External Memory Algorithms for Geometric Problems

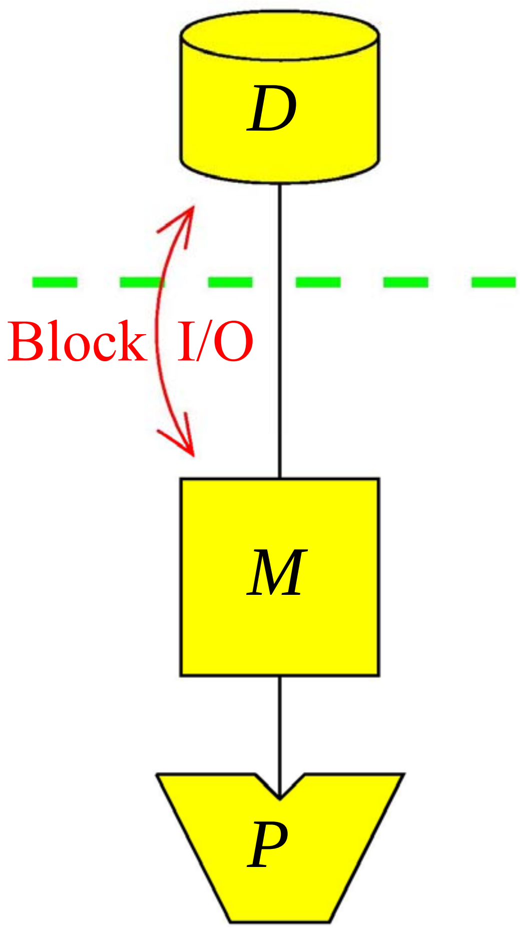
Piotr Indyk

(slides partially by Lars Arge and Jeff Vitter)

Compared to Previous Lectures

- Another way to tackle large data sets
- Exact solutions (no more embeddings)

External Memory Model



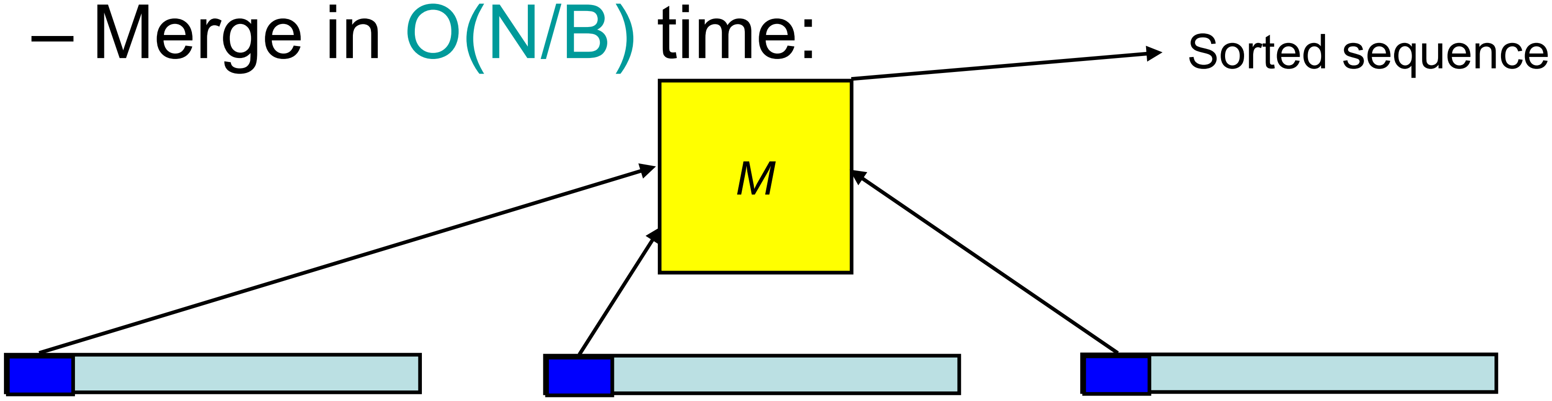
- Parameters:
  - *N* : Elements in structure
  - *B* : Elements per block
  - *M* : Elements in main memory

Today

- Sorting: $O(N/B \log_M N)$ time
- 1D data structure for searching in external memory (B-tree): $O(\log_B N)$ time
- 2D problem: finding all intersections among a set of horizontal and vertical segments: $O(N/B \log_{M/B} N)$ time

