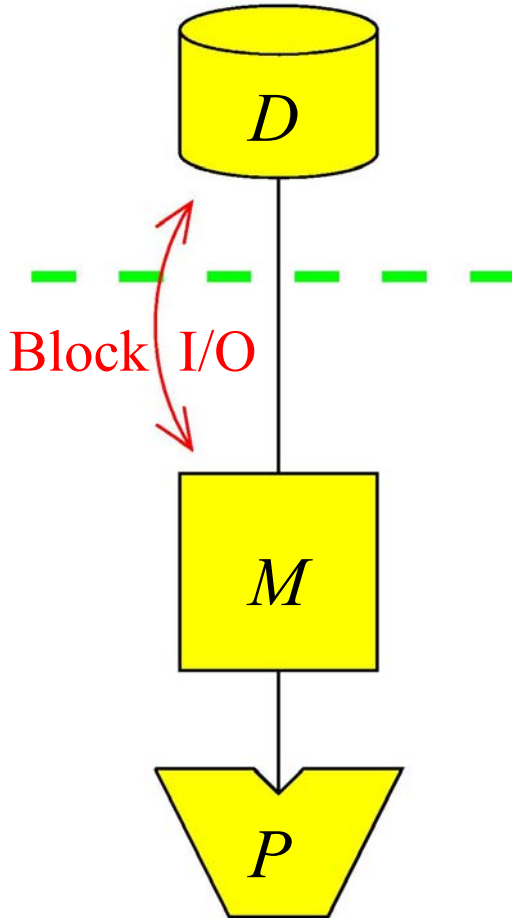


I/O-Algorithms

Lars Arge
Aarhus University

March 5, 2008

I/O-Model



- Parameters

$N = \#$ elements in problem instance

$B = \#$ elements that fits in disk block

$M = \#$ elements that fits in main memory

$T = \#$ output size in searching problem

- We often assume that $M > B^2$

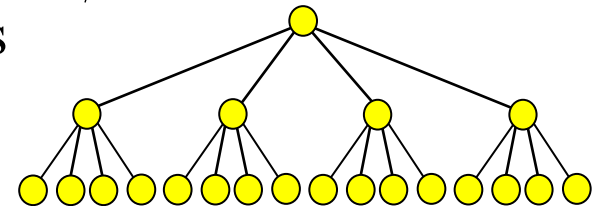
- I/O: Movement of block between memory and disk

Fundamental Bounds

	Internal	External
• Scanning:	N	$\frac{N}{B}$
• Sorting:	$N \log N$	$\frac{N}{B} \log_{M/B} \frac{N}{B}$
• Permuting	N	$\min \left\{ N, \frac{N}{B} \log_{M/B} \frac{N}{B} \right\}$
• Searching:	$\log_2 N$	$\log_B N$

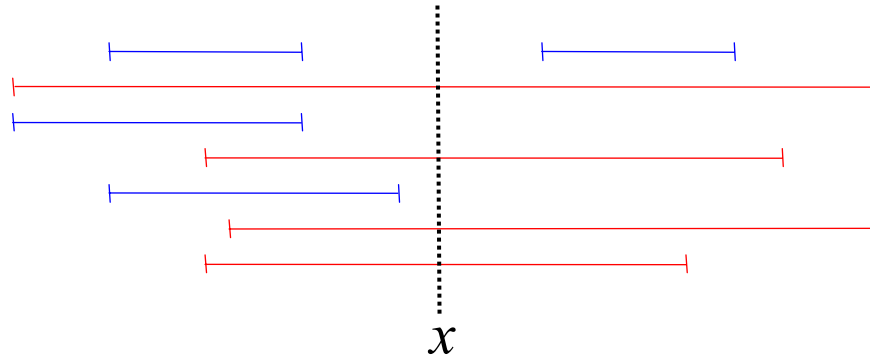
Fundamental Data Structures

- **B-trees**: Node degree $\Theta(B) \Rightarrow$ queries in $O(\log_B N + T/B)$
 - Rebalancing using split/fuse \Rightarrow updates in $O(\log_B N)$
- **Weight-balanced B-trees**: Weight rather than degree constraint
 - $\Rightarrow \Omega(w(v))$ updates below v between rebalancing operations on v
- **Persistent B-trees**:
 - Update in current version in $O(\log_B N)$
 - Search in all previous versions in $O(\log_B N + T/B)$
- **Buffer trees**
 - Batching of operations to obtain $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ bounds
 - $\Rightarrow O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ construction algorithms



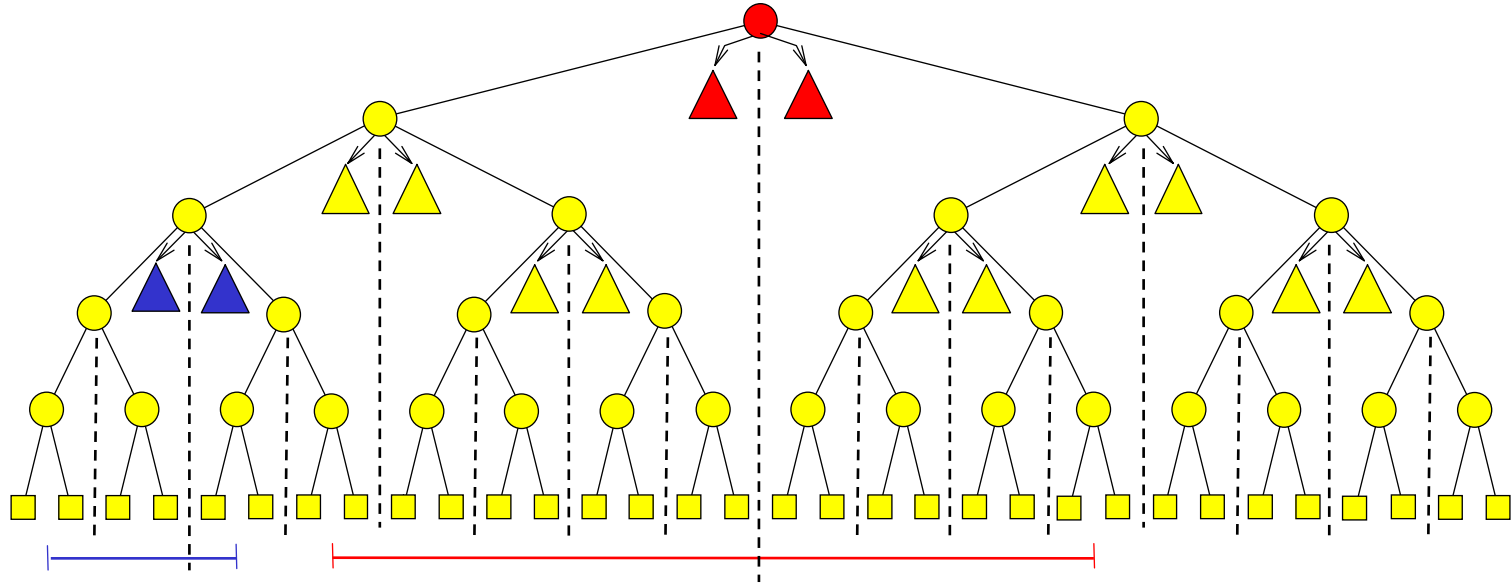
Last time: Interval management

- Maintain N intervals with **unique endpoints** dynamically such that stabbing query with point x can be answered efficiently



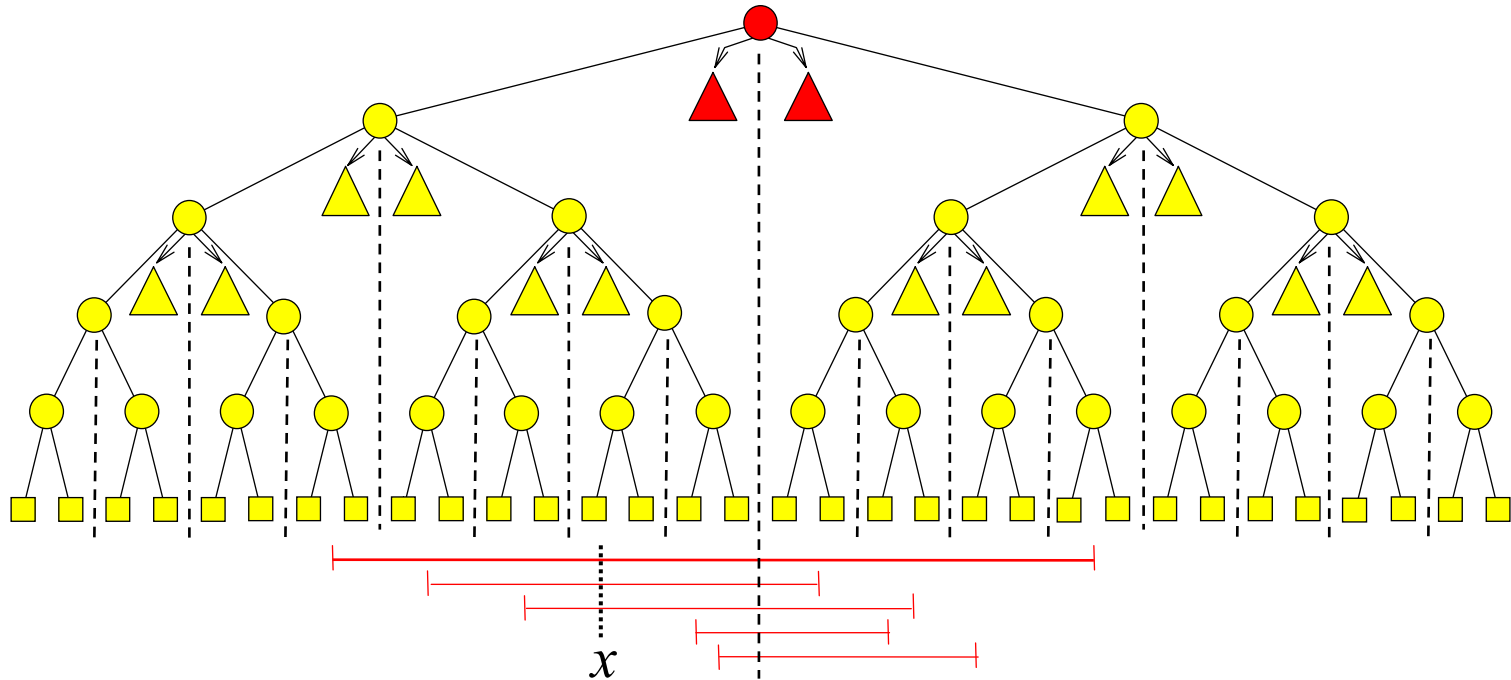
- Static solution: Persistent B-tree
 - Linear space and $O(\log_B N + T/B)$ query
- Dynamic solution: External interval tree
 - $O(\log_B N)$ update

