EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Hw6 & Q6: Today...

Hw7 (Q7?): out by Friday..

Projects: Approved

Still taking bids on alternate final exam time
(Favorite: F... AM)

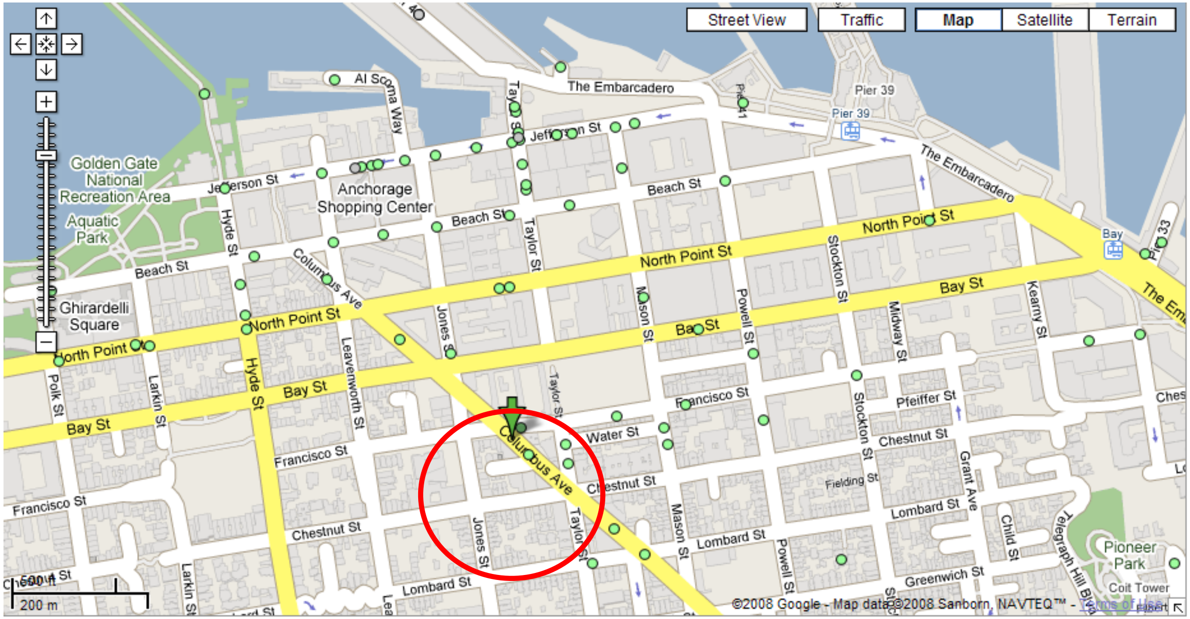
Monday slides: Corrected to compute accurate distances

Data Structures for Multidimensional Search

- So far, focused on 1-D data
  - Balanced BSTs, B+ trees, …
- Many applications involve data which is higher-dimensional
  - Astronomy (simulation of galaxies) - 3 dimensions
  - Protein folding in molecular biology - 3 dimensions
  - Lossy data compression - 4 to 64 dimensions
  - Image processing - 2 dimensions
  - Graphics - 2 or 3 dimensions
  - Animation - 3 to 4 dimensions
  - Geographical databases - 2 or 3 dimensions
  - Web searching - 200 or more dimensions
  - Machine learning - hundreds of dimensions

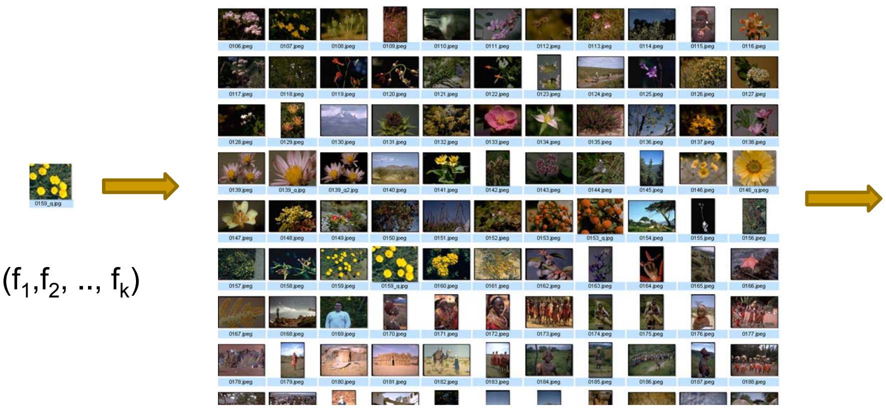
K-Nearest-Neighbor

Problem: whats are the 4 closest restaurants to my hotel



Nearest Neighbor Query in High Dimensions

- Very important and practical problem!
  - Image retrieval

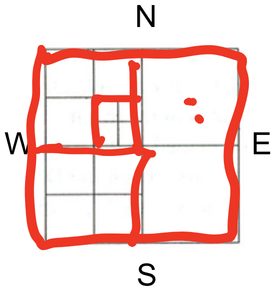


$(f_1,f_2, .., f_k)$

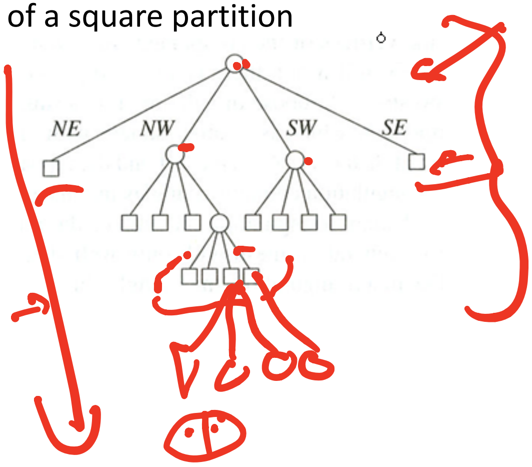
find N closest matches (i.e., N nearest neighbors)

Point-Region Quadtree

2d



- PR Quadtrees are tries
  - Trie: Decomposition based on equal division of the key space
  - Shaped like a tree, with each internal node with 4 children (some empty)
- Every internal node corresponds to a region, with midpoint used for navigation
- Leaves correspond to 2-D points
- The children of a node correspond to the four quadrants of a square partition of a region
  - The children of a node are labelled NE, NW, SW, and SE to indicate to which quadrant they correspond
- If a leaf contains more than one point, it splits into 4 subregions
- Need rule to break ties: arbitrary prefer N to S, E to W
- 3-D variant: Octrees

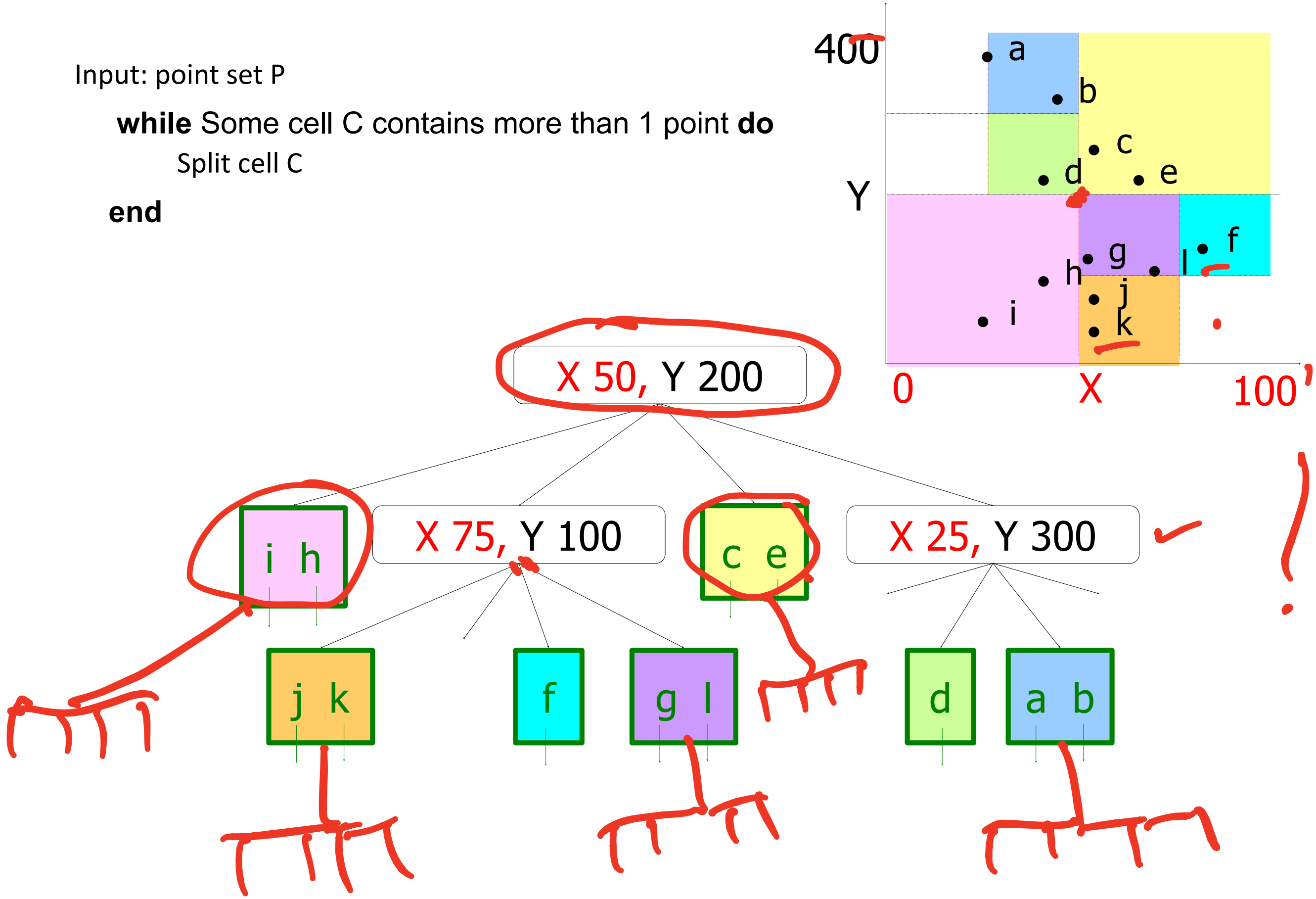


4d?:

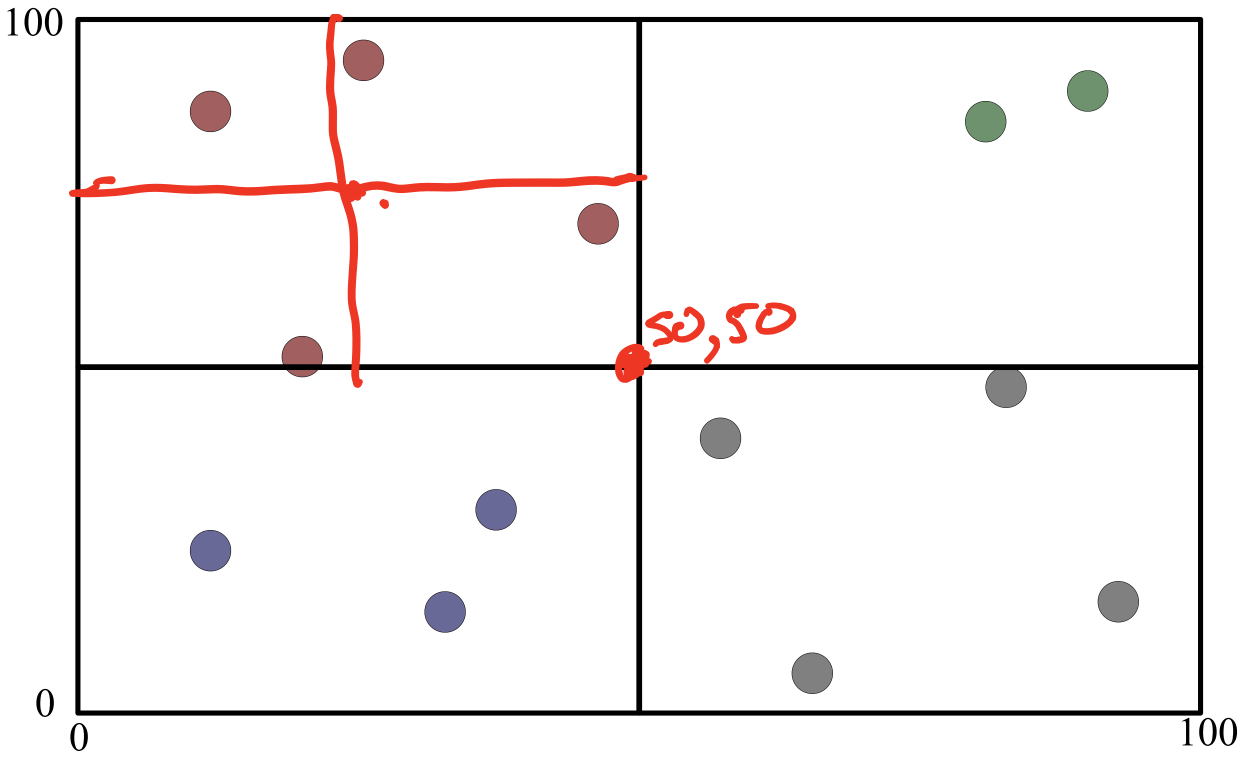
Quadtree Construction

Input: point set P

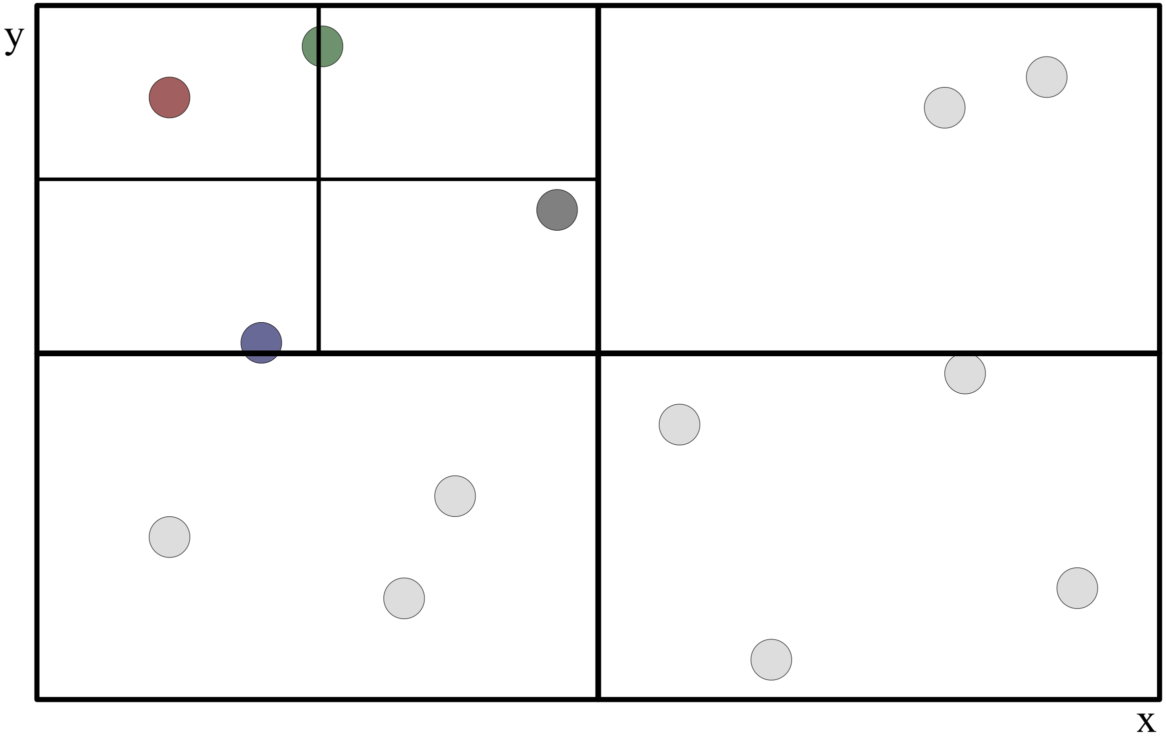
```
while Some cell C contains more than 1 point do
    Split cell C
end
```



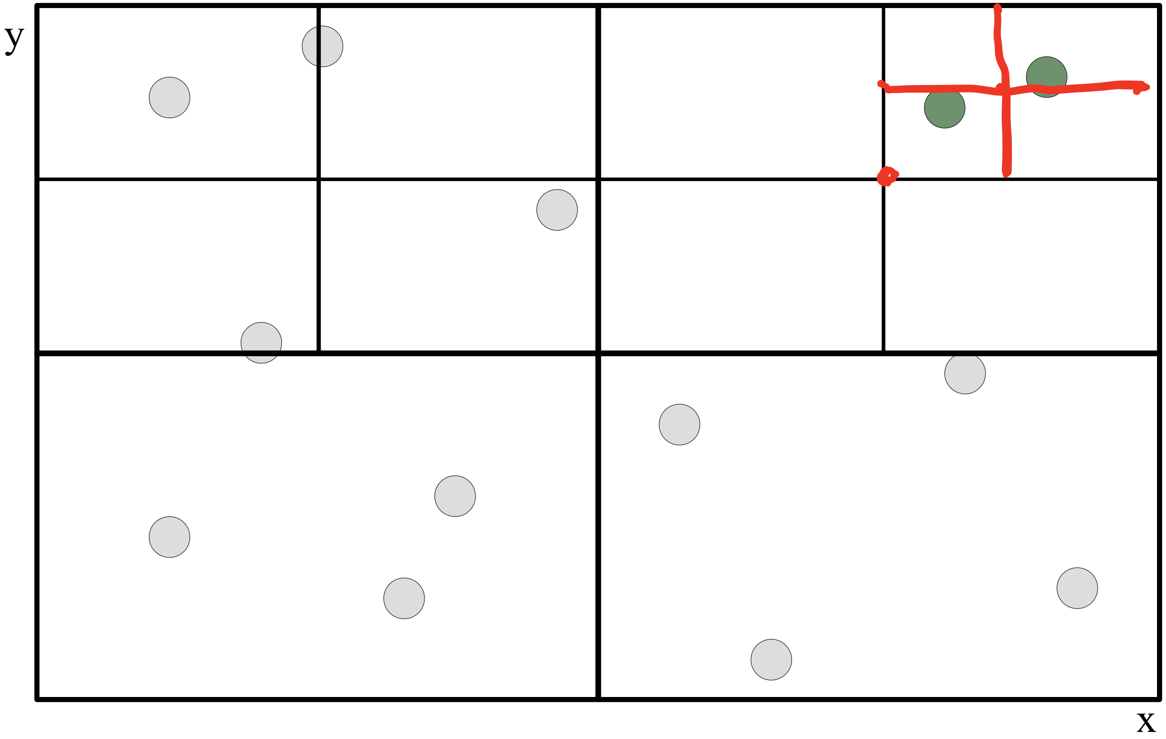
Building a Quad Tree (1/5)