Sorting

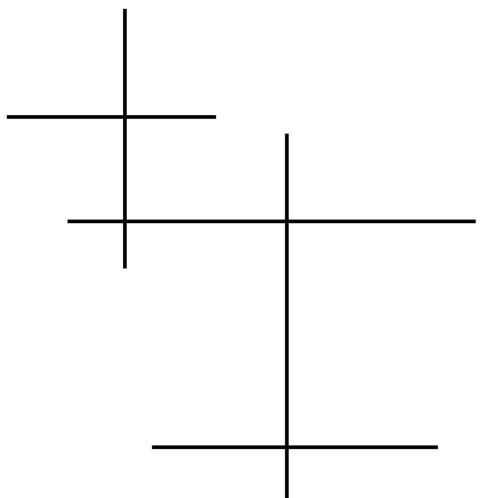
- M/B-way merge sort:
  - Split N elements into K=M/B sequences
  - Sort recursively
  - Merge in O(N/B) time:



- Recurrence: $T(N)= K\ T(N/K)+O(N/B)$
- $T(N)=O(N/B \log_K N)$

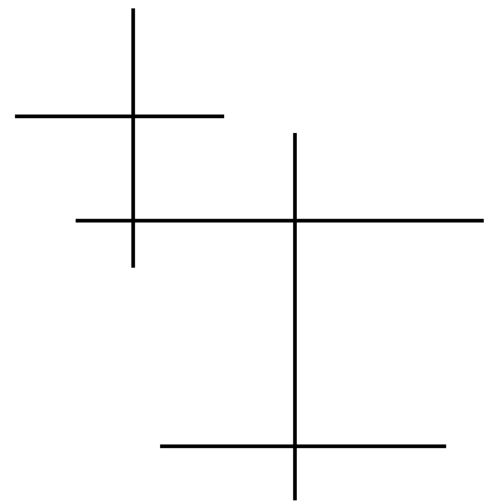
Horizontal/Vertical Line Intersection

- Given: a set of N horizontal and vertical line segments
- Goal: find all H/V intersections
- Assumption: all x and y coordinates of endpoints different



Main Memory Algorithm

- Presort the points in y-order
- Sweep the plane top down with a horizontal line
- When reaching a V-segment, store its x value in a tree. When leaving it, delete the x value from the tree
- Invariant: the balanced tree stores the V-segments hit by the sweep line
- When reaching an H-segment, search (in the tree) for its endpoints, and report all values/segments in between
- Total time is $O(N \log N + P)$



External Memory Issues

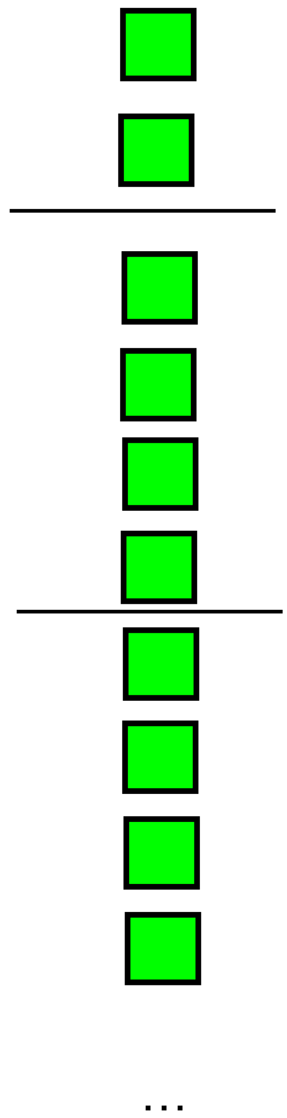
- Can use B-tree as a search tree: $O(N \log_B N)$ operations
- Still much worse than the $O(N/B * \log_{M/B} N)$ sorting bound

1D Version of the Intersection Problem

- Given: a set of N 1D horizontal and vertical line segments (i.e., intervals and points on a line)
- Goal: find all point/segment intersections
- Assumption: all x coordinates of endpoints different

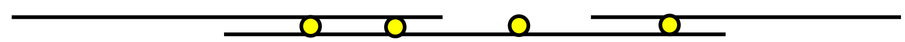
Interlude: External Stack

- Stack:
  - Push
  - Pop
- Can implement a stack in external memory using O(P/B) I/O's per P operations
  - Always keep ≤2B top elements in main memory
  - Perform disk access only when it is "earned":
    - Read when necessary
    - Write when starting a new block