Internal Interval Tree



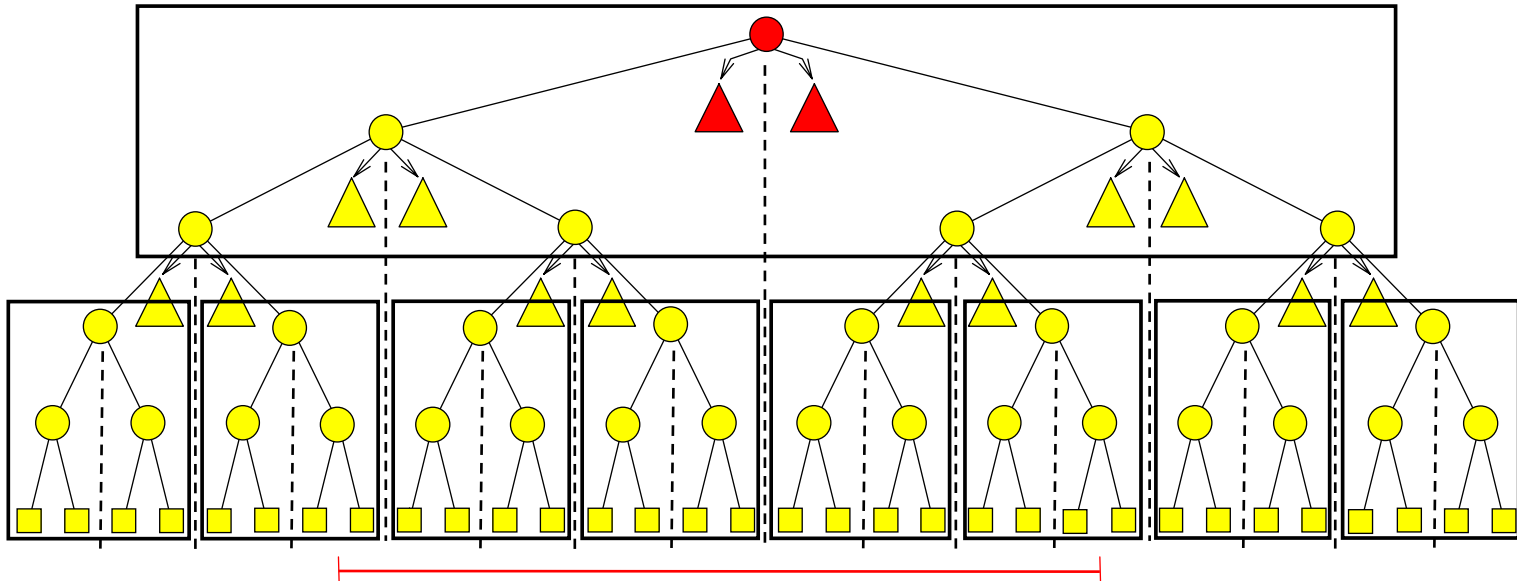
- Base tree on endpoints – “slab” X_v associated with each node v
 - Interval stored in highest node v where it contains midpoint of X_v
 - Intervals I_v associated with v stored in
 - Left slab list sorted by left endpoint (search tree)
 - Right slab list sorted by right endpoint (search tree)
- ⇒ Linear space and $O(\log N)$ update

Internal Interval Tree



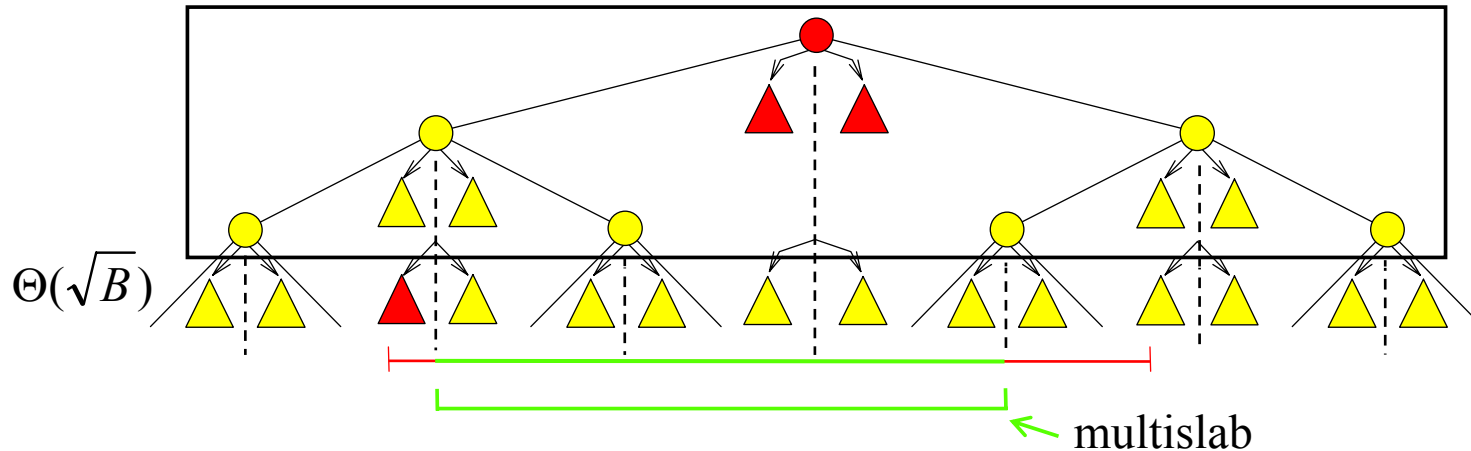
- **Query** with x on left side of midpoint of X_{root}
 - Search **left slab list** left-right until finding non-stabbed interval
 - Recurse in left child
- $\Rightarrow O(\log N + T)$ query bound

Externalizing Interval Tree



- **Natural idea:**
 - Block tree
 - Use B-tree for **slab lists**
- Number of stabbed intervals in large slab list may be small (or zero)
 - We can be forced to do I/O in each of $O(\log N)$ nodes

Externalizing Interval Tree

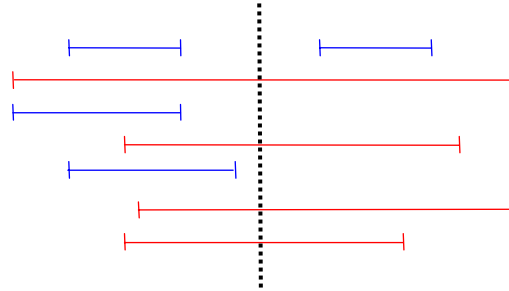


- **Idea:**

- Decrease fan-out to $\Theta(\sqrt{B}) \Rightarrow$ height remains $O(\log_B N)$
- $\Theta(\sqrt{B})$ slabs define $\Theta(B)$ **multislabs**
- Interval stored in two slab lists (as before) and one **multislab list**
- Intervals in small multislab lists collected in **underflow structure**
- Query answered in v by looking at 2 slab lists and not $O(\log N)$

External Interval Tree

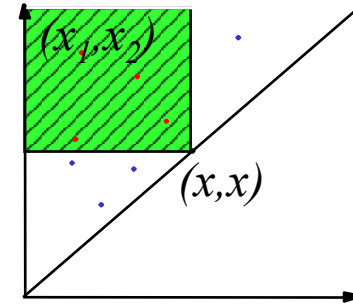
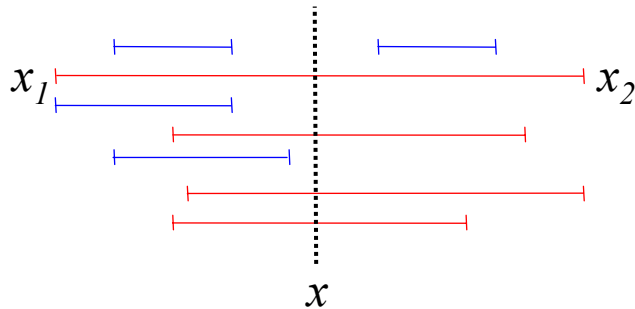
- Linear space, $O(\log_B N + T/B)$ query, $O(\log_B N)$ update



- **General solution techniques:**
 - **Filtering:** Charge part of query cost to output
 - **Bootstrapping:**
 - * Use $O(B^2)$ size structure in each internal node
 - * Constructed using persistence
 - * Dynamic using global rebuilding
 - **Weight-balanced B-tree:** Split/fuse in amortized $O(1)$

