External Sorting

Chapter 13.

Why Sort?

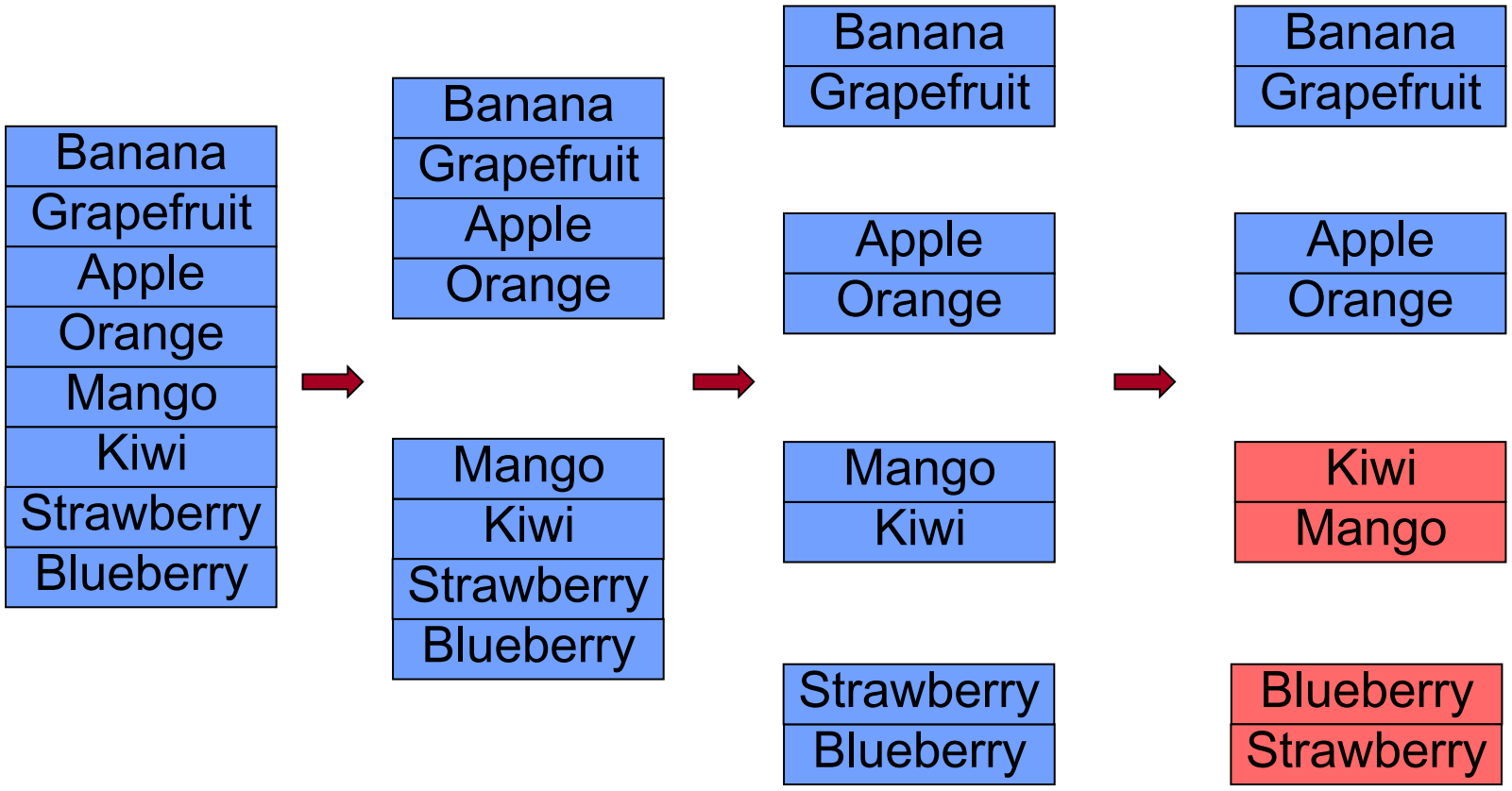
- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing *gpa* order
- First step in *bulk loading* B+ tree index.
- Useful for eliminating *duplicates* (Why?)
- Useful for summarizing groups of tuples
- *Sort-merge* join algorithm involves sorting.

- Problem: sort 100Gb of data with 1Gb of RAM.
  - why not virtual memory?

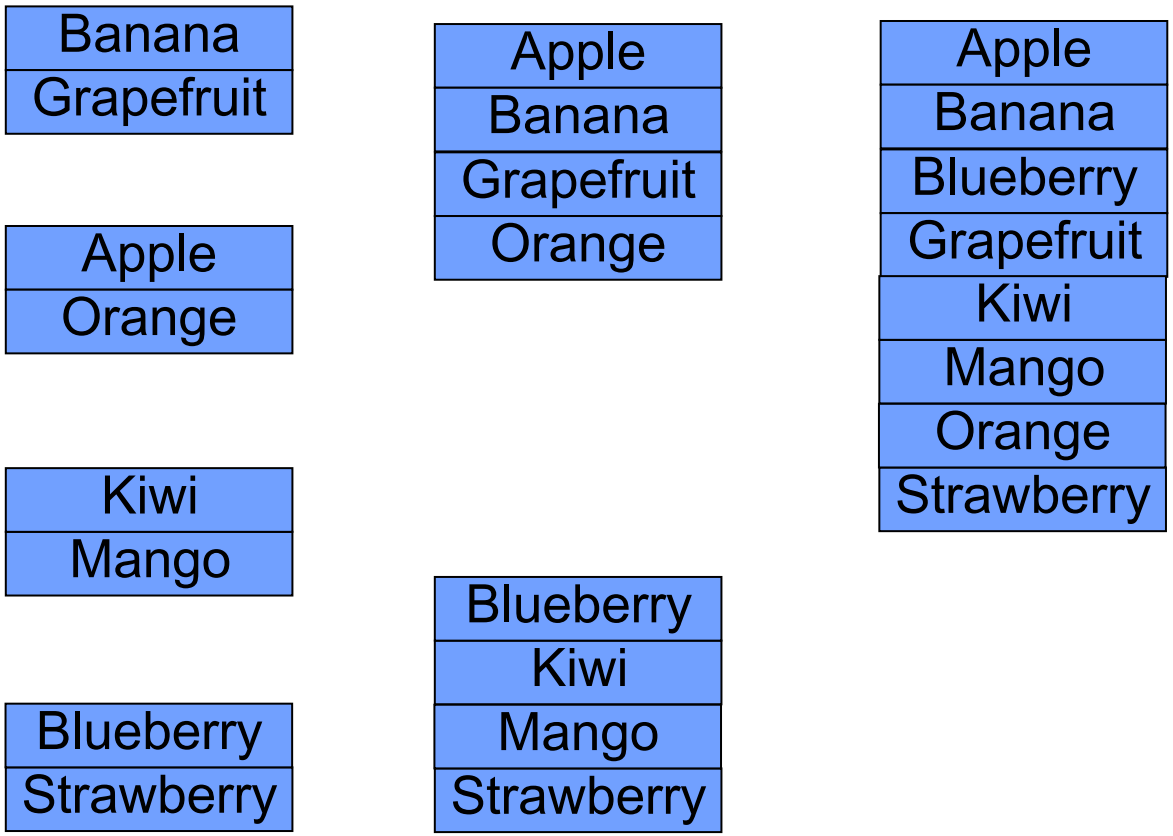
So?

- Don’t we know how to sort?
  - Quicksort
  - Mergesort
  - Heapsort
  - Selection sort
  - Insertion sort
  - Radix sort
  - Bubble sort
  - Etc.
- Why don’t these work for databases?

Example: merge sort



Example: merge sort

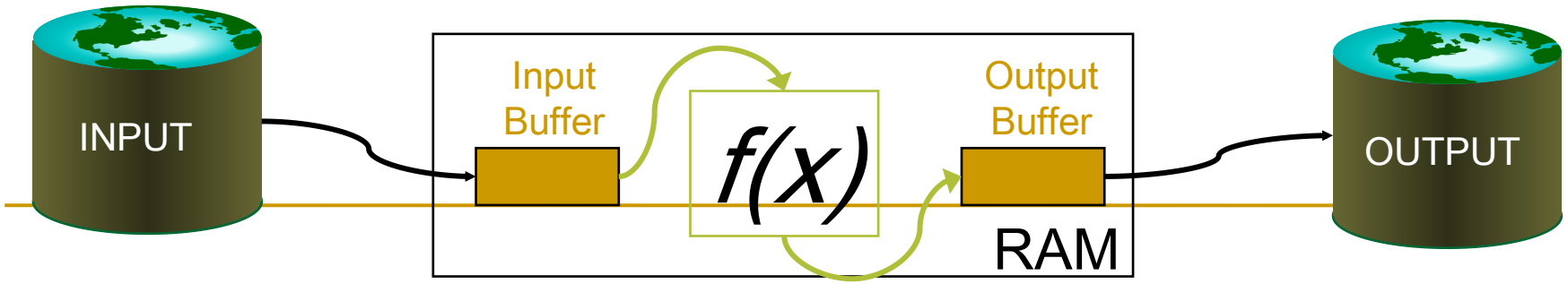


Isn't that good enough?

