Algorithms and Satisfiability

Lecture 4:
External-Memory Algorithms and Data Structures

DAT6 spring 2023

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.*

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

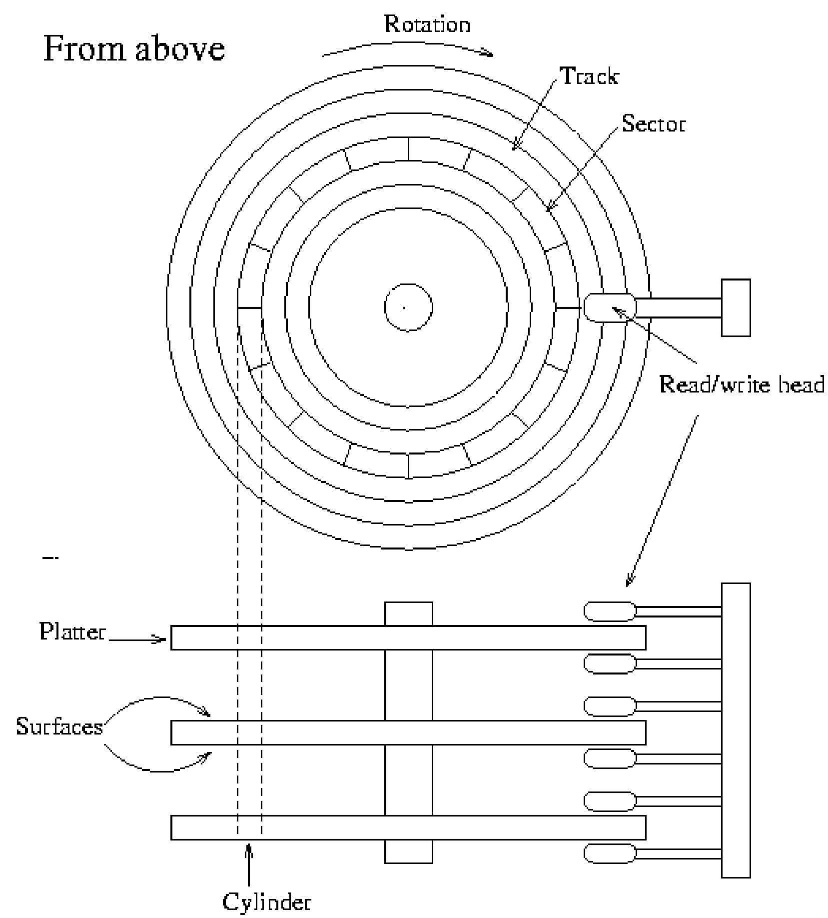
x~10
x~100
x~1000

- *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
…
```

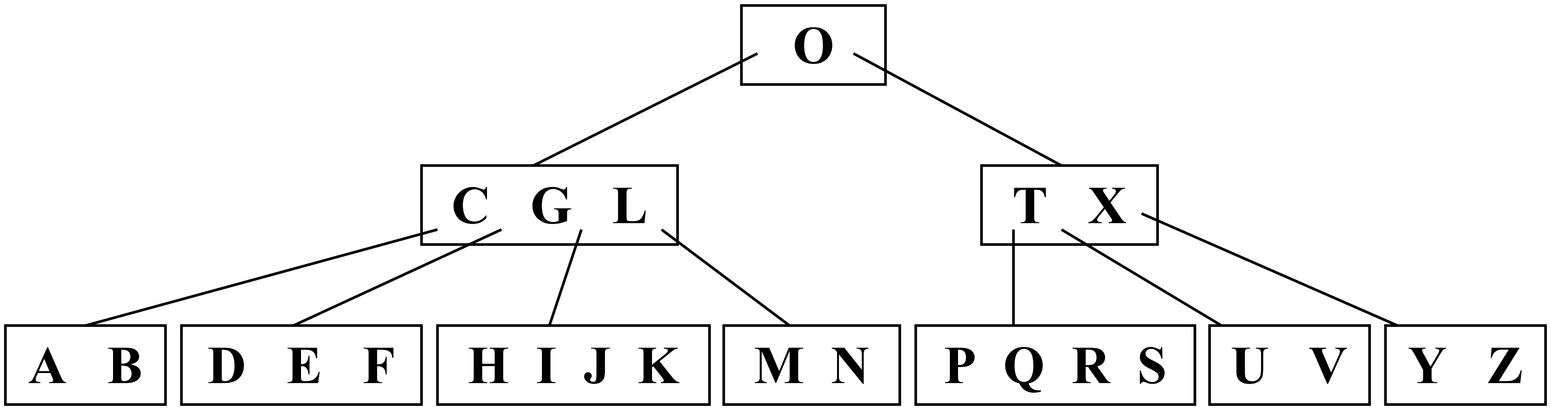
- Pointers in data-structures point to disk-pages, not locations in memory

“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

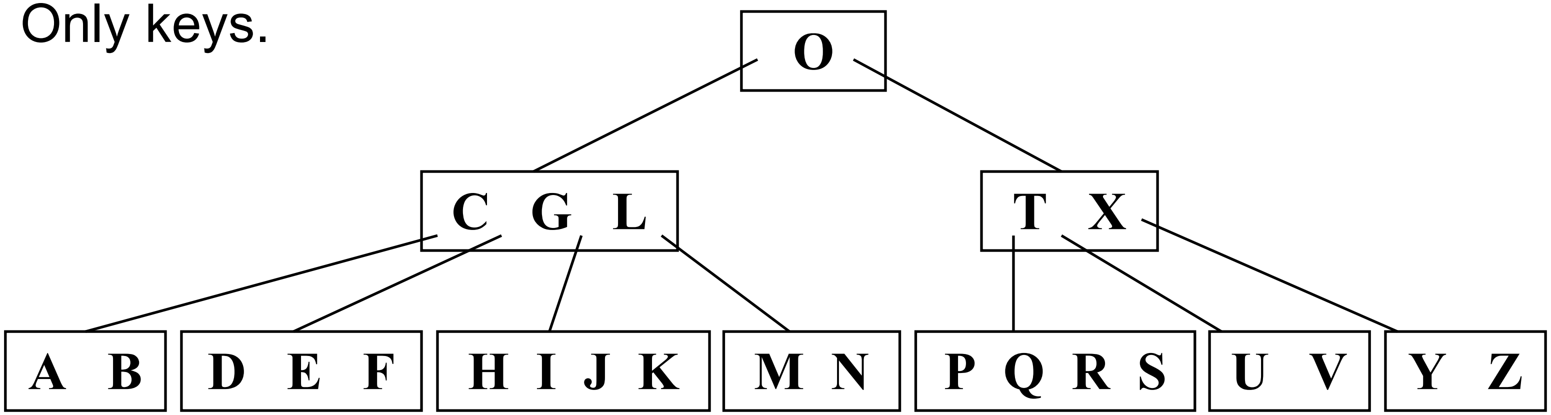
B-trees

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



Searching on 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

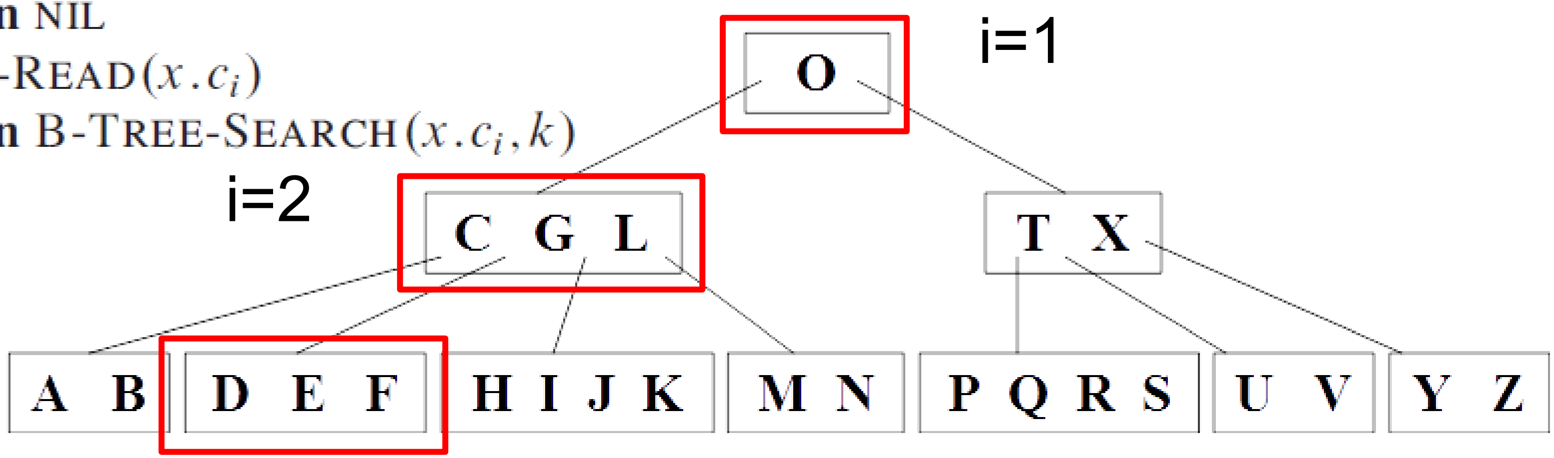
- x is a node and x.n is the number of keys in the node.
- k is the key that we are searching for.
- $x.key_i$ is the i-th key of node x; and $x.c_i$ is the i-th pointer of node x.