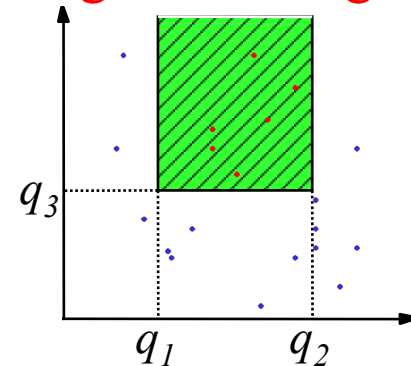
Last time: Three-Sided Range Queries

- Interval management: “1.5 dimensional” search



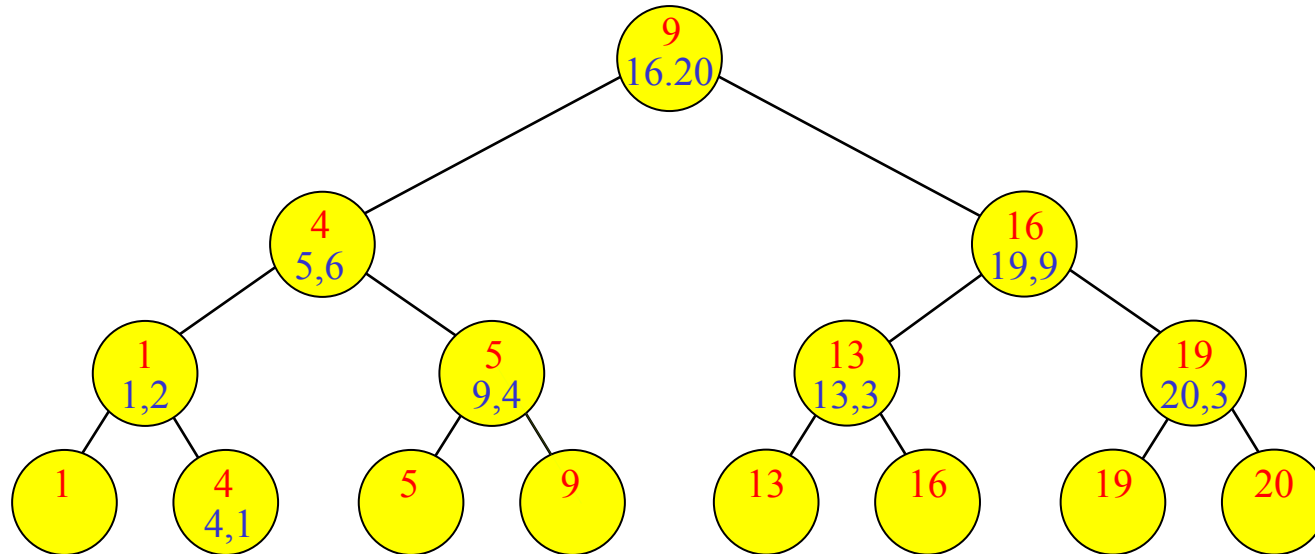
- More general 2d problem: **Dynamic 3-sided range searching**

- Maintain set of points in plane such that given query (q_1, q_2, q_3) , all points (x, y) with $q_1 \leq x \leq q_2$ and $y \geq q_3$ can be found efficiently



- Linear space, $O(\log_B N + T/B)$ query static solution using persistence
 - Dynamic: External priority search tree

