# *Algorithms and Computability*

*Lecture 4:*
*External-Memory Algorithms*

SW6 spring 2025

*Simonas Šaltenis*

AALBORG UNIVERSITY
DENMARK

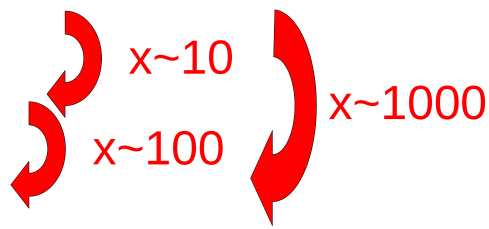
# External Mem. Algorithms and DS

- Goals of the lecture:
  - *to understand the external memory model and the principles of analysis of algorithms and data structures in this model;*
  - *(to understand the algorithms of B-tree and its variants and to be able to analyze them);*
  - *to understand the main principles of external tree structures;*
  - *to understand how the different versions of **merge-sort** derived algorithms work in external memory;*
  - *to understand why the amount of available main memory is an important parameter for the efficiency of external-memory algorithms.*
  - *Se how careful **algorithm engineerin**g can improving running time in practice*

# Memory hierarchy, prices

- In 2021, people created ~2.5 exabytes (million TBs) *per day*!
  - Where do we store that data?

- Prices:
  - HDD price: ~0.02 $/GB
  - SSD price: ~0.15 $/GB
  - DRAM price: ~5-10 $/GB

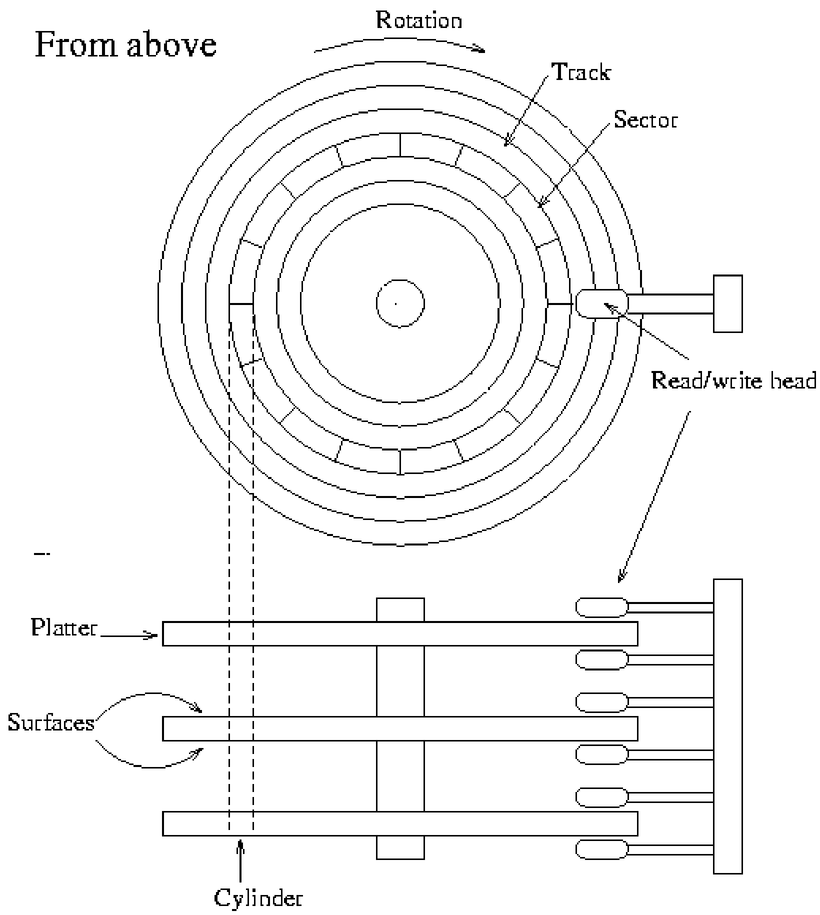



- *Memory-hierarchy is still very relevant in the age of big data!*

Sources: https://techjury.net/, https://pcpartpicker.com/

# Hard disk I

- In *real systems*, we need to cope with data that does not fit in main memory
- Reading a data element from the hard-disk:
  - *Seek* with the head
  - *Wait* while the necessary sector rotates under the head
  - *Transfer* the data



# Hard disk II

- Modern hard drives:
  - *Seek time:* 4ms-10ms
  - *Spindle speed:* ~10K RPM ⇒ *Half of rotation:* ~3ms
  - *Transfer rate:* 500 MB/s ⇒ *Transferring 1 byte:* 0.000003ms

- Conclusions:
  1. It makes sense to *read and write in large blocks* – *disk pages* (4 – 32Kb)
  2. *Sequential* access is much faster than *random* access
  3. Disk access is much slower than main-memory access

# SSDs, Memory Hierarchy

- The same, although to less extent is true for *flash*-based *solid state drives* (SSDs):
  - Efficient to read/write (especially write) in larger blocks
  - Sequential/random I/O difference is less pronounced than in disks.
- Depth of the memory hierarchy (access latency):
  - DRAM(~50ns) – x4000 → SSD(~0.2ms) – x50 → HDD(10ms)

    *If = 1s, then > 1 hour, > 2 days*

- Memory hierarchy consisting of several levels of *CPU caches* and *DRAM:*
  - Again, data between levels is transferred in *blocks*
  - In contrast to disk drives and SSDs, block reads and writes are not explicit – controlled by hardware/low level system software

# External memory model

- Running time: in *page accesses* or "I/Os"
- *B* – page size is an important parameter:
  - Not "just" a constant:
    - $\Theta(log_2 n) \neq \Theta(log_B n)$
    - $\Theta(N) \neq \Theta(N/B)$
    - *Example*: $N$ = 256MB / 8 bytes_per_object; $B$ = 4KB / 8 bytes_per_object; 0.1 ms disk access
      - $N$ disk accesses = 3200s = 53 minutes
      - $N/B$ disk accesses = 6.4s

- Operations:
    - DiskRead(x:pointer_to_a_page)
    - DiskWrite(x:pointer_to_a_page)
    - AllocatePage():pointer_to_a_page