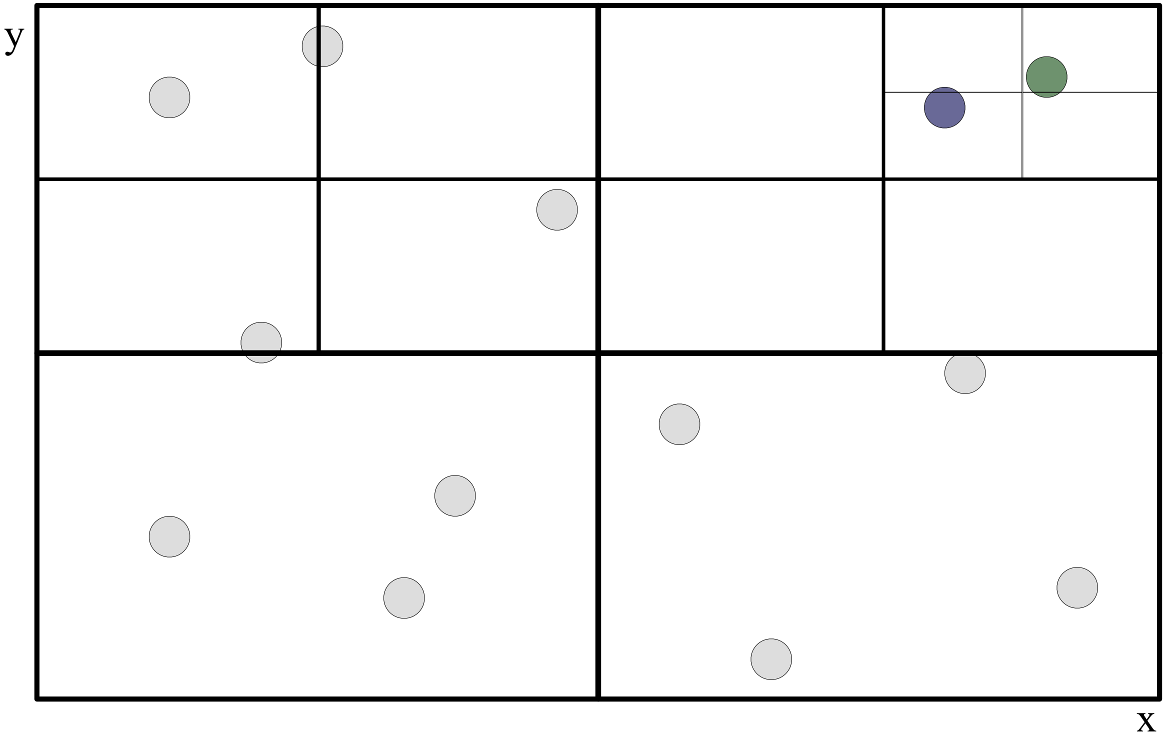
Building a Quad Tree (2/5)



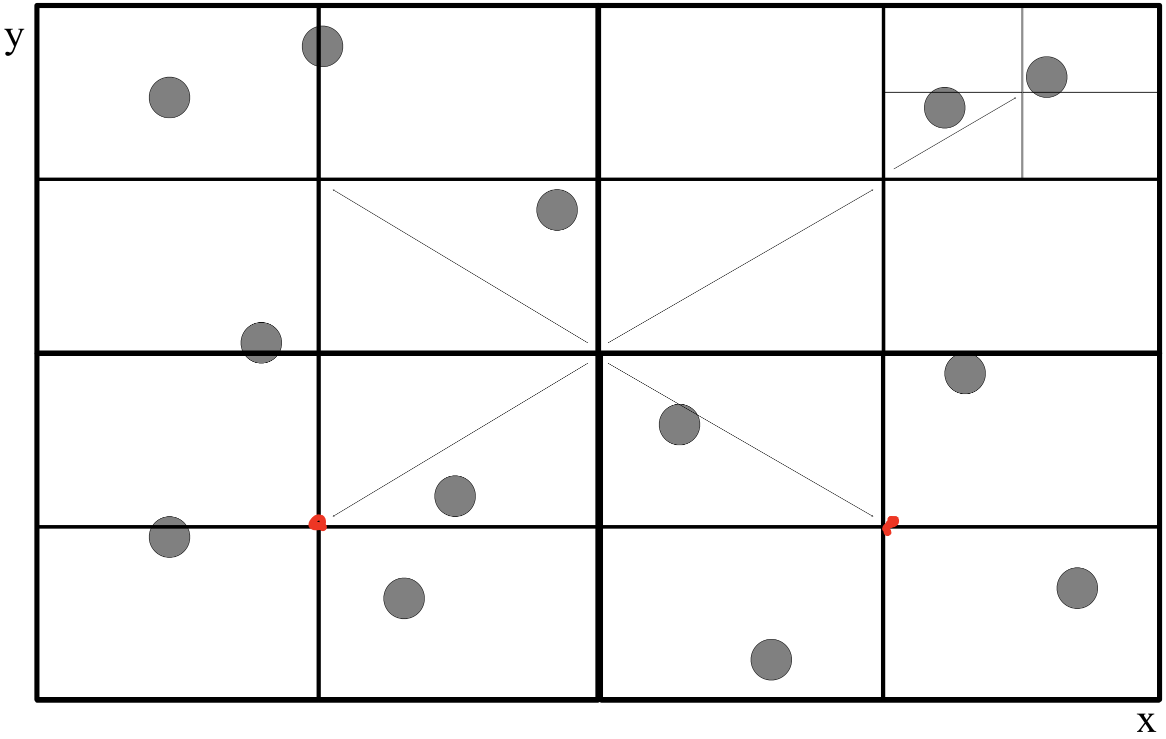
Building a Quad Tree (3/5)



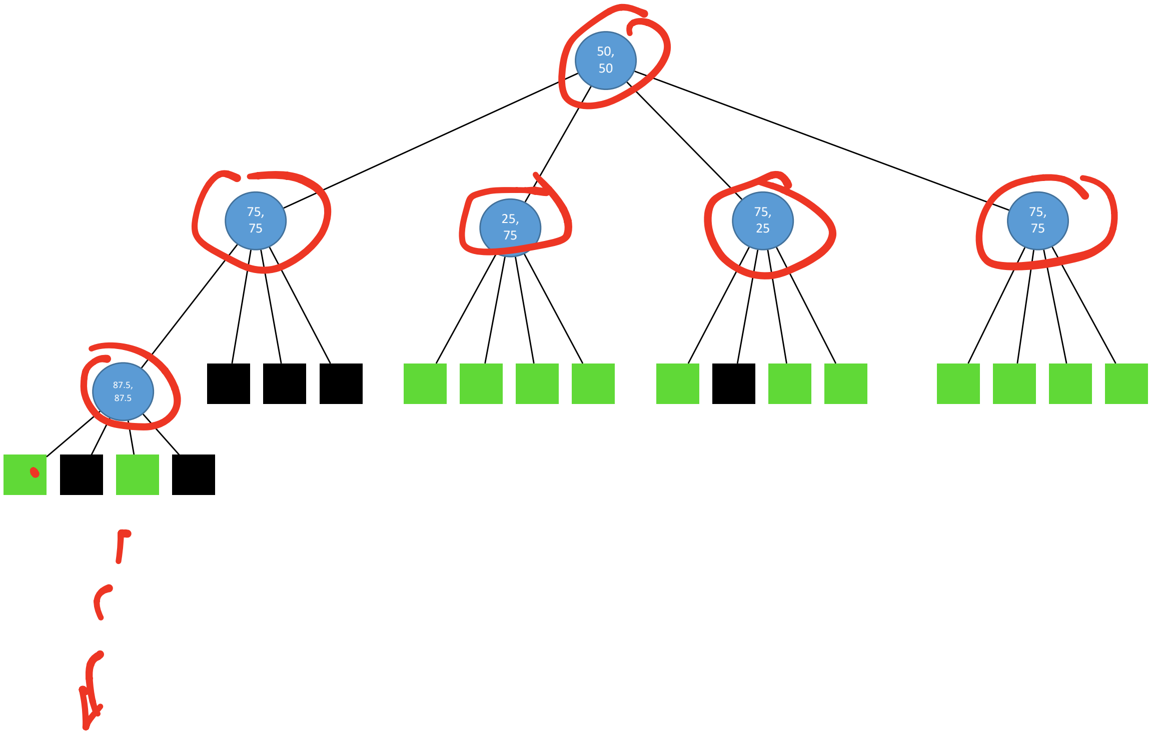
Building a Quad Tree (4/5)