```
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)

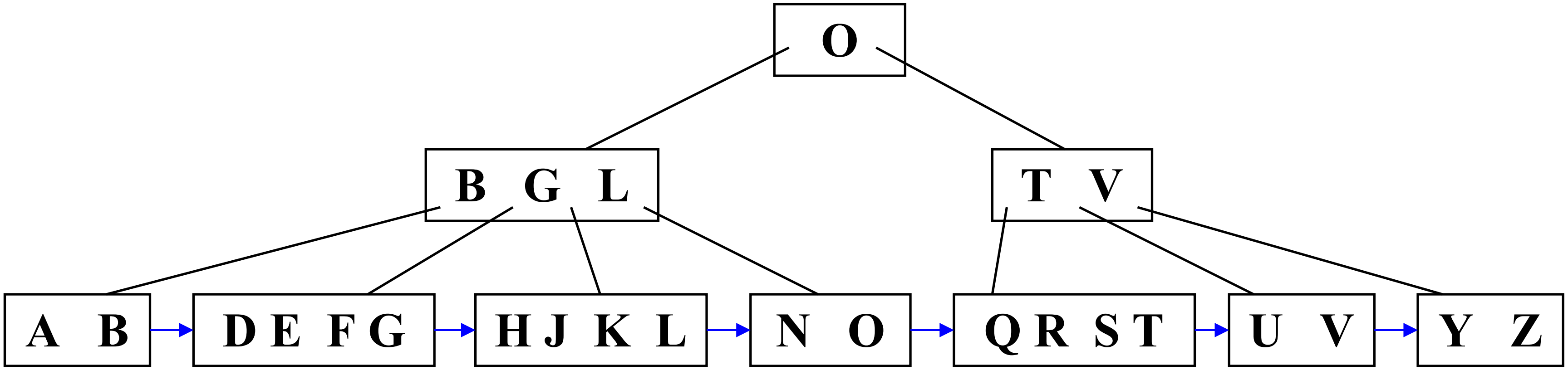
Disk access: $O(h)=O(\log_t n)$

CPU: $O(t\ h)=O(t \log_t n)$



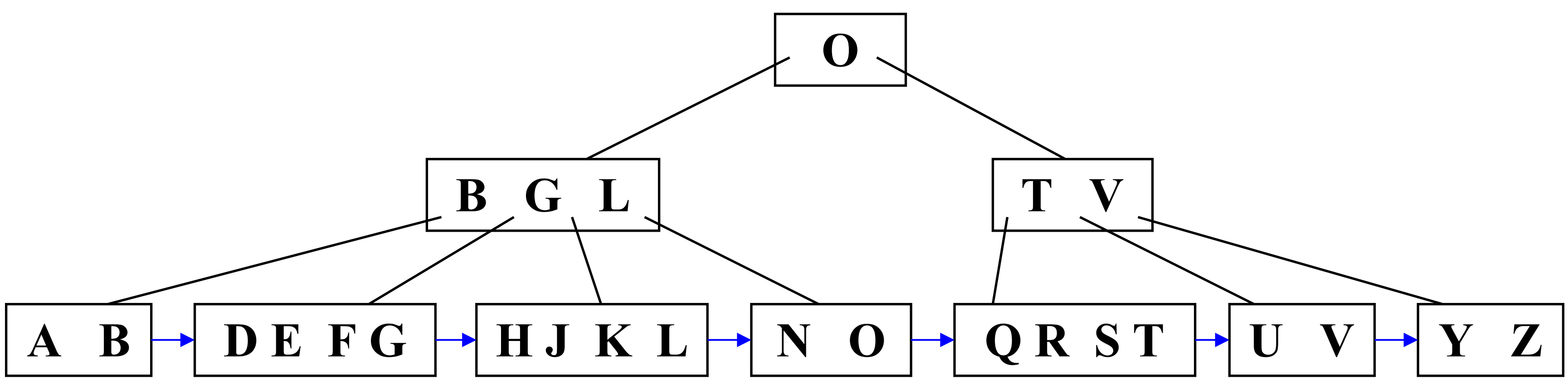
B^+-trees

- 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: $\Theta(\log_B n + k/B)$



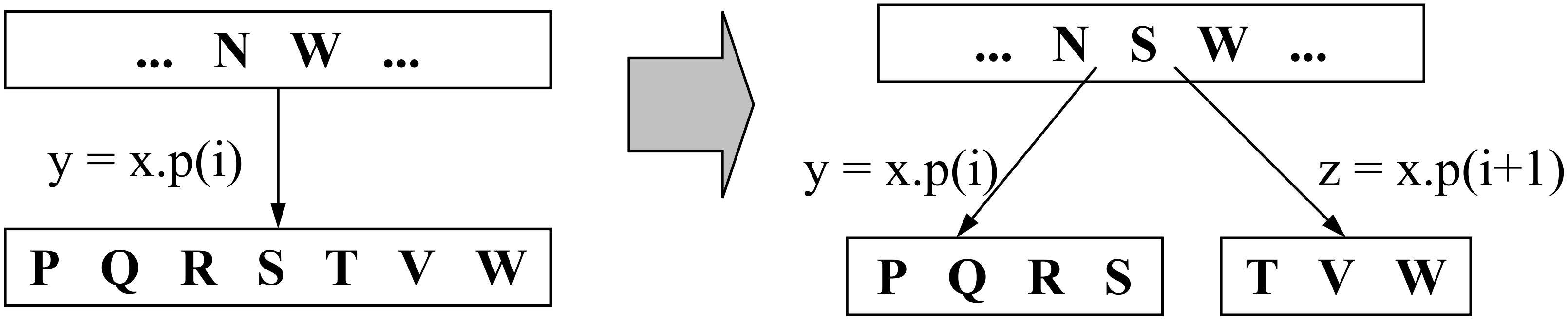
B^{+}-trees: Insertion

- 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:

