Online Data Structures in External Memory

Jeff Vitter

Duke University *&*

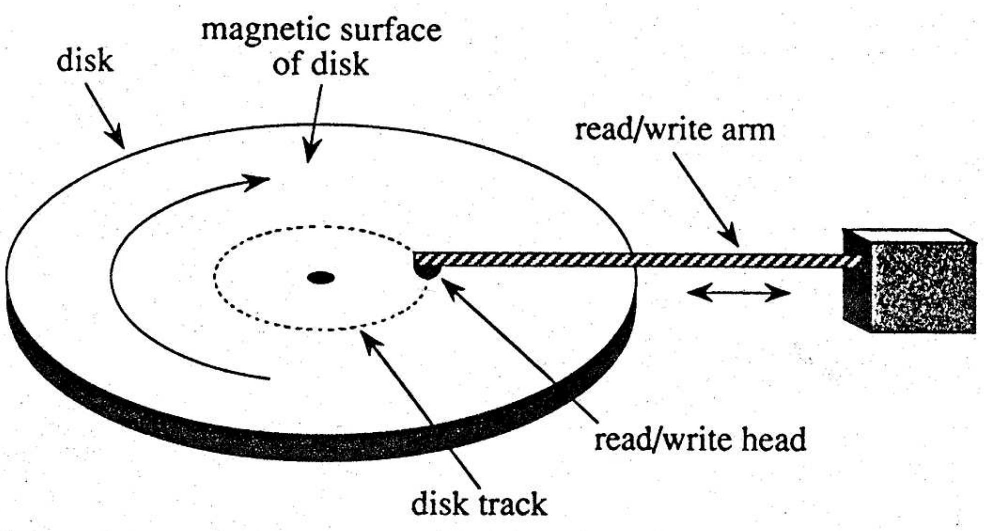
I.N.R.I.A. Sophia Antipolis

Center for Geometric Computing

http://www.cs.duke.edu/CGC/

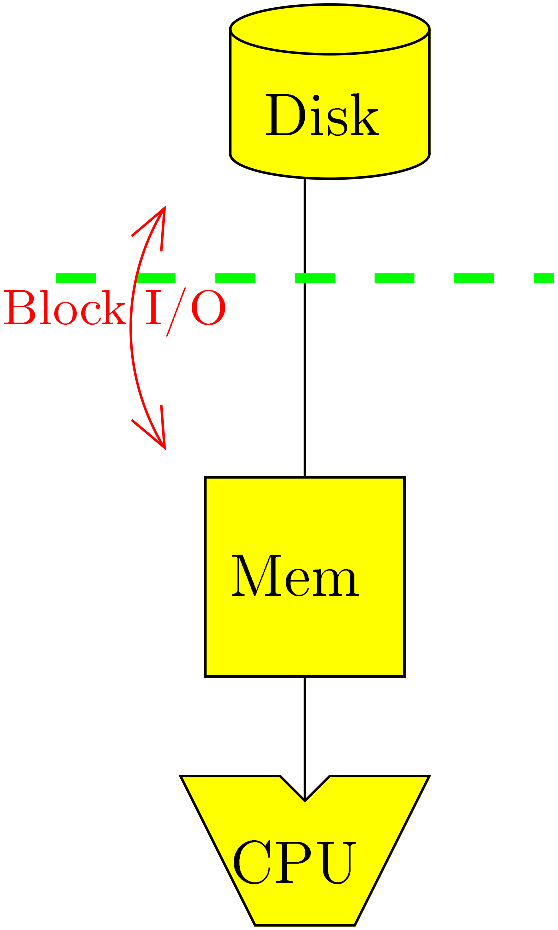
I/O Crisis!

Doug Comer: *The difference in speed between modern CPU and disk technologies is analogous to the difference in speed in sharpening a pencil by 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.*



- ☆ Time for rotation ≈ Time for seek.
- ☆ Amortize search time by large block transfer (eg., 10 KB) so that Time for rotation ≈ Time for seek ≈ Time to transfer data.
- ☆ Parallel disks.

I/O Model [Floyd73], [Knuth98], [AV87]



Parameters (in units of *data items*):

$$N = \text{problem data size.}$$

$$M = \text{size of internal memory.}$$

$$B = \text{size of disk block.}$$

$$Z = \text{query output size.}$$

Parameters (in units of *blocks*):

$$n = \frac{N}{B}, \qquad m = \frac{M}{B}, \qquad z = \frac{Z}{B}.$$

Performance measures:

I/O steps, disk space, and CPU time.

Fundamental I/O Bounds

Batched Problems:

☆ Scanning (or streaming or touching): $\Theta\left(\frac{N}{B}\right) = \Theta(n)$

☆ Sorting: $\Theta\left(\frac{N}{B}\frac{\log \frac{N}{B}}{\log \frac{M}{B}}\right) = \Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) = \Theta(n \log_m n)$

Online Problems:

☆ Search: $\Theta(\log_B N)$

☆ Query Output Reporting: $\Theta\left(\frac{Z}{B}\right) = \Theta(z)$

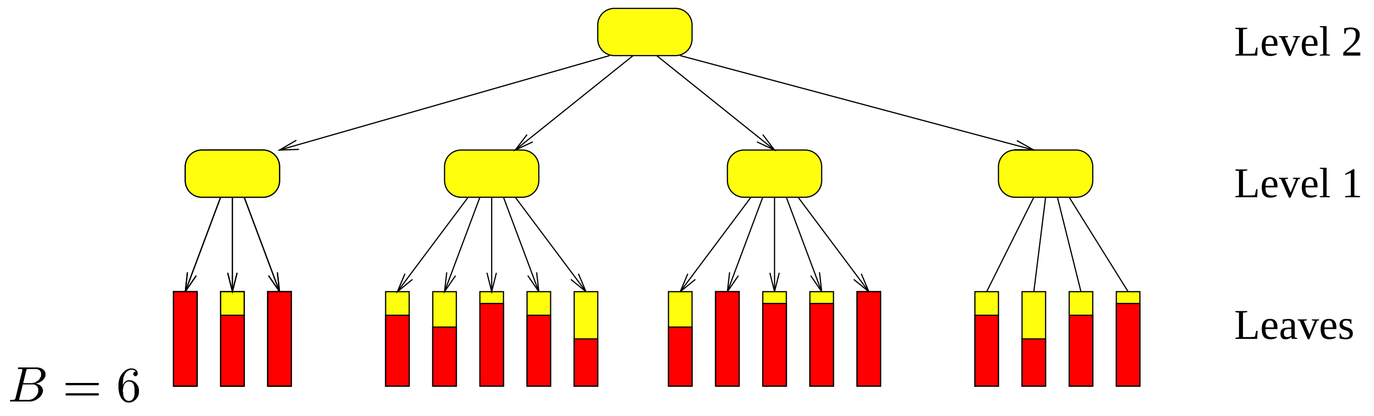
Naive use of an internal memory algorithm (using virtual memory):

