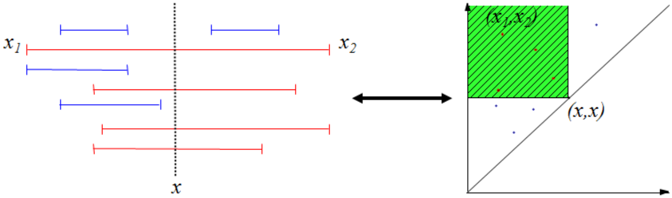
Massive Data Algorithmics

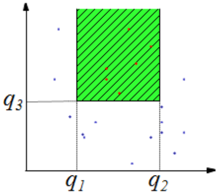
Lecture 7: Range Searching

Three-Sided Range Queries

- Interval management: 1.5 dimensional search

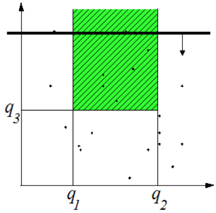


- More general $2d$ problem: Dynamic 3-sidede range searching
  - Maintain set of points in plane such that given query (q_1, q_2, q_3), all points (x, y) with $q_1 \leq x \leq q_2$ and $y \geq q_3$ can be found efficiently



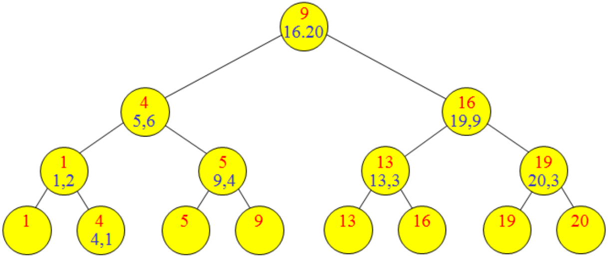
Three-Sided Range Queries

- 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



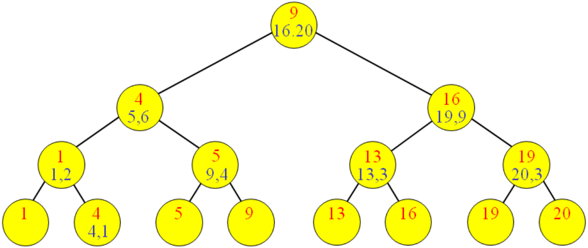
- Optimal:
  - $O(N/B)$ space
  - $O(log_B N + T/B)$ query
  - $O(N/B \log_{M/B} N/B)$ construction
- Dynamic? in internal memory: priority search tree

