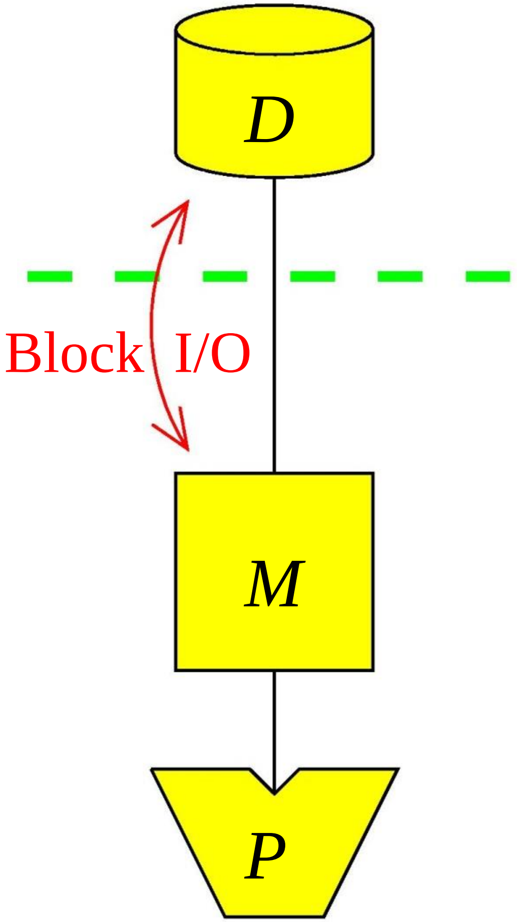
I/O-Algorithms

Lars Arge

Fall 2015

September 29, 2015

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

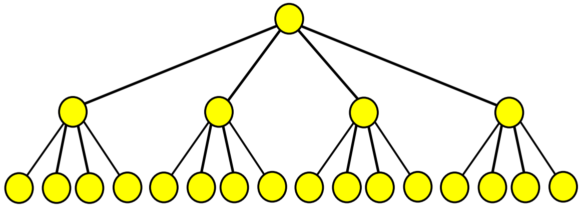
	Internal	External
• Scanning:	N	$\frac{N}{B}$
• Sorting:	$N \log N$	$\frac{N}{B} \log_{M/B} \frac{N}{B}$
• Permuting	N	$\min\{N, \frac{N}{B} \log_{M/B} \frac{N}{B}\}$
• Searching:	$\log_2 N$	$\log_B N$

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-tress: Weight rather than degree constraint

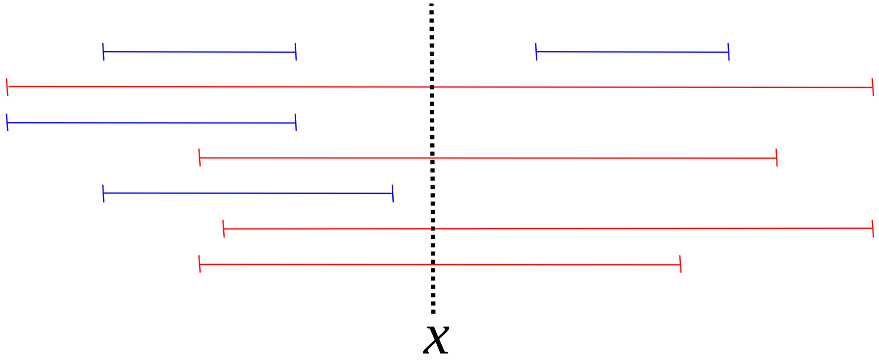
  $\Rightarrow \Omega(w(v))$ updates below v between rebalancing operations on v
- Persistent B-trees:
  - Update in current version in $O(\log_B N)$
  - Search in all previous versions in $O(\log_B N + T/_B)$
- Buffer trees
  - Batching of operations to obtain $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ bounds

  $\Rightarrow O(\frac{N}{B}\log_{M/_B}\frac{N}{B})$ construction algorithms



Interval Management

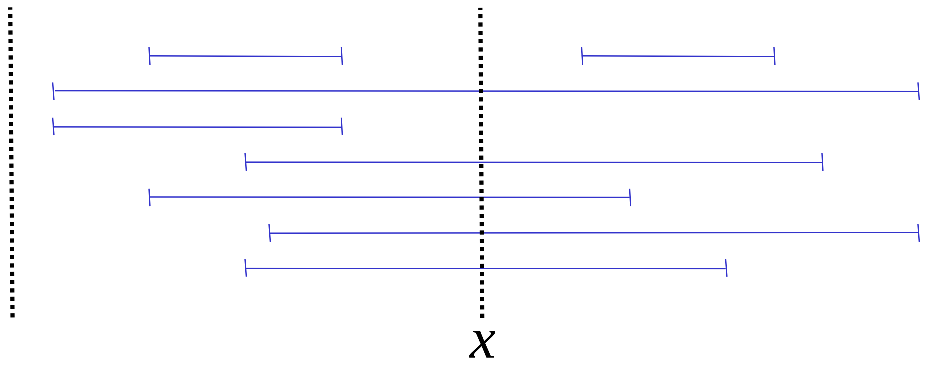
- Problem:
  - Maintain N intervals with unique endpoints dynamically such that stabbing query with point x can be answered efficiently



- As in (one-dimensional) B-tree case we are interested in
  - $O(N/B)$ space
  - $O(\log_B N)$ update
  - $O(\log_B N + T/B)$ query

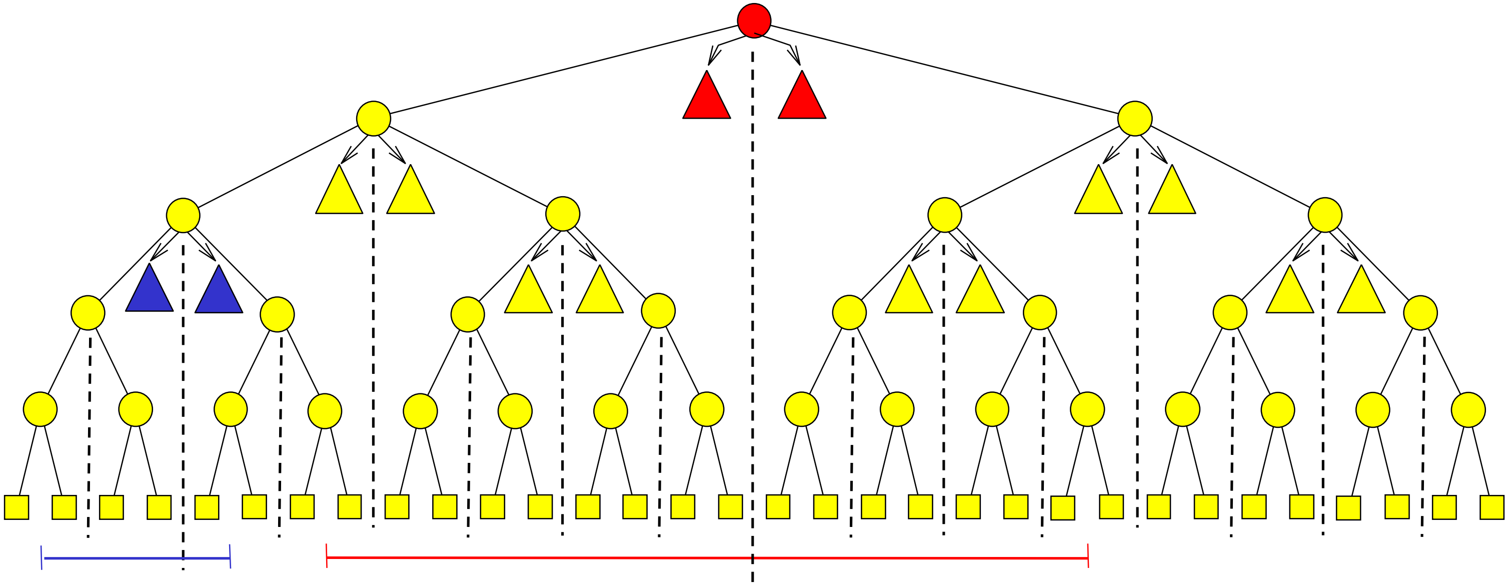
Interval Management: Static Solution

- Sweep from left to right maintaining persistent B-tree
  - Insert interval when left endpoint is reached
  - Delete interval when right endpoint is reached



- Query x answered by reporting all intervals in B-tree at “time” x
  - $O(N/B)$ space
  - $O(\log_B N + T/B)$ query
  - $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ construction using buffer technique