# Writing algorithms

- The typical working pattern for algorithms:

```
…
x ← a pointer to some object
DiskRead(x)
operations that access and/or modify x
DiskWrite(x) //omitted if nothing changed
other operations, only access no modify
…
```

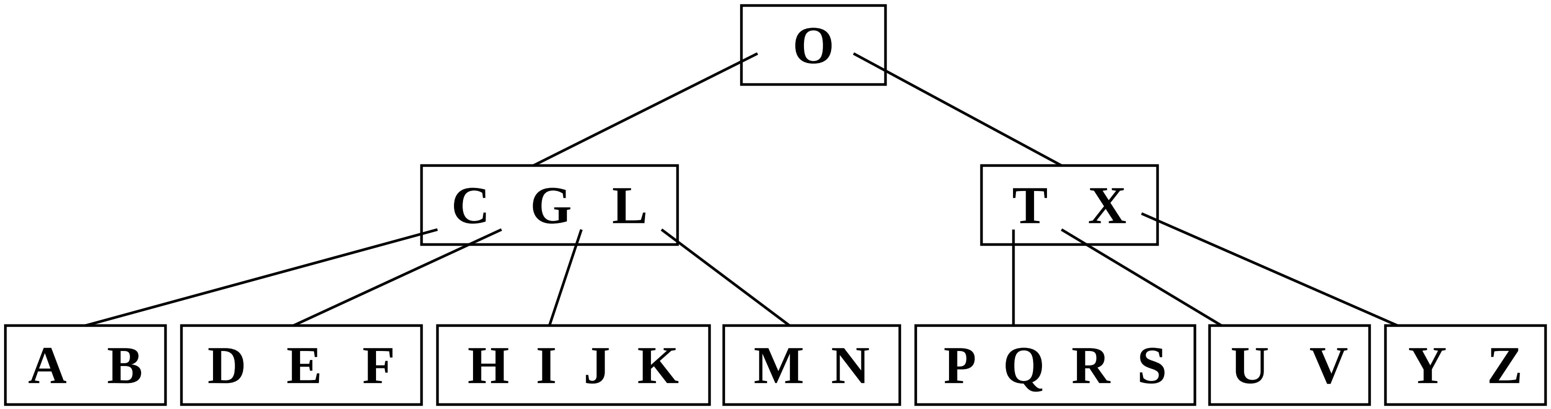
- Pointers in data-structures point to disk-pages, not locations in memory

# “Porting” main-memory DSs

- Why not “just” use the main-memory data structures and algorithms in external memory?
- Consider a balanced binary search tree.
  - *A, B, C, D, E, F, G, H, I*
- Options:
  - Each node gets a separate disk page – waist of space and search is just $\Theta(log_2 N)$
  - Nodes are somehow packed to make disk pages full – search may still be $\Theta(log_2 N)$ in the worst-case

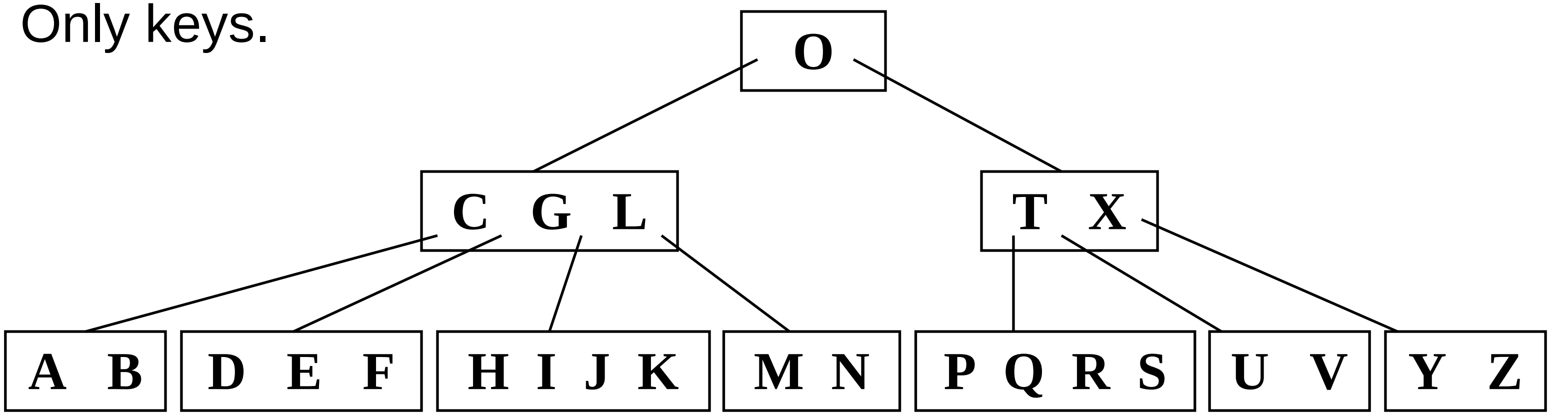
# B-trees

- We are concerned only with keys
- The nodes have high *fan-out* (many children) = $\Theta(B)$
  - *Degree* of a tree *t:*
    - *Min_fan-out = t,  Max_fan-out = 2*t = B / index_entry_size*
  - Root is the exception: can have as little as two children
- B-tree is a balanced tree, and all leaves have *the same depth*: $h = \Theta(\log_t N) = \Theta(\log_B N)$



# B-trees, nodes

- Internal nodes
  - $t$ - 1 *to* 2$t$ - 1 keys
  - pointer$_1$ key$_1$ pointer$_2$ key$_2$ pointer$_3$ key$_3$ … pointer$_x$ key$_x$ pointer$_{x+1}$
  - key$_1$ ≤ key$_2$ ≤ key$_3$ ≤ … ≤ key$_x$
  - For the first and last pointers: pointer$_1$ .key ≤ key$_1$
  - ...and key$_x$ < pointer$_{x+1}$.key
  - For the remaining pointers: key$_{i-1}$<pointer$_i$.key ≤ key$_i$
- Leave nodes
  - Only keys.



