CSE462/562: Database Systems (Fall 24)

Lecture 12: Single-table query processing: Sorting and Hashing

10/1/2024 & 10/3/2024

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Last updated: 9/10/2024 1:30 PM

Sort operator

- Use cases
  - ORDER BY
  - Group by
  - Distinct (deduplication)
  - For Sort-Merge Join
  - For bulk loading tree indexes
  - For Set operations
  - …

- If data fit in memory -- easy
  - quick sort
  - merge sort
  - …

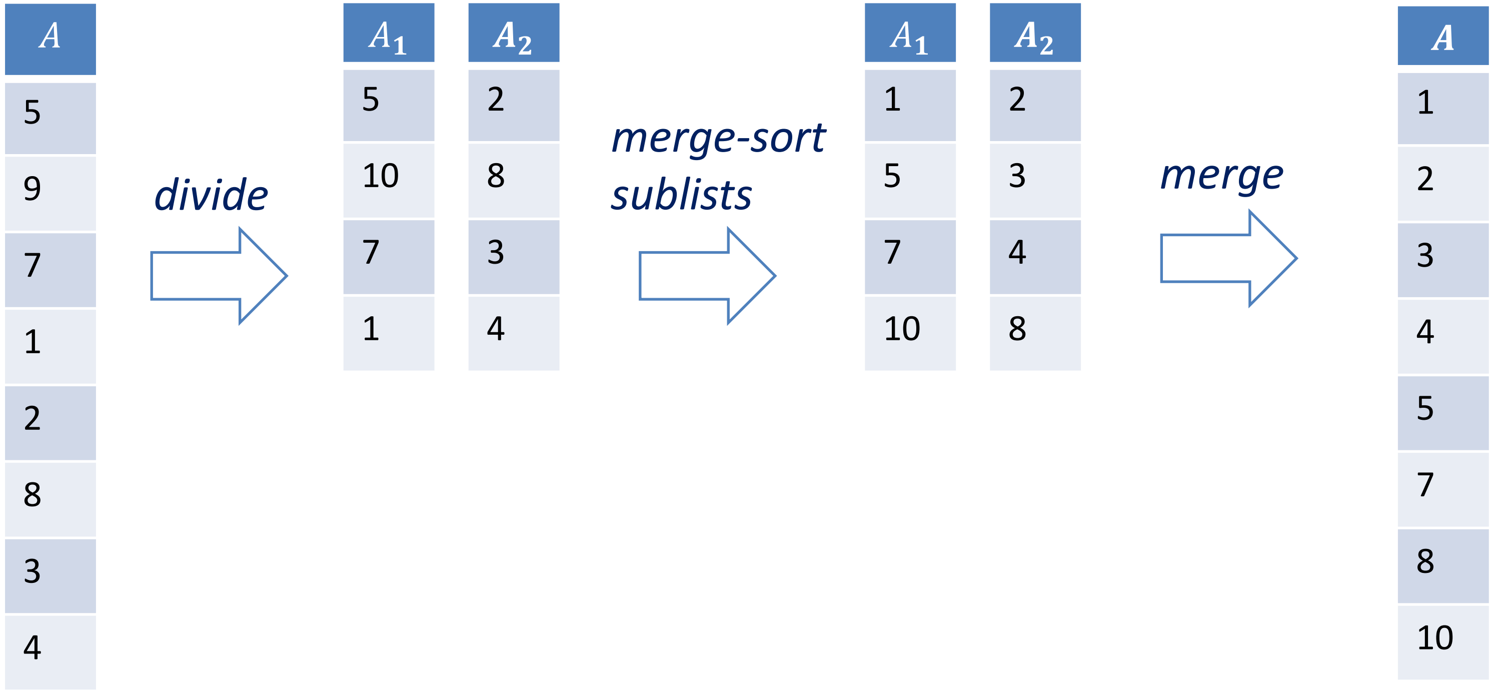
External sorting

- Problem: sort or hashing 1TB of data over 1GB of RAM
  - Why not virtual memory?
    - Swaps involve expensive random I/Os
  - Why not using B-Tree/extendible hashing/linear hashing?
    - Dynamic structures carry additional overhead for maintenance (not needed in QP)
    - Missing optimization opportunities with hybrid approach (see later)

- General wisdom:
  - I/O cost dominates the total cost
  - Design algorithms to reduce the number of I/Os

In-memory two-way merge-sort: a starting point

- Recall the two-way merge-sort
  - given a list of items in $A[0..n-1]$
  - recursively divide and conquer the problem
    - divide the list into two halves $A_1\left[0..\left\lfloor\frac{n}{2}\right\rfloor\right], A_2\left[\left\lfloor\frac{n}{2}\right\rfloor+1, n-1\right]$
    - merge-sort A_1 and A_2 individually
    - merge the two sorted list A_1, A_2