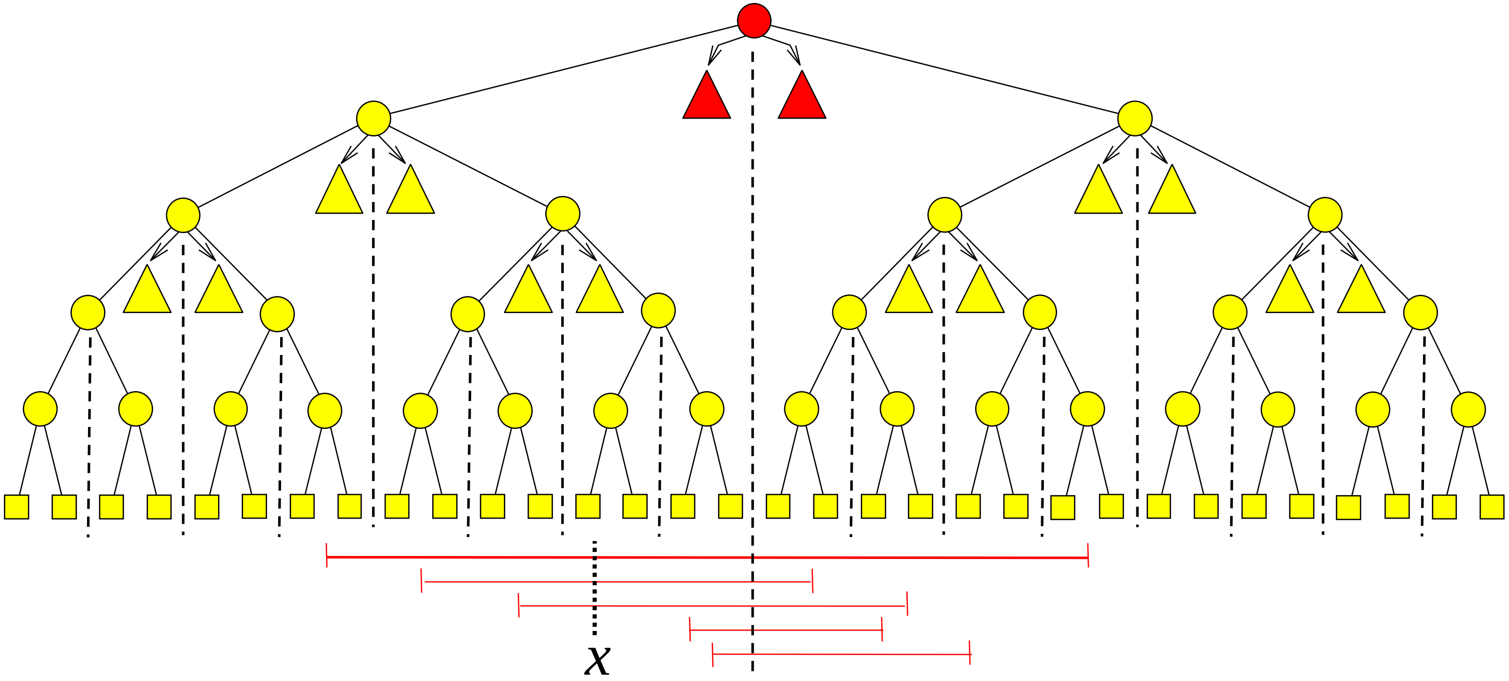
Internal Interval Tree



- 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 (assuming fixed endpoint set)

