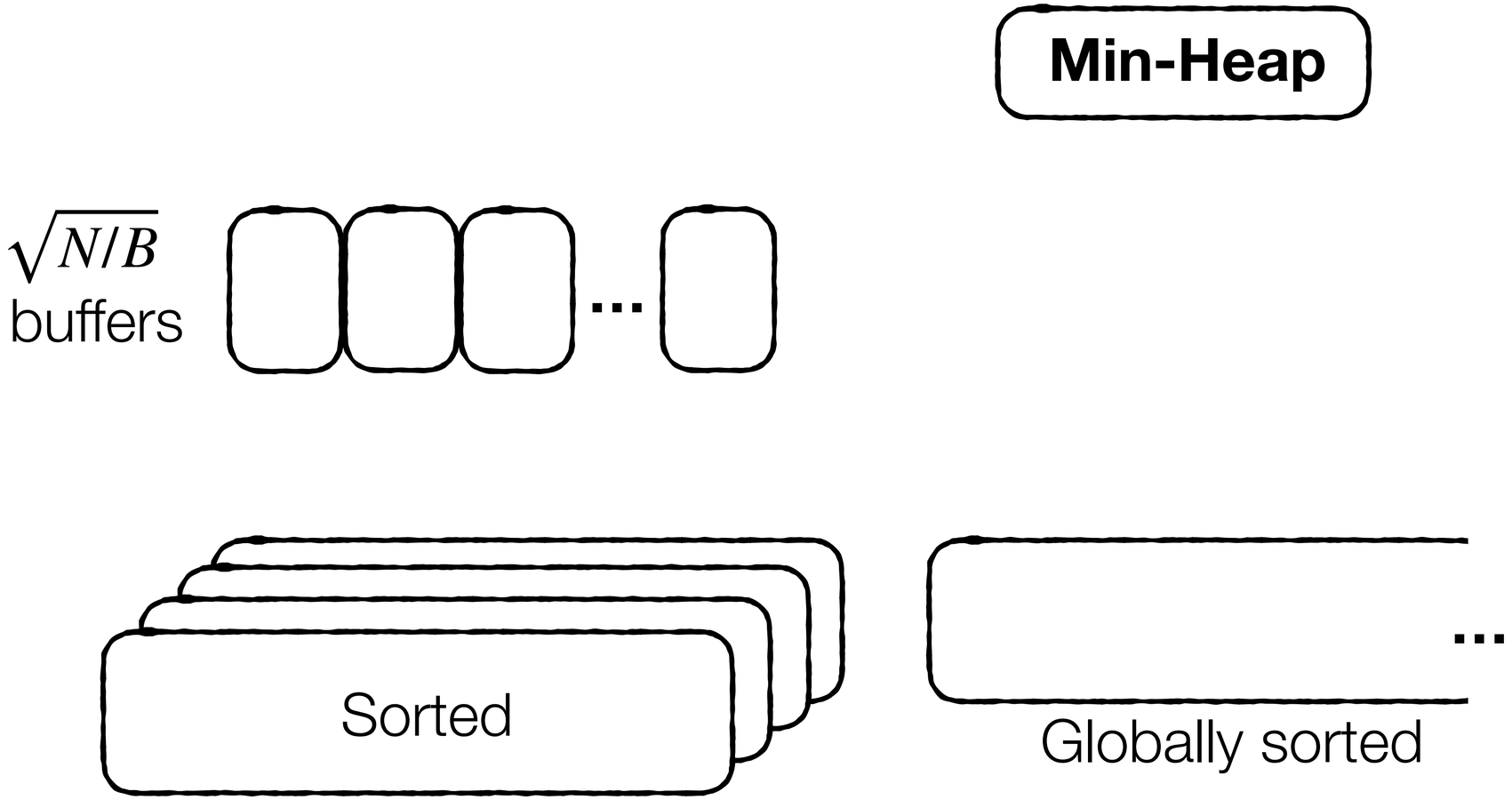
How to merge partitions?

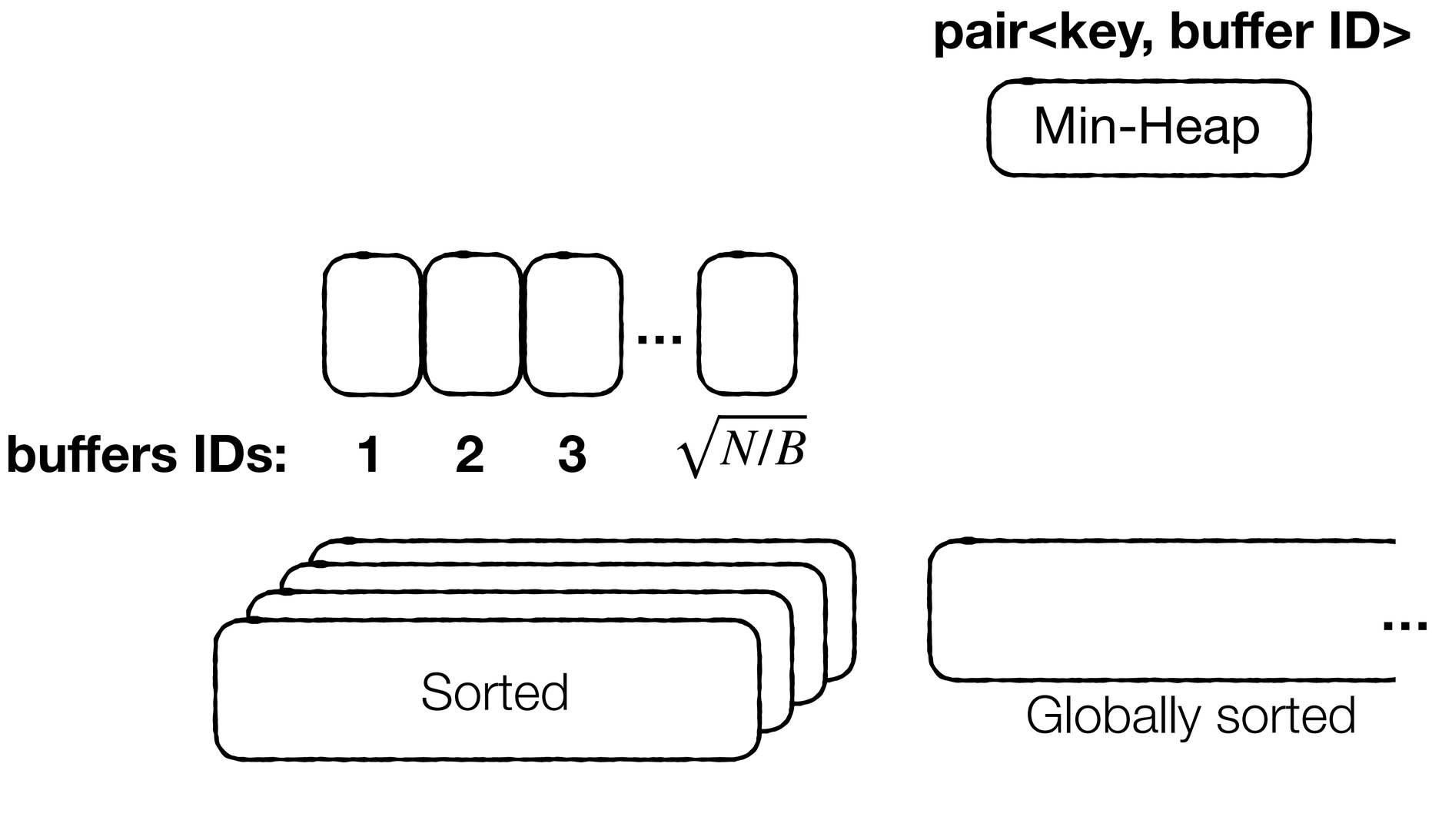


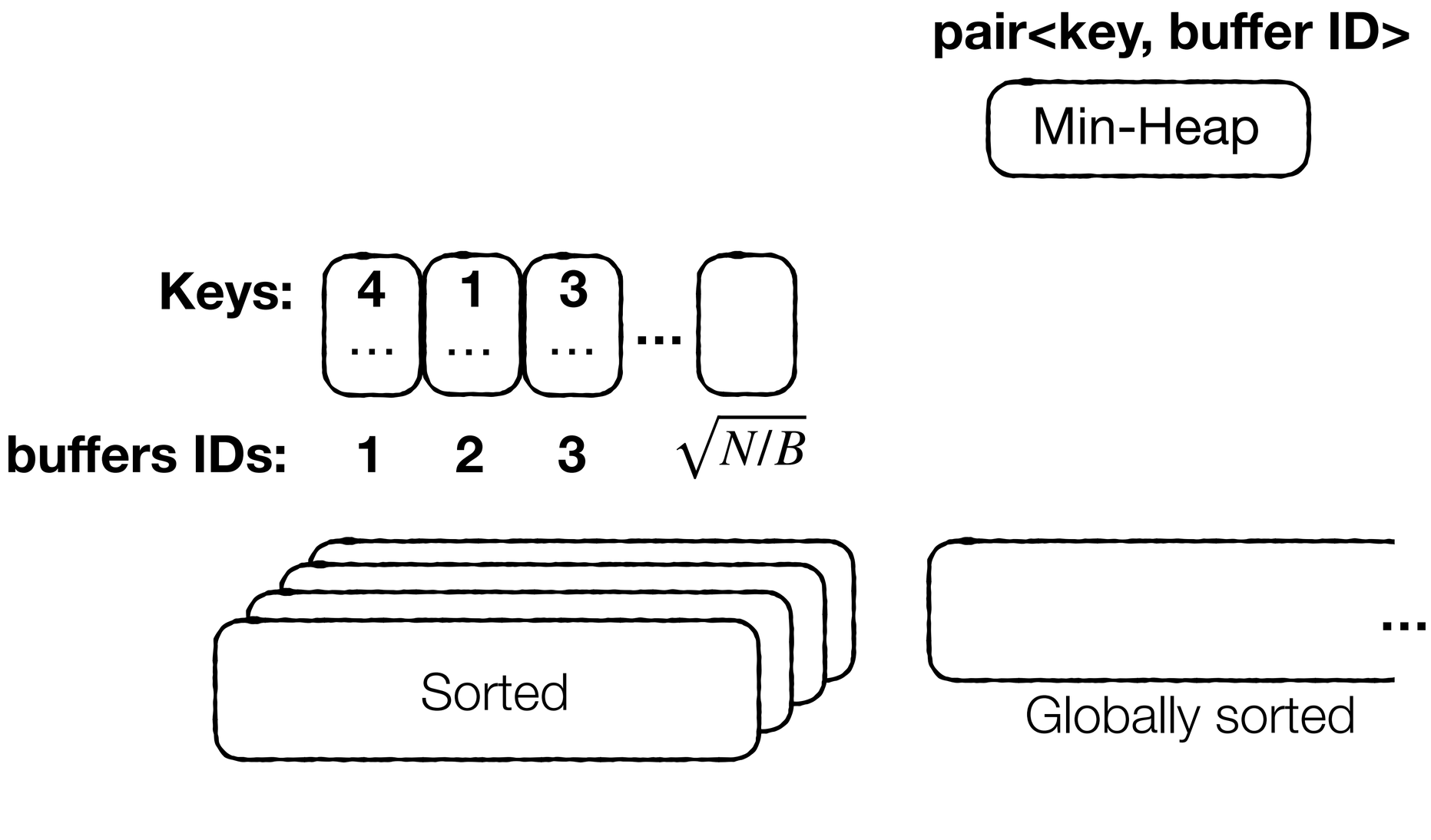
How to merge partitions?