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

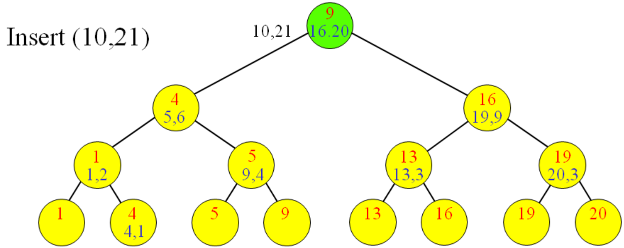
Internal Priority Search Tree



- 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

$\Rightarrow O(\log N)$ update

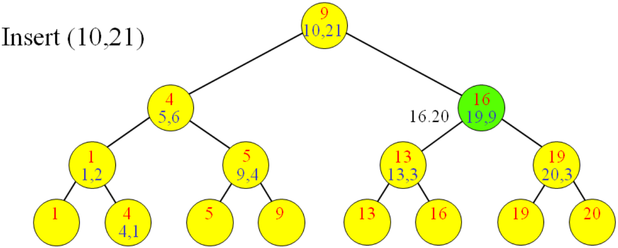
Internal Priority Search Tree



- 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

$\Rightarrow O(\log N)$ update

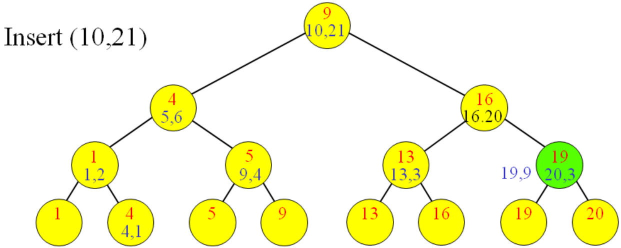
Internal Priority Search Tree



- 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

$\Rightarrow O(\log N)$ update

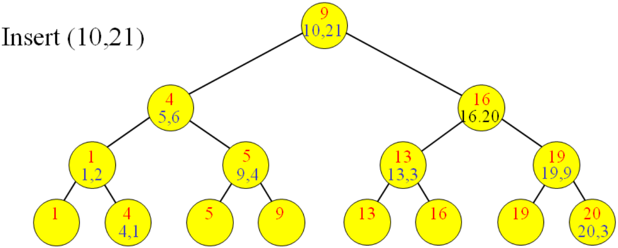
Internal Priority Search Tree



- 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

$\Rightarrow O(\log N)$ update

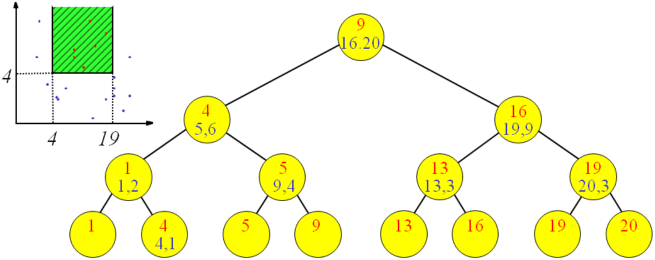
Internal Priority Search Tree



- 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

$\Rightarrow O(\log N)$ update

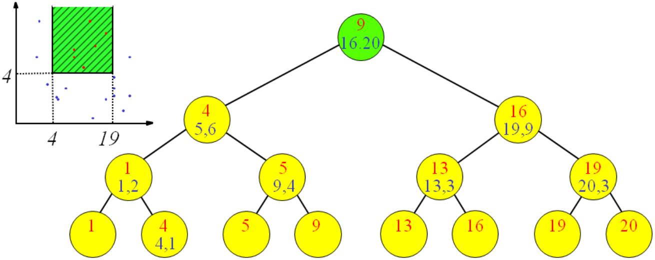
Internal Priority Search Tree



- 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

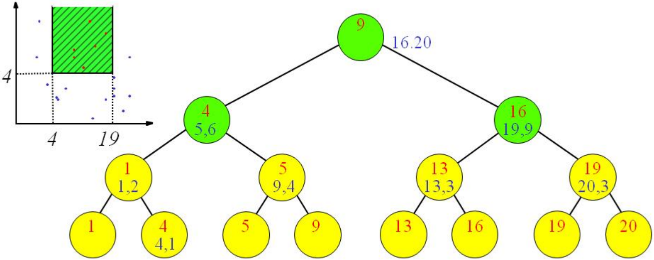
Internal Priority Search Tree



- 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

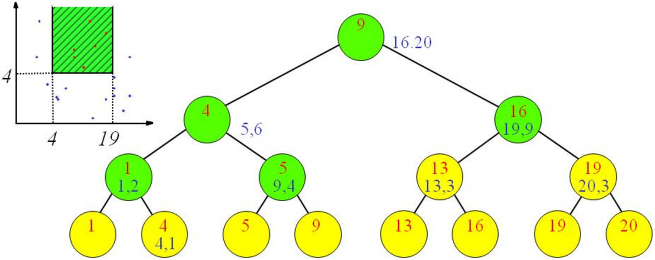
Internal Priority Search Tree



- 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

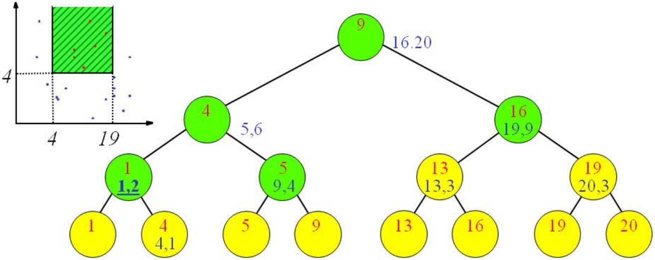
Internal Priority Search Tree



- 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

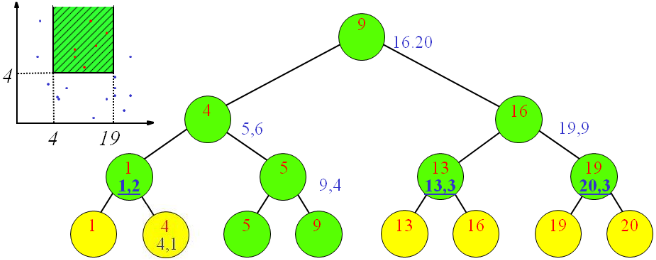
Internal Priority Search Tree



