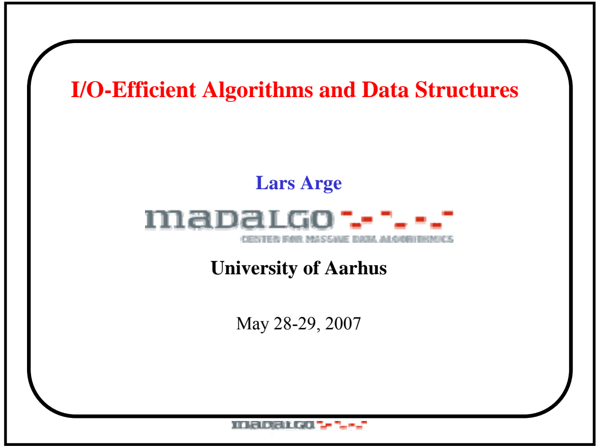
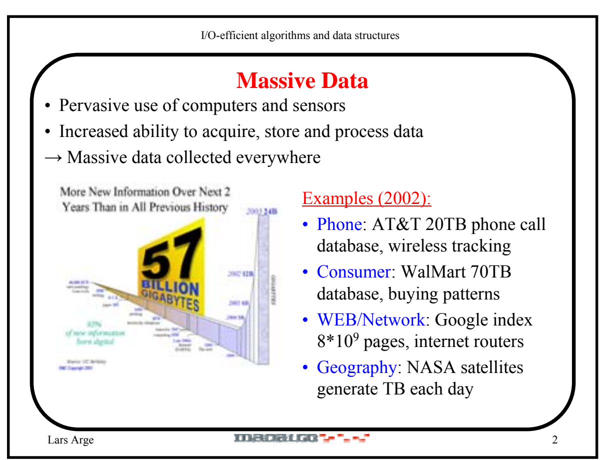
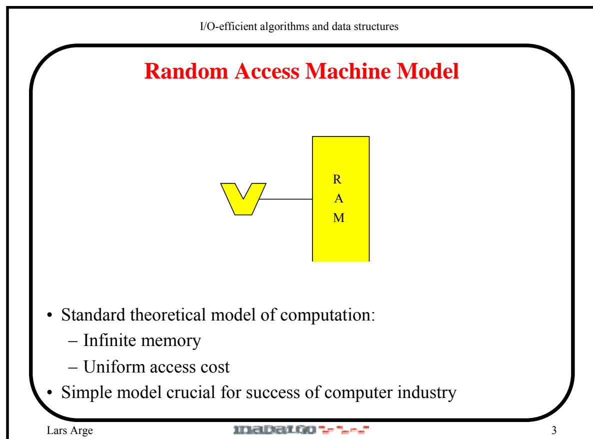
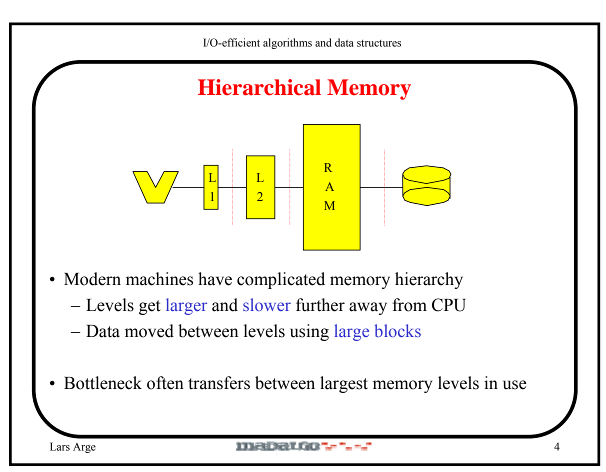
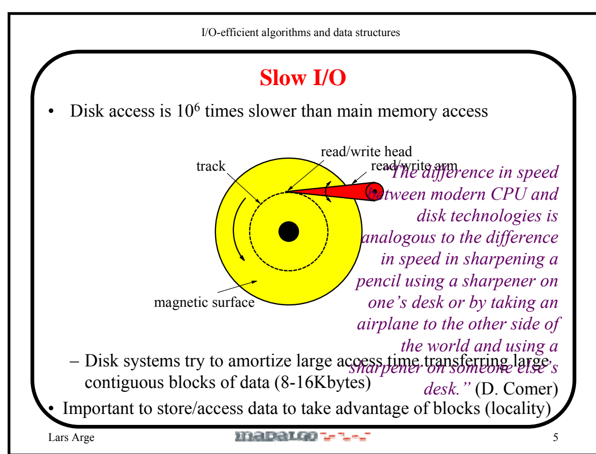
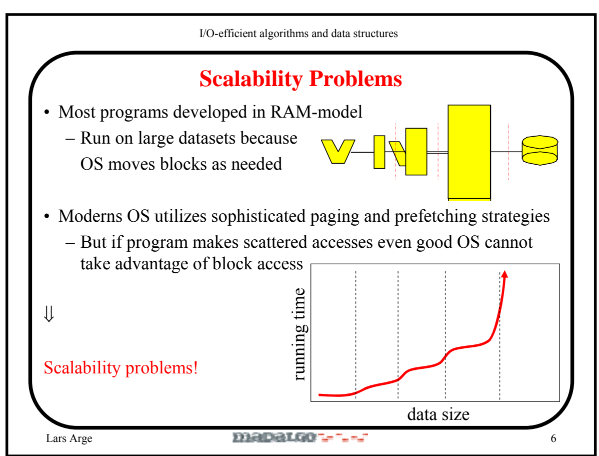

I/O-Efficient Algorithms and Data Structures
Lars Arge
University of Aarhus
May 28-29, 2007

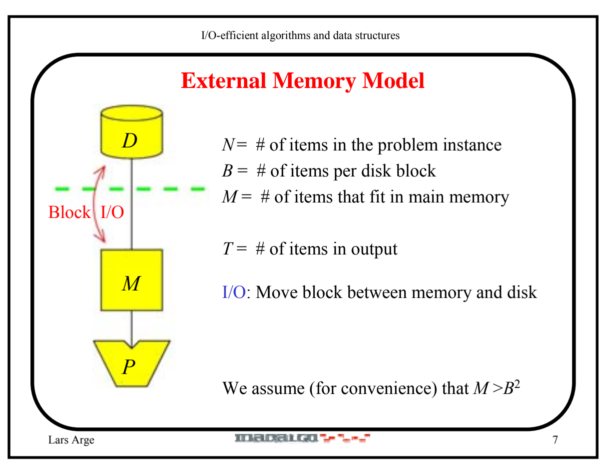

I/O-efficient algorithms and data structures
Massive Data
• Pervasive use of computers and sensors
• Increased ability to acquire, store and process data
→ Massive data collected everywhere
More New Information Over Next 2 Years Than in All Previous History
57 BILLION GIGABYTES
Examples (2002):
• Phone: AT&T 20TB phone call database, wireless tracking
• Consumer: WalMart 70TB database, buying patterns
• WEB/Network: Google index 8*10^9 pages, internet routers
• Geography: NASA satellites generate TB each day
Lars Arge
2

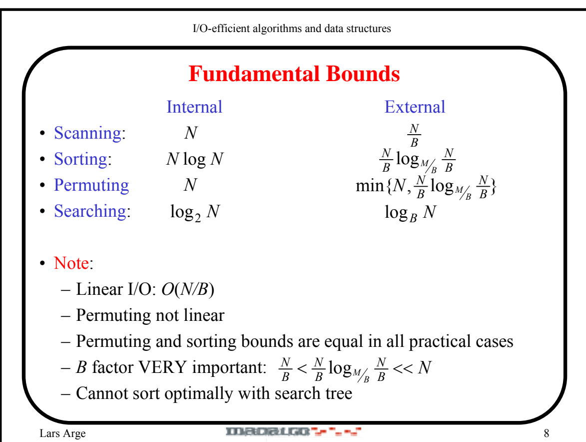

I/O-efficient algorithms and data structures
Random Access Machine Model
R A M
• Standard theoretical model of computation:
– Infinite memory
– Uniform access cost
• Simple model crucial for success of computer industry
Lars Arge
3

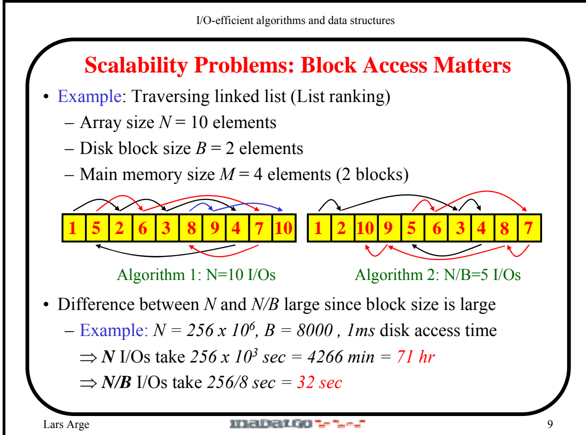

I/O-efficient algorithms and data structures
Hierarchical Memory
L 1
L 2
R A M
• Modern machines have complicated memory hierarchy
– Levels get larger and slower further away from CPU
– Data moved between levels using large blocks
• Bottleneck often transfers between largest memory levels in use
Lars Arge
4

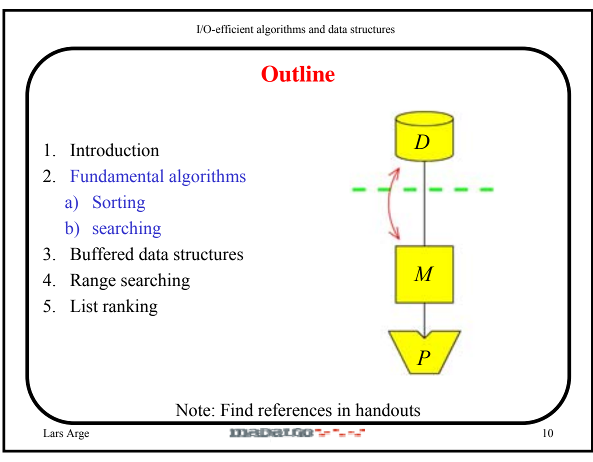

I/O-efficient algorithms and data structures
Slow I/O
• Disk access is 10^6 times slower than main memory access
track
read/write head
read/write arm
magnetic surface
"The difference in speed between modern CPU and disk technologies is analogous to the difference in speed in sharpening a pencil using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk." (D. Comer)
– Disk systems try to amortize large access time transferring large contiguous blocks of data (8-16Kbytes)
• Important to store/access data to take advantage of blocks (locality)
Lars Arge
5