# Searching in B-trees

- The root node is normally "always" in main memory.
  - No need to perform a DiskRead on the root.
- Search is very similar to a search in a binary search tree
  - Instead of making a binary branching decision at each node, we make a *(j+1)*-way branching decision, where *j* is the number of keys in a node.

# Pseudo code

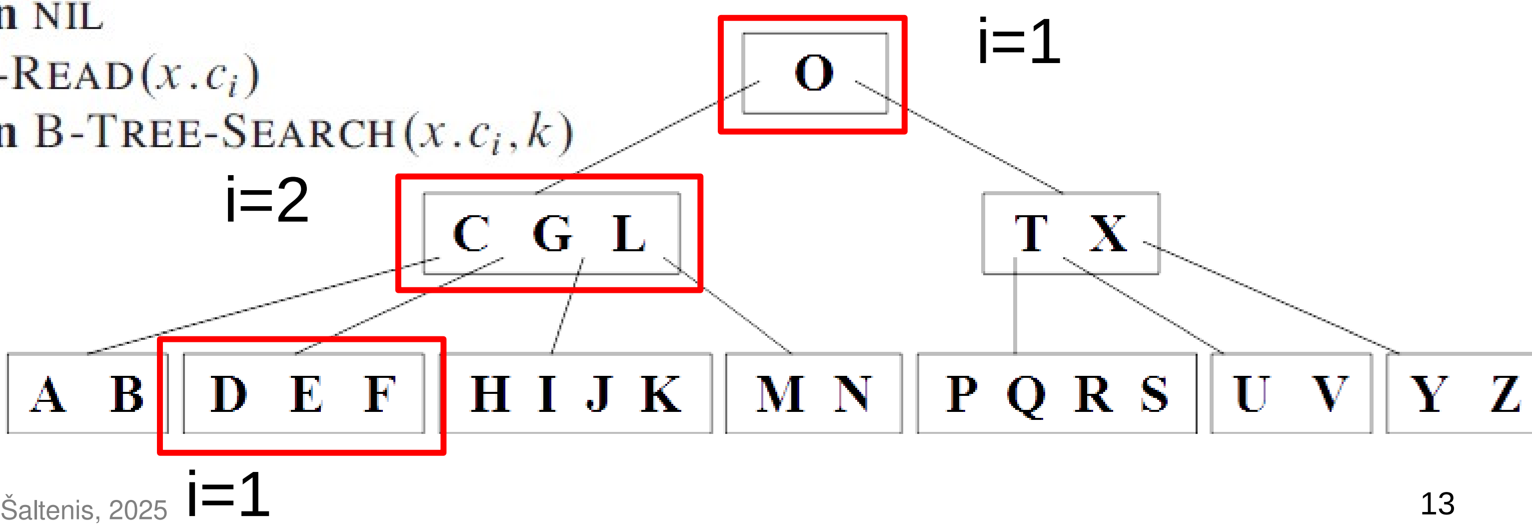
- x is a node and x.n is the number of keys in the node.
- k is the key that we are searching for.
- $x.key_i$ is the i-th key of node x; and $x.c_i$ is the i-th pointer of node x.

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

Searching for "D", i.e., k = D
B-Tree-Search(root, D)

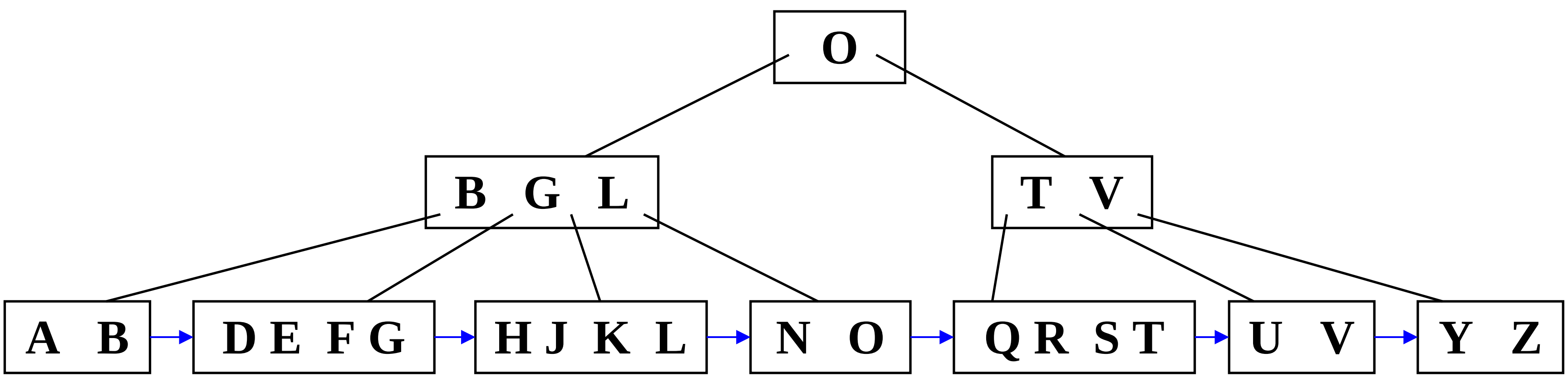
Disk access: $O(h)=O(\log_t N)$

CPU: $O(th)=O(t \log_t N)$



# B$^+$-trees

- B$^+$-trees is a variant of B-trees:
  - All data keys are in leaf nodes
    - *What is the height?*
  - Leaf-nodes are connected into a (doubly) linked list
  - Search is very similar to a search in a binary search tree
    - Always goes to a leaf
    - Range searches are convenient
    - Cost: $\Theta(\log_B N + k/B)$