A
5
9
7
1
2
8
3
4

divide

A_1	A_2
5	2
10	8
7	3
1	4

merge-sort sublists

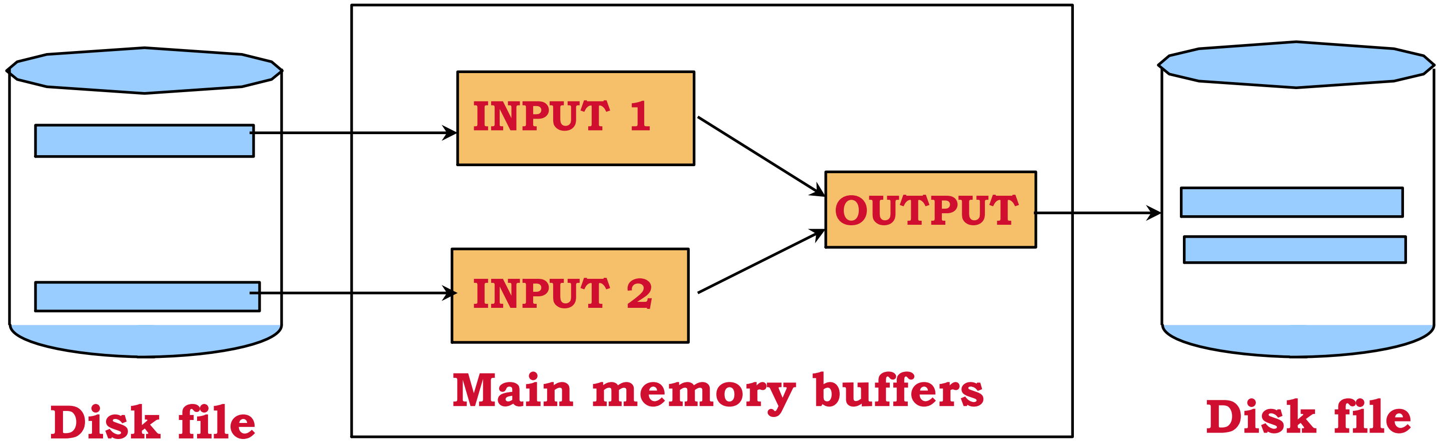
A_1	A_2
1	2
5	3
7	4
10	8

merge

A
1
2
3
4
5
7
8
10

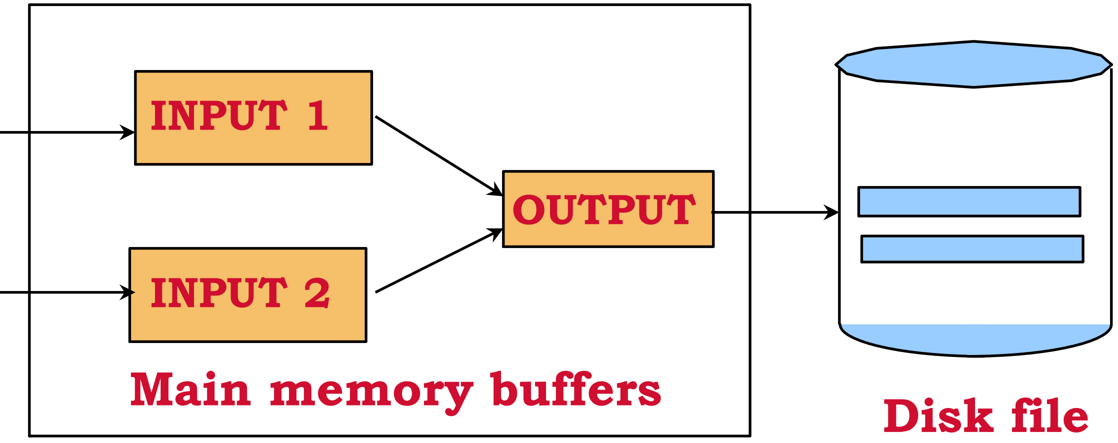
External two-way merge sort

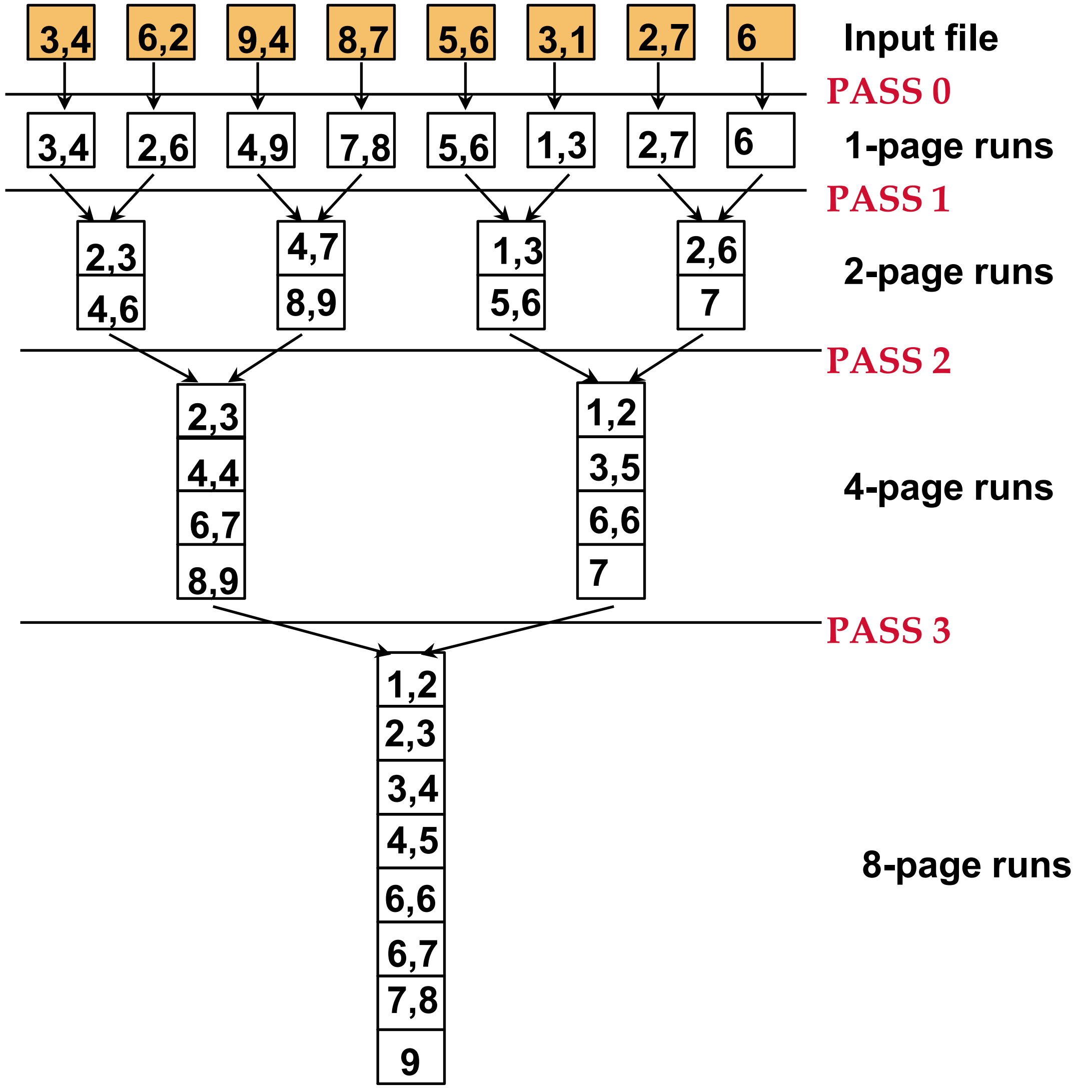
- Needs 3 buffers
- Instead of recursion
    - works bottom up from the input