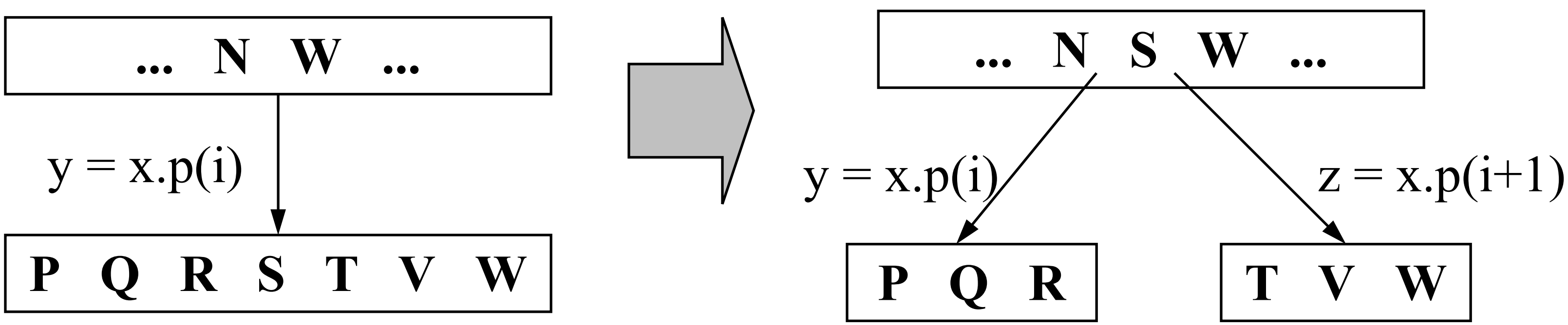


B+-trees: Node Splitting

- Leaf node (*copy* the middle key to the parent)



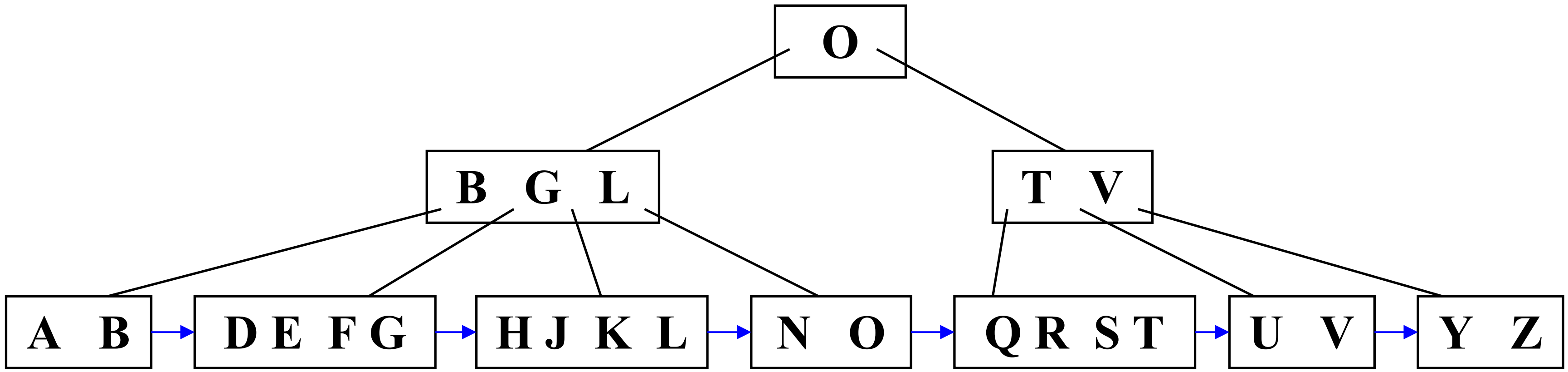
- Internal node (*move* the middle key to the parent, as in B-tree)



- The tree grows when the root is split into two nodes and their parent becomes the new root.

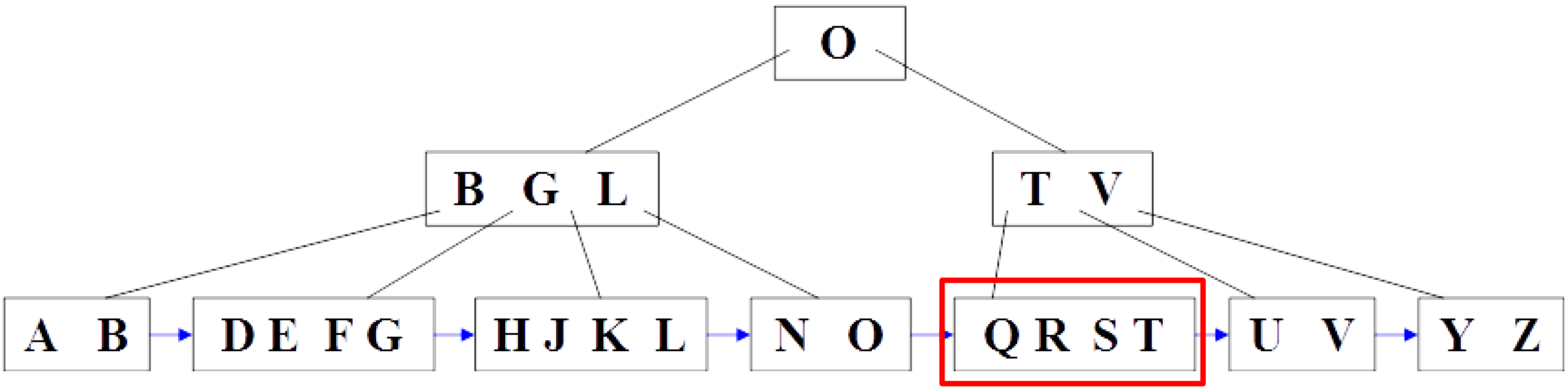
B^+-trees: Example

- *Insert P* (maximum fan-out = 5 children = 4 data entries)
- *What is the cost?*
  - $\Theta(log_B n)$
- *How much memory is used*?
  - $\Theta(log_B n)$ – can be reduced to $\Theta(1)$.

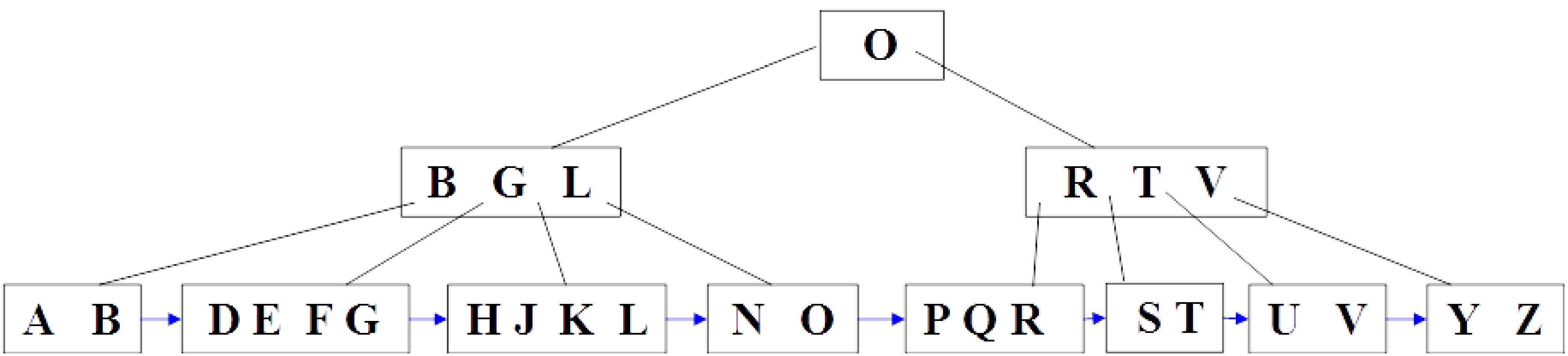


Example

- *Insert P* (assume that at most 4 keys)

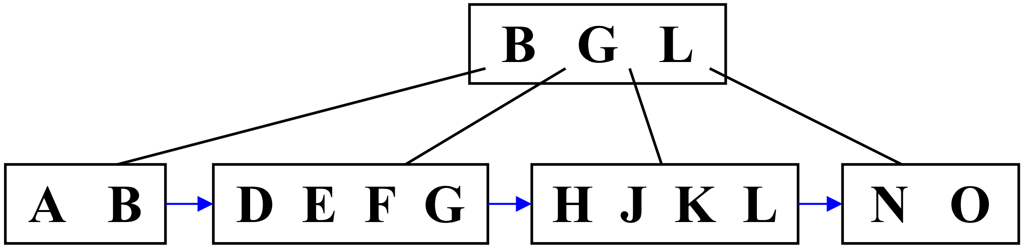


Down-phase: Leaf node
(**copy** the middle key to the parent)