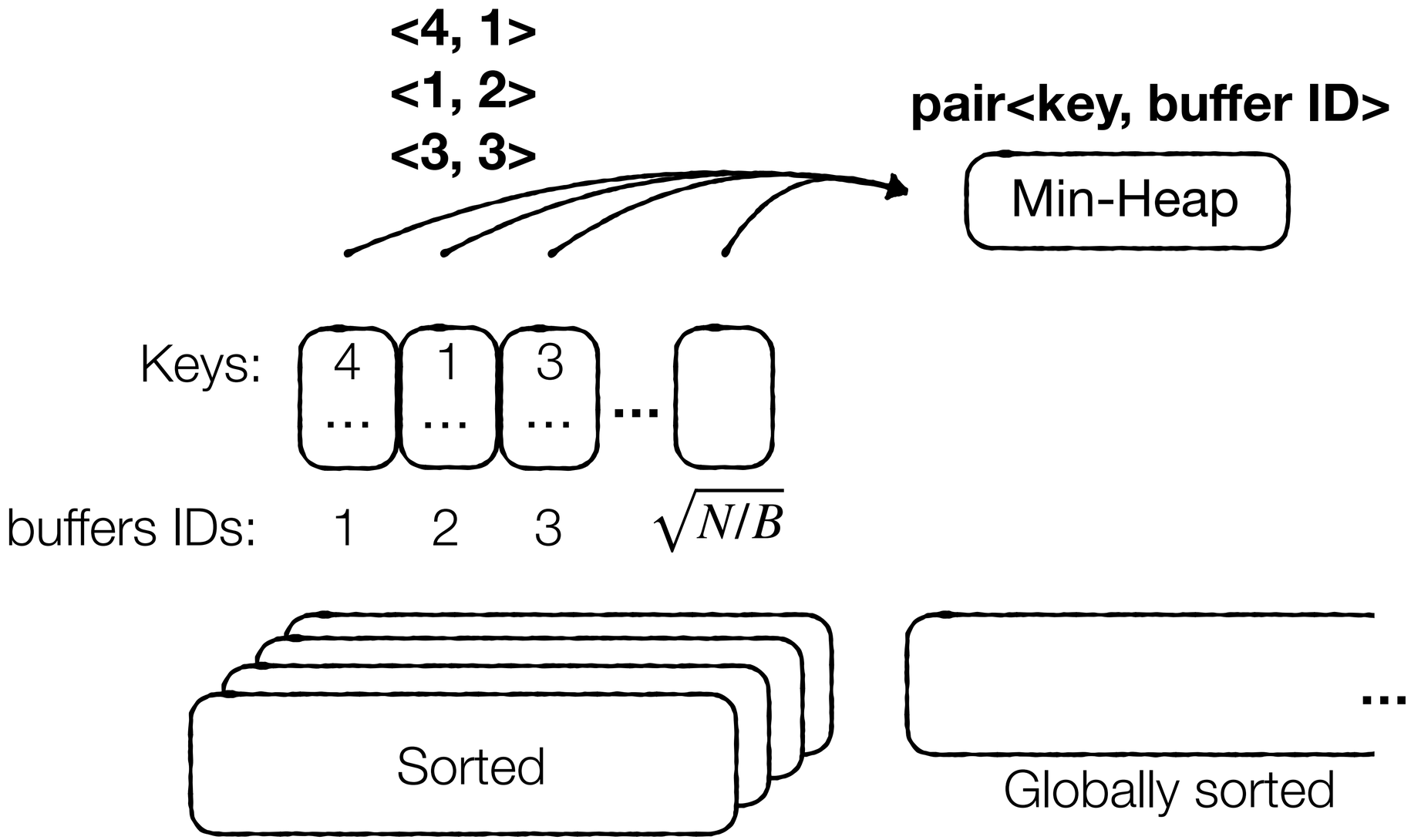


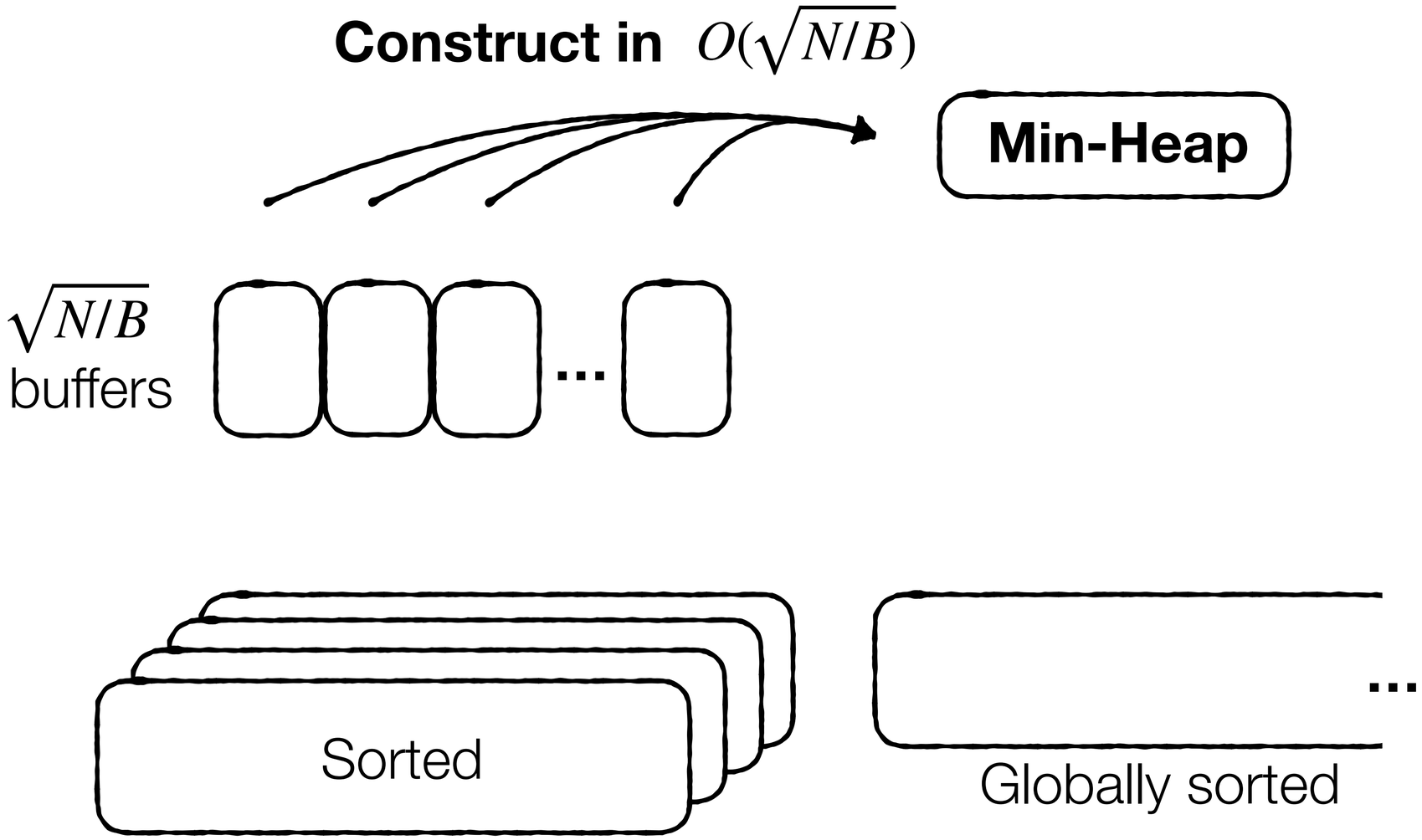
Pick min $O(\sqrt{N/B} \cdot N)$ **comparisons**