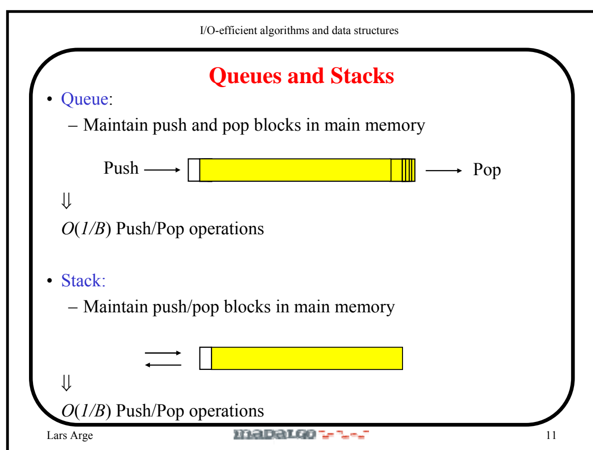

I/O-efficient algorithms and data structures
Scalability Problems
• Most programs developed in RAM-model
– Run on large datasets because OS moves blocks as needed
• Moderns OS utilizes sophisticated paging and prefetching strategies
– But if program makes scattered accesses even good OS cannot take advantage of block access
⇓
Scalability problems!
running time
data size
Lars Arge
6

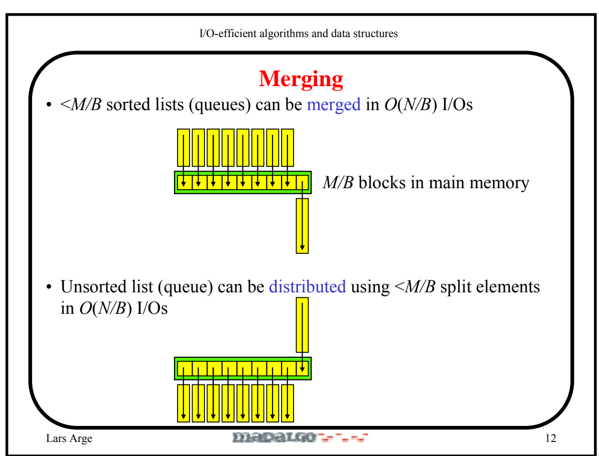

I/O-efficient algorithms and data structures
External Memory Model
D
Block I/O
M
P
N = # of items in the problem instance
B = # of items per disk block
M = # of items that fit in main memory
T = # of items in output
I/O: Move block between memory and disk
We assume (for convenience) that $M > B^2$
Lars Arge
7

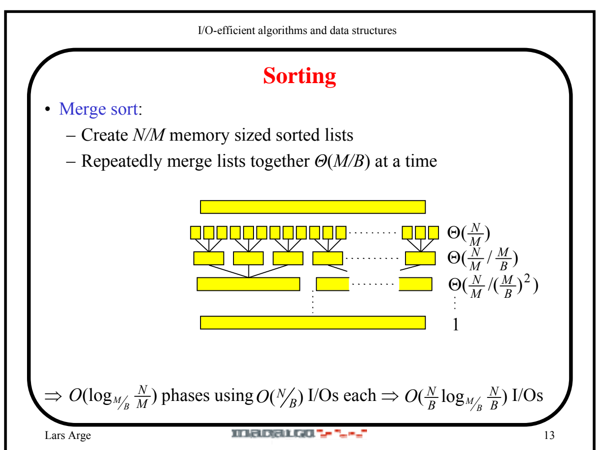

I/O-efficient algorithms and data structures
Fundamental Bounds
Internal
External
• Scanning: N $\frac{N}{B}$
• Sorting: $N \log N$ $\frac{N}{B} \log_{M/B} \frac{N}{B}$
• Permuting N $\min\{N, \frac{N}{B} \log_{M/B} \frac{N}{B}\}$
• Searching: $\log_2 N$ $\log_B N$
• Note:
– Linear I/O: $O(N/B)$
– Permuting not linear
– Permuting and sorting bounds are equal in all practical cases
– B factor VERY important: $\frac{N}{B} < \frac{N}{B} \log_{M/B} \frac{N}{B} << N$
– Cannot sort optimally with search tree
Lars Arge
8

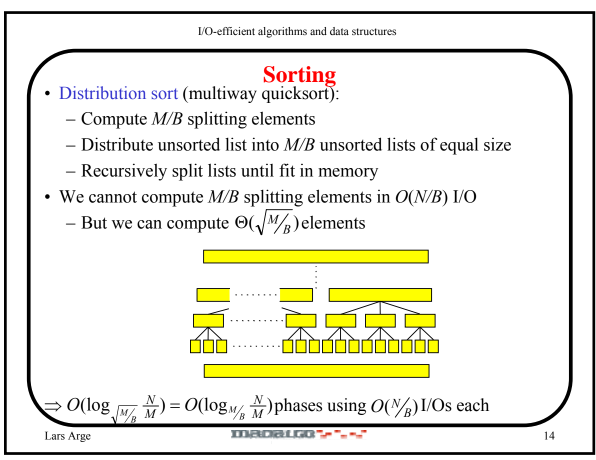

I/O-efficient algorithms and data structures
Scalability Problems: Block Access Matters
• Example: Traversing linked list (List ranking)
– Array size N = 10 elements
– Disk block size B = 2 elements
– Main memory size M = 4 elements (2 blocks)
1 5 2 6 3 8 9 4 7 10
1 2 10 9 5 6 3 4 8 7
Algorithm 1: N=10 I/Os
Algorithm 2: N/B=5 I/Os
• Difference between N and N/B large since block size is large
– Example: $N = 256 \times 10^6$, $B = 8000$, $1ms$ disk access time
⇒ N I/Os take 256×10^3 sec = 4266 min = 71 hr
⇒ N/B I/Os take 256/8 sec = 32 sec
Lars Arge
9

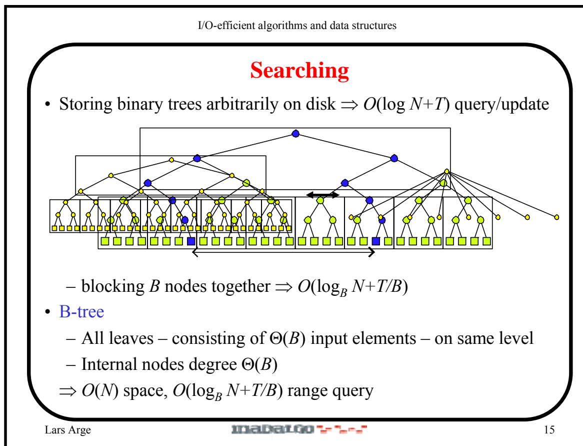

I/O-efficient algorithms and data structures
Outline
1. Introduction
2. Fundamental algorithms
a) Sorting
b) searching
3. Buffered data structures
4. Range searching
5. List ranking
D
M
P
Note: Find references in handouts
Lars Arge
10

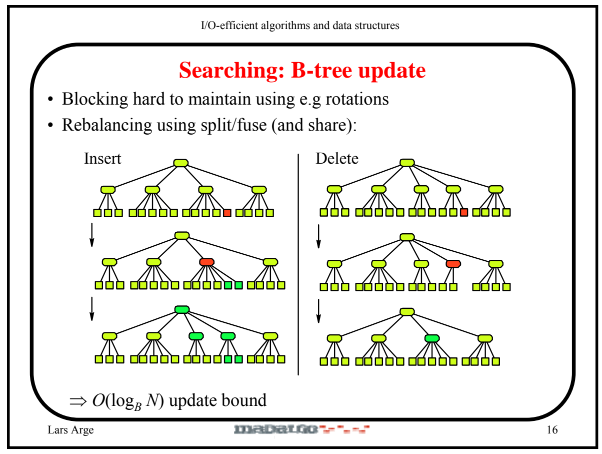

I/O-efficient algorithms and data structures
Queues and Stacks
• Queue:
– Maintain push and pop blocks in main memory
Push
Pop
⇓
$O(1/B)$ Push/Pop operations
• Stack:
– Maintain push/pop blocks in main memory
⇓
$O(1/B)$ Push/Pop operations
Lars Arge
11

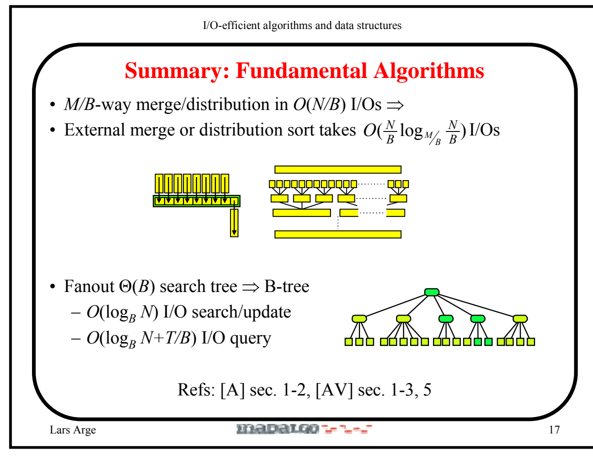

I/O-efficient algorithms and data structures
Merging
• $<M/B$ sorted lists (queues) can be merged in $O(N/B)$ I/Os
M/B blocks in main memory
• Unsorted list (queue) can be distributed using $<M/B$ split elements in $O(N/B)$ I/Os
Lars Arge
12

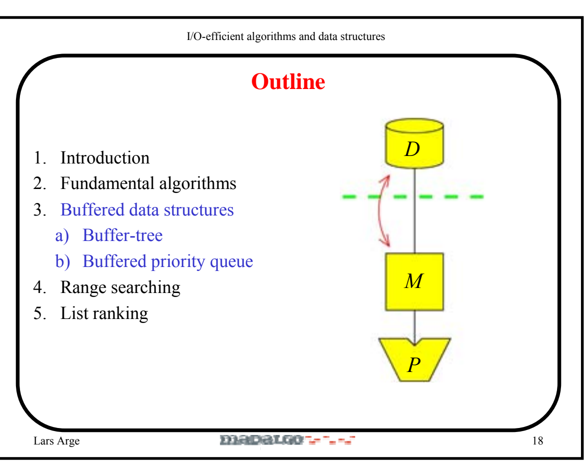

I/O-efficient algorithms and data structures
Sorting
• Merge sort:
– Create N/M memory sized sorted lists
– Repeatedly merge lists together Θ(M/B) at a time
$\Theta(\frac{N}{M})$
$\Theta(\frac{N}{M} / \frac{M}{B})$
$\Theta(\frac{N}{M} / (\frac{M}{B})^2)$
1
$\Rightarrow O(\log_{M/B} \frac{N}{M})$ phases using $O(N/B)$ I/Os each $\Rightarrow O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
Lars Arge
13


I/O-efficient algorithms and data structures
Sorting
• Distribution sort (multiway quicksort):
– Compute M/B splitting elements
– Distribute unsorted list into M/B unsorted lists of equal size
– Recursively split lists until fit in memory
• We cannot compute M/B splitting elements in O(N/B) I/O
– But we can compute $\Theta(\sqrt{M/B})$ elements
$\Rightarrow O(\log_{\sqrt{M/B}} \frac{N}{M}) = O(\log_{M/B} \frac{N}{M})$ phases using $O(N/B)$ I/Os each
Lars Arge
14