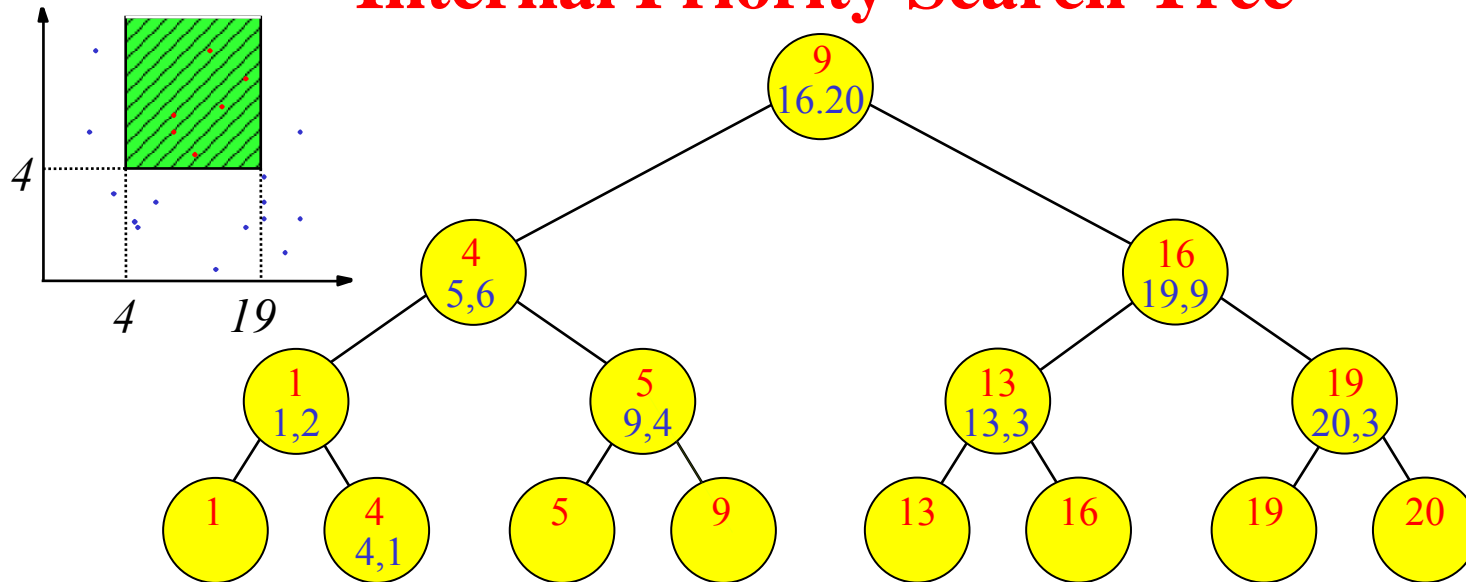
Internal Priority Search Tree



- **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

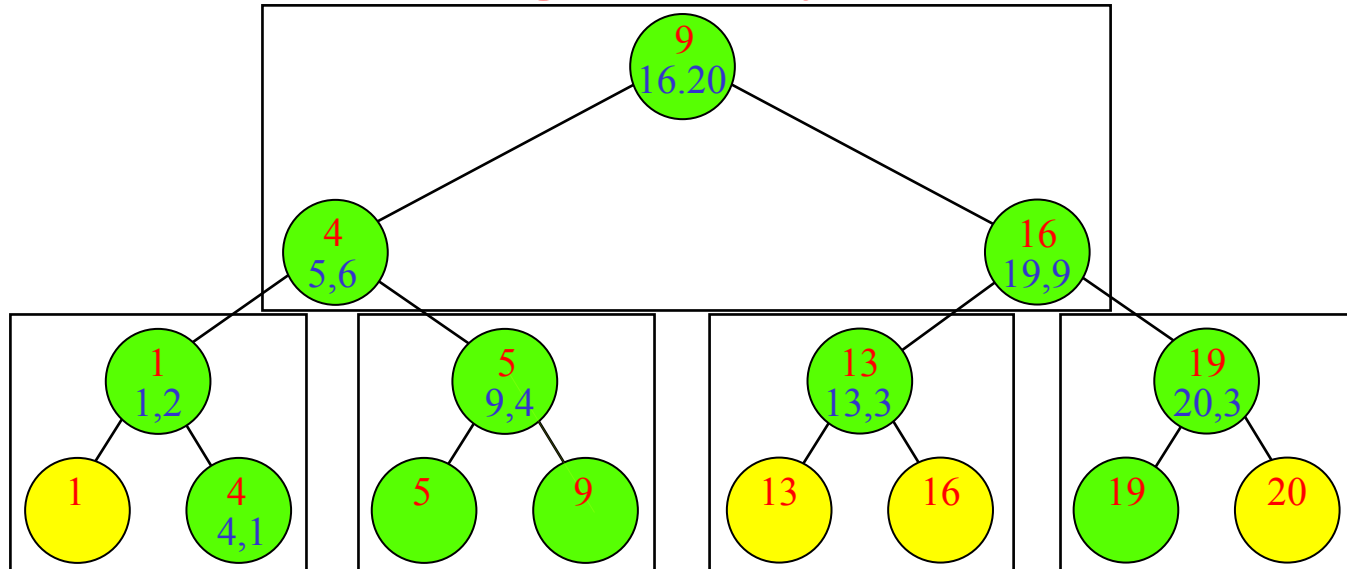
⇒ Linear space and $O(\log N)$ update

Internal Priority Search Tree



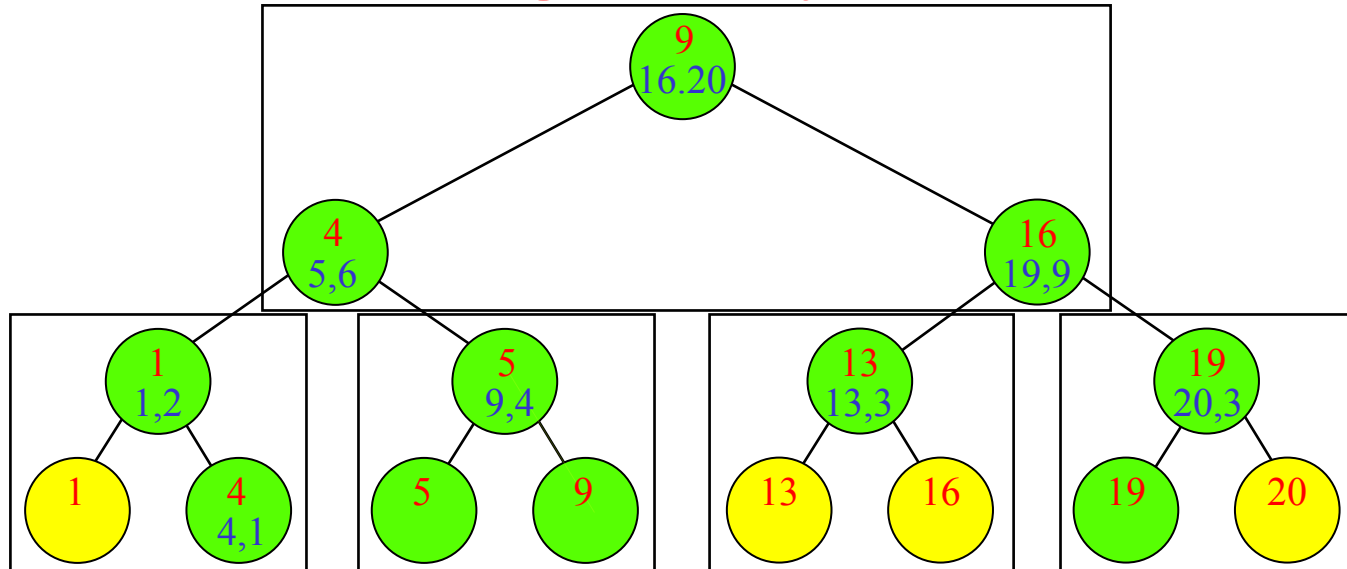
- **Query** with (q_1, q_2, q_3) starting at root v :
 - Report point in v if satisfying query
 - Visit both children of v if point reported
 - Always visit child(s) of v on path(s) to q_1 and q_2
- $\Rightarrow O(\log N + T)$ query

Externalizing Priority Search Tree



- **Natural idea:** Block tree
 - **Problem:**
 - $O(\log_B N)$ I/Os to follow paths to q_1 and q_2
 - But $O(T)$ I/Os may be used to visit other nodes (“overshooting”)
- $\Rightarrow O(\log_B N + T)$ query

Externalizing Priority Search Tree

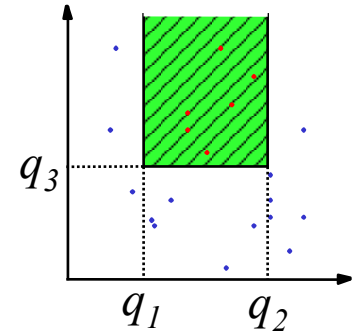
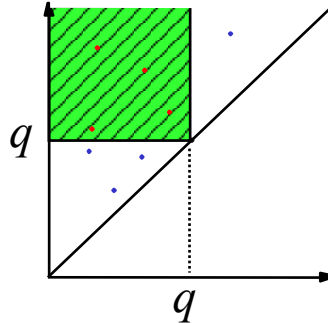


- **Solution idea:**
 - Store B points in each node \Rightarrow
 - * $O(B^2)$ points stored in each supernode
 - * B output points can pay for “overshooting”
 - **Bootstrapping:**
 - * Store $O(B^2)$ points in each supernode in static structure

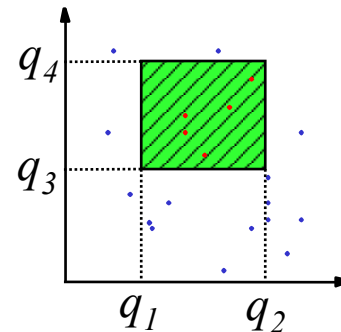
Summary/Conclusion: Priority Search Tree

- We have now discussed structures for **special cases** of two-dimensional range searching

- Space: $O(N/B)$
- Query: $O(\log_B N + T/B)$
- Updates: $O(\log_B N)$



- Cannot be obtained for general (4-sided) $2d$ range searching:
 - $O(\log_B^c N)$ query requires $\Omega\left(\frac{N}{B} \frac{\log_B N}{\log_B \log_B N}\right)$ space
 - $O\left(\frac{N}{B}\right)$ space requires $\Omega\left(\sqrt{N/B}\right)$ query



External Range Tree

- **Base tree**: Weight balanced tree with branching parameter $\Theta(\log_B N)$ and leaf parameter B on x -coordinates

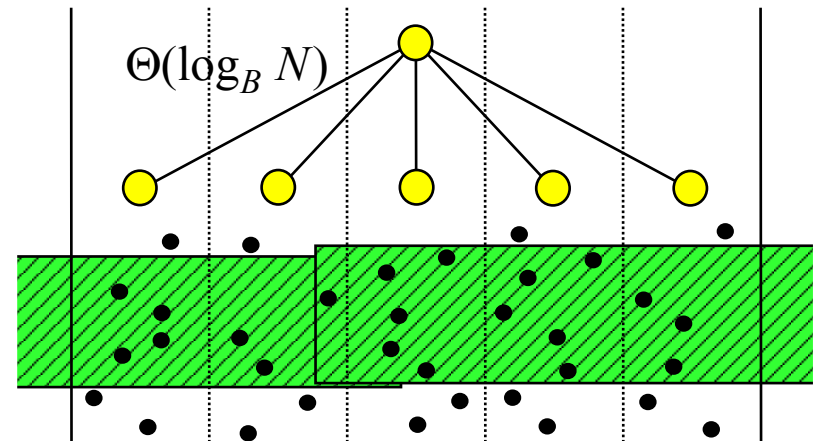
⇓

$$O(\log_{\log_B N} N) = O\left(\frac{\log_B N}{\log_B \log_B N}\right) \text{ height}$$

- Points below each node stored in 4 linear space **secondary structures**:
 - “Right” **priority search tree**
 - “Left” **priority search tree**
 - **B-tree** on y -coordinates
 - **Interval (priority search) tree**

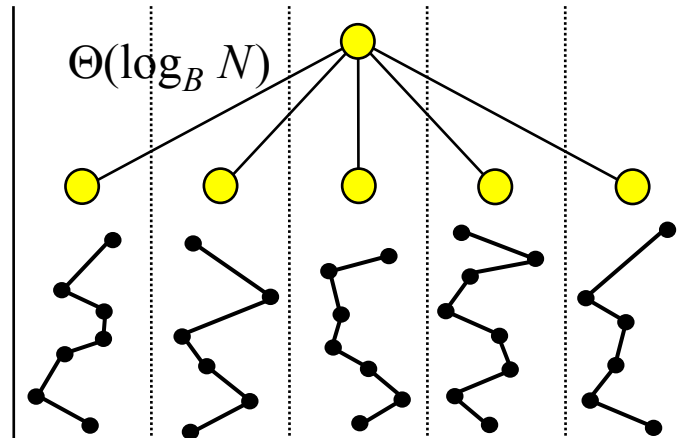
⇓

$$O\left(\frac{N}{B} \frac{\log_B N}{\log_B \log_B N}\right) \text{ space}$$



External Range Tree

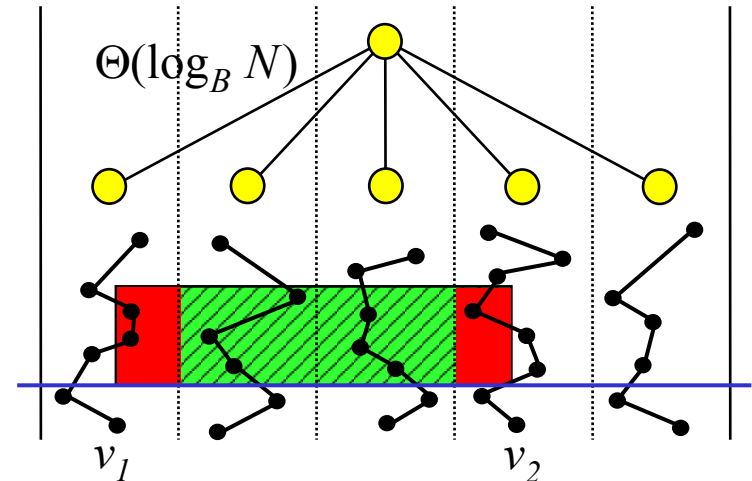
- Secondary **interval tree**:
 - Connect points in each slab in y -order
 - Project obtained segments in y -axis



- Intervals stored in interval tree
 - * Interval augmented with pointer to corresponding points in y -coordinate B-tree in corresponding child node

External Range Tree

- **Query** with (q_1, q_2, q_3, q_4) answered in top node with q_1 and q_2 in different slabs v_1 and v_2
- **Points in slab v_1**
 - Found with 3-sided query in v_1 using right priority search tree
- **Points in slab v_2**
 - Found with 3-sided query in v_2 using left priority search tree
- **Points in slabs between v_1 and v_2**
 - Answer stabbing query with q_3 using interval tree
 - \Rightarrow first point above q_3 in each of the $O(\log_B N)$ slabs
 - Find points using y-coordinate B-tree in $O(\log_B N)$ slabs