External two-way merge sort

- Needs 3 buffers
- Instead of recursion
    - works bottom-up from the input

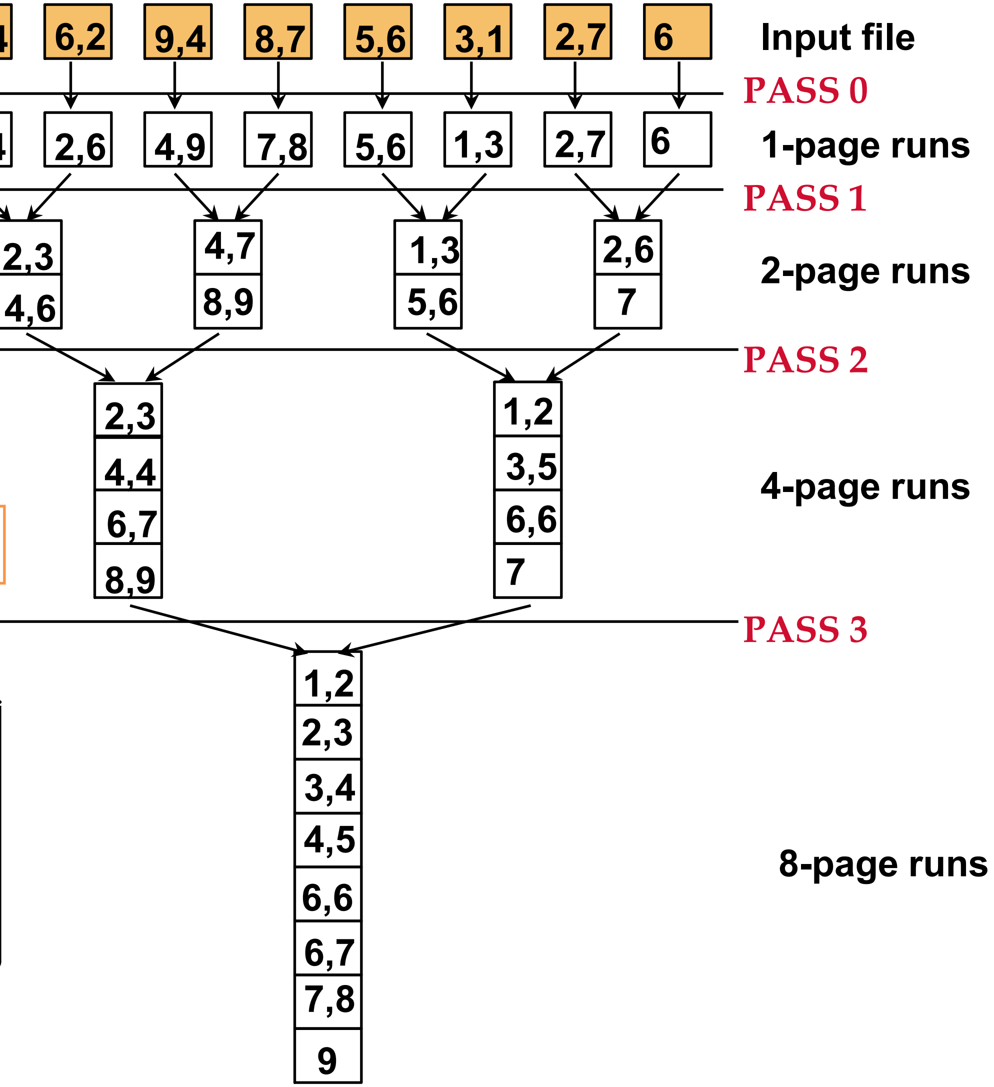


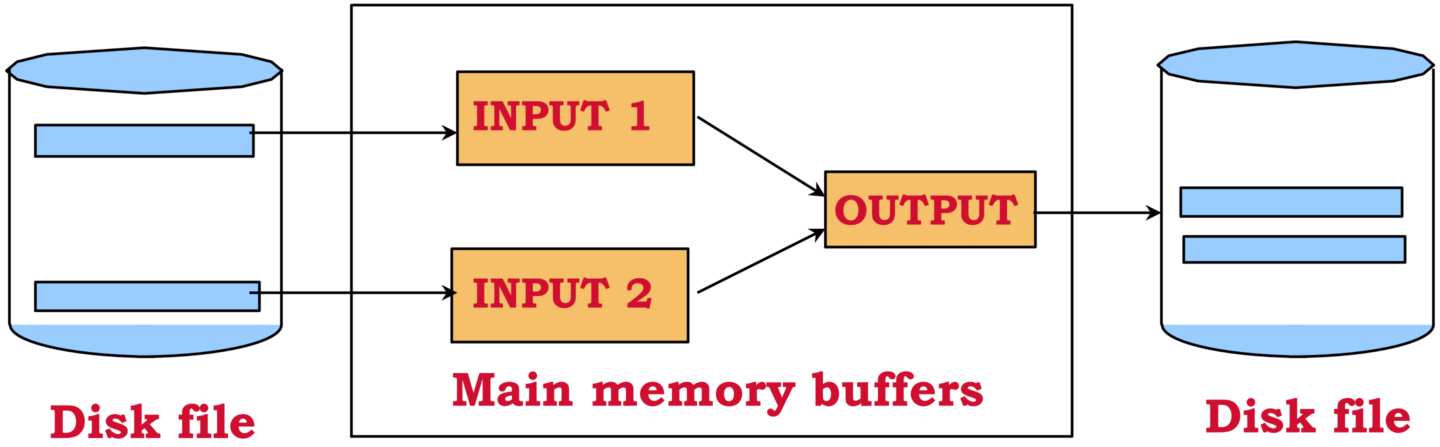


External two-way merge sort

- Input: N pages
- Cost for a pass: reading & writing N pages once
- # of passes: height of the tree = $\lceil \log_2 N \rceil + 1$
- Total cost: $2N(\lceil \log_2 N \rceil + 1)$ I/Os
  - Transfer cost: $2t_T N(\lceil \log_2 N \rceil + 1)$
  - *Seek cost:* $2t_S N(\lceil log_2 N \rceil + 1)$
  - *total* = $2(t_T + t_S) N(\lceil \log_2 N \rceil + 1)$

Not so efficient!





External multi-way merge sort

- How do we fully utilize all the M buffers?
  - Solution: (M-1)-way merge-sort
- Pass 0: internal sort to produce initial runs
  - read every M pages into memory
  - use some internal sorting algorithm (e.g., quick sort)
    - *can produce even larger runs (later)*
  - write all the M pages as a run

N pages in input

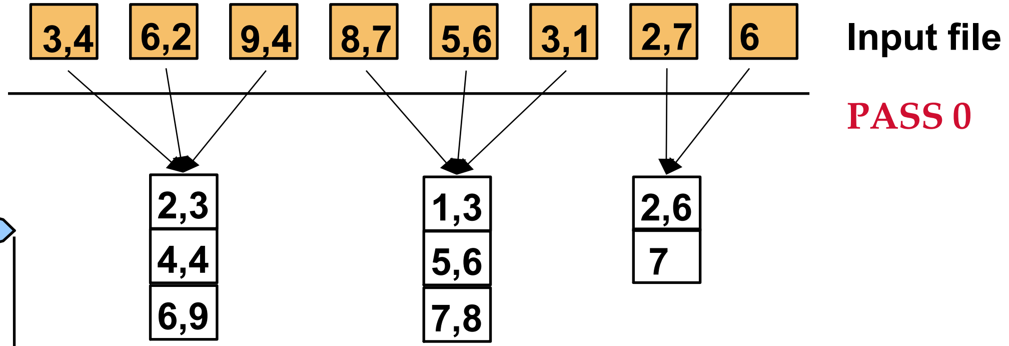
$\lceil \frac{N}{M} \rceil$ runs after pass 0

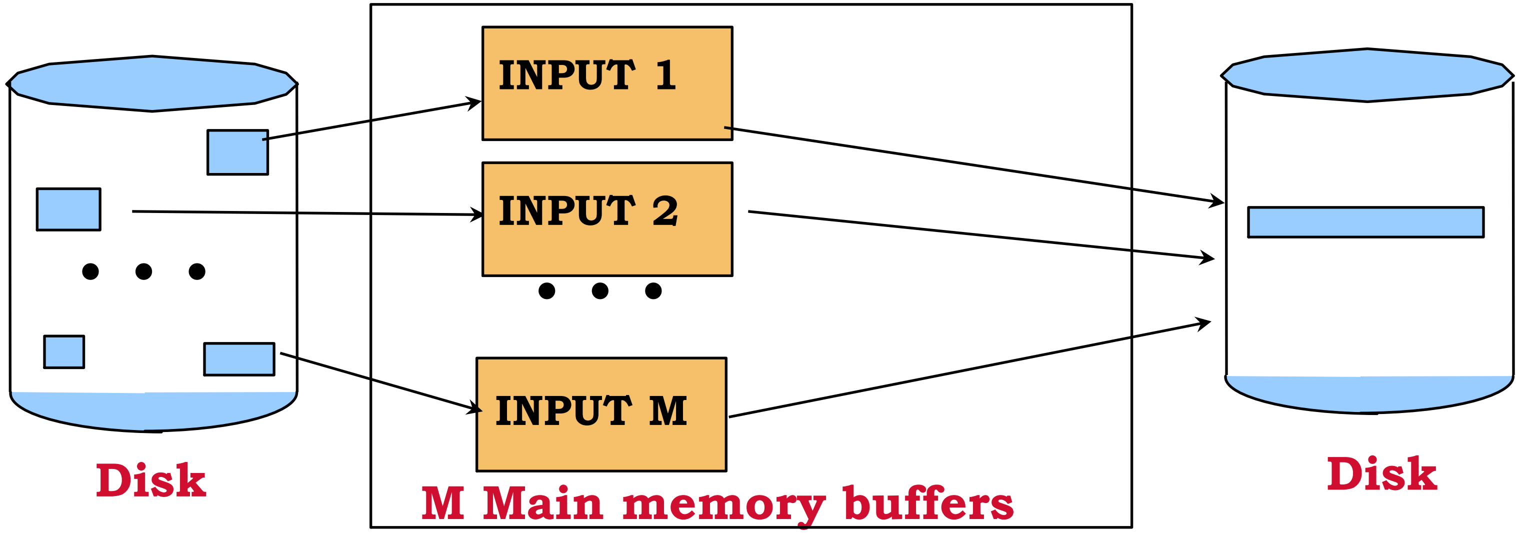
Cost:

$2N$ pages read/written +

$2\left\lceil \frac{N}{M} \right\rceil$ seeks

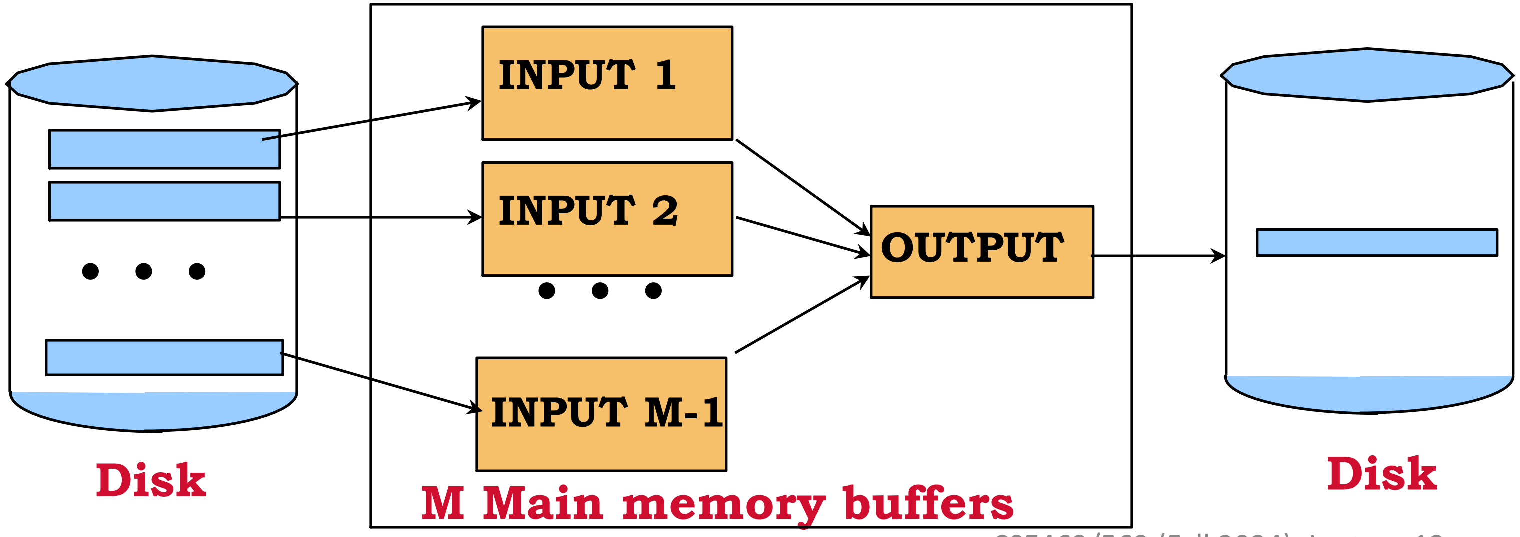
i.e. $2Nt_T + 2\left\lceil \frac{N}{M} \right\rceil t_S$

