AALBORG UNIVERSITY

Advanced Algorithms

Lecture 9
External-Memory Algorithms and Data Structures

Bin Yang
byang@cs.aau.dk

Intended Learning Outcomes (ILO)

- After taking this course, you should acquire the following knowledge
  - Algorithm **design** techniques such as divide-and-conquer, greedy algorithms, dynamic programming, back-tracking, branch-and-bound algorithms, and plane-sweep algorithms;
  - Algorithm **analysis** techniques such as recursion, amortized analysis;
  - A collection of **core** algorithms and data structures to solve a number problems from various computer science areas: *algorithms for external memory*, multiple-threaded algorithms, advanced graph algorithms, heuristic search and geometric calculations;
  - There will also enter into one or more **optional subjects** in advanced algorithms, including, but not limited to: *approximate algorithms*, randomized algorithms, search for text, linear programming and number theoretic algorithms such as cryptosystems.

ILO of Lecture 9

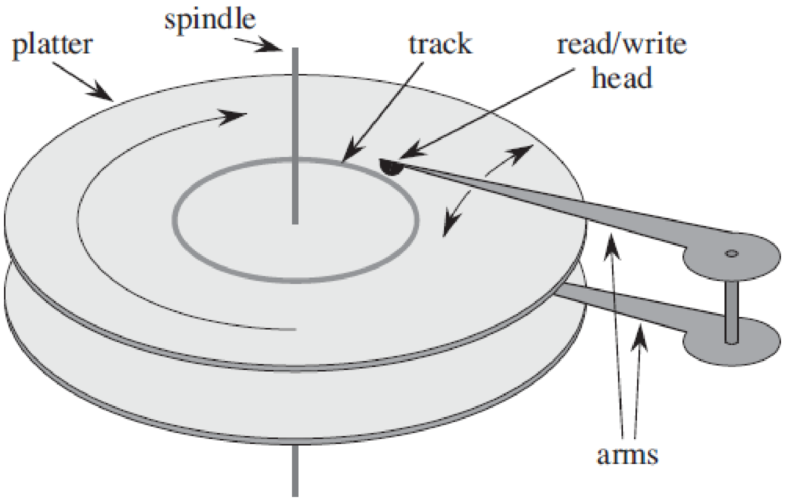
- External memory algorithms and data structures
  - to understand the external memory model and the principles of analysis of algorithms and data structures in this model;
  - to understand the algorithms of B-tree and its variants and to be able to analyze the complexity;
  - to understand the main principles of external tree structures;
  - to understand how the different versions of merge-sort 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.

Agenda

- External memory
- B-trees, B^+-trees, and R-trees
- External memory merge sort

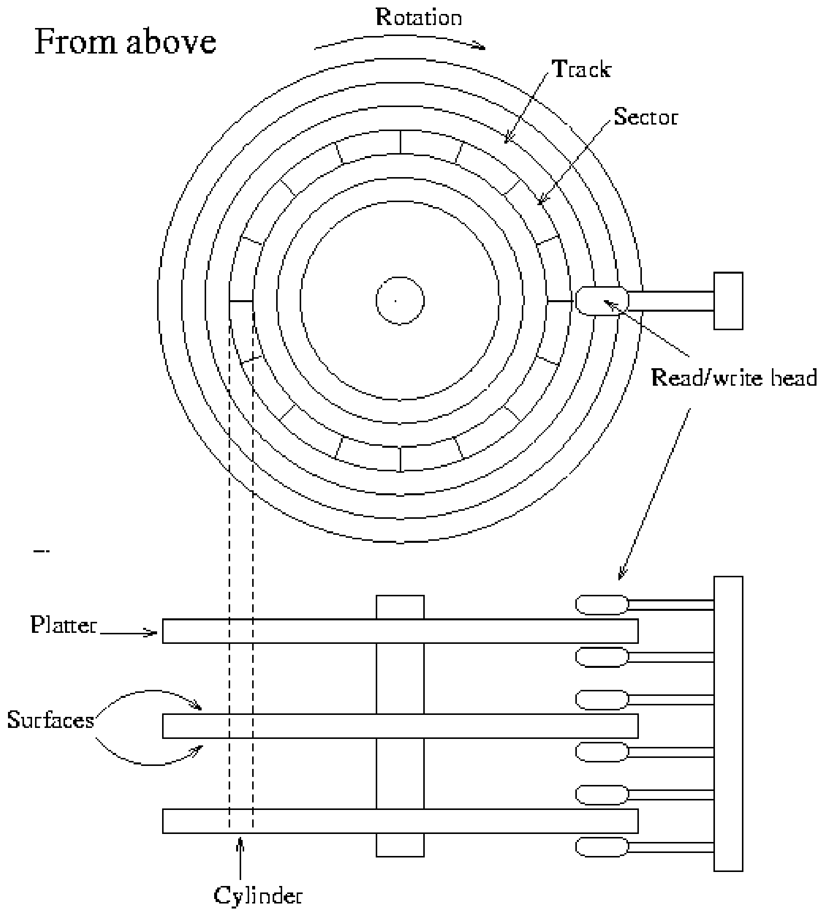
Hard disks, magnetic disks

- In real systems, we need to cope with data that does not fit in main memory.
- How does a hard disk work:
  - It has one or more *platters* that rotate around a *spindle*.
  - Each platter is read/write with a head at the end of an arm.
  - Arms rotate around a common pivot axis.
  - Track is the surface under the head.



Hard disks, magnetic disks

- Reading data from the hard disk:
  - *Seek* with the head.
  - *Wait* while the necessary sector rotates under the head
  - *Transfer* the data.



Some numbers

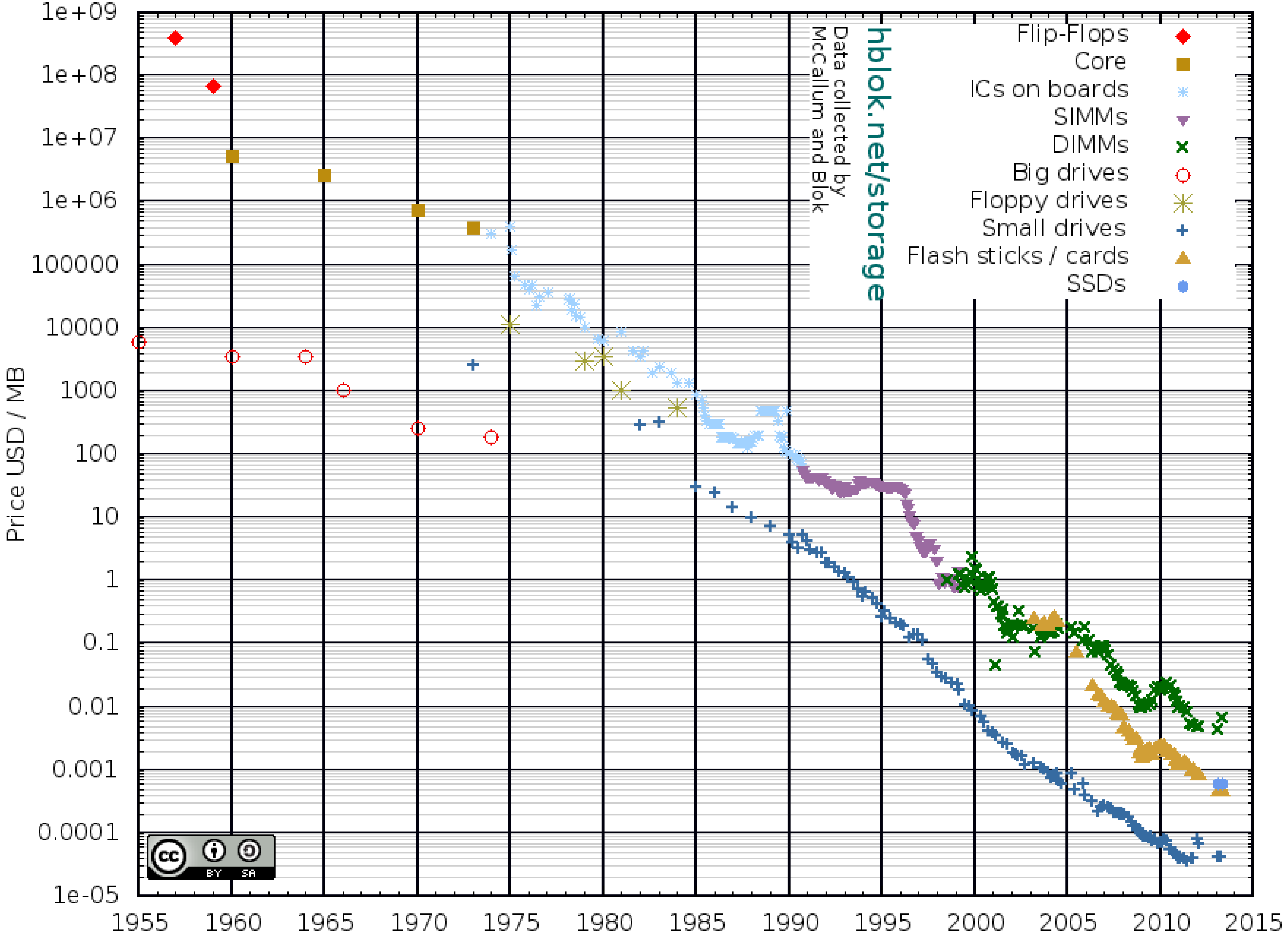
- Rotation speed: 5,400 to 15,000 RPM.
  - Laptop hard disk, normally 5,400 or 7,200.
  - My laptop's hard disk: WD5000LPVX: 5,400
- Let's consider a hard disk with 7,200 RPM
  - One rotation takes 60/7200≈8.3 milliseconds (10^{-3})
  - Plus some time for moving the arms.
  - Main memory: 50 nanoseconds (10^{-9})
  - The hard disk takes 5 orders of magnitude longer access times.