I/O-efficient algorithms and data structures
Searching
• Storing binary trees arbitrarily on disk ⇒ O(log N+T) query/update
– blocking B nodes together ⇒ $O(\log_B N+T/B)$
• B-tree
– All leaves – consisting of Θ(B) input elements – on same level
– Internal nodes degree Θ(B)
⇒ O(N) space, $O(\log_B N+T/B)$ range query
Lars Arge
15


I/O-efficient algorithms and data structures
Searching: B-tree update
• Blocking hard to maintain using e.g rotations
• Rebalancing using split/fuse (and share):
Insert
Delete
⇒ $O(\log_B N)$ update bound
Lars Arge
16


I/O-efficient algorithms and data structures
Summary: Fundamental Algorithms
• M/B-way merge/distribution in O(N/B) I/Os ⇒
• External merge or distribution sort takes $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
• Fanout Θ(B) search tree ⇒ B-tree
– $O(\log_B N)$ I/O search/update
– $O(\log_B N+T/B)$ I/O query
Refs: [A] sec. 1-2, [AV] sec. 1-3, 5
Lars Arge
17

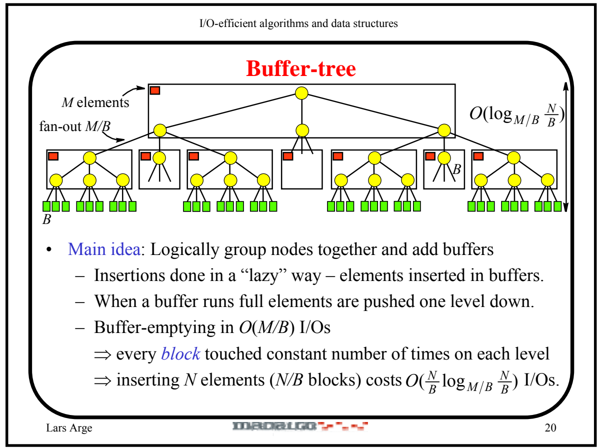

I/O-efficient algorithms and data structures
Outline
1. Introduction
2. Fundamental algorithms
3. Buffered data structures
a) Buffer-tree
b) Buffered priority queue
4. Range searching
5. List ranking
D
M
P
Lars Arge
18

Buffered Data Structures

- Use of the (on-line) efficient B-tree in external memory algorithms does not lead to efficient algorithms
- Example: Sorting using search tree
  - Insert all elements in search tree one-by-one (construct tree)
  - Output in sorted order using in-order traversal

  $\Rightarrow$ Optimal $O(N \log N)$ time in internal memory

  $\Rightarrow$ non-optimal $O(N \log_B N)$ I/Os in external memory
- Need $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ operations to obtain efficient algorithms
  - $O(N)\cdot O(\frac{1}{B}\log_{M/B}\frac{N}{B}) = O(\frac{N}{B}\log_{M/B}\frac{N}{B})$



Buffer-tree

- Insert (and deletes) on buffer-tree takes $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os amortized

  $\Rightarrow$ Buffer tree can be used in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ sorting algorithm
- One-dim. rangesearch operations can also be supported in $O(\frac{1}{B}\log_{M/B}\frac{N}{B}+\frac{T}{B})$ I/Os amortized
  - Search elements handle lazily like updates
  - All elements in relevant sub-trees reported during buffer-emptying
  - Buffer-emptying in $O(X/B+T'/B)$, where T' is reported elements

Buffered Priority Queue

- Buffer-tree can also be used in external priority queue
- To delete minimal element
  - Empty all buffers on leftmost path
  - Delete M elements in leftmost leaves and keep in memory
  - (Insertions checked against minimal elements)

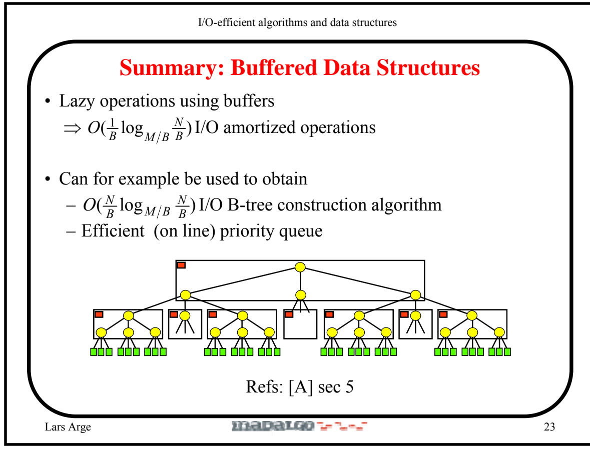
$\Theta(\frac{M}{B})$

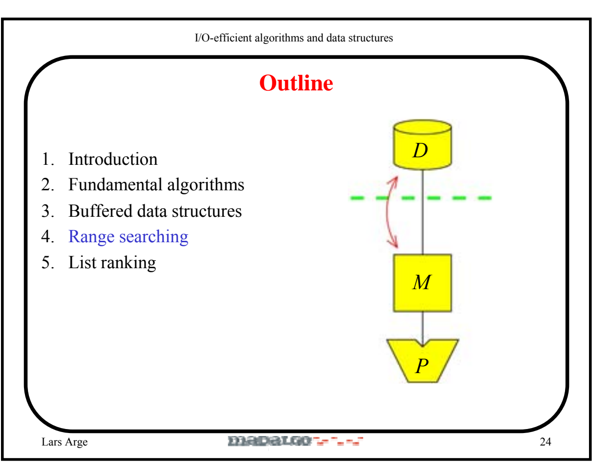
B

$\Downarrow$

$O(\frac{M}{B}\log_{M/B}\frac{N}{B})$ I/Os every $O(M)$ delete $\Rightarrow$ $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized

- Buffer technique can also be used on heap and tournament tree





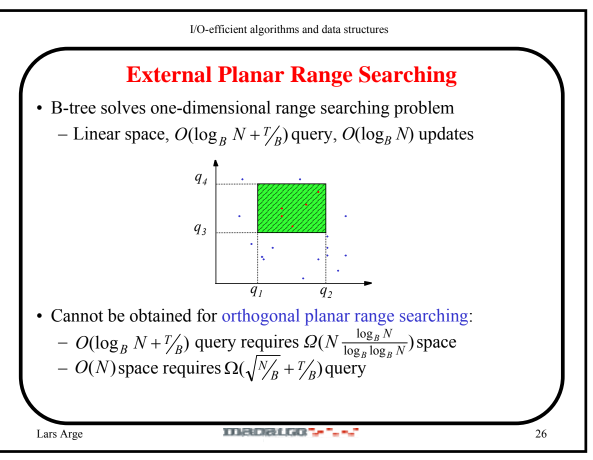
Exercises

1) Design an algorithm for removing duplicates from a multiset.
The output from the algorithm should be the K distinct elements among the N input elements in sorted order.
The algorithm should use $O(\max\{\frac{N}{B}, \frac{N}{B}\log_{M/B}\frac{N}{B} - \sum_{i=1}^{K}\frac{N_i}{B}\log_{M/B}\frac{N_i}{B}\})$ I/Os, where N_i is the number of copies of the i'th element
– *Hint:* Modify merge-sort to remove copies as soon as found

2) Design a I/O-efficient version of a heap that supports insert and deletemin operations in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os amortized.
– *Hint/one idea:* Let the heap have fanout *M/B* (rather than *2*) and store *M* minimal elements in each node (rather than one). Buffer *M* inserts in memory before performing them.

External Planar Range Searching

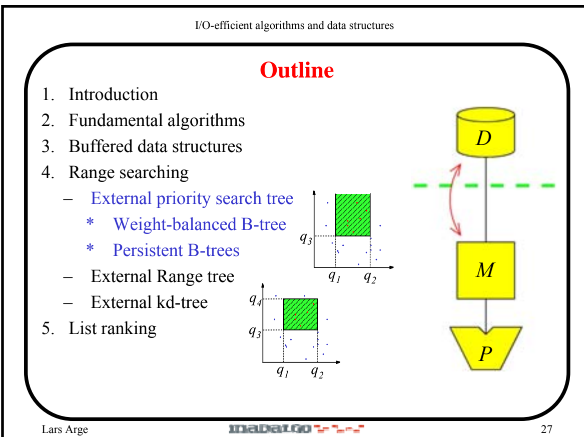
- B-tree solves one-dimensional range searching problem
  – Linear space, $O(\log_B N + T/B)$ query, $O(\log_B N)$ updates



- Cannot be obtained for orthogonal planar range searching:
  – $O(\log_B N + T/B)$ query requires $\Omega(N\frac{\log_B N}{\log_B \log_B N})$ space
  – $O(N)$ space requires $\Omega(\sqrt{N/B} + T/B)$ query

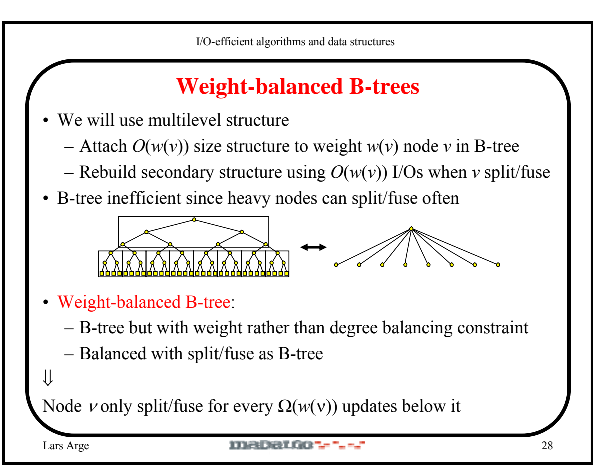
Outline

1. Introduction
2. Fundamental algorithms
3. Buffered data structures
4. Range searching
   – External priority search tree
     * Weight-balanced B-tree
     * Persistent B-trees
   – External Range tree
   – External kd-tree
5. List ranking



Weight-balanced B-trees

- We will use multilevel structure
  – Attach $O(w(v))$ size structure to weight $w(v)$ node v in B-tree
  – Rebuild secondary structure using $O(w(v))$ I/Os when v split/fuse
- B-tree inefficient since heavy nodes can split/fuse often



- Weight-balanced B-tree:
  – B-tree but with weight rather than degree balancing constraint
  – Balanced with split/fuse as B-tree

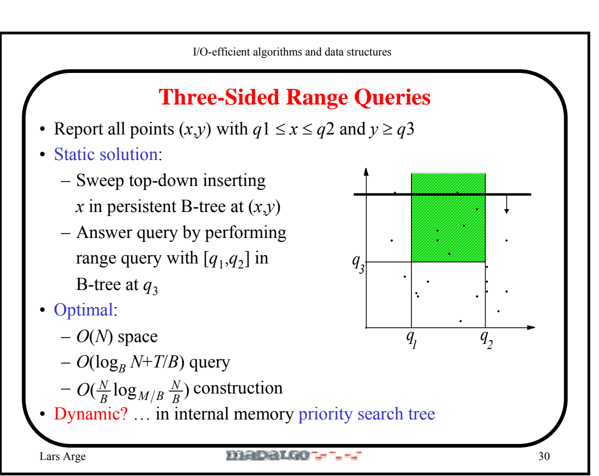
⇓
Node v only split/fuse for every $\Omega(w(v))$ updates below it