Building a Quad Tree (5/5)



Quadtree Representation

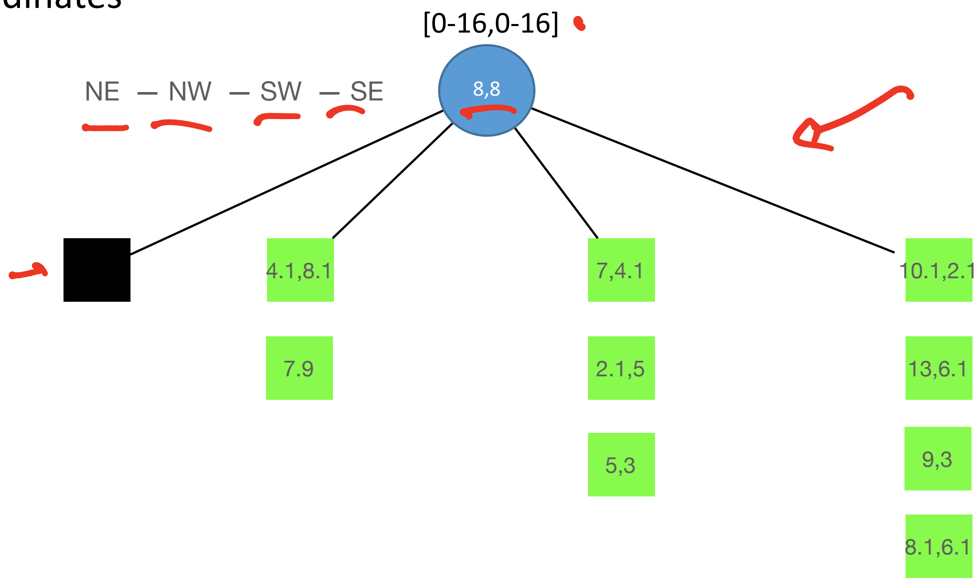


Quadtree Properties

- The depth of a quadtree for a set P of points in the plane is at most $O(\log(s/c))$, where c is the smallest distance between any ~~to~~ 2 points in P and s is the side length of the initial square.
- A quadtree of depth d which stores a set of n points has $O((d + 1)n)$ nodes and can be constructed in $O((d + 1)n)$ time.
- The neighbor of a given node in a given direction can be found in $O(d +1)$ time.

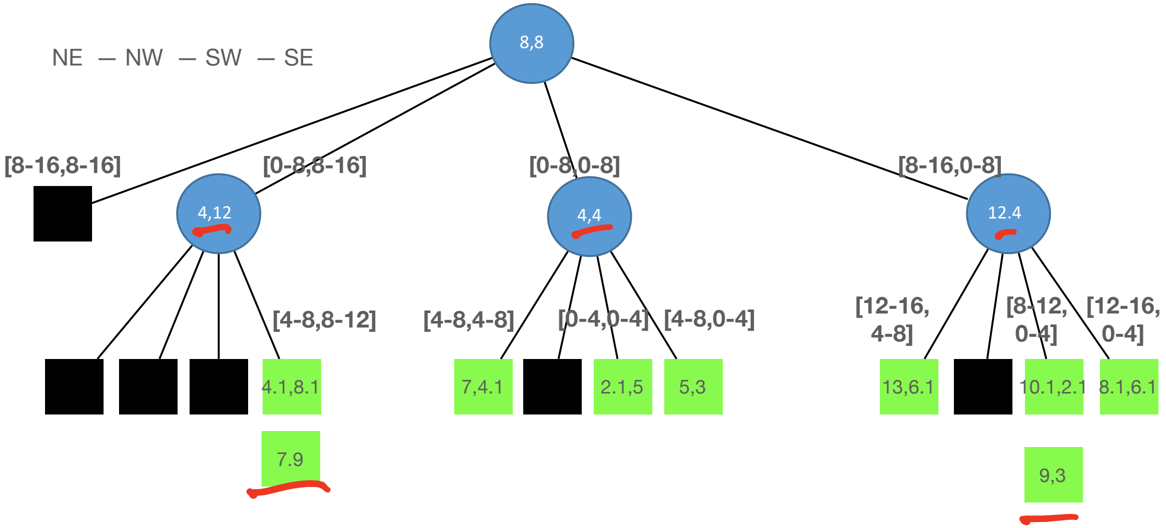
Build Example - 1

- Coordinates: (7,4.1), (2.1,5), (4.1,8.1), (5,3), (8.1,6.1), (10.1,2.1), (13,6.1) (7,9), (9,3). Arrange them in a quadtree, using the range 0-16 for each of the coordinates



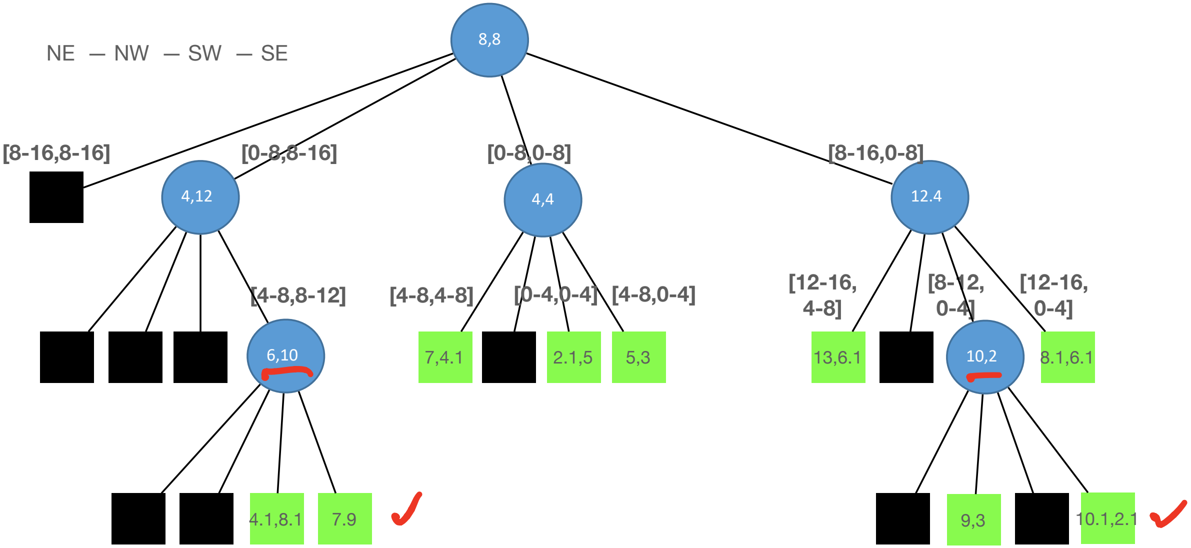
Build Example - 2

- Coordinates: (7,4.1), (2.1,5), (4.1,8.1), (5,3), (8.1,6.1), (10.1,2.1), (13,6.1) (7,9), (9,3). Arrange them in a quadtree, using the range 0-16 for each of the coordinates



Build Example - 3

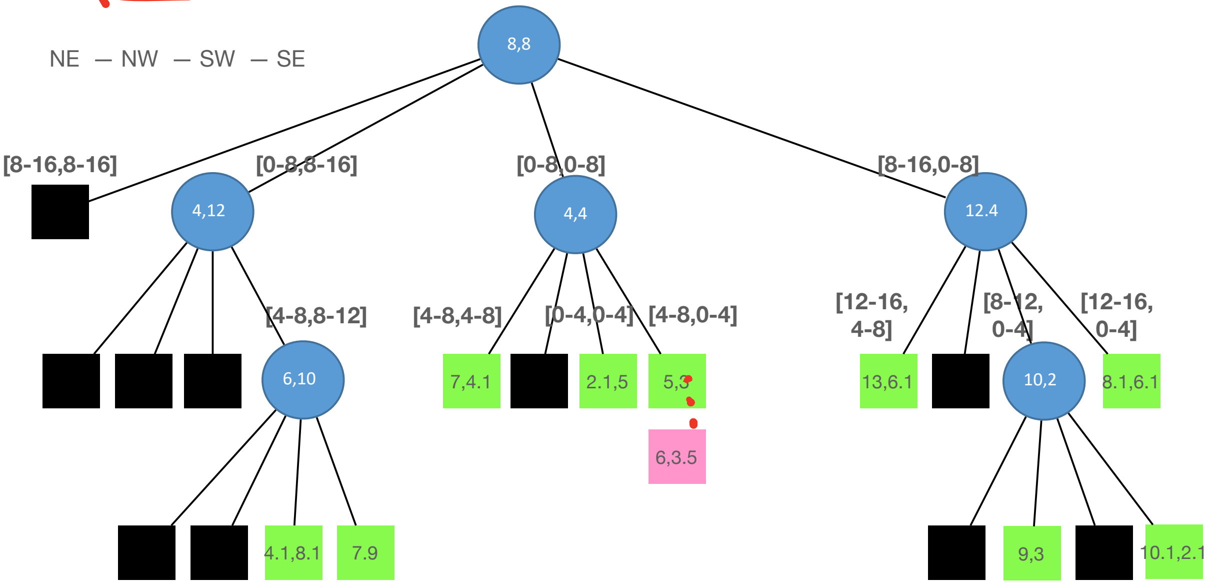
- Coordinates: (7,4.1), (2.1,5), (4.1,8.1), (5,3), (8.1,6.1), (10.1,2.1), (13,6.1) (7,9), (9,3). Arrange them in a quadtree, using the range 0-16 for each of the coordinates