- Conclusions for using hard disks
  - Disk access is much slower than main-memory access
  - Thus, it makes sense to read and write in large blocks – *disk pages* (4 – 32Kb)
  - *Sequential* access is much faster than *random* access

SSDs

- The same, although to less extent is true for *flash*-based *solid state drives* (SSDs):
  - It is more 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) – x2000 → SSD(~0.1ms) – x50 → HDD(~5ms)

Historical Cost of Computer Memory and Storage



Picture from hblok.net/blog/storage

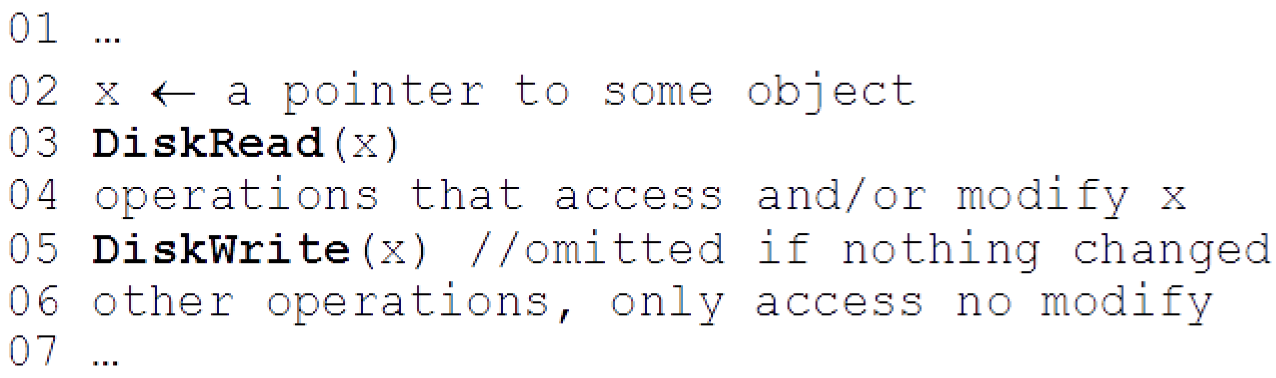
External memory model

- Two principal components of the running time analysis:
  - Not only, the CPU time, i.e., the computing time.
  - But also, and more importantly, the number of disk page accesses, i.e., the number of I/O.

- *B* – page size is an important parameter:
  - *Example*: n = 256MB, *B* = 4KB, 0.1 ms disk access
    - *n* disk accesses: $256*10^6*0.1*10^{-3} = 25\,600s = 7.1$ hours
    - *n/B* disk accesses: $= 64*10^3*0.1*10^{-3} = 6.4s$
    - Read/Write in blocks but not individual data instances
  - Not "just" a constant:
    - $\Theta\ (log_2 n\ ) \neq \Theta\ (log_B n\ )$
    - $\Theta(n) \neq \Theta(n/B\ )$