- Consider a file with N records
- Merge sort is O(N lg N) comparisons
- We want to minimize disk I/Os
  - Don't want to go to disk O(N lg N) times!
- Key insight: sort based on pages, not records
  - Read whole pages into RAM, not individual records
  - Do some in-memory processing
  - Write processed blocks out to disk
  - Repeat

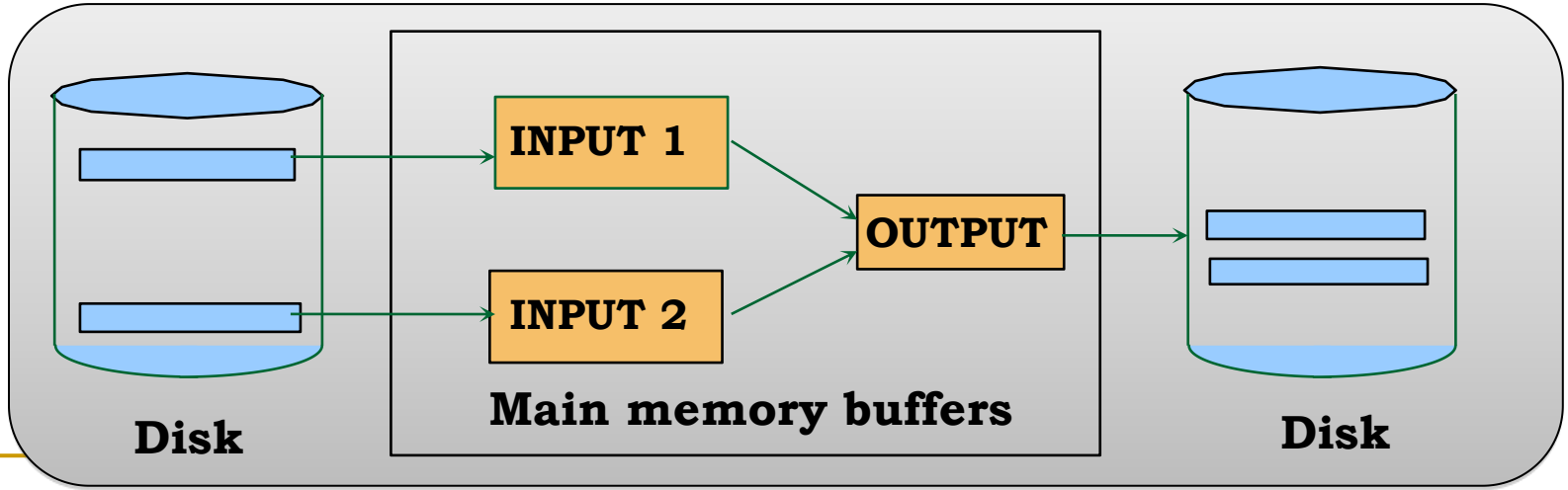
Streaming Data Through RAM

- An important method for sorting & other DB operations
- Simple case:
  - Compute *f(x)* for each record, write out the result
  - Read a page from INPUT to Input Buffer
  - Write *f(x)* for each item to Output Buffer
  - When Input Buffer is consumed, read another page
  - When Output Buffer fills, write it to OUTPUT
- Reads and Writes are *not* coordinated
  - E.g., if f() is Compress(), you read many pages per write.
  - E.g., if f() is DeCompress(), you write many pages per read.



2-Way Sort: Requires 3 Buffers

- Pass 0: Read a page, sort it, write it.
  - only one buffer page is used (as in previous slide)
- Pass 1, 2, 3, …, etc.:
  - requires 3 buffer pages
  - merge pairs of runs into runs twice as long
  - three buffer pages used.



Merging Runs

Two-Way External Merge Sort

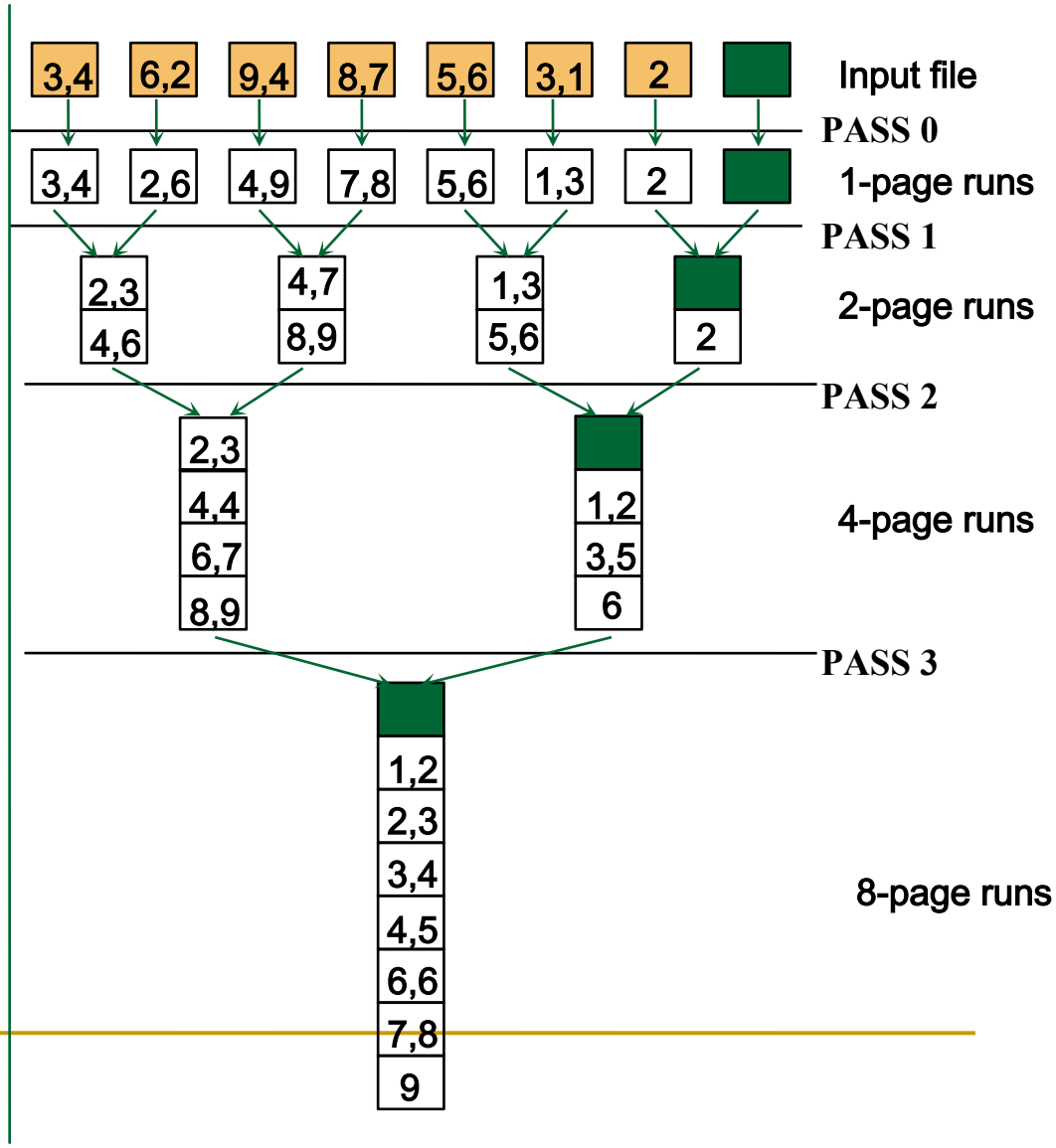
- Each pass we read + write each page in file.
- N pages in the file => the number of passes

