

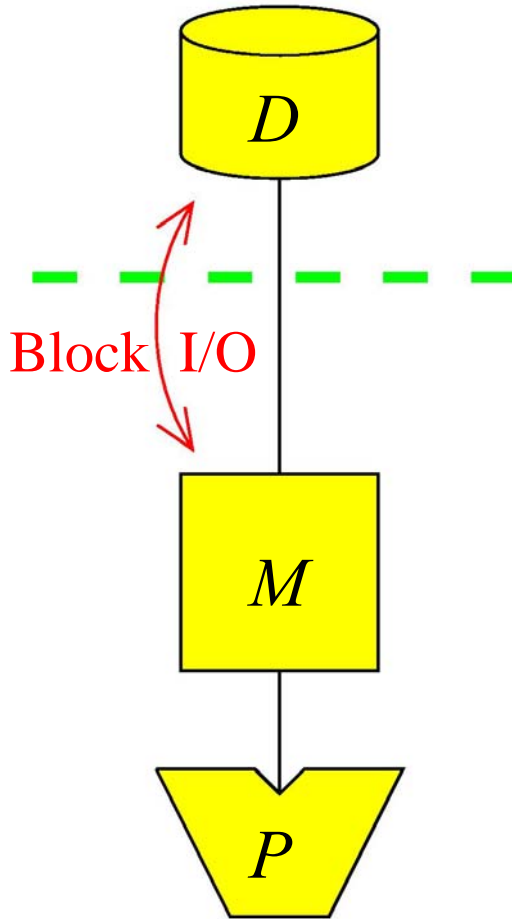
I/O-Algorithms

Lars Arge

Aarhus University

April 10, 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 $M > B^2$

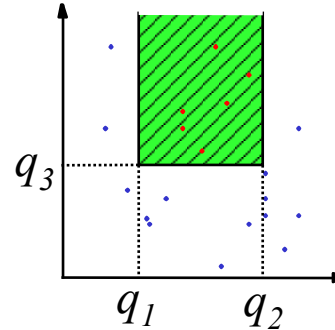
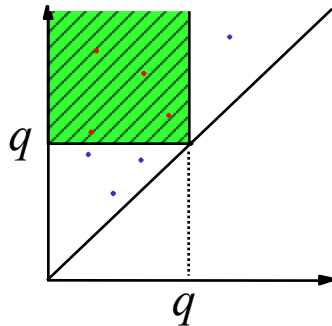
- **I/O**: Movement of block between memory and disk

Until now: Data Structures

- **B-trees**
 - Trees with fanout B , balanced using split/fuse
 - $O(\log_B N + T/B)$ query, $O(N/B)$ space, $O(\log_B N)$ update
- **Weight-balanced B-trees**
 - Weight balancing constraint rather than degree constraint
 - $\Omega(w(v))$ updates below v between consecutive operations on v
- **Persistent B-trees**
 - Update current version (getting new version)
 - Query all versions
- **Buffer-trees**
 - $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ amortized bounds using buffers and laziness

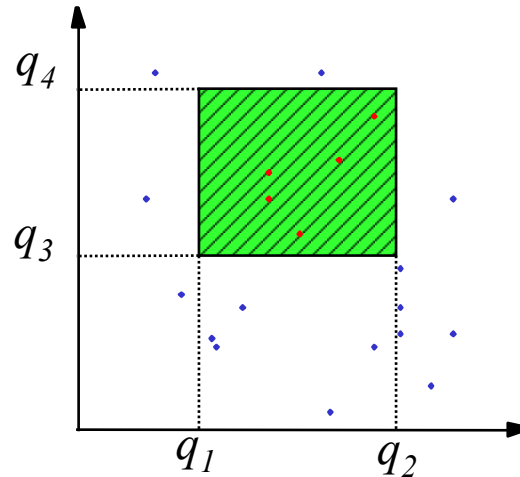
Until now: Data Structures

- Special cases of two-dimensional range search:
 - Diagonal corner queries: External interval tree
 - Three-sided queries: External priority search tree
- $O(\log_B N + T/B)$ query, $O(N/B)$ space, $O(\log_B N)$ update



- Same bounds cannot be obtained for general planar range searching

Until now: Data Structures

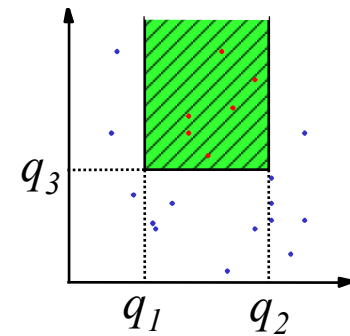
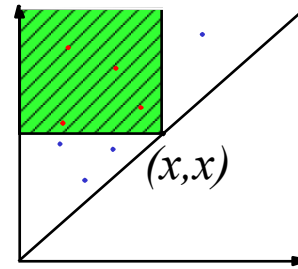


- General planer range searching:
 - **External range tree**: $O(\log_B N + T/B)$ query, $O(\frac{N}{B} \frac{\log_B N}{\log_B \log_B N})$ space, $O(\frac{\log_B^2 N}{\log_B \log_B N})$ update
 - **O-tree**: $O(\sqrt{N/B} + T/B)$ query, $O(\frac{N}{B})$ space, $O(\log_B N)$ update

Techniques

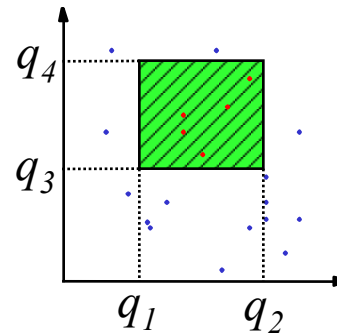
- **Tools:**

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



- **Techniques:**

- Bootstrapping
- Filtering

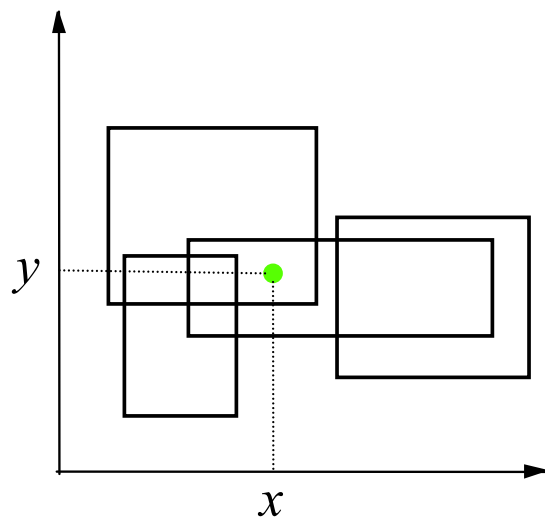


Other results

- 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

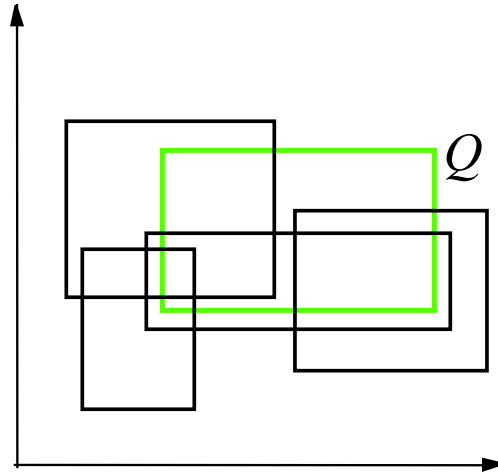
- Similarity between internal and external results (*space, query*)

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

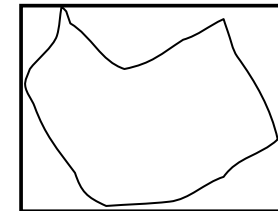
– in general tradeoff between space and query I/O

Rectangle Range Searching

- Report all rectangles intersecting query rectangle Q

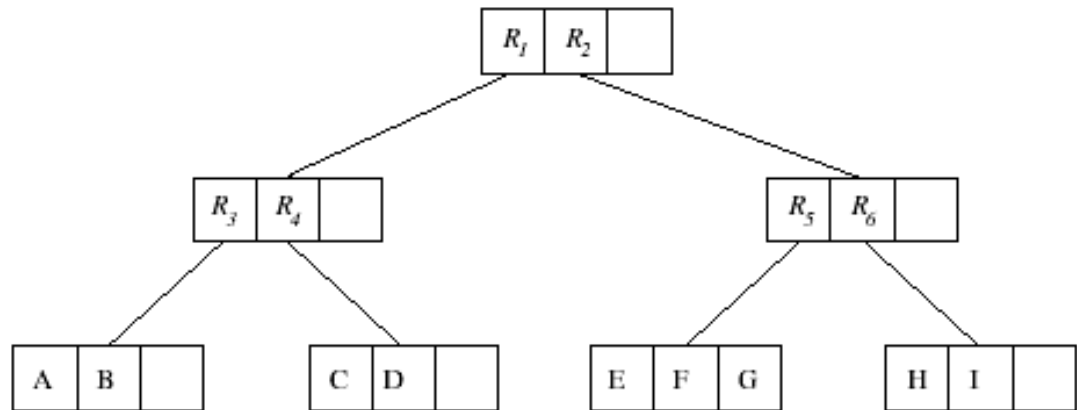
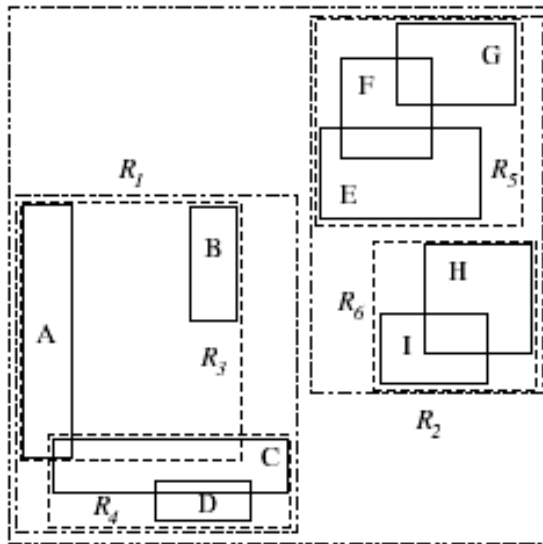


- Often used in practice when handling complex geometric objects
 - Store minimal bounding rectangles (MBR)



Rectangle Data Structures: R-Tree

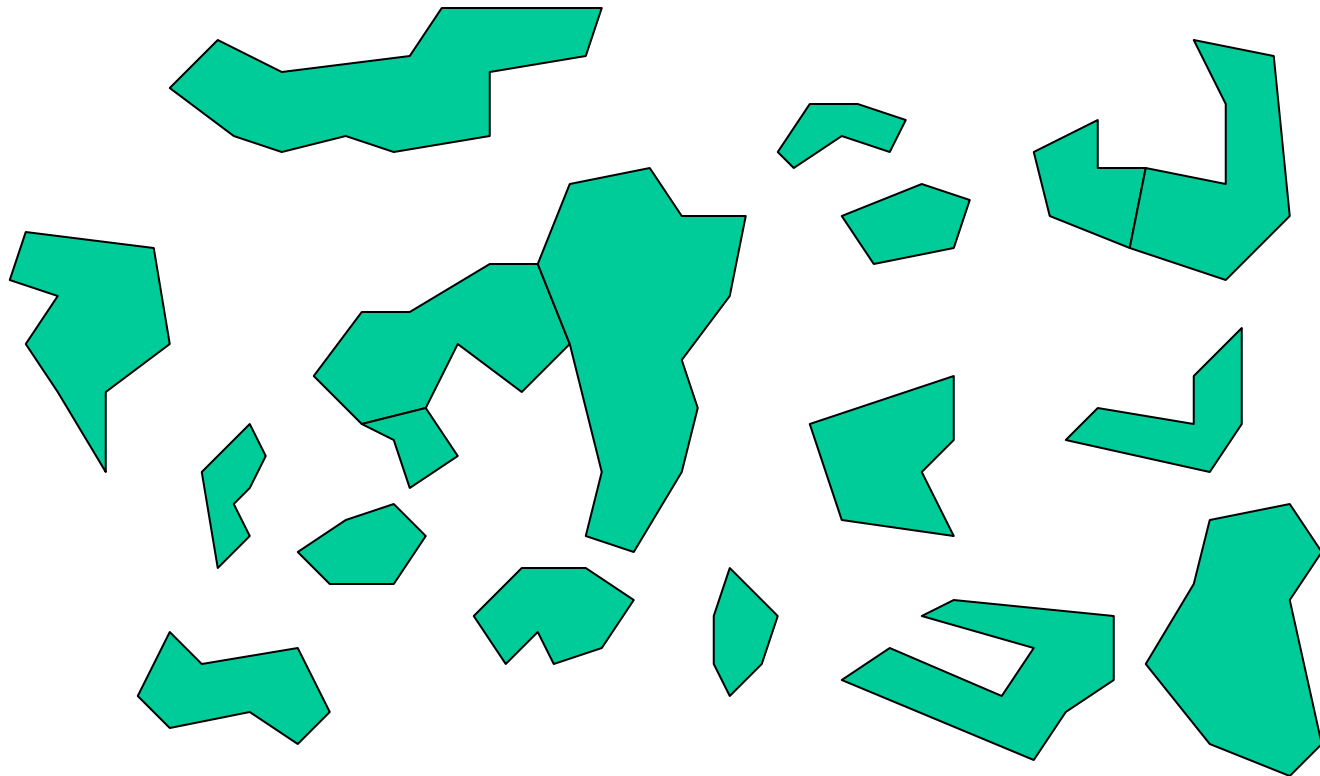
- Most common practically used rectangle range searching structure
- Similar to B-tree
 - Rectangles in leaves (on same level)
 - Internal nodes contain MBR of rectangles below each child