$\Theta(N \log n)$ I/Os	*We want* $\Theta(n \log_m n)$ *I/Os !!!*
$\Theta(\log N + Z)$ I/Os	*We want* $\Theta(\log_B N + z)$ *I/Os !!!*

Outline of the Talk

1. Basic data structures
   - ⋆ 1-D range searching: B-trees
   - ⋆ Batched dynamic: Buffer trees
   - ⋆ Multidimensional: R-trees
   - ⋆ Experiments on bulk-loading R-trees
   - ⋆ Weight-balanced B-trees for rebuilding.
   - ⋆ Level-balanced B-trees for leaf-to-root traversals.
2. Other Online Problems
   - ⋆ Proximity queries, nearest neighbor, clustering
   - ⋆ Point location
   - ⋆ Ray shooting
   - ⋆ Range searching
     - Stabbing queries: External interval trees
     - Orthogonal range queries in 2-D: External priority search tree
     - 3-D queries
     - Halfspace range searching
3. Open problems

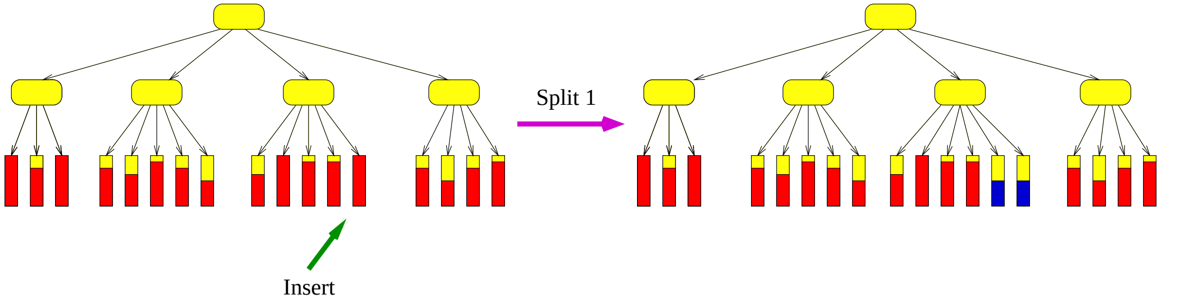
B-Trees [Bayer-McCreight72]



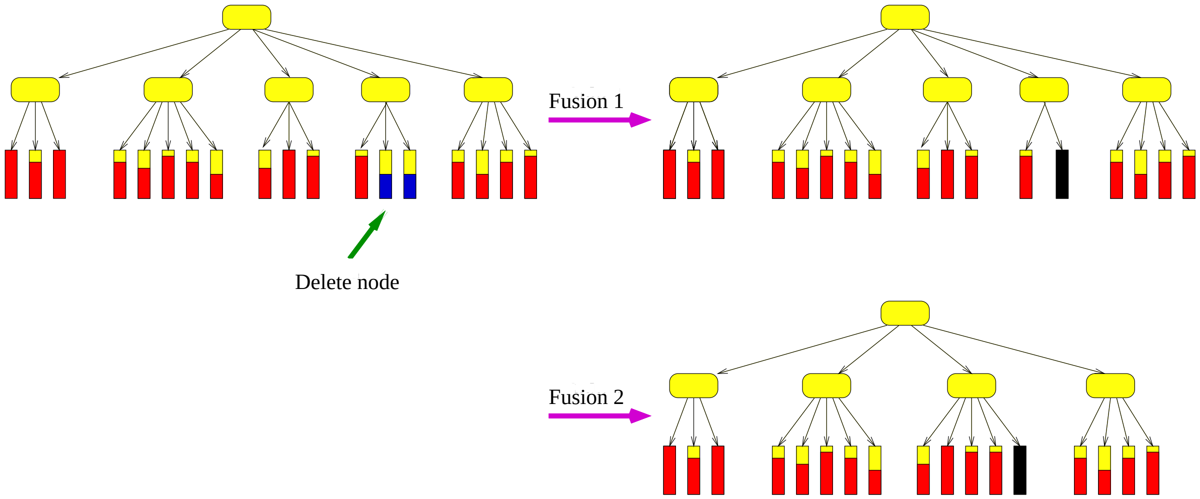
- ☆ Each node is stored in one disk block.
- ☆ Leaf nodes store all the data items.
- ☆ Internal nodes store keys and pointers to guide the searching.
- ☆ $\frac{B}{2} \leq$ node degree $\leq B - 1$, except for the root.
- ☆ One I/O per level
  $\Longrightarrow \quad O(\log_B N + z)$ I/Os for a 1-D range query.

B-Trees—Updates in $O(\log_B N)$ I/Os

Insertion may cause splits:



Deletions may cause nodes to combine:



Sharing also possible (good packing).

Batched Dynamic Processing

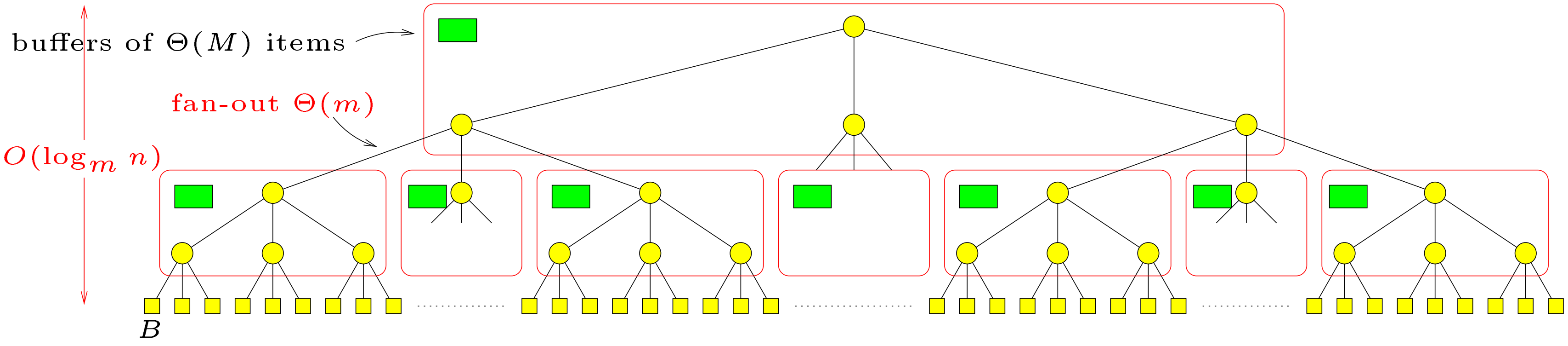
Motivation: Sometimes we may want to handle inserts, deletes, and queries in batches.