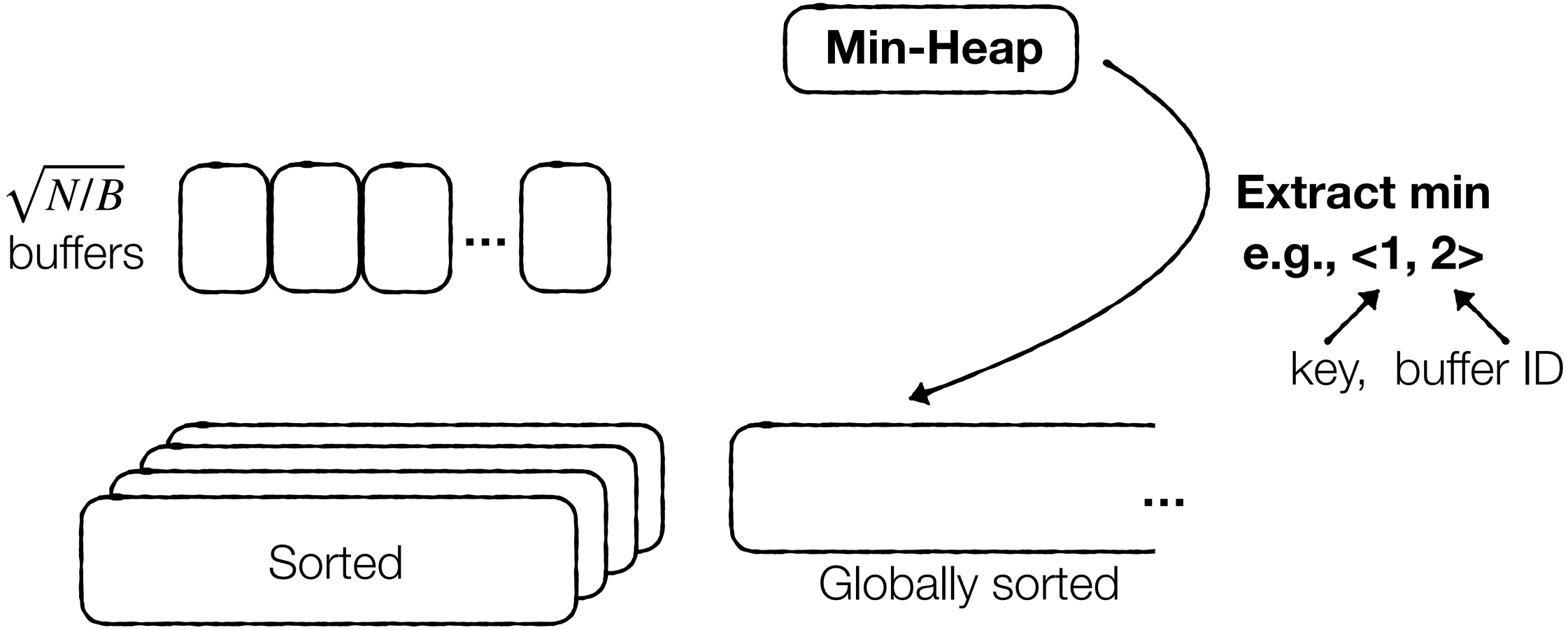

Min-Heap
$\sqrt{N/B}$
buffers
Sorted
Globally sorted

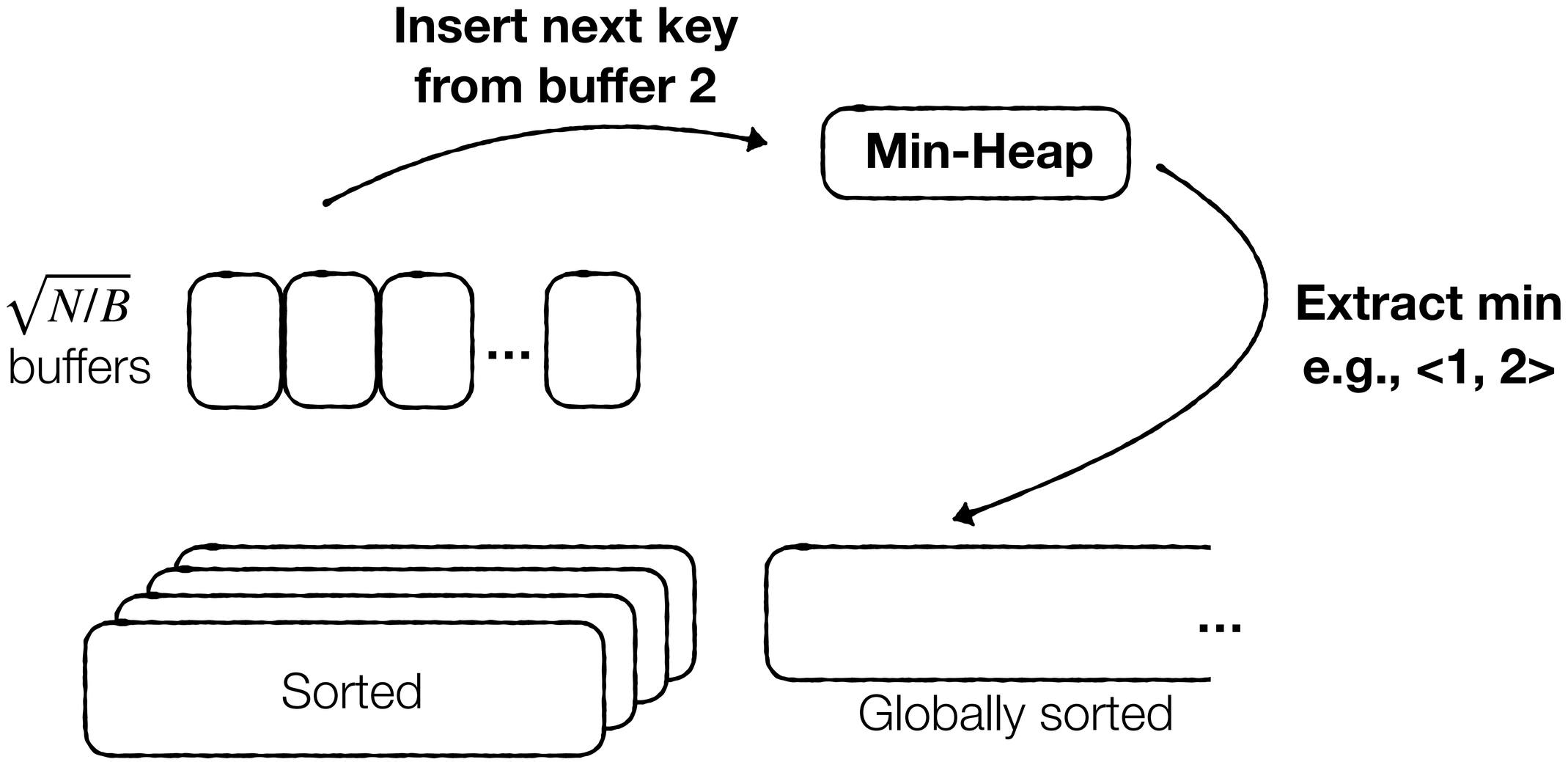

pair<key, buffer ID>
Min-Heap
...
buffers IDs: 1 2 3 $\sqrt{N/B}$
Sorted
...
Globally sorted

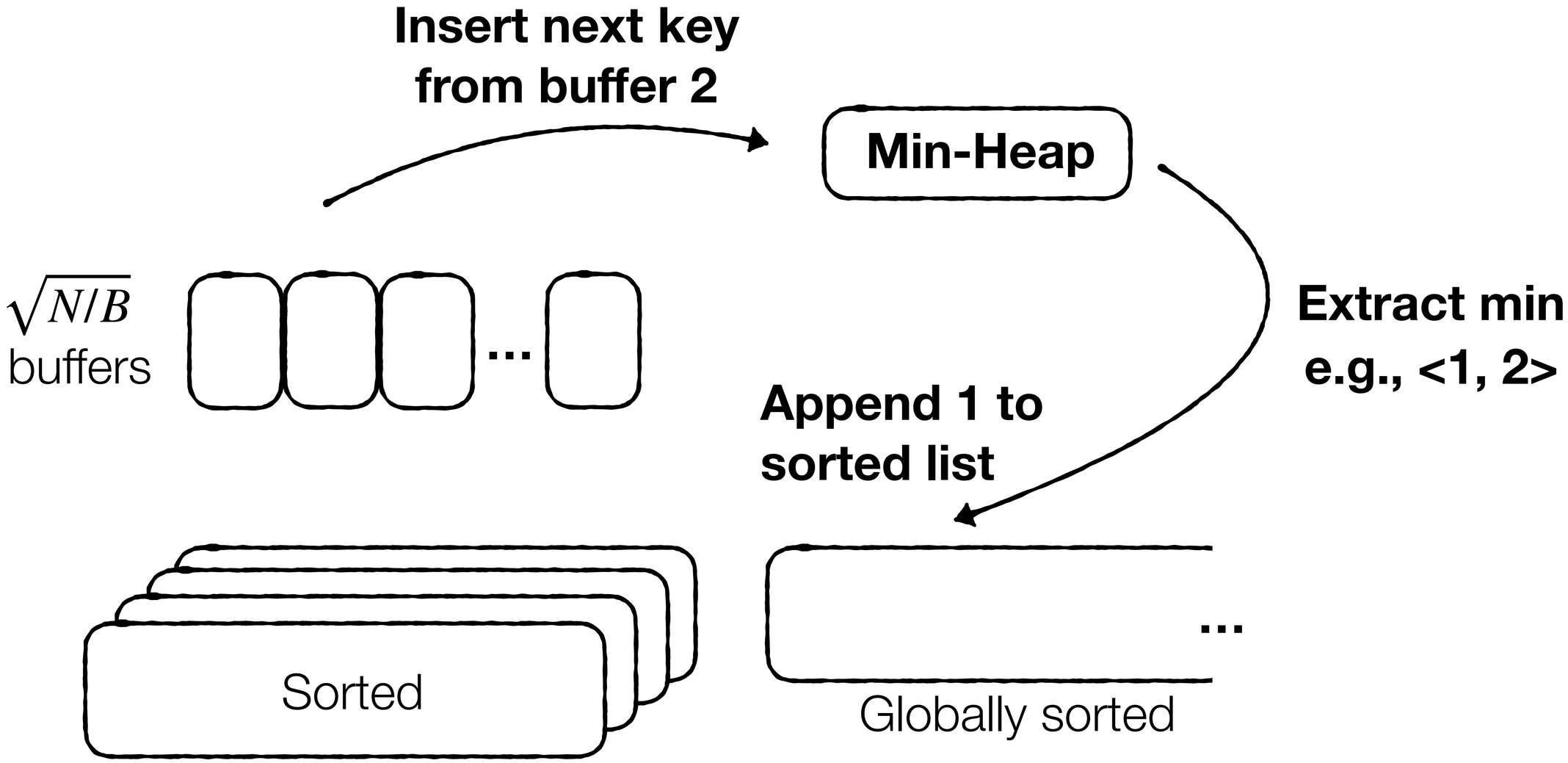

pair<key, buffer ID>
Min-Heap
Keys:
4
1
3
...
buffers IDs:
1
2
3
$\sqrt{N/B}$
Sorted
...
Globally sorted