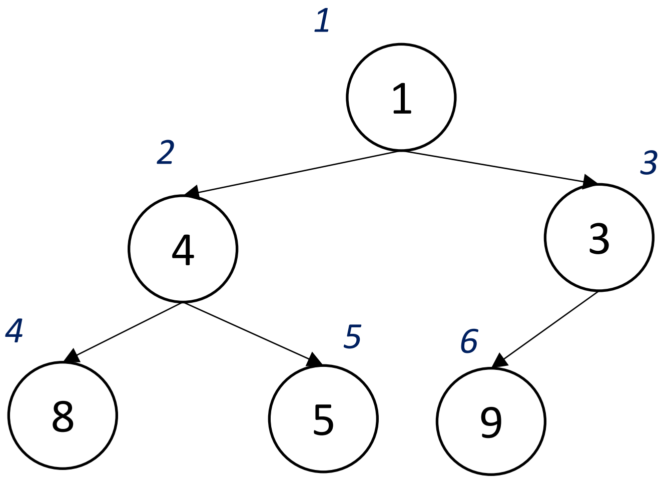




General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap/max-heap (aka priority queue)*
    - supports $O(logM)$ time insertion of any item and deletion of the smallest/largest item
    - a complete binary tree where parent is smaller/larger than both children
    - how to implement
      - numbering nodes level by level sequentially from 1, store in an array $A[1..n]$
        - (how to translate 1-based index to 0-based in C/C++?)
      - parent of $A[i]$ is $A[i/2]$, left child of $A[i]$ is $A[i*2]$, right child of $A[i]$ is $A[i*2+1]$
      - push-down or push-up to maintain the variant



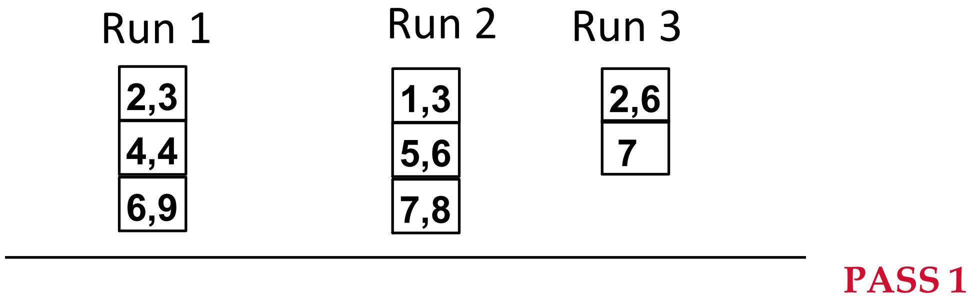


A	1	4	3	8	5	9

General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
    - maintain *a min-heap*
    - load one page from each of the $M - 1$ runs
        - and maintain pointers of next page to read
    - for each loaded page
        - insert the first key into the min-heap
        - maintain next slot ids for each page
    - Repeatedly remove the smallest item from the min heap
        - and replace it with the next key in its run
        - write out the output page once it's full

For illustration, let's now assume M = 4 instead of 3 from now on.



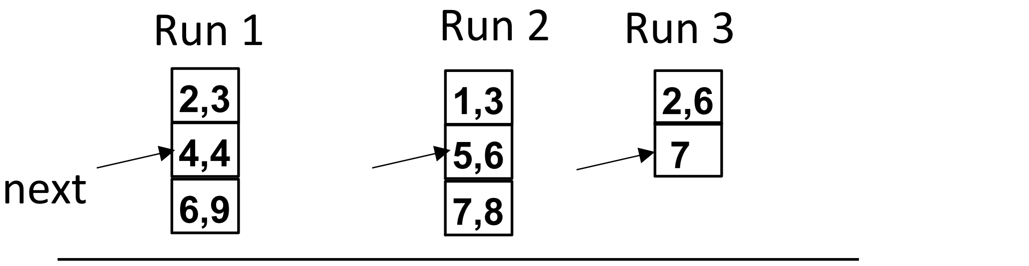
Run 1	Run 2	Run 3
2,3	1,3	2,6
4,4	5,6	7
6,9	7,8	

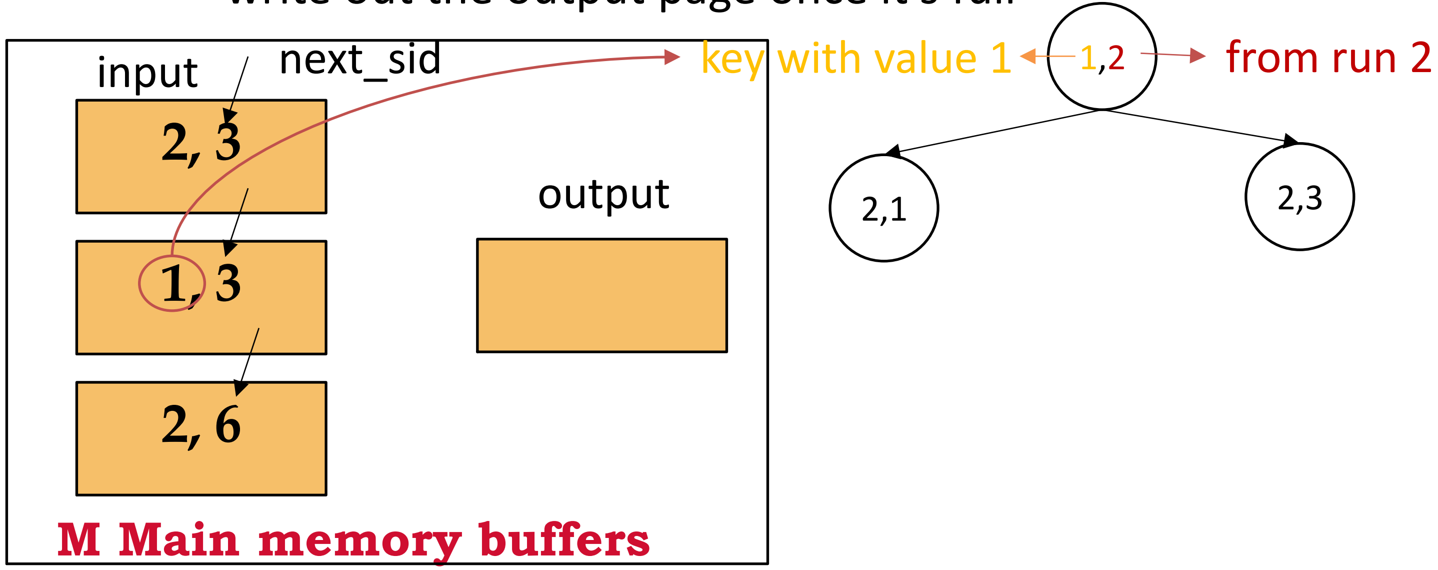
PASS 1

General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap*
  - load one page from each of the $M-1$ runs
    - and maintain pointers of next page to read
  - for each loaded page
    - insert the first key into the min-heap
    - maintain next slot ids for each page
  - Repeatedly remove the smallest item from the min heap
    - and replace it with the next key in its run
    - write out the output page once it's full

For illustration, let's now assume $M = 4$ instead of 3 from now on.