Example: Sweep line algorithm for finding rectangle intersections.

- ⋆ Move a horizontal sweep line from top to bottom.
- ⋆ Store rectangles hit by sweep line in a B-tree.
- ⋆ Query and update the B-tree when each new rectangle is hit $\Longrightarrow$ $O(N \log_B N)$ I/Os. *Bad !!!*
- ⋆ We want $O(n \log_m n)$ I/Os. *Good !!!*

Another example (coming soon): Bulk loading of R-trees.

The Buffer Tree [Arge95]



Main idea: Combine degree-B nodes to form supernodes of degree $m = M/B$. Each supernode gets a buffer of size M.

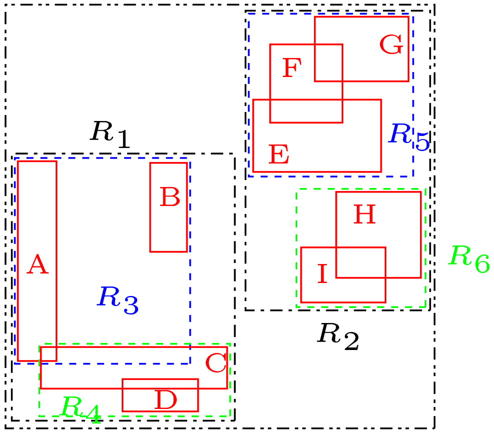
- ☆ Insertions are done "lazily": Items are inserted into buffers. When a buffer runs full, its items are pushed one level down.

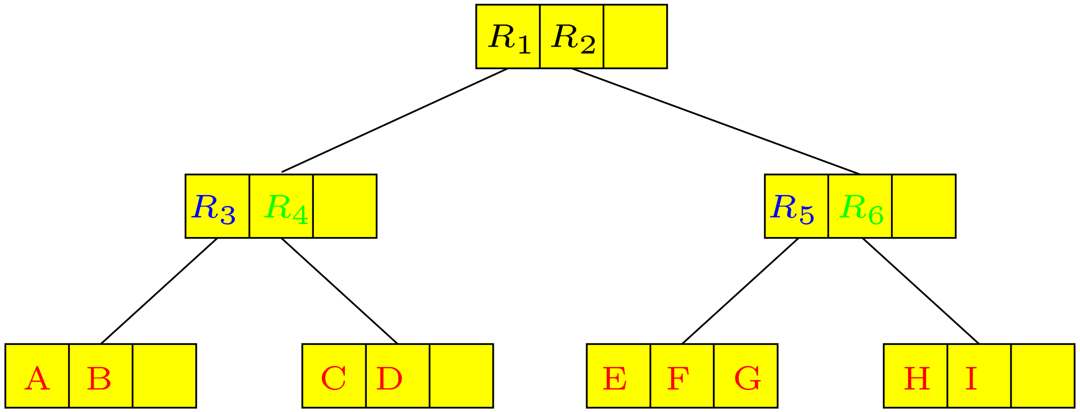
- ☆ Buffer-emptying of M items takes $O(m)$ I/Os

  $\Longrightarrow \quad O(m/M) = O(1/B)$ I/Os per item per level

  $\Longrightarrow \quad N$ items inserted in $O\left(\frac{N}{B}\log_m n\right) = O(n \log_m n)$ I/Os.

R-trees





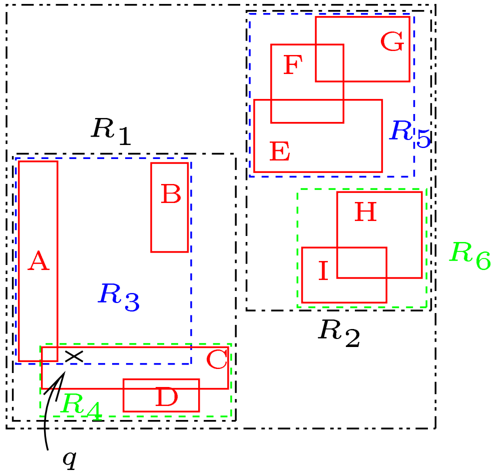
⋆ Multiway structure for storing d-dimensional objects (points, segments, rectangles, polyhedra, etc.).

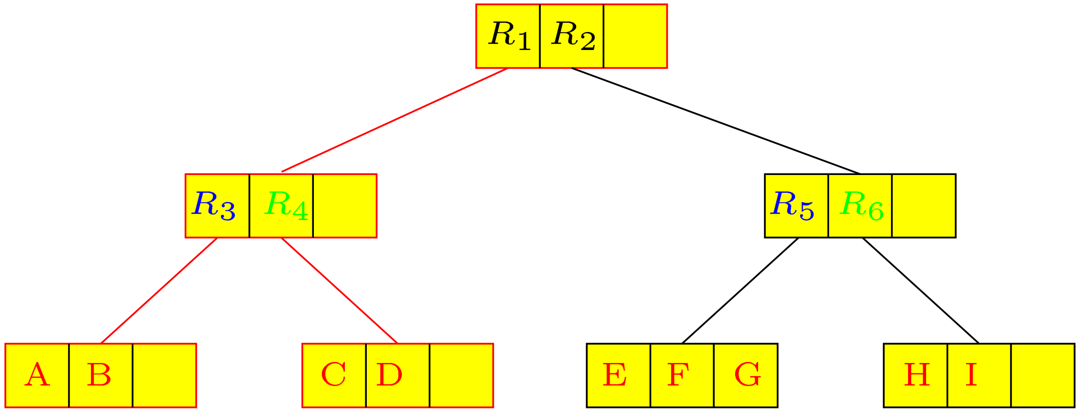
⋆ Structured like B-tree:

- Data in leaves.
- Fan-out B.
- Rebalancing basically like in B-trees.

⋆ Each internal node stores the minimal bounding rectangle of its children.

Querying R-trees





- ☆ Query with point q: Visit all nodes whose minimal bounding rectangle contains q.
- ☆ Minimal bounding rectangles allowed to overlap:
  - Small overlap or perimeter desirable.
  - Several insert/split heuristics (R^+-trees, R*-trees, Hilbert trees, ...) have been proposed, surveyed in [G89, GG98].
- ☆ *Help !* Worst case bad. Typical-case analysis needed.

Constructing ("Bulk Loading") an R-tree

☆ Using repeated insertion takes $O(N \log_B n)$ I/Os.