Persistent B-trees

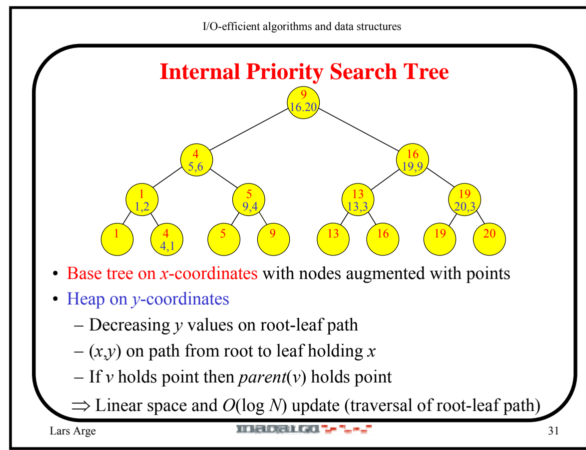
- We will use (partial) persistent B-tree
  – Update current version, query all previous versions
- Partial persistent B-tree (multi-version B-tree) can be obtained using standard techniques
  – $O(\log_B N)$ update, $O(\log_B N + T/B)$ query, $O(N)$ space
  – N is total number of operations performed
  – Batch of N updates in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os using buffer technique
- Idea:
  – Elements and nodes augmented with existence intervals
  – Maintain that every node contains $\Theta(B)$ alive elements in its existence interval

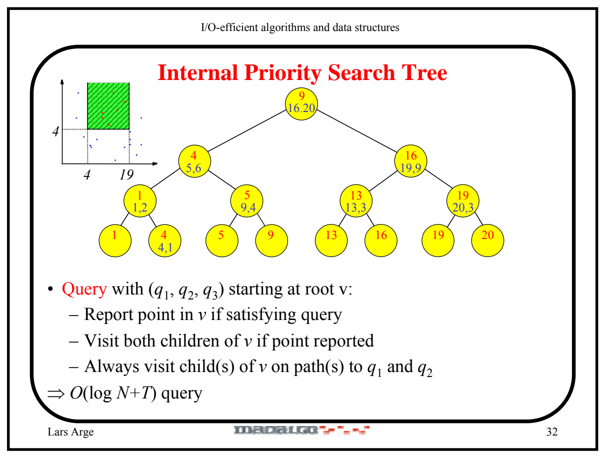
Three-Sided Range Queries

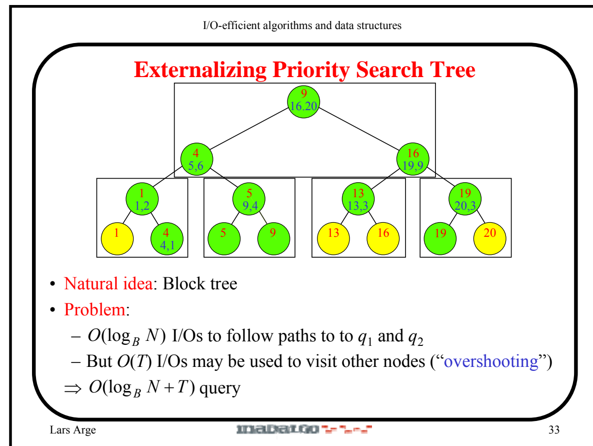
- Report all points (x,y) with $q1 \le x \le q2$ and $y \ge q3$
- Static solution:
  – Sweep top-down inserting x in persistent B-tree at (x,y)
  – Answer query by performing range query with $[q_1,q_2]$ in B-tree at q_3

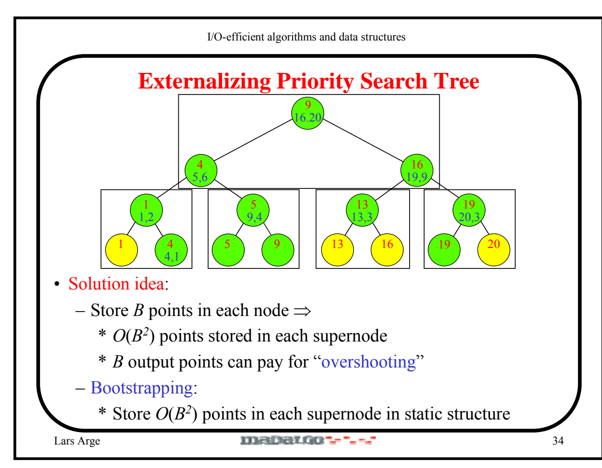


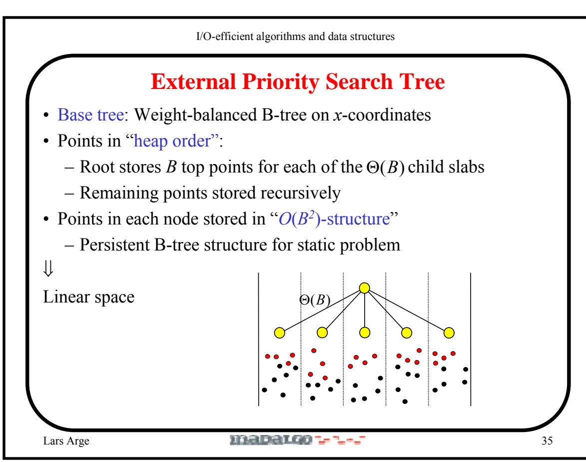
- Optimal:
  – $O(N)$ space
  – $O(\log_B N + T/B)$ query
  – $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ construction
- Dynamic? … in internal memory priority search tree

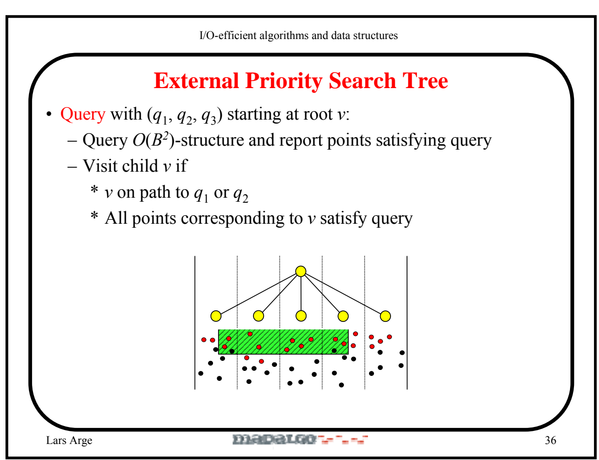

I/O-efficient algorithms and data structures
Internal Priority Search Tree
Base tree on x-coordinates with nodes augmented with points
Heap on y-coordinates
Decreasing y values on root-leaf path
(x,y) on path from root to leaf holding x
If v holds point then parent(v) holds point
⇒ Linear space and O(log N) update (traversal of root-leaf path)
Lars Arge
31

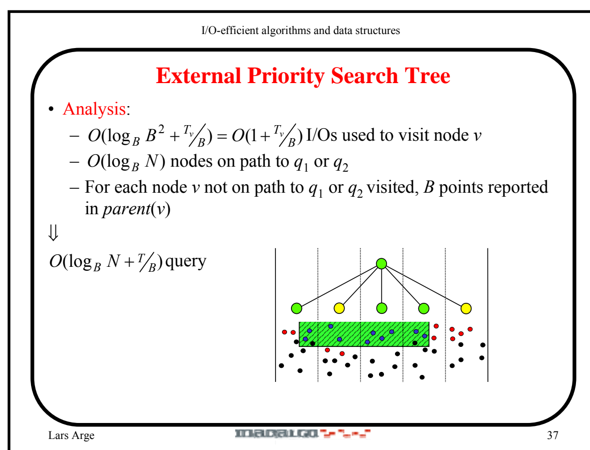

I/O-efficient algorithms and data structures
Internal Priority Search Tree
Query with (q1, q2, q3) starting at root v:
Report point in v if satisfying query
Visit both children of v if point reported
Always visit child(s) of v on path(s) to q1 and q2
⇒ O(log N+T) query
Lars Arge
32

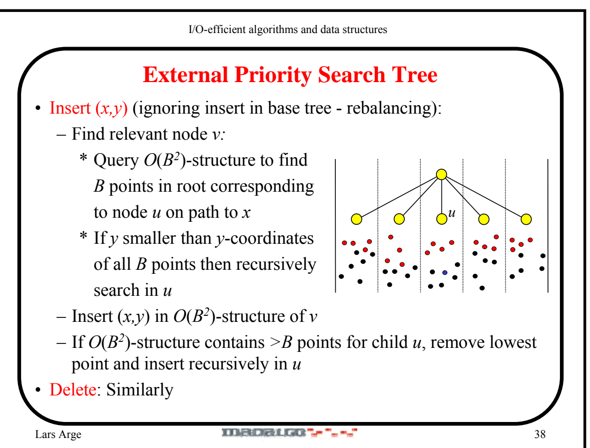

I/O-efficient algorithms and data structures
Externalizing Priority Search Tree
Natural idea: Block tree
Problem:
O(logB N) I/Os to follow paths to to q1 and q2
But O(T) I/Os may be used to visit other nodes ("overshooting")
⇒ O(logB N + T) query
Lars Arge
33

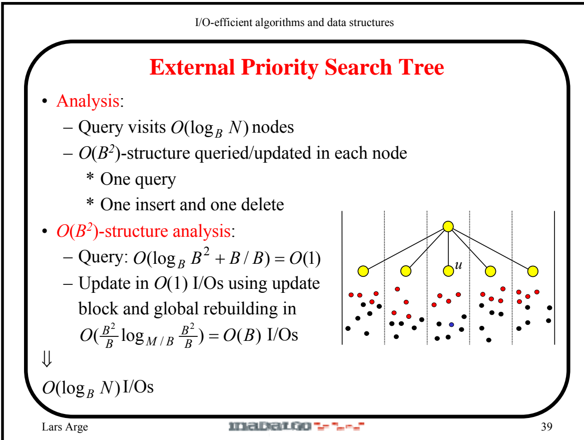

I/O-efficient algorithms and data structures
Externalizing Priority Search Tree
Solution idea:
Store B points in each node ⇒
O(B2) points stored in each supernode
B output points can pay for "overshooting"
Bootstrapping:
Store O(B2) points in each supernode in static structure
Lars Arge
34