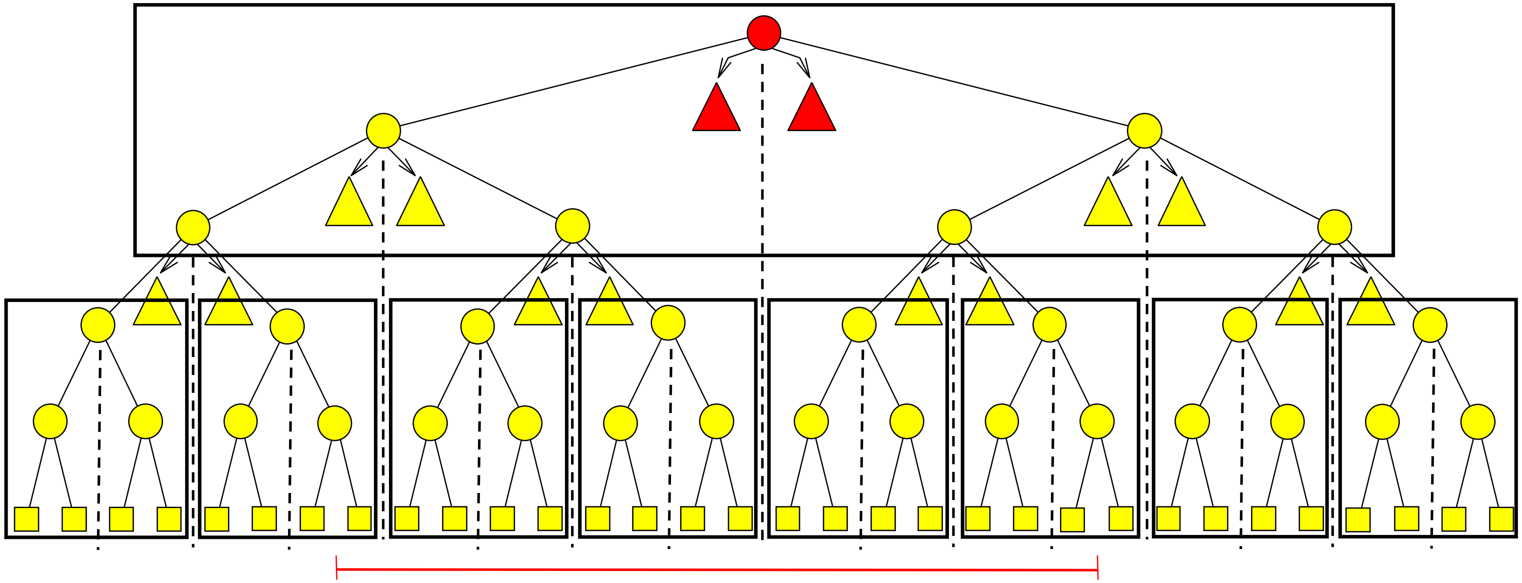
Internal Interval Tree



- 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

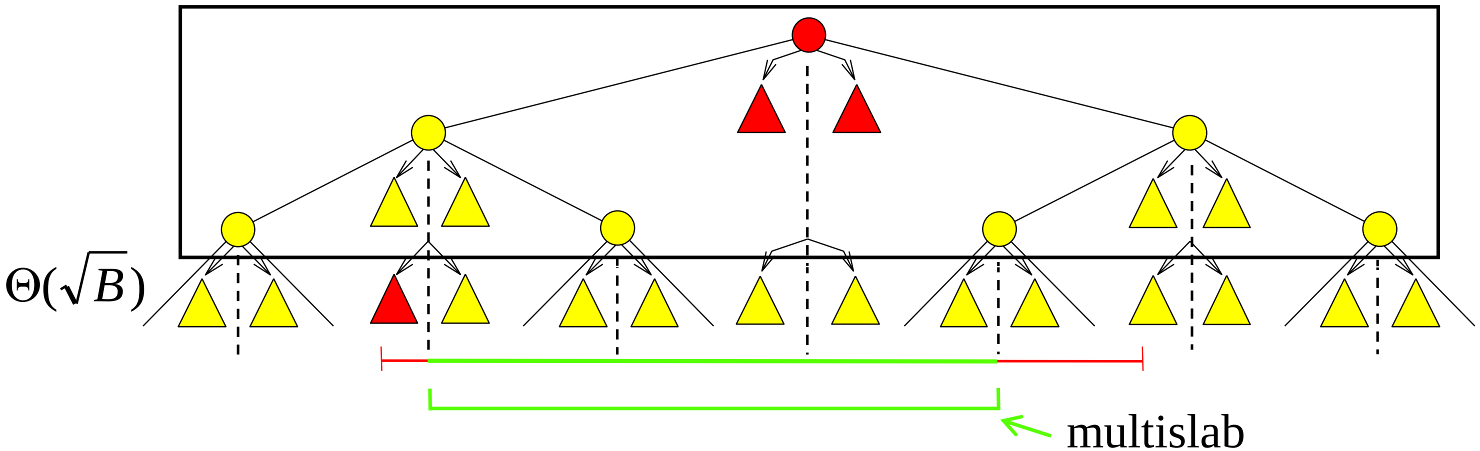
⇒ $O(\log N+T)$ query bound

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

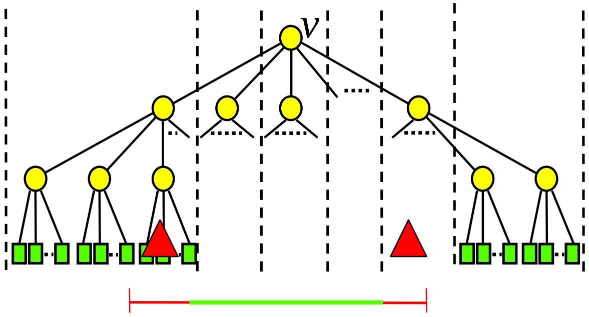
Externalizing Interval Tree



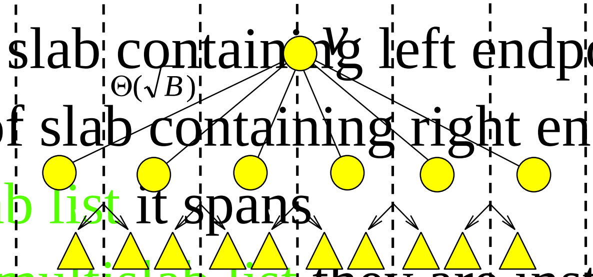
- Idea:
  - Decrease fan-out to $\Theta(\sqrt{B})$ $\Rightarrow$ height remains $O(\log_B N)$
  - $\Theta(\sqrt{B})$ slabs define $\Theta(B)$ multislabs
  - Interval stored in two slab lists (as before) and one multislab list
  - Intervals in small multislab lists collected in underflow structure
  -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

External Interval Tree

- Base tree: Weight-balanced B-tree with branching parameter $\frac{1}{4}\sqrt{B}$ and leaf parameter B on endpoints
  - Interval stored in highest node v where it contains slab boundary
- Each internal node v contains:
  - Left slab list for each of $\Theta(\sqrt{B})$ slabs
  - Right slab lists for each of $\Theta(\sqrt{B})$ slabs
  - $\Theta(B)$ multislab lists
  - Underflow structure



- Interval in set I_v of intervals associated with v stored in
  - Left slab list of slab containing left endpoint
  - Right slab list of slab containing right endpoint
  - Widest multislab list it spans
- If < B intervals in multislab list they are instead stored in underflow structure (⇒ contains ≤ B^2 intervals)

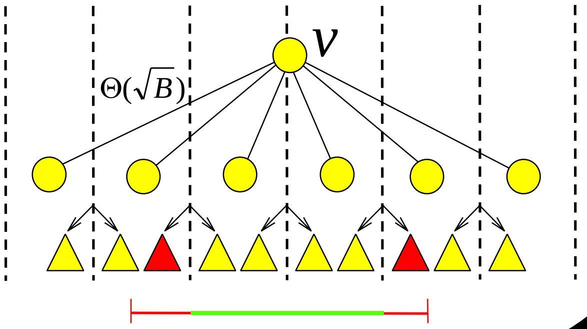


External Interval tree

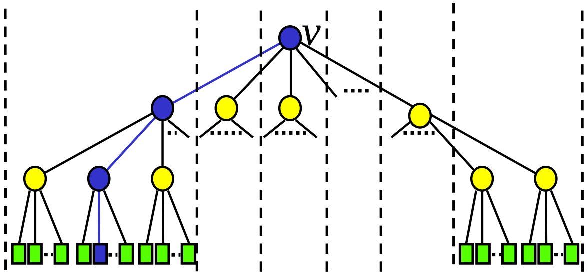
- Each leaf contains <*B/2* intervals (unique endpoint assumption)
  - Stored in one block
- Slab lists implemented using B-trees
  - $O(1+T_v/B)$ query
  - Linear space
    - We may "wasted" a block for each of the $\Theta(\sqrt{B})$ lists in node
    - But only $\Theta(\frac{N}{B\sqrt{B}})$ internal nodes
- Underflow structure implemented using static structure
  - $O(\log_B B^2 + T_v/B) = O(1+T_v/B)$ query
  - Linear space

$\Downarrow$

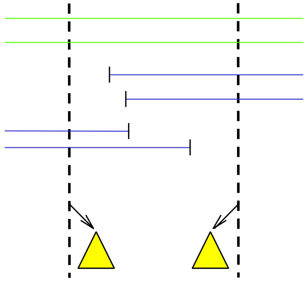
- Linear space



External Interval Tree



- Query with x
  - Search down tree for x while in node v reporting all intervals in I_v stabbed by x
- In node v
  - Query two slab lists
  - Report all intervals in relevant multislab lists
  - Query underflow structure



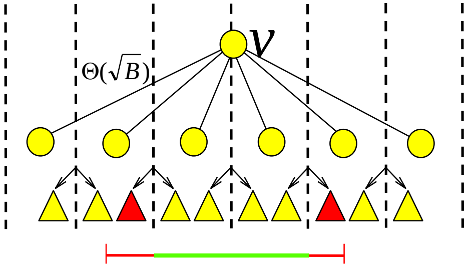
- Analysis:
  - Visit $O(\log_B N)$ nodes
  - Query slab lists
  - Query multislab lists
  - Query underflow structure

$$\left.\begin{array}{l}\text{Visit } O(\log_B N) \text{ nodes}\\ \left.\begin{array}{l}\text{Query slab lists}\\ \text{Query multislab lists}\\ \text{Query underflow structure}\end{array}\right\} O(1+{}^{T_v}\!/\!_B)\end{array}\right\} \Rightarrow O(\log_B N + {}^{T}\!/\!_B)$$

External Interval Tree

- Update – ignoring base tree update/rebalancing:
  - Search for relevant node
  - Update two slab lists
  - Update multislab list or underflow structure

$\left.\begin{array}{l}\text{Search for relevant node}\\ \text{Update two slab lists}\end{array}\right\} O(\log_B N)$



- Update of underflow structure in $O(1)$ I/Os amortized:
  - Maintain update block with $\leq B$ updates
  - Check of update block adds $O(1)$ I/Os to query bound
  - Rebuild structure when B updates have been collected using $O(\frac{B^2}{B}\log_B B^2) = O(B)$ I/Os (Global rebuilding)

$\Downarrow$

Update in $O(\log_B N)$ I/Os amortized

External Interval Tree

- Note:
  - Insert may increase number of intervals in underflow structure for some multislab to *B*
  - Delete may decrease number of intervals in multislab to *B*

$\Downarrow$

Need to move *B* intervals to/from multislab/underflow structure

- We only move
  - Intervals from multislab list when decreasing to size *B/2*
  - Intervals to multislab list when increasing to size *B*

$\Downarrow$

$O(1)$ I/Os amortized used to move intervals

Base Tree Update

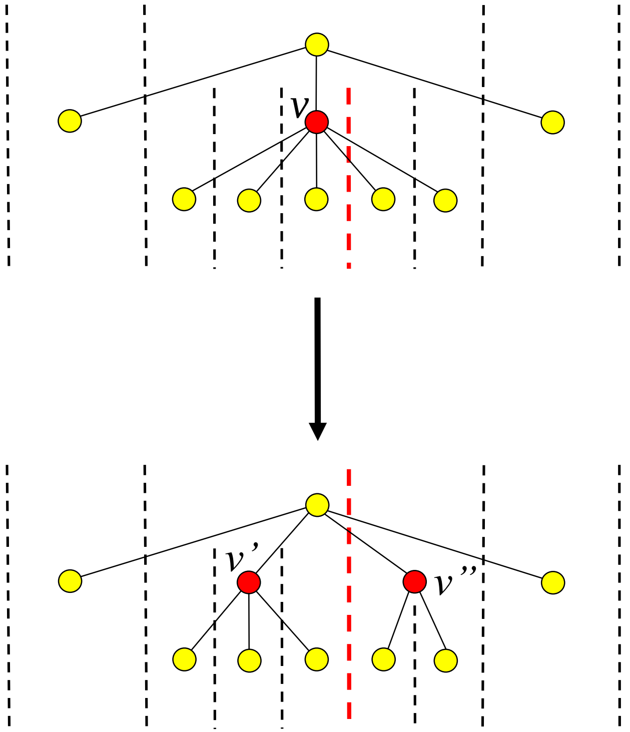
- Before inserting new interval we insert new endpoints in base tree using $O(\log_B N)$ I/Os
  - Leads to rebalancing using splits

    ⇓

    Boundary in *v* becomes boundary in *parent*(*v*)