- 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

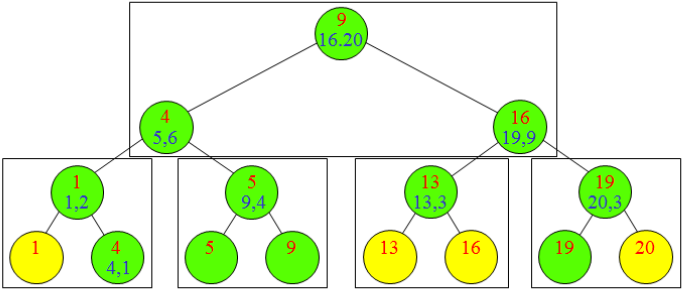
Internal Priority Search Tree



- 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

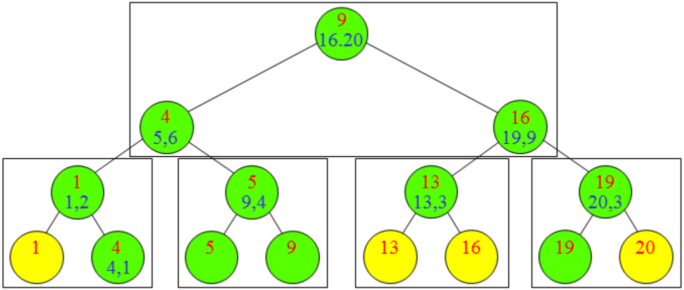
Externalizing Priority Search Tree



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

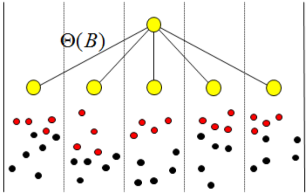
Externalizing Priority Search Tree



- 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

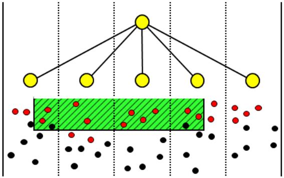
Externalizing Priority Search Tree

- 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
- $\Rightarrow$ Linear space



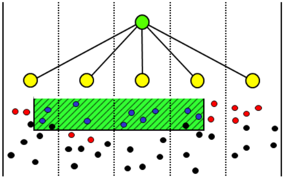
Externalizing Priority Search Tree

- 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



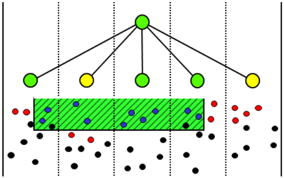
Externalizing Priority Search Tree

- 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



Externalizing Priority Search Tree

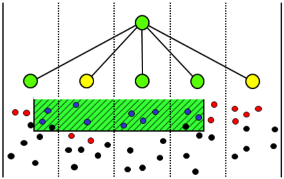
- Query with (q_1, q_2, q_3) starting at root v:
  - Query B^2-structure and report points satisfying query
  - Visit child v if
    - v on path to q_1 or q_2
    - All points corresponding to v satisfy query



Externalizing Priority Search Tree

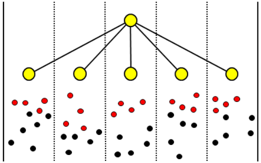
- Analysis:
  - $O(\log_B B^2 + T_v/B) = O(1 + T_v/B)$ I/Os used to visit node v
  - $O(\log_B N)$ nodes on path to q_1 or q_2
  - For each node v not on path to q_1 or q_2 visited, B points reported in parent(v)