<4, 1>
<1, 2>
<3, 3>
pair<key, buffer ID>
Min-Heap
Keys:
4
1
3
...
buffers IDs:
1
2
3
$\sqrt{N/B}$
Sorted
...
Globally sorted

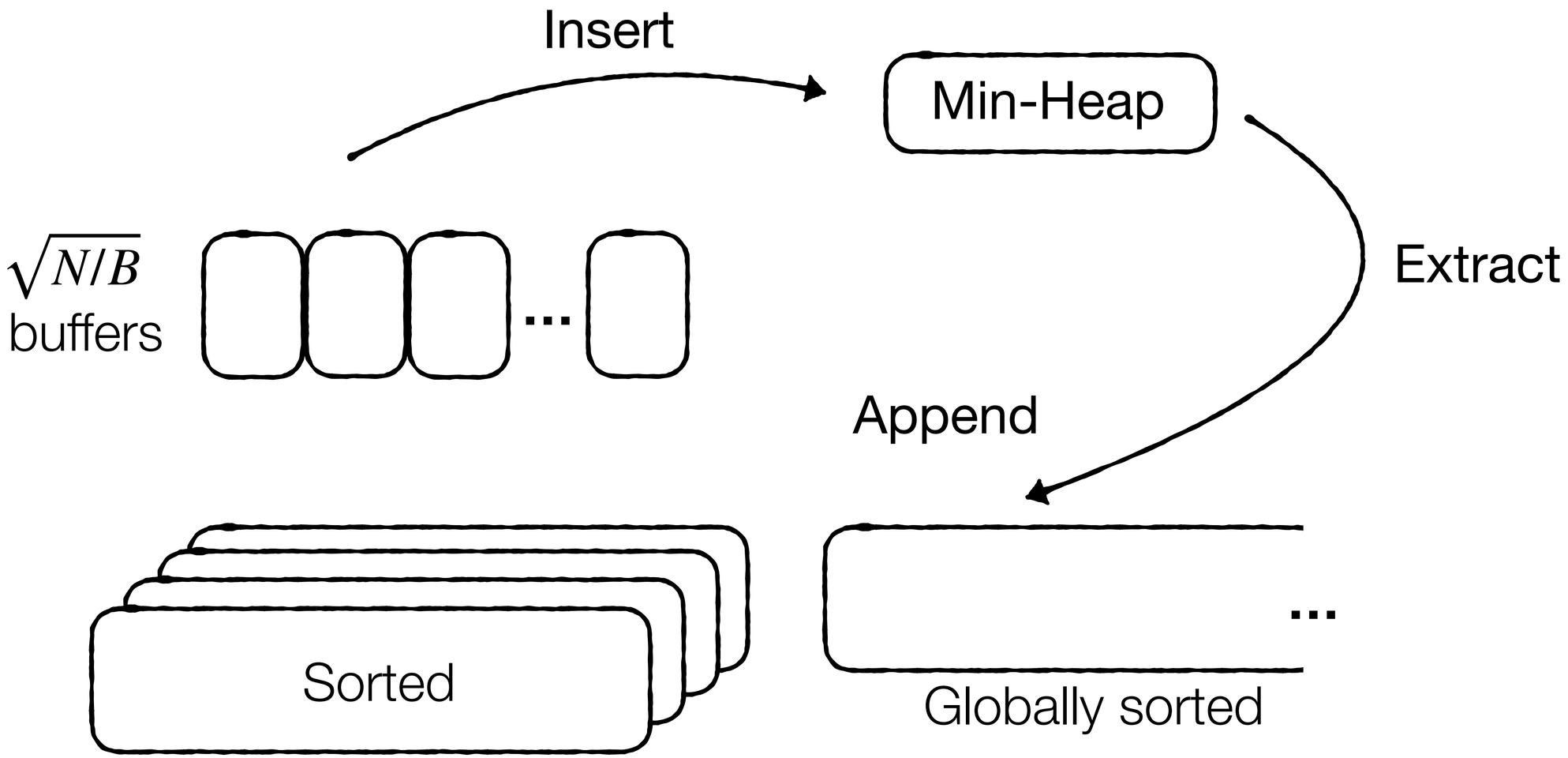

Construct in $O(\sqrt{N/B})$
Min-Heap
$\sqrt{N/B}$
buffers
...
Sorted
...
Globally sorted


Min-Heap
$\sqrt{N/B}$
buffers
Extract min
e.g., <1, 2>
key, buffer ID
Sorted
Globally sorted

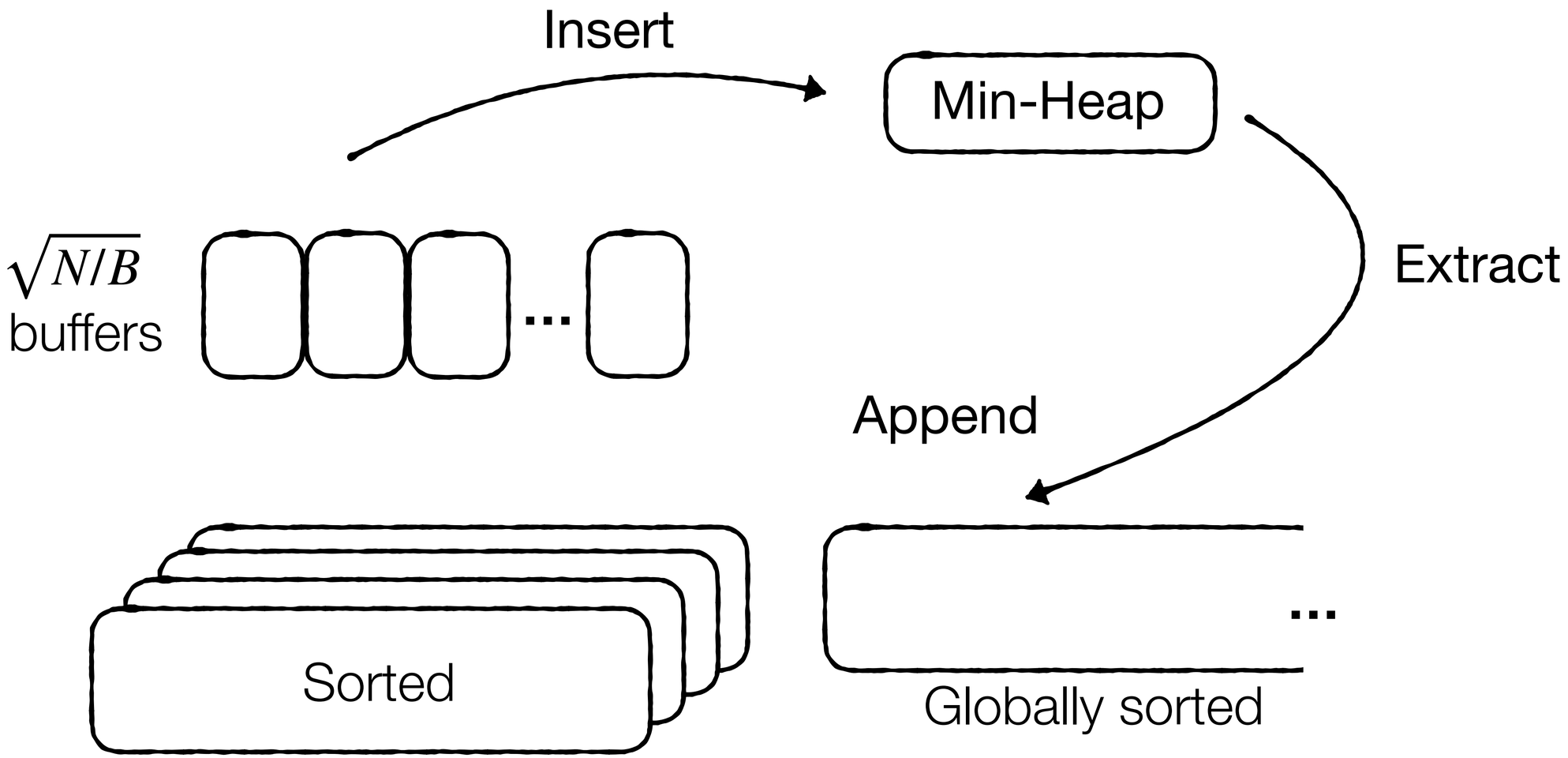

Insert next key
from buffer 2
Min-Heap
$\sqrt{N/B}$
buffers
Extract min
e.g., <1, 2>
Sorted
Globally sorted