Back to 1D Intersection Problem

- Will use fast stack and sorting implementations

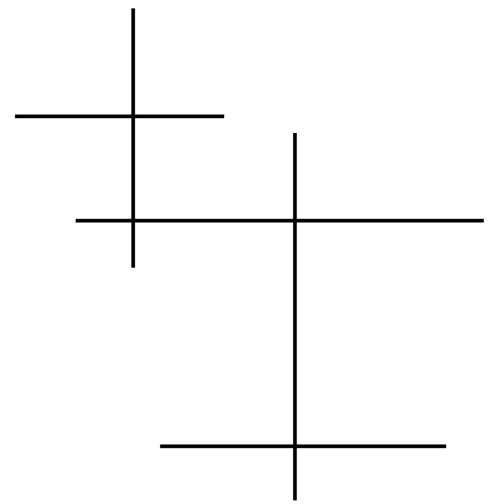


- Sort all points and intervals in x-order (of the left endpoint)
- Iterate over consecutive (end)points p
  - If p is a left endpoint of I, add I to the stack S
  - If p is a point, pop all intervals I from stack S and push them on stack S' , while:
    - Eliminating all "dead" intervals
    - Reporting all "alive" intervals
  - Push the intervals back from S' to S

Analysis

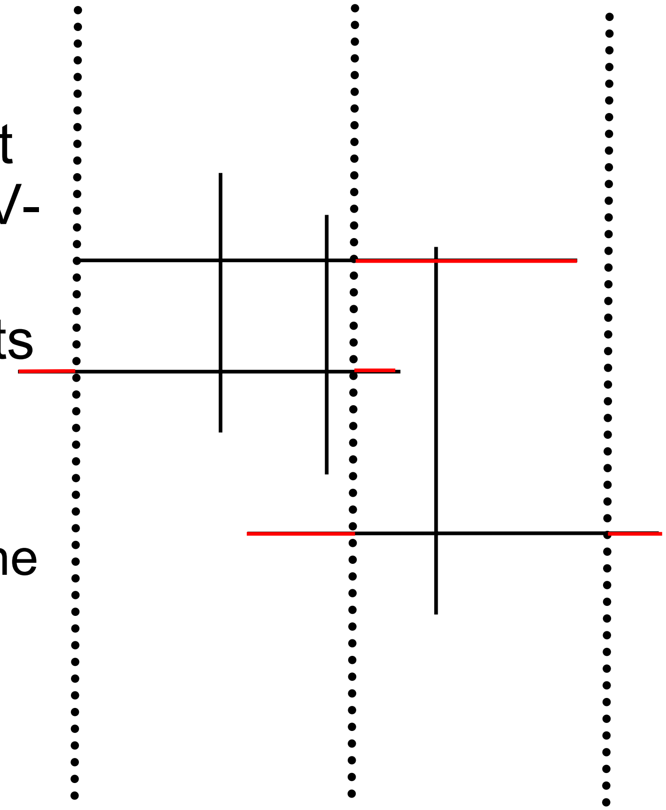
- Sorting: $O(N/B * \log_{M/B} N)$ I/O's
- Each interval is pushed/popped when:
  - An intersection is reported, or
  - Is eliminated as "dead"
- Total stack operations: $O(N+P)$
- Total stack I/O's: $O((N+P)/B)$