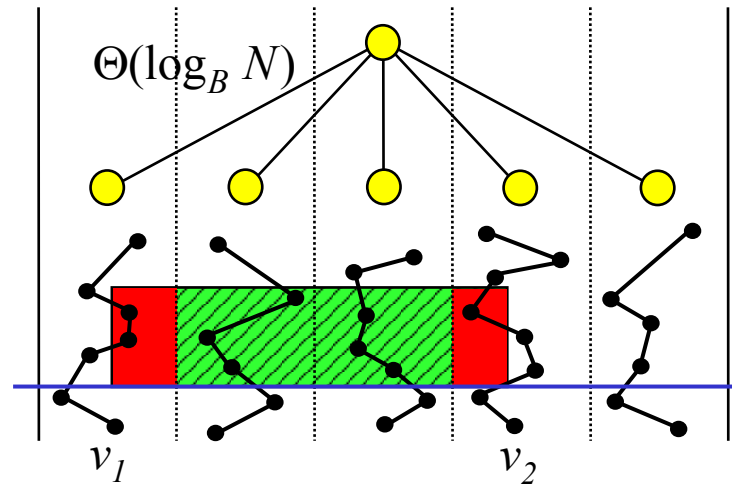


External Range Tree

- **Query analysis:**
 - $O(\log_B N)$ I/Os to find relevant node
 - $O(\log_B N + T/B)$ I/Os to answer two 3-sided queries
 - $O(\log_B N + \log_B N/B) = O(\log_B N)$ I/Os to query interval tree
 - $O(\log_B N + T/B)$ I/Os to traverse $O(\log_B N)$ B-trees

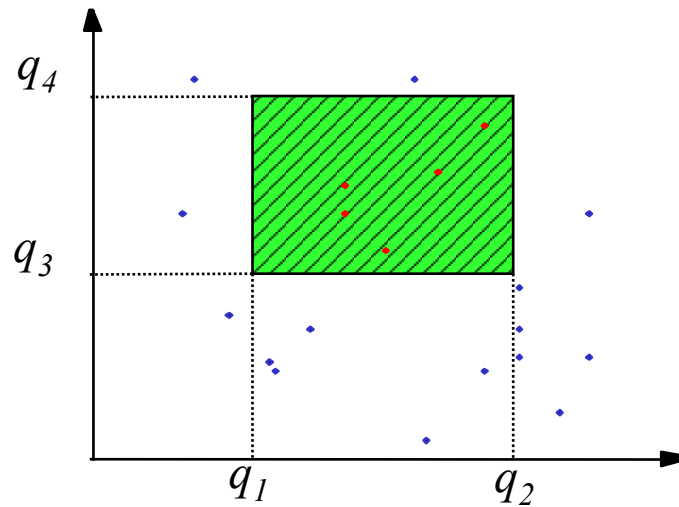


$O(\log_B N + T/B)$ I/Os

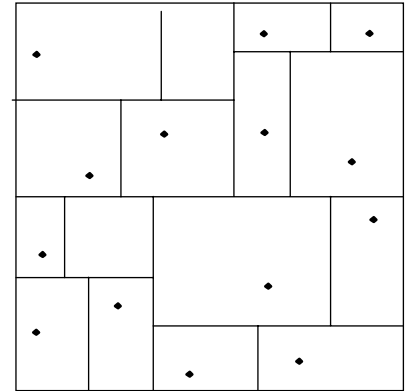
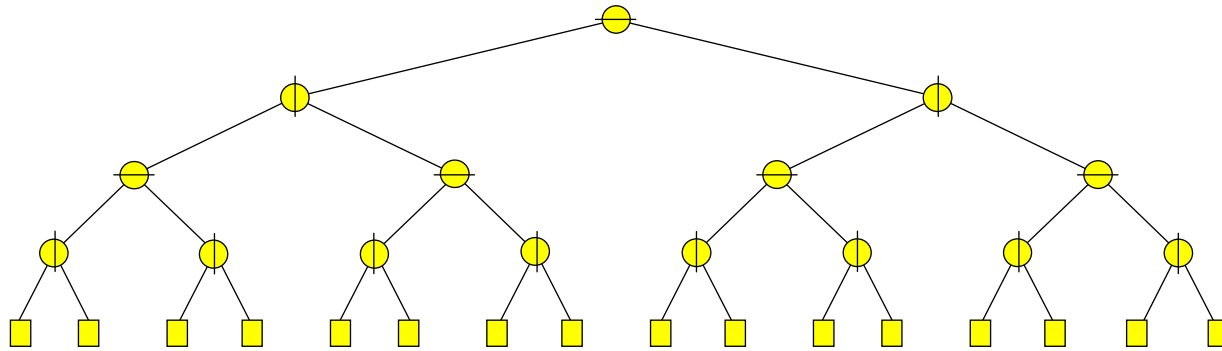


Summary: External Range Tree

- *2d* range searching in $O\left(\frac{N}{B} \frac{\log_B N}{\log_B \log_B N}\right)$ space
 - $O(\log_B N + T/B)$ I/O query
 - $O\left(\frac{\log_B^2 N}{\log_B \log_B N}\right)$ I/O update
- **Optimal** among $O(\log_B N + T/B)$ query structures



kdB-tree

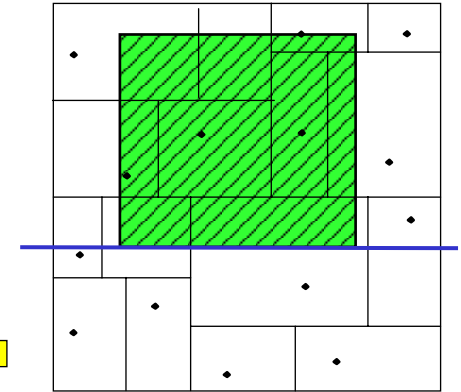
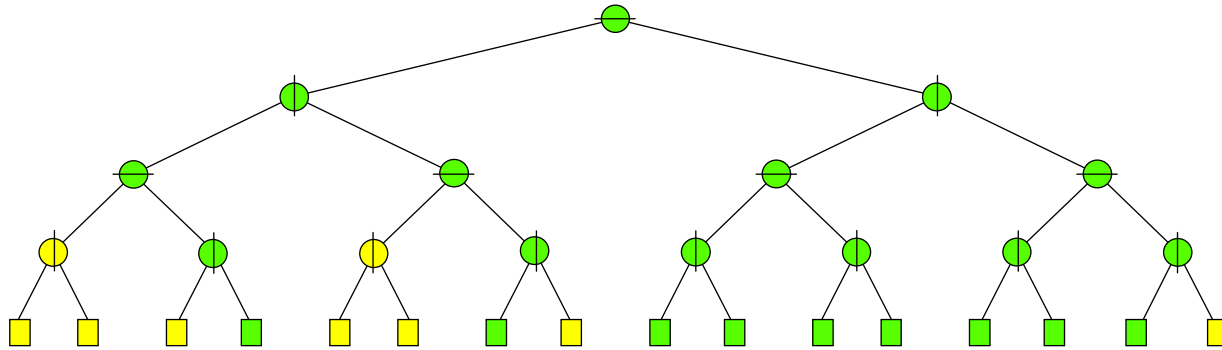


- **kd-tree:**
 - Recursive subdivision of point-set into two half using vertical/horizontal line
 - Horizontal line on even levels, vertical on uneven levels
 - One point in each leaf



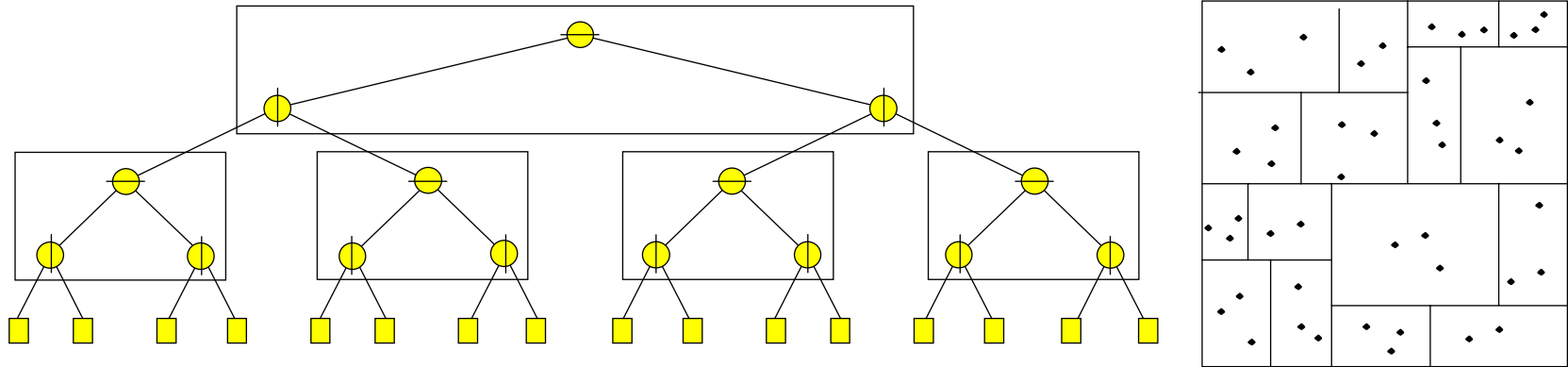
Linear space and logarithmic height

kd-Tree: Query



- Query
 - Recursively visit nodes corresponding to regions intersecting query
 - Report point in trees/nodes completely contained in query
- 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