Insert Example

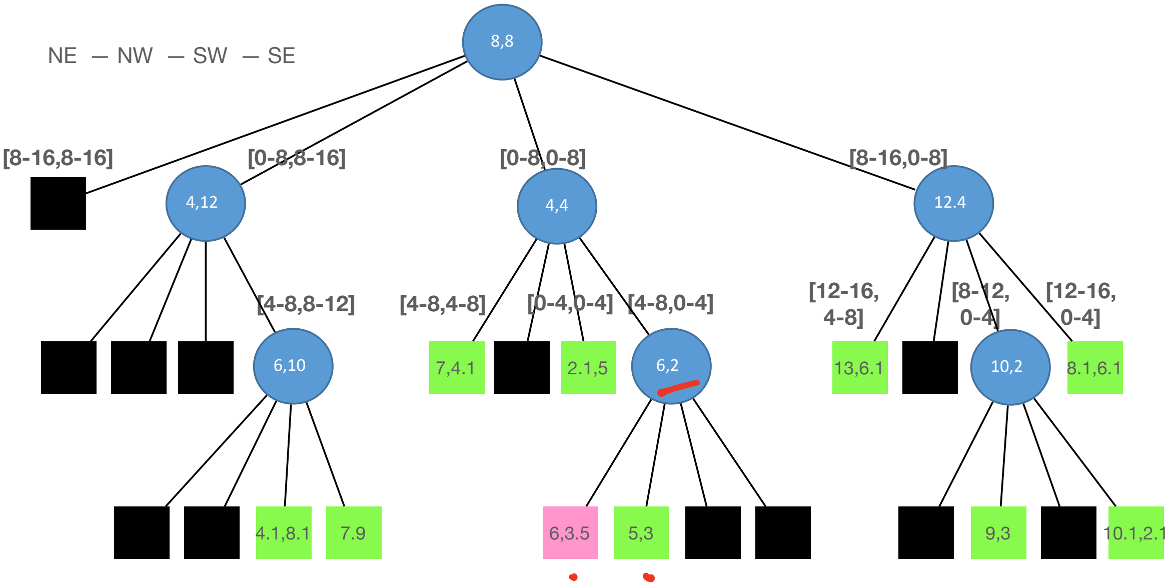
- Insert (6,3.5)

NE — NW — SW — SE



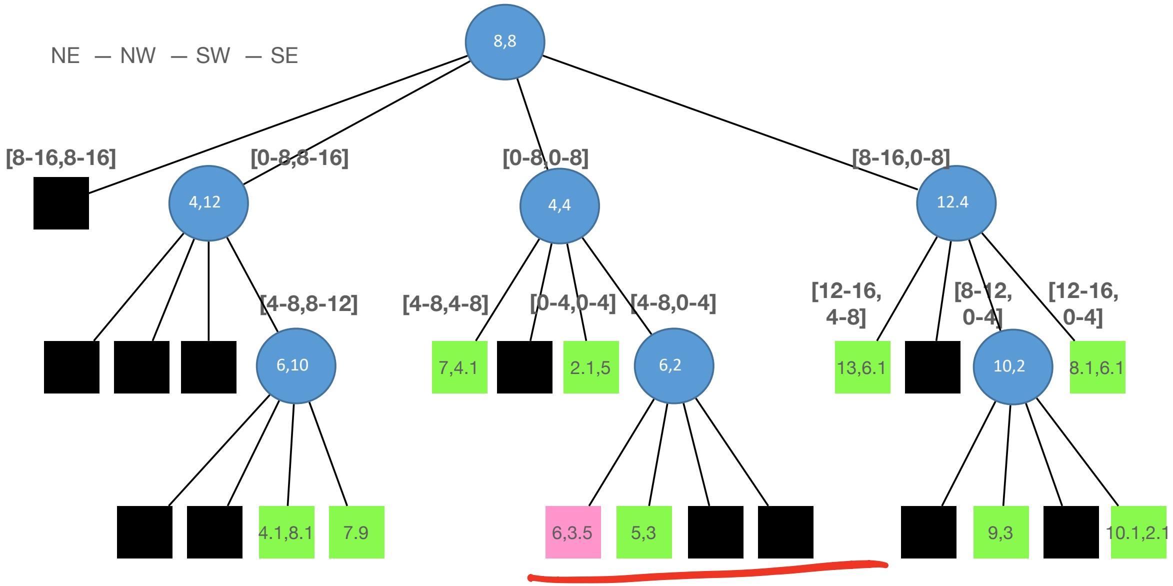
Insert Example - 2

- Insert (6,3.5)



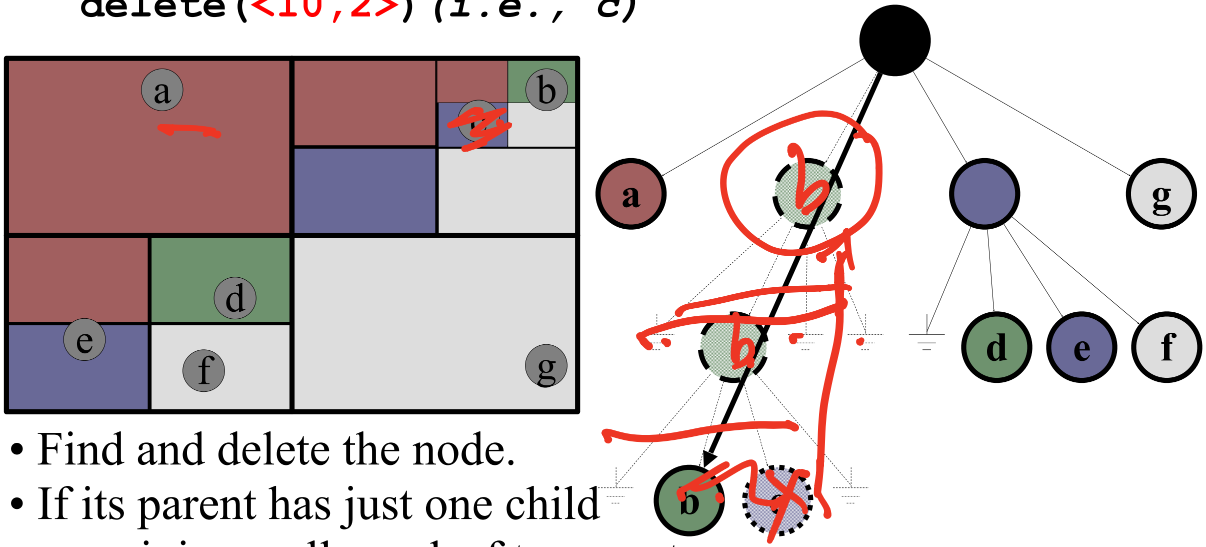
Insert Example - 2

- Insert (6,3.5)