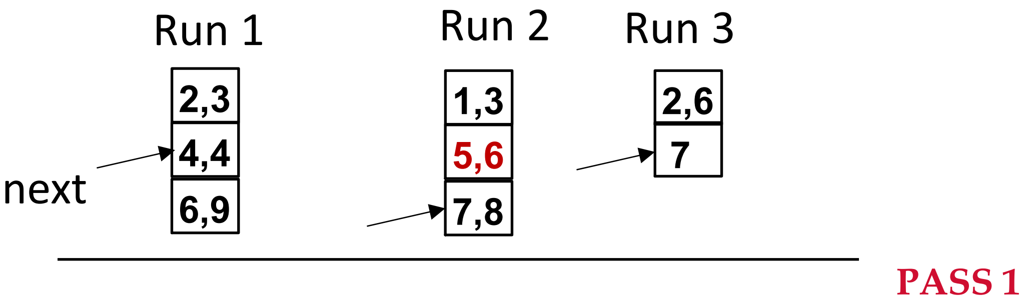


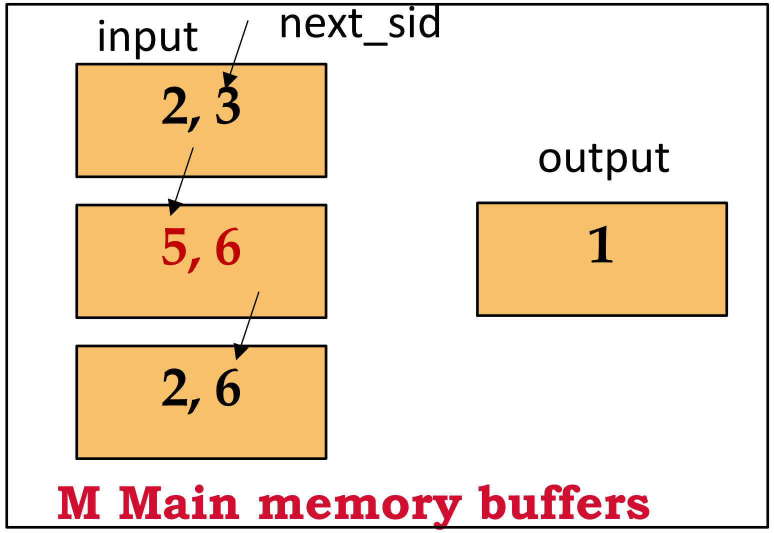


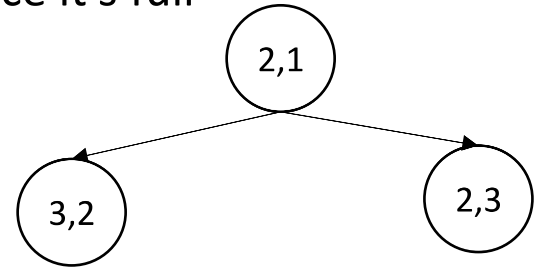
General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap*
  - load one page from each of the $M - 1$ runs
    - and maintain pointers of next page to read
  - for each loaded page
    - insert the first key into the min-heap
    - maintain next slot ids for each page
  - Repeatedly remove the smallest item from the min heap
    - and replace it with the next key in its run
    - write out the output page once it's full

For illustration, let's now assume $M = 4$ instead of 3 from now on.



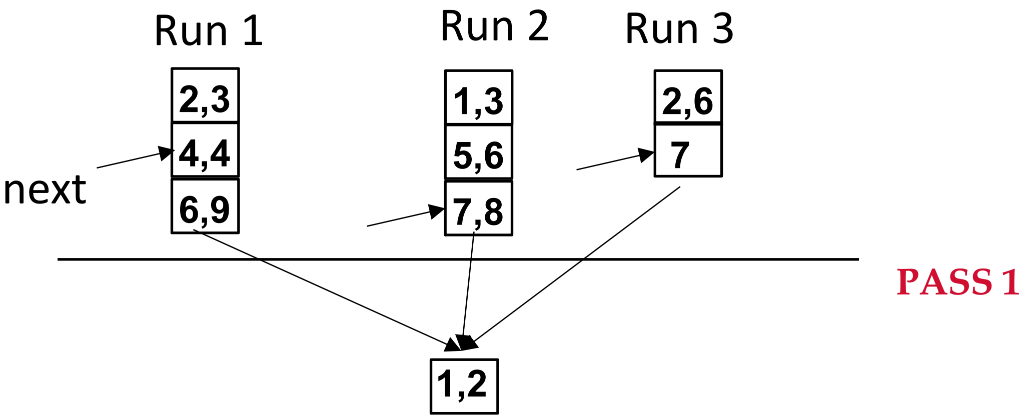


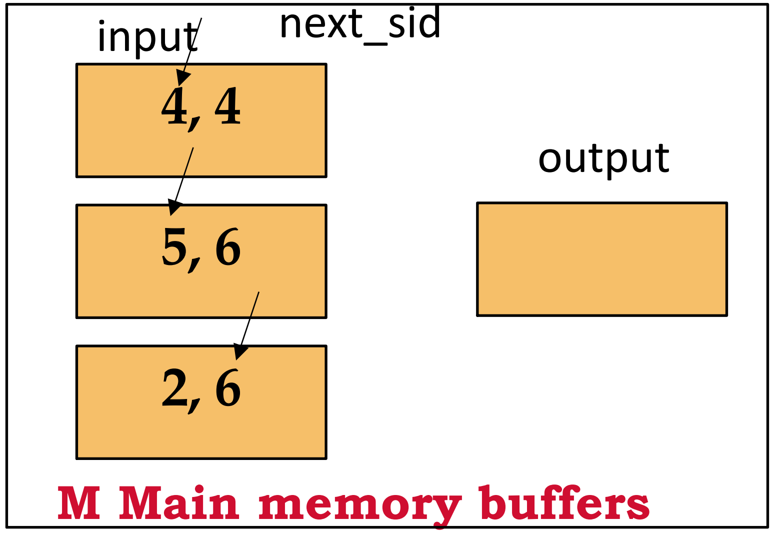


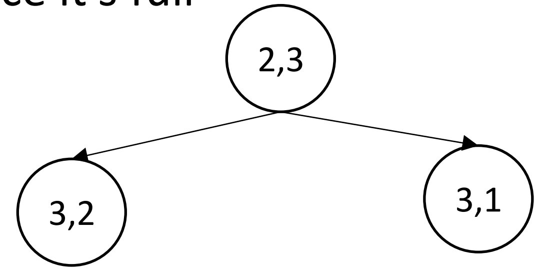
General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap*
  - load one page from each of the $M - 1$ runs
    - and maintain pointers of next page to read
  - for each loaded page
    - insert the first key into the min-heap
    - maintain next slot ids for each page
  - Repeatedly remove the smallest item from the min heap
    - and replace it with the next key in its run
    - write out the output page once it's full

For illustration, let's now assume $M = 4$ instead of 3 from now on.