    ⇓

    Intervals need to be moved

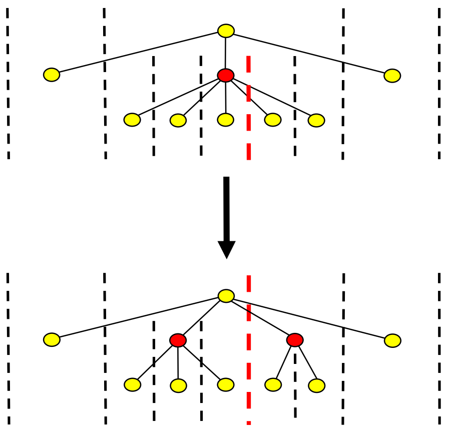


- Move intervals (update secondary structures) in $O(w(v))$ I/Os

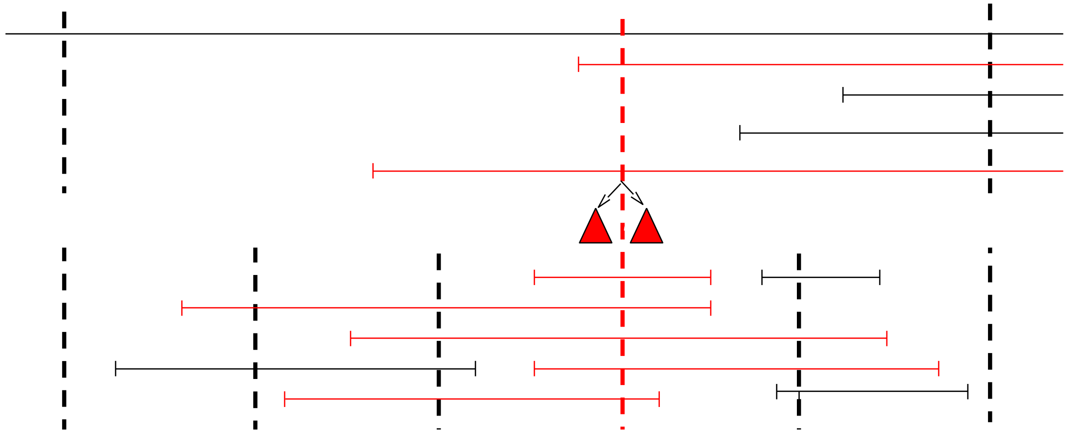
$\Rightarrow O(1)$ amortized split bound (weight balanced B-tree)

$\Rightarrow O(\log_B N)$ amortized insert bound

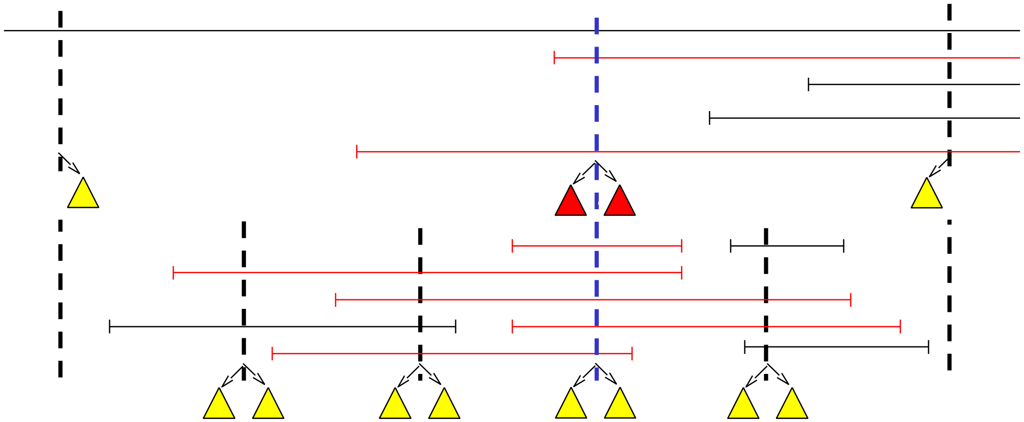
Splitting Interval Tree Node



- When *v* splits we may need to move $O(w(v))$ intervals
  - Intervals in *v* containing boundary
  - Intervals in *parent*(*v*) with endpoints in X_v containing boundary
- Intervals move to two new slab and multislab lists in *parent*(*v*)

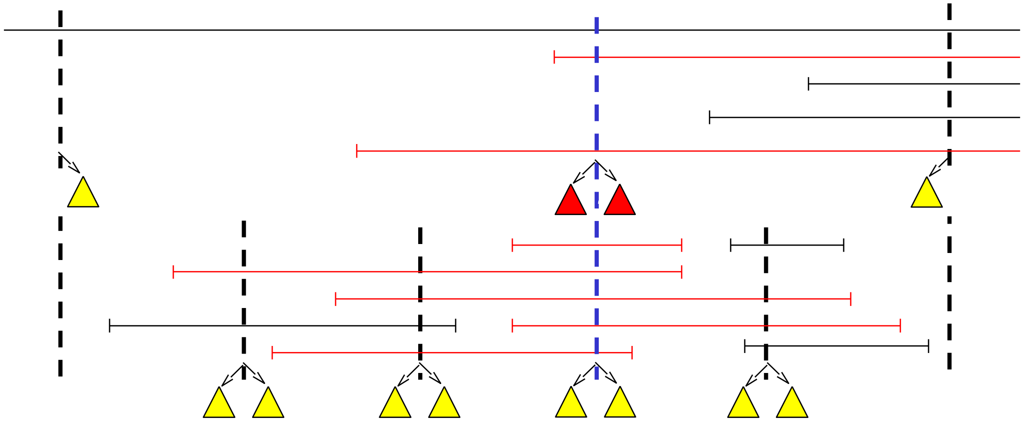


Splitting Interval Tree Node



- Moving intervals in *v* in *O*(*w*(*v*)) I/Os
  - Collect in left order (and remove) by scanning left slab lists
  - Collect in right order (and remove) by scanning right slab lists
  - Remove multislab lists containing boundary
  - Remove from underflow structure by rebuilding it
  - Construct lists and underflow structure for *v'* and *v''* similarly

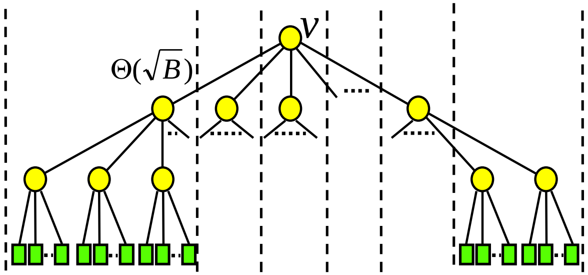
Splitting Interval Tree Node



- Moving intervals in *parent*(*v*) in $O(w(v))$ I/Os
  - Collect in left order by scanning left slab list
  - Collect in right order by scanning right slab list
  - Merge with intervals collected in *v* ⇒ two new slab lists
  - Construct new multislab lists by splitting relevant multislab list
  - Insert intervals in small multislab lists in underflow structure

External Interval Tree

- Split in $O(1)$ I/Os amortized
  - Space: $O(N/B)$
  - Query: $O(\log_B N + T/B)$
  - Insert: $O(\log_B N)$ I/Os amortized

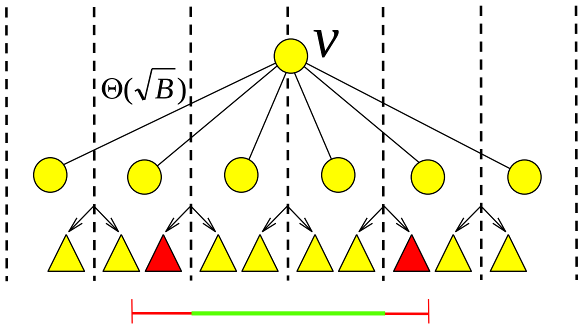


- Deletes in $O(\log_B N)$ I/Os amortized using global rebuilding:
  - Delete interval as previously using $O(\log_B N)$ I/Os
  - Mark relevant endpoint as deleted
  - Rebuild structure in $O(N \log_B N)$ after $N/2$ deletes

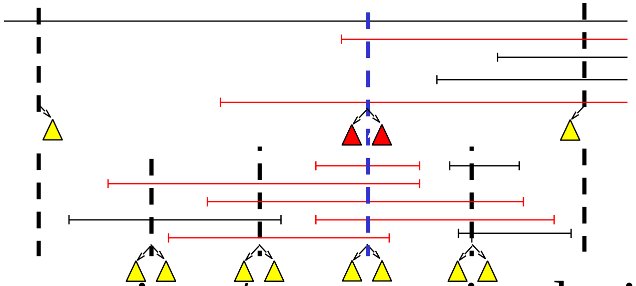
- Note: Deletes can also be handled using fuse operations