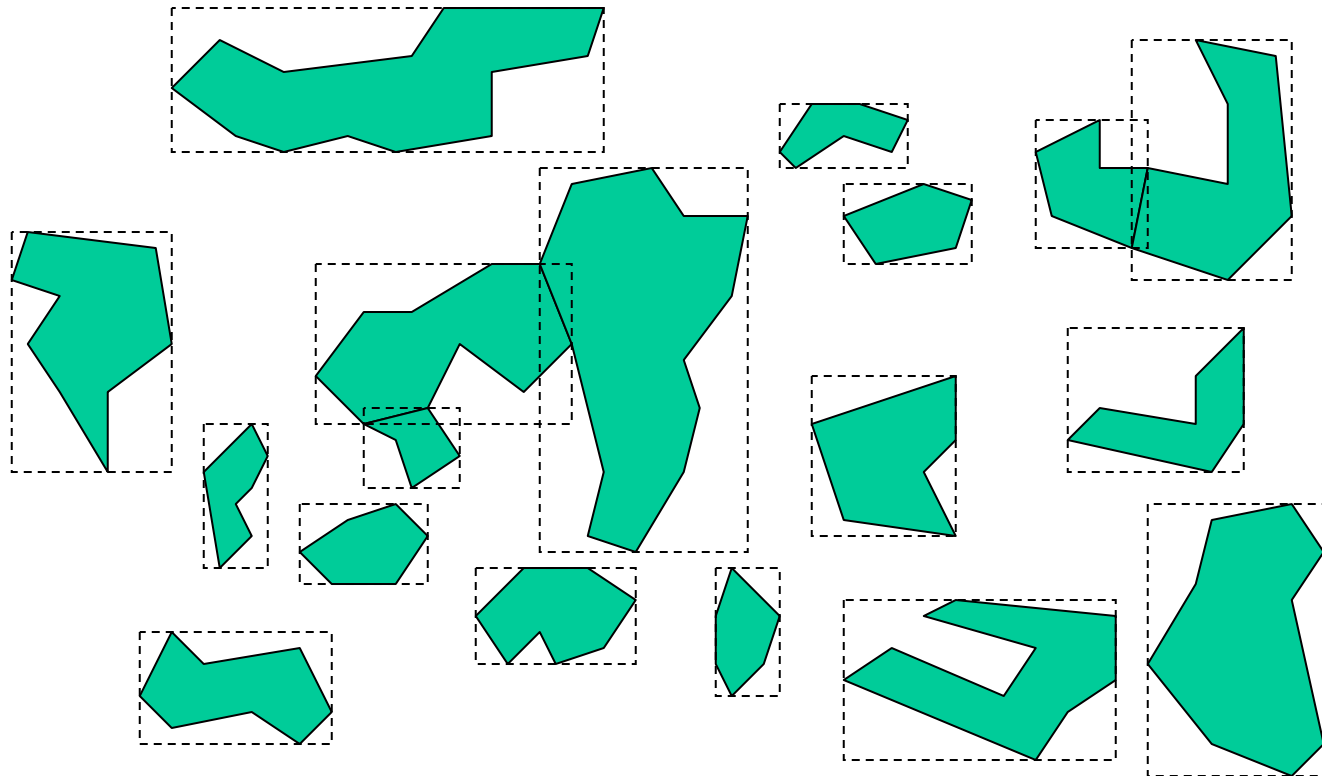
- Note: Arbitrary order in leaves/grouping order