# Some questions regarding B+-trees

- *The length of all root-leaf paths in a B+-tree is the same?*
  - True/False

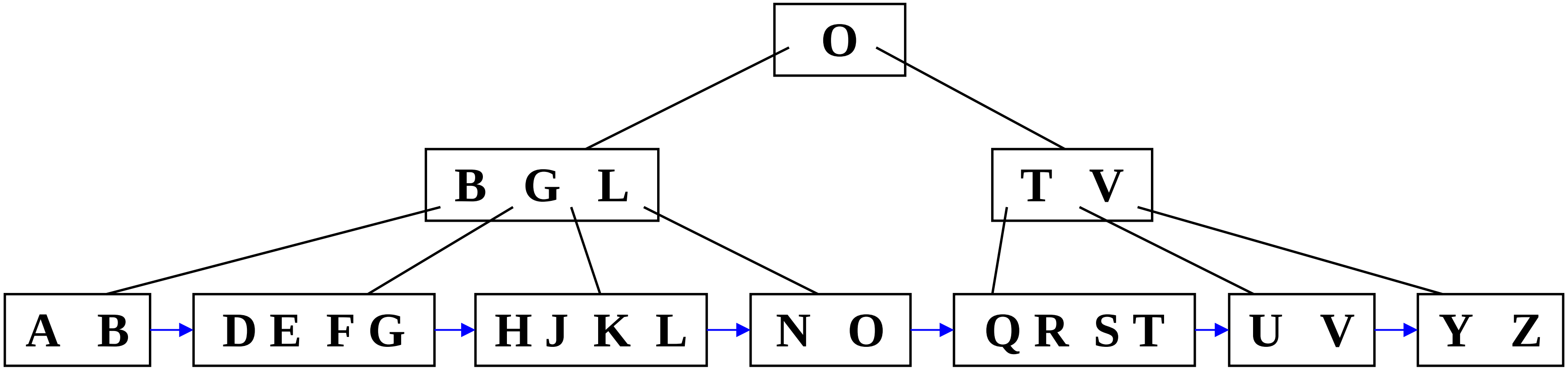
- *The B+-tree grows in height:*
  - A: at the root
  - B: at leaves

- *The number of node splits in a B+-tree insertion is:*
  - A: 0 or 1
  - B: always 1
  - C: [0 .. $\log_B N$]
  - D: [1 .. $\log_B N$]
  - E: other

- *Go to Socrative and vote*

# B$^+$-trees: Insertion

- Skeleton of the algorithm:
  - *Down-phase:* recursively traverse down and find the leaf (as in search)
  - *Up-phase:* Insert the key. If necessary, *split* nodes and propagate the splits up the tree
- Assumption:
  - In the *down-phase* pointers to traversed nodes are saved in the stack as there are *no parent pointers*!
- Insert M:



# Insertion cost

- *What is the cost of insertion?*
  - $\Theta(log_B N)$
- *How much memory is used*?
  - $\Theta(log_B N)$ – can be reduced to $\Theta(1)$: split full nodes while going down!

# $B^+$-trees: Deletion

- First:
  - *Why parent pointers are usually not used in B-trees, in contrast to binary search trees*?
- Deletion – opposite of insertion:
  - Phase 1: traverse down to find the key in a leaf
  - Phase 2: remove the key and traverse up handling underfull nodes
- Tree shrinking: if the root has only one child, remove the root.

# External DS: Summary

- Two practical data structures:
  - **B-trees** and **$B^+$-trees**: supports point and range queries, insertions, deletions
    - Point query: $\Theta(log_B N)$
    - Range query: $\Theta(log_B N + k/B)$
    - Insertion, deletion: $\Theta(log_B N)$
  - Both structures have $\Theta(N/B)$ size
  - *So what are the main differences between main-memory and external data structures?*

# External-memory Algs: notation

- Assumptions and notation:
  - Disk page size:
    - $B$ data elements
  - Data file size:
    - $N$ elements, $n = N/B$ disk pages
  - Available main memory:
    - $M$ elements, $m = M/B$ pages

# Warm-up example

- *Simple problem: print all duplicates in a file*
  - Conditions: in place; order to be preserved
  - Main-memory solution: nested-loop algorithm. *Complexity*?

- *How do we port it to external memory?*

```
Print-Duplicates(X)
for i = 1 to NumPages(X) by l
   DiskRead(Bf1, X, i, l)
   for j = i+l to NumPages(X) by m-l
      DiskRead(Bf2, X, j, m-l)
      for each e ∈ Bf1:
         print e if e ∈ Bf2
```

- *What is the running time?*
- *Depends on how we use memory*
  - Efficient way to use it: $m$ – 1 pages for $Bf_1$, and 1 page for $Bf_2$
  - $\Theta(n^2/m)$ *I/Os*

# External-Memory Sorting

- External-memory algorithms
  - When data do not fit in main-memory
- External-memory sorting
  - Rough idea: sort pieces that fit in main-memory and “merge” them
- Main-memory merge sort:
  - The main part of the algorithm is Merge
  - Let’s merge:
    - 3, 6, 7, 11, 13
    - 1, 5, 8, 9, 10

# Main-Memory Merge Sort

```
Merge-Sort(A)
if length(A) > 1 then
   Copy the first half of A into array A1
   Copy the second half of A into array A2
   Merge-Sort(A1)
   Merge-Sort(A2)
   Merge(A, A1, A2)
```