External Interval Tree

- External interval tree
  - Space: $O(N/B)$
  - Query: $O(\log_B N + {}^{T}\!/_{B})$
  - Updates: $O(\log_B N)$ I/Os amortized



- Removing amortization:
  - Moving intervals to/from underflow structure
  - Delete global rebuilding
  - Underflow structure update
  - Base node tree splits



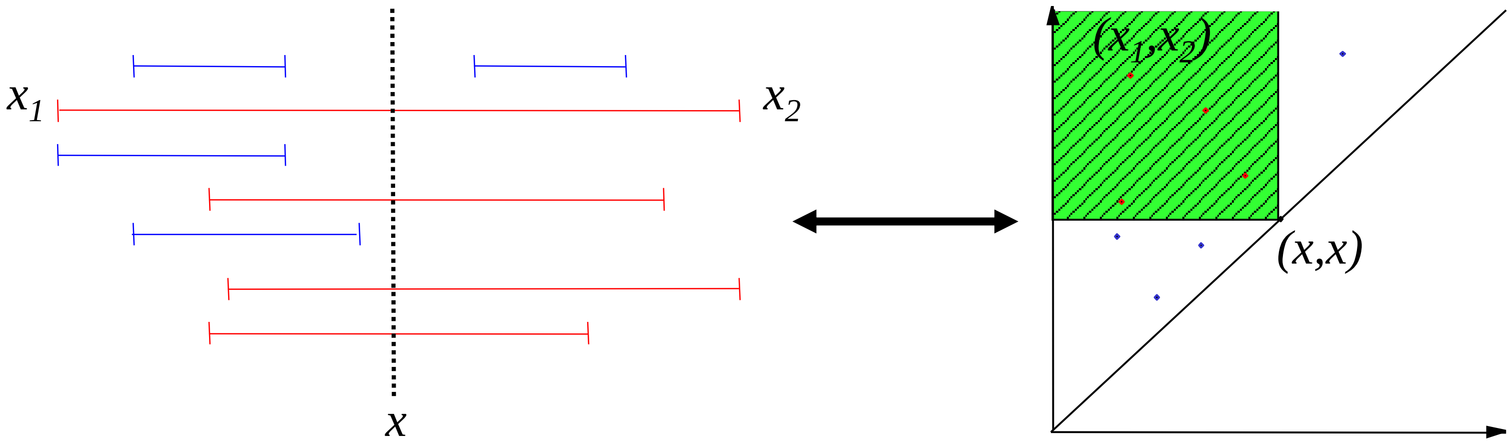
Perform operations/construction lazily

Move lazily – complicated:

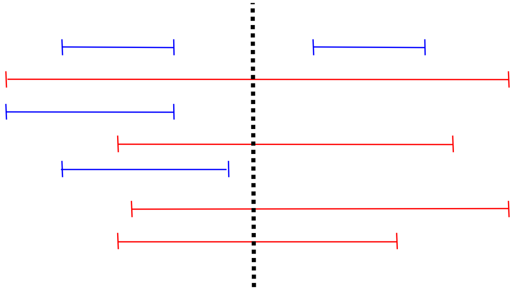
- Interference
- Queries

Summary/Conclusion: Interval Management

- Interval management corresponds to simple form of *2d* range search
  - Diagonal corner queries
- We obtained the same bounds as for the *1d* case
  - Space: $O(N/B)$
  - Query: $O(\log_B N + {}^{T}\!/_{B})$
  - Updates: $O(\log_B N)$ I/Os



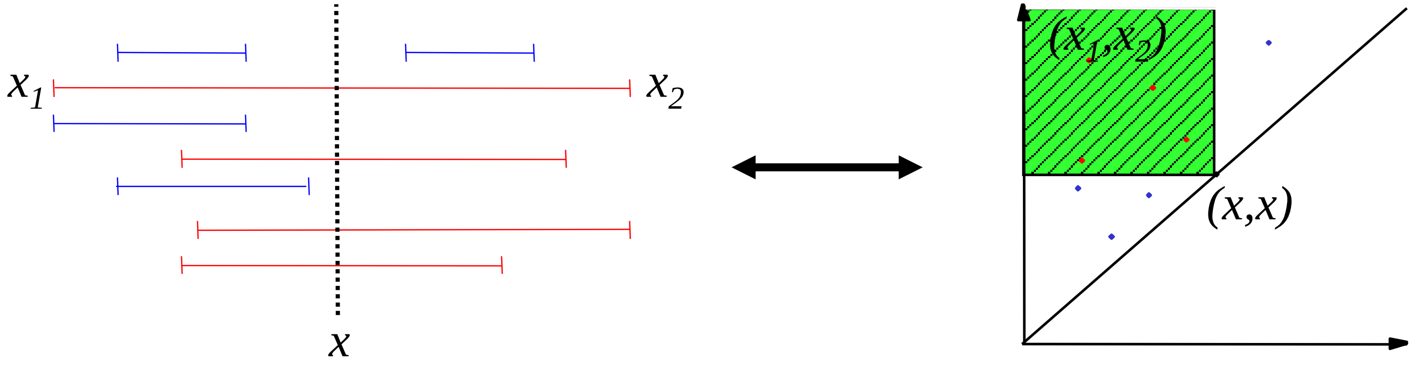
Summary/Conclusion: Interval Management



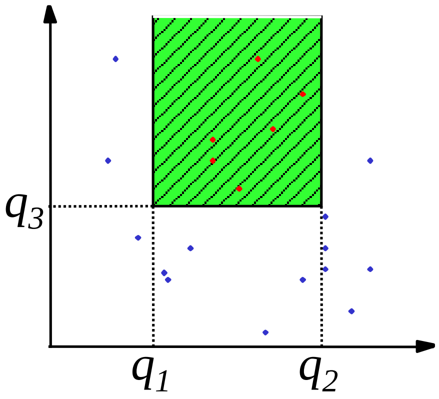
- Main problem in designing structure:
  - Binary → large fan-out
- Large fan-out resulted in the need for
  - Multislabs and multislab lists
  - Underflow structure to avoid $O(B)$-cost in each node
- General solution techniques:
  - Filtering: Charge part of query cost to output
  - Bootstrapping:
    - * 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 $O(1)$

Three-Sided Range Queries

- Interval management: “1.5 dimensional” search

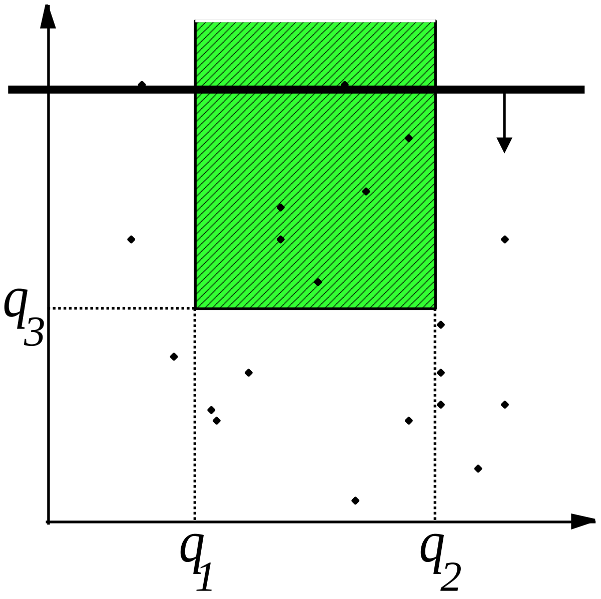


- More general 2d problem: Dynamic 3-sidede range searching
  - Maintain set of points in plane such that given query (q_1, q_2, q_3), all points (x,y) with $q_1 \leq x \leq q_2$ and $y \geq q_3$ can be found efficiently