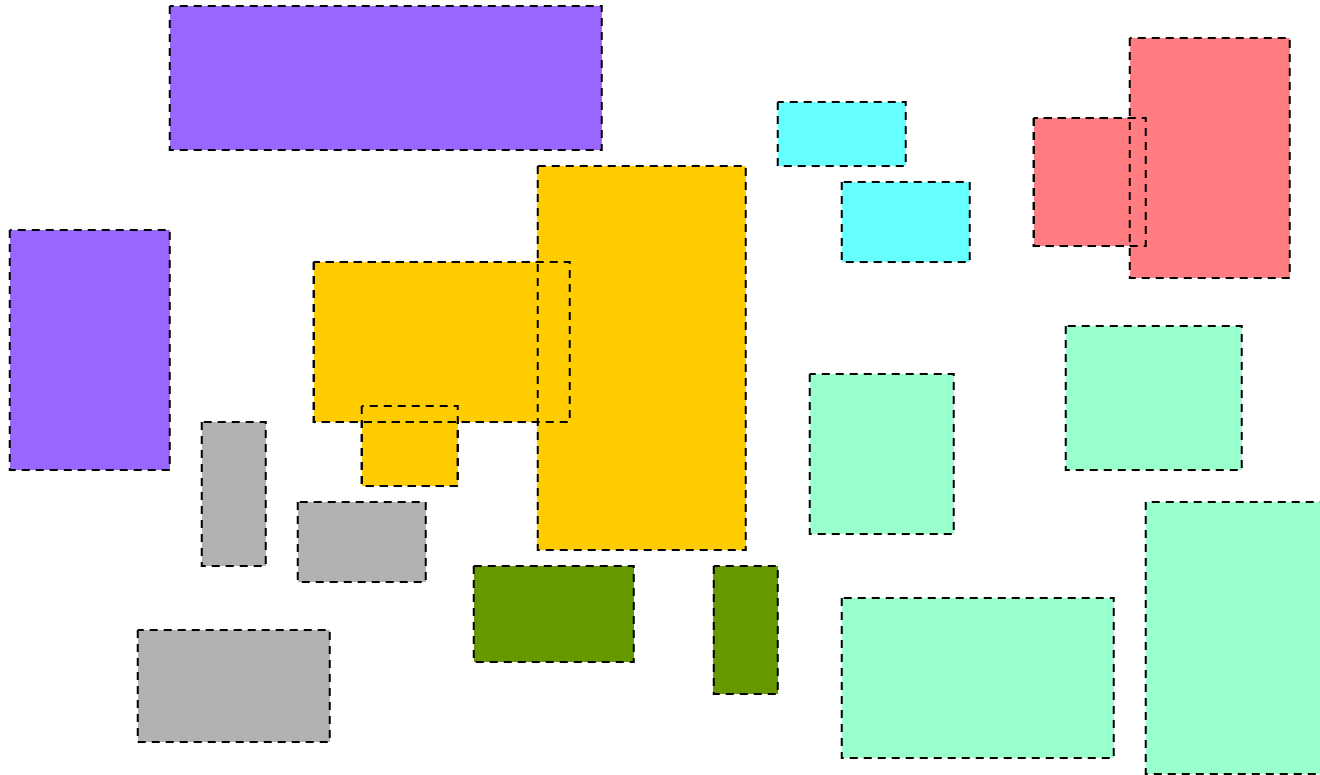
Example



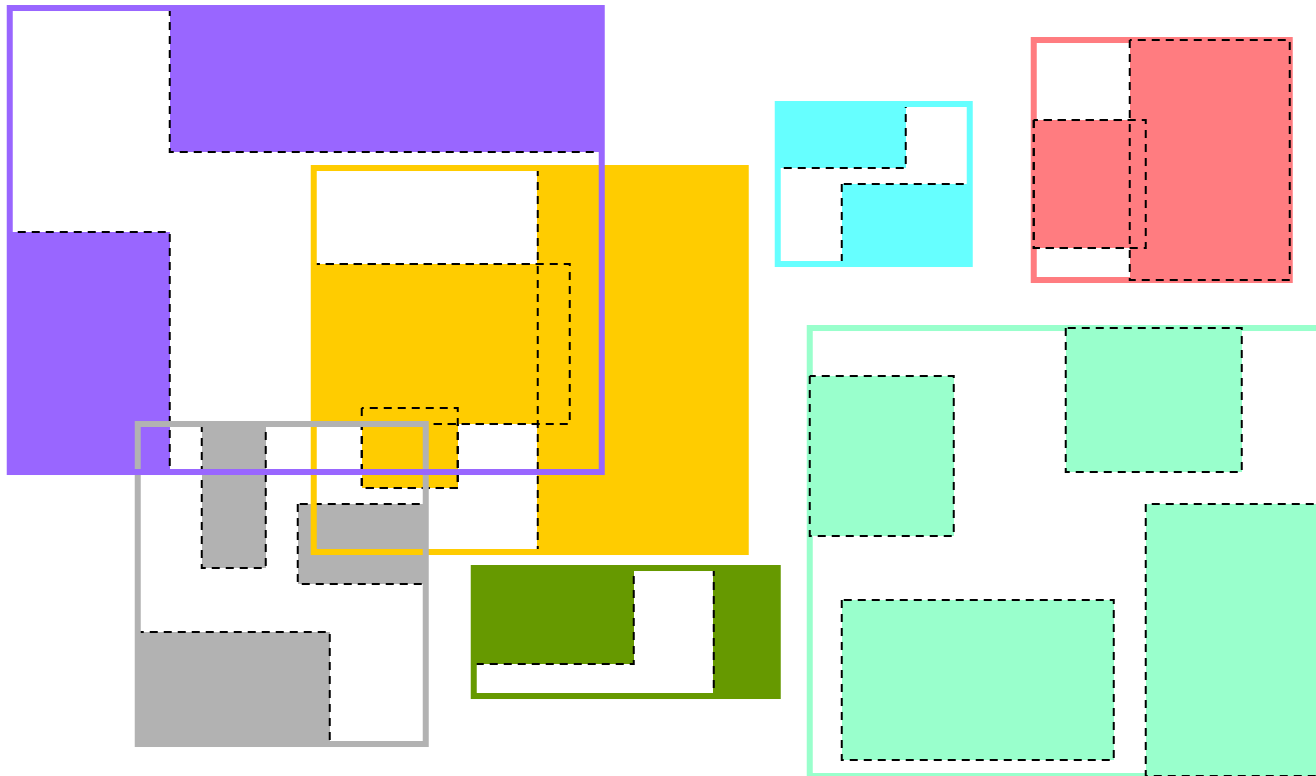
Example



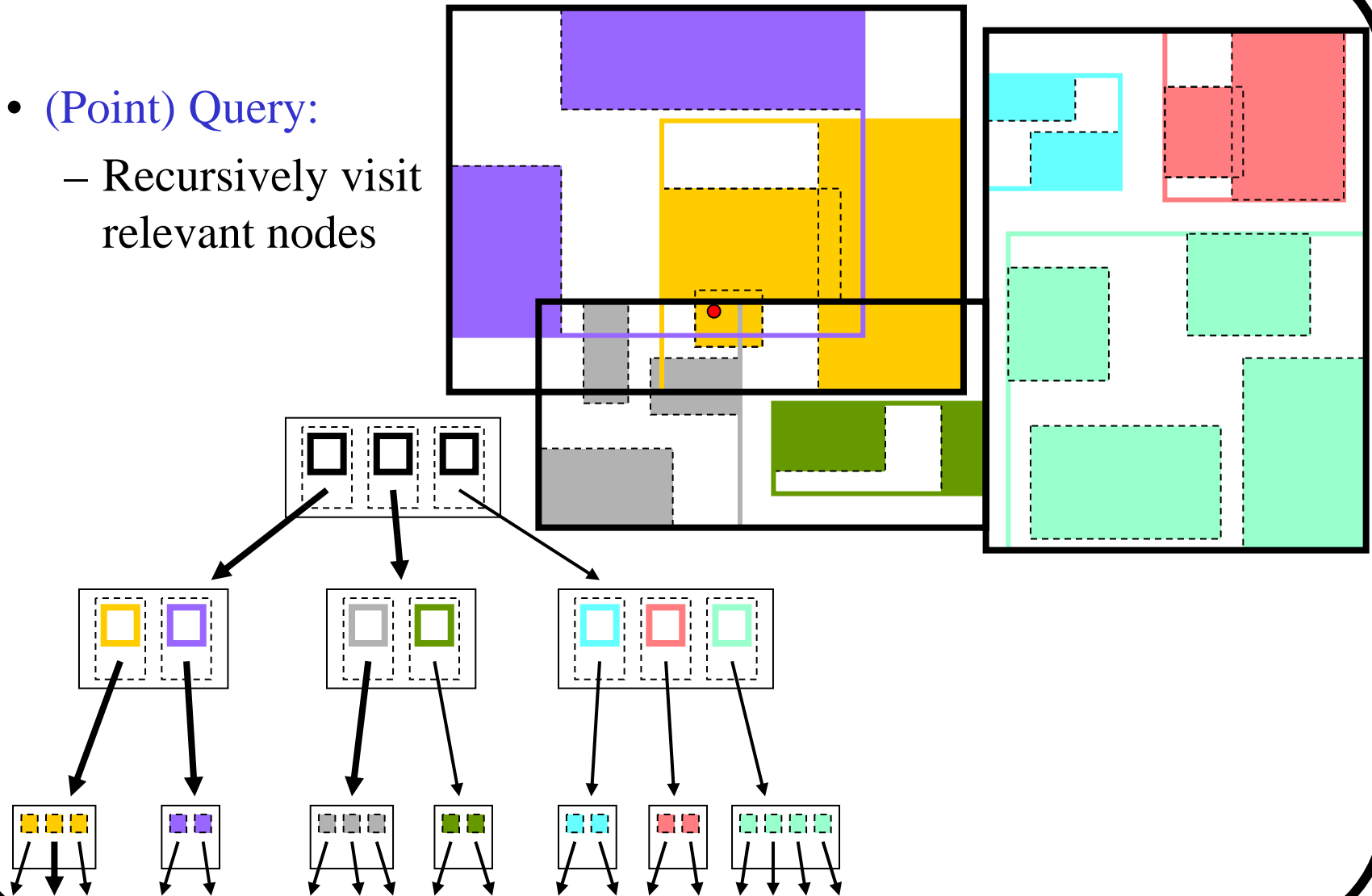
Example



Example

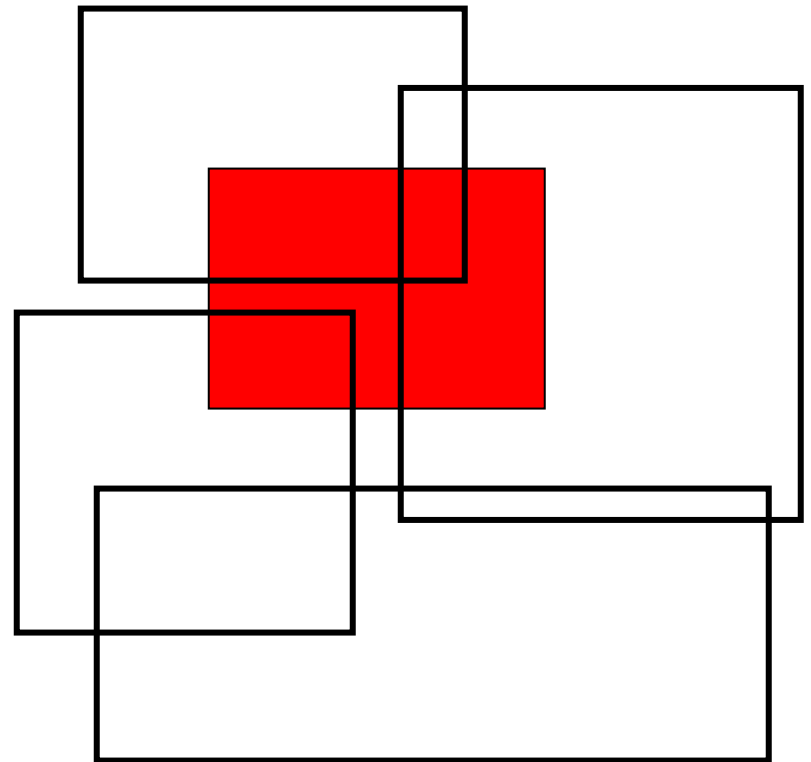
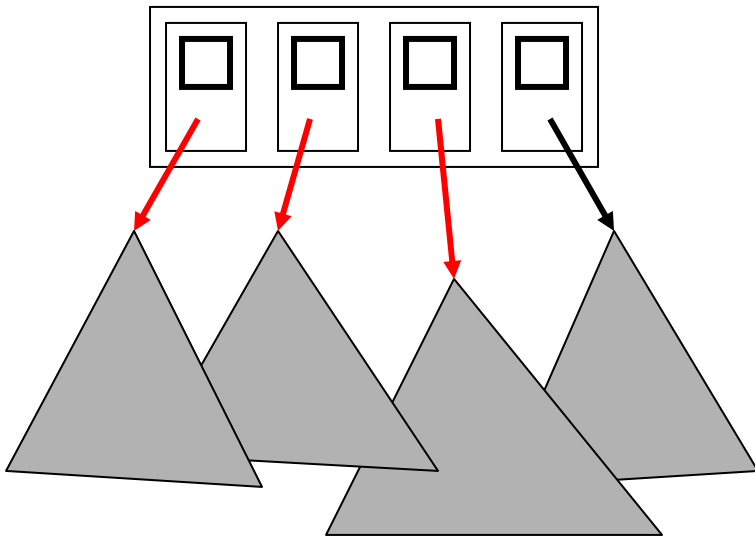


- (Point) Query:
 - Recursively visit relevant nodes



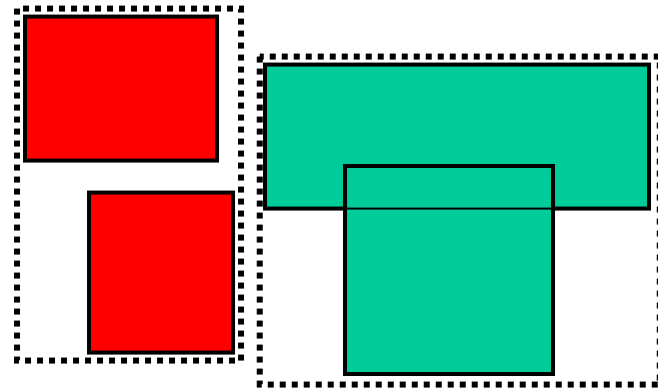
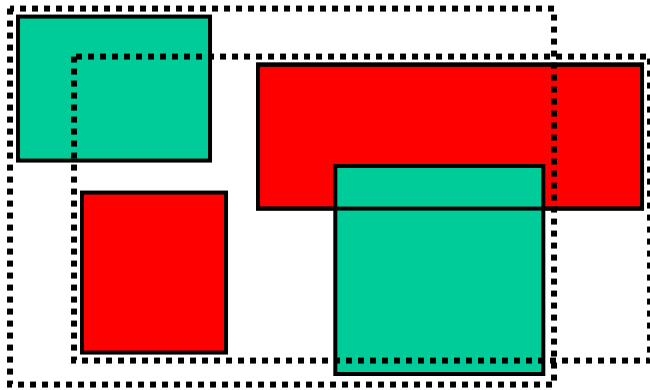
Query Efficiency

- The fewer rectangles intersected the better



Rectangle Order

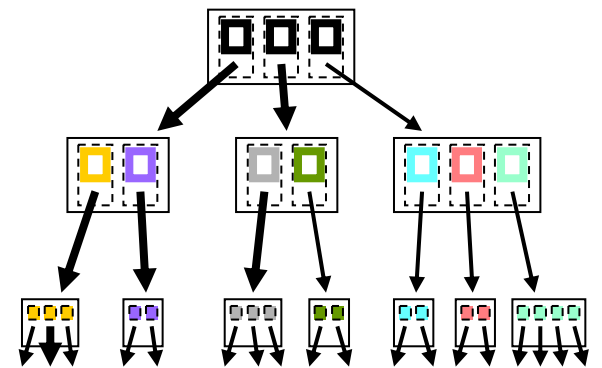
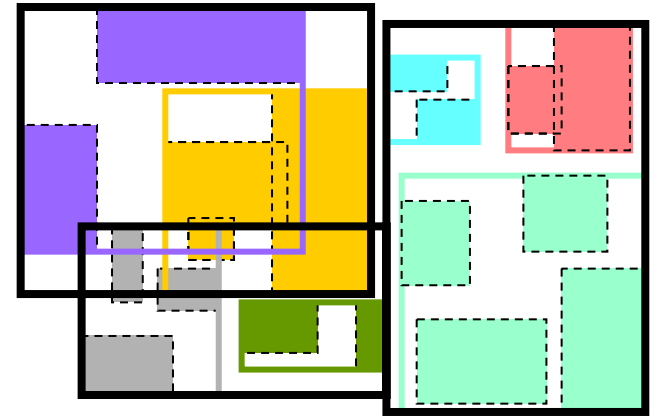
- Intuitively
 - Objects close together in same leaves
 - ⇒ small rectangles ⇒ queries descend in few subtrees



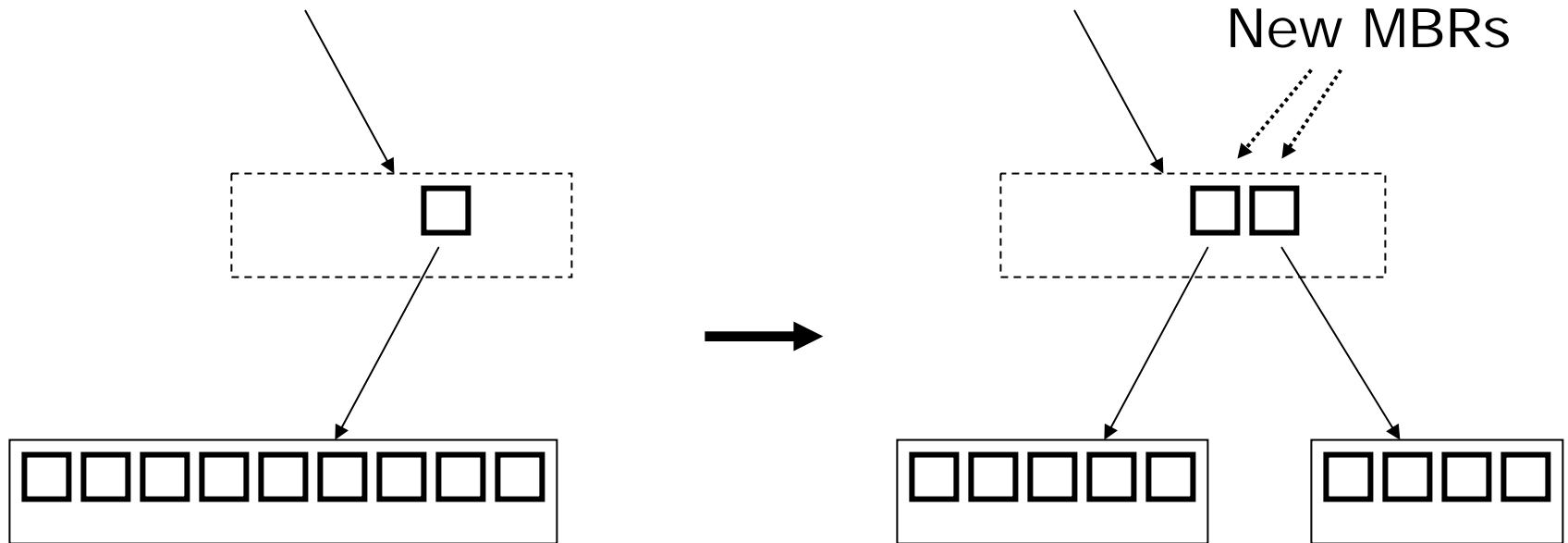
- Grouping in internal nodes?
 - Small area of MBRs
 - Small perimeter of MBRs
 - Little overlap among MBRs

R-tree Insertion Algorithm

- When not yet at a leaf (*choose subtree*):
 - Determine rectangle whose area increment after insertion is smallest (small area heuristic)
 - Increase this rectangle if necessary and recurse
- At a leaf:
 - Insert if room, otherwise *Split Node* (while trying to minimize area)

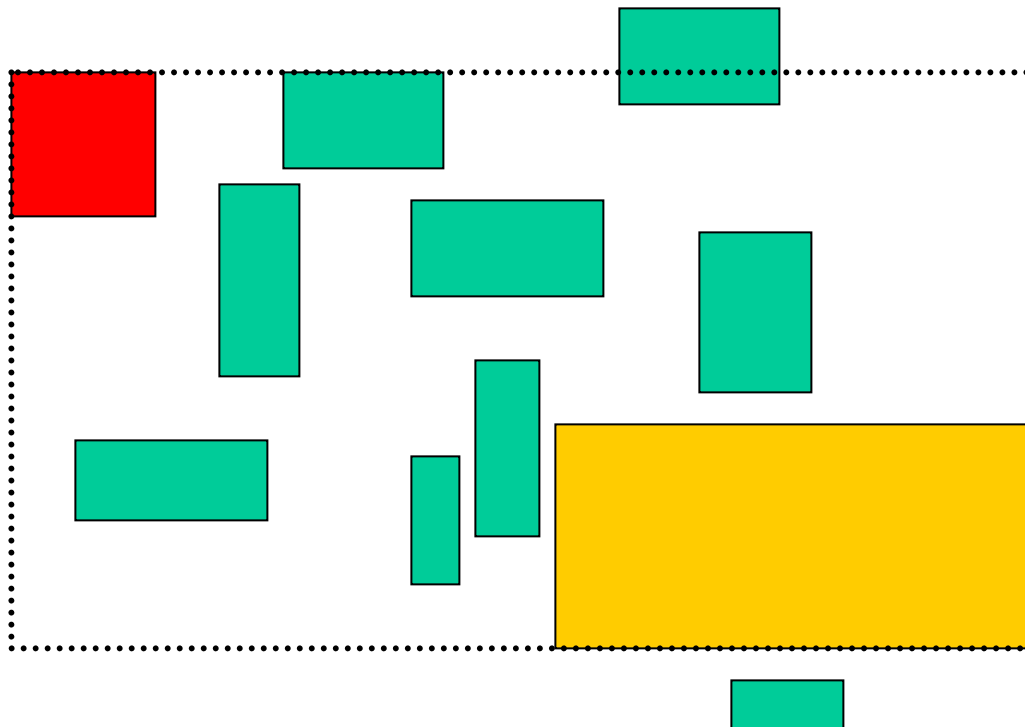


Node Split



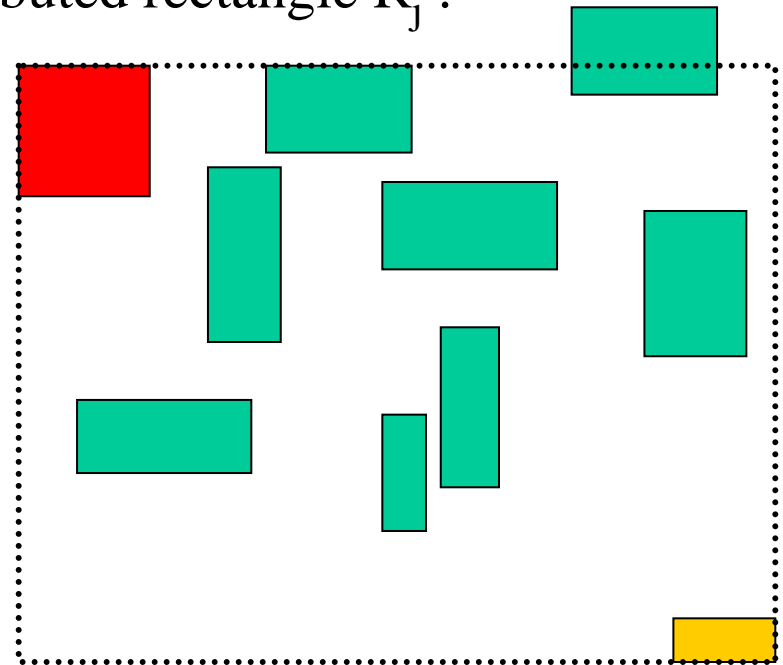
Linear Split Heuristic

- Determine R_1 and R_2 with largest MBR: the *seeds* for sets S_1 and S_2
- While not all MBRs distributed
 - Add next MBR to the set whose MBR increases the least