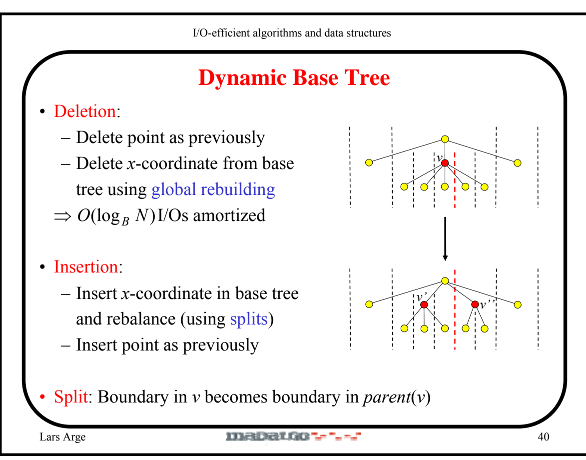

I/O-efficient algorithms and data structures
External Priority Search Tree
Base tree: Weight-balanced B-tree on x-coordinates
Points in "heap order":
Root stores B top points for each of the Θ(B) child slabs
Remaining points stored recursively
Points in each node stored in "O(B2)-structure"
Persistent B-tree structure for static problem
⇓
Linear space
Θ(B)
Lars Arge
35

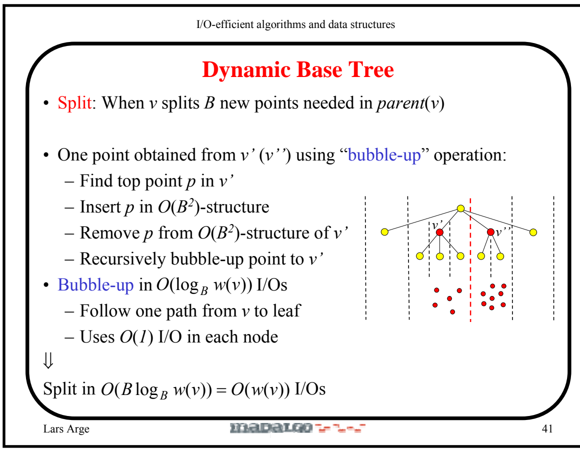

I/O-efficient algorithms and data structures
External Priority Search Tree
Query with (q1, q2, q3) starting at root v:
Query O(B2)-structure and report points satisfying query
Visit child v if
v on path to q1 or q2
All points corresponding to v satisfy query
Lars Arge
36

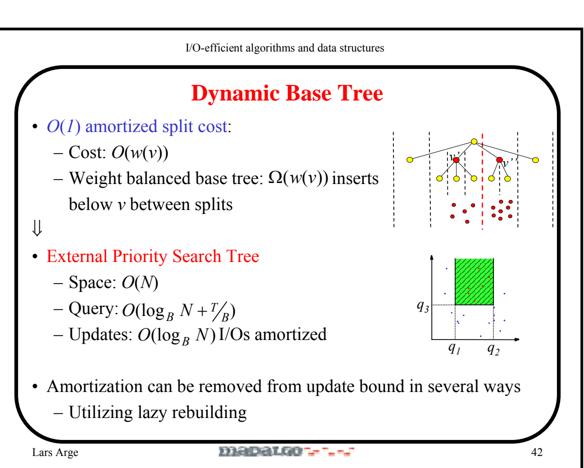

I/O-efficient algorithms and data structures
External Priority Search Tree
• Analysis:
– $O(\log_B B^2 + T_v/B) = O(1 + T_v/B)$ I/Os used to visit node v
– $O(\log_B N)$ nodes on path to q_1 or q_2
– For each node v not on path to q_1 or q_2 visited, B points reported in $parent(v)$
⇓
$O(\log_B N + T/B)$ query
Lars Arge
37

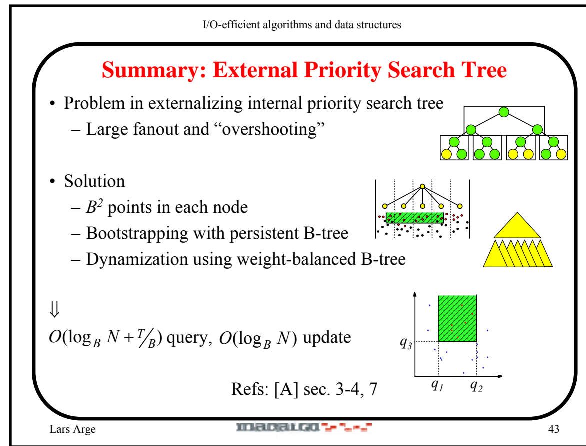

I/O-efficient algorithms and data structures
External Priority Search Tree
• Insert (x,y) (ignoring insert in base tree - rebalancing):
– Find relevant node v:
* Query $O(B^2)$-structure to find B points in root corresponding to node u on path to x
* If y smaller than y-coordinates of all B points then recursively search in u
– Insert (x,y) in $O(B^2)$-structure of v
– If $O(B^2)$-structure contains $>B$ points for child u, remove lowest point and insert recursively in u
• Delete: Similarly
u
Lars Arge
38

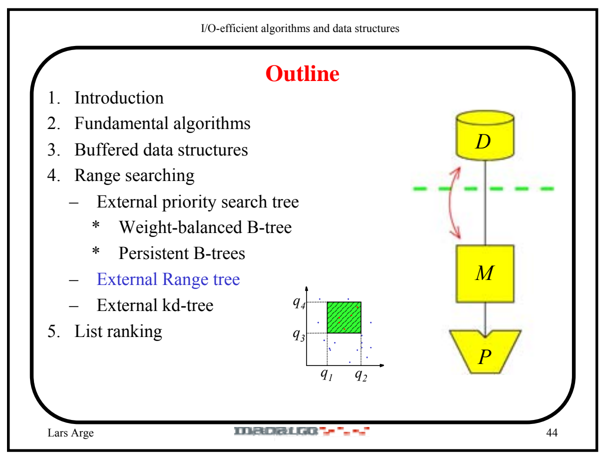

I/O-efficient algorithms and data structures
External Priority Search Tree
• Analysis:
– Query visits $O(\log_B N)$ nodes
– $O(B^2)$-structure queried/updated in each node
* One query
* One insert and one delete
• $O(B^2)$-structure analysis:
– Query: $O(\log_B B^2 + B/B) = O(1)$
– Update in $O(1)$ I/Os using update block and global rebuilding in $O(\frac{B^2}{B} \log_{M/B} \frac{B^2}{B}) = O(B)$ I/Os
⇓
$O(\log_B N)$ I/Os
u
Lars Arge
39

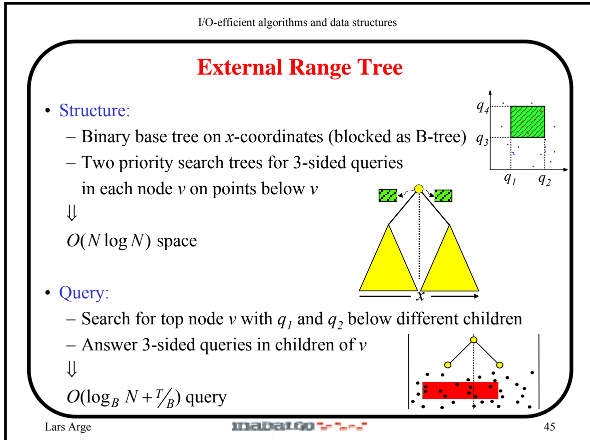

I/O-efficient algorithms and data structures
Dynamic Base Tree
• Deletion:
– Delete point as previously
– Delete x-coordinate from base tree using global rebuilding
⇒ $O(\log_B N)$ I/Os amortized
• Insertion:
– Insert x-coordinate in base tree and rebalance (using splits)
– Insert point as previously
• Split: Boundary in v becomes boundary in $parent(v)$
v
v'
v''
Lars Arge
40

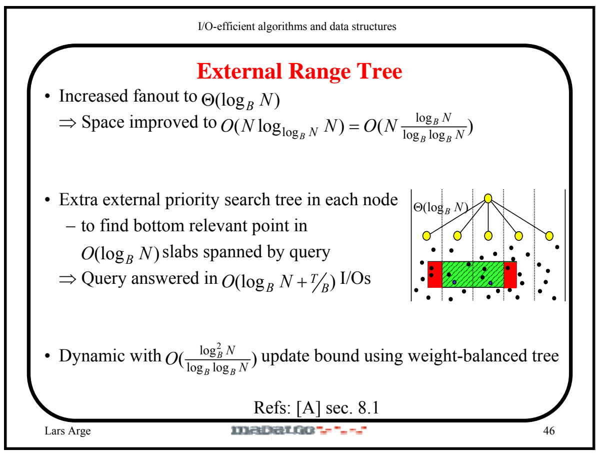

I/O-efficient algorithms and data structures
Dynamic Base Tree
• Split: When v splits B new points needed in $parent(v)$
• One point obtained from v' (v'') using "bubble-up" operation:
– Find top point p in v'
– Insert p in $O(B^2)$-structure
– Remove p from $O(B^2)$-structure of v'
– Recursively bubble-up point to v'
• Bubble-up in $O(\log_B w(v))$ I/Os
– Follow one path from v to leaf
– Uses $O(1)$ I/O in each node
⇓
Split in $O(B \log_B w(v)) = O(w(v))$ I/Os
v'
v''
Lars Arge
41

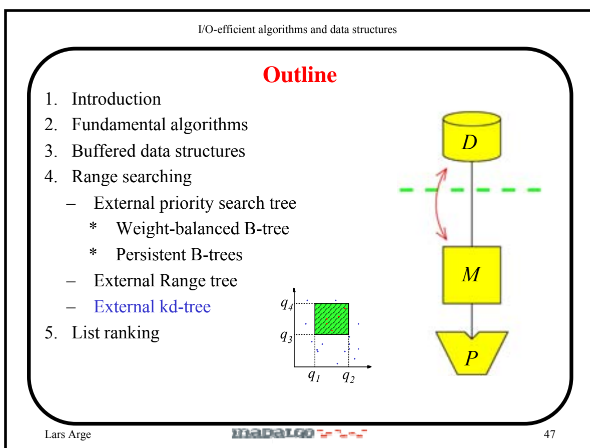

I/O-efficient algorithms and data structures
Dynamic Base Tree
• $O(1)$ amortized split cost:
– Cost: $O(w(v))$
– Weight balanced base tree: $\Omega(w(v))$ inserts below v between splits
⇓
• External Priority Search Tree
– Space: $O(N)$
– Query: $O(\log_B N + T/B)$
– Updates: $O(\log_B N)$ I/Os amortized
• Amortization can be removed from update bound in several ways
– Utilizing lazy rebuilding
q_3
q_1
q_2
Lars Arge
42