Insert next key
from buffer 2
Min-Heap
$\sqrt{N/B}$
buffers
Extract min
e.g., <1, 2>
Append 1 to
sorted list
Sorted
Globally sorted

can do this with insert_and_extract

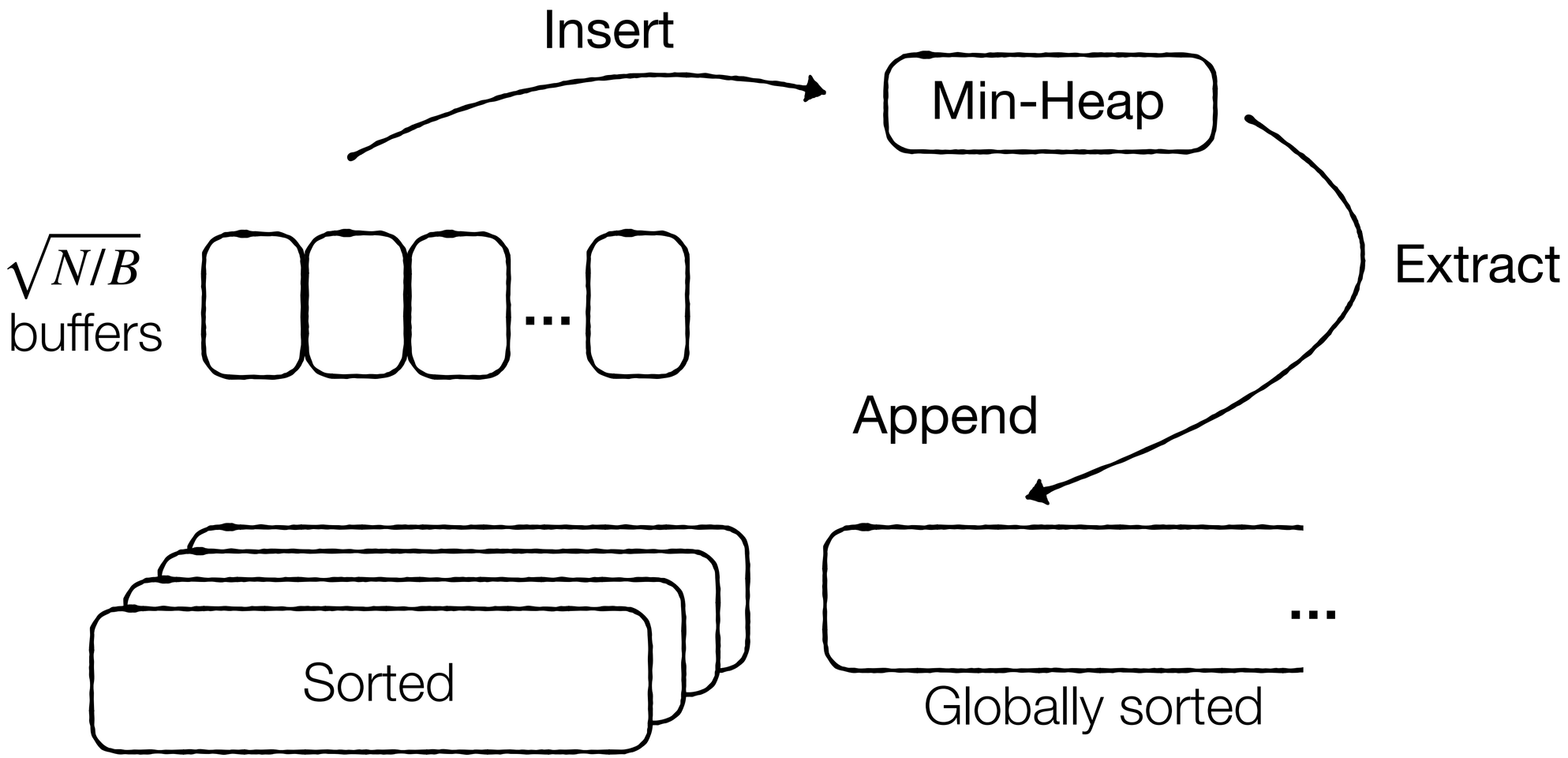


can do this with insert_and_extract: $O(log_2\sqrt{N/B})$ **per entry**



can do this with insert_and_extract: $O(log_2\sqrt{N/B})$

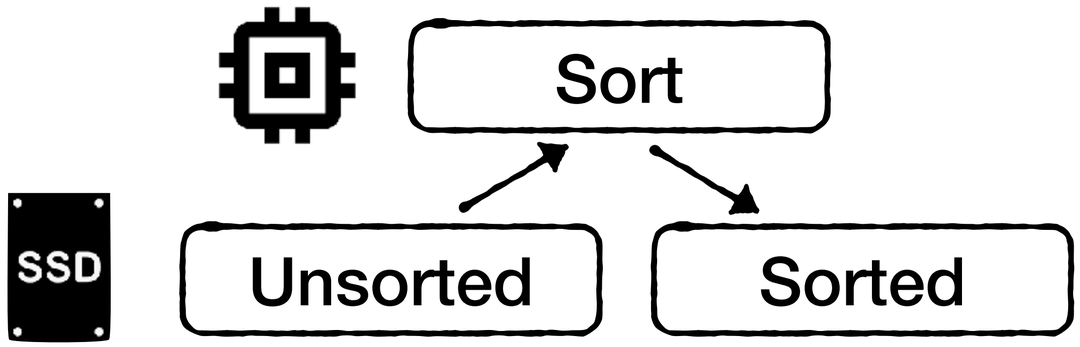
$O(N \cdot log_2\sqrt{N/B})$ **overall**