B^+-trees: insert excercise

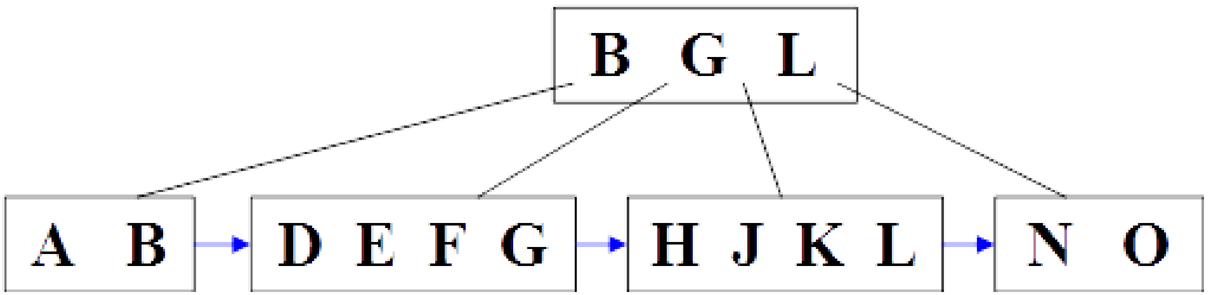
- Excercise:
  - Assume maximum fan-out = 5 children = 4 data entries
  - *Insert I, C*



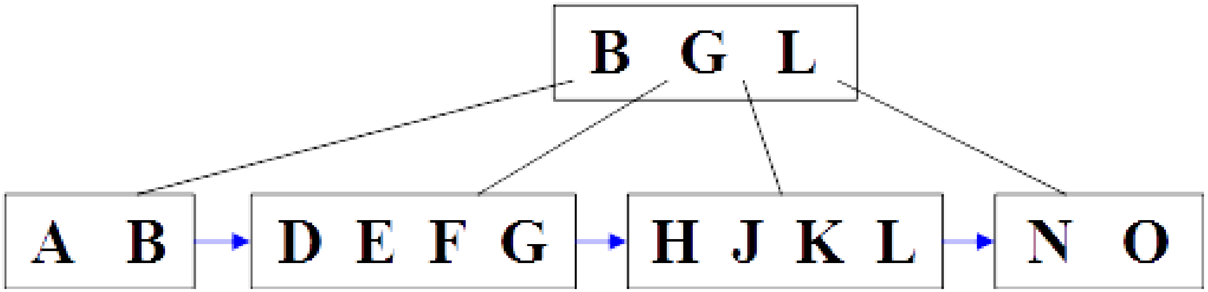
- *So, why parent pointers are usually not used in B-trees, in contrast to binary search trees?*

Example

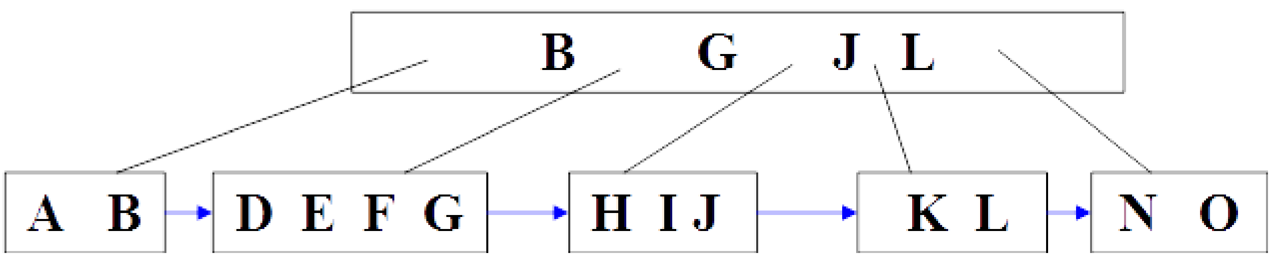
- At most 4 keys
- After inserting I, C
- How does the root node look like?



Example

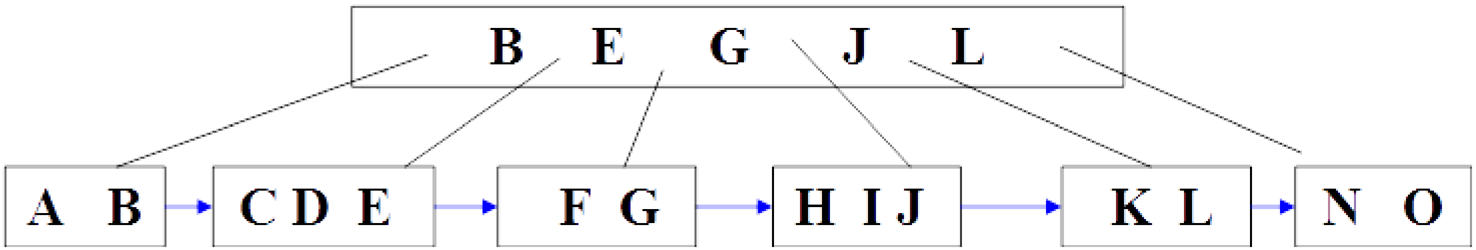


- Insert I
  - HJKL becomes HIJKL. Then split HIJ, KL. J is copied to its parent.

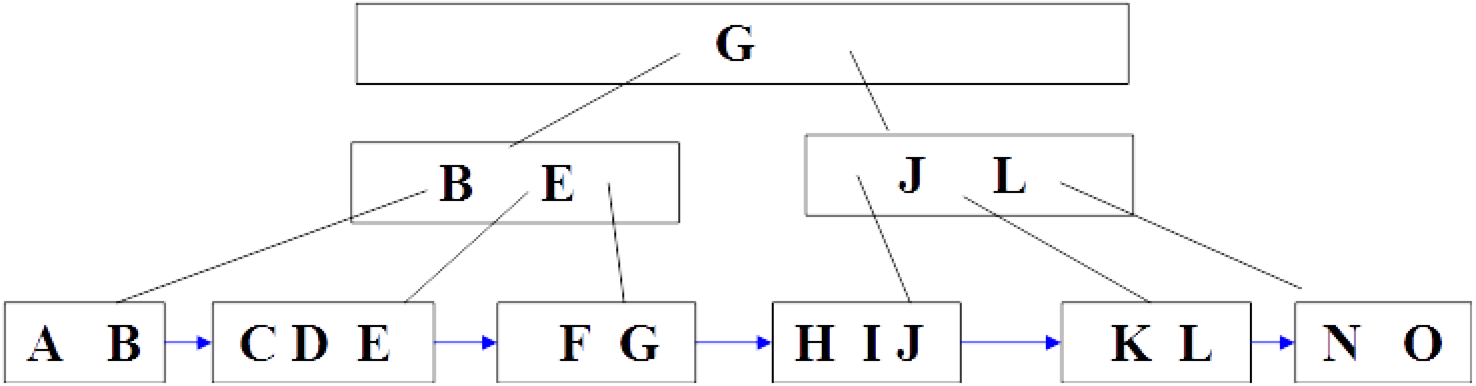


Example

- Insert C
  - DEFG becomes CDEFG. Split CDE, FG. E is copied to its parent.



  - BEGJL splits to BE and JL and a new root with G is created.