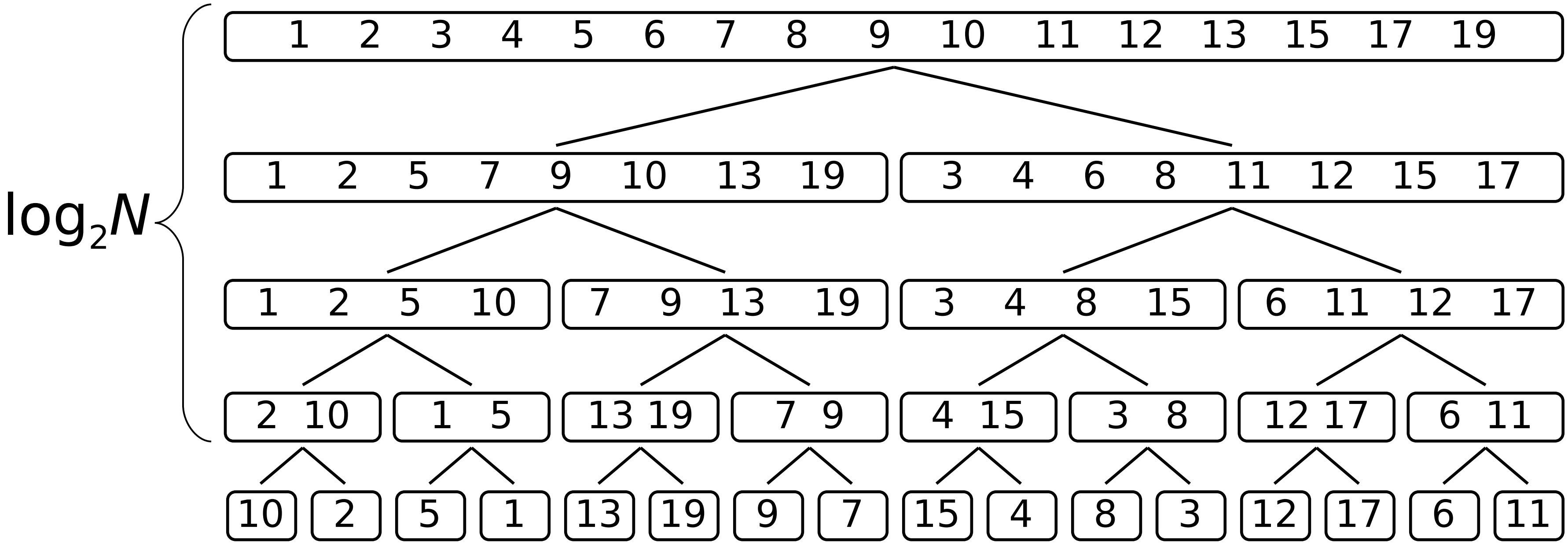
*Divide*

*Conquer*

*Combine*

- Running time?

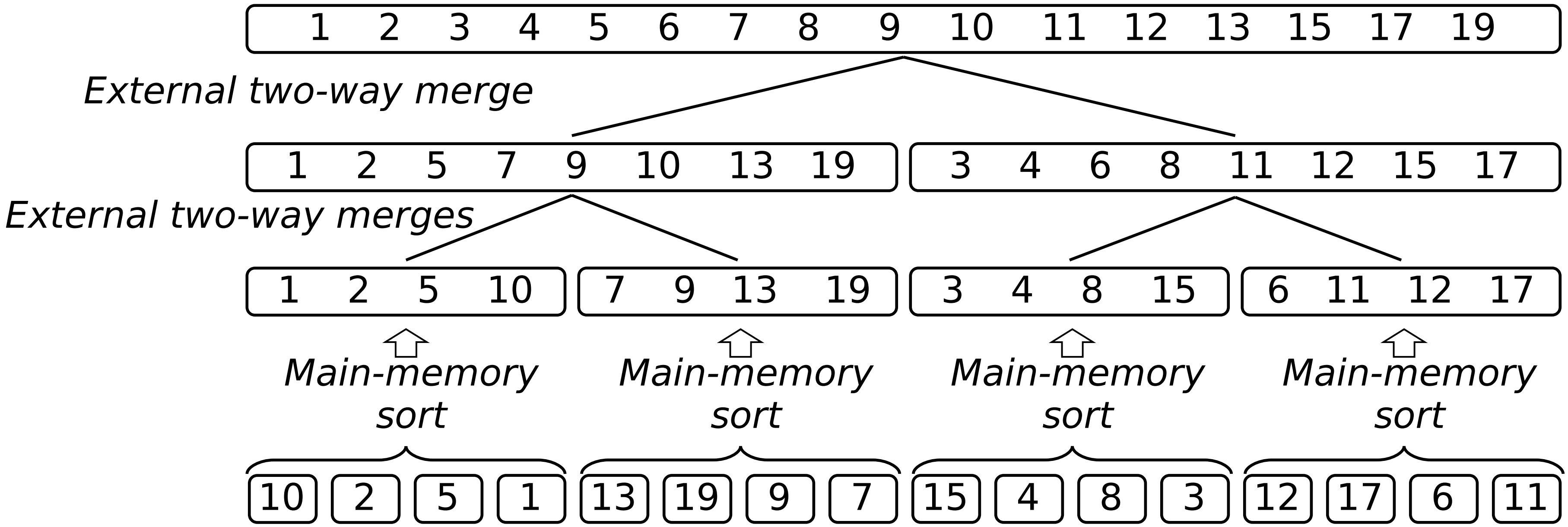
# Merge-Sort Recursion Tree




- In each level: merge *runs* (sorted sequences) of size *x* into runs of size 2*x*, decrease the number of runs twofold.
- What would it mean to run this on a file in external memory?

# External-Memory Merge-Sort

- Idea: increase the size of initial runs!
  - Initial runs – the size of available main memory (*M* data elements)