Analyzing CPU

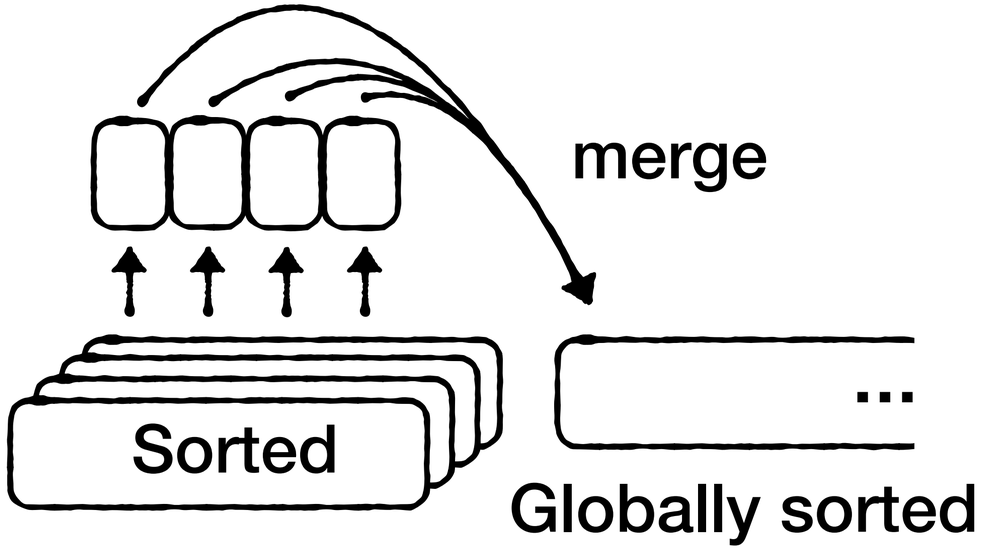
Partitioning Phase

$O(N \cdot log_2 M)$



Merging Phase

$O(N \cdot log_2 \sqrt{N/B})$

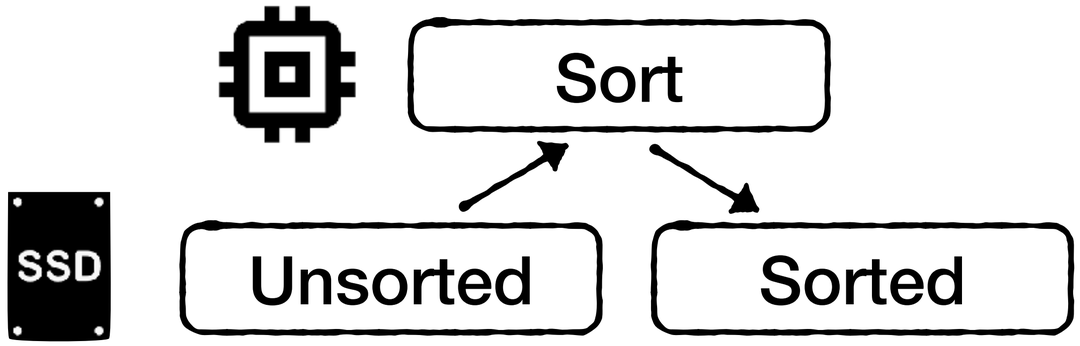


Analyzing CPU

Partitioning Phase

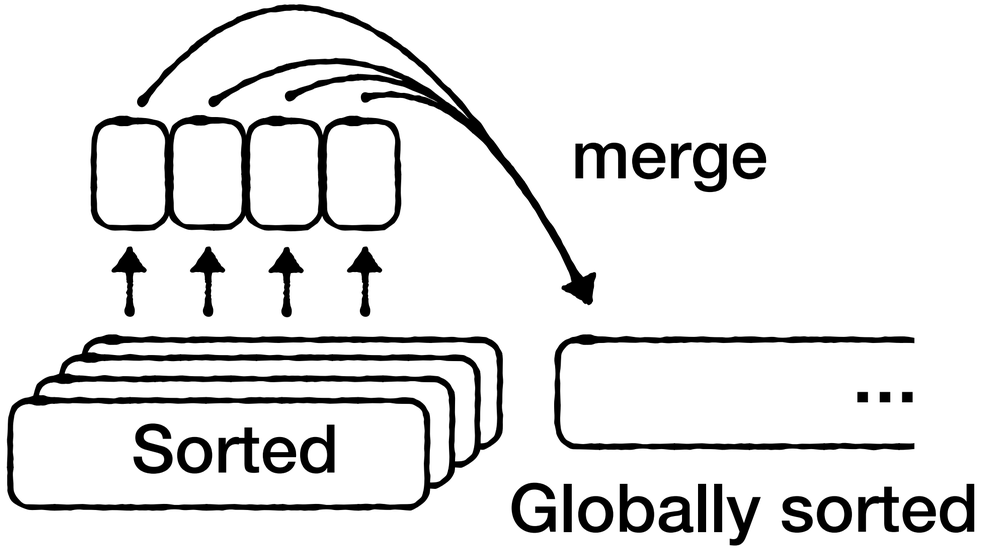
$$O(N \cdot log_2 M)$$

$$= O(N \cdot log_2 \sqrt{N \cdot B})$$



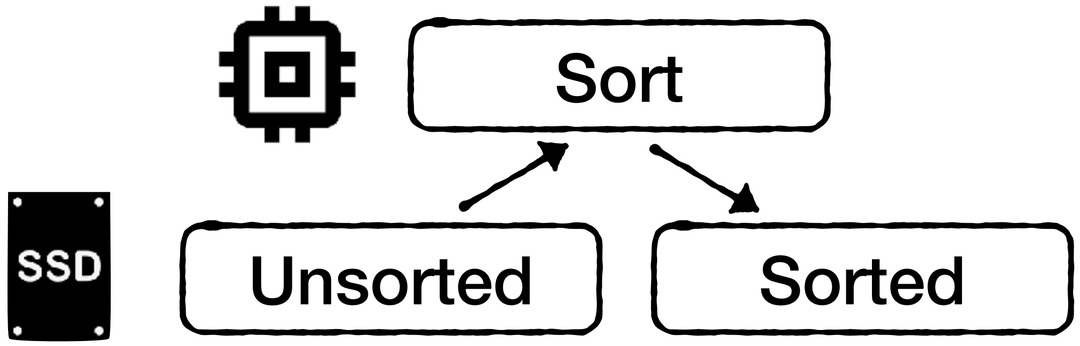
Merging Phase

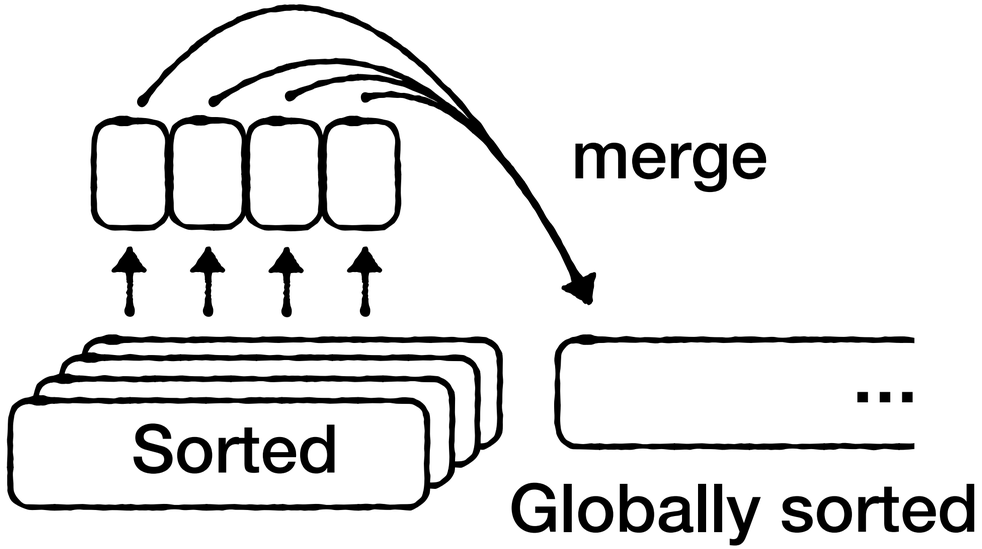
$$O(N \cdot log_2 \sqrt{N/B})$$



Analyzing CPU

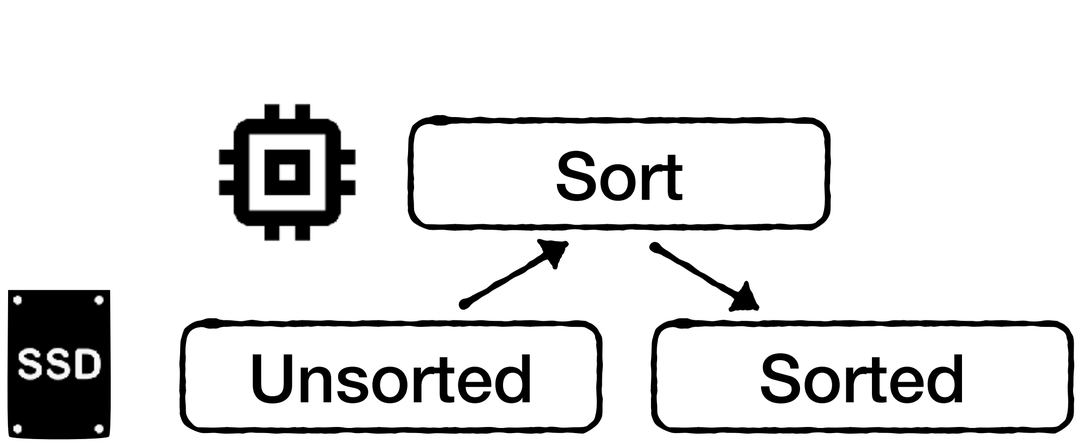
Partitioning Phase		Merging Phase		**Total cost**
$O(N \cdot log_2\sqrt{N \cdot B})$	+	$O(N \cdot log_2\sqrt{N/B})$	=	$O(N \cdot log_2 N)$

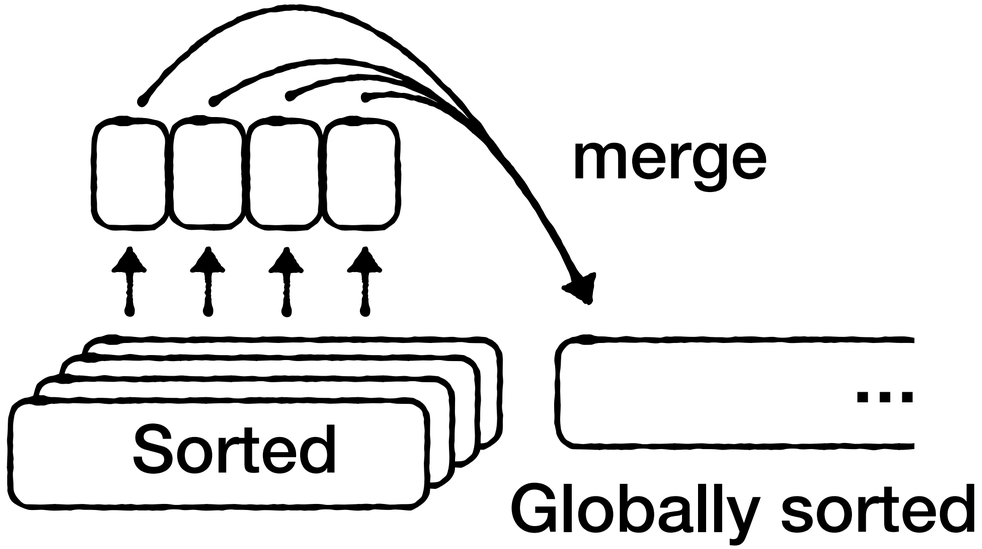




Same as in-memory algorithms :)