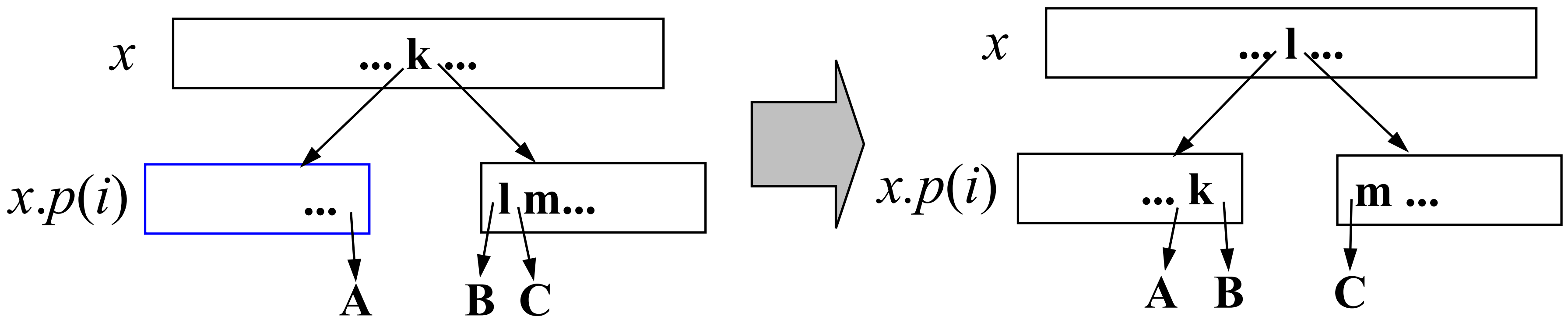


B^+-trees: Deletion

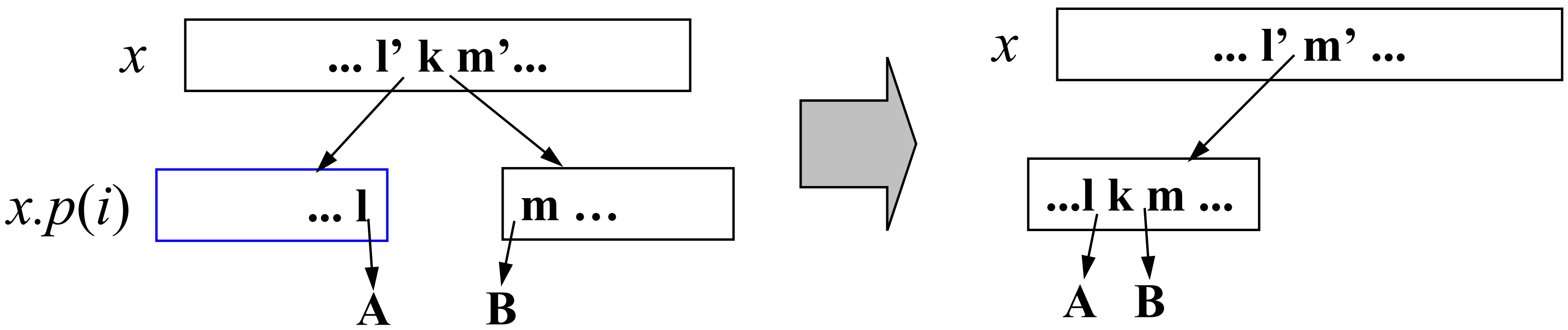
- Opposite of insertion:
  - Phase 1: traverse down to find the key in a leaf
  - Phase 2: remove the key and traverse up handling underfull nodes
- Tree shrinking: if the root has only one child, remove the root.

B⁺-trees: Deletion (internal nodes)

- Underfull handling, case 1: *Distributing*

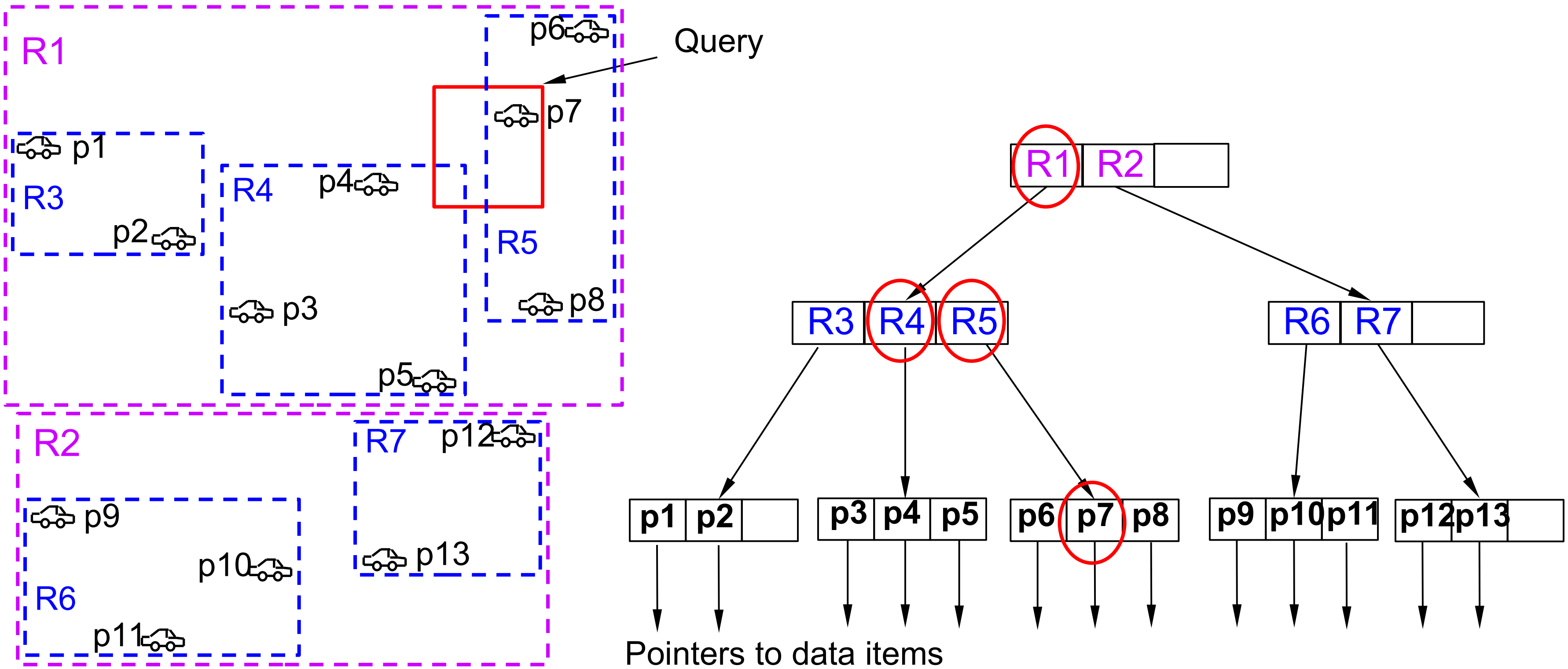


- Underfull handling, case 2: *Merging*



R-trees

- Example



Grow-Post Trees

Grow-Post trees (Generalized Search Trees - GiST)