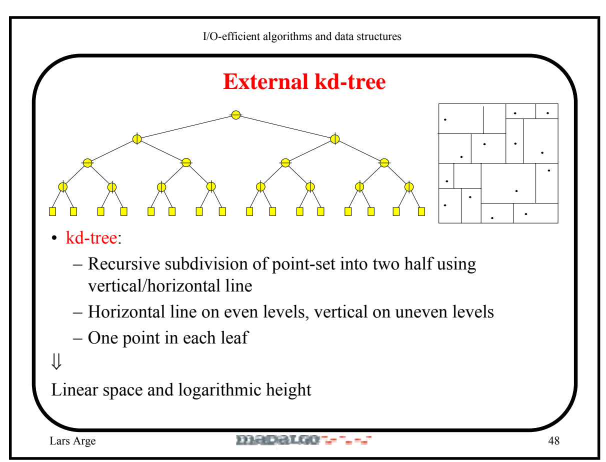

I/O-efficient algorithms and data structures
Summary: External Priority Search Tree
• Problem in externalizing internal priority search tree
– Large fanout and "overshooting"
• Solution
– B^2 points in each node
– Bootstrapping with persistent B-tree
– Dynamization using weight-balanced B-tree
⇓
$O(\log_B N + T/B)$ query, $O(\log_B N)$ update
q_3
q_1 q_2
Refs: [A] sec. 3-4, 7
Lars Arge
43

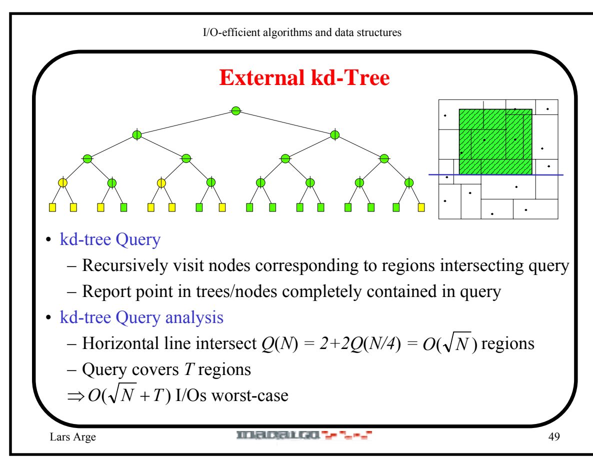

I/O-efficient algorithms and data structures
Outline
1. Introduction
2. Fundamental algorithms
3. Buffered data structures
4. Range searching
– External priority search tree
* Weight-balanced B-tree
* Persistent B-trees
– External Range tree
– External kd-tree
5. List ranking
q_4
q_3
q_1 q_2
D
M
P
Lars Arge
44

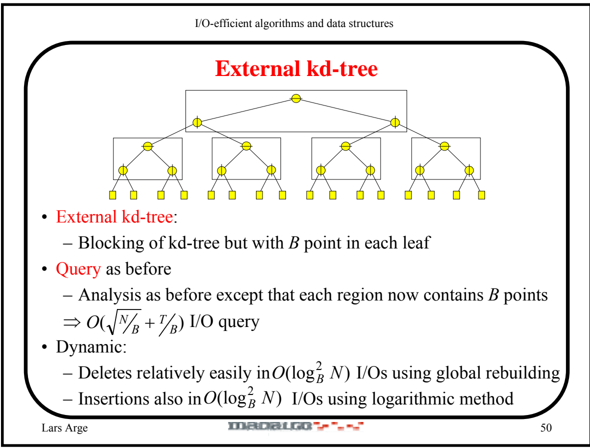

I/O-efficient algorithms and data structures
External Range Tree
• Structure:
– Binary base tree on x-coordinates (blocked as B-tree)
– Two priority search trees for 3-sided queries
in each node v on points below v
⇓
$O(N \log N)$ space
• Query:
– Search for top node v with q_1 and q_2 below different children
– Answer 3-sided queries in children of v
⇓
$O(\log_B N + T/B)$ query
q_4
q_3
q_1 q_2
x
Lars Arge
45

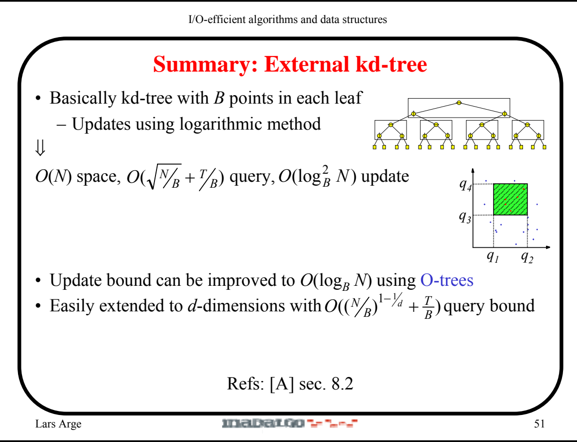

I/O-efficient algorithms and data structures
External Range Tree
• Increased fanout to $\Theta(\log_B N)$
⇒ Space improved to $O(N \log_{\log_B N} N) = O(N \frac{\log_B N}{\log_B \log_B N})$
• Extra external priority search tree in each node
– to find bottom relevant point in
$O(\log_B N)$ slabs spanned by query
⇒ Query answered in $O(\log_B N + T/B)$ I/Os
$\Theta(\log_B N)$
• Dynamic with $O(\frac{\log_B^2 N}{\log_B \log_B N})$ update bound using weight-balanced tree
Refs: [A] sec. 8.1
Lars Arge
46

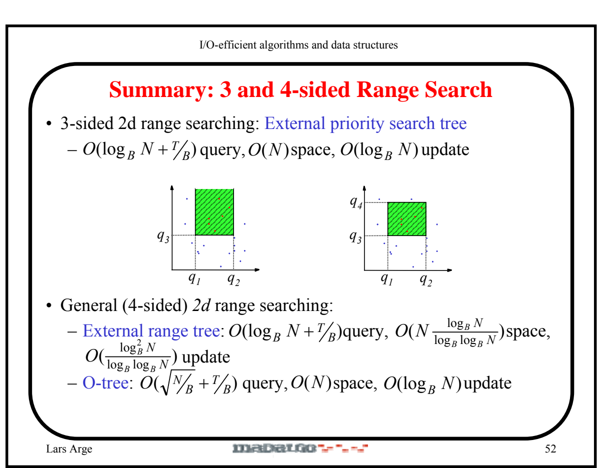

I/O-efficient algorithms and data structures
Outline
1. Introduction
2. Fundamental algorithms
3. Buffered data structures
4. Range searching
– External priority search tree
* Weight-balanced B-tree
* Persistent B-trees
– External Range tree
– External kd-tree
5. List ranking
q_4
q_3
q_1 q_2
D
M
P
Lars Arge
47

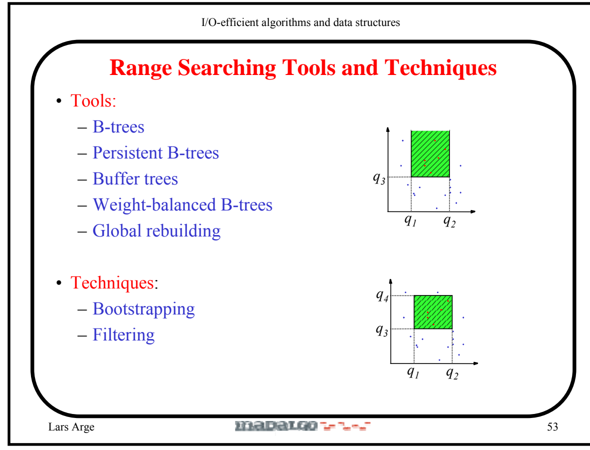

I/O-efficient algorithms and data structures
External kd-tree
• kd-tree:
– Recursive subdivision of point-set into two half using
vertical/horizontal line
– Horizontal line on even levels, vertical on uneven levels
– One point in each leaf
⇓
Linear space and logarithmic height
Lars Arge
48

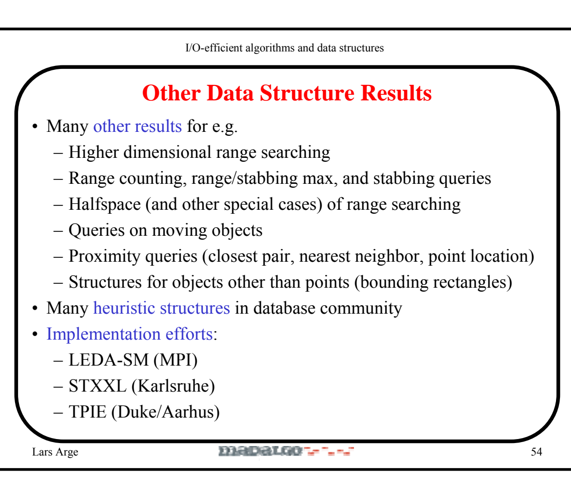

I/O-efficient algorithms and data structures
External kd-Tree
• kd-tree Query
– Recursively visit nodes corresponding to regions intersecting query
– Report point in trees/nodes completely contained in query
• kd-tree Query analysis
– Horizontal line intersect $Q(N) = 2+2Q(N/4) = O(\sqrt{N})$ regions
– Query covers T regions
$\Rightarrow O(\sqrt{N}+T)$ I/Os worst-case
Lars Arge
49

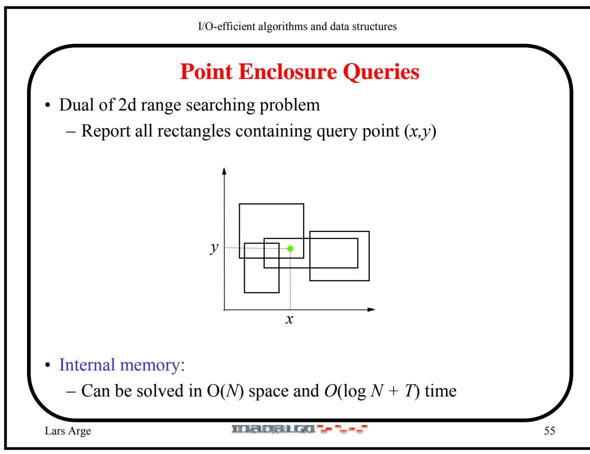

I/O-efficient algorithms and data structures
External kd-tree
• External kd-tree:
– Blocking of kd-tree but with B point in each leaf
• Query as before
– Analysis as before except that each region now contains B points
$\Rightarrow O(\sqrt{N/B}+T/B)$ I/O query
• Dynamic:
– Deletes relatively easily in $O(\log_B^2 N)$ I/Os using global rebuilding
– Insertions also in $O(\log_B^2 N)$ I/Os using logarithmic method
Lars Arge
50