kdB-tree

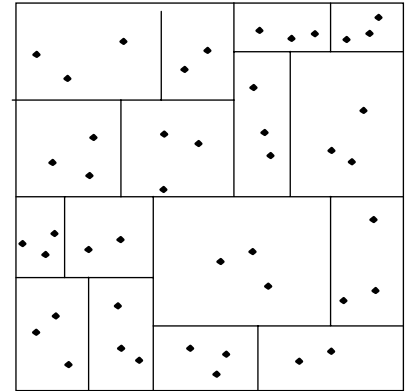
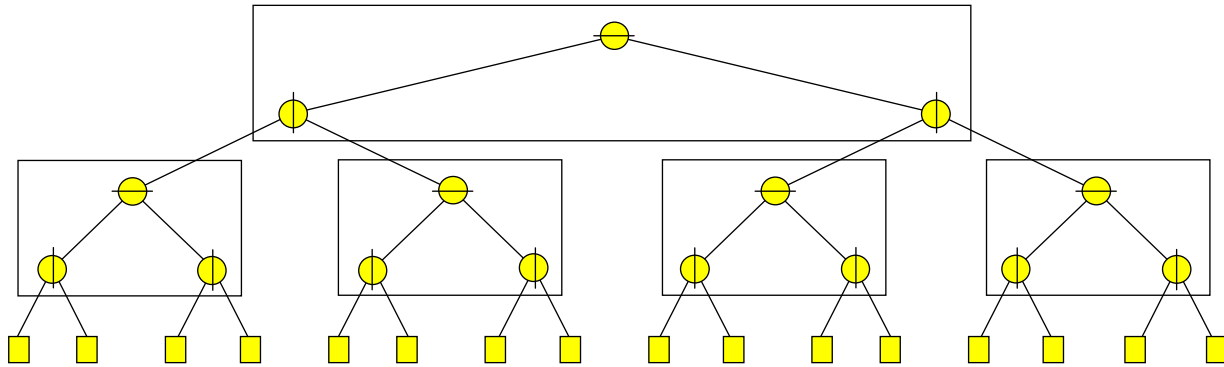


- **kdB-tree:**
 - Stop subdivision when leaf contains between $B/2$ and B points
 - BFS-blocking of internal nodes
- **Query** as before
 - Analysis as before but each region now contains $\Theta(B)$ points

⇓

$$O(\sqrt{N/B} + T/B) \text{ I/O query}$$

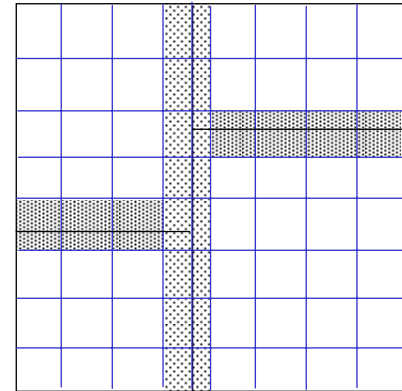
Construction of kdB-tree



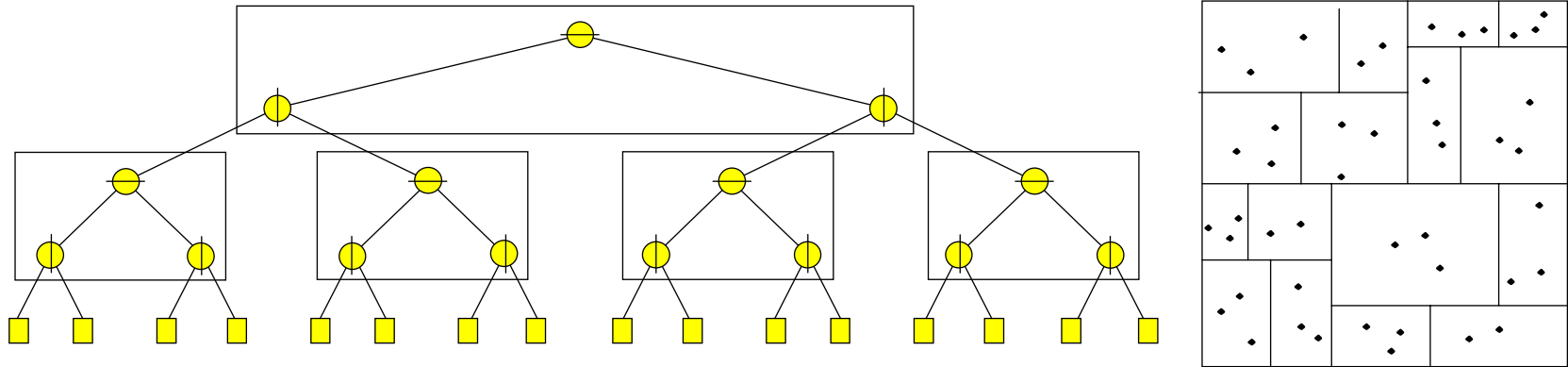
- Simple $O(\frac{N}{B} \log_2 \frac{N}{B})$ algorithm
 - Find median of y -coordinates (construct root)
 - Distribute point based on median
 - Recursively build subtrees
 - Construct BFS-blocking top-down
- Idea in improved $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ algorithm
 - Construct $\log \sqrt{M/B}$ levels at a time using $O(N/B)$ I/Os

Construction of kdB-tree

- Sort N points by x - and by y -coordinates using $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
 - Building $\log \sqrt{M/B}$ levels ($\sqrt{M/B}$ nodes) in $O(N/B)$ I/Os:
 1. Construct $\sqrt{M/B}$ by $\sqrt{M/B}$ grid with $\frac{N}{\sqrt{M/B}}$ points in each slab
 2. Count number of points in each grid cell and store in memory
 3. Find slab s with median x -coordinate
 4. Scan slab s to find median x -coordinate and construct node
 5. Split slab containing median x -coordinate and update counts
 6. Recurse on each side of median x -coordinate using grid (step 3)
- ⇒ Grid grows to $M/B + \sqrt{M/B} \cdot \sqrt{M/B} = \Theta(M/B)$ during algorithm
- ⇒ Each node constructed in $O(N / (\sqrt{M/B} \cdot B))$ I/Os



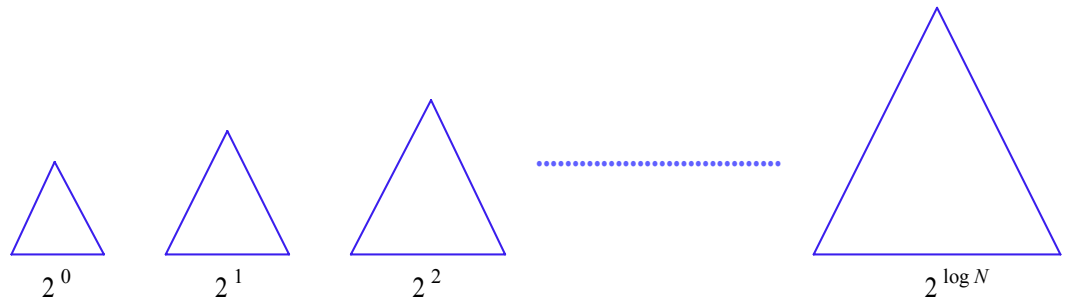
kdB-tree



- kdB-tree:
 - Linear space
 - Query in $O(\sqrt{N/B} + T/B)$ I/Os
 - Construction in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
 - Point search in $O(\log_B N)$ I/Os
- Dynamic?
 - Deletions relatively easily in $O(\log_B^2 N)$ I/Os (partial rebuilding)

kdB-tree Insertion using Logarithmic Method

- Partition pointset S into subsets $S_0, S_1, \dots, S_{\log N}$, $|S_i| = 2^i$ or $|S_i| = 0$
- Build kdB-tree D_i on S_i



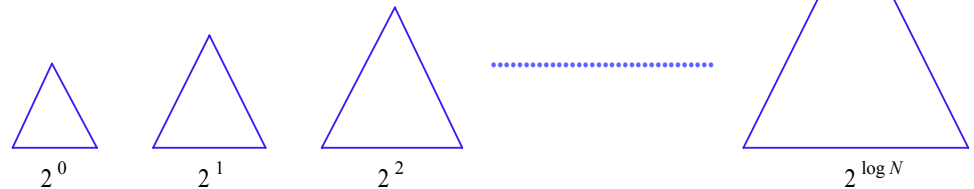
- **Query:** Query each $D_i \Rightarrow \sum_{i=0}^{\log N} O(\sqrt{2^i/B} + T_i/B) = O(\sqrt{N/B} + T/B)$
- **Insert:** Find first empty D_i and construct D_i out of $1 + \sum_{j=0}^{i-1} 2^j = 2^i$ elements in S_0, S_1, \dots, S_{i-1}
 - $O(\frac{2^i}{B} \log_{M/B} \frac{2^i}{B})$ I/Os $\Rightarrow O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ per moved point
 - Point moved $O(\log N)$ times
 - $\Rightarrow O(\frac{1}{B} \log_{M/B} \frac{N}{B} \log N) = O(\log_B^2 N)$ I/Os amortized

kdB-tree Insertion and Deletion

- **Insert:** Use logarithmic method ignoring deletes
- **Delete:** Simply delete point p from relevant D_i
 - i can be calculated based on # insertions since p was inserted
 - # insertions calculated by storing insertion number of each point in separate B-tree