Bounding predicate (*BP*) = *something that describes entries in a subtree*

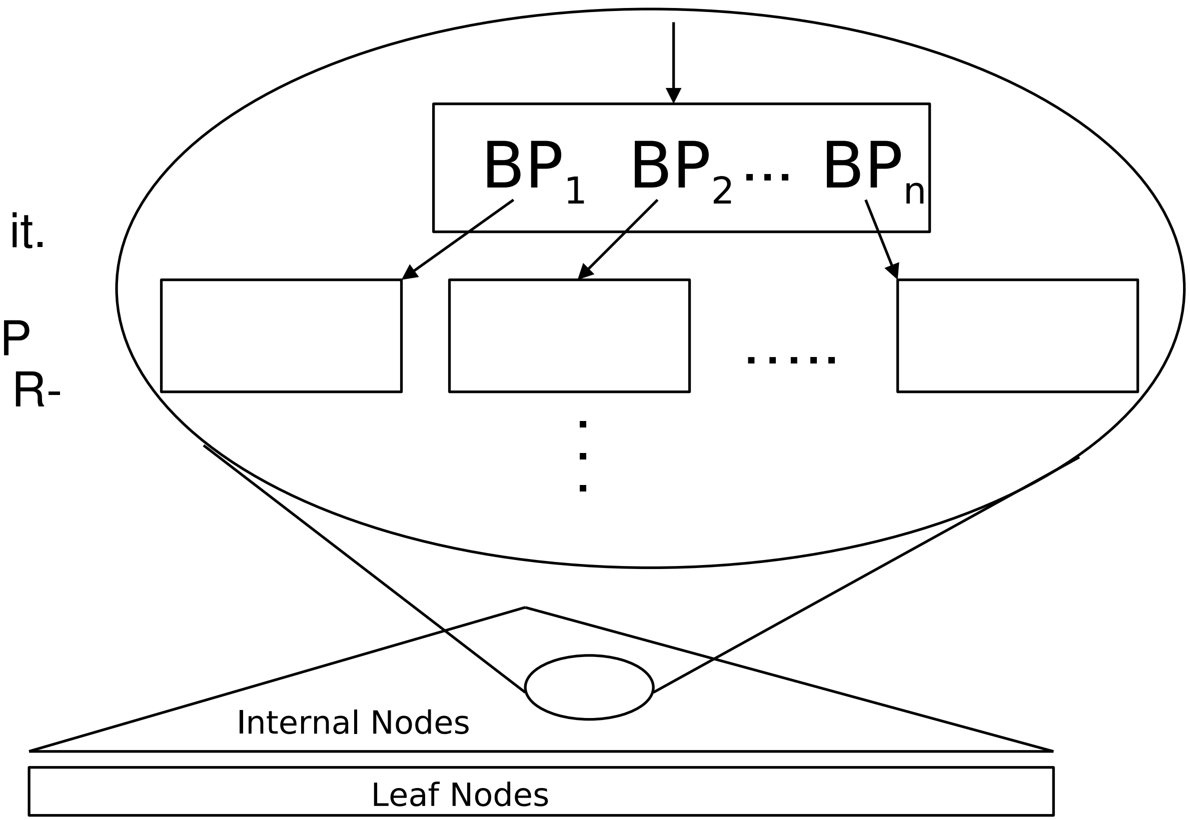
Building blocks of algorithms:

- **Consistent**(*BP*, *Q*) – returns *true* if results of query *Q* can be under *BP* (in the R-tree, MBR intersects *Q*).

- **Penalty**(*BP, E*) – returns an estimate how "worse" *BP* becomes if *E* is inserted under it.

- **Union**(*node*) – computes a BP of a collection of entries (in the R-tree, computes an MBR, minimum bounding rectangle)

- **PickSplit**(*node*) – splits a page of entries into two groups



External DS: Summary

- Two practical data structures (n is the number of pages):
  - **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: $\Theta(log_B n)$
  - **R-trees**: supports multi-dimensional point and range queries, on point and extended objects:
    - Point/range query, deletion: $\Theta(n)$, but usually much better on average
    - Insertion: $\Theta(log_B n)$
  - Both structures have $\Theta(n)$ size.