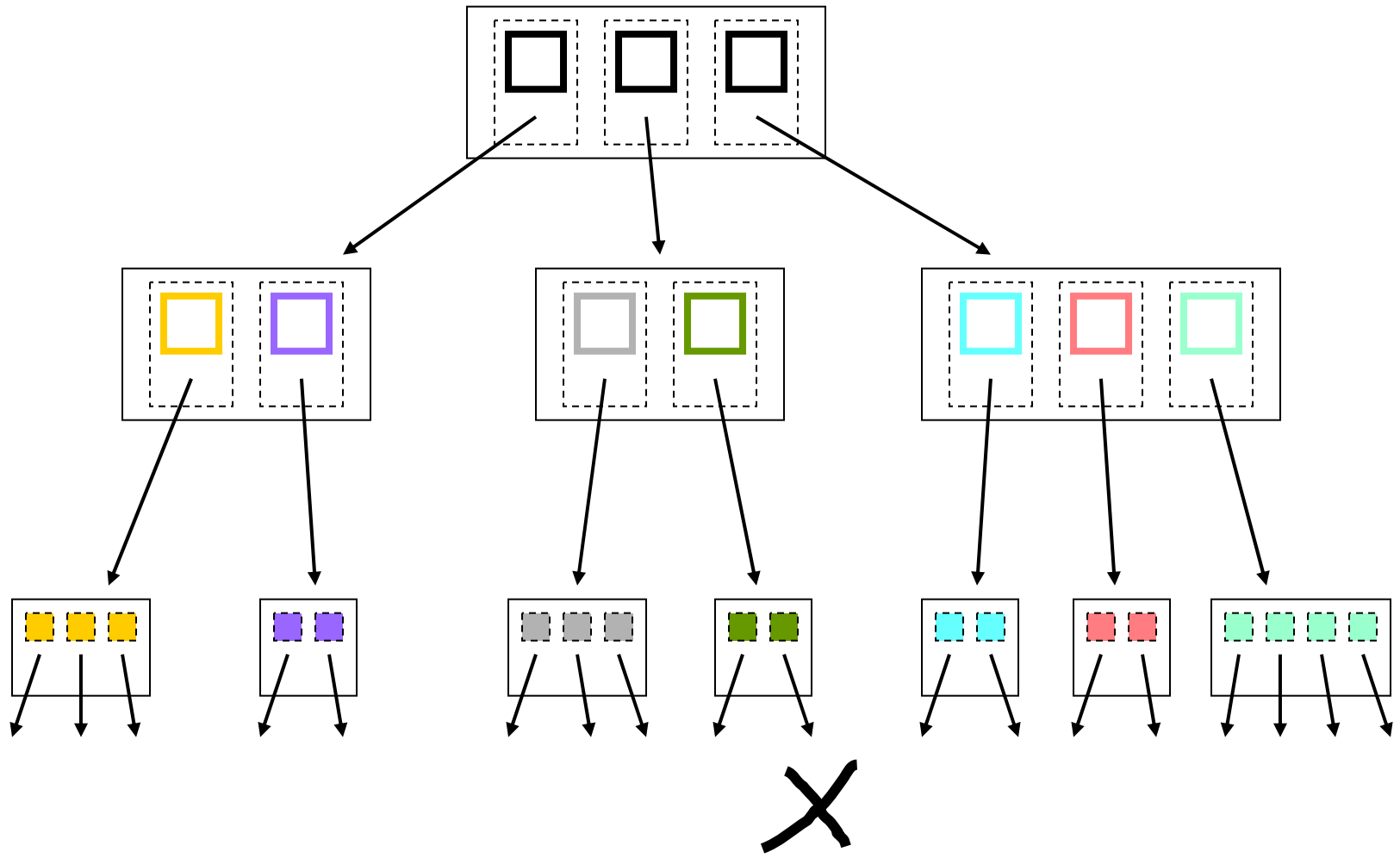
Quadratic Split Heuristic

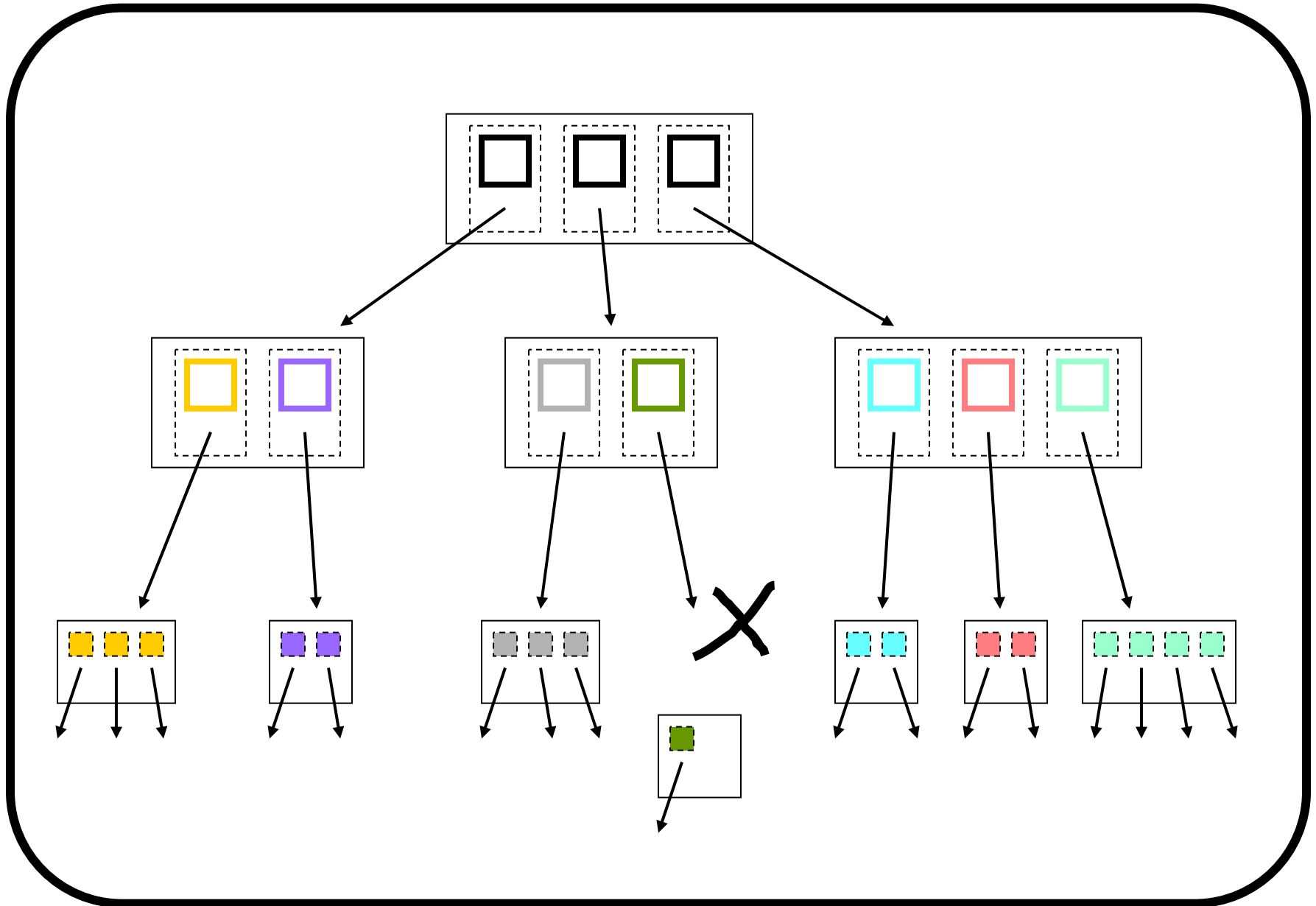
- Determine R_1 and R_2 with largest $area(MBR) - area(R_1) - area(R_2)$: the *seeds* for sets S_1 and S_2
- While not all MBRs distributed
 - Determine of every not yet distributed rectangle R_j :
 - $d_1 = \text{area increment of } S_1 \cup R_j$
 - $d_2 = \text{area increment of } S_2 \cup R_j$
 - Choose R_i with maximal $|d_1 - d_2|$ and add to the set with smallest area increment

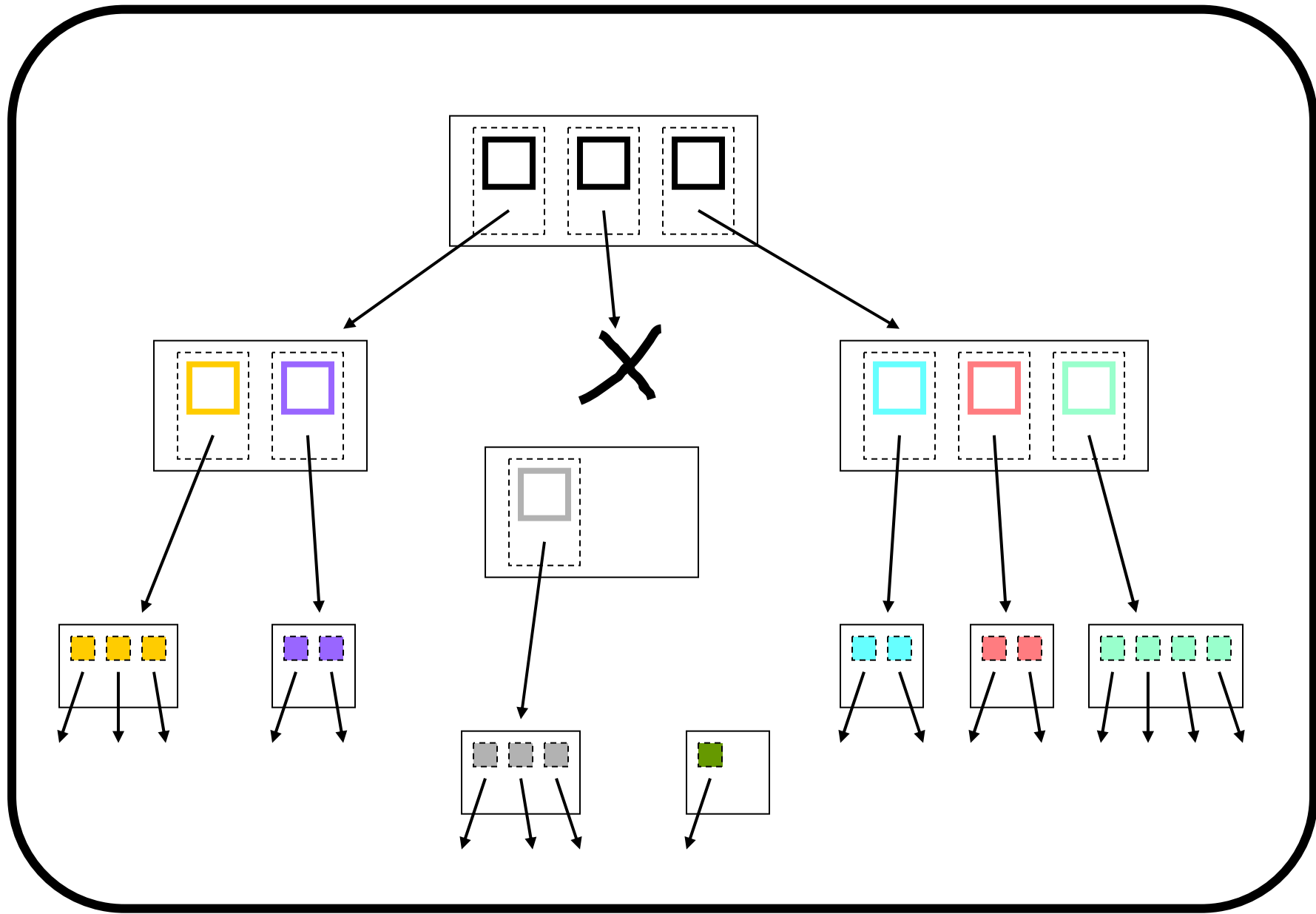


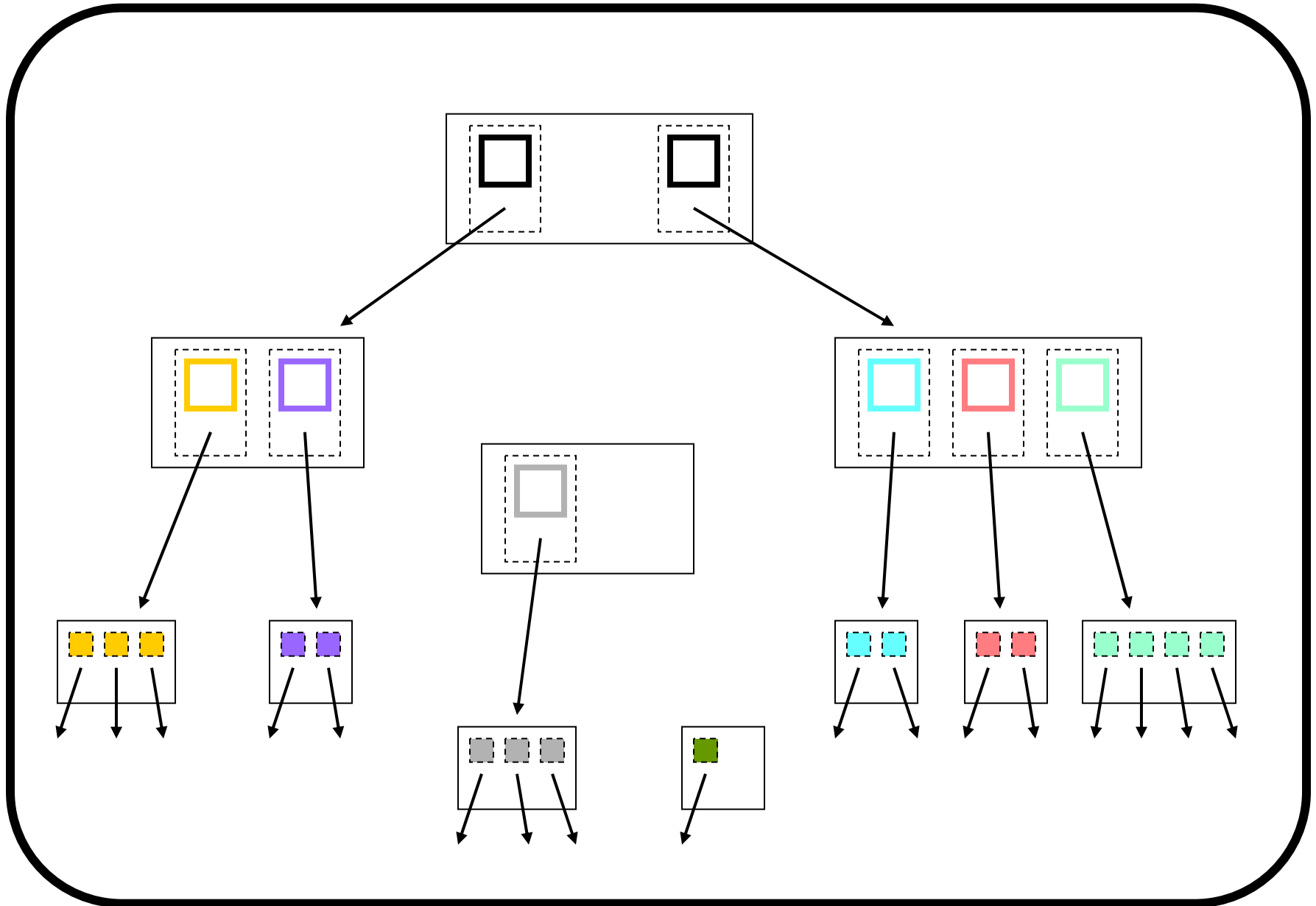
R-tree Deletion Algorithm

- Find the leaf (node) and delete object; determine new (possibly smaller) MBR
- If the node is too empty:
 - Delete the node recursively at its parent
 - Insert all entries of the deleted node into the R-tree
- Note: Insertions of entries/subtrees always occurs at the level where it came from