External memory 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
…
```

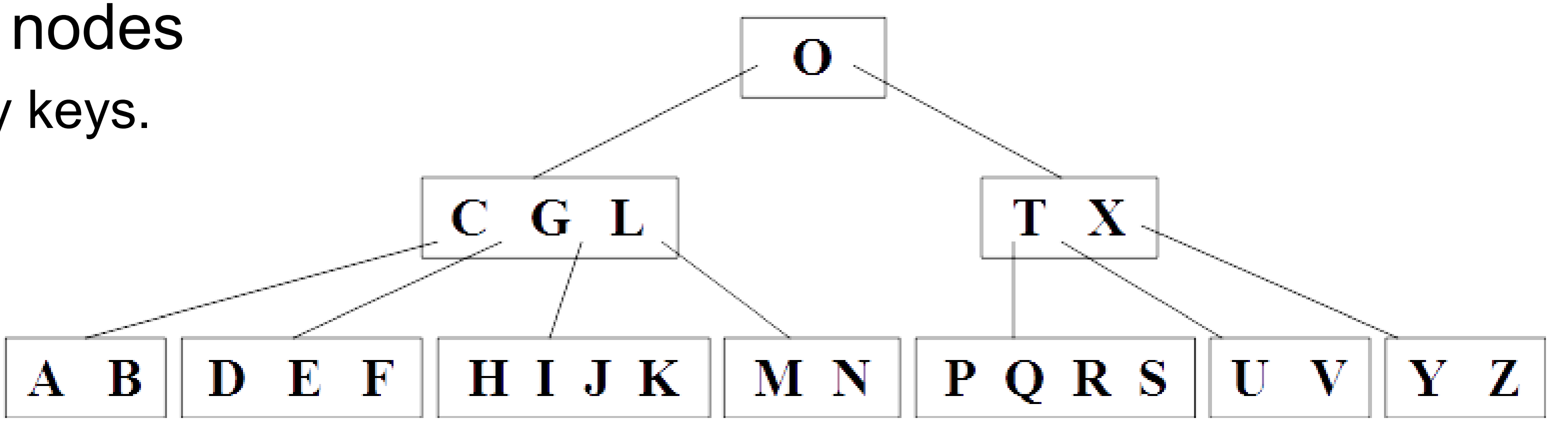
- If the object referred to by x resides on disk, **DiskRead**(x) needs to read object x into main memory before we can access or modify x.
- If the object referred to by x is already in main memory, **DiskRead**(x) does nothing and does not incur another time disk access.

Agenda

- External memory
- B-trees, B^+-trees, and R-trees
- External memory merge sort

B-trees

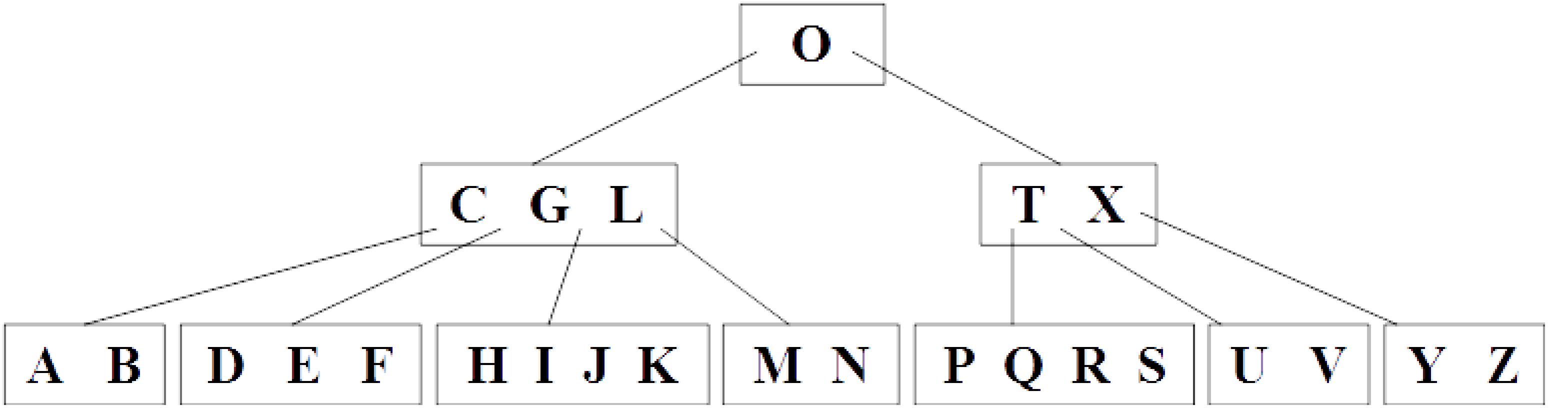
- B-tree is a balanced tree.
  - All leave nodes have the same depth.
- Internal nodes
  - *x* multiple keys and *x+1* pointers to child nodes.
  - pointer_1 key_1 pointer_2 key_2 pointer_3 key_3 … pointer_x key_x pointer_{x+1}
  - $\text{key}_1 \leq \text{key}_2 \leq \text{key}_3 \leq \ldots \leq \text{key}_x$
  - 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.



B-trees

- Each node resides on a disk page.
- Given a page size ***B***, how many keys can a page hold at most?
  - Assume that a key is a 4 bytes integer and let a pointer be 8 bytes.
  - For a B=4k page setup, we have $4m+8(m+1)\leq 4096$. Then, $m=340$.
- Nodes have lower and upper bounds on the number of keys they can contain
  - Upper bound: each node has at most m keys and m+1 children (max_fan-out)
  - Lower bound: each node has at least ***t***= $\lfloor m/2 \rfloor$ keys and t+1 children (min_fan-out)
  - Example:
    - m=5: at most 5 keys 6 children, and at least t=2 keys 3 children.
    - m=6: at most 6 keys 7 children, and at least t=3 keys 4 children.
- Root is an exception: root node can have as little as only one key and two children.

B-trees



- Each node resides on a page.
- m=4, $t = \lfloor m/2 \rfloor = 2$