$O(\log_B N)$ extra update cost

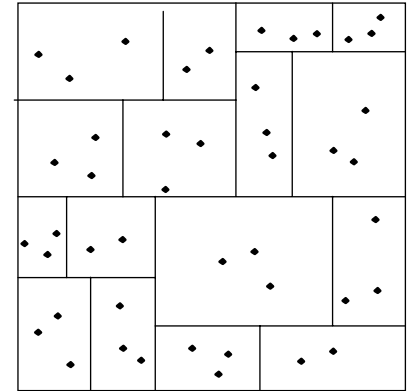
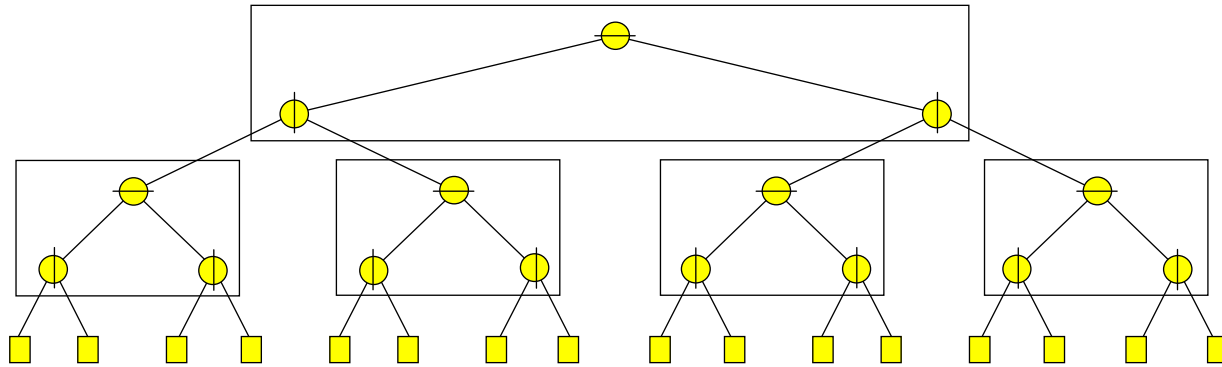


- To maintain $O(\log N)$ structures D_i
 - Perform global rebuild after every $\Theta(N)$ updates

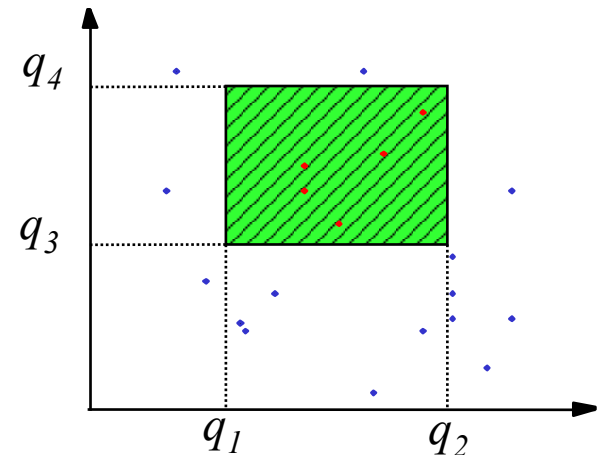


$O\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right) = O(\log_B N)$ extra update cost

Summary: kdB-tree



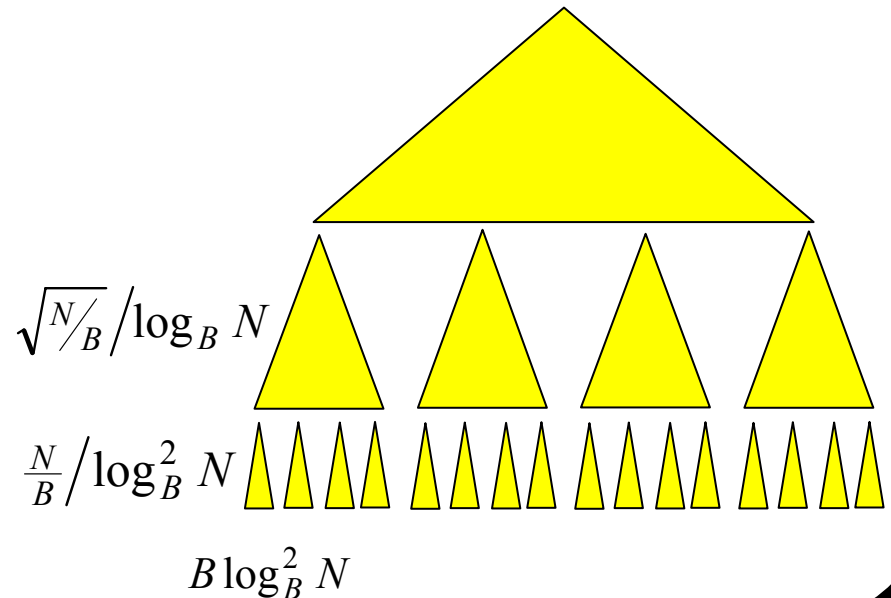
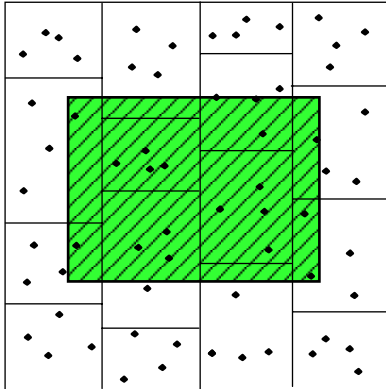
- *2d range searching* in $O(N/B)$ space
 - Query in $O(\sqrt{N/B} + T/B)$ I/Os
 - Construction in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
 - Updates in $O(\log_B^2 N)$ I/Os
- *Optimal* query among linear space structures



O-Tree Structure

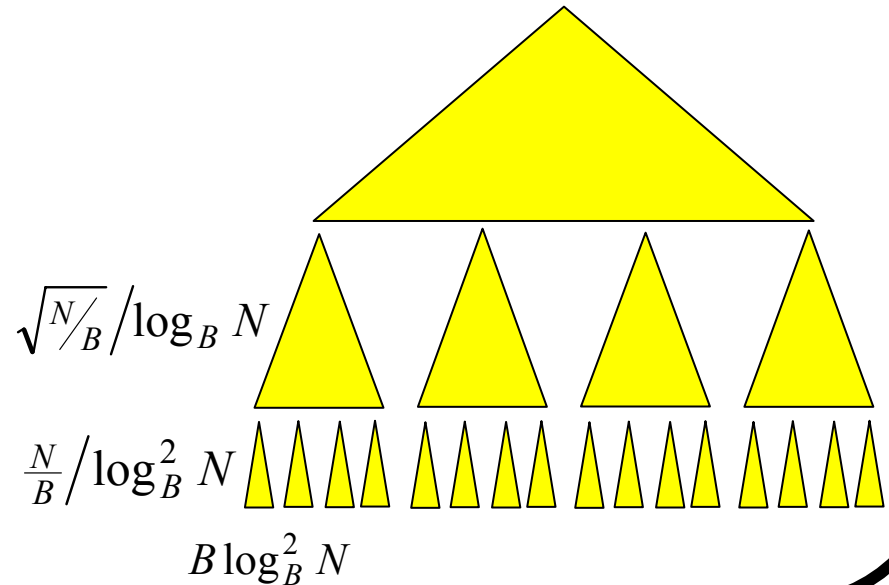
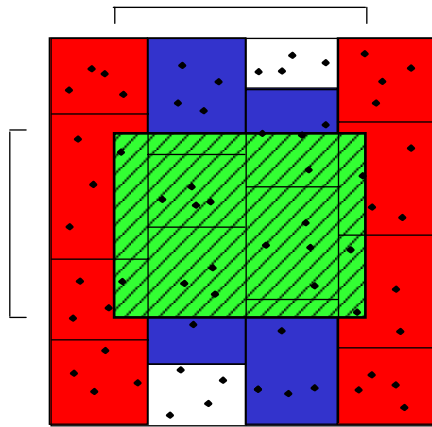
- **O-tree:**

- B-tree on $\Theta(\sqrt{N/B}/\log_B N)$ **vertical** slabs
- B-tree on $\Theta(\sqrt{N/B}/\log_B N)$ **horizontal** slabs in each vertical slab
- **kdB-tree** on $\Theta\left(\frac{N}{(\sqrt{N/B}/\log_B N)^2}\right) = \Theta(B \log_B^2 N)$ points in each leaf



O-Tree Query

- Perform rangearch with q_1 and q_2 in **vertical** B-tree
 - Query all **kdB-trees** in leaves of two **horizontal** B-trees with x -interval intersected but not spanned by query
 - Perform rangearch with q_3 and q_4 **horizontal** B-trees with x -interval spanned by query
 - * Query all **kdB-trees** with range intersected by query

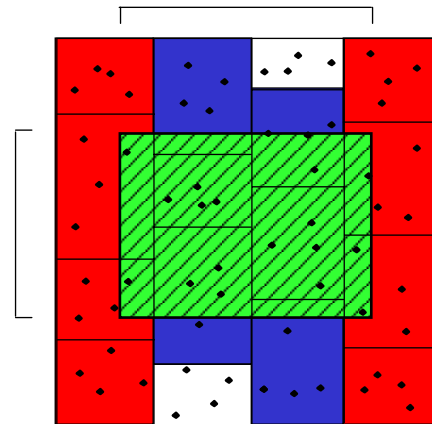


O-Tree Query Analysis

- **Vertical** B-tree query: $O(\log_B(\sqrt{N/B}/\log_B N)) = O(\sqrt{N/B})$
- Query of all **kdB-trees** in leaves of two **horizontal** B-trees:
 $O(\sqrt{N/B}/\log_B N) \cdot O(\sqrt{B \log_B^2 N/B}) + O(\frac{T}{B}) = O(\sqrt{N/B} + \frac{T}{B})$
- Query $O(\sqrt{N/B}/\log_B N)$ **horizontal** B-trees:
 $O(\sqrt{N/B}/\log_B N) \cdot O(\log_B(\sqrt{N/B}/\log_B N)) = O(\sqrt{N/B})$
- Query $2 \cdot O(\sqrt{N/B}/\log_B N)$ **kdB-trees** not completely in query
 $2 \cdot O(\sqrt{N/B}/\log_B N) \cdot O(\sqrt{B \log_B^2 N/B}) + O(\frac{T}{B}) = O(\sqrt{N/B} + \frac{T}{B})$
- Query in **kdB-trees** completely contained in query: $O(\frac{T}{B})$