☆ Other bottom-up algorithms [RL85, KF93, DKLPY94, LLE96, vdBSW97]:

- Rectangles are sorted (using space-filling curve)

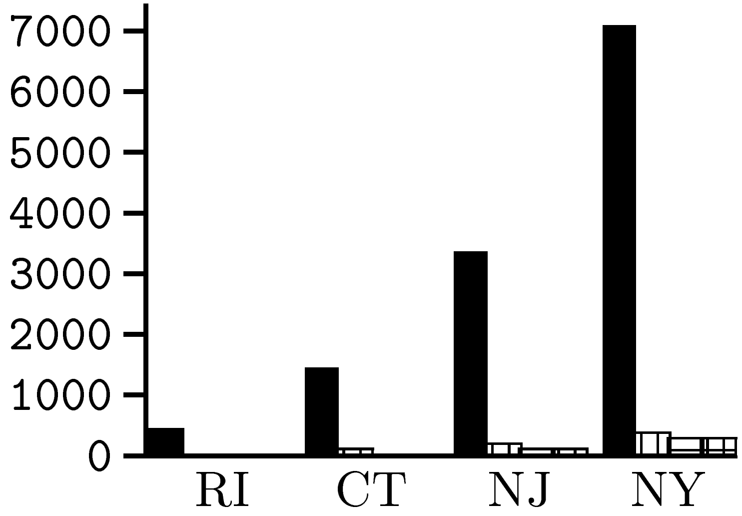
  $\Longrightarrow \quad O(n \log_m n)$-I/O algorithm.
- Can only handle construction—not e.g. "bulk updates."
- Questionable query performance, esp. in high dimensions.

☆ Buffer technique immediately applies to repeated-insertion:

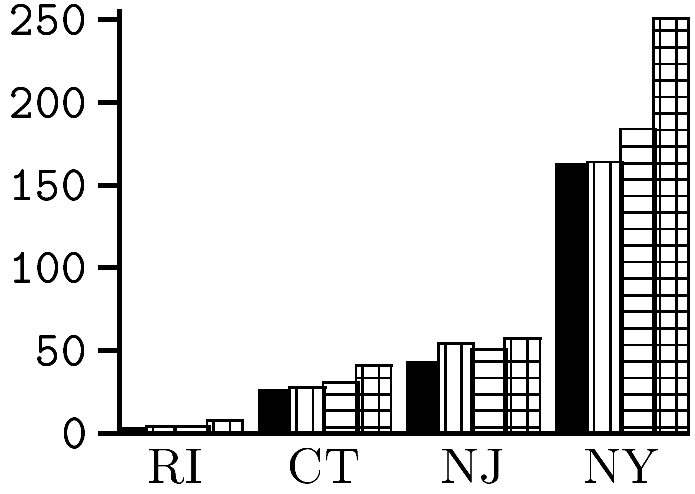
- Conceptually simple.
- Modular design (all R-tree insert heuristics can be used).
- Good query performance.
- Handles all "bulk" operations.

Experimental Results: R-tree

⋆ Implemented using TPIE (see http://www.cs.duke.edu/TPIE/)



Building R-tree on road data
(I/Os in thousands)



Query with (1/10) hydro data
(I/Os in thousands)

⋆ Naive repeated insertion:

⋆ Buffer: Size of buffer $\frac{B^2}{4}$ $\frac{B^2}{2}$ $2B^2$

Weight-balanced B-tree

Problem: If each B-tree node has a substructure (like interval trees, segment trees), a node split can require complete rebuilding.

Solution: *weight-balanced B-trees* [AV96]

- ☆ Weight of a node is the # items in its subtree.
- ☆ All leaves have weight $\Theta(B)$.
  Nodes on level ℓ have weight $\Theta(B^{\ell+1})$.
- ☆ Once a node on level ℓ splits, it won't split again until $\Theta(B^{\ell+1}) = \Theta(\text{weight of node})$ accesses touch the node.
  $\Longrightarrow$ Can do $O(\text{weight of node})$ I/Os to rebuild node.
  $\Longrightarrow$ Amortized cost of accessing node is $O(1)$ I/Os.
  $\Longrightarrow$ Total cost for update is $O(\log_B N)$ I/Os amortized.
- ☆ Can be made worst-case efficient.

Applications of Weight-balanced B-trees

- ⋆ External memory interval tree (*coming soon!*).
- ⋆ External memory segment tree.
- ⋆ Used in many efficient online data structures (point location, reachability queries, range searching, contour-line extraction).
- ⋆ Automatically converting binary and quad search structures (k-d trees, quad trees) to multiway search structures.
- ⋆ In internal memory, it can be used as alternative to BB[α] tree.
  - Interval trees and segment trees without fixed universe.
  - Simpler alternative.
  - Adds range restriction capabilities to data structures.

B-Trees + Parent Pointers

Motivation: Useful in applications where the left-to-right traversal of the tree defines the order of the leaves, and queries are of the form: Is leaf x to the left of leaf y?

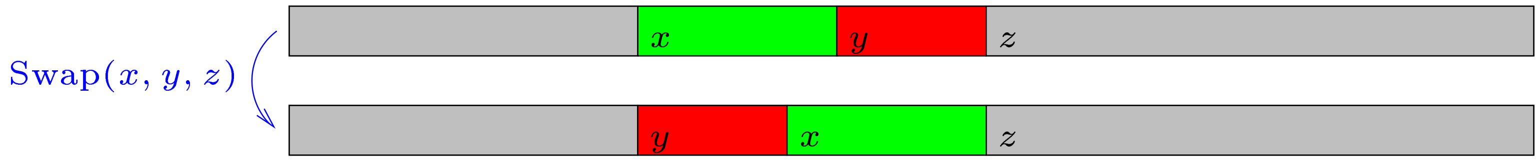
- ⋆ Point location in monotone subdivisions
- ⋆ Reachability queries in planar s, t graphs

Algorithm: Answer query by traversing the two leaf-to-root paths starting at x and y until the nearest common ancestor of x and y is found.

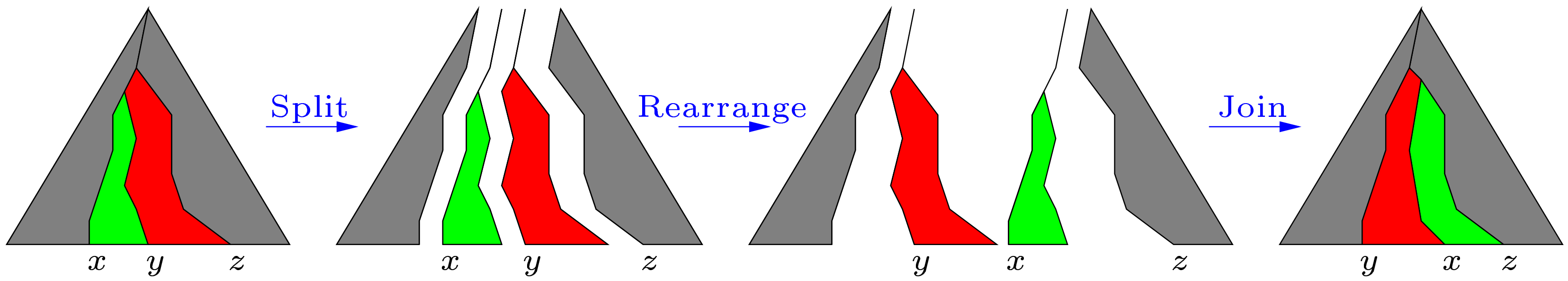
Problem: Updates to the B-tree (especially cuts and joins) require $\Omega(B \log_B N)$ I/Os each to rearrange parent pointers.