  - Each node has at least 2 keys and at most 4 keys.
  - Each node has at least 3 children and at most 5 children.
  - Root is an exception, only has 1 key and 2 children.

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.

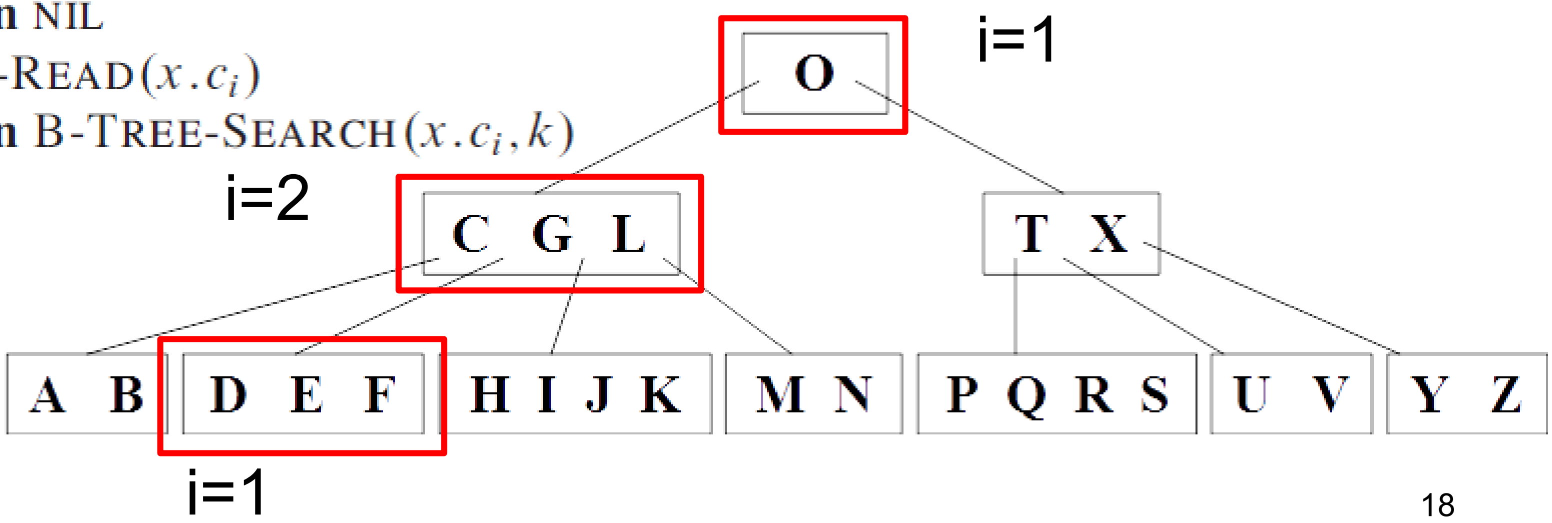
Pseudo code

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

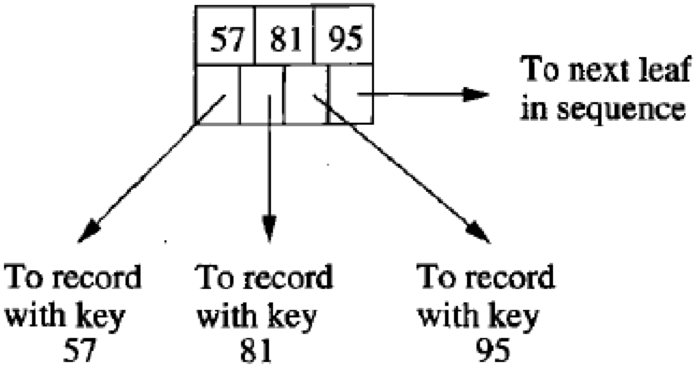
Searching for “D”, i.e., k = D
B-Tree-Search(root, D)

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



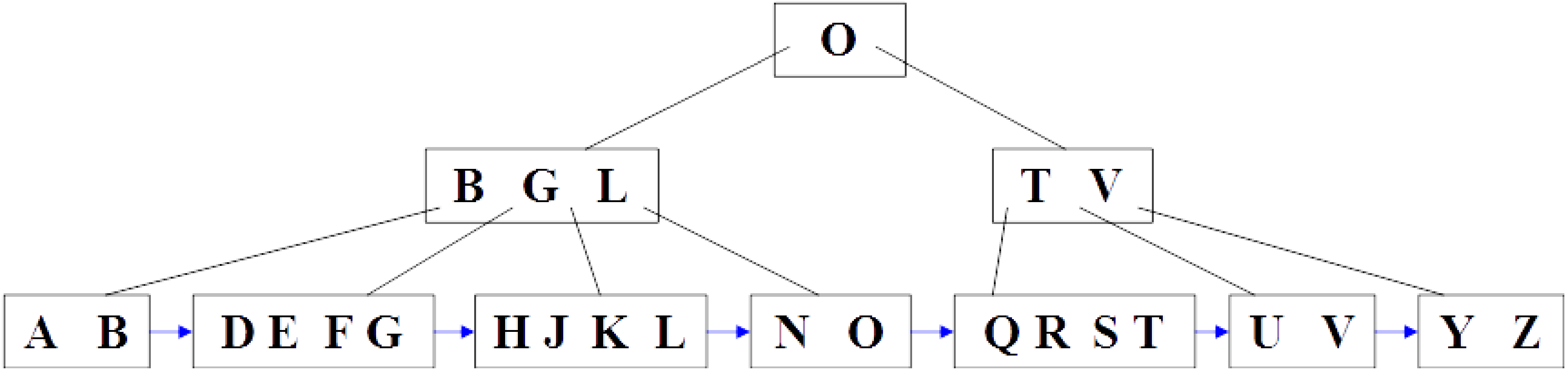
B$^+$-trees

- B$^+$-trees is a variant of B-trees:
  - All data keys are in leaf nodes.
  - Leaf nodes are connected into a linked list.
  - Extensively used in database systems for indexing data records that are stored in a database.



Searching on B$^+$-trees

- Searching
  - Always goes to a leaf
    - O in B-trees vs. Θ in B$^+$-trees
  - Point search: $\Theta(\log_t n)$
  - Range search: $\Theta(\log_t n + k/t)$

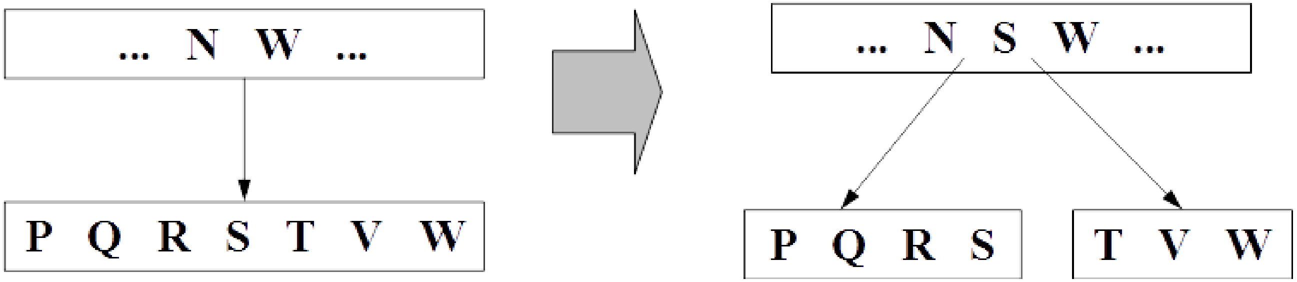


B$^+$-trees: Insertion

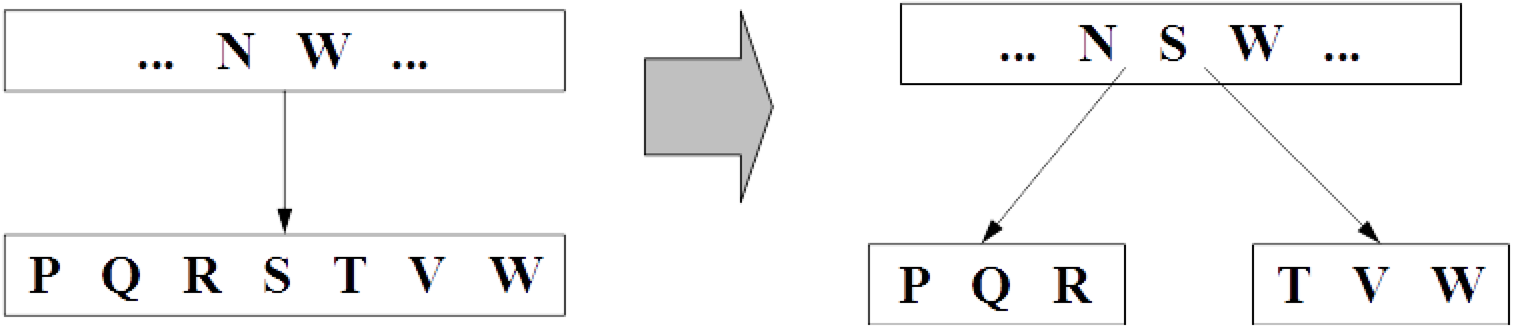
- Skeleton of the algorithm:
  - *Down-phase:* recursively traverse down and find the leaf (as in search)
    - If the leaf has room, just insert it in the leaf. Otherwise:
  - *Up-phase: split* nodes and propagate the splits up the tree
    - Split the leaf into two leaves and divide the keys between the two new nodes. Copy the middle key to its parent.
    - Apply the same strategy to insert the middle key to its parent.
      - If there is room, insert directly. Otherwise, split and move the middle key to its parent.
    - When the root is full, then we create a new root with a single key.

B^{+}-trees: node splitting

- Assume that at most 6 keys in a node.
- Leaf node (**copy** the middle key to the parent)