⇓

$$O(\sqrt{N/B} + \frac{T}{B}) \text{ I/Os}$$



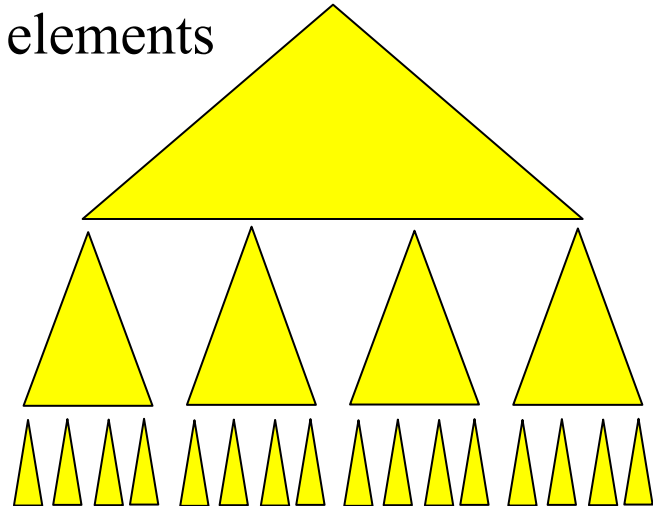
O-Tree Update

- **Insert:**
 - Search in **vertical** B-tree: $O(\log_B N)$ I/Os
 - Search in **horizontal** B-tree: $O(\log_B N)$ I/Os
 - Insert in **kdB-tree**: $O(\log_B^2 (B \log_B^2 N)) = O(\log_B N)$ I/Os
- Use **global rebuilding** when structures grow too big/small
 - B-trees not contain $\Theta(\sqrt{N/B} / \log_B N)$ elements
 - kB-trees not contain $\Theta(B \log_B^2 N)$ elements



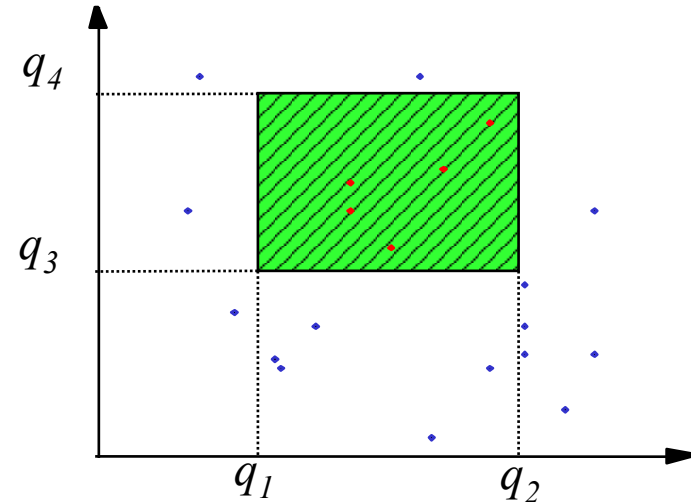
$O(\log_B N)$ I/Os

- **Deletes** can be handled
in $O(\log_B N)$ I/Os similarly



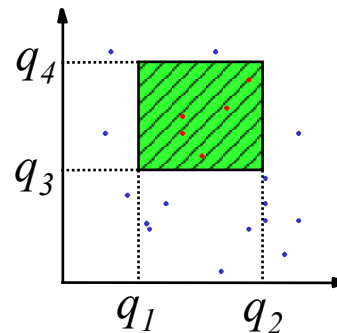
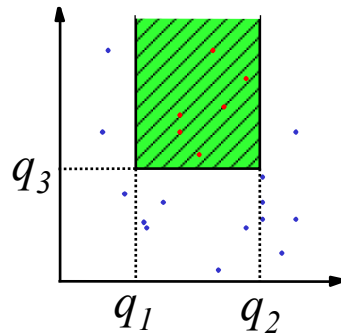
Summary: O-Tree

- $2d$ range searching in linear space
 - $O(\sqrt{N/B} + \frac{T}{B})$ I/O query
 - $O(\log_B N)$ I/O update
- **Optimal** among structures using linear space
- Can be extended to work in d -dimensions with optimal query bound $O((\frac{N}{B})^{1-\frac{1}{d}} + \frac{T}{B})$



Summary/Conclusion: 3 and 4-sided Queries

- 3-sided 2d range searching: **External priority search tree**
 - $O(\log_B N + T/B)$ query, $O(\frac{N}{B})$ space, $O(\log_B N)$ update

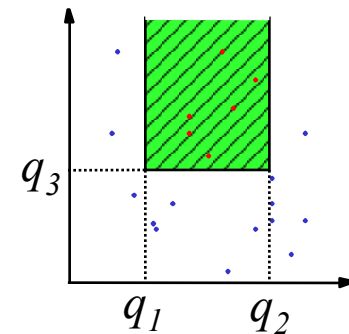
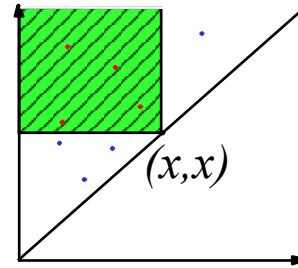


- General (4-sided) 2d range searching:
 - **External range tree**: $O(\log_B N + T/B)$ query, $O(\frac{N}{B} \frac{\log_B N}{\log_B \log_B N})$ space, $O(\frac{\log_B^2 N}{\log_B \log_B N})$ update
 - **O-tree**: $O(\sqrt{N/B} + T/B)$ query, $O(\frac{N}{B})$ space, $O(\log_B N)$ update

Summary/Conclusion: Tools and Techniques

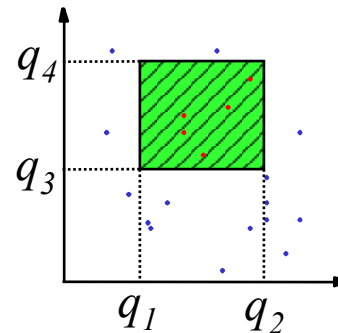
- **Tools:**

- B-trees
- Persistent B-trees
- Buffer trees
- Logarithmic method
- Weight-balanced B-trees
- Global rebuilding



- **Techniques:**

- Bootstrapping
- Filtering

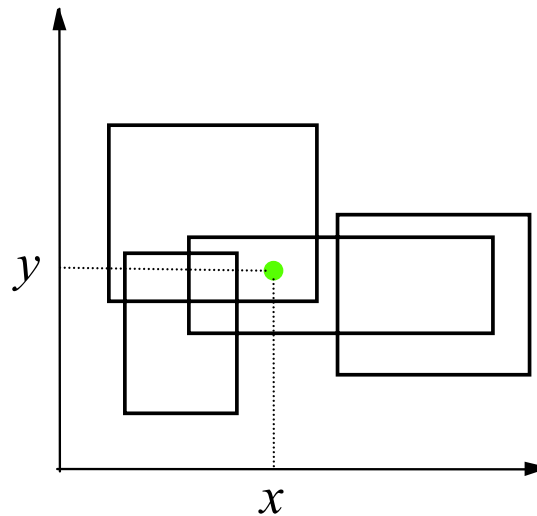


Other results

- 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

Point Enclosure Queries

- Dual of planar range searching problem
 - Report all rectangles containing query point (x,y)



- **Internal memory:**
 - Can be solved in $O(N)$ space and $O(\log N + T)$ time

Point Enclosure Queries

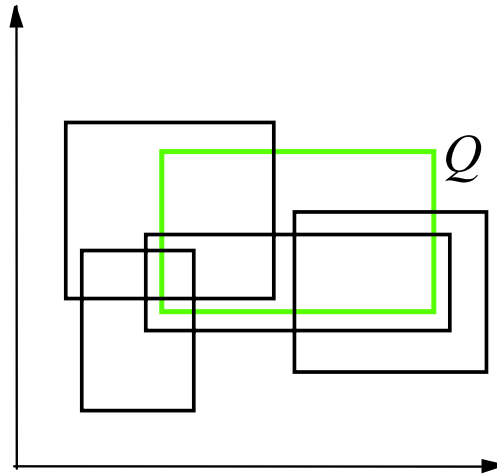
- Similarity between internal and external results (*space, query*)

	Internal	External
1d range search	$(N, \log N + T)$	$(N/B, \log_B N + T/B)$
3-sided 2d range search	$(N, \log N + T)$	$(N/B, \log_B N + T/B)$
2d range search	$(N, \sqrt{N} + T)$ $(N \frac{\log N}{\log \log N}, \log N + T)$	$(N/B, \sqrt{N/B} + T/B)$ $(\frac{N}{B} \frac{\log_B N}{\log_B \log_B N}, \log_B N + T/B)$
2d point enclosure	$(N, \log N + T)$	$(N/B, \log N + T/B)$ $(N/B, \log_B N + T/B)?$ $(N/B^{1-\epsilon}, \log_B N + T/B)$

– in general tradeoff between space and query I/O

Next Time: Rectangle Range Searching

- Report all rectangles intersecting query rectangle Q



References

- **External Memory Geometric Data Structures**
Lecture notes by Lars Arge.
 - Section 8-9