# External-Memory Merge Sort

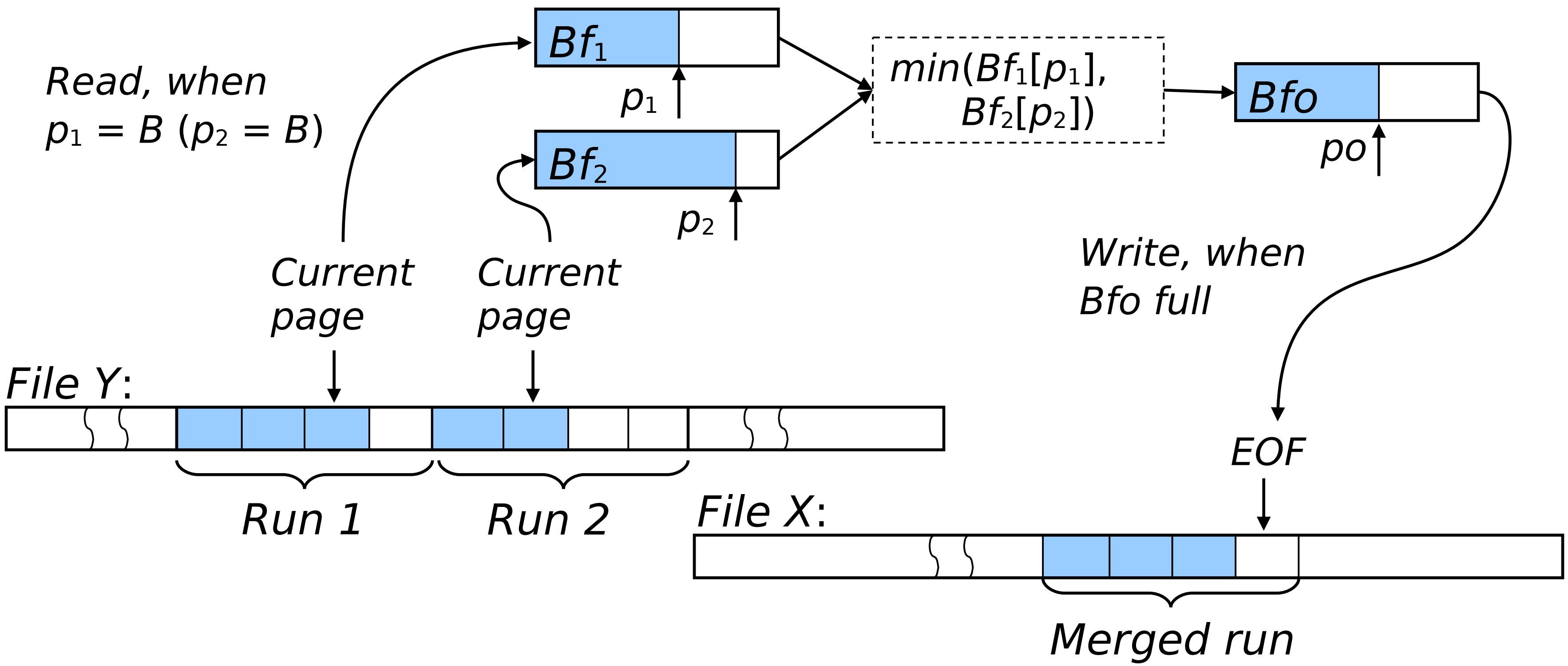
- Input file *X*, empty file Y

- *Phase* 1: Repeat until the end of file *X*:
  - Read the next *M* elements from *X*
  - Sort them in main-memory
  - Write them at the end of file *Y*

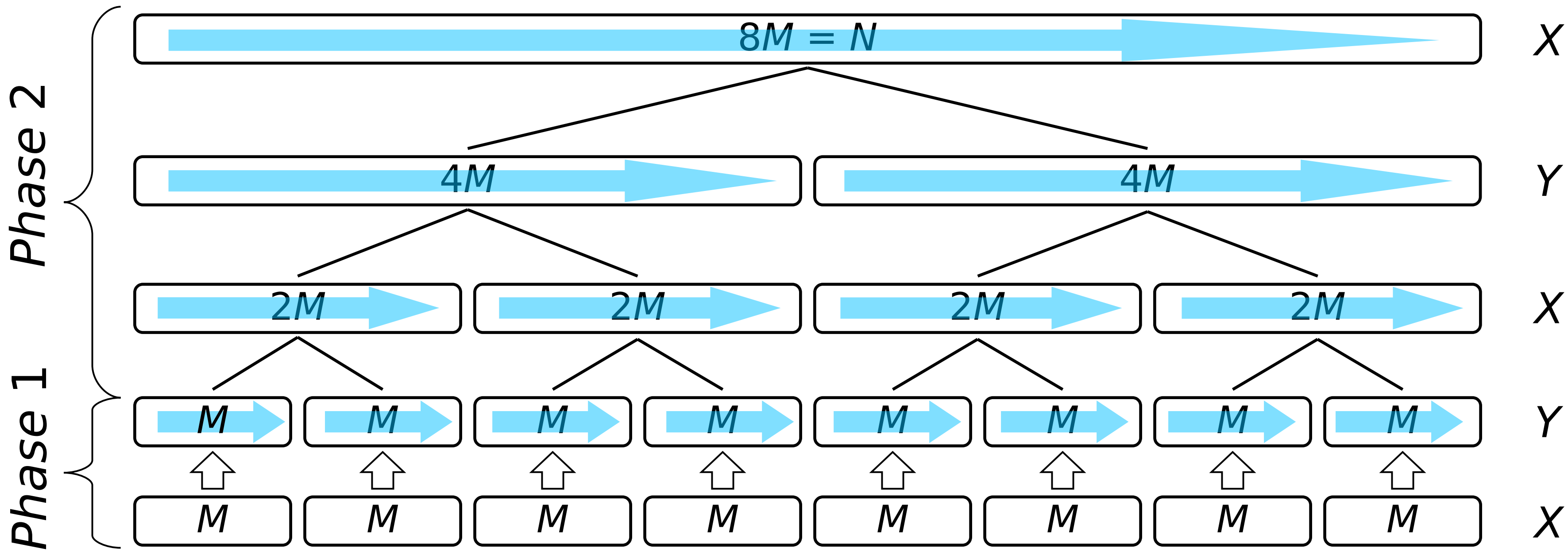
- *Phase* 2: Repeat while there is more than one run in *Y:*
  - Empty *X*
  - *MergeAllRuns*(*Y*, *X*)
  - *X* is now called *Y*, *Y* is now called *X*

# External-Memory Merging

- *MergeAllRuns*(*Y*, *X*): repeat until the end of *Y*:
  - Call *TwowayMerge* to merge the next two runs from *Y* into one run, which is written at the end of *X*
- *TwowayMerge*: uses three main-memory arrays of size *B*