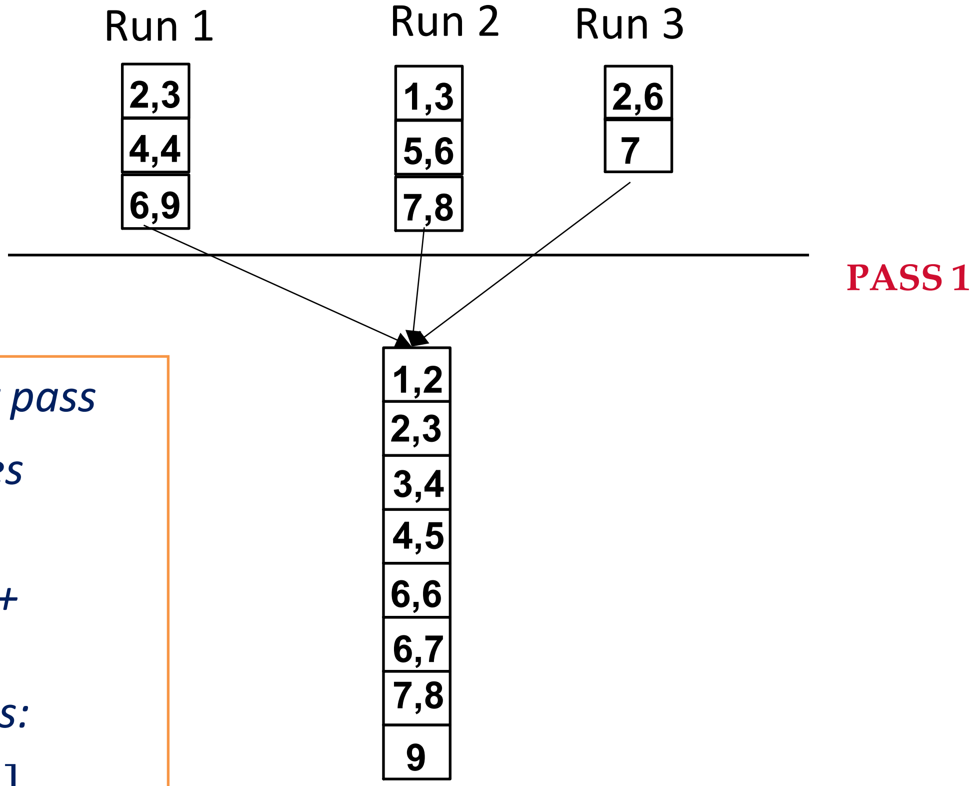


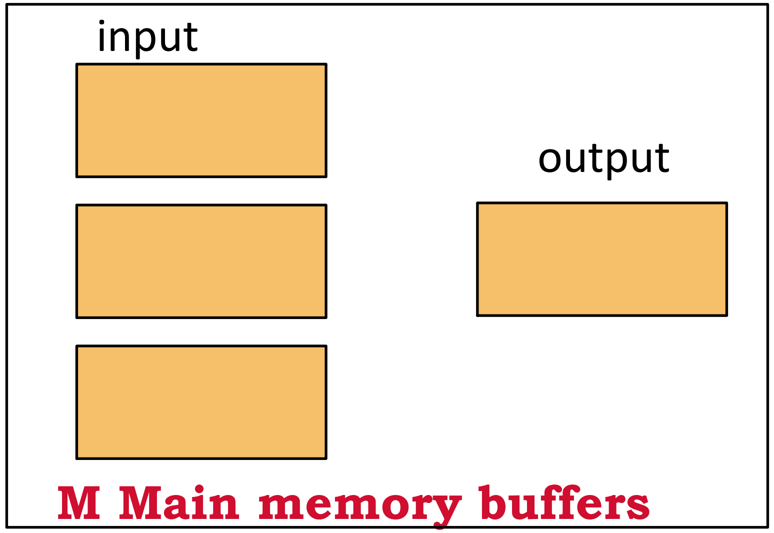


General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap*
  - load one page from each of the $M-1$ runs
    - and maintain pointers of next page to read
  - for each loaded page
    - insert the first key into the min-heap
    - maintain next slot ids for each page
  - Repeatedly remove the smallest item from the min heap
    - and replace it with the next key in its run
    - write out the output page once it's full

For illustration, let's now assume M = 4 instead of 3 from now on.


Run 1
2,3
4,4
6,9
Run 2
1,3
5,6
7,8
Run 3
2,6
7
PASS 1
1,2
2,3
3,4
4,5
6,6
6,7
7,8
9


input
output
M Main memory buffers

N pages to read/write per pass

$\left\lceil log_{M-1}\left\lceil\frac{N}{M}\right\rceil\right\rceil$ merge passes

Cost per merge pass:

$2N$ pages read/written +

$2N$ seeks

Total cost for merge passes:

$$2(t_T+t_S)N\lceil\log_{M-1}\lceil\frac{N}{M}\rceil\rceil$$

Cost analysis

- Cost analysis:
  - Pass 0: $2Nt_T + 2\left\lceil \frac{N}{M} \right\rceil t_S$
  - Pass 1, 2, … combined: $2(t_T + t_S)N\lceil \log_{M-1}\lceil \frac{N}{M} \rceil \rceil$
  - Total = $2t_T N\left(\left\lceil log_{M-1}\left\lceil \frac{N}{M} \right\rceil \right\rceil + 1\right) + 2t_S\left(\left\lceil \frac{N}{M} \right\rceil + N\lceil log_{M-1}\lceil \frac{N}{M} \rceil \rceil\right)$

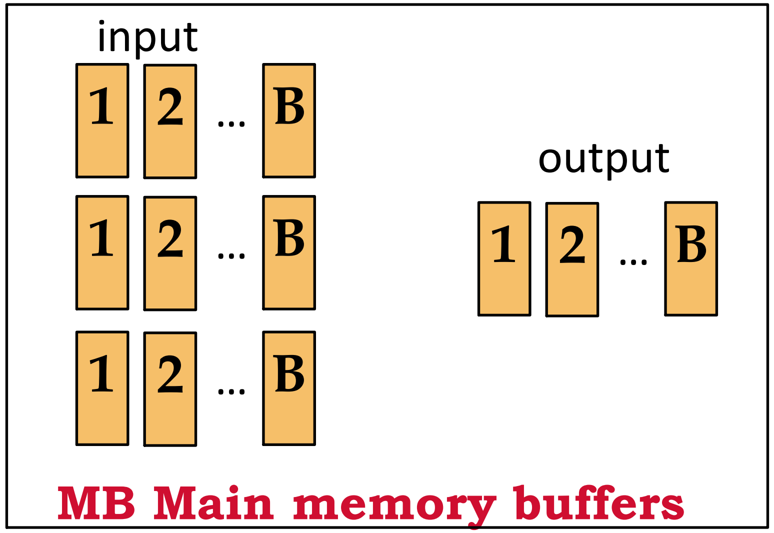
- *gain of utilizing all available buffers*
- *importance of a high fan-in during merging*

N	M=3	=5	=9	=17	=129	=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 we do it better?

Batching I/Os for merge sort

- Refinement 1
  - reducing random I/Os by reading/writing B pages per run during merge
  - using $(M-1)$-way merge sort
    - memory usage increases to MB pages
    - number of pages transferred do not change
    - but the number of random seeks per merge pass reduced to approximately $2\lceil\frac{N}{B}\rceil$
  - total cost reduced to $2t_T N\left(\left\lceil log_{M-1}\left\lceil\frac{N}{MB}\right\rceil\right\rceil+1\right)+2t_S\left(\left\lceil\frac{N}{MB}\right\rceil+\lceil\frac{N}{B}\rceil\lceil log_{M-1}\lceil\frac{N}{MB}\rceil\rceil\right)$

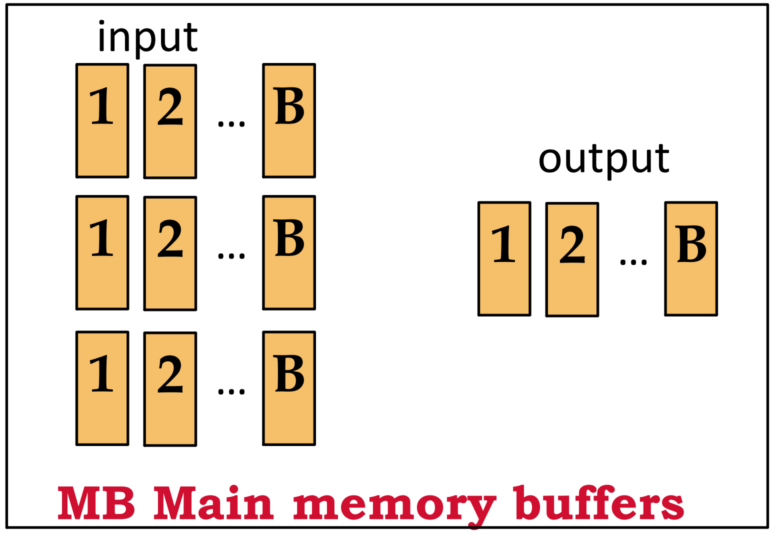


Exercise: what if we only have M pages instead of MB pages and still read/write pages in B-page batches?

$$2t_T N\left(\left\lceil log_{\lfloor\frac{M}{B}\rfloor-1}\left\lceil\frac{N}{M}\right\rceil\right\rceil+1\right)+2t_S\left(\left\lceil\frac{N}{M}\right\rceil+\lceil\frac{N}{B}\rceil\lceil log_{\lfloor\frac{M}{B}\rfloor-1}\lceil\frac{N}{M}\rceil\rceil\right)$$

Pipelining output

- Refinement 2
  - in most cases, do not need to write the final file
    - pipelining to the next operator
    - or output to user
  - Hence, no need to count the write of the final pass
  - total cost reduced to $t_T N \left(2 \left\lceil log_{\lfloor \frac{M}{B} \rfloor - 1} \left\lceil \frac{N}{M} \right\rceil \right\rceil + 1\right) + t_S \left(2 \left\lceil \frac{N}{M} \right\rceil + \lceil \frac{N}{B} \rceil (2\lceil log_{\lfloor \frac{M}{B} \rfloor - 1} \lceil \frac{N}{M} \rceil \rceil - 1)\right)$