Back to the 2D Case



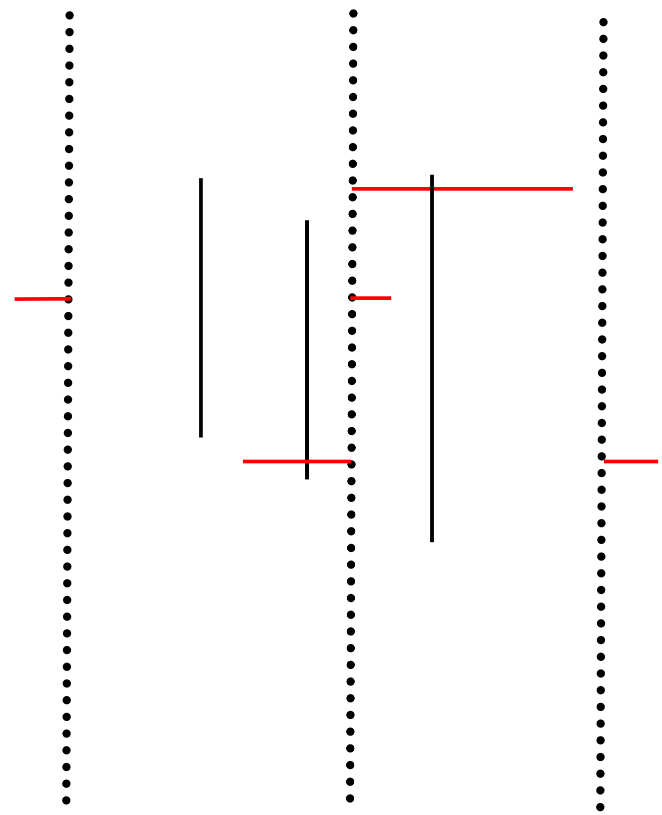
Algorithm

- Divide the x-range into M/B *slabs,* so that each slab contains the same number of V-segments
- Each slab has a stack storing V-segments
- Sort all segments in the y-order
- For each segment I:
  - If I is a V-segment, add I to the stack in the proper slab
  - If I is an H-segment, then for all slabs S which intersect I:
    - If I *spans* S, proceed as in the 1D case
    - Otherwise, store the intersection of S and I for later
- For each slab, recurse on the segments stored in that slab



The recursion

- For each slab *separately* we apply the same algorithm
- On the bottom level we have only one V-segment, which is easy to handle
- Recursion depth: $\log_{M/B} N$

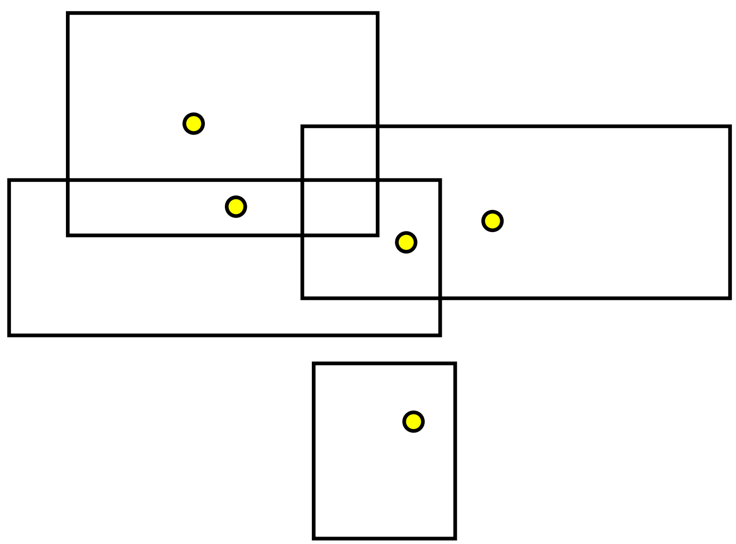


Analysis

- Initial presorting: $O(N/B * \log_{M/B} N)$ I/O's
- First level of recursion:
  - At most $O(N+P)$ pop/push operations
  - At most 2N of H-segments stored
  - Total: $O((N+P)/B)$ I/O's
- Further recursion levels:
  - The total number of H-segment pieces (over *all* slabs) is at most twice the number of the input H-segments; it does *not* double at each level
  - By the above argument we pay $O(N/B)$ I/O's per level
- Total: $O(P/B+N/B * \log_{M/B} N)$ I/O's

Off-line Range Queries

- Given: N points in 2D and N’ rectangles
- Goal: Find all pairs p, R such that p is in R



Summary

- On-line queries: $O(\log_B N)$ I/O's
- Off-line queries: $O(1/B * \log_{M/B} N)$ I/O's amortized
- Powerful techniques:
  - Sorting
  - Stack
  - Distribution sweep

References

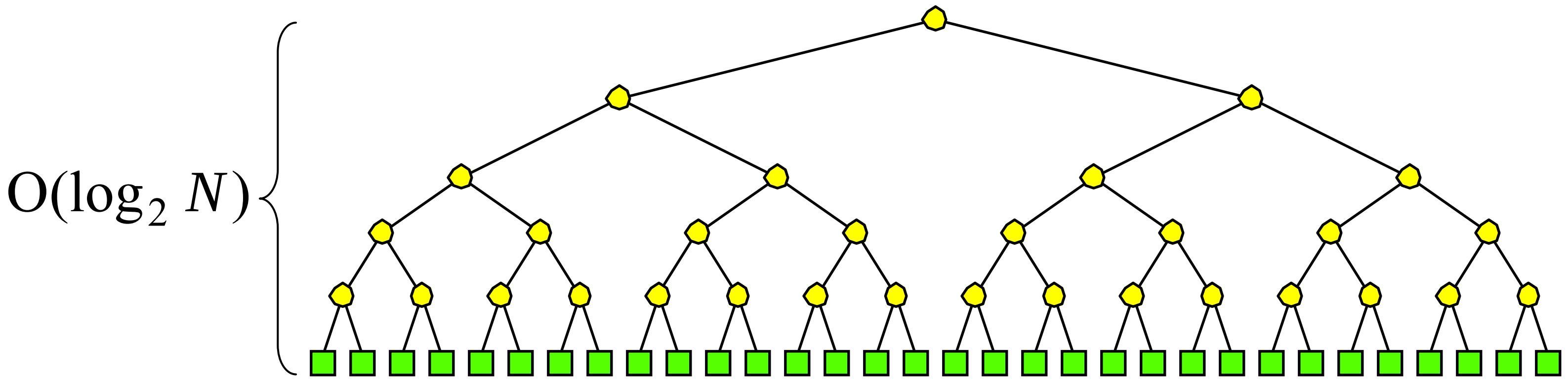
- See http://www.brics.dk/MassiveData02, especially:
  - First lecture by Lars Arge (for B-trees etc)
  - Second lecture by Jeff Vitter (for distribution sweep)