$$= \lceil \log_2 N \rceil + 1$$

- So total cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

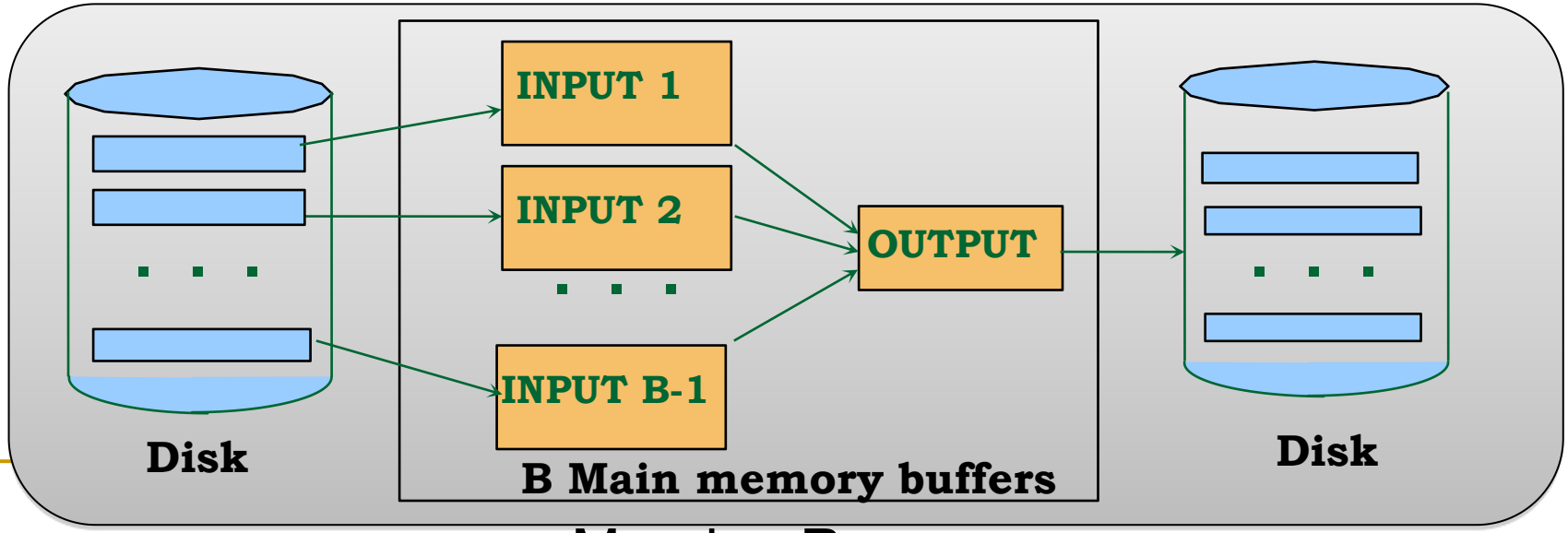
- *Idea: Divide and conquer:* sort subfiles and merge



General External Merge Sort

✉ *More than 3 buffer pages. How can we utilize them?*

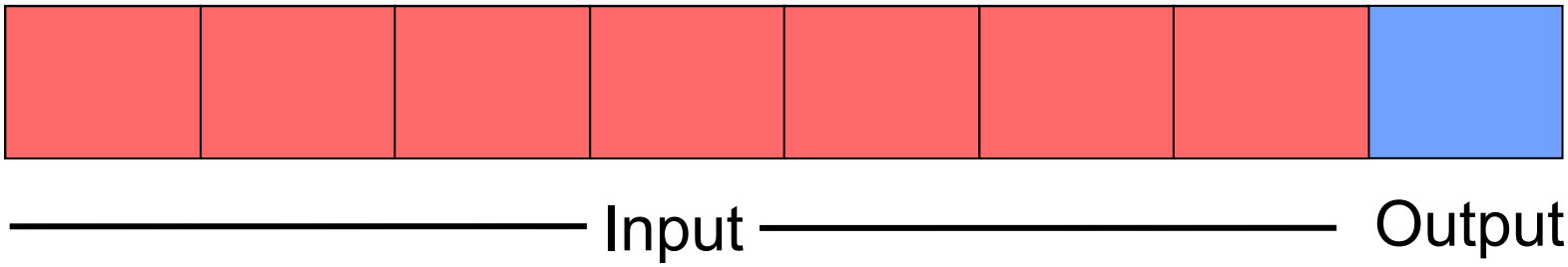
- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: use *B* buffer pages. Produce $\lceil N / B \rceil$ sorted runs of *B* pages each.
  - Pass 1, 2, …, etc.: merge *B-1* runs.