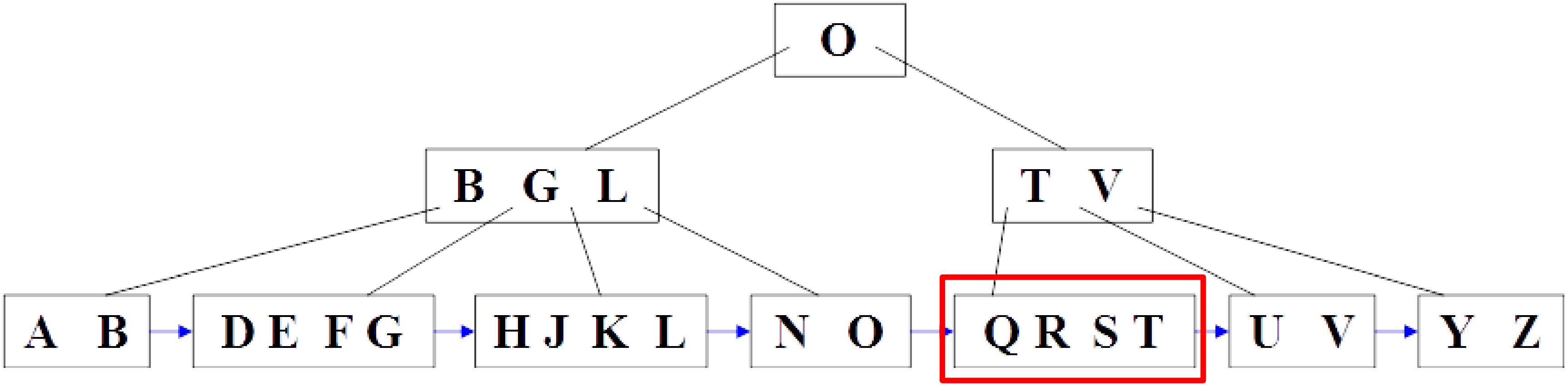
- Internal node (**move** the middle key to the parent)



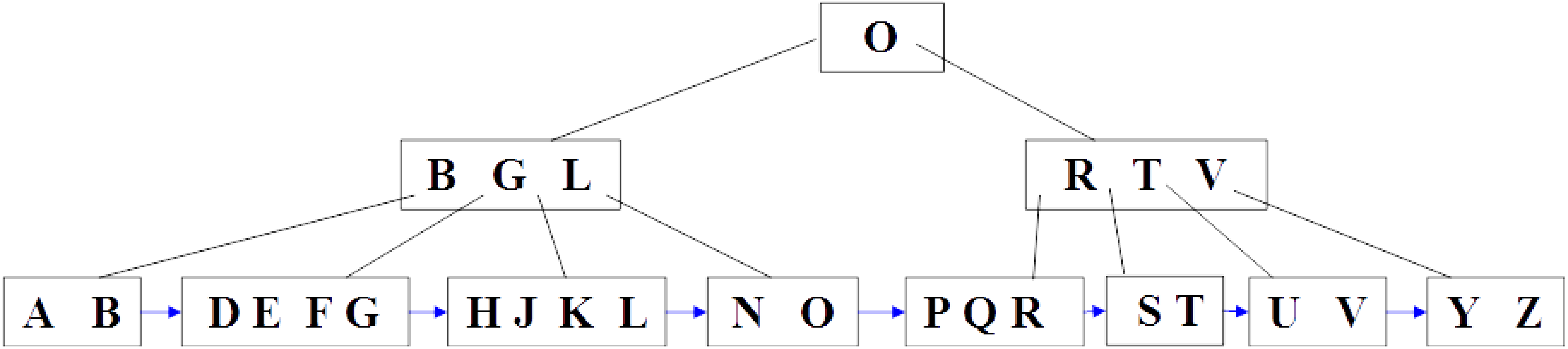
- The tree grows when the root is split into two nodes and their parent becomes the new root.

Example

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

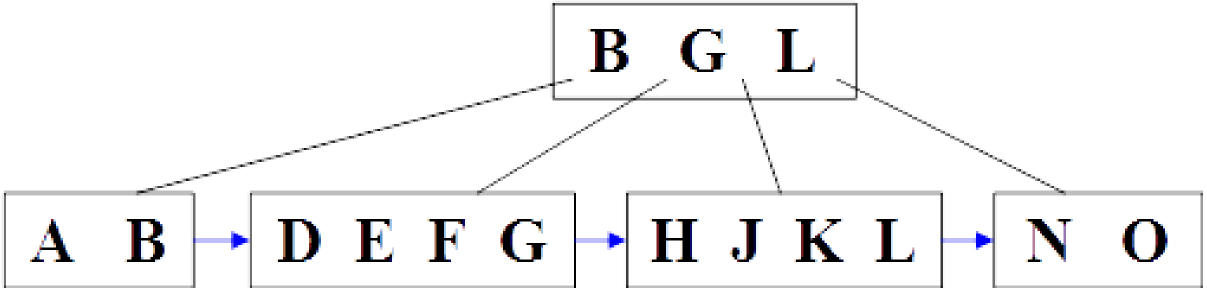


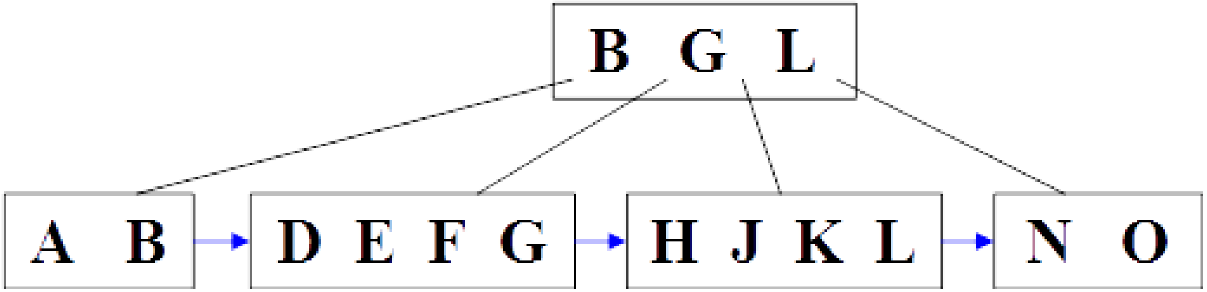
Down-phase: Leaf node
(**copy** the middle key to the parent)



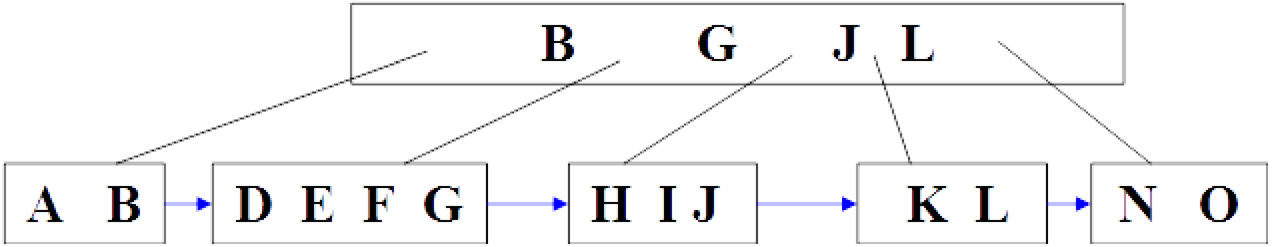
Mini-quiz (also on Moodle)

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

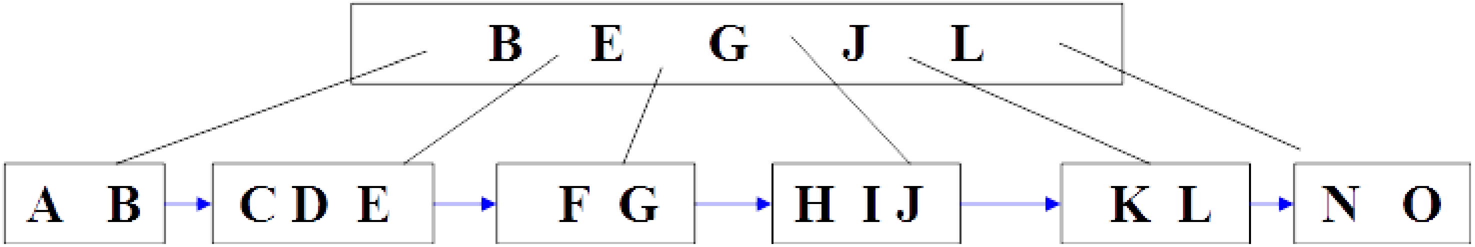




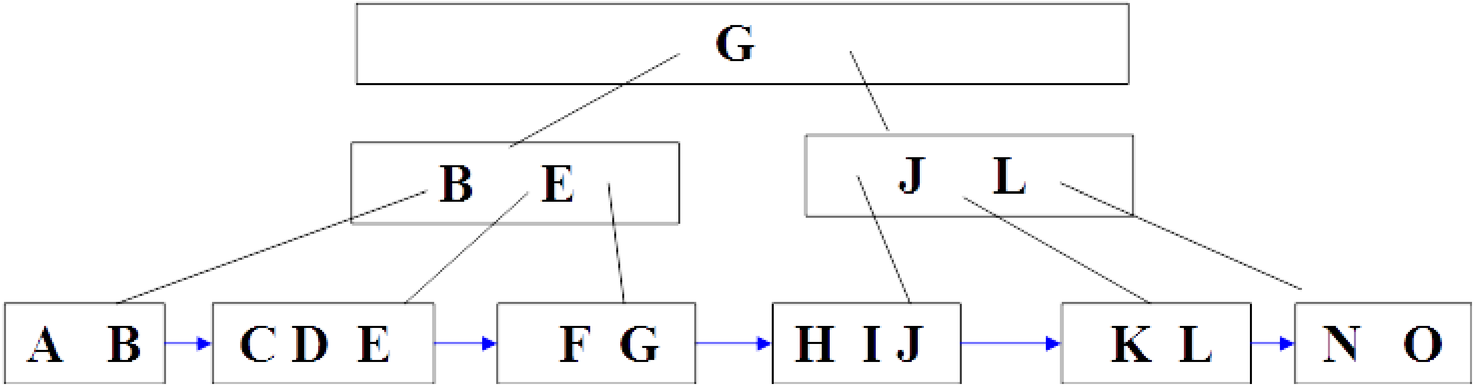
- Insert I
  - HJKL becomes HIJKL. Then split HIJ, KL. J is copied to its parent.



- 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

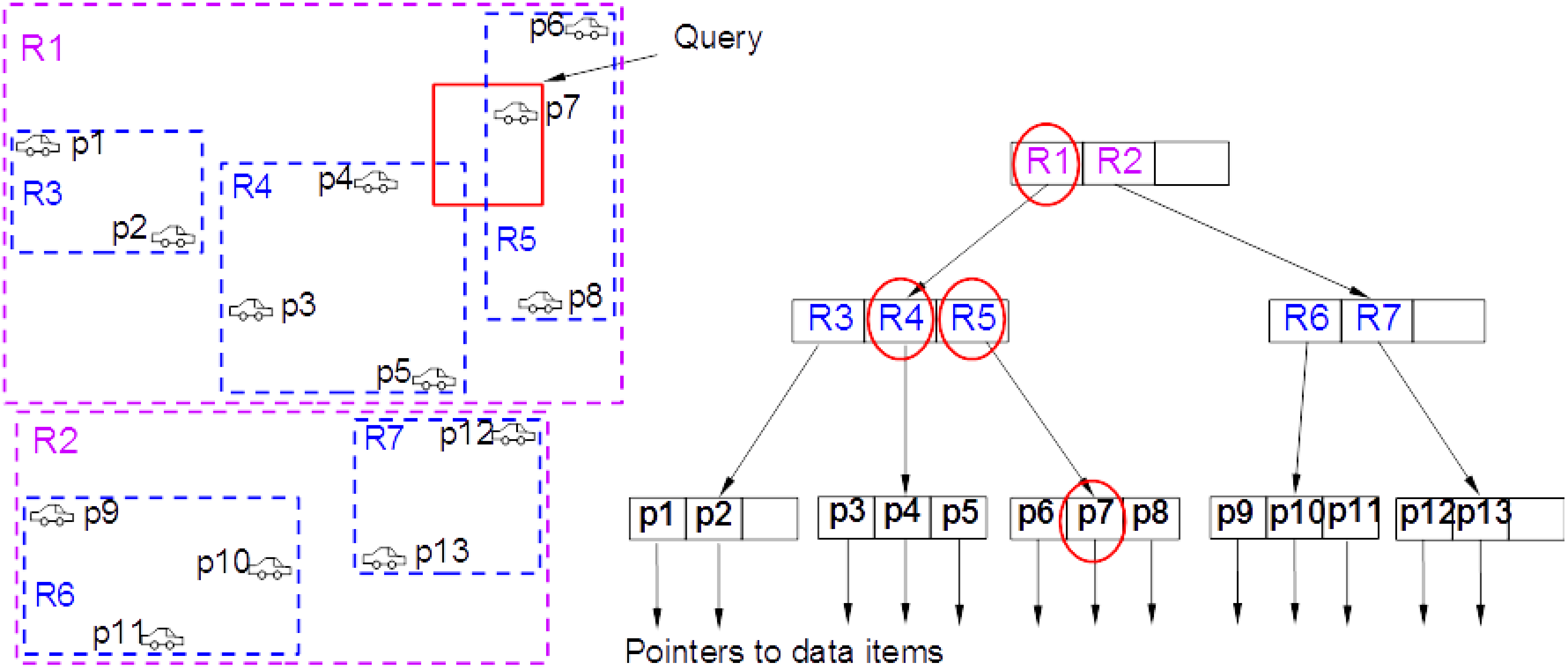
- Opposite of insertion:
  - Down-Phase: traverse down to find the key in a leaf
  - Up-Phase: remove the key and traverse up handling underfull nodes
- Tree shrinking: if the root has only one child, remove the root.

B^+-trees: Deletion (under-full)

- After deleting a key from node N, N becomes underful, i.e., less than the minimum number of keys/children.
- Case 1: If one of the adjacent sibling nodes of N has more than the minimum number of keys, move one from its sibling node to N.
- Case 2: If neither adjacent sibling node can provide the extra key. Merge N and one of its adjacent sibling into one node.
  - One node has less the minimum number of keys, and one has exactly the minimum number of keys. Combining them into one node won’t exceed the maximum number of keys.
  - This is why we set the minimum number of keys to be half of the maximum number of keys.

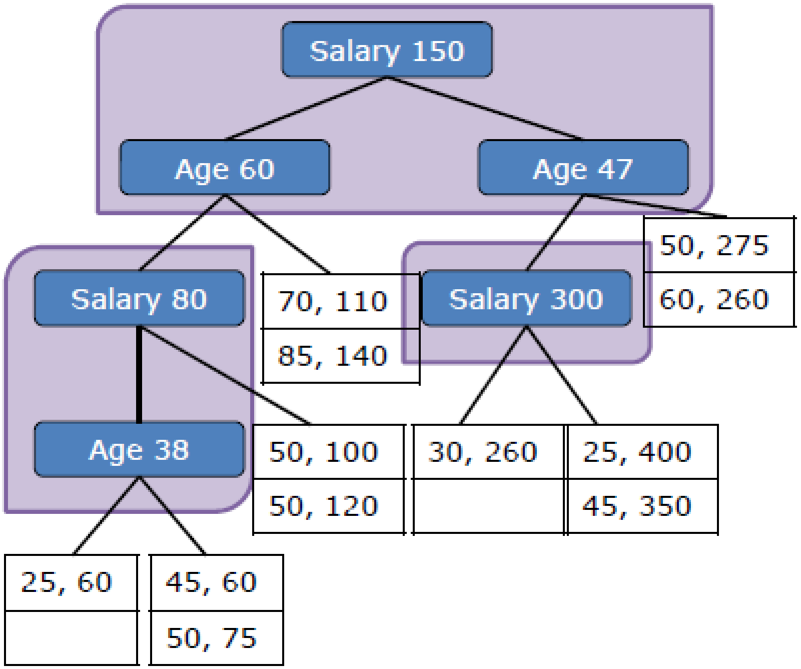