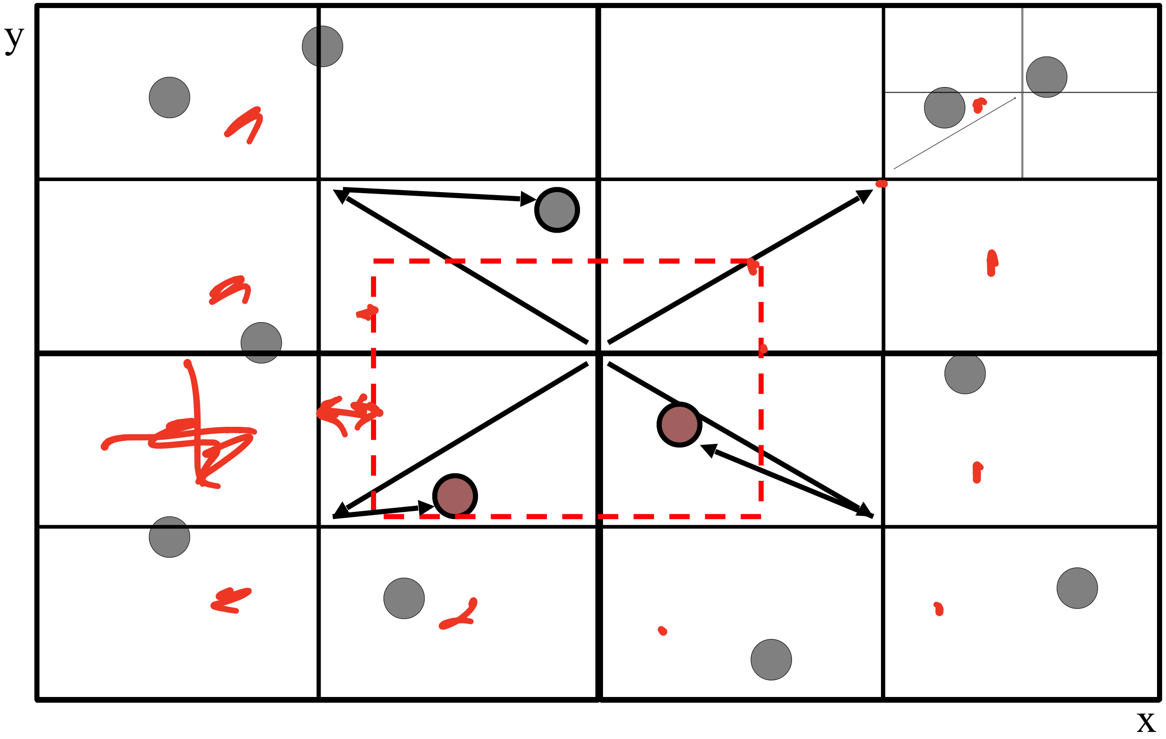
Delete Example

`delete(<10,2>) (i.e., c)`

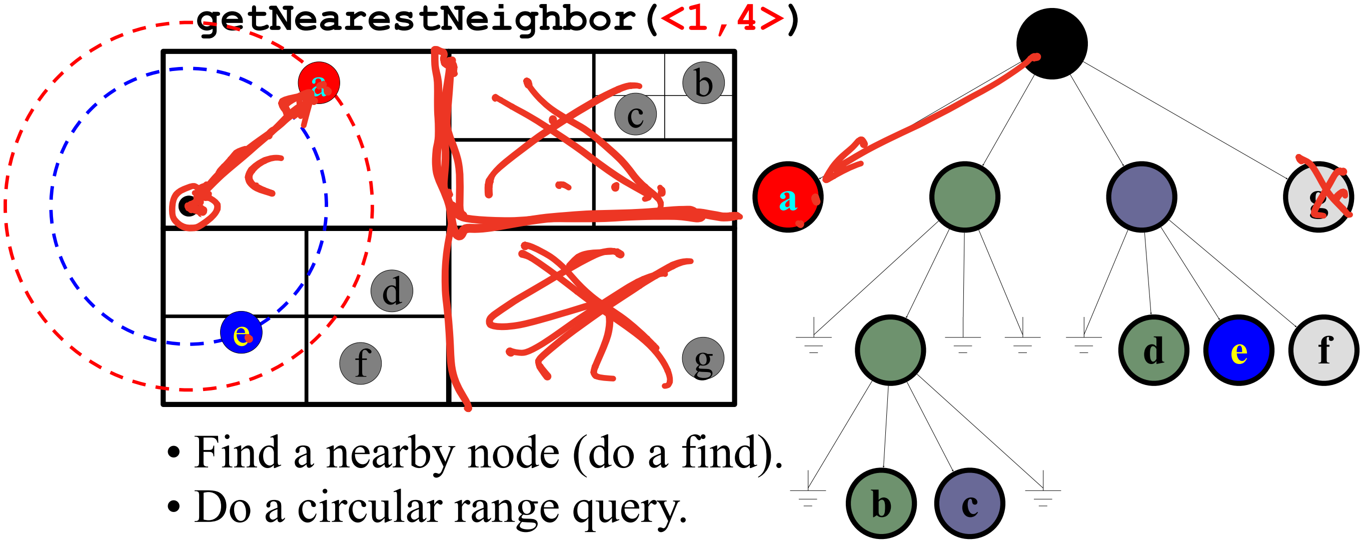


- Find and delete the node.
- If its parent has just one child remaining, collapse leaf to parent
- Propagate upward

2-D Range Querying in Quad Trees

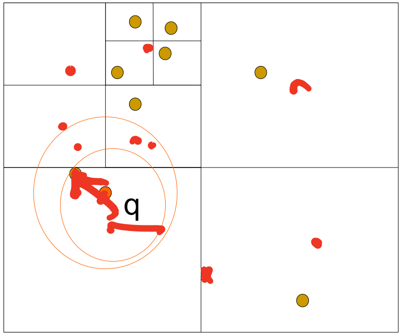


Nearest Neighbor Search



- Find a nearby node (do a find).
- Do a circular range query.
- As you get results, tighten the circle.
- Continue until no closer node in query.

Quadtree– Nearest Neighbor Search



Algorithm

Initialize range search with large r

Put the root on a stack

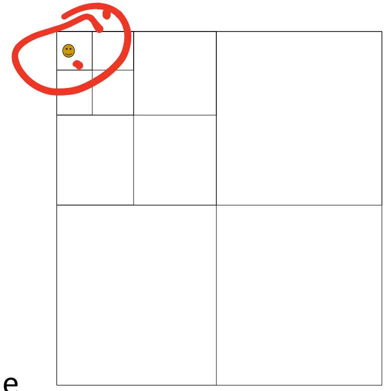
Repeat

- Pop the next node *T* from the stack
- For each child *C* of *T*
  - if *C* intersects with a circle (ball) of radius *r* around *q*, add *C* to the stack
  - if *C* is a leaf, examine point(s) in *C* and update r

- Whenever a point is found, update r (i.e., current minimum)
- Only investigate nodes with respect to current r.

[illegible]

Quadtree

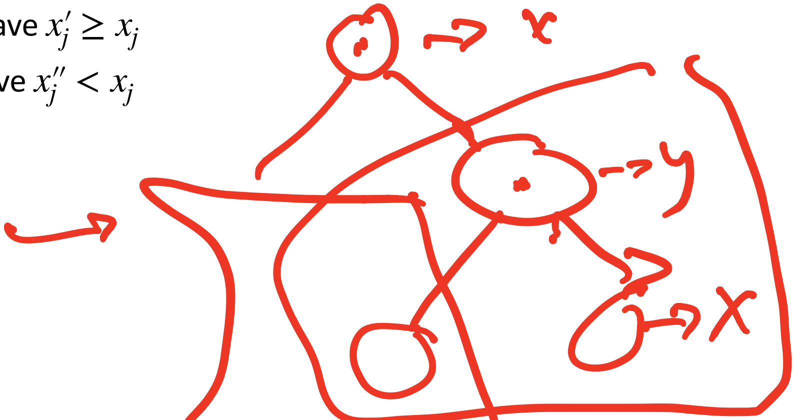


- Simple data structure.
- Easy to implement.
- But, it might not be efficient:
  - A quadtree could have a lot of empty cells
  - If the points form sparse clouds, it takes a while to reach nearest neighbors
- Want to consider partitions that are not fixed to key range (e.g. tries), but are done relative to key values (e.g. Search trees)

kd-trees (k-dimensional trees)

Main ideas:

- Generalize Binary Search Trees to k-dimensional data
- k-d tree: binary search tree where search decisions are made based on different coordinates at each level
  - Root is level 0
  - At level i, splitting decision is made based on coordinate (i mod k) + 1
- Property: node at i level use discriminator index j = (i mod k) + 1 with key value x_j
  - Right descendants $(x'_1, \ldots, x'_k)$ must have $x'_j \geq x_j$
  - Left descendants $(x''_1, \ldots, x''_k)$ must have $x''_j < x_j$



y

$\mathbf{2}$-dimensional kd-trees

A data structure to support nearest neighbor and rangequeries in **R^2.**

- Not the most efficient solution in theory.
- Everyone uses it in practice.

Algorithm: Batch construction

- Choose **x** or **y** coordinate (alternate).
- Choose the median of the coordinate; this defines a horizontal or vertical line. ✓
- Recurse on both sides until there is only one point left, which is stored as a leaf.

We get a binary tree

- Size O(n).
- Construction time O(nlogn) ✓
- Depth O(logn)
- K-NN query time: $O(n^{1/2}+k)$…under many assumptions

$$d=2 \quad O(n^{1-\frac{1}{2}}+k)$$

$$d=10 = O(n^{1-1/10}+k)$$

$\mathbf{d}$-dimensional kd-trees

- A data structure to support range queries in $\mathbf{R}^{d}$

- The construction algorithm is similar as in **2-d**

  At the root we split the set of points into two subsets of same size by a hyperplane vertical to x_1-axis.