Example: Swapping by Cut and Join

Swap(x, y, z) swaps two consecutive subsequences in a list



If the list is represented by a B-tree, the Swap operation can be implemented by 3 cut operations on the tree and 3 join operations:



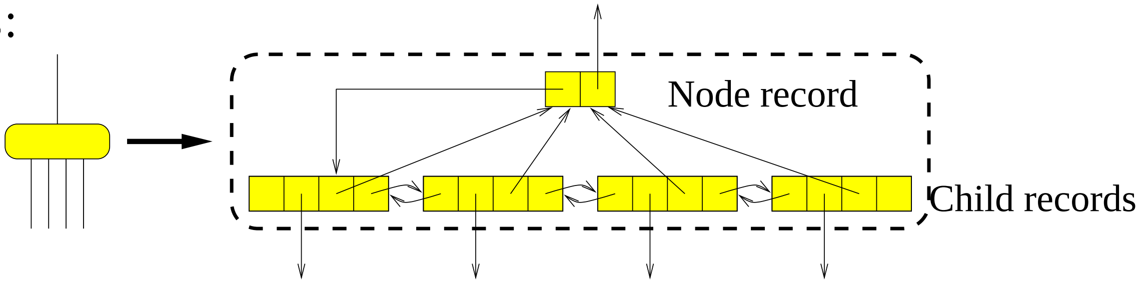
A Swap requires $\Omega(1)$ node splittings/fusions/sharings per level

Without parent pointers $\Longrightarrow$ Swap requires $O(\log_B N)$ I/Os

With parent pointers $\Longrightarrow$ Swap requires $O(B \log_B N)$ I/Os

Level-Balanced B-Trees

- ☆ Each node has degree at least 1 and at most b, for some $2b \leq B$.
- ☆ Replace local balancing condition by a global balancing: Level ℓ contains $\leq 2N_\ell$ non-root nodes, where $N_h = \ell/(\frac{b}{3})^\ell$.
- ☆ A node is stored as a node record and doubly-linked list of child records:



- ☆ Each disk block stores only node records or only child records.
- ☆ All the child records of a node are stored in the same block.
- ☆ When splitting a node, keep the child records in the same block, and add a new node record to the node record's block.
- ☆ Rebalance a level (and its ancestors) when it gets too big, by sorting $\implies$ $O\big((\log_m n)\log_b N\big)$ I/Os amortized per op.

Other Online Problems

Beyond B-trees:

- ☆ Proximity queries, nearest neighbor, clustering
- ☆ Point location, ray shooting
- ☆ Range searching in 2-D, 3-D, orthogonal, halfspace

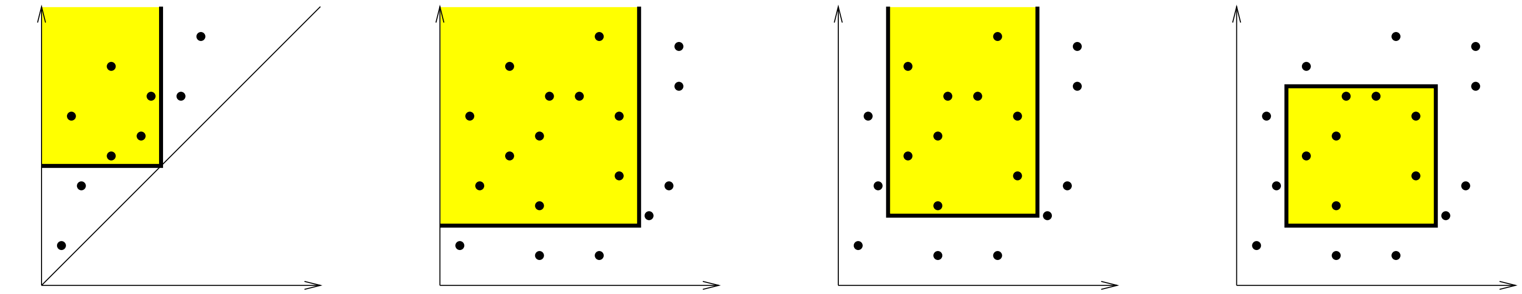
We want data structures that use $O(\log_B N + z)$ I/Os per query, $O(\log_B N)$ I/Os per update, and $O(n)$ space.

Two useful paradigms:

1. Filtering [Chazelle86]: You can afford to do an extra I/O during search if you can charge it to $\Theta(B)$ query outputs.
2. Bootstrapping [AV96, ASV99] Use an external version of an efficient internal-memory data structure as the global search structure. Use efficient static data structures on $O(B^2)$ points to guide the search in the global structure.
   - ☆ Static structure rebuilt in $O(B)$ I/Os after $O(B)$ updates.
   - ☆ Amortized update time $O(1)$ I/Os (also worst-case!).

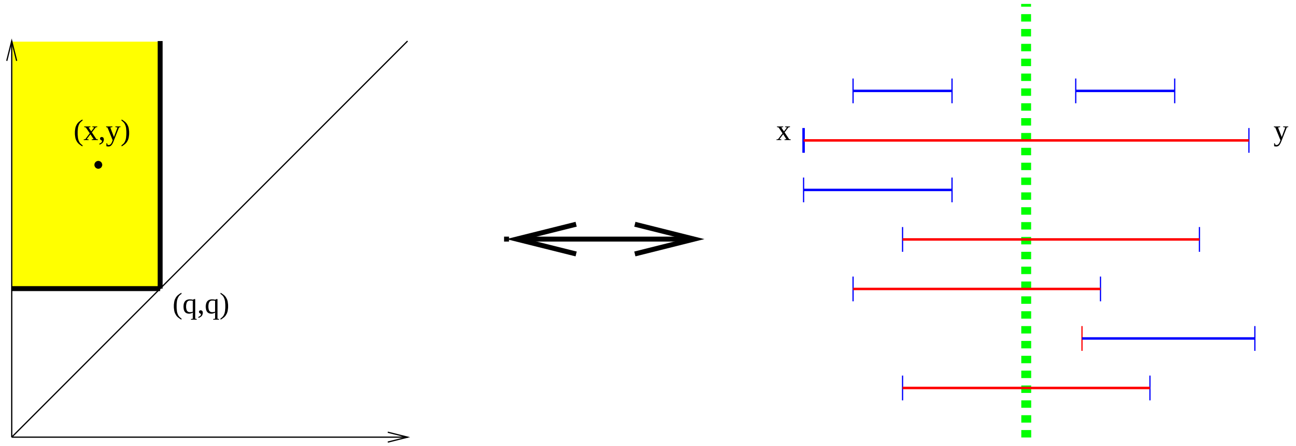
Online Range Searching

Applications: geographic information systems (GIS), spatial databases, indexing for new data models (constraint, temporal, object-oriented).