Merging Runs

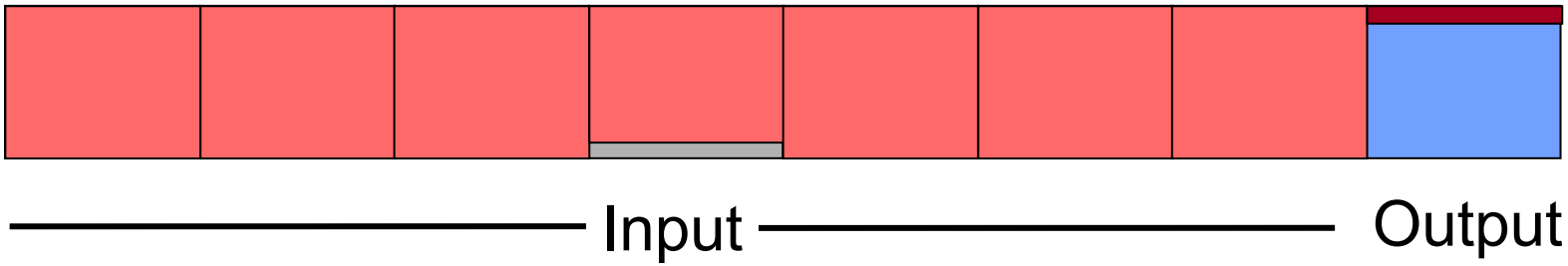
Example

七路红军，选先头部队

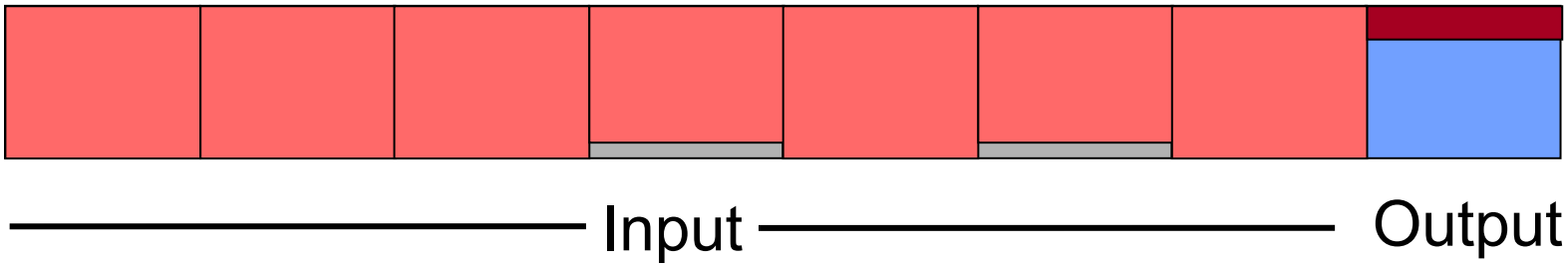


Example

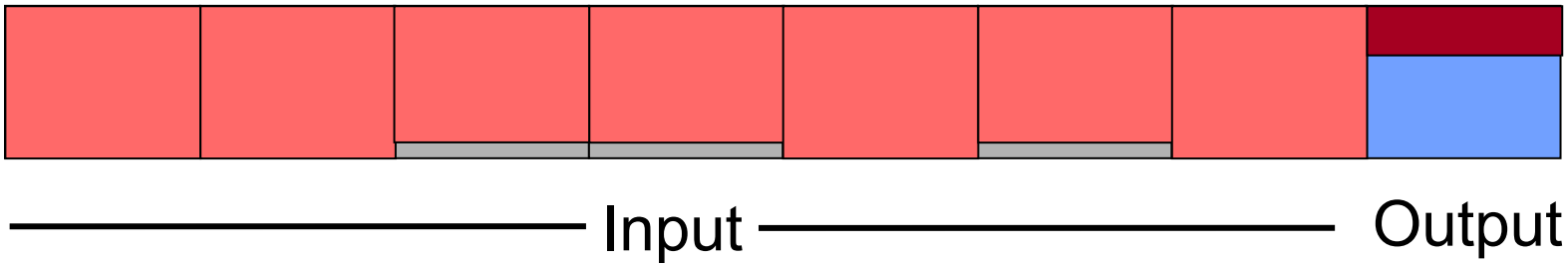
从7路排头士兵的，选最强壮的



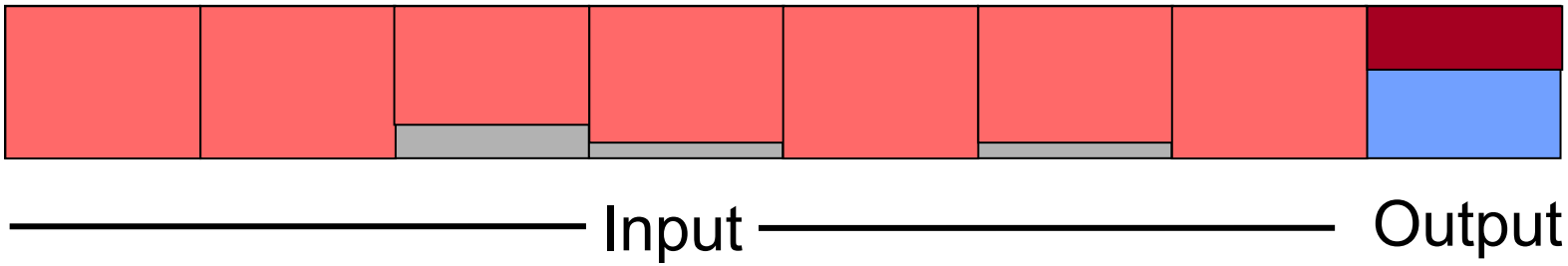
Example



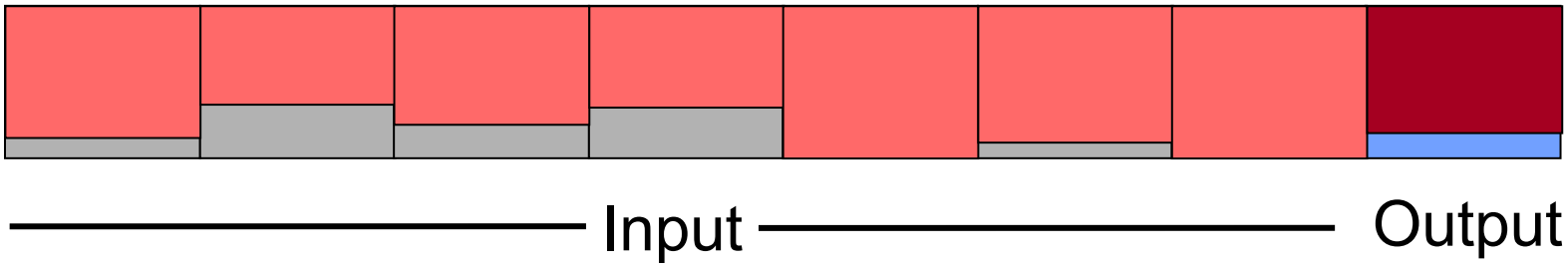
Example



Example

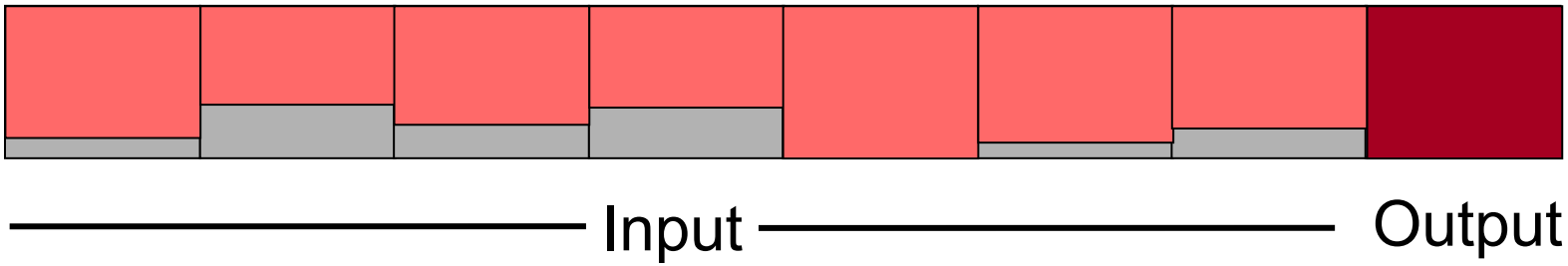


Example

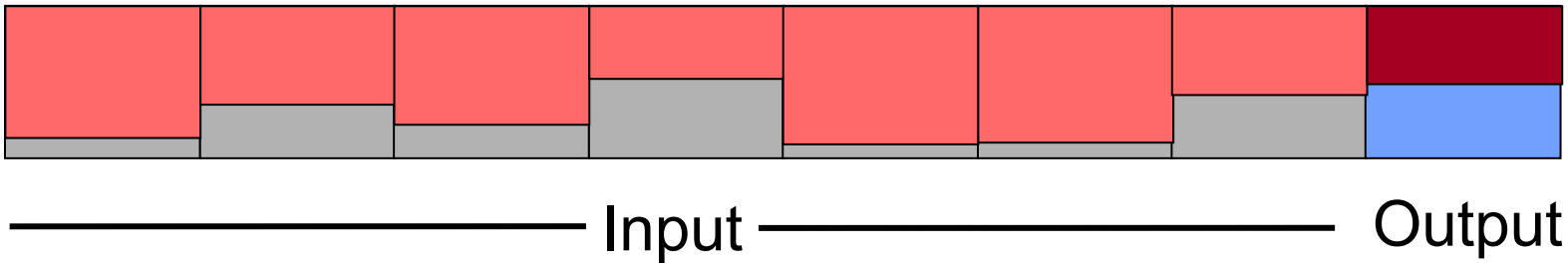


Example

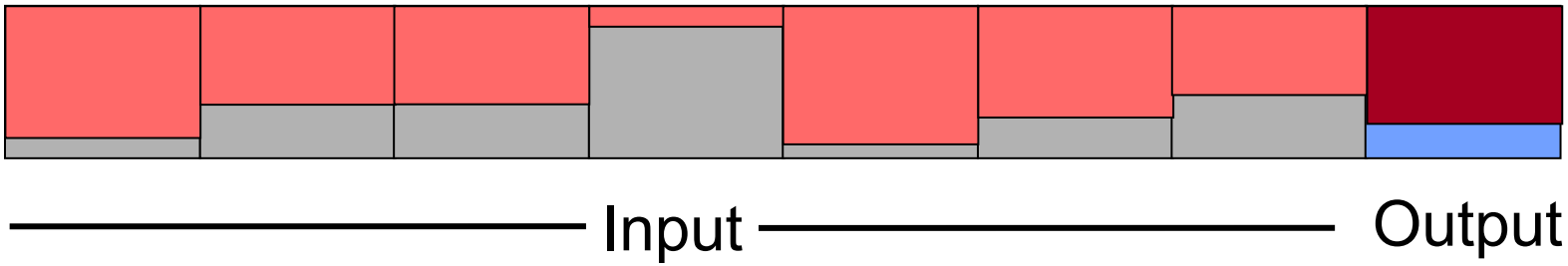
最强壮的小分队形成，去打头阵



Example

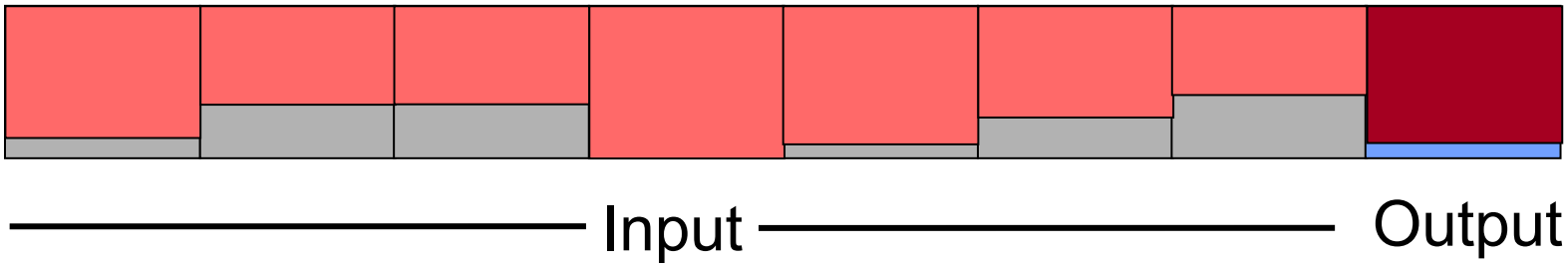


Example

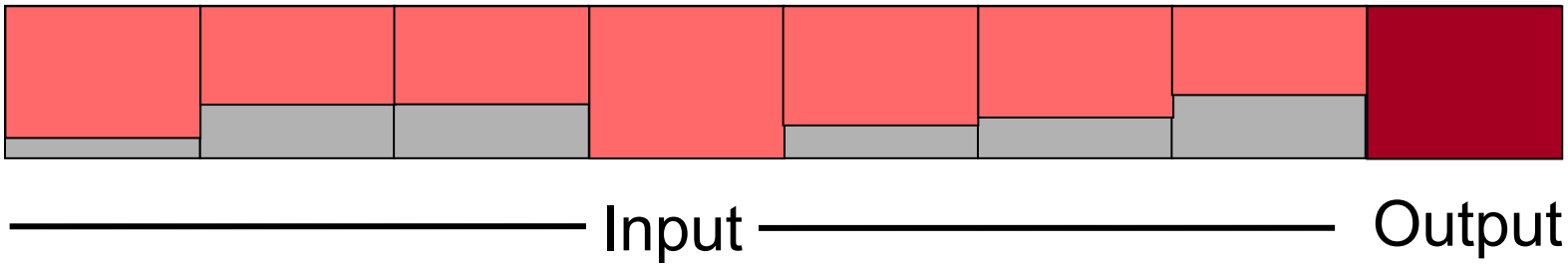


Example

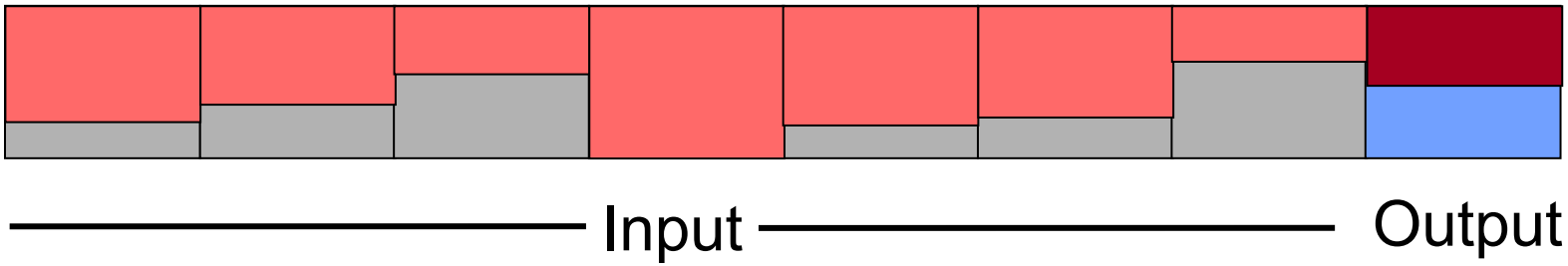
4路军最强，他们第一组已全部打前锋去了，
后续的士兵马上补上！



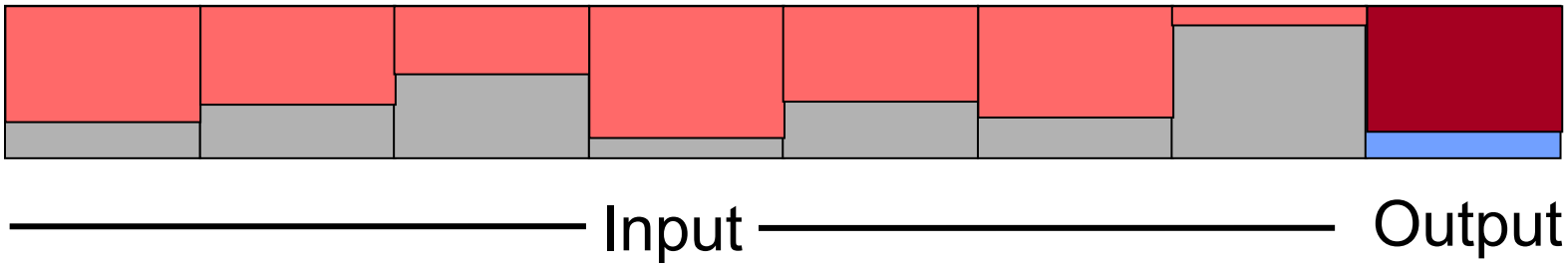
Example



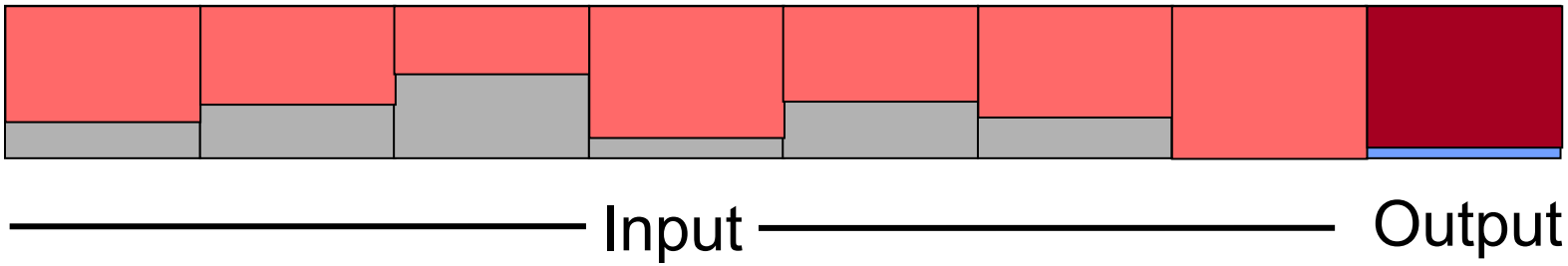
Example



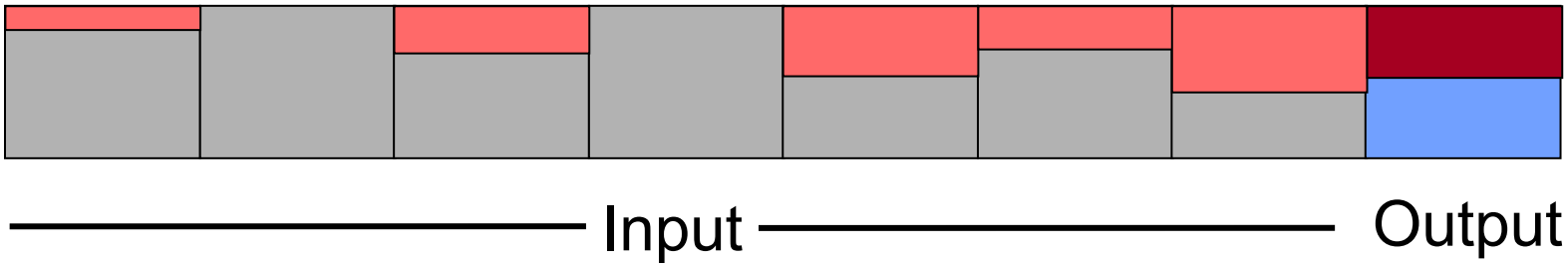