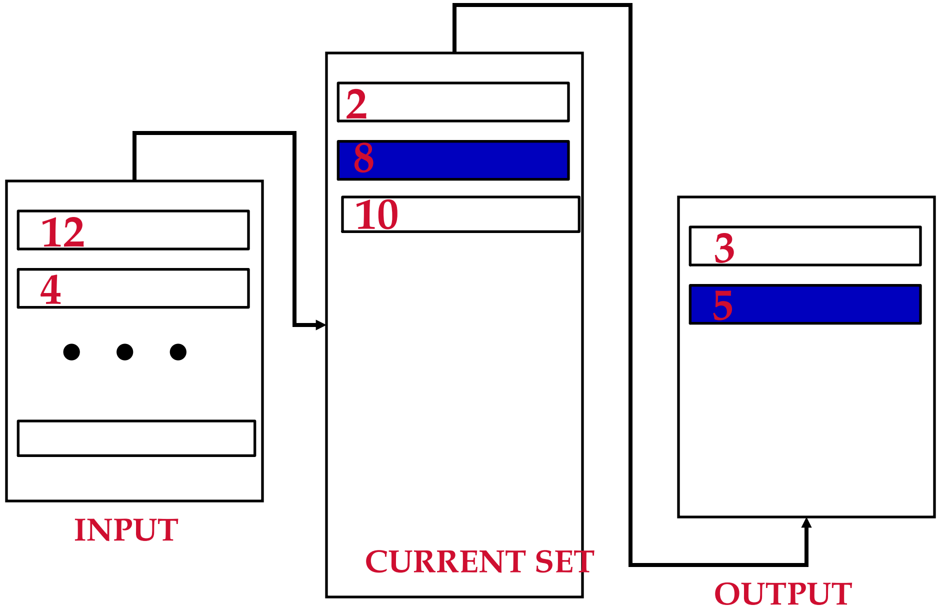
Tournament sort

- Refinement 3
  - producing initial runs as large as possible in pass 0
  - Alternative to quick-sort: “tournament sort” (a.k.a. “heapsort”, “replacement selection”)
- Keep two heaps in memory, H1 and H2, reserve an input buffer page and an output buffer page

```
read M-2 pages of records, inserting into H1;
while (records left) {
    m = H1.removemin();  put m in output buffer;
    if (H1 is empty)
        swap H1 and H2 (pointer swap only!); start new output run;
    else
        read in a new record r (use 1 buffer for input pages);
        if (r < m)  H2.insert(r);
        else        H1.insert(r);
}
H1.output();  start new run;  H2.output();
```

Tournament sort

- Tournament sort explained:



- 1 input, 1 output, M - 2 for current and next set (min heaps)
- Main idea: ensure the *smallest* key in the current set (H1) is *greater* than any key that has been written to this output run.
  - If it can’t be satisfied, write to the next set (H2), which goes into the next run.
- Memory usage of the min-heaps combined never exceeds the M-2 pages

Tournament sort

- Fact: average length of a run is *2(M-2)*
- Total cost reduced to on average

$$t_T N \left(2 \left\lceil log_{\left\lfloor \frac{M}{B} \right\rfloor - 1} \left\lceil \frac{N}{2M-4} \right\rceil \right\rceil + 1 \right) + t_S \left(2 \left\lceil \frac{N}{2M-4} \right\rceil + \lceil \frac{N}{B} \rceil (2 \lceil log_{\left\lfloor \frac{M}{B} \right\rfloor - 1} \lceil \frac{N}{2M-4} \rceil \rceil - 1) \right)$$

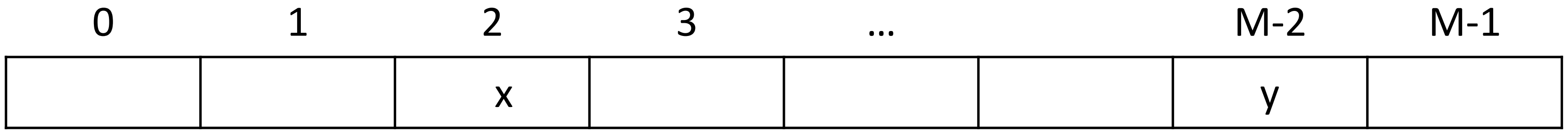
- Worst-Case:
  - What is min length of a run?
  - How does this arise?
- Best-Case:
  - What is max length of a run?
  - How does this arise?
- Quicksort is faster, but … longer runs often means fewer passes!

Hashing basics

- Hash function $h: U \rightarrow [M]$
    - U: key domain, $[M] = \{0,1,2, \ldots M-1\}$
    - *Deterministic*
    - Examples:
        - Multiplicative hashing for integers: $h(x) = \lfloor M \cdot frac(x * a) \rfloor$
            - a: a real number with a good mixture of 0s and 1s
            - $frac(y)$: the fractional part of a real number
            - can be efficiently implemented as $h(x) = \left(\frac{ax}{2^q}\right)\ mod\ m$ for appropriately chosen integers a, q, M
        - String hashing: SHA-1, MD5
            - often available off the shelf
            - can combine a *salt* to create different hash functions
                - e.g., SHA-1(concat(a, s)) for some randomly chosen string a
            - not that secure, but works well due to its efficiency

Hashing basics

- (In-memory) hash table
  - With a hash function $h: U \rightarrow [M]$



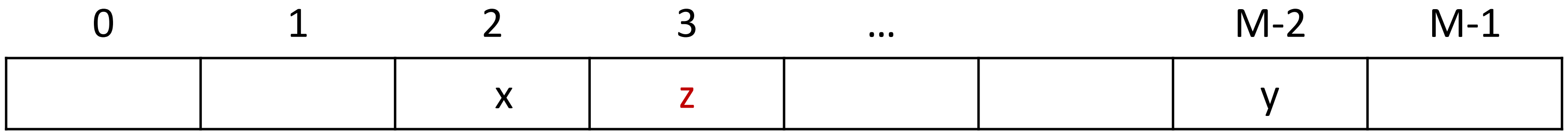
0	1	2	3	…		M-2	M-1
		x				y	

h(x) = 2
h(y) = m-2

Hashing basics

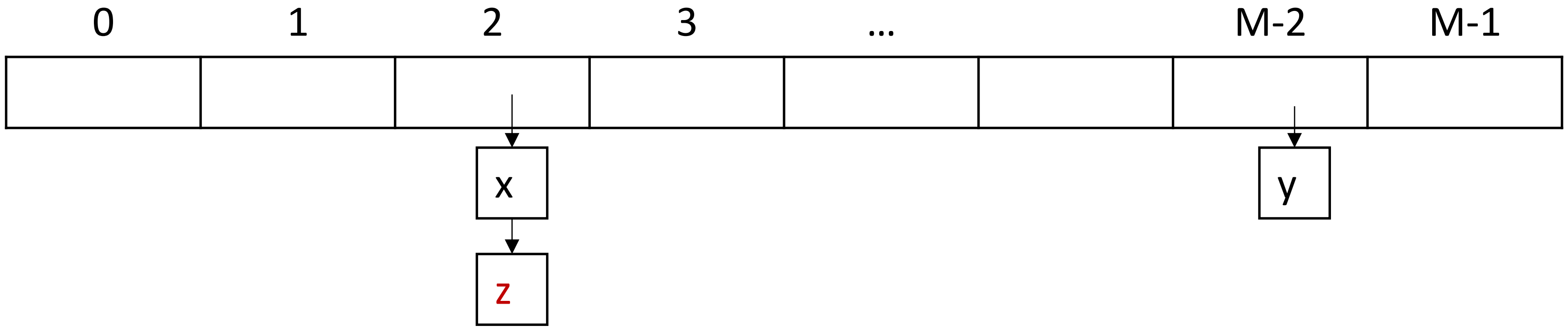
- (In-memory) hash table
  - With a hash function $h: U \to [M]$
  - How to handle collision?
    - Closed hashing vs open hashing
    - Sometimes also called open addressing vs closed addressing