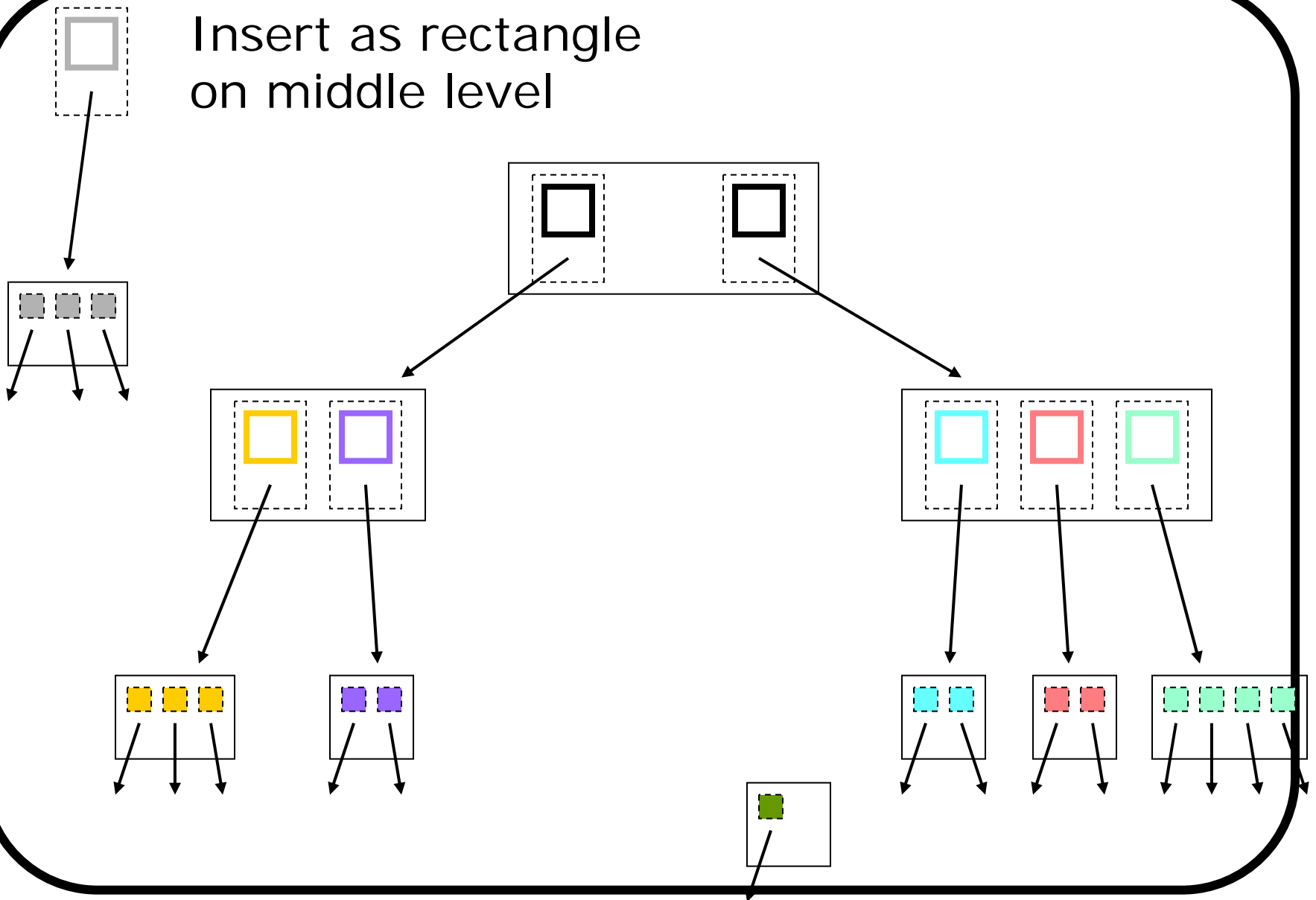


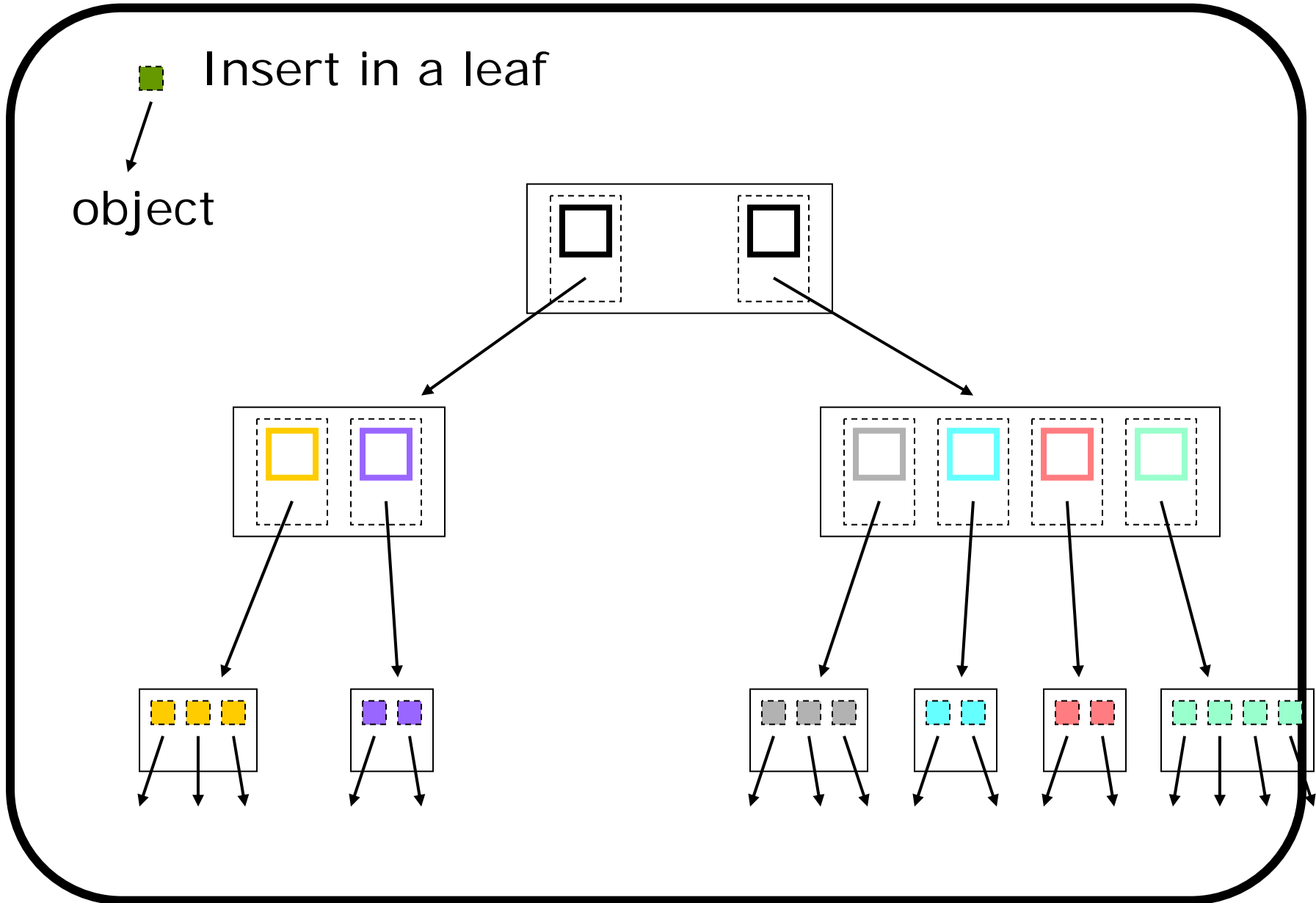






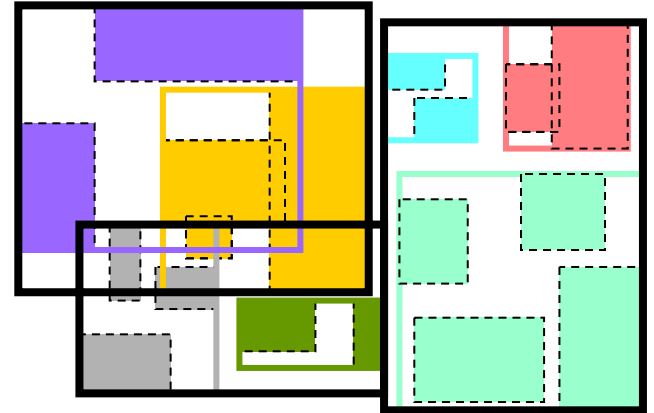
Insert as rectangle
on middle level



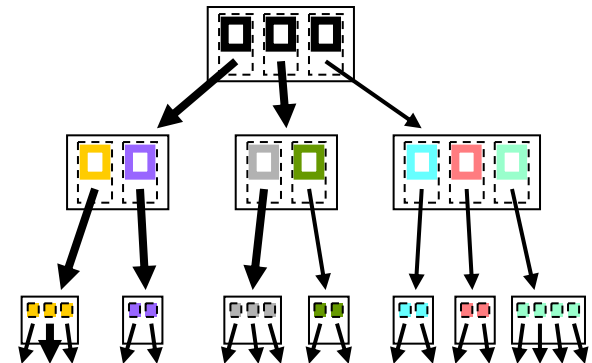


R*-trees

- Why try to minimize area?
 - Why not overlap, perimeter,...

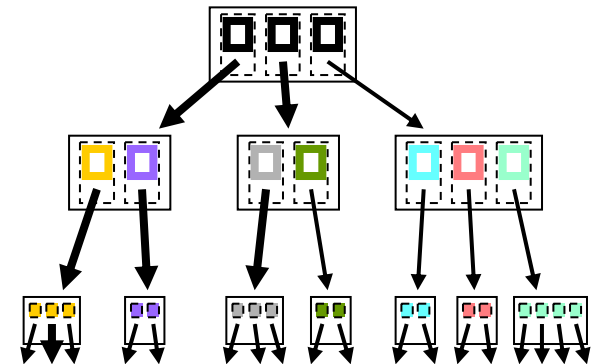
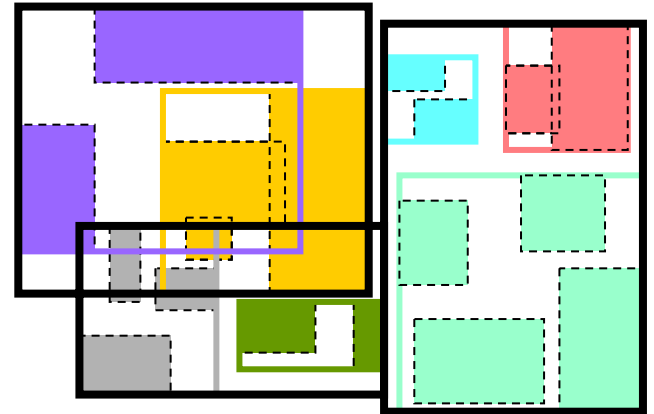


- R*-tree:
 - Experimentally determined algorithms for *Choose Subtree* and *Split Node*



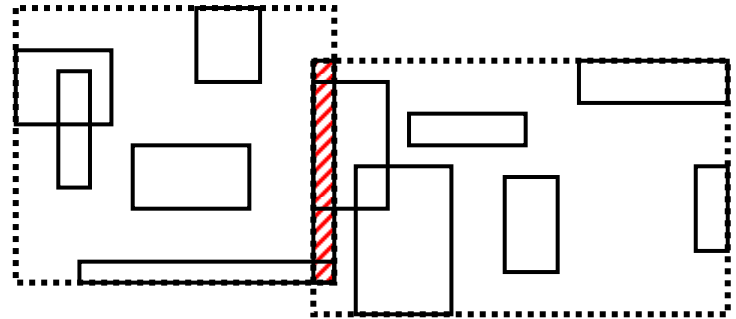
R*-trees; Choose Subtree

- At nodes directly above leaves:
 - Choose entry (rectangle) with smallest **overlap**-increase
- At higher nodes:
 - Choose entry (rectangle) with smallest **area**-increase



R*-trees; Split Node

- Determine split axis: For both the x - and the y -axis:
 - Sort the rectangles by smallest and largest coordinate
 - Determine the $M-2m+2$ allowed distributions into two groups
 - Determine for each the perimeter of the two MBRs
 - Add up all perimeters
- Choose the axis with smallest sum of perimeters
- Determine split index (given the split axis):
 - Choose the allowed distribution among the $M - 2m + 2$ with the smallest area of intersection of the MBRs