Example



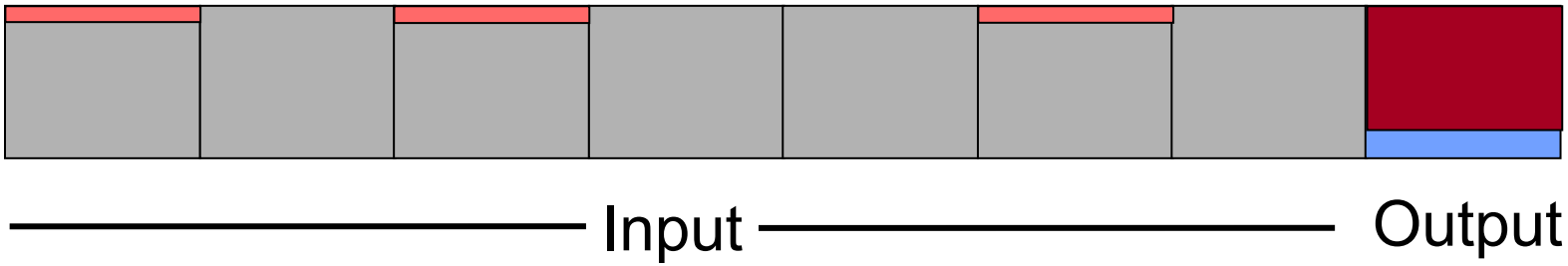
Example



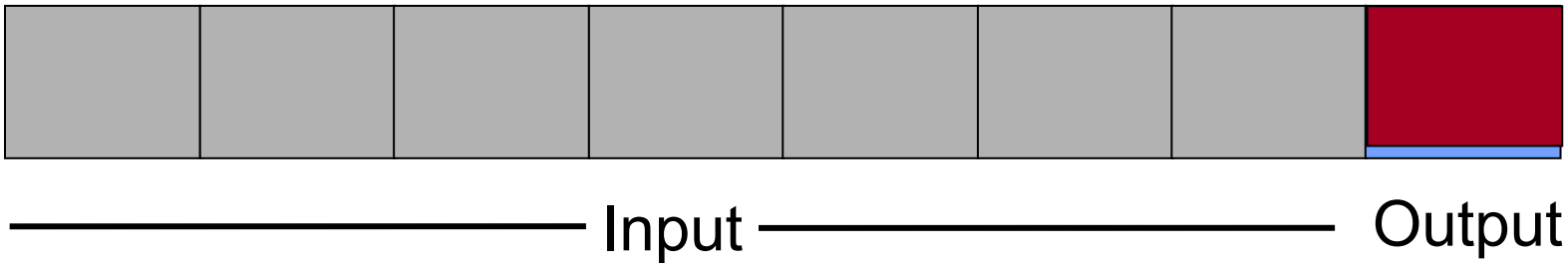
Example



Example



Example



Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Formula check: $\lceil \log_4 22 \rceil = 3 \ldots + 1 \rightarrow$ 4 passes √

Number of Passes of External Sort

(I/O cost is 2N times number of passes)

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Can I do two passes?

- Pass 0: sort runs
  - 用内排序方法，例如快排
  - 使用所有buffer作为输入和输出，不需要分开为输入或输出
- Pass 1: merge runs
  - 用归并排序
  - 使用1个buffer作输出，其他作为输入
- N pages file
- Given B buffers
- Need:
  - No more than B-1 runs (输入只有B-1个pages)
  - Each run no longer than B pages（Pass0的内排最多支持B个pages）
- Can do two passes if $N \leq B * (B-1)$
- Question: what's the largest file we can sort in three passes? K passes?

Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- An alternative is “tournament sort”(a.k.a. “heapsort”)
  - **Top: Read in *B blocks***
  - **Output: move smallest record to output buffer**
  - Read in a new record *r*
  - insert *r into “heap”*
  - if *r not smallest, then **GOTO Output***
  - else remove *r from “heap”*
  - output “heap” in order; **GOTO Top**

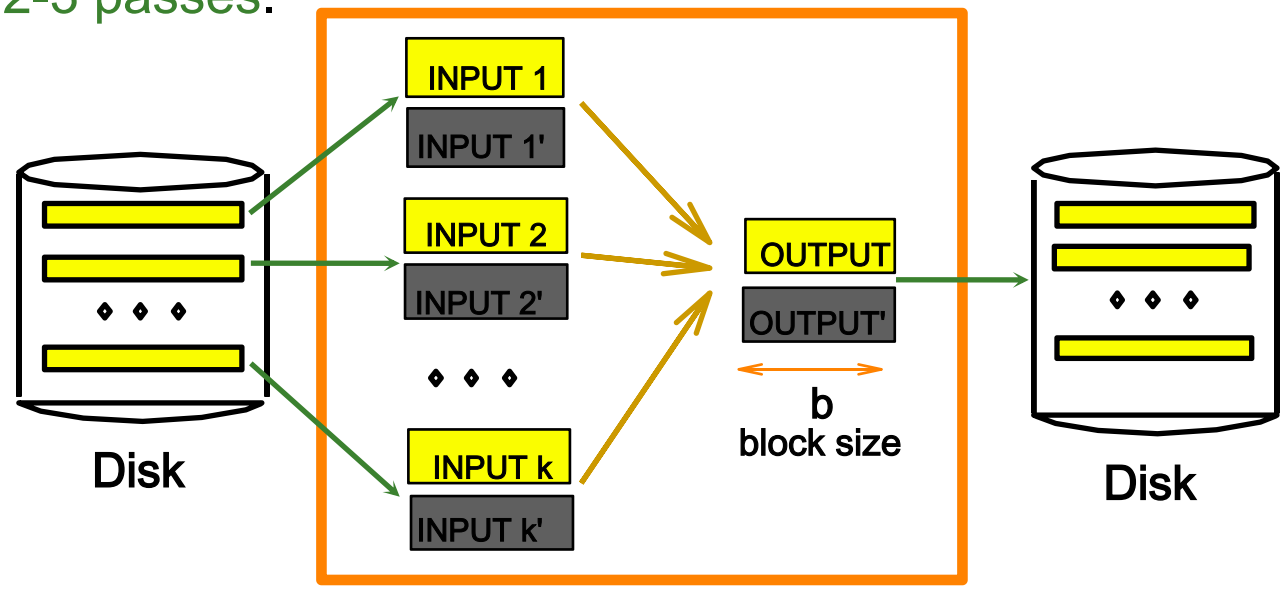
Blocked I/O for External Merge Sort

- Do I/O a page at a time
  - Not one I/O per record
- In fact, read a *block (chunk)* of pages sequentially!
- Suggests we should make each buffer (input/output) be a *block* of pages.
  - But this will reduce fan-in during merge passes!
  - In practice, most files still sorted in 2-3 passes.

Theme: Amortize a random I/O across more data read. But pay for it in memory footprint

Double Buffering

- Goal: reduce wait time for I/O requests during merge
- Idea: 2 blocks RAM per run, disk reader fills one while sort merges the other
  - Potentially, more passes; in practice, most files *still* sorted in 2-3 passes.

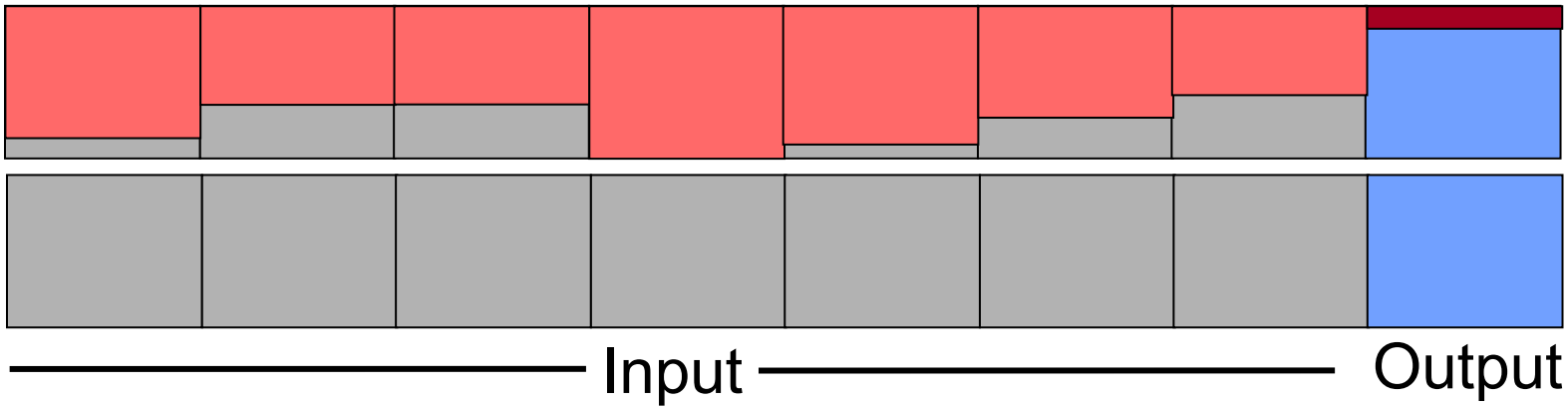


B main memory buffers, k-way merge

Theme: overlap I/O and CPU activity via read-ahead (prefetching)

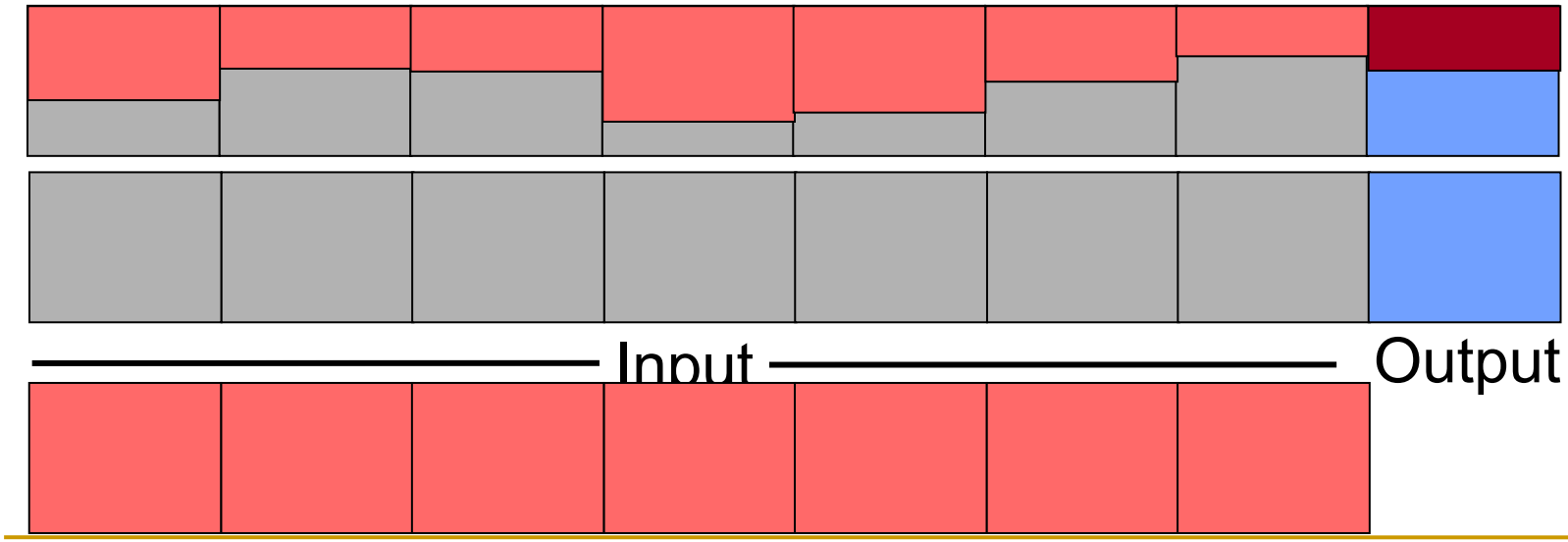
Solution: double buffering

- Keep a second set of buffers
  - Process one set while waiting for disk I/O to fill the other set