# Analysis



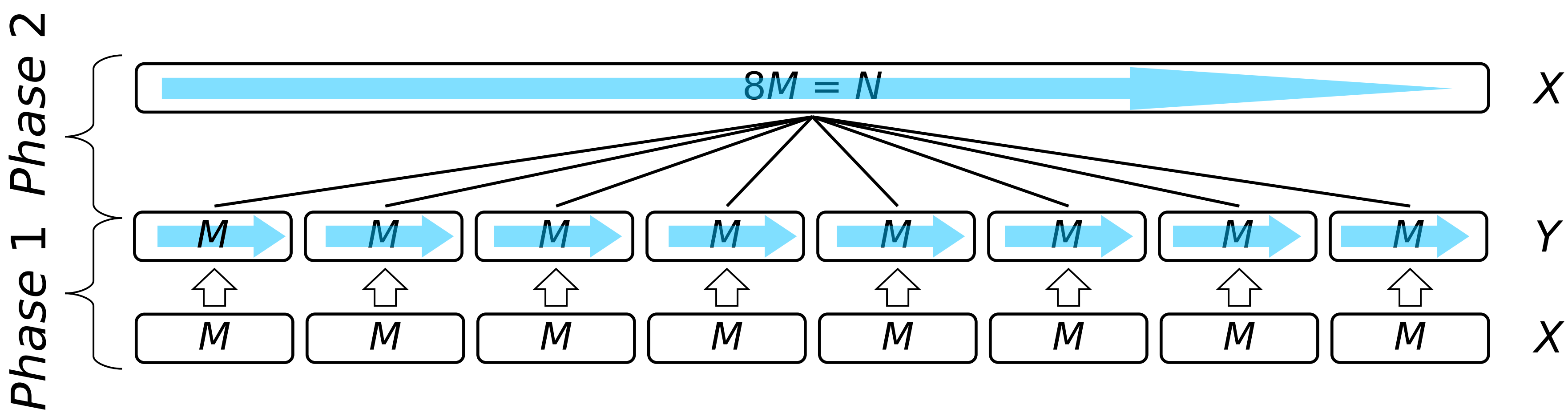

- Phase 1:
  - Read file X, write file Y: 2$n$ = Θ($n$) I/Os
- Phase 2:
  - One iteration: Read file Y, write file X: 2$n$ = Θ($n$) I/Os
  - Number of iterations: $\log_2 N/M = \log_2 n/m$

# Analysis: Conclusions

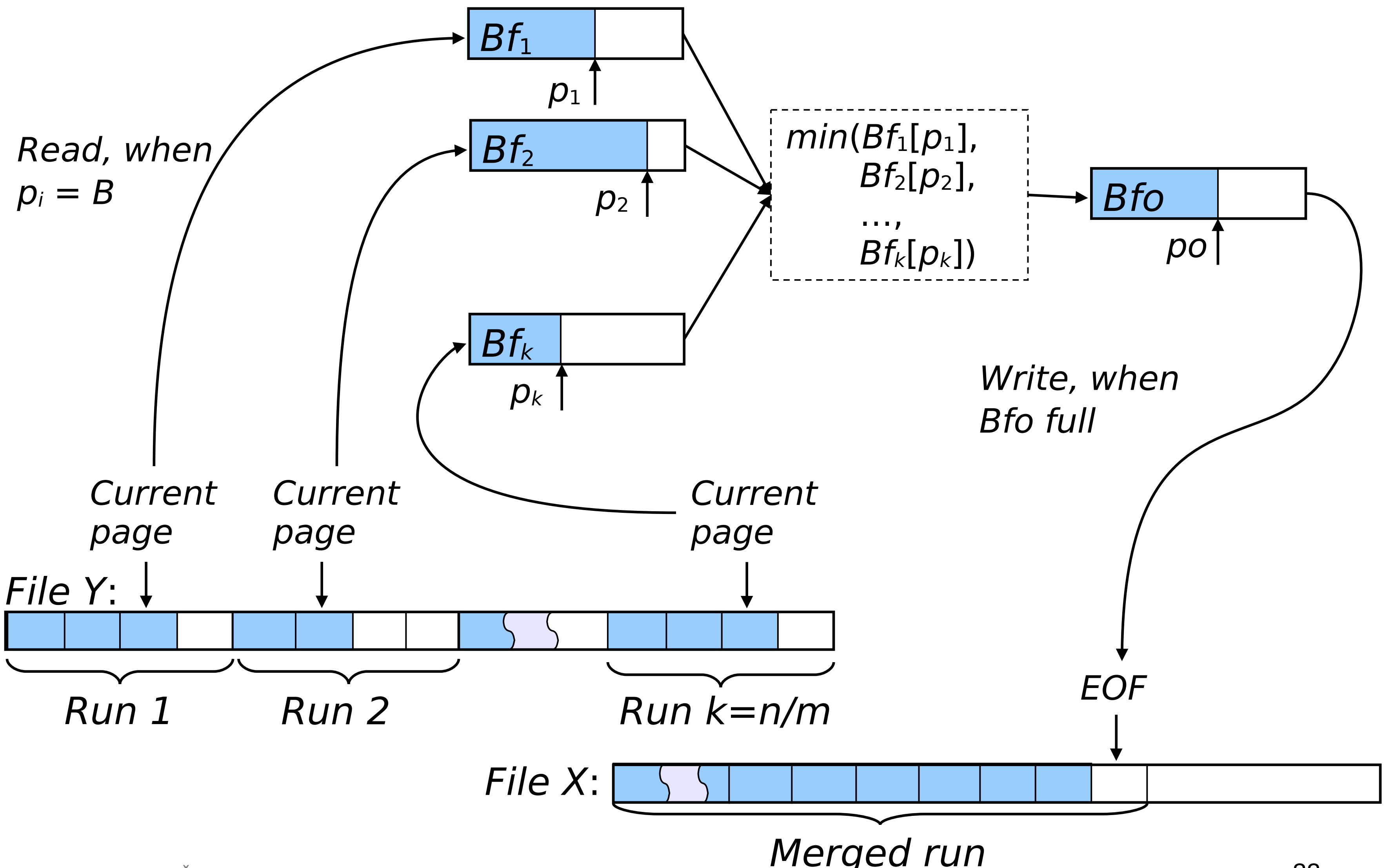
- Total running time of external-memory merge sort: $\Theta(n \log_2 n/m)$
- We can do better!
- Observation:
  - Phase 1 uses all available memory
  - Phase 2 uses just 3 pages out of *m* available!!!