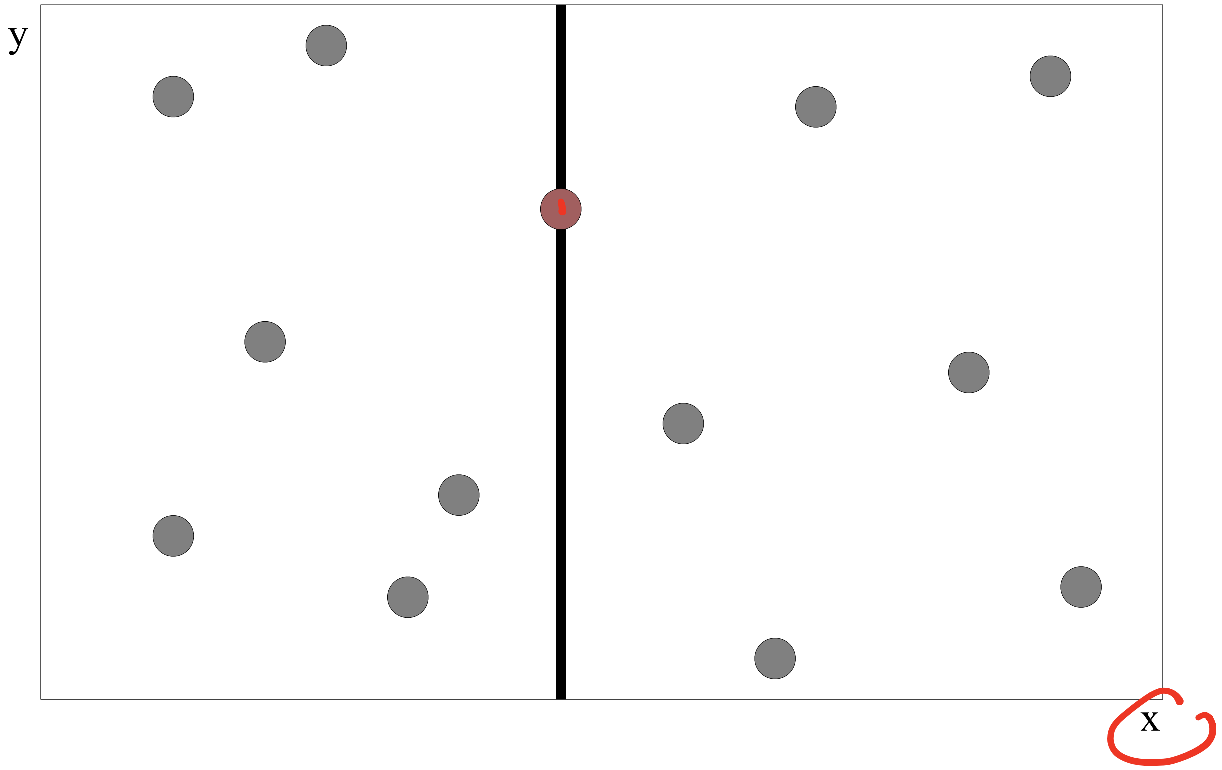
  At the children of the root, the partition is based on the second coordinate: x_2 Coordinate.

  At depth **d**, we start all over again by partitioning on the first coordinate.

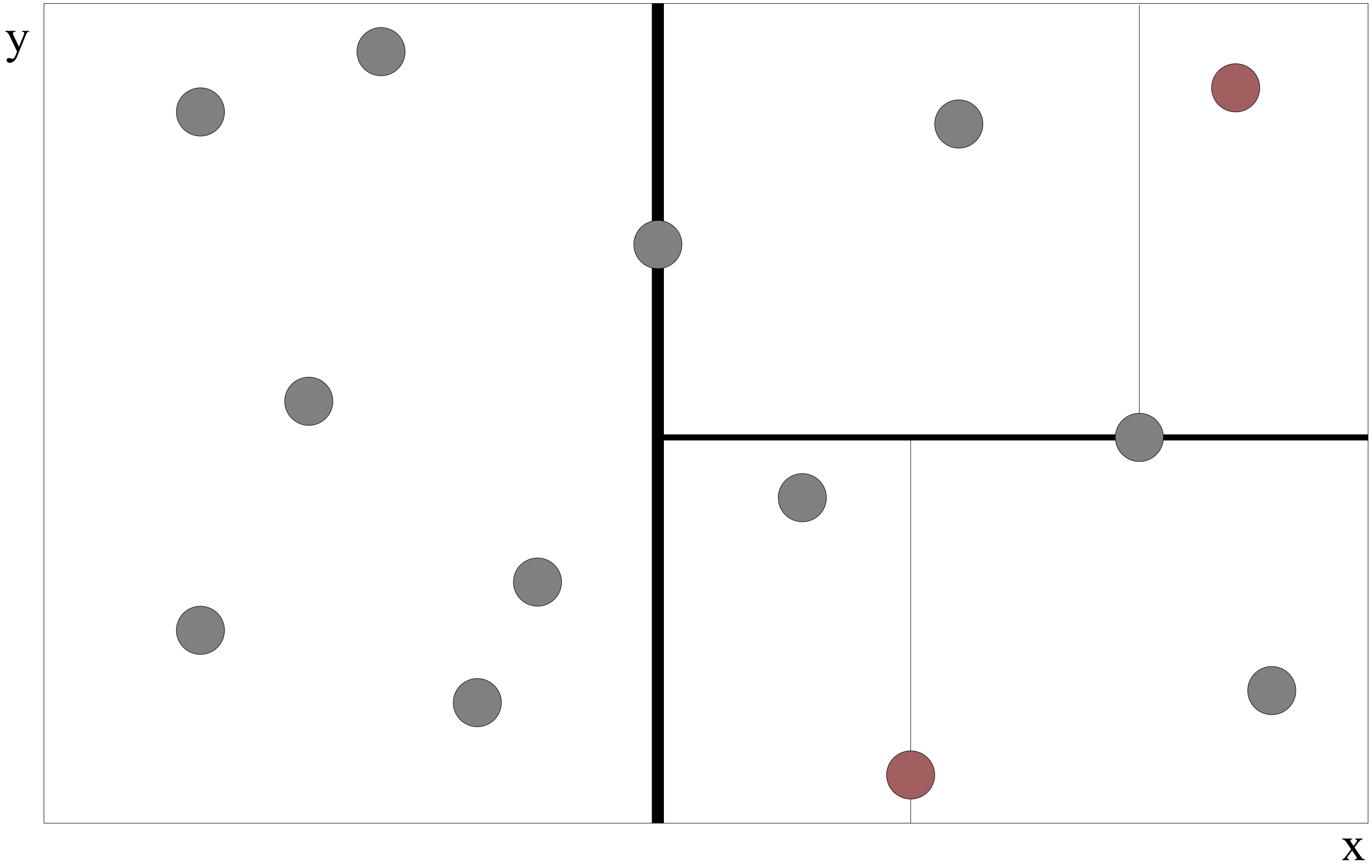
  The recursion stops until there is only one point left, which is stored as a leaf.

- Preprocessing time: **O(nlogn).**
- Space complexity: **O(n).**
- k-NN query time: $\underline{O(n^{1-1/d}+k)}$. ✓