$$O(N \cdot log_2 N)$$

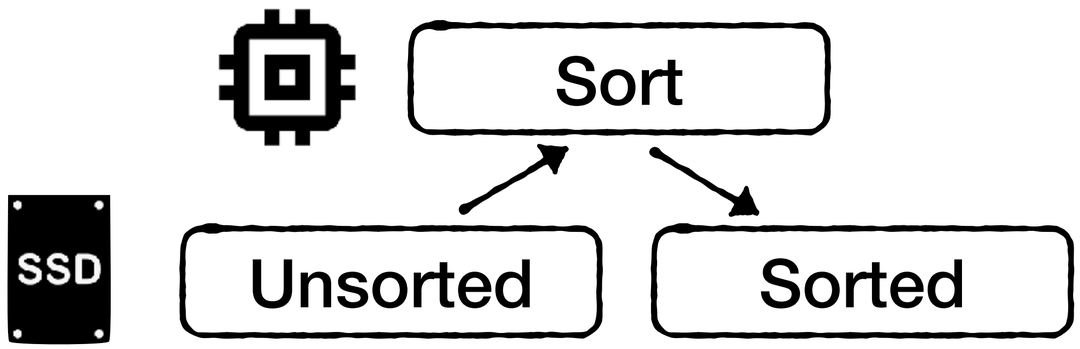


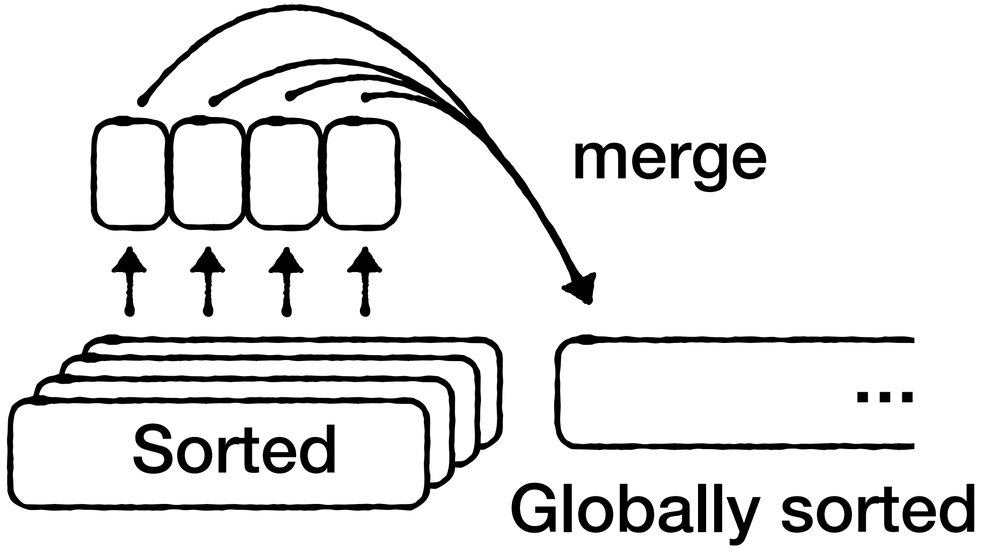


Overall costs

$O(N \cdot log_2 N)$ **CPU**

$O(N/B \cdot log_{M/B}(N/B))$ **I/O**

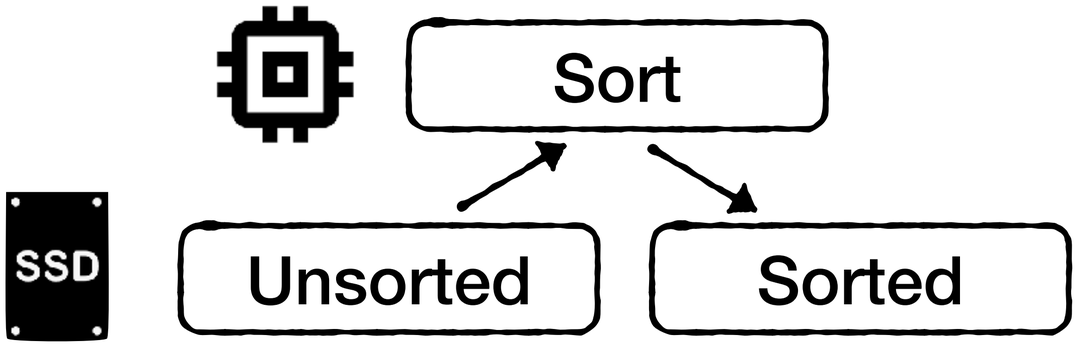


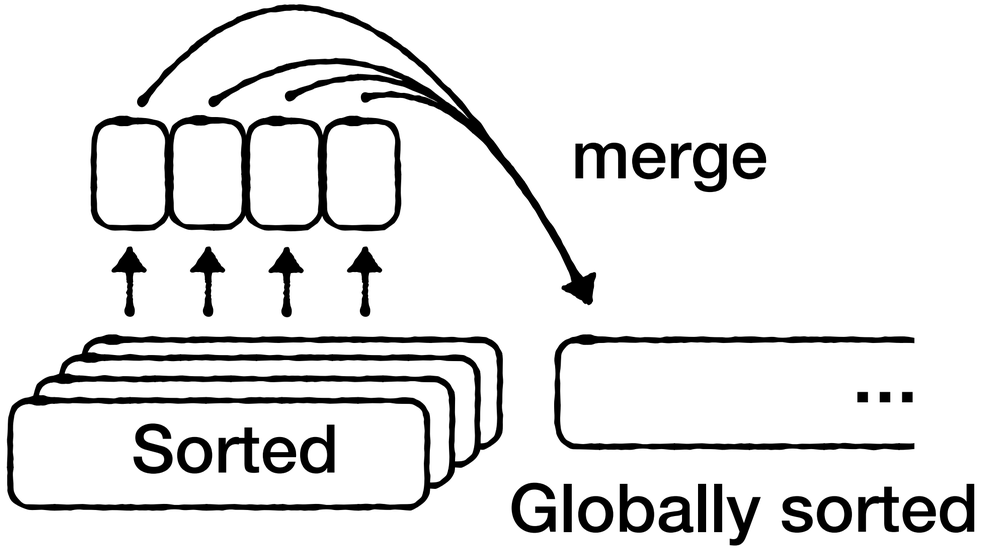


Overall costs

$O(N \cdot log_2 N)$ **CPU**

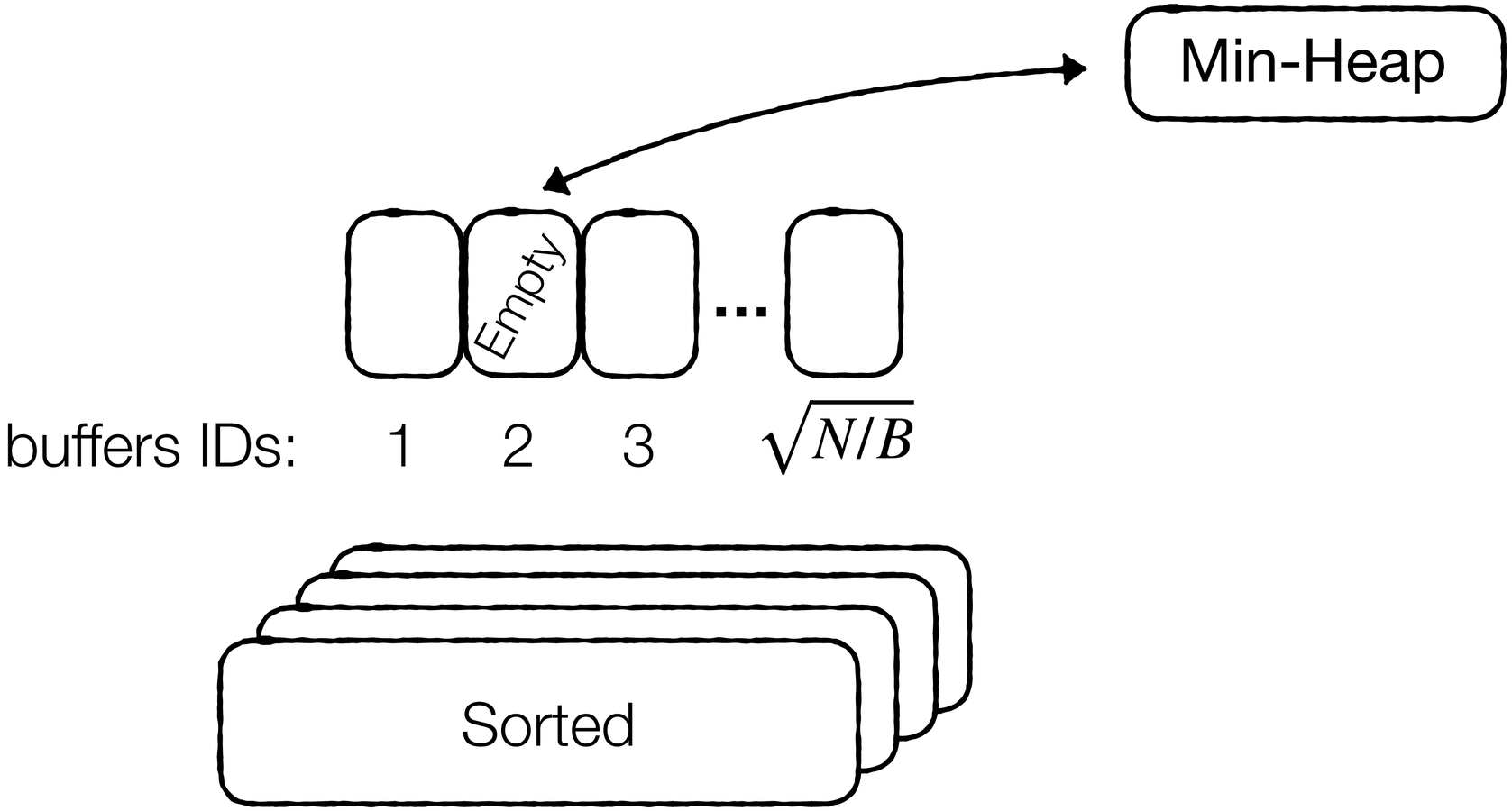
$O(N/B \cdot log_{M/B}(N/B))$ **I/O** **or** $O(N/B)$ **when** $M > \sqrt{N \cdot B}$