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



Externalizing Priority Search Tree

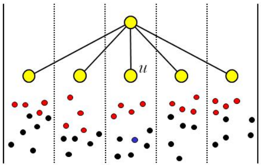
- Insert (x,y) (ignoring insert in base tree - rebalancing):
  - Find relevant node u:
    - * Query B^2-structure to find B points in root corresponding to node u on path to x
    - * If y smaller than y-coordinates of all B points then recursively search in u
  - Insert (x,y) in B^2-structure of v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



- Delete: Similarly

Externalizing Priority Search Tree

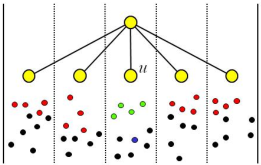
- Insert (x,y) (ignoring insert in base tree - rebalancing):
  - Find relevant node u:
    - Query B^2-structure to find B points in root corresponding to node u on path to x
    - If y smaller than y-coordinates of all B points then recursively search in u



  - 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

Externalizing Priority Search Tree

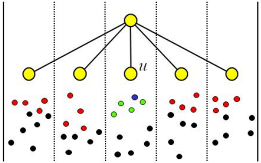
- Insert (x,y) (ignoring insert in base tree - rebalancing):
  - Find relevant node u:
    - Query B^2-structure to find B points in root corresponding to node u on path to x
    - If y smaller than y-coordinates of all B points then recursively search in u



  - 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

Externalizing Priority Search Tree

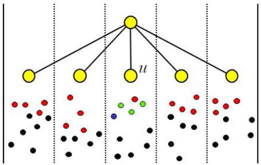
- Insert (x,y) (ignoring insert in base tree - rebalancing):
  - Find relevant node u:
    - * Query B^2-structure to find B points in root corresponding to node u on path to x
    - * If y smaller than y-coordinates of all B points then recursively search in u
  - Insert (x,y) in B^2-structure of v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



- Delete: Similarly

Externalizing 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

Externalizing Priority Search Tree

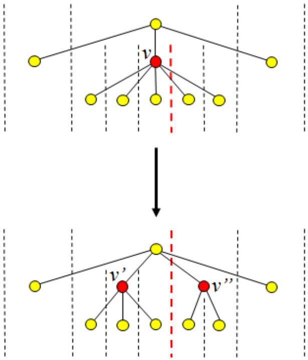
- Analysis:
  - Update visits $O(\log_B N)$ nodes
  - B^2-structure queried/updated in each node
    - One query
    - One insert and one delete
- B^2-structure analysis:
  - Query: $O(\log_B B^2 + B/B) = O(1)$ I/Os
  - Update: $O(1)$ using global rebuilding
    - Store updates in update block
    -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

$\Rightarrow O(\log_B N)$ update

Dynamic Base Tree

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

  $\Rightarrow O(\log_B N)$ I/Os
- Insertion:
  - Insert x-coordinate in base tree and rebalance (using splits)
  - Insert point as previously
- Split: Boundary in v becomes boundary in parent(v)

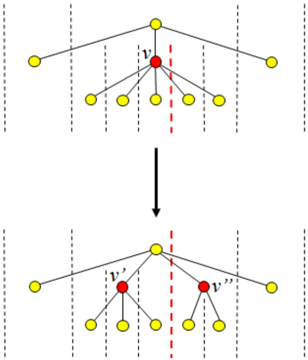


Dynamic Base Tree

- 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'

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

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



Dynamic Base Tree

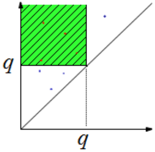
- $O(1)$ amortized split cost:
  - Cost: $O(w(v))$
  - Weight balanced base tree: $\Omega(w(v))$ inserts below v between splits

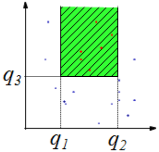
$\Downarrow$

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

Summary/Conclusion: Range Search

- 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
  - $O(N/B)$ space requires $\Omega(\sqrt{N/B})$ query