R-trees

- A multi-dimensional extension of B-trees.
- Key is not a value, but a multi-dimensional range.
  - 2D: A rectangle
- Mini-quiz: how many rectangles can a node hold?



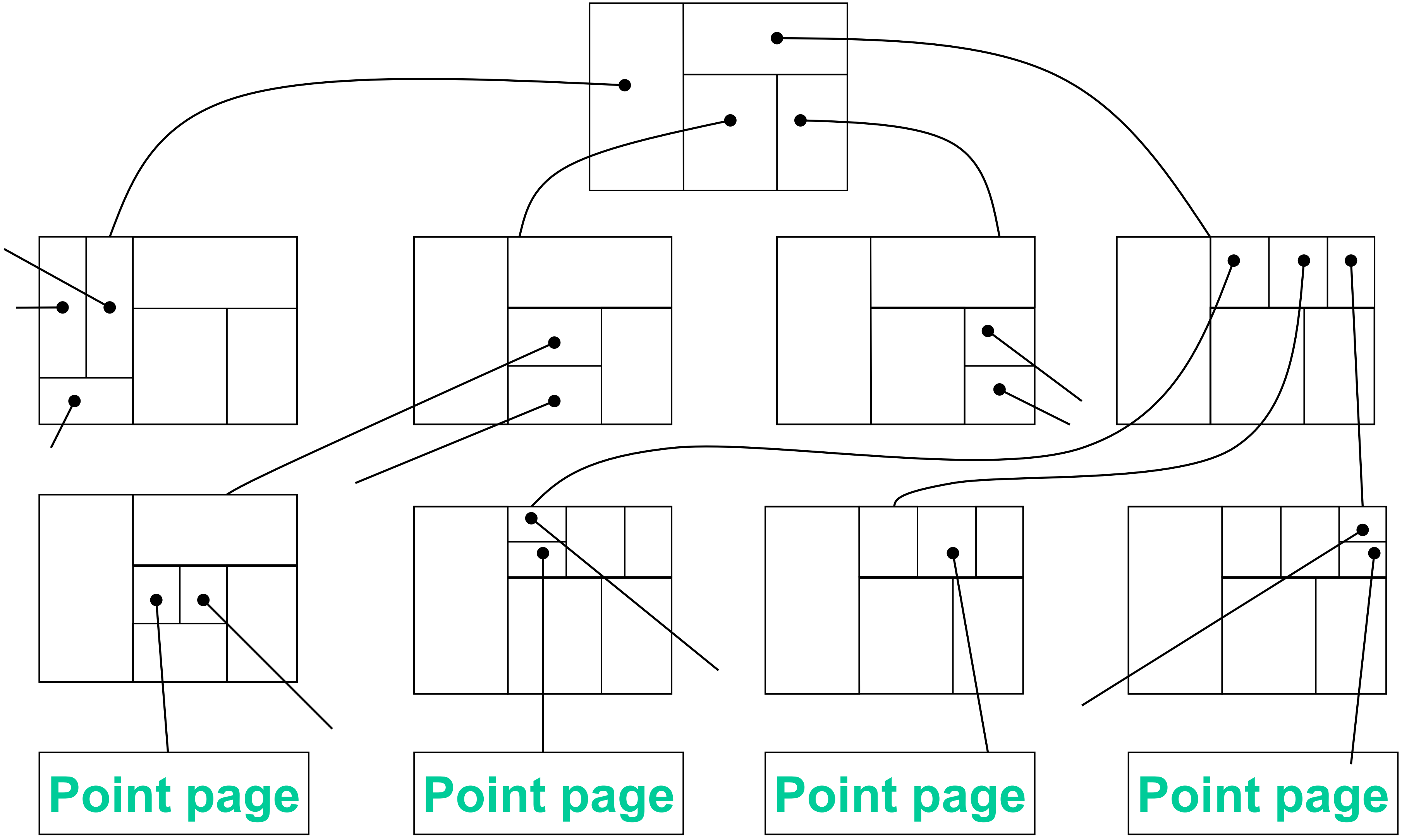
What about kd-trees?

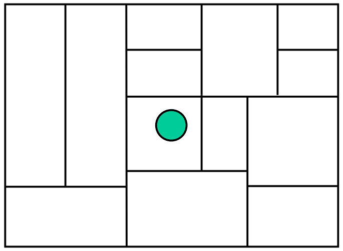
- Since inner node of a kd-tree is tiny, we may group several inner nodes and store them in one disk page.
  - To minimize the number of disk pages that we must read from disk while traveling down one path, it is good to group a node with its children inner nodes for some number of levels.



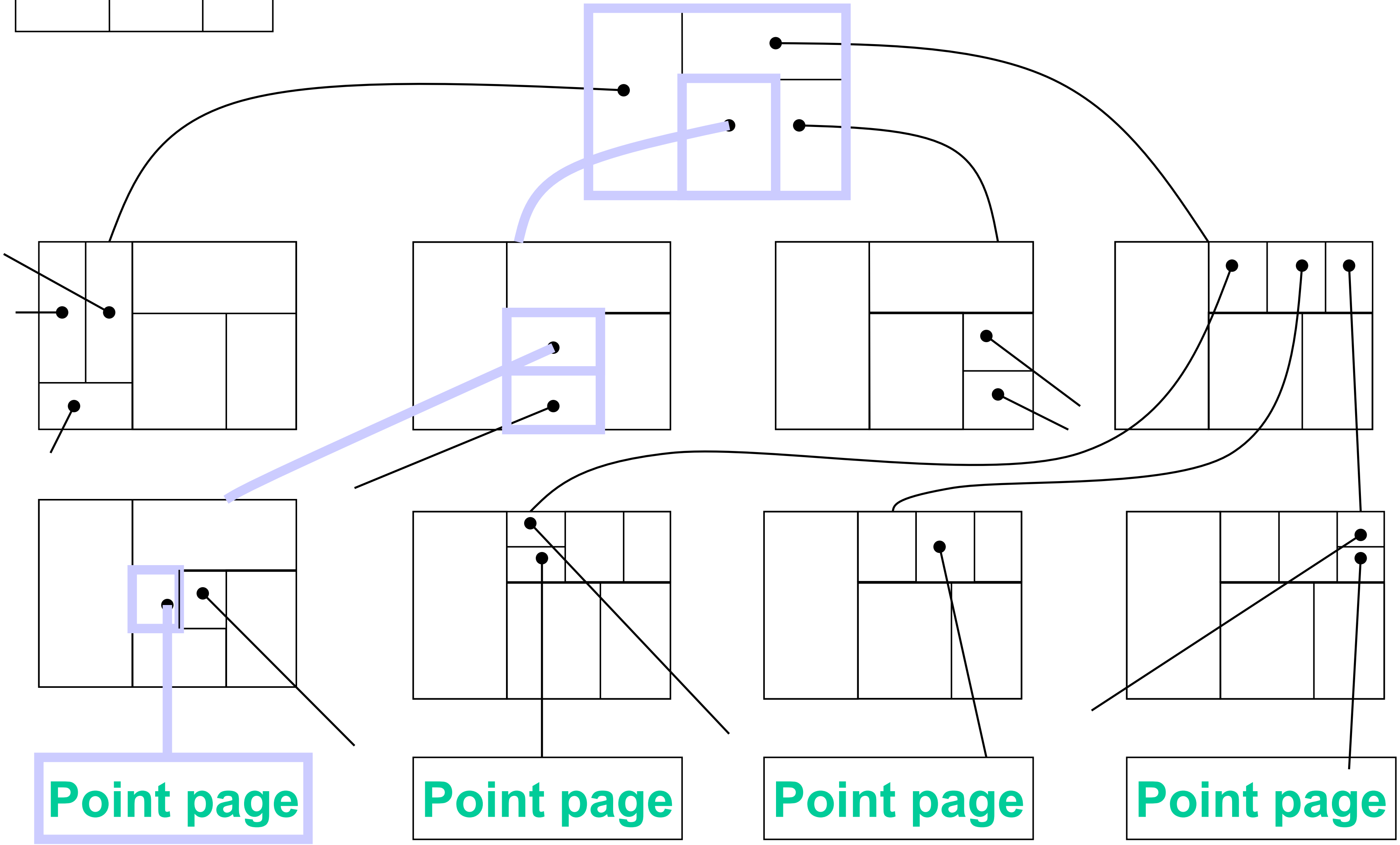
kdb-tree

- Combining the characters of kd-trees and B-trees
  - Multidimensional search efficiency of kd trees.
  - I/O efficiency of B-trees, multi-way branch, balanced tree.
    - Fan out determined by page size and size of entries.
- Region nodes (internal nodes)
  - (region, pointer-to-a-child-node) pairs.
  - Region is defined as min/max per dimension.
  - Regions are disjoint, and union of all regions in a node is a region.
  - Region of the root is the whole space.
- Point nodes (leaf nodes)
  - (point, pointer-to-data-record) pairs.
- Nodes=pages, each node is stored on a disk page.


Point page
Point page
Point page
Point page



Point query



Design principles of external memory trees

- What do we want?
  - A short tree, because it decides how many pages will be loaded, i.e., the number of I/Os.
- How to achieve this?
  - Fully utilize a page to have many entries in a node.
- Short, but wide, trees.

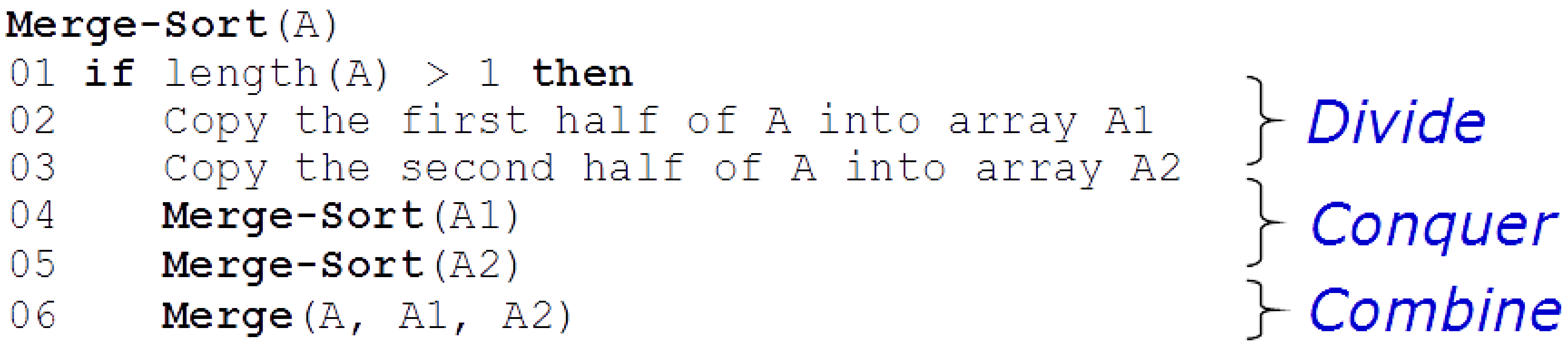
Agenda

- External memory
- B-trees, B^+-trees, and R-trees
- External memory merge sort

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 sorting