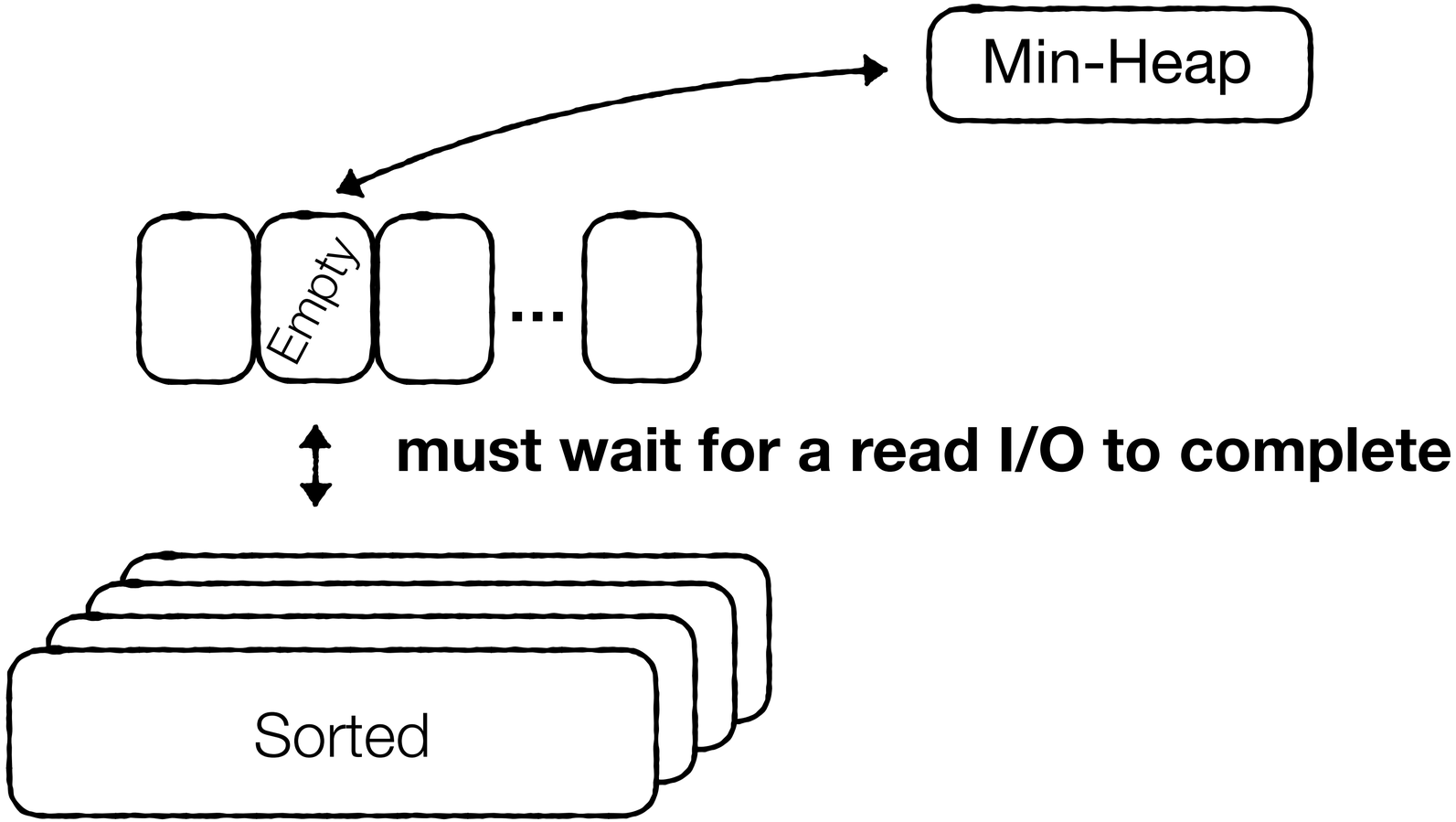


And now: TA office hours

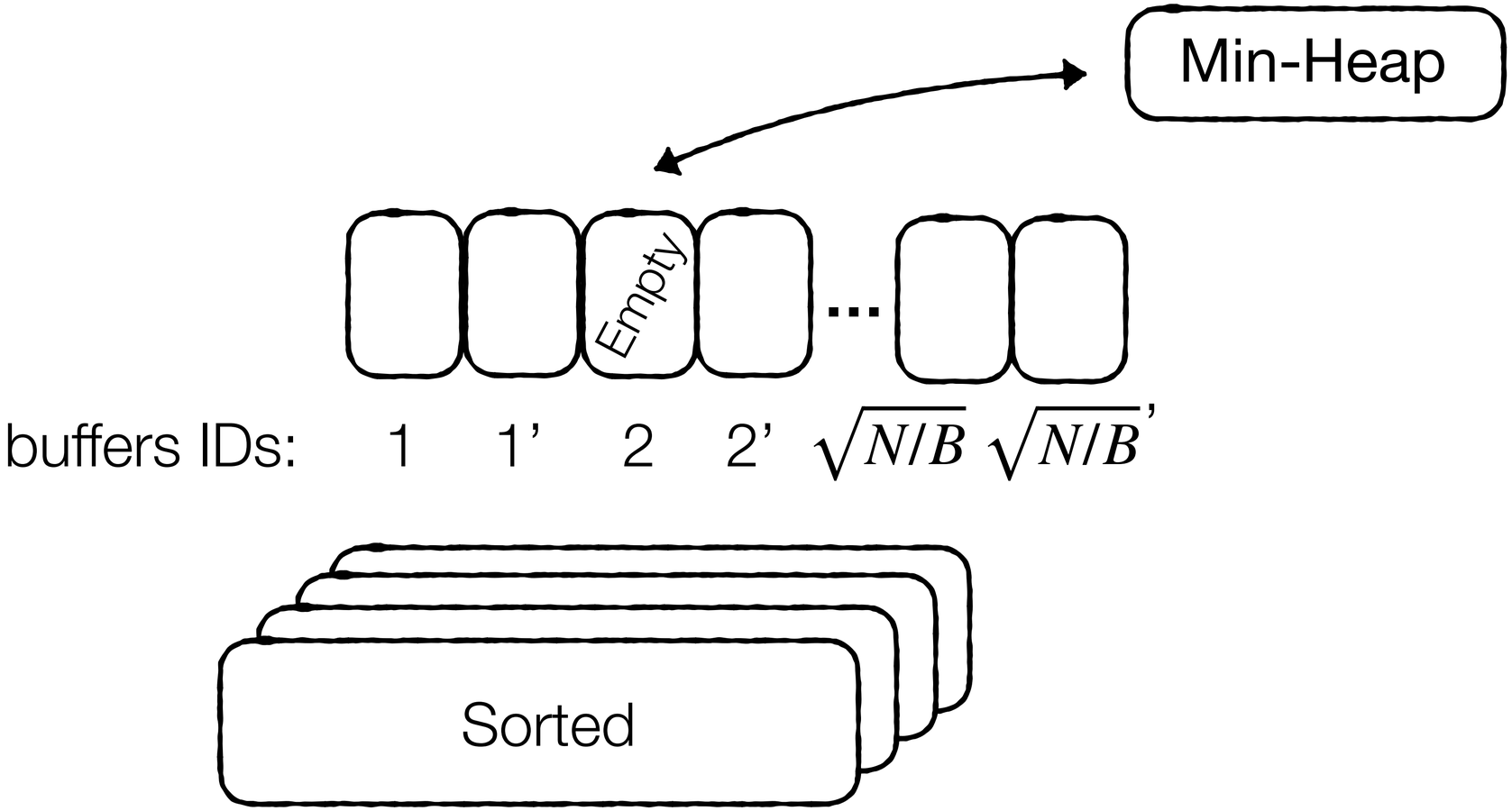
Suppose we need next min entry from buffer 2 but it is empty.



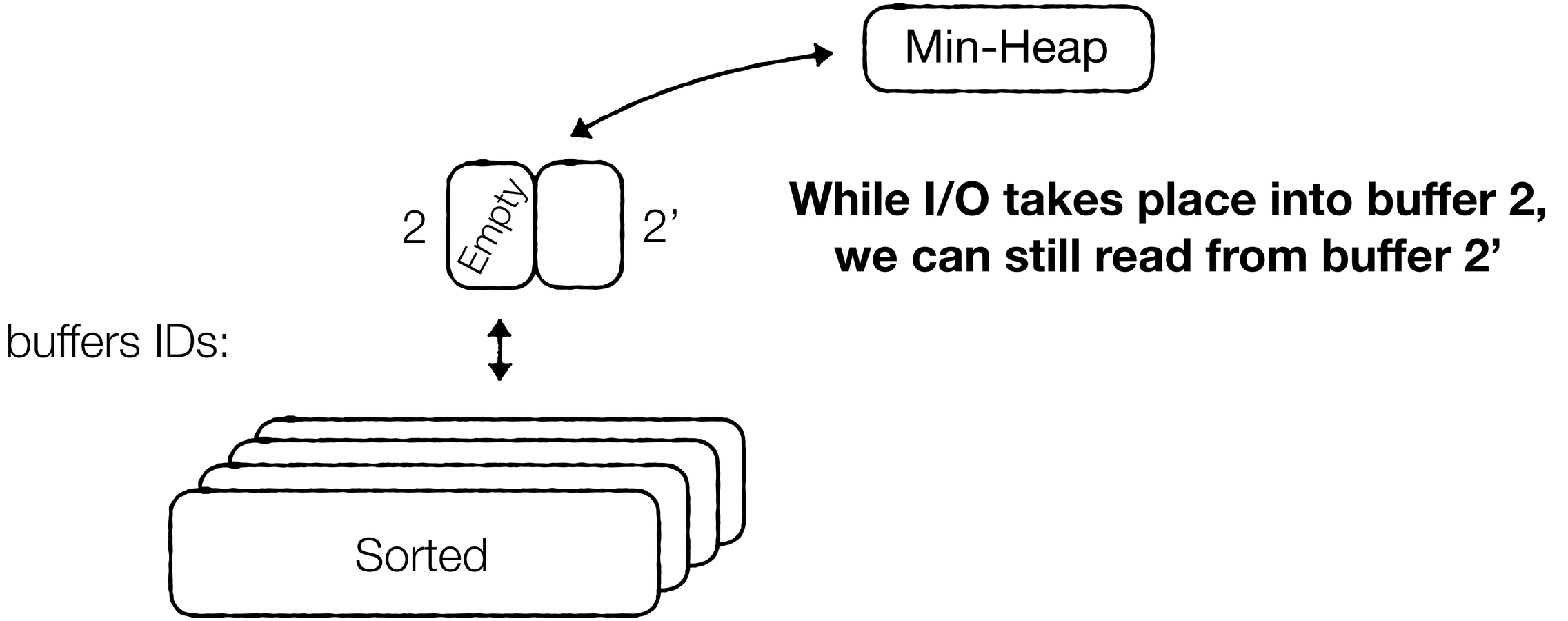
Suppose we need next min entry from buffer 2 but it is empty.



Double buffering: load one additional buffer preemptively for each partition before the first buffer empties



Double buffering: load one additional buffer preemptively for each partition before the first buffer empties



While I/O takes place into buffer 2, we can still read from buffer 2'

Larger though fewer buffers: more groups, so potentially more iterations, but each I/O reads more data