diagonal corner query　　2-sided query　　3-sided query　　4-sided query

For example, indexing constraints in constraint query languages:



diagonal corner query　　stabbing query with q

External Range Searching Results

- ☆ *Corner (Interval tree):* $O(n)$ space, $O(\log_B N + z)$ I/Os query, $O(\log_B N)$ I/Os update [AV96]. *Optimal !*

- ☆ *2-sided and 3-sided:* $O(n)$ space, $O(\log_B N + z)$ I/Os query, $O(\log_B N)$ I/Os update [ASV99]. *Optimal !*

- ☆ *4-sided:* $O\big(n(\log n)/\log\log_B N\big)$ space, $O(\log_B N + z)$ I/Os query, $O\big((\log_B N)(\log n)/\log\log_B N\big)$ I/Os update [ASV99]. *Optimal except update!*

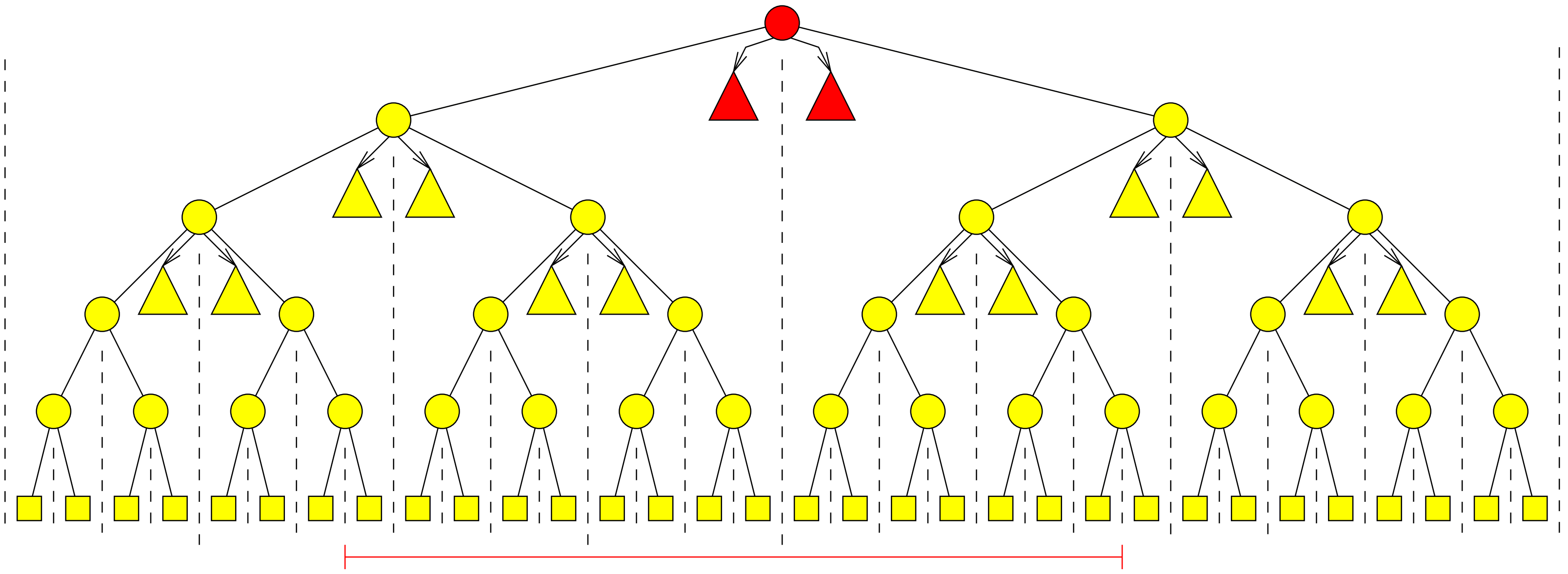
- ☆ Lower bounds: Cannot simultaneously achieve $O\big(n(\log n)/\log\log_B N\big)$ space & $O\big((\log_B n)^c + z\big)$ I/Os query [SR95].

- ☆ Three-dimensional range queries: $O\big(n(\log n)^k/(\log(\log_B N))^k\big)$ space for $(3+k)$-sided 3-D queries, $O(\log_B N + z)$ I/Os query [VV96, Vit99]. *Static: optimal for $k=1$ (octant) and $k=2$!*

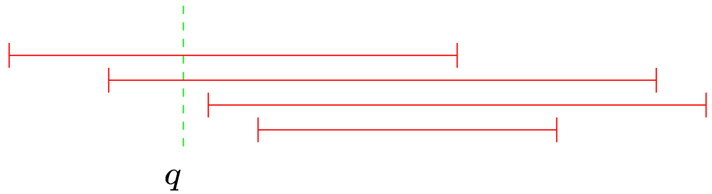
- ☆ Halfspace queries: $O(n)$ space, $O(\log_B n + z)$ I/Os query [AAEFV99]. *Static: optimal !*

- ☆ Open problems: Higher dimensions, typical-case analysis, restricted sets of queries, non-orthogonal searching, point location, etc.

Internal Interval Tree (fixed endpoint set)