```
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?

Recurrence: $T(n)=2T(n/2)+\theta(n)$

$\theta(n\lg n)$

External-memory merge sort

- Settings
  - The total number of input elements that we want to sort is N.
  - Available memory can hold M elements for in-memory merge sort.
  - A disk page can hold B elements.
  - N>M>B.
  - n=N/B: the total number of disk pages of the input.
  - m=M/B: the total number of disk pages that fit in the available memory.
- Example: $N=10^6 > M=10^3 > B=10^2$
  - $n = 10^4$ disk pages of elements in total to be sorted.
  - m = 10 disk pages can fit in the available memory.

External-memory merge sort

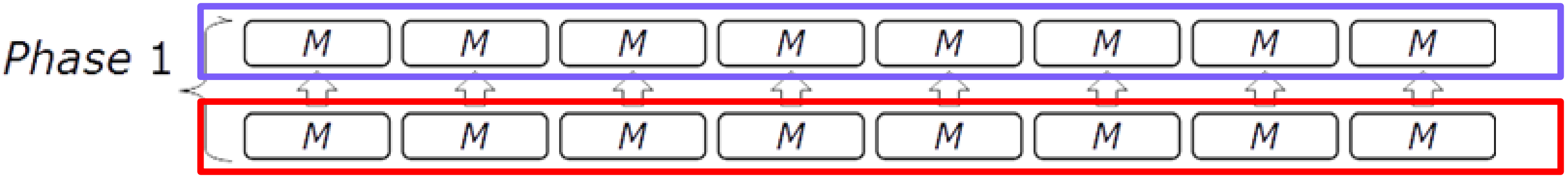
- Input file *X*, empty file Y with the same size of X.
- *Phase* 1: Repeat until the end of file *X* :
  - Read the next *M* elements from *X.*
  - Sort them in main-memory.
    - We call them a run, i.e., a sorted sub-array.
  - Write them at the end of file *Y.*
  - *In total, N/M runs, each with M sorted elements.*

What is the complexity w.r.t. disk page visits (i.e., # of I/O) in the first phase?

2n=2*320

- Assume that: N=8000, M=1000, B=25
  - n=320 pages, m=40 pages
- After phase 1, we have 8 runs in file Y.
  - For each run, we have 1000 elements which are sorted.

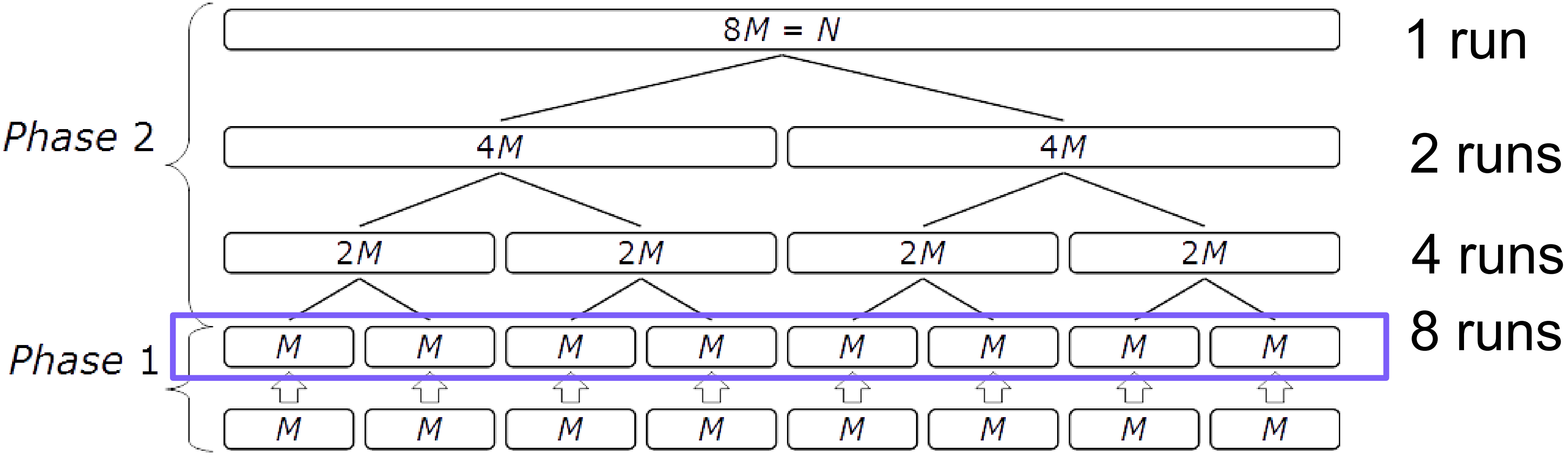
Y: 8 runs. Each run is sorted already.



Input file X. Unsorted.

External-memory merge sort

- *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*

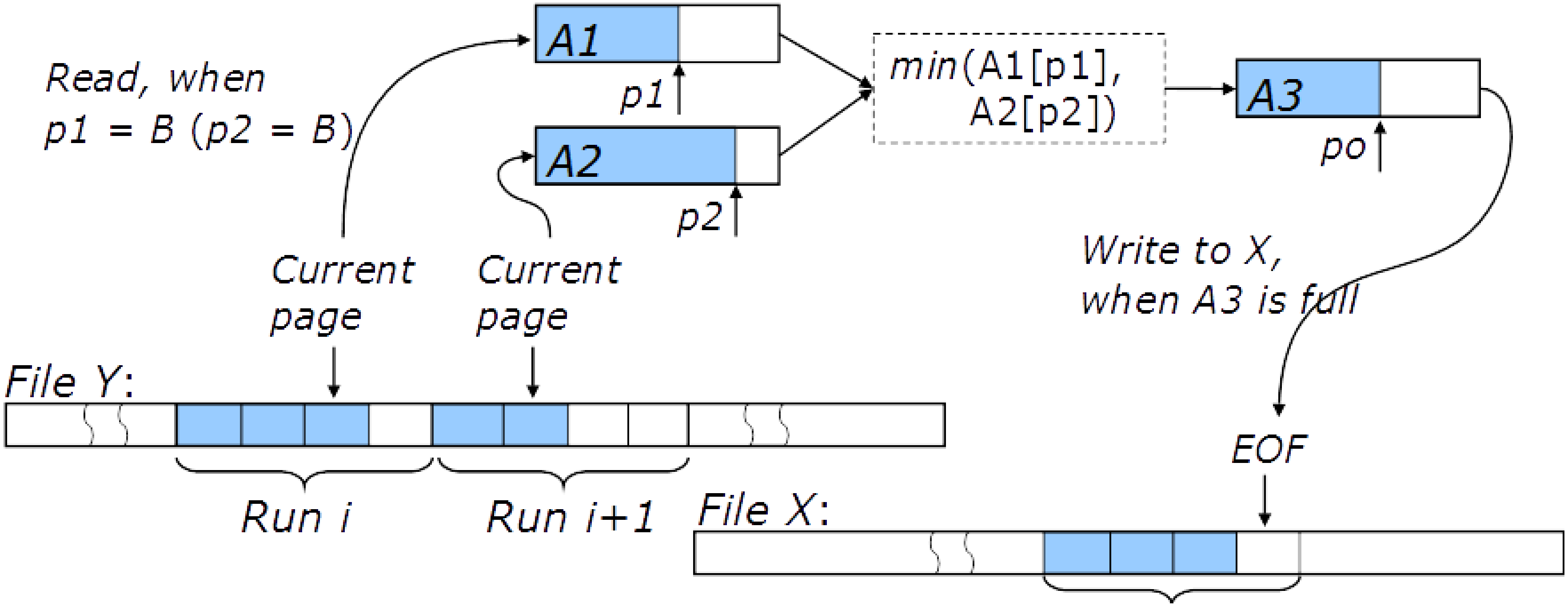


External two-way merge

- In Phase 2, we cannot use Merge() function from the main-memory merge sorting
    - Because each run has M elements already, and we cannot load both runs into the main memory to perform the Merge() function.
- Instead, we use ***two-way*** merge to merge ***two*** runs into ***one longer*** run.
    - We use three main-memory arrays of size *B, i.e., 3 main-memory pages*
        - A1, A2, A3.
    - Copy one page from the first run to A1
    - Copy one page from the second run to A2.
    - Merge A1 and A2 to A3 using main-memory Merge.
    - Whenever A3 is full, store A3 to file X.

External two-way merge

- Every time, merge two runs into one longer run.
  - Copy one page from the first run to A1
  - Copy one page from the second run to A2.
  - Merge A1 and A2 to A3 using main-memory merge.
  - Store A3 to file X when A3 is full.