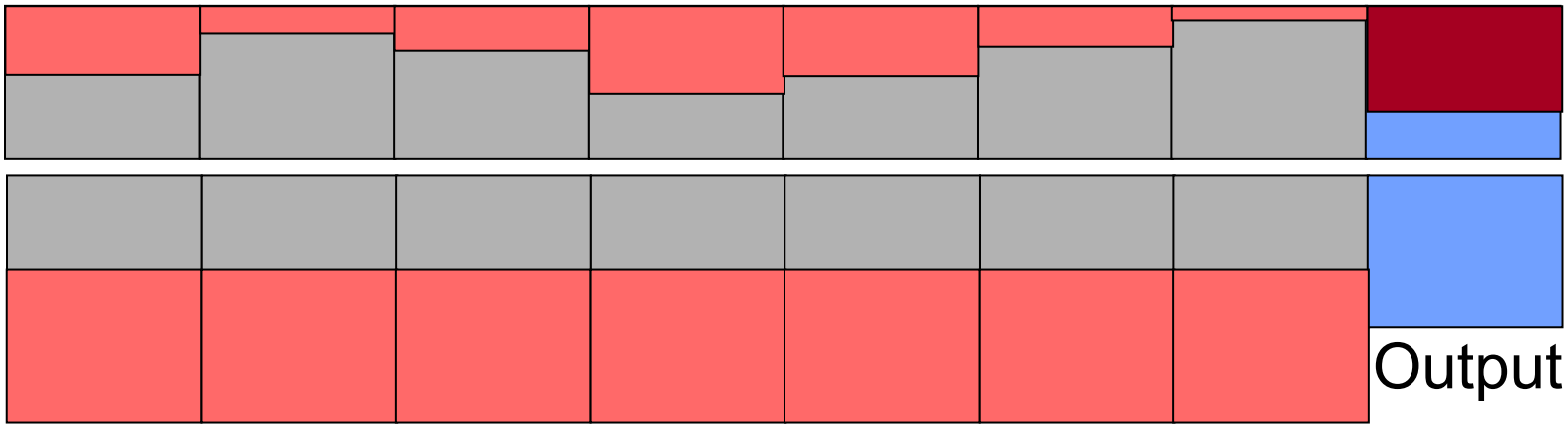
Solution: double buffering

- Keep a second set of buffers
  - Process one set while waiting for disk I/O to fill the other set



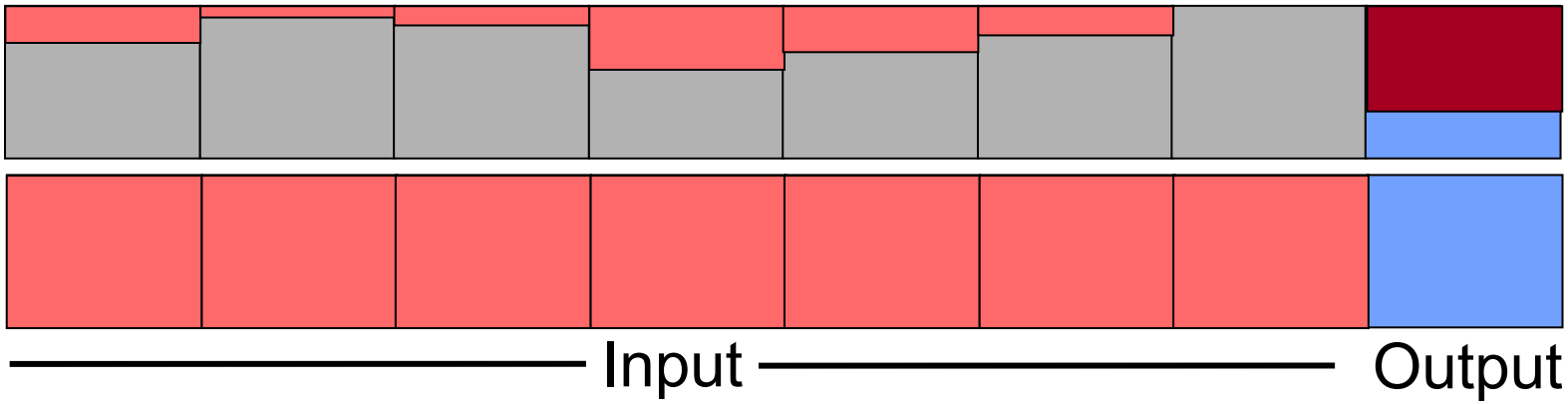
Solution: double buffering

- Keep a second set of buffers
  - Process one set while waiting for disk I/O to fill the other set



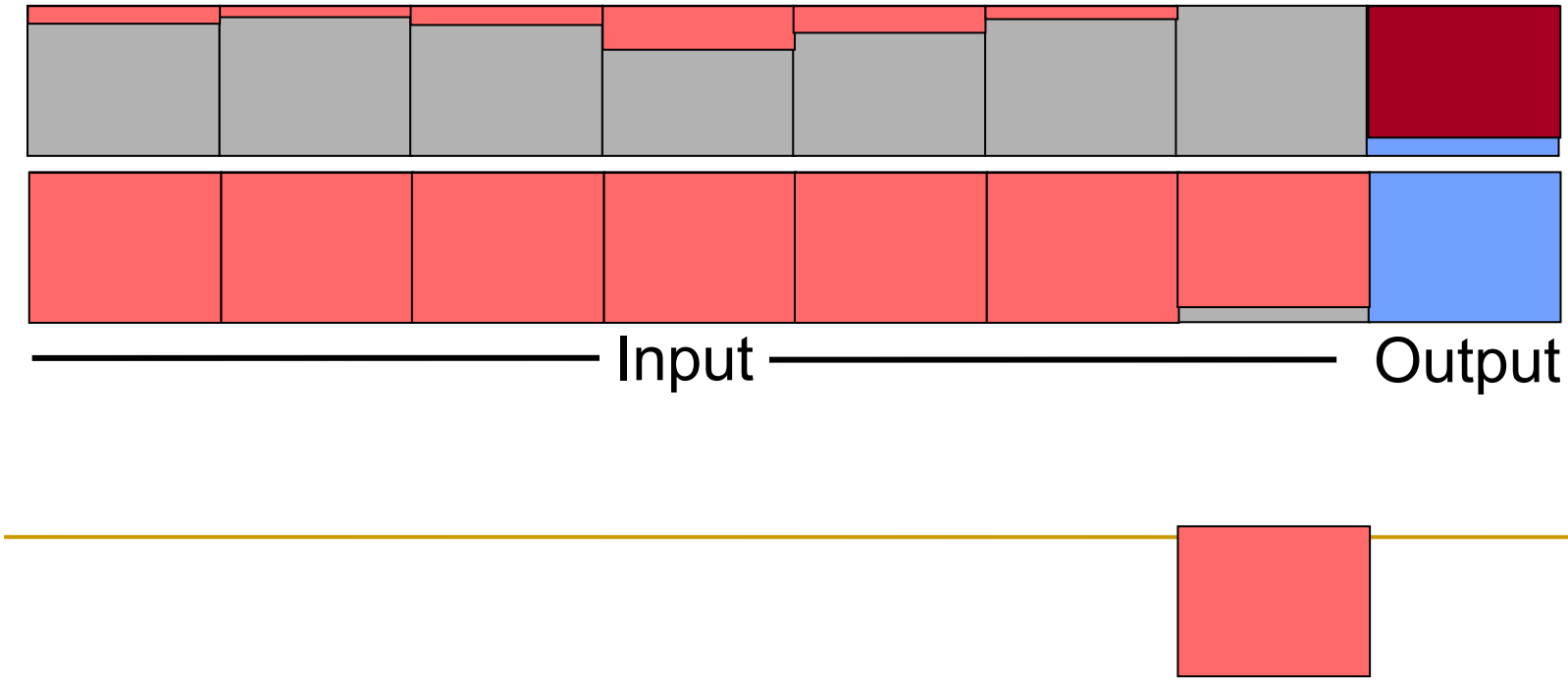
Solution: double buffering

- Keep a second set of buffers
  - Process one set while waiting for disk I/O to fill the other set