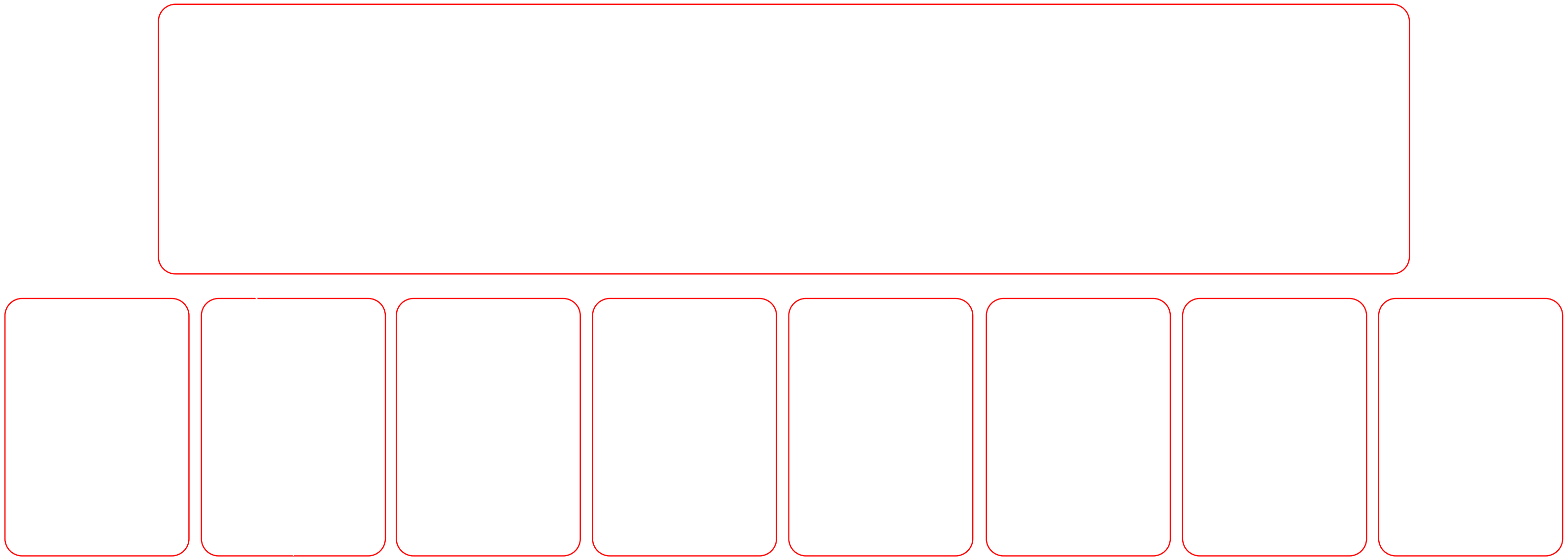


- ☆ An interval X_v is associated with each node v.
- ☆ A segment is stored in the highest node v where it contains the midpoint of X_v.
- ☆ Segments in node v are stored in two secondary search trees, sorted according to left and right endpoint, respectively $\Longrightarrow$ $O(N)$ *space* and $O(\log_2 N)$-*time update* bounds.



☆ Query with q on left side of midpoint of X_{root}:

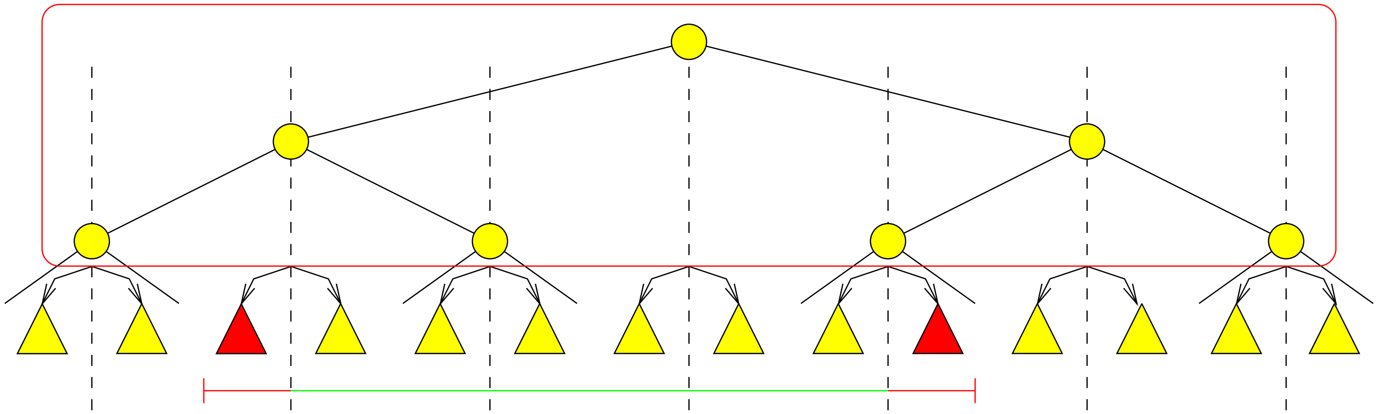
- Search from left-to-right in secondary search tree sorted according to left endpoint.
- Recurse in left son $\Longrightarrow$ $O(\log_2 N + Z)$-*time query* bound.



☆ Number of *stabbed* segments in a secondary structure may be small (or zero).

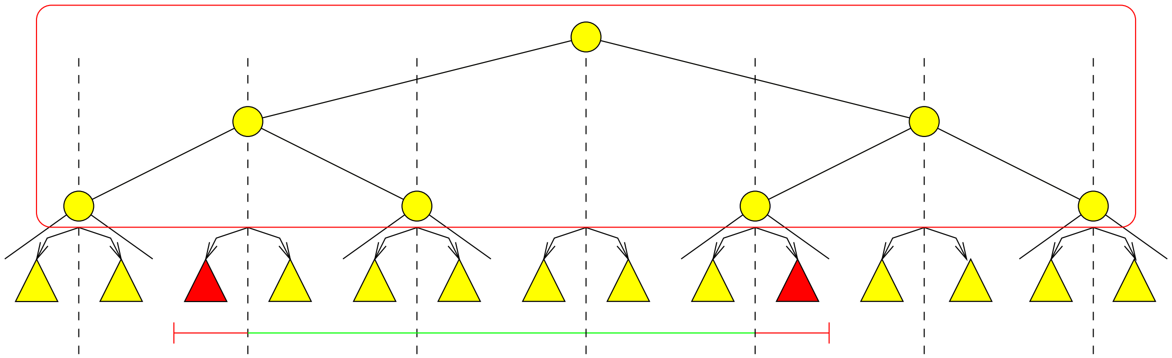
- We can be forced to use an I/O in each of $O(\log_2 N)$ nodes without reporting anything.

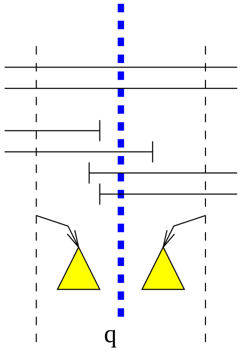
External Interval Tree



- ⋆ Decrease fan-out to $\sqrt{B}$. Height remains $O(\log_B N)$.
- ⋆ $\sqrt{B}$ “slabs” define $O(\sqrt{B}^2) = O(B)$ “multislabs”.
- ⋆ Secondary list structure (B-tree) for each of the $2 \cdot \sqrt{B}$ slabs and $O(B)$ multislabs.
- ⋆ Interval stored in two slab lists and (maybe) one multislab list $\Longrightarrow$ *$O(n)$ space and $O(\log_B N)$ I/Os per update.*
- ⋆ Need weight-balanced B-trees to remove fixed-endpoint assumption. Rebalancing node v requires $O\big(\mathit{weight}(v)\big)$ I/Os.

Query in External Interval Tree





Search for q

- ⋆ Search down the structure and in every of the $O(\log_B N)$ nodes:
  - "Search" in two slab lists: $2 \cdot O(z+1)$ I/Os.
  - Report all segments in relevant multislab lists

— $O(z+B)$ *!!!*

— $O(z)$.