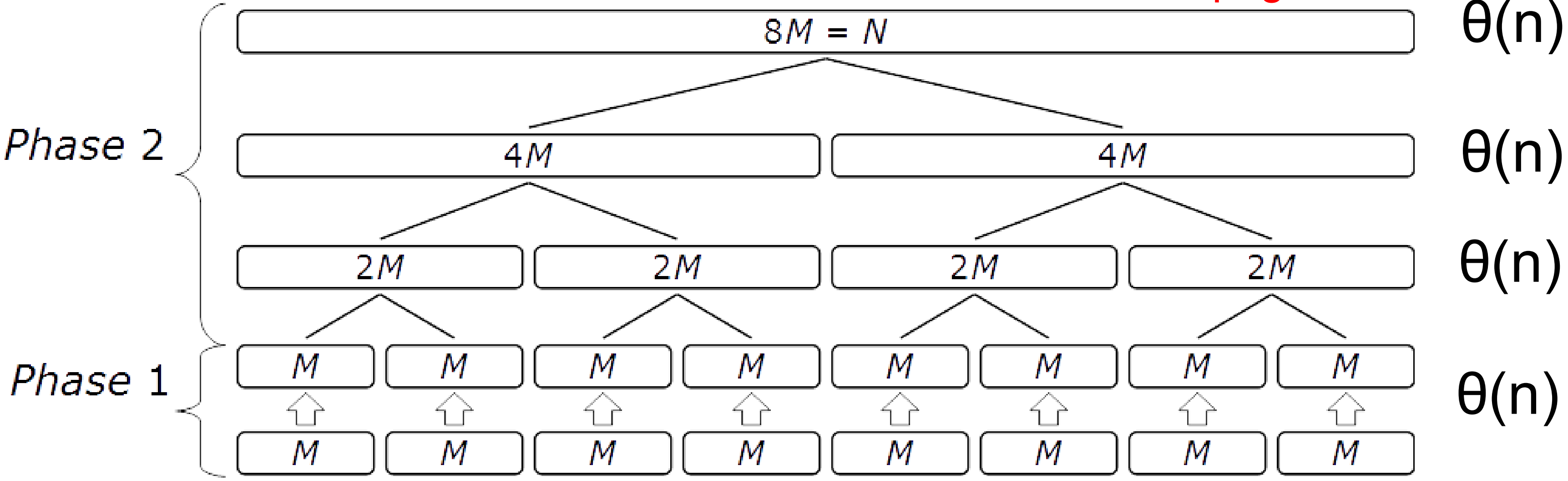


Analysis

- Phase 1: n=N/B reads and n=N/B writes. Thus θ(n).
- Phase 2:
  - Each iteration: n reads and n writes. Thus θ(n).
  - How many iterations:
    - We start with $\lceil N/M \rceil = \lceil n/m \rceil$ runs and stop with only 1 run.
    - Each iteration we reduce the number of runs by half.
    - θ(lg(n/m)) iterations in total.
  - Thus, θ(nlg(n/m)) read/write.

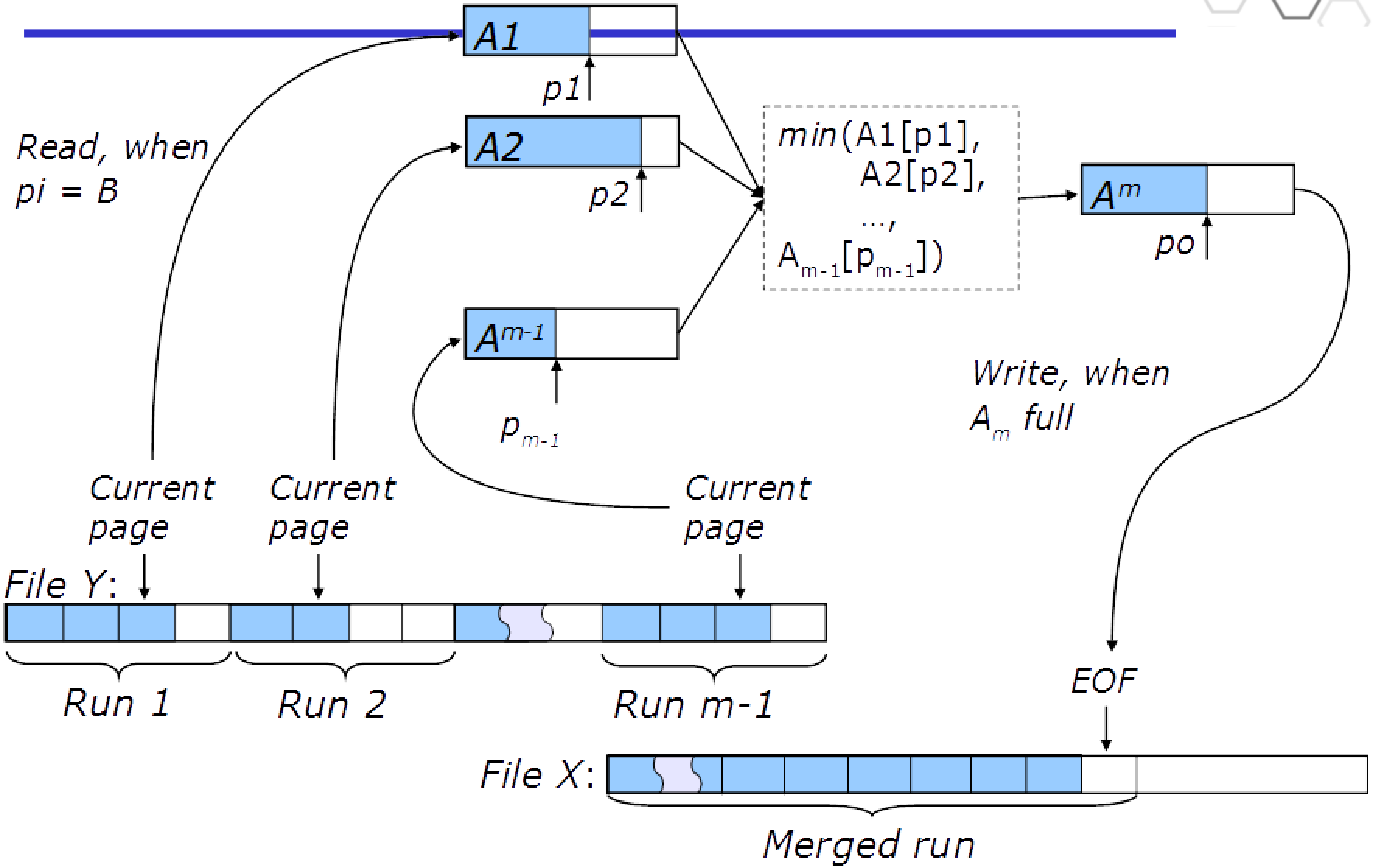
We can do better!

Observation: Although phase 1 uses all available memory, phase 2 uses just 3 pages out of *m* available pages!



External multi-way merge sort

- In phase 2, we use ***multi-way*** merge to merge ***m-1*** runs into ***one longer*** run.
  - Instead of using ***only three*** main-memory pages of size *B,* we use ***all m*** main-memory pages of size B.
  - Copy one page from the first run to A_1
  - Copy one page from the second run to A_2.
  - Copy one page from the third run to A_3.
  - …
  - Copy one page from the (m-1)-th run to A_{m-1}.
  - Merge A_1 A_2,…, A_{m-1} to A_m using main-memory merge.
  - Whenever A_m is full, store A_m to file X.
- In our running example, we can afford merging 39 runs into one longer run in one iteration.
  - We only have 8 runs in total. So only one iteration is needed.
  - In contrast, we need 3 iterations when using two-way merge sort.



A1

p1

A2

p2

A^{m-1}

p_{m-1}

Read, when
$p_i = B$

$min(A1[p1], A2[p2], \ldots, A_{m-1}[p_{m-1}])$

A^m

po

Write, when
A_m full

Current page

Current page

Current page

File Y:

Run 1

Run 2

Run m-1

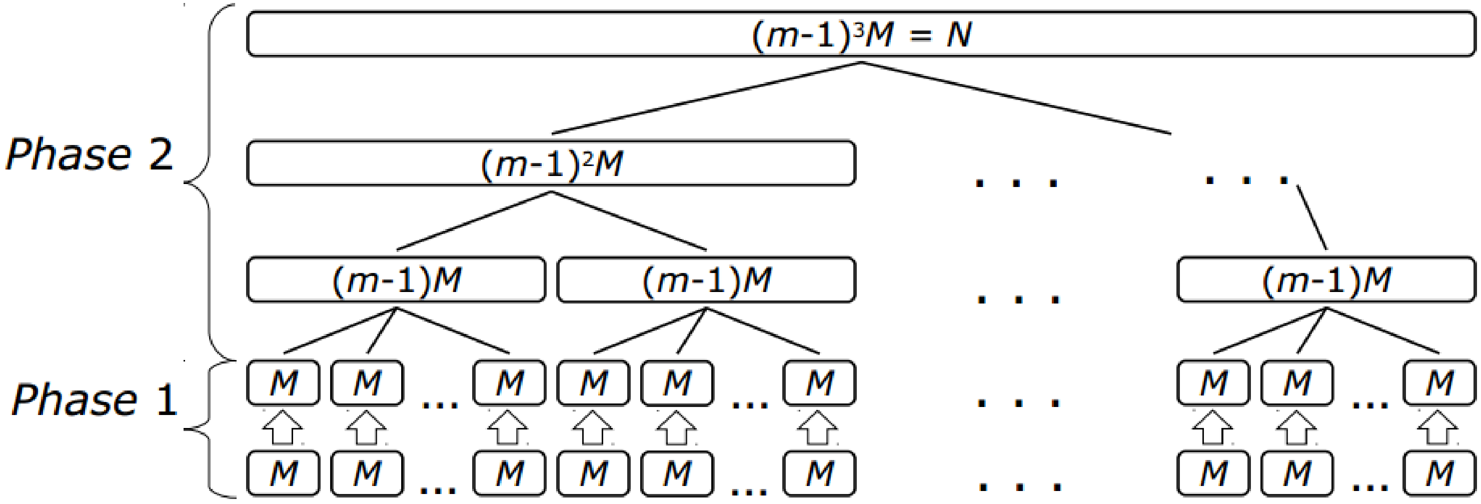
EOF

File X:

Merged run

External multi-way merge sort

- In most cases, if the input file is ***not extremely large***, we only need to do phase 2 only once.
  - Refer to the reading materials on Moodle to get an idea about what is “extremely large”.
  - θ(n)
- If the input file is extremely large, we have to do multi-way merge $\log_{m-1}(n/m) = \theta(\log_m n)$ times.
  - $\theta(n \log_m n)$



ILO of Lecture 9

- External memory algorithms and data structures
  - 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.