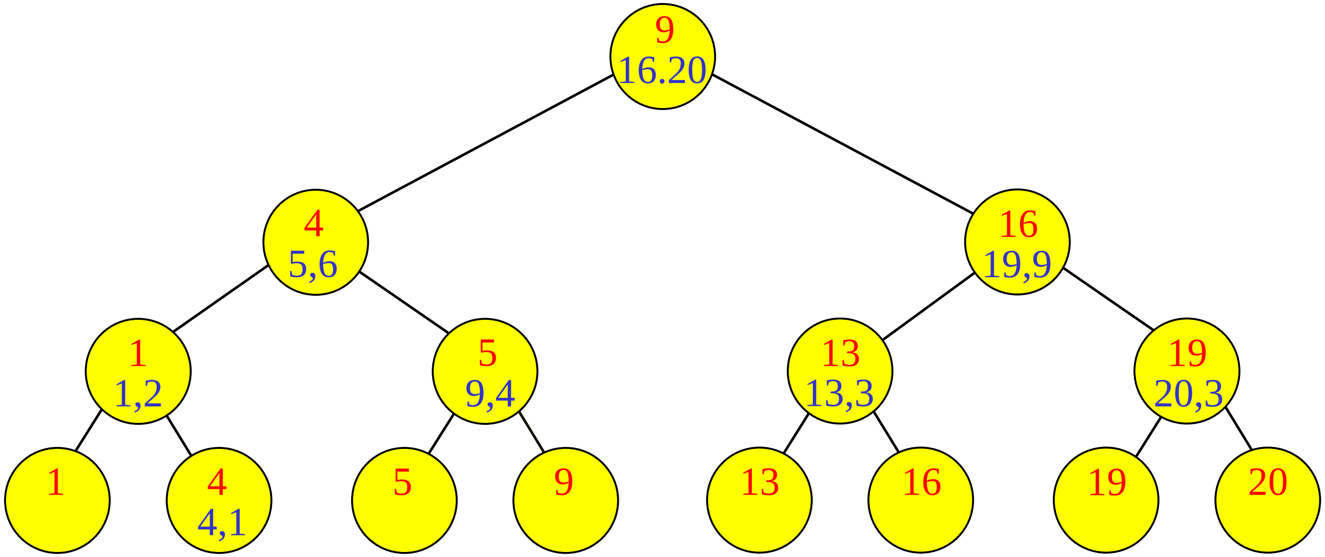
Three-Sided Range Queries

- Report all points (x,y) with $q_1 \leq x \leq q_2$ and $y \geq q_3$
- 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:
  - $O(N/B)$ 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

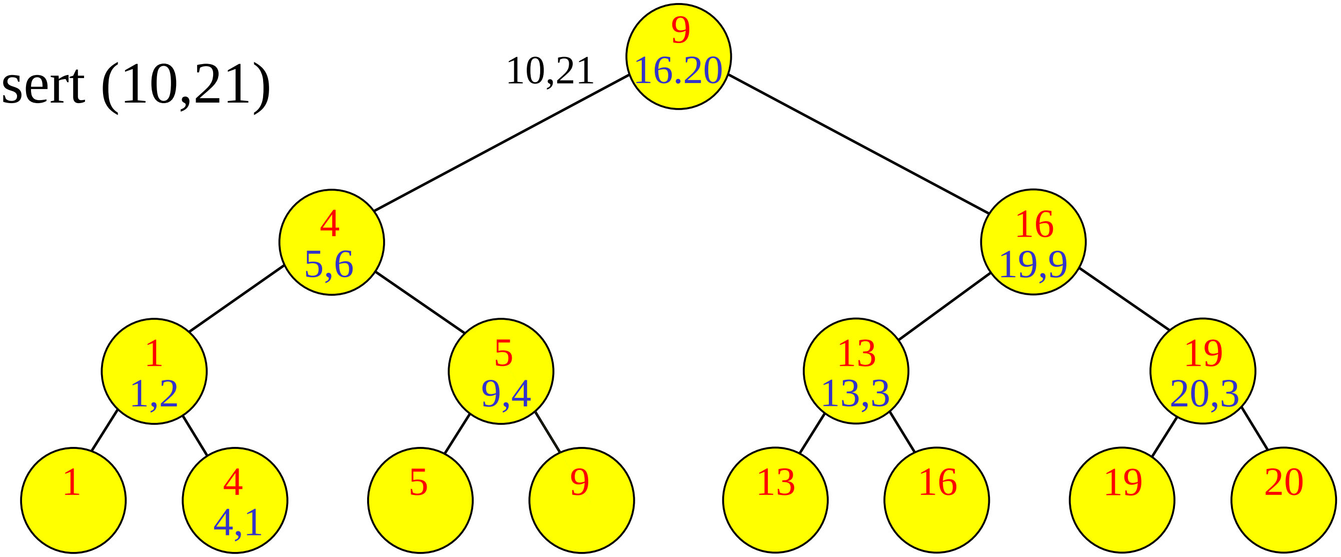
Internal Priority Search Tree



- 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

Internal Priority Search Tree

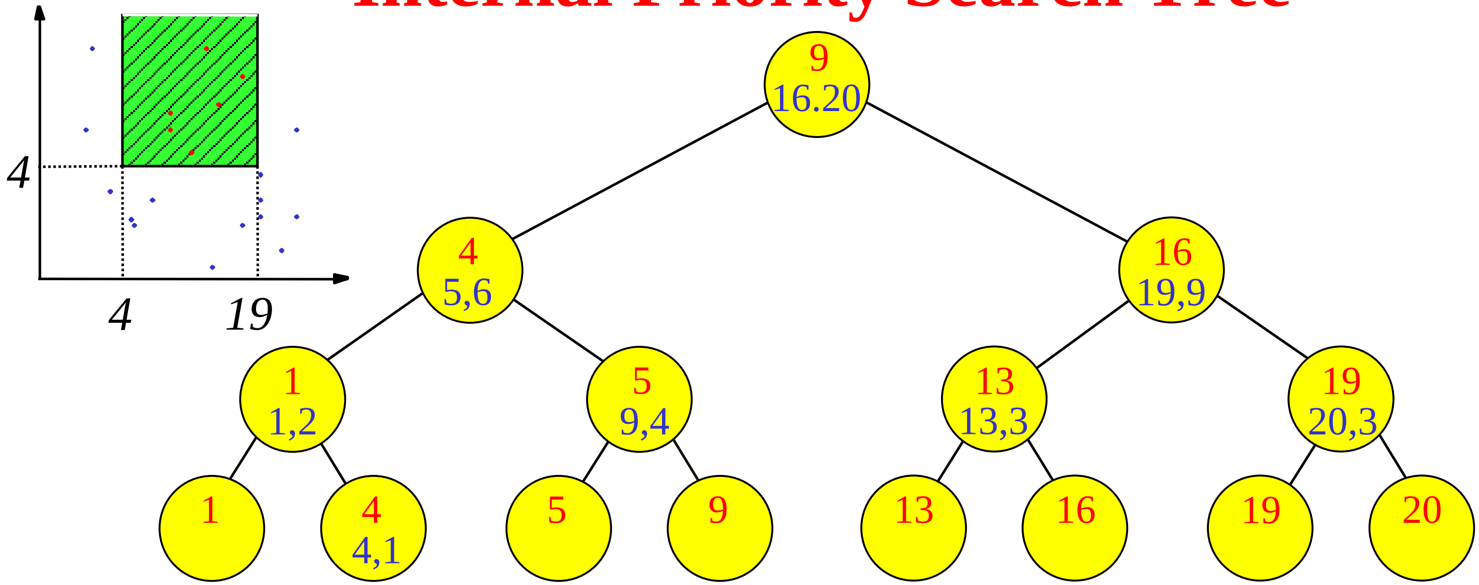
Insert (10,21)



- Linear space
- Insert of (*x*,*y*) (assuming fixed *x*-coordinate set):
  - – Compare y with *y*-coordinate in root
  - – Smaller: Recursively insert (*x*,*y*) in subtree on path to *x*
  - – Bigger: Insert in root and recursively insert old point in subtree