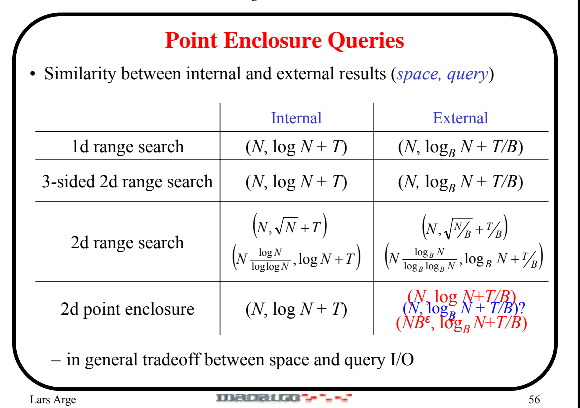

I/O-efficient algorithms and data structures
Summary: External kd-tree
• Basically kd-tree with B points in each leaf
– Updates using logarithmic method
$\Downarrow$
$O(N)$ space, $O(\sqrt{N/B}+T/B)$ query, $O(\log_B^2 N)$ update
q_4
q_3
q_1 q_2
• Update bound can be improved to $O(\log_B N)$ using O-trees
• Easily extended to d-dimensions with $O((N/B)^{1-1/d}+\frac{T}{B})$ query bound
Refs: [A] sec. 8.2
Lars Arge
51

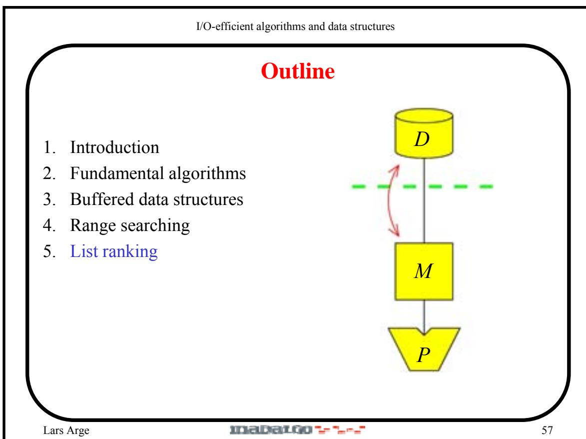

I/O-efficient algorithms and data structures
Summary: 3 and 4-sided Range Search
• 3-sided 2d range searching: External priority search tree
– $O(\log_B N+T/B)$ query, $O(N)$ space, $O(\log_B N)$ update
q_3
q_1 q_2
q_4
q_3
q_1 q_2
• General (4-sided) 2d range searching:
– External range tree: $O(\log_B N+T/B)$ query, $O(N\frac{\log_B N}{\log_B \log_B N})$ space, $O(\frac{\log_B^2 N}{\log_B \log_B N})$ update
– O-tree: $O(\sqrt{N/B}+T/B)$ query, $O(N)$ space, $O(\log_B N)$ update
Lars Arge
52

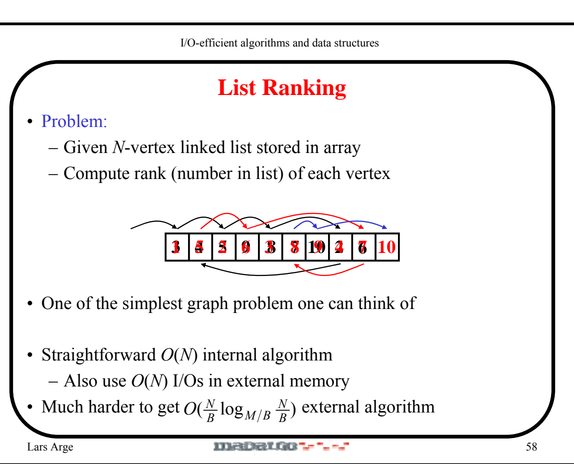

I/O-efficient algorithms and data structures
Range Searching Tools and Techniques
• Tools:
– B-trees
– Persistent B-trees
– Buffer trees
– Weight-balanced B-trees
– Global rebuilding
• Techniques:
– Bootstrapping
– Filtering
q_3
q_1 q_2
q_4
q_3
q_1 q_2
Lars Arge
53

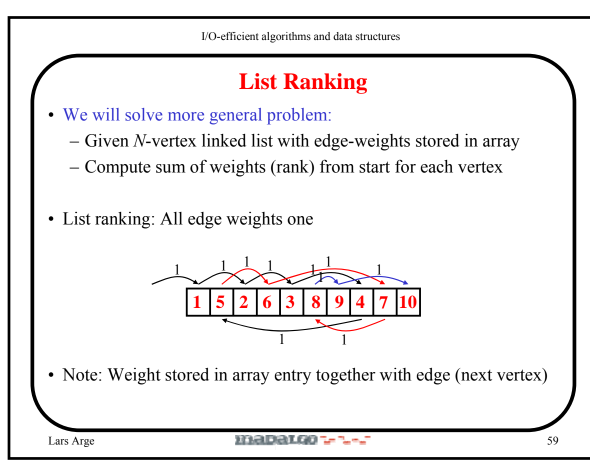

I/O-efficient algorithms and data structures
Other Data Structure Results
• Many other results for e.g.
– Higher dimensional range searching
– Range counting, range/stabbing max, and stabbing queries
– Halfspace (and other special cases) of range searching
– Queries on moving objects
– Proximity queries (closest pair, nearest neighbor, point location)
– Structures for objects other than points (bounding rectangles)
• Many heuristic structures in database community
• Implementation efforts:
– LEDA-SM (MPI)
– STXXL (Karlsruhe)
– TPIE (Duke/Aarhus)
Lars Arge
54

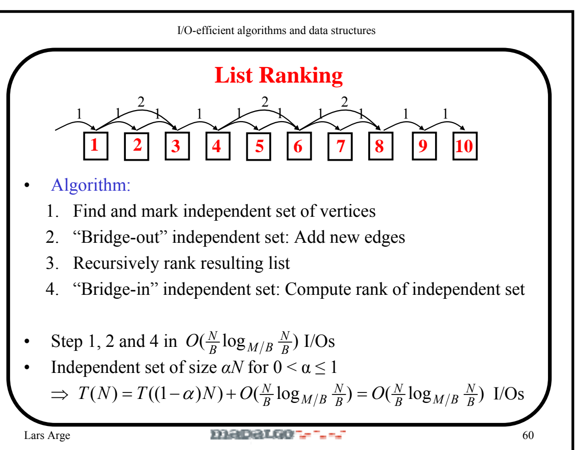

I/O-efficient algorithms and data structures
Point Enclosure Queries
• Dual of 2d range searching problem
– Report all rectangles containing query point (x,y)
y
x
• Internal memory:
– Can be solved in O(N) space and O(log N + T) time
Lars Arge
55


I/O-efficient algorithms and data structures
Point Enclosure Queries
• Similarity between internal and external results (space, query)
Internal
External
1d range search
$(N, \log N + T)$
$(N, \log_B N + T/B)$
3-sided 2d range search
$(N, \log N + T)$
$(N, \log_B N + T/B)$
2d range search
$(N, \sqrt{N} + T)$
$(N \frac{\log N}{\log\log N}, \log N + T)$
$(N, \sqrt{N/B} + T/B)$
$(N \frac{\log_B N}{\log_B \log_B N}, \log_B N + T/B)$
2d point enclosure
$(N, \log N + T)$
$(N, \log N + T/B)$
$(N, \log_B N + T/B)$?
$(NB^{\varepsilon}, \log_B N + T/B)$
– in general tradeoff between space and query I/O
Lars Arge
56


I/O-efficient algorithms and data structures
Outline
1. Introduction
2. Fundamental algorithms
3. Buffered data structures
4. Range searching
5. List ranking
D
M
P
Lars Arge
57


I/O-efficient algorithms and data structures
List Ranking
• Problem:
– Given N-vertex linked list stored in array
– Compute rank (number in list) of each vertex
• One of the simplest graph problem one can think of
• Straightforward O(N) internal algorithm
– Also use O(N) I/Os in external memory
• Much harder to get $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ external algorithm
Lars Arge
58


I/O-efficient algorithms and data structures
List Ranking
• We will solve more general problem:
– Given N-vertex linked list with edge-weights stored in array
– Compute sum of weights (rank) from start for each vertex
• List ranking: All edge weights one
1 5 2 6 3 8 9 4 7 10
• Note: Weight stored in array entry together with edge (next vertex)
Lars Arge
59


I/O-efficient algorithms and data structures
List Ranking
1 2 3 4 5 6 7 8 9 10
• Algorithm:
1. Find and mark independent set of vertices
2. "Bridge-out" independent set: Add new edges
3. Recursively rank resulting list
4. "Bridge-in" independent set: Compute rank of independent set
• Step 1, 2 and 4 in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os
• Independent set of size αN for 0 < α ≤ 1
$\Rightarrow T(N) = T((1-\alpha)N) + O(\frac{N}{B}\log_{M/B}\frac{N}{B}) = O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os
Lars Arge
60

List Ranking: Bridge-out/in

- Obtain information (edge or rang) of successor
  - Make copy of original list
  - Sort original list by successor id
  - Scan original and copy together to obtain successor information
  - Sort modified original list by id

$\Rightarrow O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os

List Ranking: Independent Set

- Easy to design $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ randomized algorithm:
  - Scan list and flip a coin for each vertex
  - Independent set is vertices with head and successor with tails

$\Rightarrow$ Independent set of expected size *N/4*

- Deterministic algorithm:
  - 3-color vertices (no vertex same color as predecessor/successor)
  - Independent set is vertices with most popular color

$\Rightarrow$ Independent set of size at least *N/3*

- $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ 3-coloring $\Rightarrow O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/O algorithm

List Ranking: 3-coloring

- Algorithm:
  - Consider forward and backward lists (heads/tails in two lists)
  - Color forward lists (except tail) alternately red and blue
  - Color backward lists (except tail) alternately green and blue

$\Downarrow$

3-coloring

List Ranking: Forward List Coloring

- Identify heads and tails
- For each head, insert red element in priority-queue (priority=position)
- Repeatedly:
  - Extract minimal element from queue
  - Access and color corresponding element in list
  - Insert opposite color element corresponding to successor in queue

- Scan of list
- O(N) priority-queue operations

$\Rightarrow O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os

Summary: List Ranking

- Simplest graph problem: Traverse linked list

- Very easy $O(N)$ algorithm in internal memory
- Much more difficult $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ external memory
  - Finding independent set via 3-coloring
  - Bridging vertices in/out
- Permuting bound $O(\min\{N, \frac{N}{B}\log_{M/B}\frac{N}{B}\})$ best possible
  - Also true for other graph problems

Refs: [Z] sec. 2, 4.2

Summary: List Ranking

- External list ranking algorithm similar to PRAM algorithm
  - Sometimes external algorithms by "PRAM algorithm simulation"

- Forward list coloring algorithm example of "time forward processing"
  - Use external priority-queue to send information "forward in time" to vertices to be processed later

Other Graph Algorithm Results

- Most tree problems solved in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os
- Most planar graph problems solved in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os

- Most other problems on general graphs not satisfactory solved
  - Directed DFS/BFS: $O(V+E/B)\log_2 V)$ or $O(V+\frac{E}{B}\frac{V}{M})$
  - Undirected BFS: $O(V+\frac{E}{B}\log_{M/B}\frac{E}{B})$ or $O(\sqrt{VE/B}+\frac{E}{B}\log_{M/B}\frac{E}{B})$
  - MSF: $O(V+\frac{E}{B}\log_{M/B}\frac{E}{B})$ or $O(\log_2\log_2\frac{VB}{E}\cdot\frac{E}{B}\log_{M/B}\frac{E}{B})$
  - SSSP: $O(V+\frac{E}{B}\log_2\frac{E}{B})$