closed hashing with linear probing



h(x) = 2
h(y) = M-2
h(z) = 2

open hashing with linked list

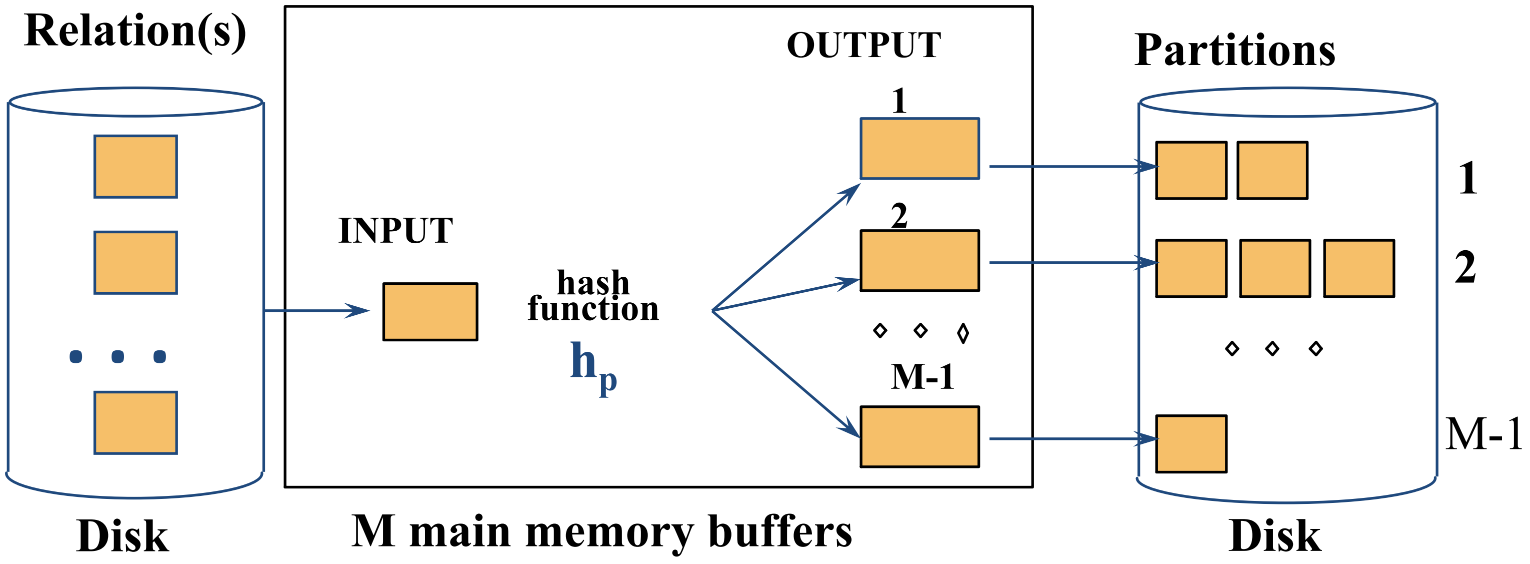


What might go wrong with hashing?

- Too many items with the same hash value
  - Any hash table design will fail in this case
- Why can that happen?
  1. Too many entries with the same key?
     - Not much that we can do, but we can try to incorporate other fields to make the keys distinct if it’s possible for the user to provide the entire key during lookups
     - Alternatively, consider using other types of index
  2. Hash collision
     - Some hash functions are prone to too many hash collisions
       - For instance, you’re hashing pointers of int64_t,
         - using modular hashing $h(x) = x \ mod \ m$ with $m = 2^d$ for some d is going to leave many buckets completely empty
- Desirable properties for good hash functions: small size, “uniform” over hash space, etc.
  - Examples include universal hashing, perfect hashing, …

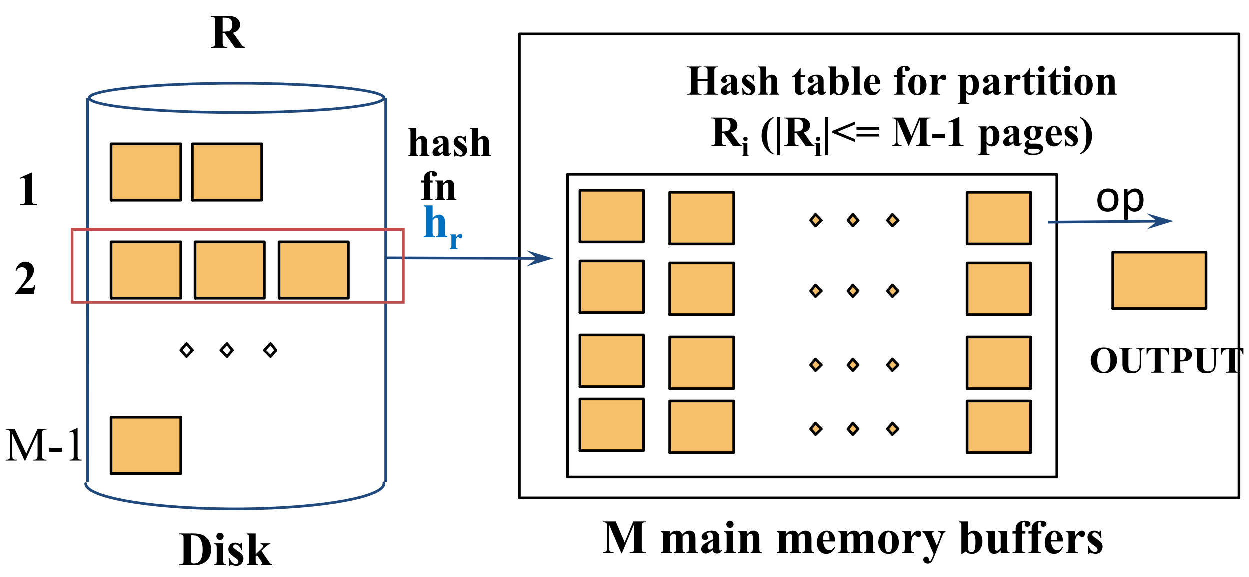
When hash table doesn’t fit in memory?

- Recursive partitioning
- Basic idea:
  1. Use partition hash function $h_p: \mathrm{K} \rightarrow [M]$ to partition data into M roughly equally-sized partitions



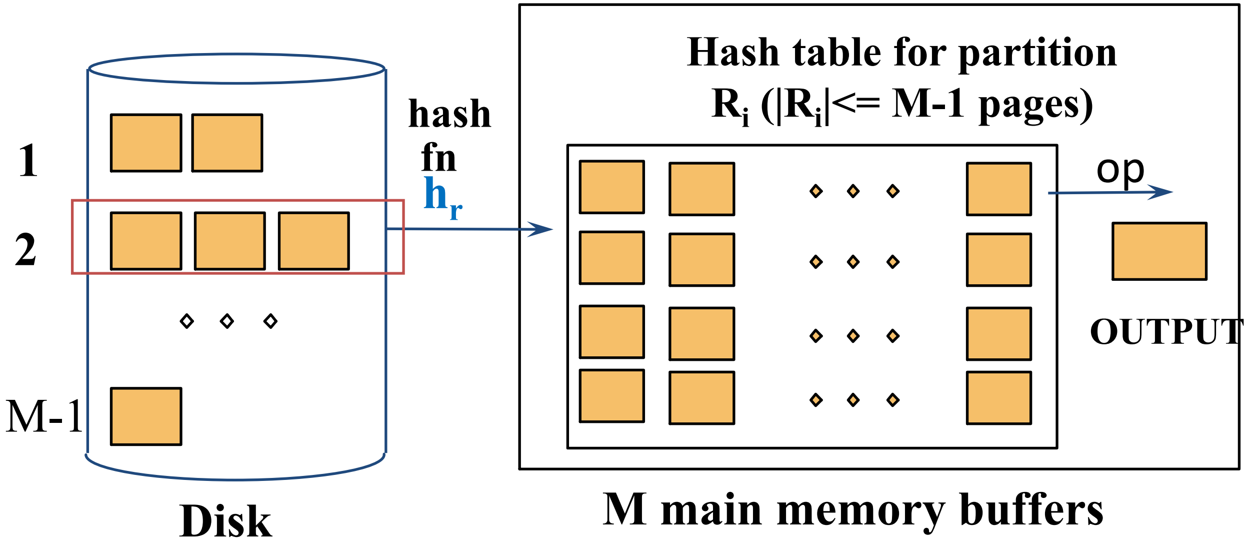
When hash table doesn’t fit in memory?

- Recursive partitioning
- Basic idea:
  1. Use partition hash function $h_p\colon \mathrm{K} \to [M]$ to partition data into M roughly equally-sized partitions
  2. For each partition, if it doesn’t fit in an in-memory hash table -> recurse to step 1.
     - Otherwise, build an in-memory hash table using a *different hash function* h_r and apply the desired operation



Example: deduplication

- Use the hash table as a hash set, i.e., no duplicates
- key = the entire tuple
- op :=
  - If key exists in HT, skip.
  - If key does not exist in HT,
    - copy it to output.



1
2
M-1
Disk
hash fn
h_r
Hash table for partition
R_i ($|R_i|$<= M-1 pages)
op
OUTPUT
M main memory buffers