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)
```

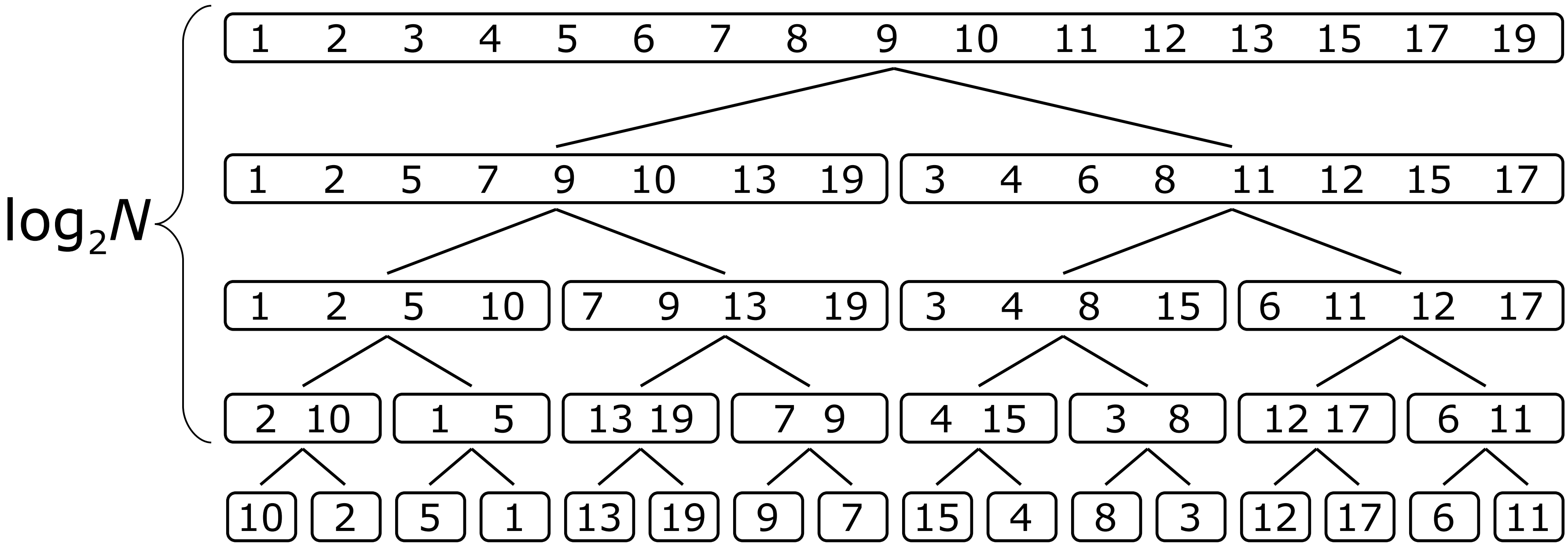
Divide

Conquer

Combine

- Running time?

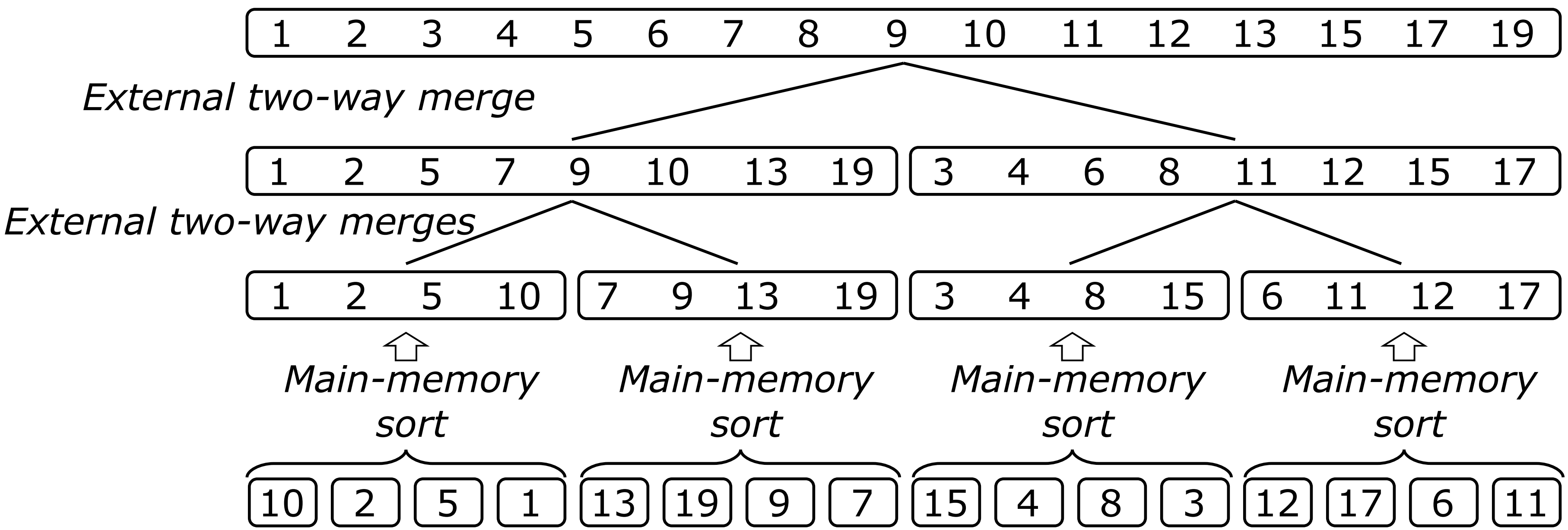
Merge-Sort Recursion Tree



- In each level: merge *runs* (sorted sequences) of size x into runs of size $2x$, 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

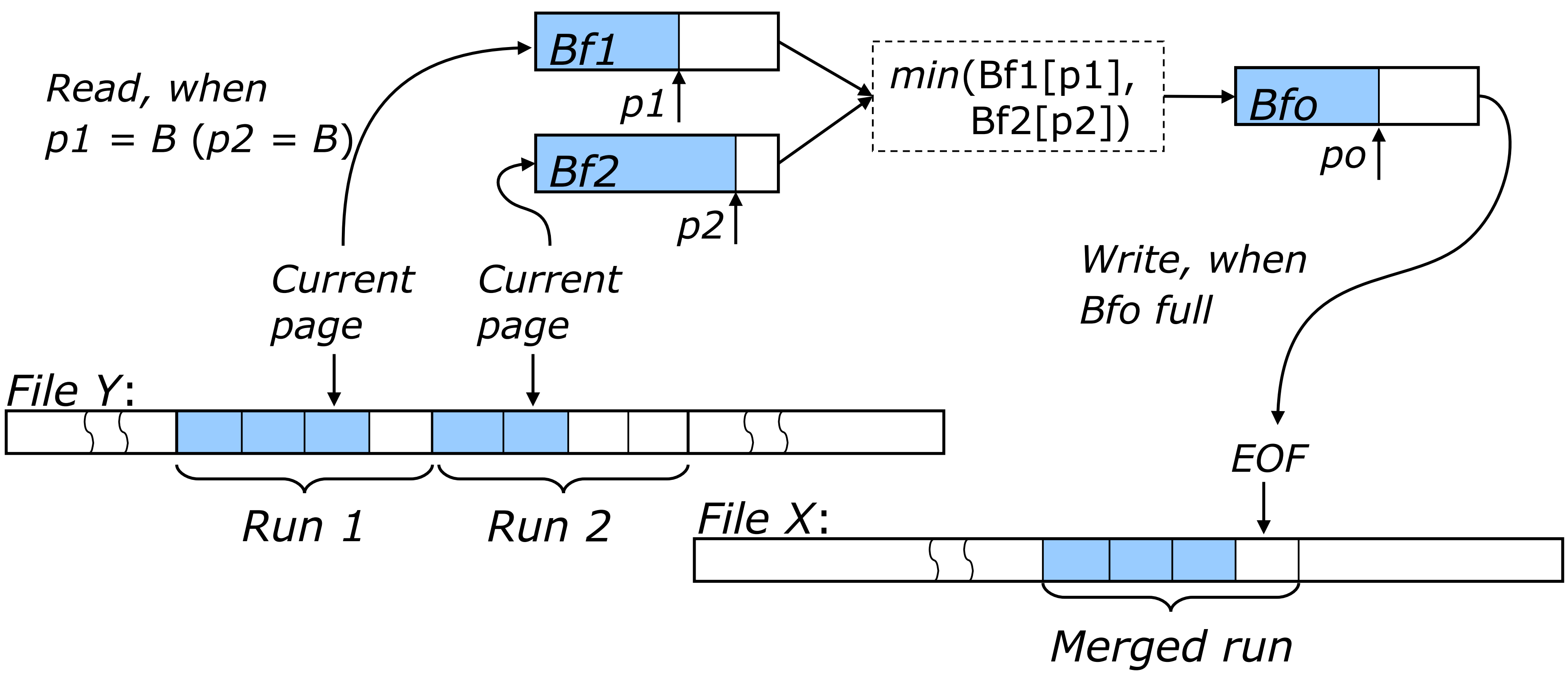
- Input file *X*, empty file Y

- *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

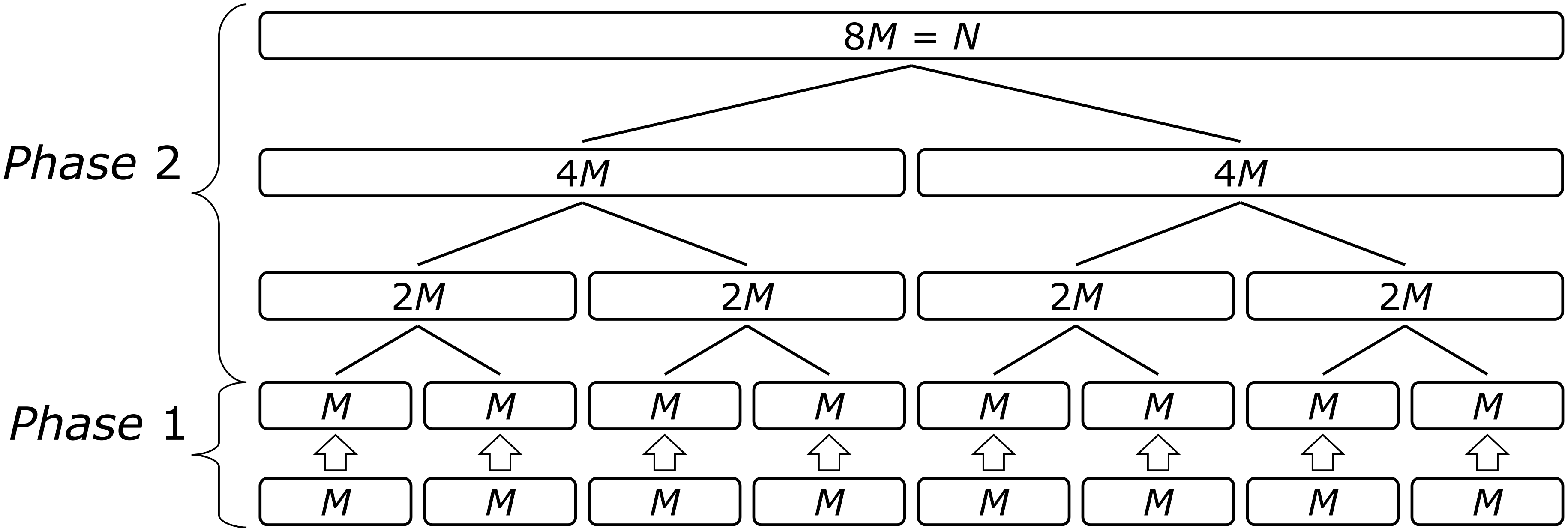
- *MergeAllRuns*(*Y*, *X*): repeat until the end of *Y*:
  - Call *TwowayMerge* to merge the next two runs from *Y* into one run, which is written at the end of *X*
- *TwowayMerge*: uses three main-memory arrays of size *B*



Analysis: Assumptions