- No other than permutation lower bound $O(\min\{E,\frac{E}{B}\log_{M/B}\frac{E}{B}\})$ known

Exercise

Given a grid terrain model (an $\sqrt{N}\times\sqrt{N}$ height grid)

design an $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/O algorithm for computing flow accumulation grid:

- Initially one unit of water in each grid cell
- Water (initial and received) distributed from each cell to lowest lower neighbor cell (if existing)
- Flow accumulation of cell is total flow through it

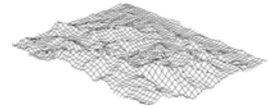
Flow Accumulation

- Problem can easily be solved in $O(N\log N)$ time:

- Process (sweep) points by decreasing height. At each cell:
  - Read flow from flow grid and neighbor heights from height grid
  - Update flow (flow grid) for downslope neighbors

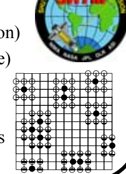
  $\Downarrow$

  One sweep $\Rightarrow$ $O(N\log N)$ time algorithm



Geometric I/O-bottleneck Example

- Computed for Appalachian Mountains (800km x 800km) by Duke University environmental researchers
  - 100m resolution $\Rightarrow$ ~ 64M cells
    - $\Rightarrow$ ~128MB raw data (~500MB processing)
    - $\Rightarrow$ 14 days (on 512MB machine)
- Dataset could me much larger:
  - ~ 1.2GB at 30m resolution (80% of earth covered by NASA SRTM mission)
  - ~ 12GB at 10m resolution (much of US available)
  - ~ 1.2TB at 1m resolution



- Problem: Scattered access to grid cells $\Rightarrow$ $O(N)$ I/Os / Use of implementation of $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ algorithm

  $\Rightarrow$ Appalachian Mountains in 3 hours!

Exercise

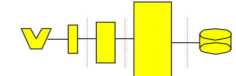
Given a grid terrain model (an $\sqrt{N}\times\sqrt{N}$ height grid)

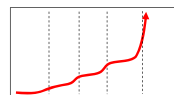
design an $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/O algorithm for computing flow accumulation grid:

Hints:

1. Store all neighbor heights with each cell
2. Distribute water to neighbors using time forward processing

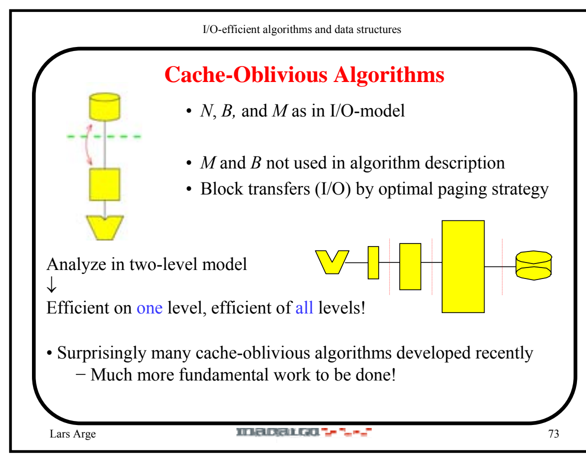
Cache-Oblivious Algorithms

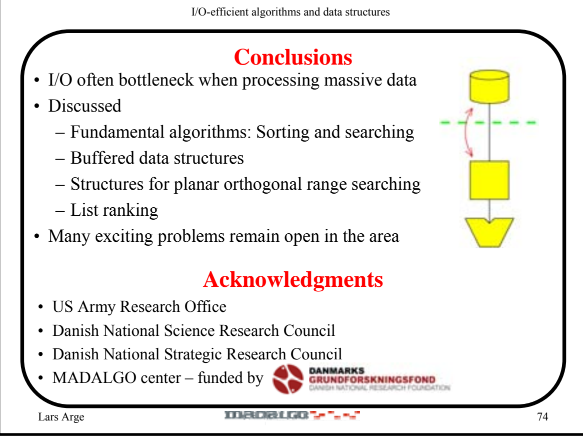


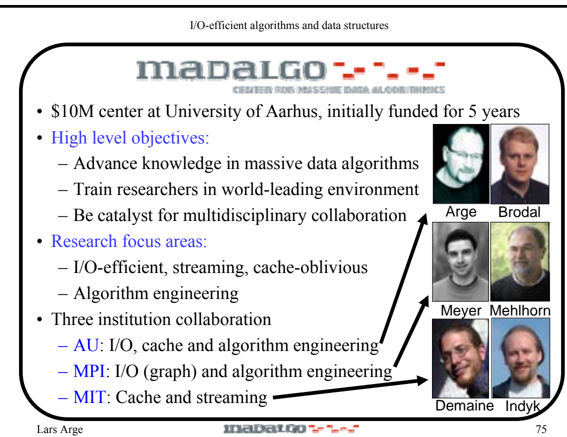


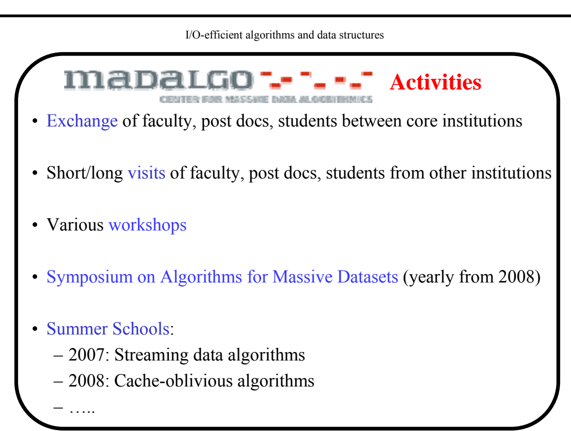
- Block access important on all levels of memory hierarchy
  - But complicated to model whole hierarchy

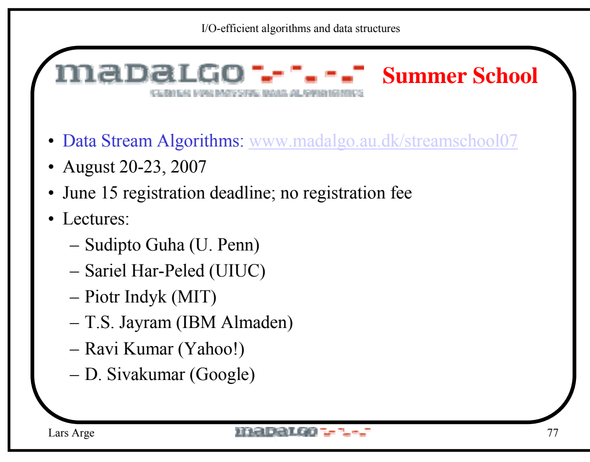
- I/O-model can be used on all levels
  - But dominating level can change during computation
  - Characteristics of hierarchy may not be known

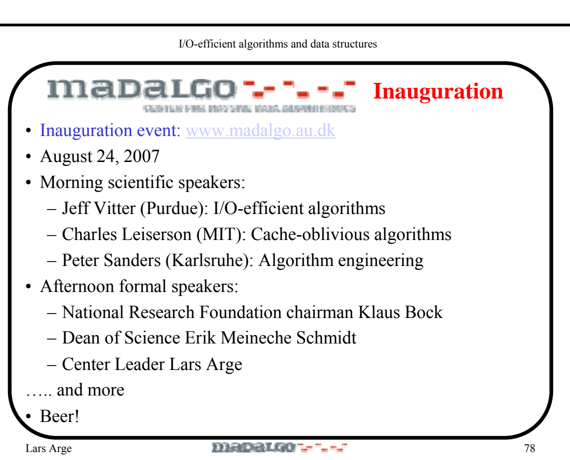

I/O-efficient algorithms and data structures
Cache-Oblivious Algorithms
• N, B, and M as in I/O-model
• M and B not used in algorithm description
• Block transfers (I/O) by optimal paging strategy
Analyze in two-level model
↓
Efficient on one level, efficient of all levels!
• Surprisingly many cache-oblivious algorithms developed recently
– Much more fundamental work to be done!
Lars Arge
73


I/O-efficient algorithms and data structures
Conclusions
• I/O often bottleneck when processing massive data
• Discussed
– Fundamental algorithms: Sorting and searching
– Buffered data structures
– Structures for planar orthogonal range searching
– List ranking
• Many exciting problems remain open in the area
Acknowledgments
• US Army Research Office
• Danish National Science Research Council
• Danish National Strategic Research Council
• MADALGO center – funded by
DANMARKS GRUNDFORSKNINGSFOND
Lars Arge
74


I/O-efficient algorithms and data structures
MADALGO
• $10M center at University of Aarhus, initially funded for 5 years
• High level objectives:
– Advance knowledge in massive data algorithms
– Train researchers in world-leading environment
– Be catalyst for multidisciplinary collaboration
• Research focus areas:
– I/O-efficient, streaming, cache-oblivious
– Algorithm engineering
• Three institution collaboration
– AU: I/O, cache and algorithm engineering
– MPI: I/O (graph) and algorithm engineering
– MIT: Cache and streaming
Arge
Brodal
Meyer
Mehlhorn
Demaine
Indyk
Lars Arge
75


I/O-efficient algorithms and data structures
MADALGO
Activities
• Exchange of faculty, post docs, students between core institutions
• Short/long visits of faculty, post docs, students from other institutions
• Various workshops
• Symposium on Algorithms for Massive Datasets (yearly from 2008)
• Summer Schools:
– 2007: Streaming data algorithms
– 2008: Cache-oblivious algorithms
– …..
Lars Arge
76


I/O-efficient algorithms and data structures
MADALGO
Summer School
• Data Stream Algorithms: www.madalgo.au.dk/streamschool07
• August 20-23, 2007
• June 15 registration deadline; no registration fee
• Lectures:
– Sudipto Guha (U. Penn)
– Sariel Har-Peled (UIUC)
– Piotr Indyk (MIT)
– T.S. Jayram (IBM Almaden)
– Ravi Kumar (Yahoo!)
– D. Sivakumar (Google)
Lars Arge
77


I/O-efficient algorithms and data structures
MADALGO
Inauguration
• Inauguration event: www.madalgo.au.dk
• August 24, 2007
• Morning scientific speakers:
– Jeff Vitter (Purdue): I/O-efficient algorithms
– Charles Leiserson (MIT): Cache-oblivious algorithms
– Peter Sanders (Karlsruhe): Algorithm engineering
• Afternoon formal speakers:
– National Research Foundation chairman Klaus Bock
– Dean of Science Erik Meineche Schmidt
– Center Leader Lars Arge
….. and more
• Beer!
Lars Arge
78