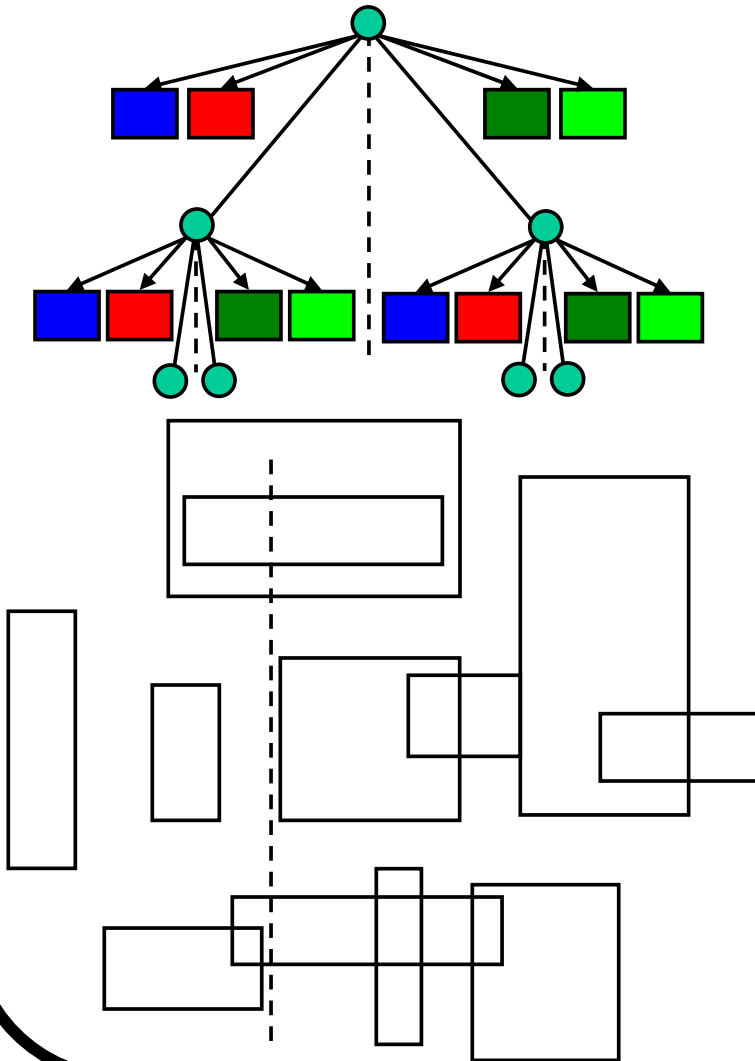
R*-trees; Forced reinsert

- Intuition:
 - When building R-tree by repeated insertion first inserted rectangles are possibly badly placed
- Experiment:
 - Make R-tree by inserting 20.000 rectangles
 - Delete the first inserted 10.000 and insert them again
- Search time improvement of 20-50%

R-Tree Variants

- Many, many R-tree variants (heuristics) have been proposed
- Often bulk-loaded R-trees are used (forced reinsertion intuition)
 - Much faster than repeated insertions
 - Better space utilization
 - Can optimize more “globally”
 - Can be updated using previous update algorithms
- Recently first worst-case efficient structure: PR-tree

Pseudo-PR-Tree



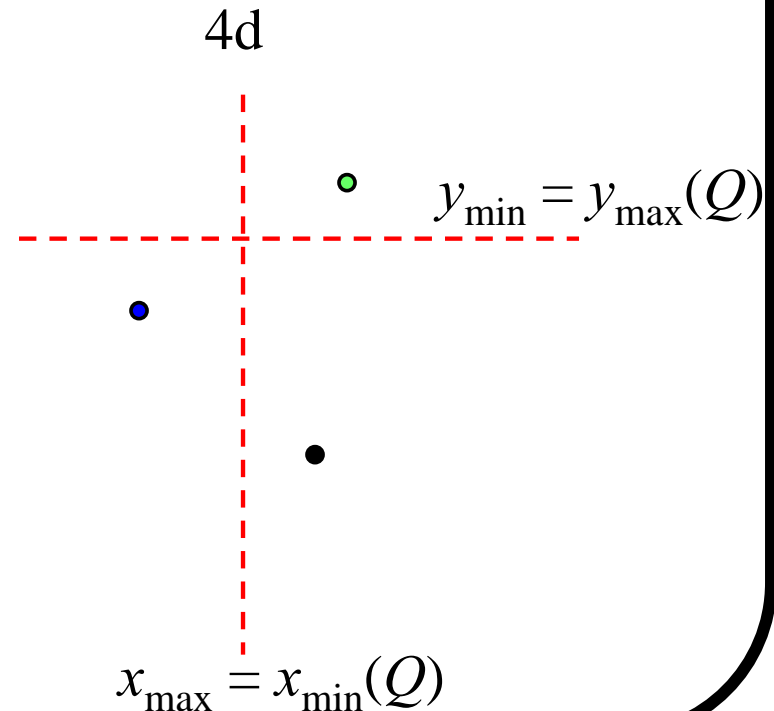
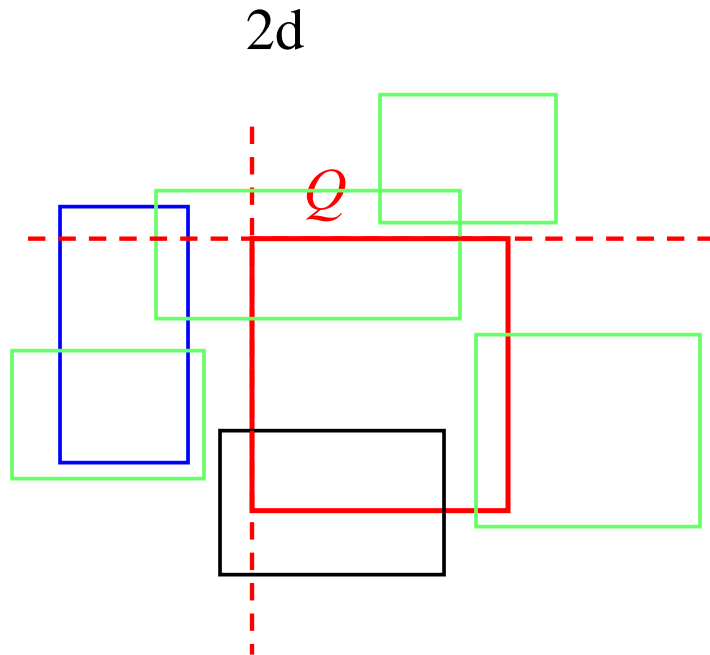
1. Place B extreme rectangles from each direction in **priority leaves**
2. Split remaining rectangles by x_{\min} coordinates (round-robin using x_{\min} , y_{\min} , x_{\max} , y_{\max} – like a 4d kd-tree)
3. Recursively build sub-trees

Query in $O(\sqrt{N/B} + T/B)$ I/Os

- $O(T/B)$ nodes with priority leaf completely reported
- $O(\sqrt{N/B})$ nodes with no priority leaf completely reported

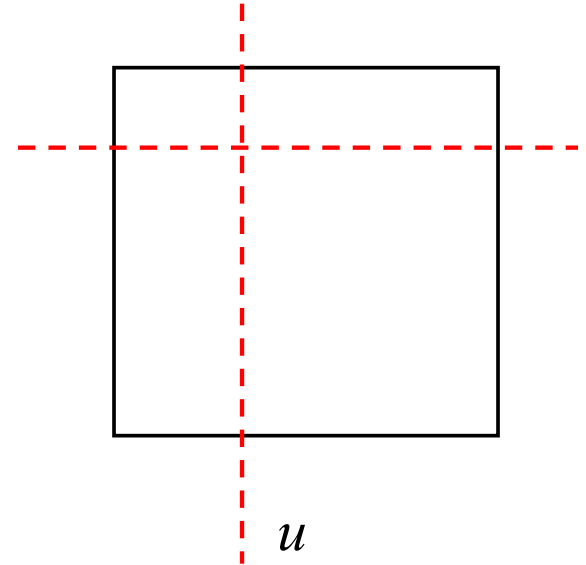
Pseudo-PR-Tree: Query Complexity

- Nodes v visited where all rectangles in at least one of the priority leaves of v 's parent are reported: $O(T/B)$
- Let v be a node visited but none of the priority leaves at its parent are reported completely, consider v 's parent u

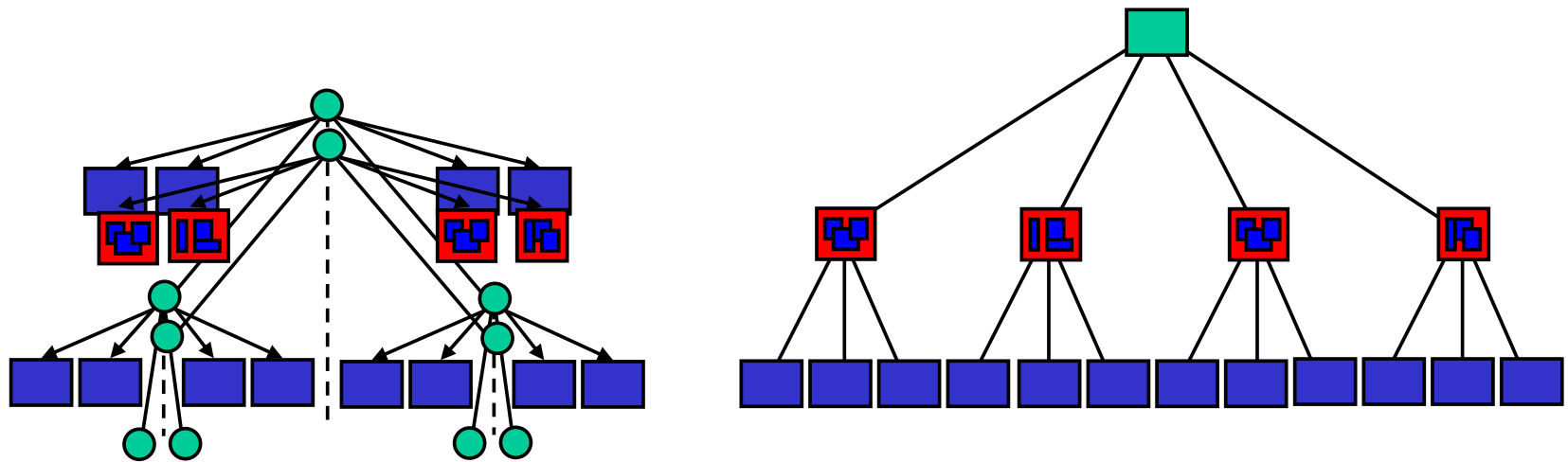


Pseudo-PR-Tree: Query Complexity

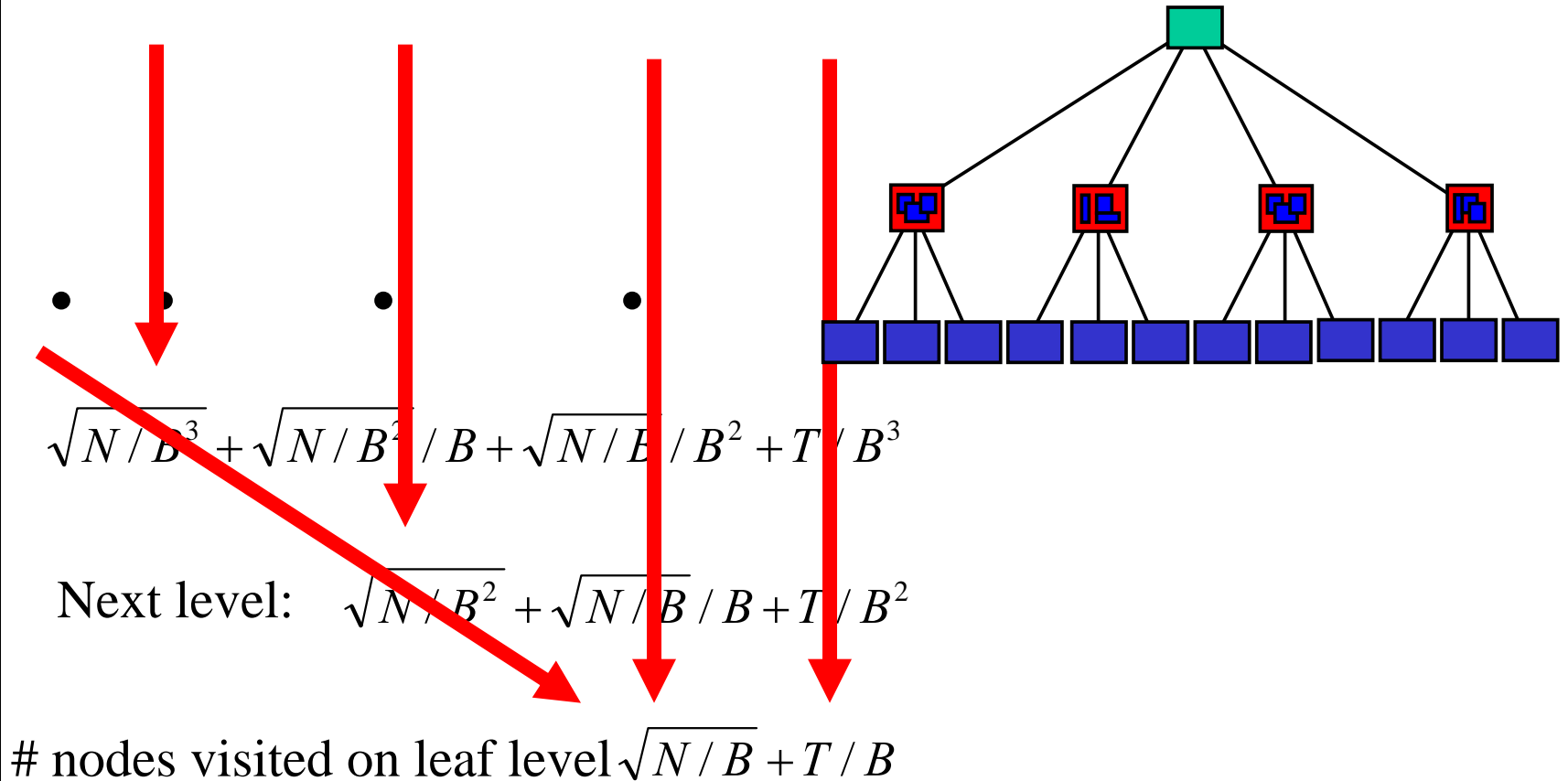
- The cell in the 4d kd-tree of u is intersected by two different 3-dimensional hyper-planes defined by sides of query Q
- The intersection of each pair of such 3-dimensional hyper-planes is a 2-dimensional hyper-plane
- **Lemma:** # of cells in a d -dimensional kd-tree that intersect an axis-parallel f -dimensional hyper-plane is $O((N/B)^{f/d})$
- So, # such cells in a 4d kd-tree: $O(\sqrt{N/B})$
- Total # nodes visited: $O(\sqrt{N/B} + T/B)$