⇒ $O(\log N)$ update

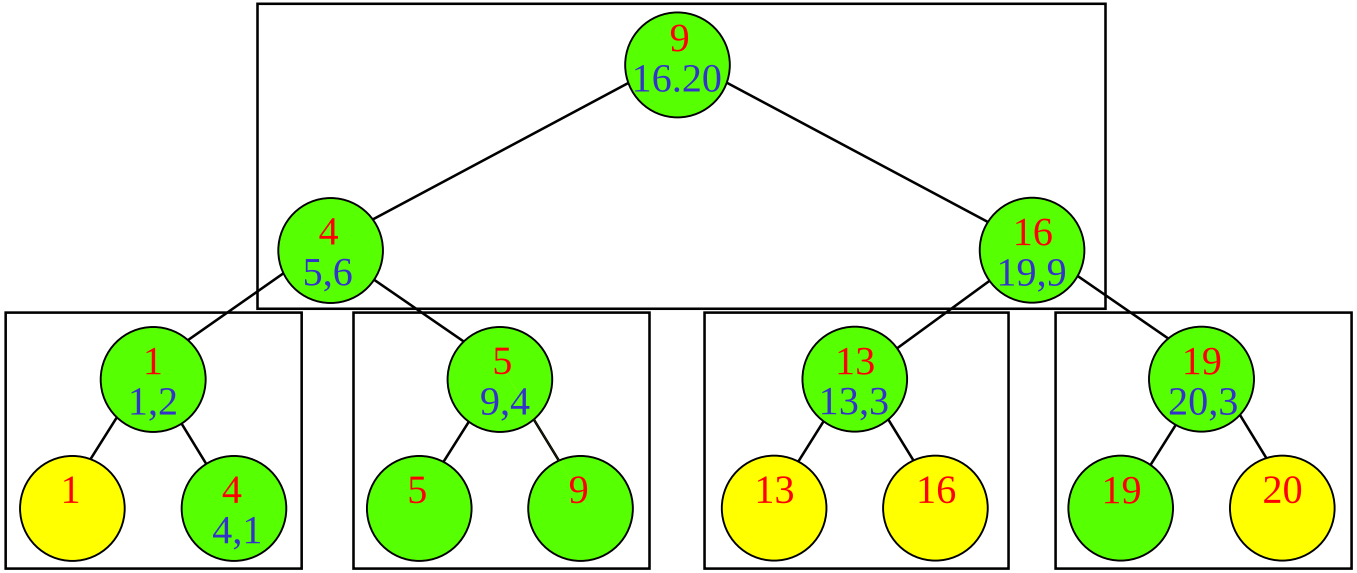
Internal Priority Search Tree



- <span>Query</span> 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

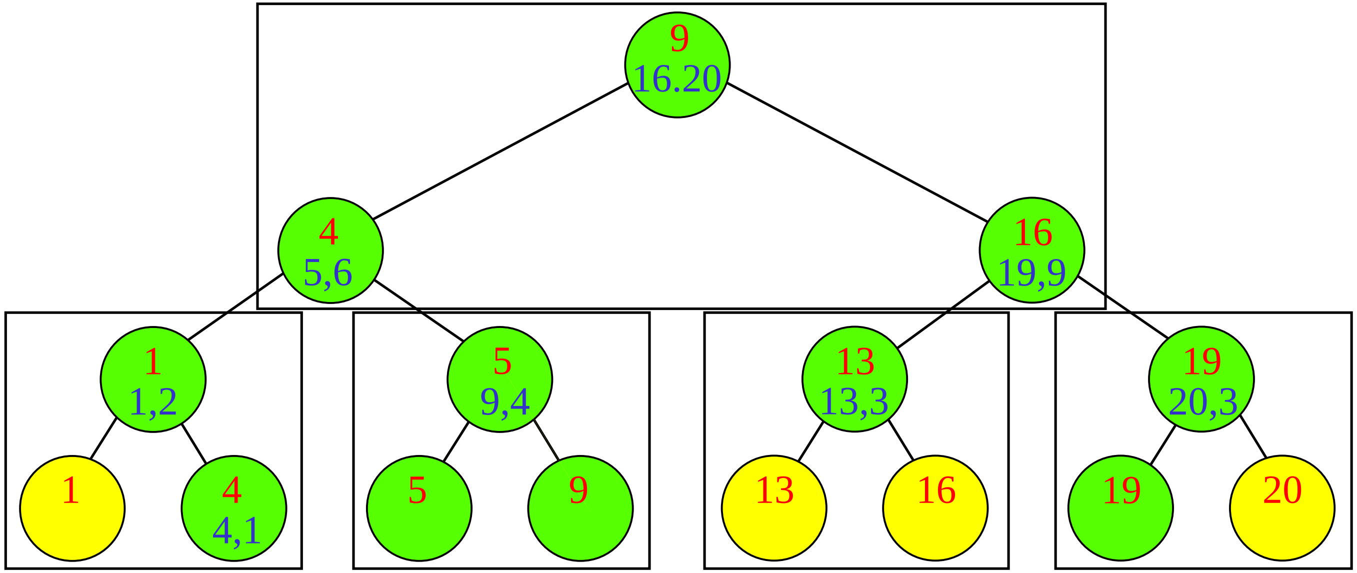
Externalizing Priority Search Tree



- Natural idea: Block tree
- Problem:
  - $O(\log_B N)$ I/Os to follow paths to 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



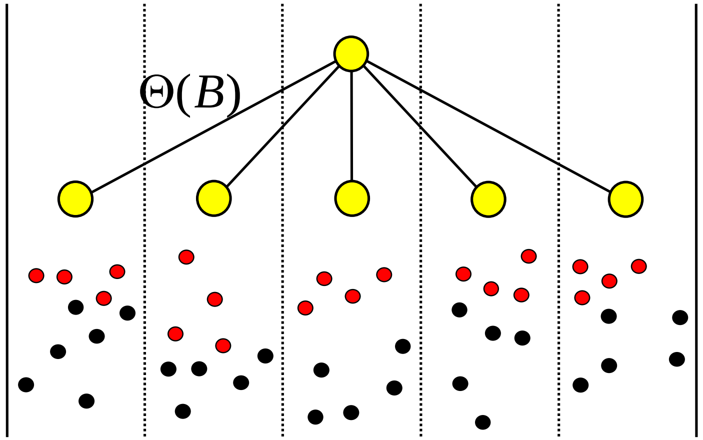
- Solution idea:
  - Store *B* points in each node ⇒
    - * $O(B^2)$ points stored in each supernode
    - * *B* output points can pay for “overshooting”
  - Bootstrapping:
    - * Store $O(B^2)$ points in each supernode in static structure

External Priority Search Tree

- Base tree: Weight-balanced B-tree with branching parameter *B/4* and leaf parameter *B* on *x*-coordinates
- Points in "heap order":
  - Root stores *B* top points for each of the $\Theta(B)$ child slabs
  - Remaining points stored recursively
- Points in each node stored in "B^2-structure"
  - Persistent B-tree structure for static problem

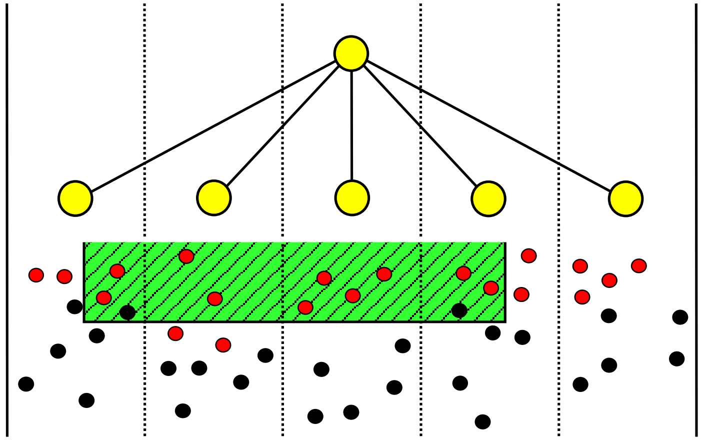
⇓

Linear space



External Priority Search Tree

- Query with (q_1, q_2, q_3) starting at root v:
  - Query B^2-structure and report points satisfying query
  - Visit child v if
    - * v on path to q_1 or q_2
    - * All points corresponding to v satisfy query