- 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

Analysis



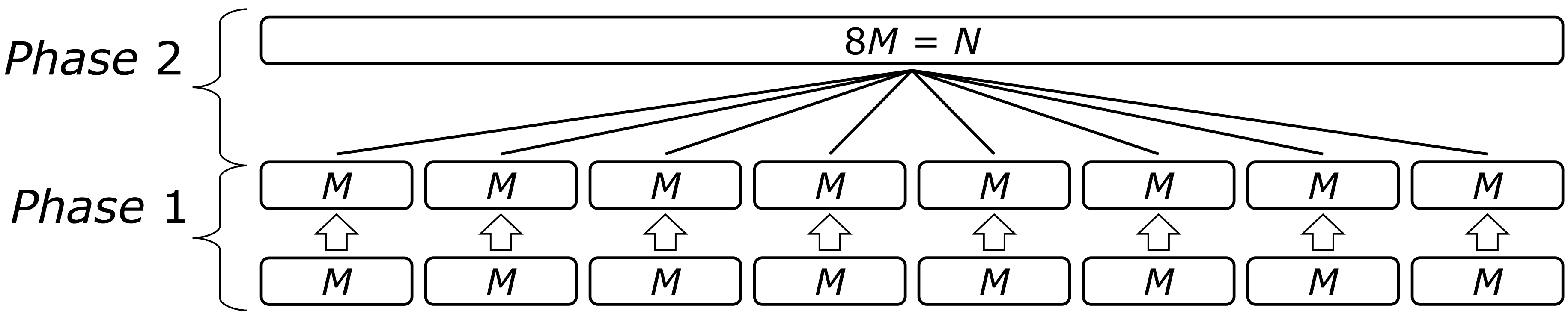
- Phase 1:
  - Read file X, write file Y: $2n = \Theta(n)$ I/Os
- Phase 2:
  - One iteration: Read file Y, write file X: $2n = \Theta(n)$ I/Os
  - Number of iterations: $\log_2 N/M = \log_2 n/m$

Analysis: Conclusions

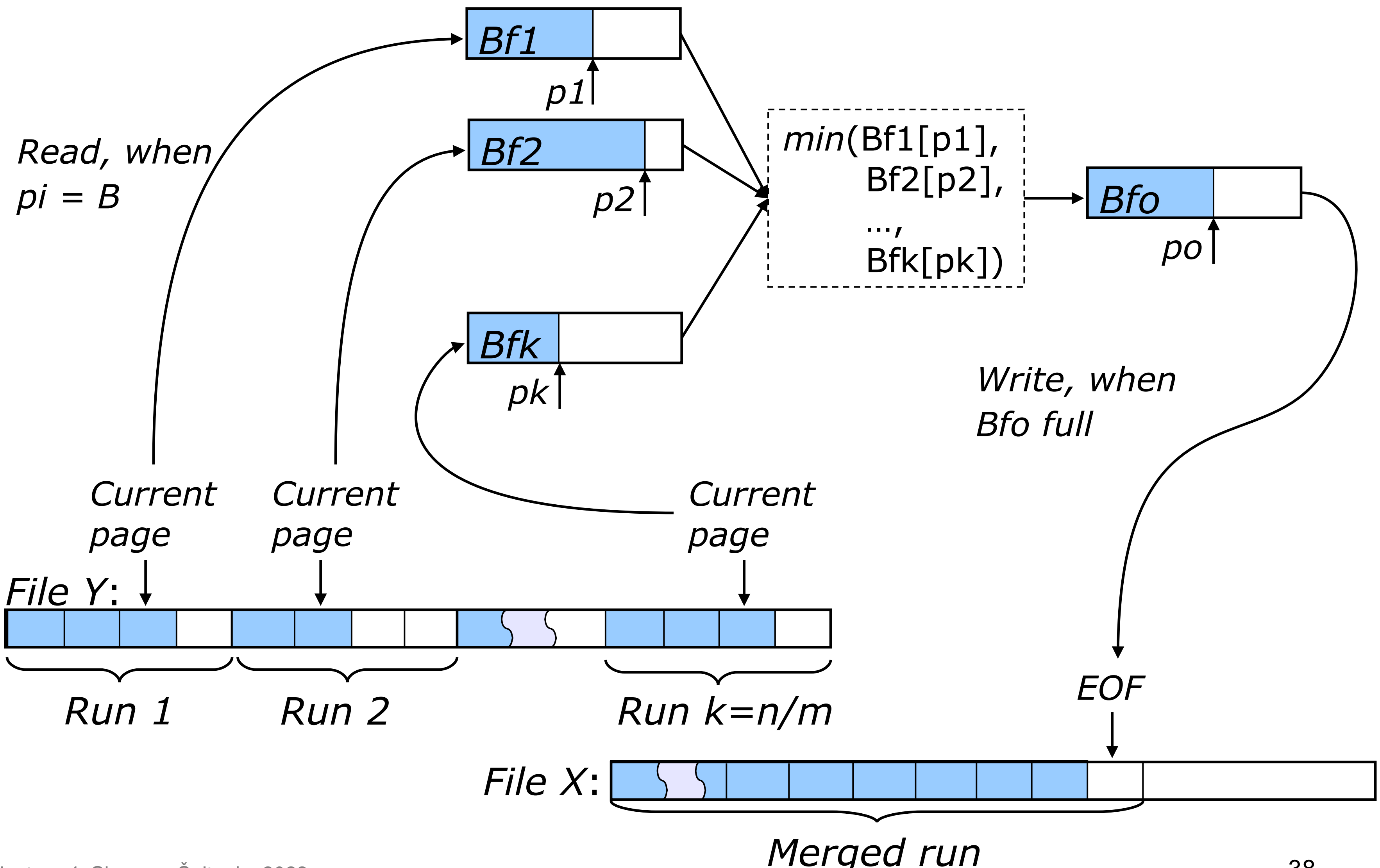
- Total running time of external-memory merge sort: $\Theta(n \log_2 n/m)$
- We can do better!
- Observation:
  - Phase 1 uses all available memory
  - Phase 2 uses just 3 pages out of m available!!!

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



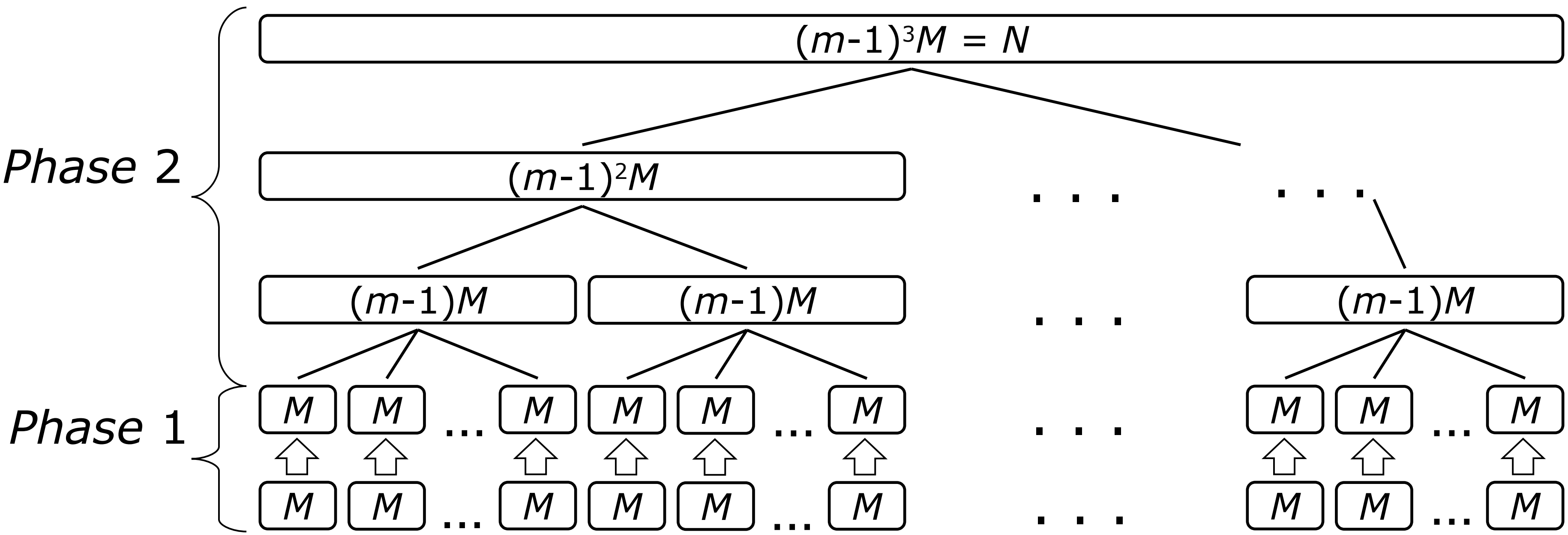
Analysis of TPMMS

- 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

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