# Two-Phase, Multiway Merge Sort

- Idea: merge all runs at once!
  - Phase 1: the same (do internal sorts)
  - Phase 2: perform *MultiwayMerge*(*Y*,*X*)



# Multiway Merging



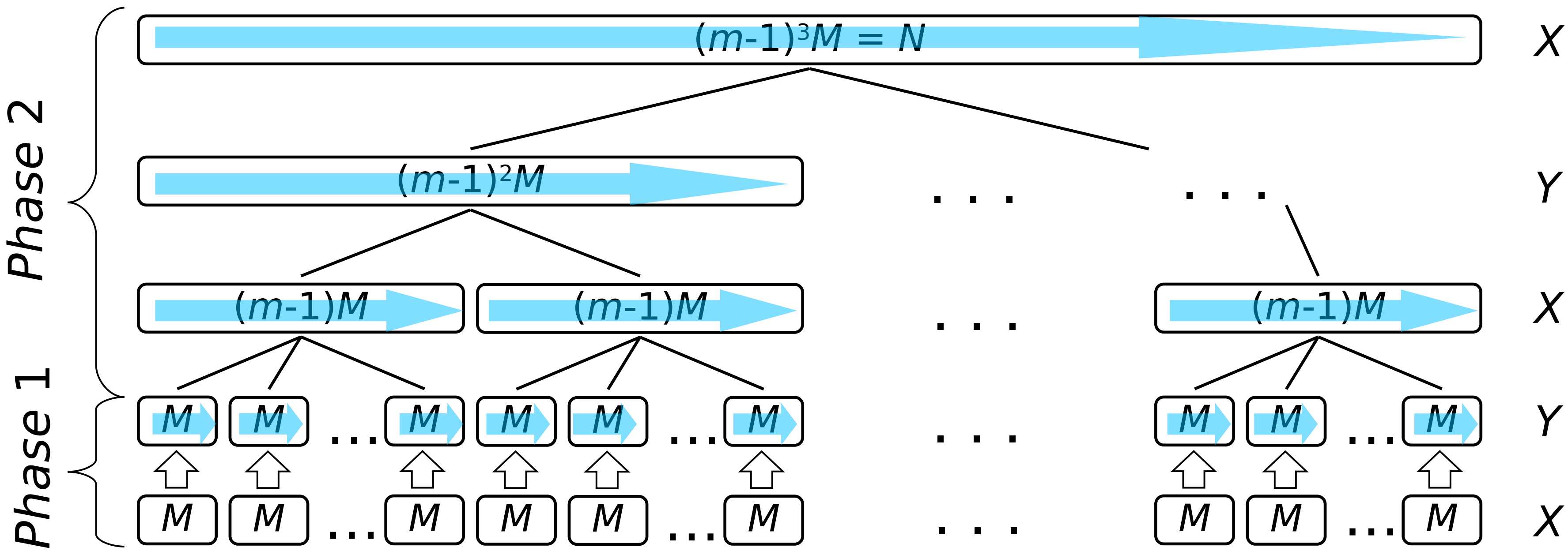
# Analysis of TPMMS

- Phase 1: $\Theta(n)$, Phase 2: $\Theta(n)$
- Total: $\Theta(n)$ I/Os!
- The catch: files only of “limited” size can be sorted
  - Phase 2 can merge a maximum of $m$-1 runs.
- Which means: $N/M \leq m - 1$ ($n/m \leq m-1$)
  - *How large files can we sort with TPMMS on a machine with 128MiB main memory and disk page size of 16KiB?*

# General Multiway Merge Sort

- What if a file is **very** large or memory is small?
- General *multiway merge sort*:
  - Phase 1: the same (do internal sorts)
  - Phase 2: do as many iterations of merging as necessary until only one run remains
    - Each iteration repeatedly calls *MultiwayMerge*(*Y, X*) to merge groups of *m-1* runs until the end of file *Y* is reached

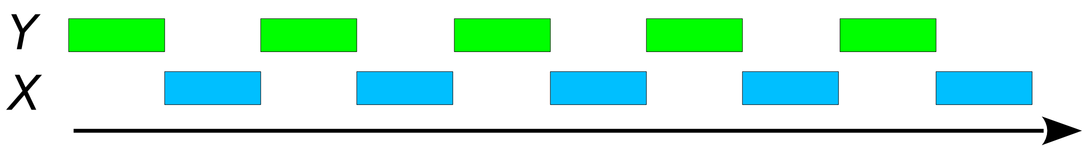
# Analysis




- Phase 1: $\Theta(n)$, each iteration of phase 2: $\Theta(n)$
- How many iterations are there in phase 2?
  - Number of iterations: $\log_{m-1} N/M = \Theta(\log_m n)$
- Total running time: $\Theta(n \log_m n)$ I/Os

# Algorithm engineering ideas

- Often *two disks* are available: for file *Y* (*reading*) and for file *X* (*writing*).
  - External multiway merge sort is *I/O bound* – most of the time waiting for an I/O operation to finish.
  - With two disks – the disks are idle half of the time.
    - A read/write I/O waits for another read/write I/O to finish.

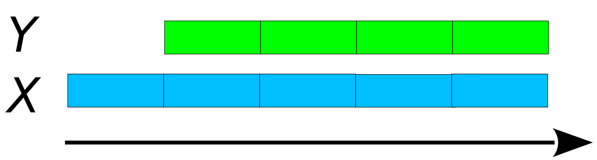



  - With equal amounts of reads and writes, we want both disks to be busy all the time (in both phases of the algorithm)

# Parallel reading and writing

- Phase 2 (merging): increase the input and output buffers to *two pages.*
  - One page can be read/written, while the other is being processed/filled.
  - Reads wait only for other reads, writes wait only for other writes.

- Phase 1 (RAM sorting):
  - Start reading the new run as soon as the first page of the current run is written.




  - With an appropriate RAM sorting algorithm, this can be started while the sorting of a run is ongoing. For example, *heapsort* forms a sorted sequence of the smallest elements at the beginning.
    - A soon as one page of the smallest elements is ready, it can be written to output. Then, a page of a new (unsorted) run can be read immediately in its place and so on.

# Replacement selection

- In phase 1:
- We can keep those elements from the newly read page in the current run that are larger than the largest element we have written to disk in the current run!
  - Can be shown that it allows to extend the size of initial runs two times on average.

# Conclusions

- External sorting can be done in $\Theta(n \log_m n)$ I/O operations for any $n$
  - This is asymptotically optimal
- In practice, we can usually sort in $\Theta(n)$ I/Os
  - Use two-phase, multiway merge-sort