PR-tree from Pseudo-PR-Tree



Query Complexity Remains Unchanged



PR-Tree

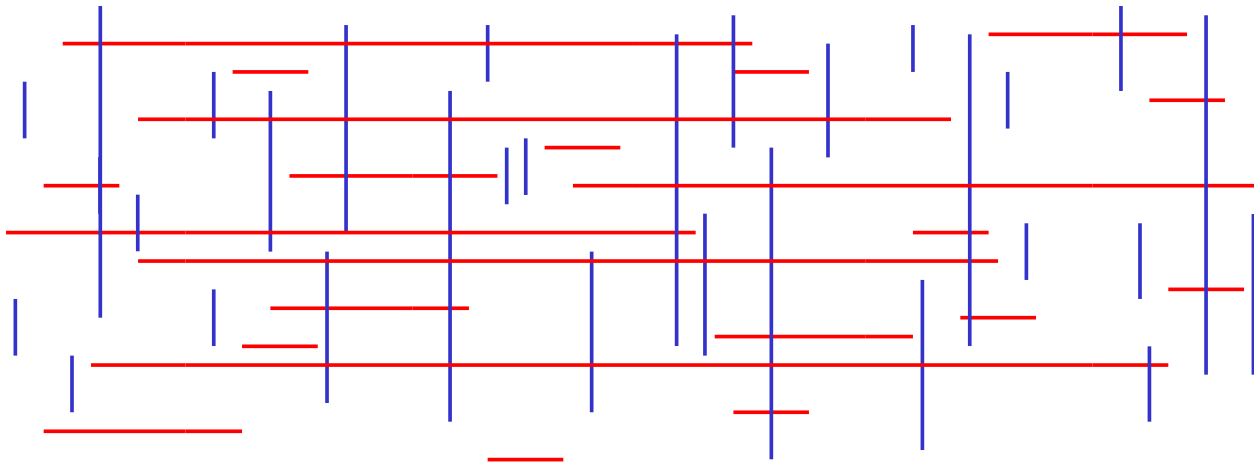
- PR-tree **construction** in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
 - Pseudo-PR-tree in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
 - Cost dominated by leaf level
- **Updates**
 - $O(\log_B N)$ I/Os using known heuristics
 - * Loss of worst-case query guarantee
 - $O(\log_B^2 N)$ I/Os using logarithmic method
 - * Worst-case query efficiency maintained
- Extending to **d -dimensions**
 - Optimal $O((N/B)^{1-1/d} + T/B)$ query

External Data Structure Implementations

- B-trees — LEDA-SM, TPIE
- Persistent B-tree [ADT01] — TPIE
- Buffer trees [HMSV97,BCMF99] — LEDA-SM
- Priority queues [BCMF99,ATV00] — LEDA-SM, TPIE
- Interval trees [CS97]
- Priority search tree [APSV01] — TPIE
- Range counting [GAA01] — TPIE
- Point location [ADT01] — TPIE
- R-trees [AHVV99] — TPIE
- kdB-trees [PAAV01] — TPIE
- Graphs — LEDA-SM
- Strings data structures [FG96,CF99] — LEDA-SM

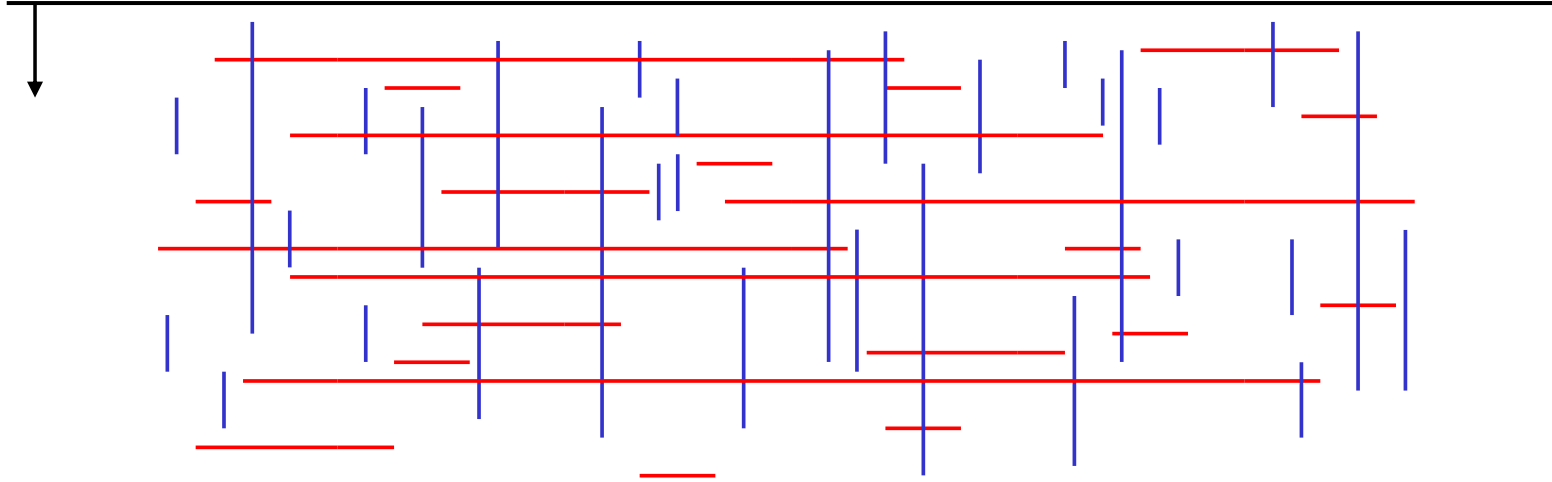
Geometric Algorithms

- We will now (quickly) look at geometric algorithms
 - Solves problem on set of objects
- Example: **Orthogonal line segment intersection**
 - Given set of axis-parallel line segments, report all intersections



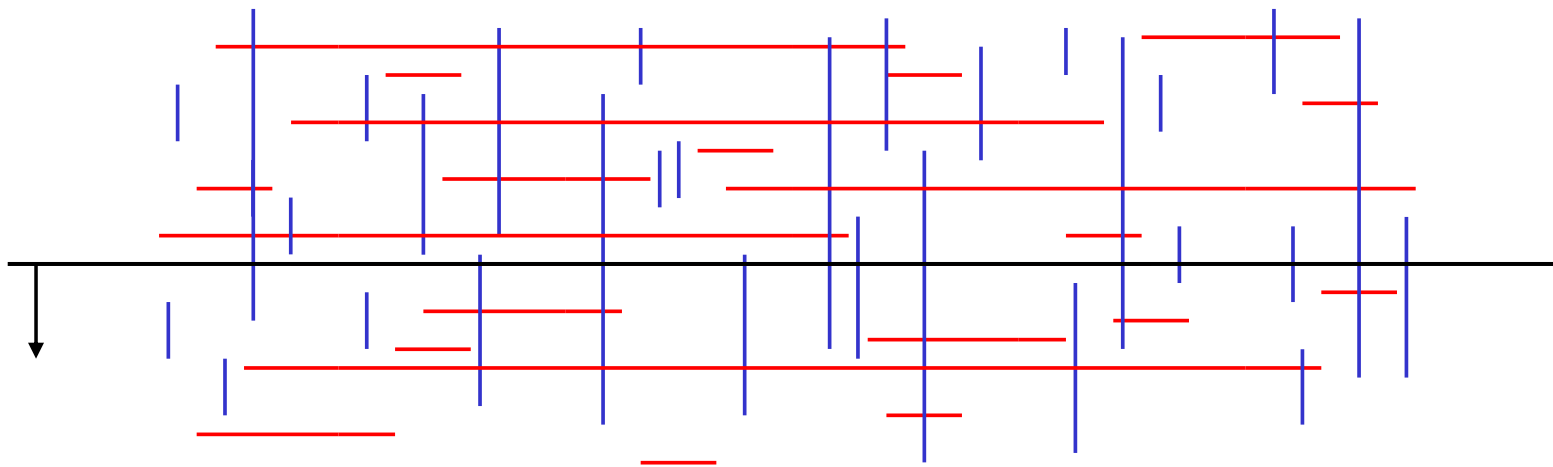
- In internal memory many problems is solved using sweeping

Plane Sweeping



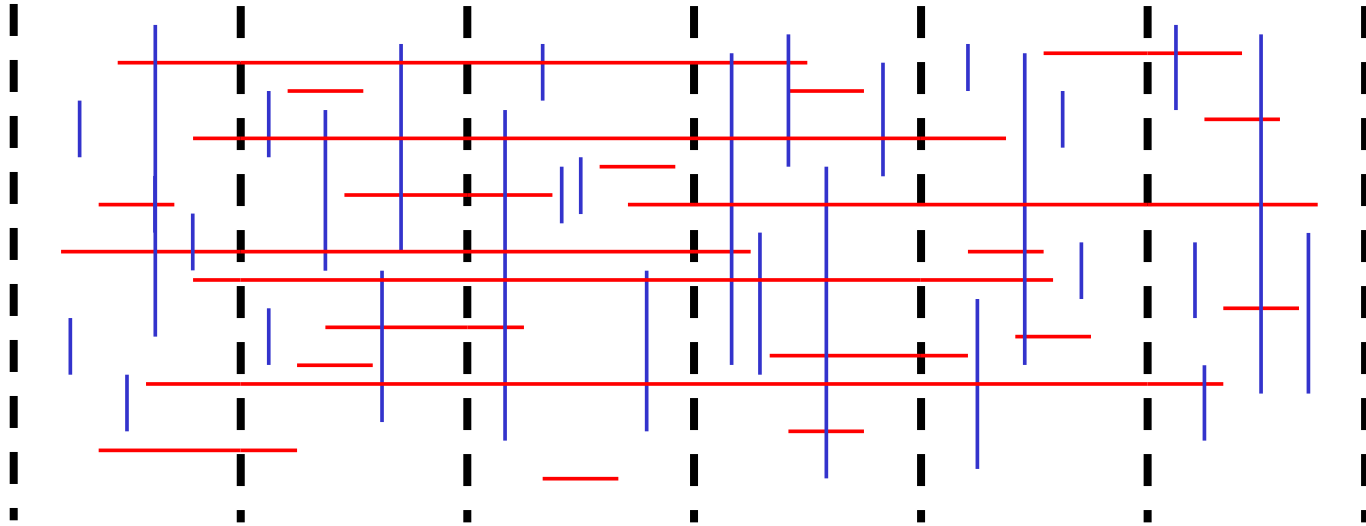
- Sweep plane top-down while maintaining search tree T on vertical segments crossing sweep line (by x -coordinates)
 - Top endpoint of vertical segment: Insert in T
 - Bottom endpoint of vertical segment: Delete from T
 - Horizontal segment: Perform range query with x -interval on T

Plane Sweeping



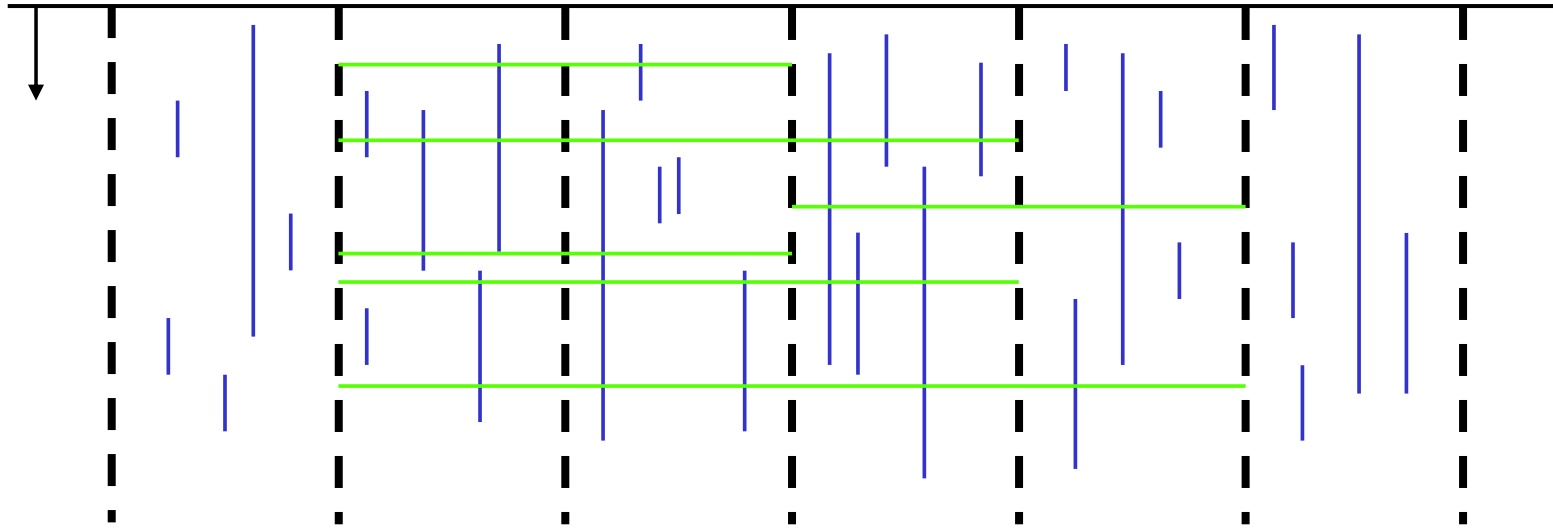
- In internal memory algorithm runs in optimal $O(M \log N + T)$ time
- In external memory algorithm performs badly ($>N$ I/Os) if $|T| > M$
 - Even if we implement T as B-tree $\Rightarrow O(M \log_B N + T/B)$ I/Os
- Solution: **Distribution sweeping**