—The segments for any multislab containing less than $B/2$ segments are collectively stored in the node's "corner structure" (of total size $O(B^2)$). We avoid looking at those multislabs by instead querying the corner structure — $O(z+1)$.

$\Longrightarrow$ *Optimal query bound of* $O(\log_B N + z)$ *I/Os.*

Corner Structure of $O(B^2)$ points

Corner Structure: A set of $k \leq B^2$ segments can be represented using $O(k/B)$ blocks such that a stabbing query can be answered in $O(z+1)$ I/Os [KRVV93].

- ⋆ The corner structure can be made dynamic with $O(1)$ amortized update bound: Keep a special update block and rebuild the structure after B updates have been performed [KRVV93].
- ⋆ Amortization can be removed by rebuilding the corner structure lazily.

3-sided Range Searches in Two Dimensions

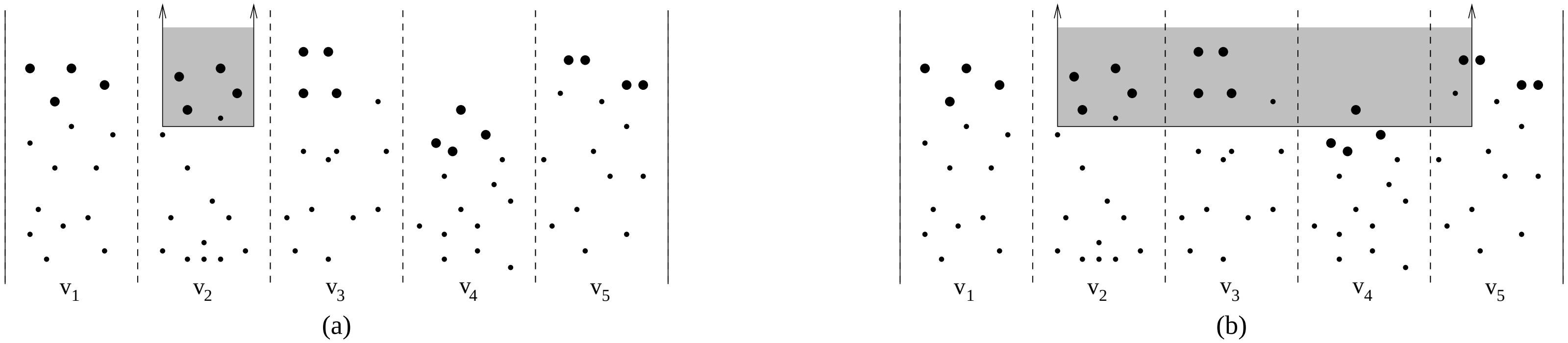
Morale: For stabbing queries, the bootstrapping paradigm worked well: an external version of an internal memory data structure (interval trees), combined with efficient static structures for $O(B^2)$ points.

Main Result for 3-sided Range Search: Data structure with $O(n)$ space, $O(\log_B N + z)$ I/Os per query, $O(\log_B N)$ I/Os per update.

Use bootstrapping paradigm applied to priority search tree:

- ☆ Build weight-balanced B-tree T based upon x-coordinate.
- ☆ Each node v in T corresponds to an x-interval.
- ☆ Each point is stored in child cache of exactly one node.
- ☆ Node v stores the following points in its child cache:
  The Y-set for each of its $\Theta(B)$ children.
- ☆ The Y-set of a child w consists of the highest $\Theta(B)$ points in w's x-range that are not already stored in an ancestor of v.

3-sided Range Query

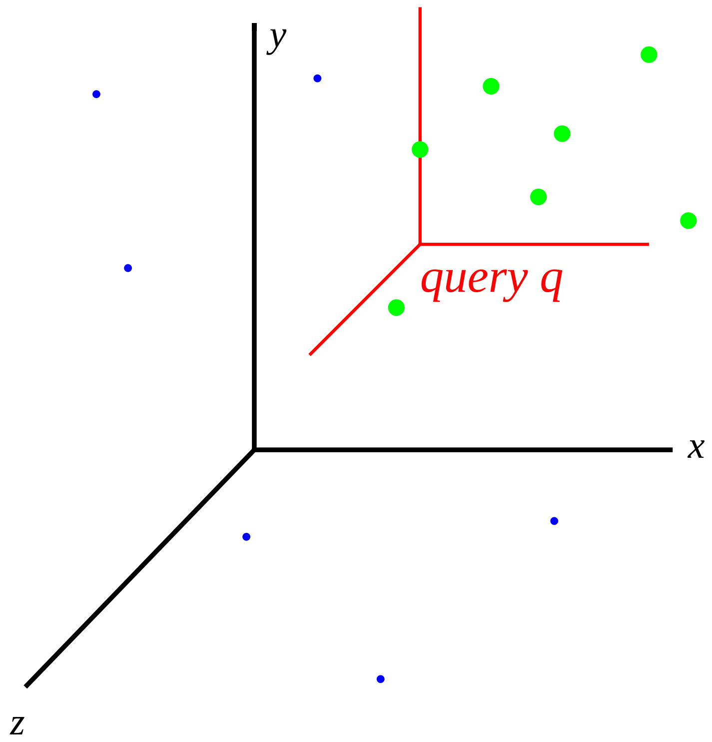


Visit node v:

1. Do 3-sided query on v's child cache and report output.
2. For each child w of v, recursively visit node w iff one of the following conditions hold:
   (a) w is along the left search path for the query or the right search path for the query. —$O(\log_B N)$ such nodes
   (b) the entire Y-set of w was reported while querying v's child cache.
   —I/O cost for visiting w can be charged to the Y-set.

$\Longrightarrow$ $O(\log_B N + z)$ I/Os per query.

Octant Searching (3-sided range queries) in 3-D



Ridge structure:

- ☆ $O(n)$ space, $O(\log_B N)$ I/Os octant query. *Optimal !*
- ☆ Static structure only.
- ☆ Setting $B = 2$: Improves internal memory bound of [CE87].

Conclusions and Open Problems

☆ *Répertoire of useful data structures (B-trees, external interval tree, external priority search tree) for important problems.*

- Worst-case optimality requires overhead.
- Simpler versions are practical!
- Building blocks for external data structures (e.g., interval tree allows brute force rebuilding)

☆ *Lots of open problems in external memory data structures!*

- 4-sided 2-D orthogonal range searches.
- High dimensions, non-orthogonal queries, average-case.
- Typical-case behavior of popular data structures (e.g., R-trees).
- Weaken assumptions on lower bounds.
- Point location, non-monotone subdivisions.
- Nearest neighbors, clustering, ray shooting.
- Hierarchical memory models (L2 cache, L1 cache, . . .).
- . . .