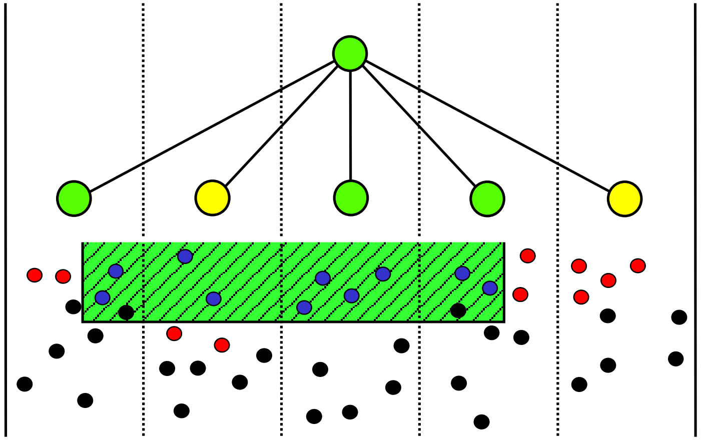


External Priority Search Tree

- 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 $parent(v)$

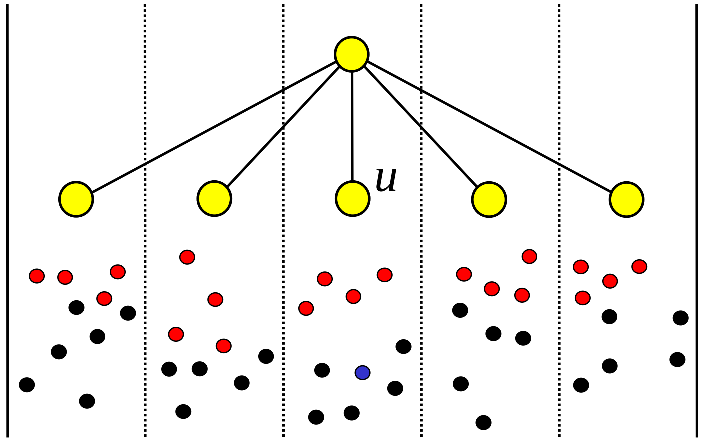
$\Downarrow$

$O(\log_B N + T/B)$ query



External Priority Search Tree

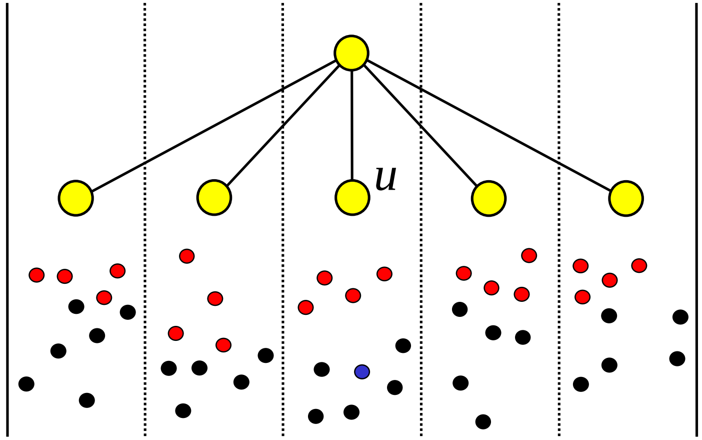
- Insert (x,y) (ignoring insert in base tree - rebalancing):
  - Find relevant node *u:*
    - Query B^2-structure to find *B* points in root corresponding to node *u* on path to *x*
    - If *y* smaller than *y*-coordinates of all *B* points then recursively search in *u*



  - Insert (x,y) in B^2-structure of *v*
  - If B^2-structure contains >*B* points for child *u*, remove lowest point and insert recursively in *u*
- Delete: Similarly

External Priority Search Tree

- Analysis:
  - Update visits $O(\log_B N)$ nodes
  - B^2-structure queried/updated in each node
    - * One query
    - * One insert and one delete
- B^2-structure analysis:
  - Query: $O(\log_B B^2 + B/B) = O(1)$
  - Update: $O(1)$ using global rebuilding
    - * Store updates in update block
    - * Rebuild after B updates using $O(\frac{B^2}{B}\log_{M/B}\frac{B^2}{B}) = O(B)$ I/Os



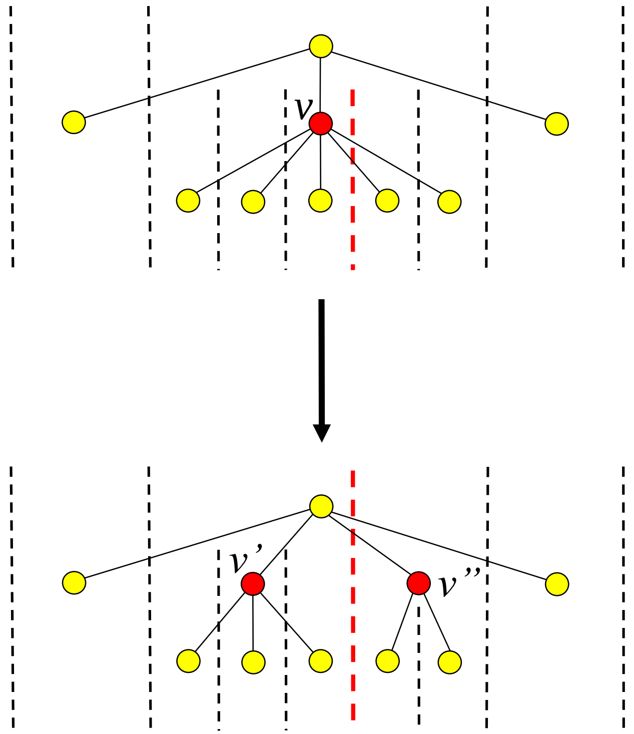
$\Downarrow$

$O(\log_B N)$ I/O updates

Dynamic Base Tree

- Deletion:
  - Delete point as previously
  - Delete x-coordinate from base tree using global rebuilding

  $\Rightarrow O(\log_B N)$ I/Os amortized

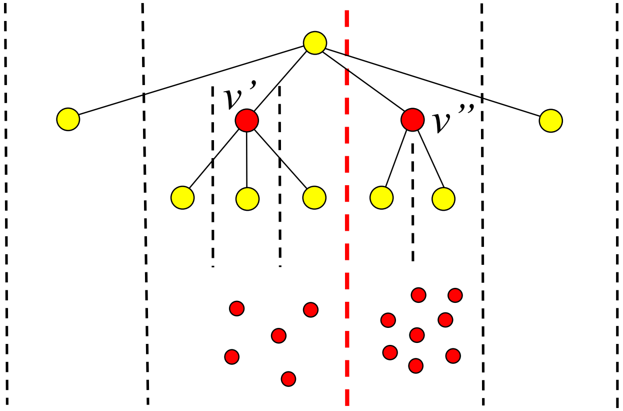


- Insertion:
  - Insert x-coordinate in base tree and rebalance (using splits)
  - Insert point as previously
- Split: Boundary in v becomes boundary in $parent(v)$

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 B^2-structure
  - Remove *p* from 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



$\Downarrow$

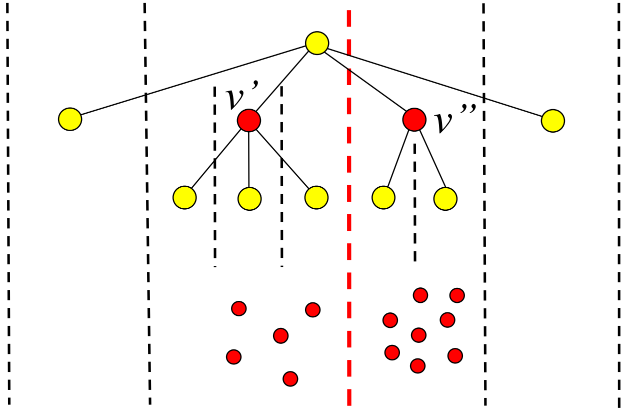
Split in $O(B \log_B w(v)) = O(w(v))$ I/Os

Dynamic Base Tree

- $O(1)$ amortized split cost:
  - Cost: $O(w(v))$
  - Weight balanced base tree: $\Omega(w(v))$ inserts below v between splits

$\Downarrow$

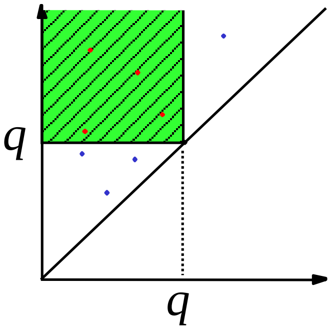
- External Priority Search Tree
  - Space: $O(N/B)$
  - Query: $O(\log_B N + T/B)$
  - Updates: $O(\log_B N)$ I/Os amortized

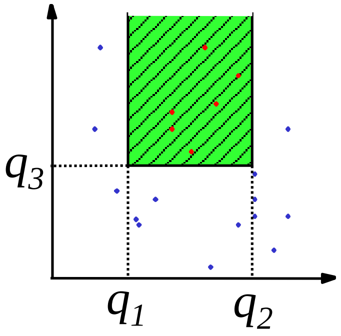


- Amortization can be removed from update bound in several ways
  - Utilizing lazy rebuilding

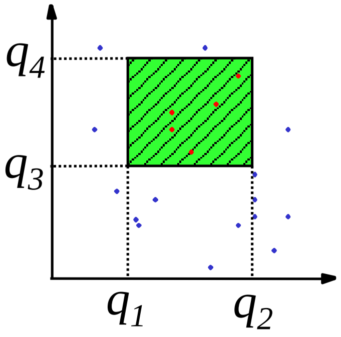
Summary/Conclusion: Priority Search Tree

- 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 $\Omega(\frac{N}{B}\frac{\log_B N}{\log_B \log_B N})$ space
  - $O(\frac{N}{B})$ space requires $\Omega(\sqrt{{}^{N}\!/_{B}})$ query



References

- **External Memory Geometric Data Structures**

  Lecture notes by Lars Arge.

  – Section 6-7