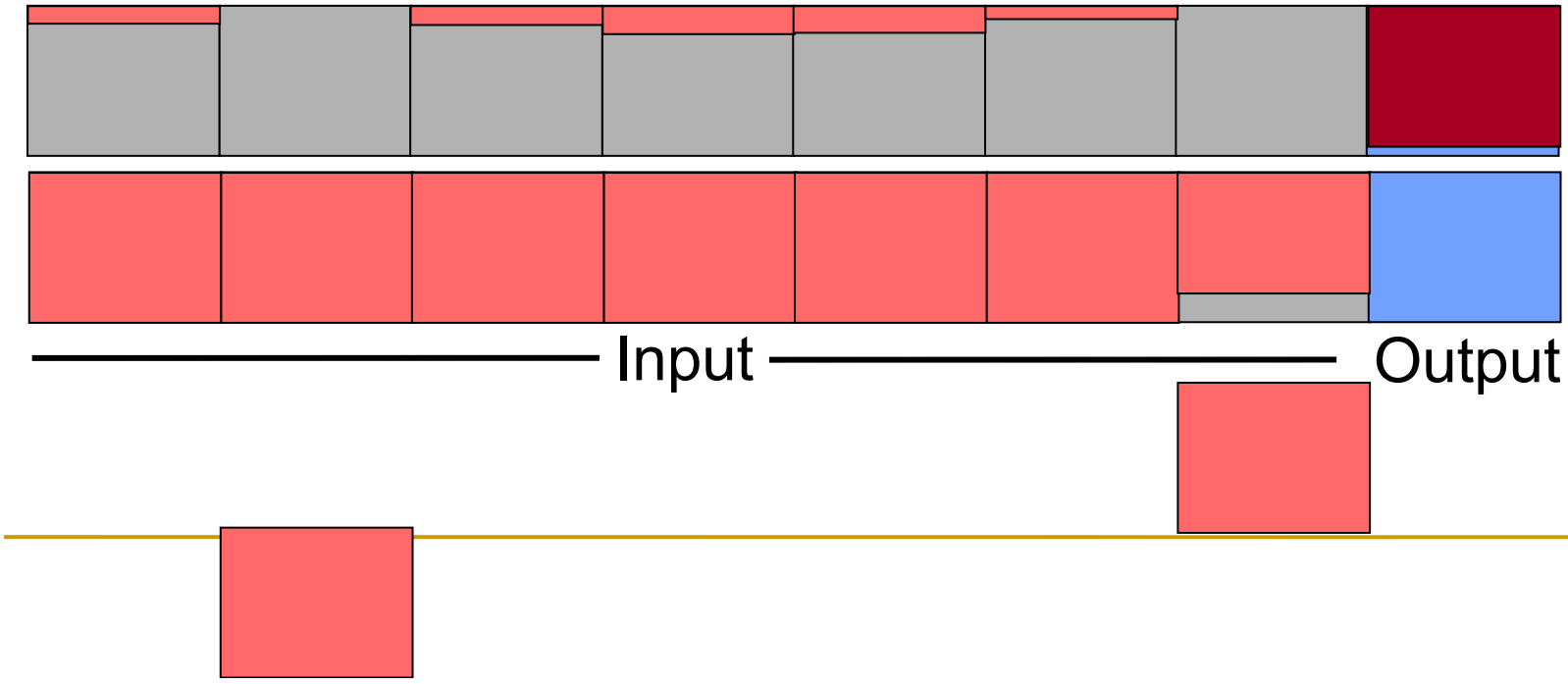
Solution: double buffering

- Keep a second set of buffers
  - Process one set while waiting for disk I/O to fill the other set



Solution: double buffering

- Keep a second set of buffers
  - Process one set while waiting for disk I/O to fill the other set

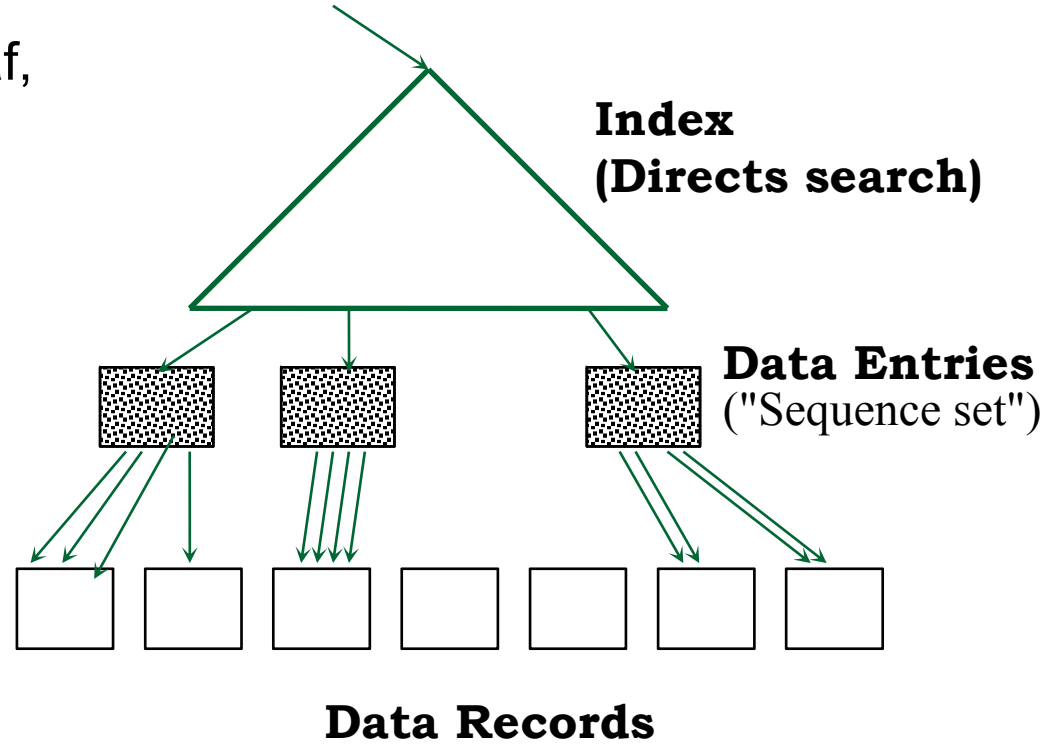


Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- *Is this a good idea?*
- Cases to consider:
  - B+ tree is clustered — ***Good idea!***
  - B+ tree is not clustered — ***Could be a very bad idea!***

Clustered B+ Tree Used for Sorting

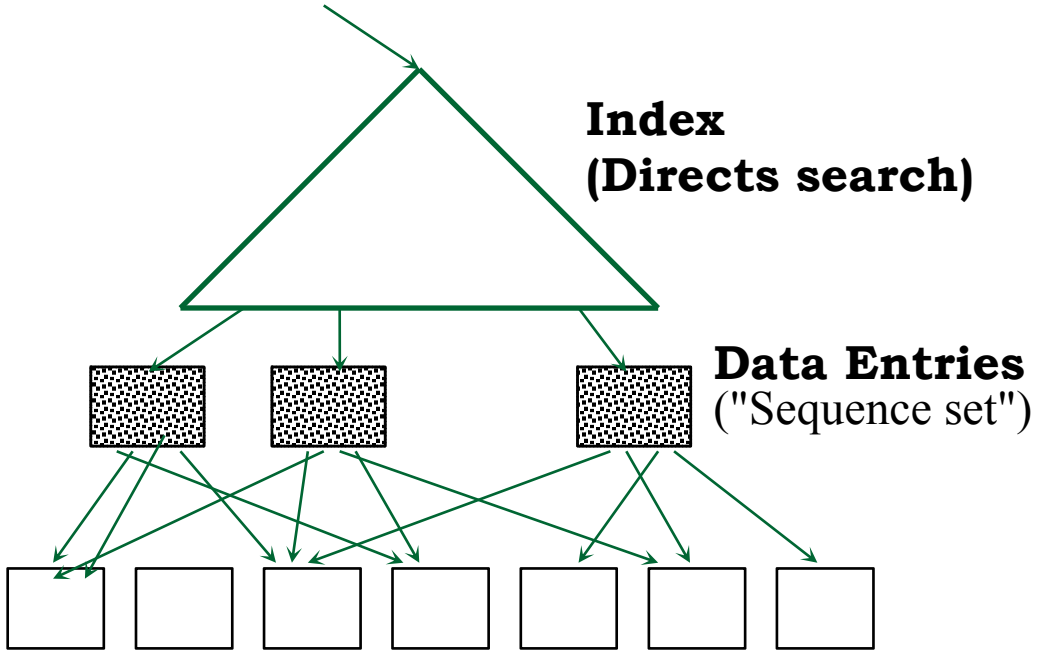
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.



✉ *Always better than external sorting!*

Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains *rid* of a data record. In general, one I/O per data record!



External Sorting vs. Unclustered Index

$$Cost = 2N * (1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil)$$

N	Sorting	p=1	p=10	p=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

- *p*: # of records per page
- *B=1,000 and block size=32 for sorting*
- *p=100 is the more realistic value.*

What did that cost us?

- Traverse B+ tree to left-most leaf page
- Read all leaf pages
  - For each leaf page, read data pages
- Data not in B+ tree:
  - Height + Width + Data pages
- Data in B+ tree:
  - Height + Width

Example

- 1,000,000 records, 12,500 data pages
- Assume keys are 10 bytes, disk pointers are 8 bytes
  - So ≈ 300 entries per 8 KB B+ tree page (if two-thirds full)
- Data not in B+ tree
  - 12,500 entries needed = 42 leaf pages
  - Two level B+tree
  - Total cost: 1 + 42 + 12,500 = 12,543 I/Os
  - 2 minutes versus 17 minutes for external merge sort
- Data in B+ tree
  - Three level B+ tree, 12,500 leaf pages
  - Total cost: 2 + 12,500 = 12,502 I/Os
  - Also about 2 minutes

Summary

- External sorting is important
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# buffer pages). Later passes: ***merge*** runs.
  - # of runs merged at a time depends on ***B,*** and ***block size.***
  - Larger block size means less I/O cost per page.
  - Larger block size means smaller # runs merged.
  - In practice, # of runs rarely more than 2 or 3.

Summary (cont.)

- Choice of internal sort algorithm may matter:
  - Quicksort: Quick!
  - Heap/tournament sort: slower (2x), longer runs
- The best sorts are wildly fast:
  - Despite 40+ years of research, still improving!
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.