Searching in External Memory

- Dictionary (or successor) data structure for 1D data:
  - Maintains elements (e.g., numbers) under insertions and deletions
  - Given a *key* K, reports the successor of K; i.e., the smallest element which is greater or equal to K

Search Trees

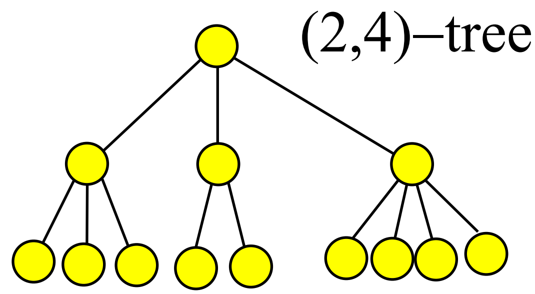
- Binary search tree:
  - Standard method for search among *N* elements
  - We assume elements in leaves



  - Search traces at least one root-leaf path

(a,b)-tree (or B-tree)

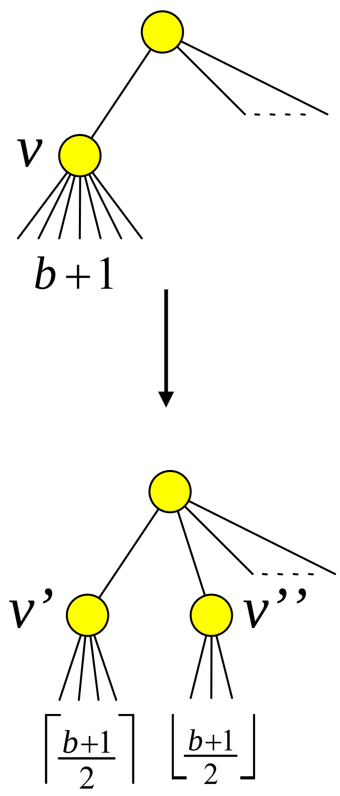
- *T* is an (*a*,*b*)-tree ($a \geq 2$ and $b \geq 2a-1$)
  - All leaves on the same level
  - Except for the root, all nodes have degree between *a* and *b*
  - Root has degree between 2 and *b*
- Choose $a,b=\Theta(B)$ to have
  - Depth: $O(\log_B N)$
  - Space: $O(N/B)$ blocks



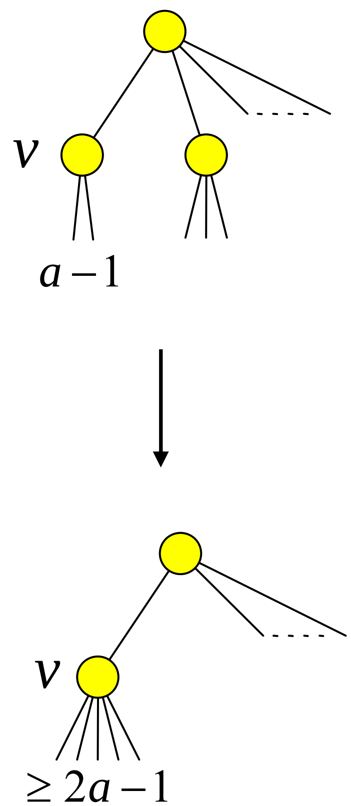
(*a*,*b*)-Tree Insert

- Insert:
  - Search and insert element in leaf *v*
  - *WHILE* *v* has *b+1* elements

    Split *v*:
    - make nodes *v'* and *v''* with (b+1)/2 elements each
    - insert element in *parent* (make new root if necessary)
    - *v=parent(v)*
- Insert touches $O(\log_B N)$ nodes
- Delete is analogous



(a,b)-Tree Delete



B-trees

- Used everywhere in databases
- Typical depth is 3 or 4
- Top two levels kept in main memory – only 1-2 I/O's per element