Distribution Sweeping



- Divide plane into $M/B-1$ slabs with $O(N/(M/B))$ endpoints each
- **Sweep** plane top-down while reporting intersections between
 - part of horizontal segment spanning slab(s) and vertical segments
- **Distribute** data to $M/B-1$ slabs
 - vertical segments and non-spanning parts of horizontal segments
- Recurse in each slab

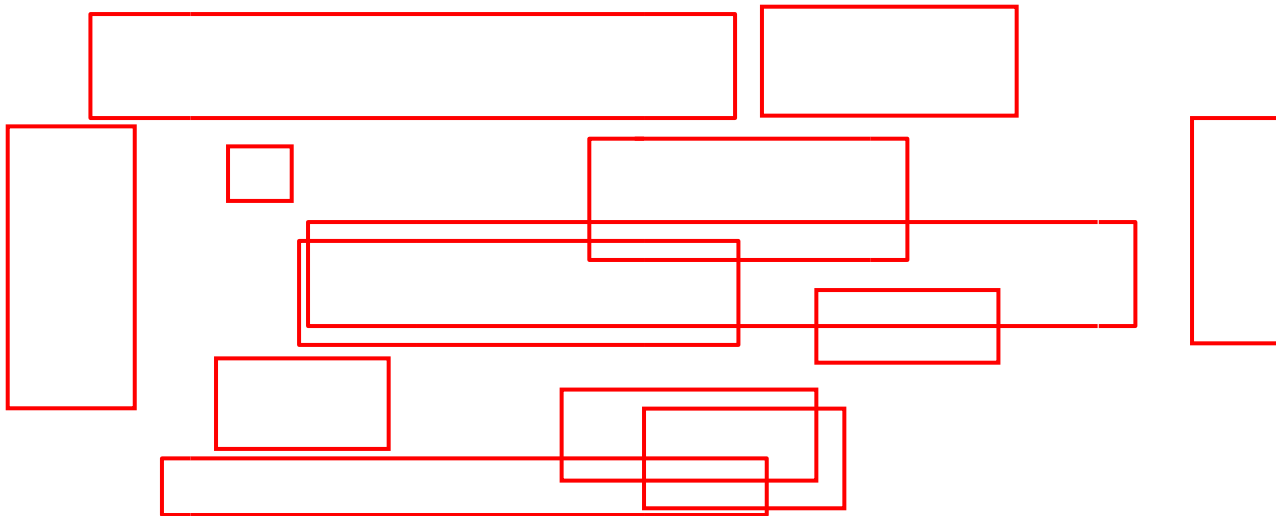
Distribution Sweeping



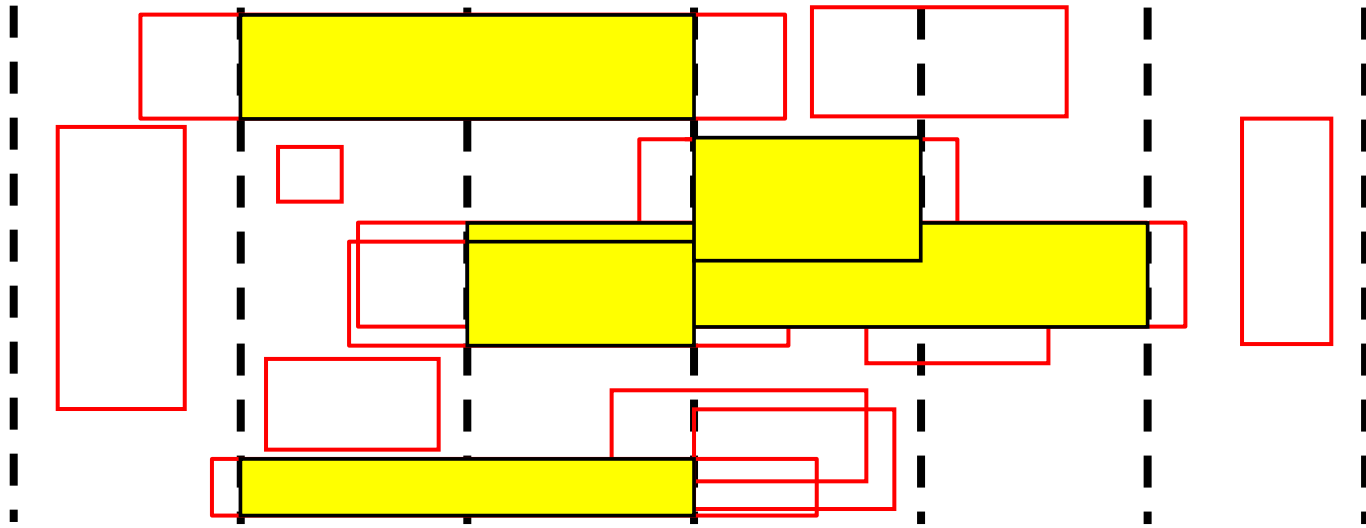
- Sweep performed in $O(N/B + T'/B)$ I/Os $\Rightarrow O(\frac{N}{B} \log_{M/B} \frac{N}{B} + \frac{T}{B})$ I/Os
- Maintain active list of vertical segments for each slab ($< B$ in memory)
 - **Top endpoint of vertical segment:** Insert in active list
 - **Horizontal segment:** Scan through all relevant active lists
 - * Removing “expired” vertical segments
 - * Reporting intersections with “non-expired” vertical segments

Distribution Sweeping

- Other example: **Rectangle intersection**
 - Given set of axis-parallel rectangles, report all intersections.

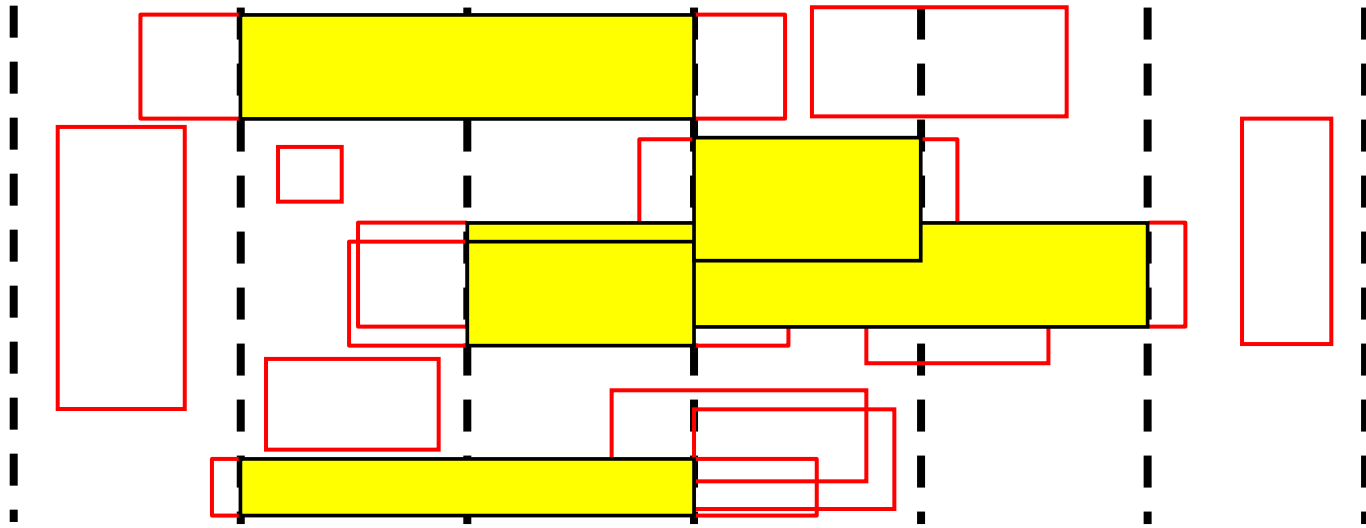


Distribution Sweeping



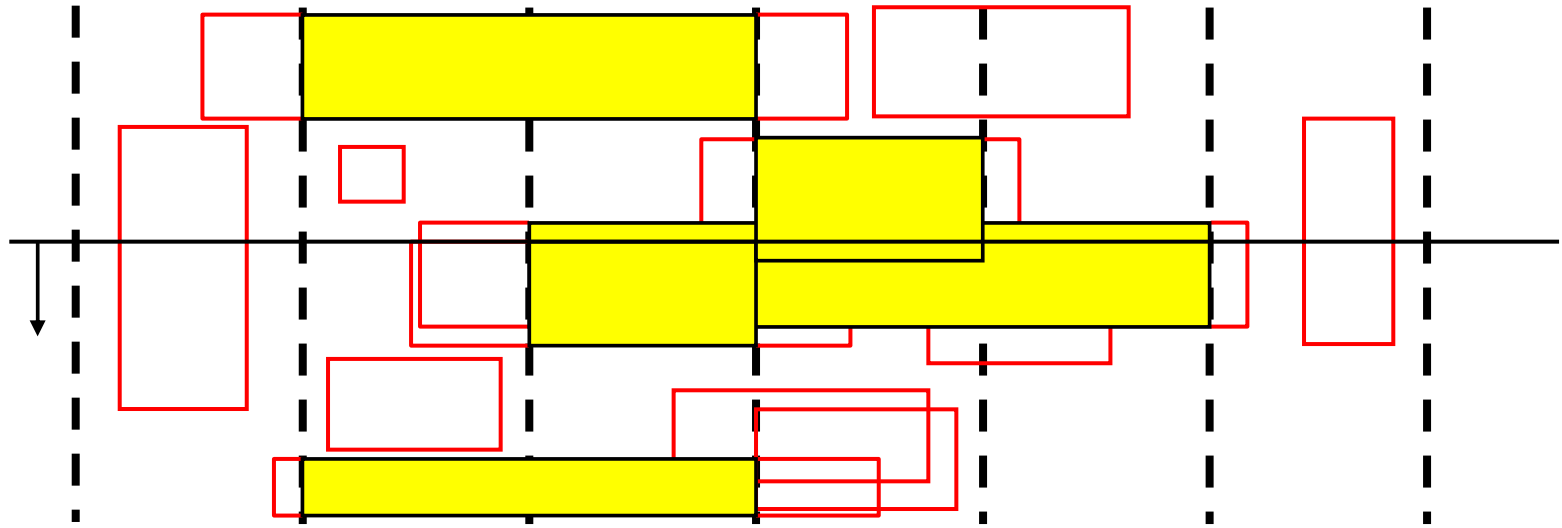
- Divide plane into $M/B-1$ slabs with $O(N/(M/B))$ endpoints each
- **Sweep** plane top-down while reporting intersections between
 - part of rectangles spanning slab(s) and other rectangles
- **Distribute** data to $M/B-1$ slabs
 - Non-spanning parts of rectangles
- Recurse in each slab

Distribution Sweeping



- Seems hard to perform sweep in $O(N/B + T'/B)$ I/Os
- Solution: **Multislabs**
 - Reduce fanout of distribution to $\Theta(\sqrt{M/B})$
 - Recursion height still $O(\log_{M/B} \frac{N}{B})$
 - Room for block from each multislabs (activlist) in memory

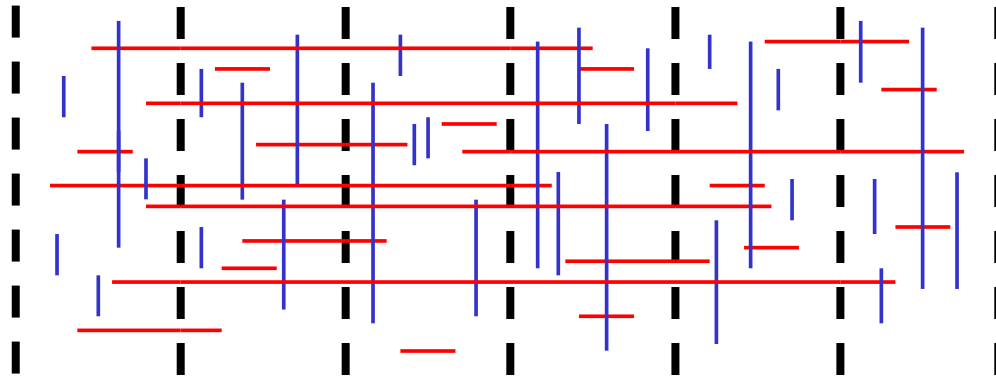
Distribution Sweeping



- Sweep while maintaining rectangle active list for each multislab
 - **Top side of spanning rectangle:** Insert in active multislab list
 - **Each rectangle:** Scan through all relevant multislab lists
 - * Removing “expired” rectangles
 - * Reporting intersections with “non-expired” rectangles
- $\Rightarrow O\left(\frac{N}{B} \log_{M/B} \frac{N}{B} + \frac{T}{B}\right)$ I/Os

Distribution Sweeping

- Distribution sweeping can relatively easily be used to solve a number of other problems in the plane I/O-efficiently
- By decreasing distribution fanout to $O((\frac{M}{B})^{1/c})$ for $c \geq 1$ a number of higher-dimensional problems can also be solved I/O-efficiently



Other Results

- Other geometric algorithms results include:
 - Red blue line segment intersection
(using distribution sweep, buffer trees/*batched filtering*, external fractional cascading)
 - General planar line segment intersection
(as above and external priority queue)
 - 2d and 3d Convex hull:
(Complicated deterministic 3d and simpler randomized)
 - 2d Delaunay triangulation

References

- **The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree.** L. Arge, M. de Berg, H. Haverkort, and K. Yi. Proc. SIGMOD'04.
 - Section 1-2
- **External-Memory Computational Geometry.** M.T. Goodrich, J-J. Tsay, D.E. Vengroff, and J.S. Vitter. Proc. *FOCS'93*.
 - Section 2.0-2.1