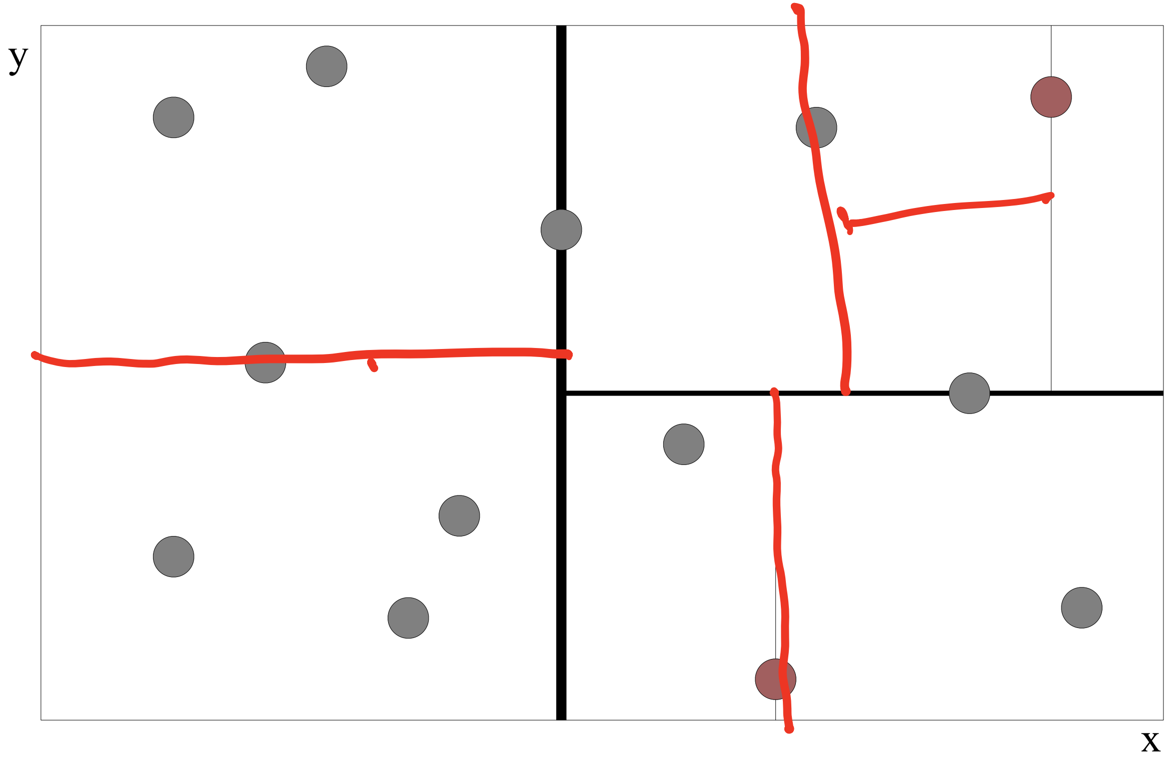
Building a 2-D Tree from Batch - 1



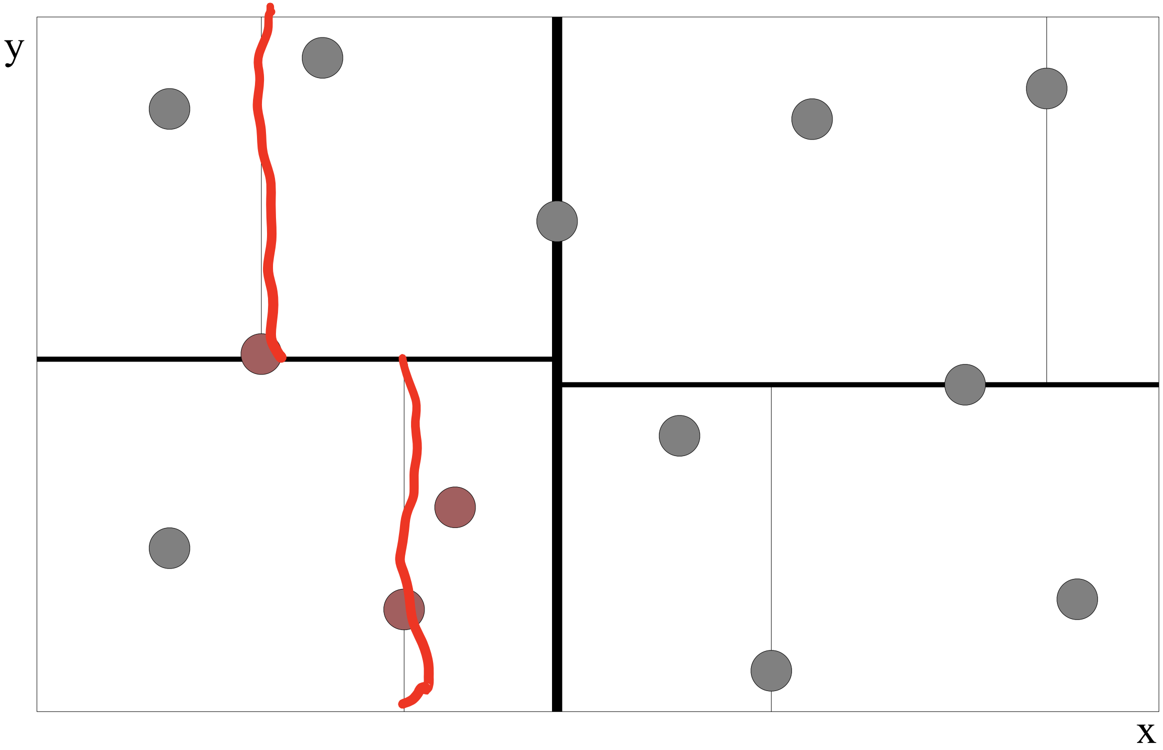
Building a 2-D Tree from Batch - 3



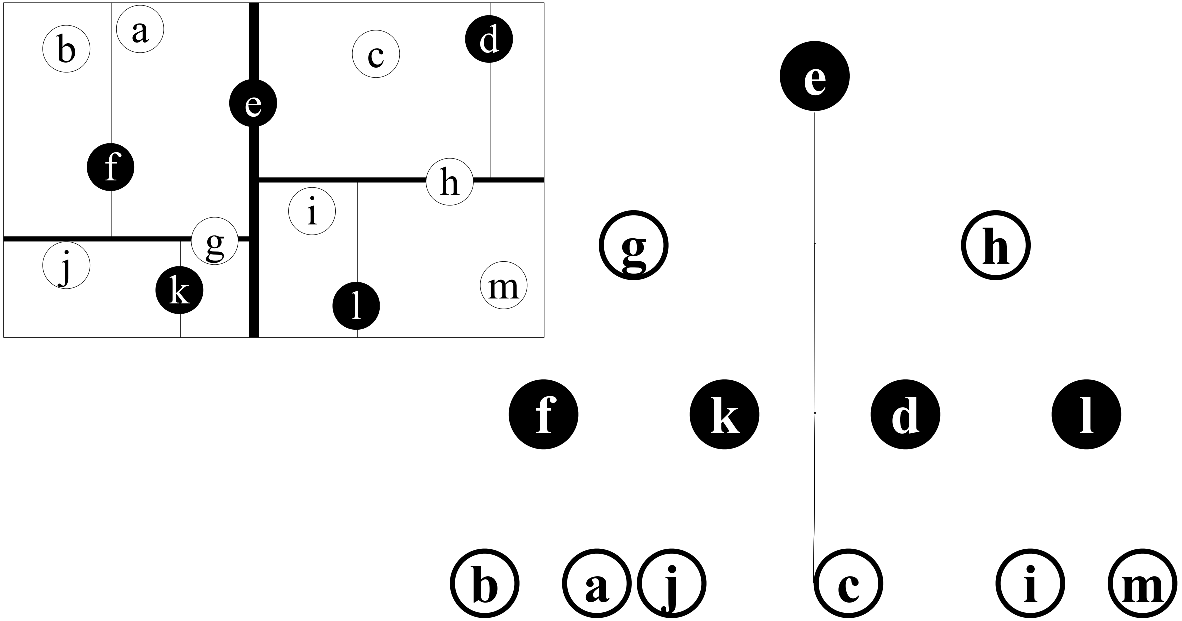
Building a 2-D Tree from Batch - 3



Building a 2-D Tree from Batch - 4



k-D Tree



Kd Trees Can Be Inefficient if built sequentially (but not when built in batch!)

insert(<5,0>)

insert(<6,9>)

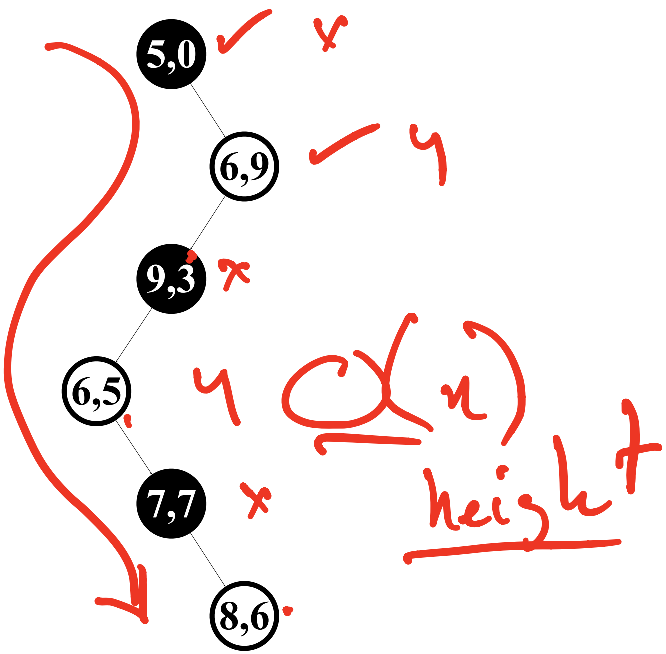
insert(<9,3>)

insert(<6,5>)

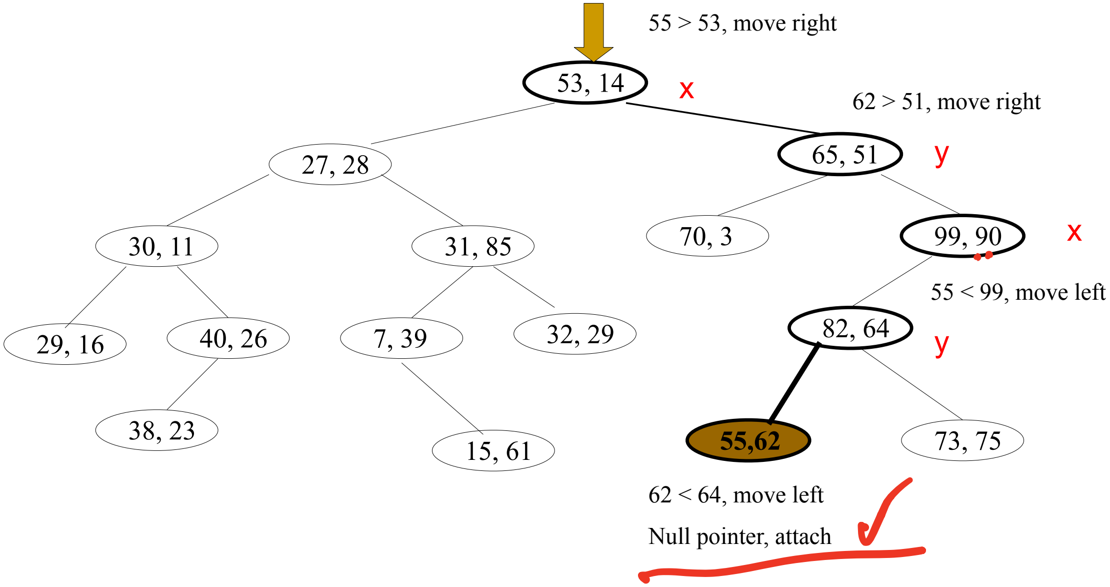
insert(<7,7>)

insert(<8,6>)

Incremental inserts not good…



Insert (55, 62)



55 > 53, move right

62 > 51, move right

55 < 99, move left

62 < 64, move left

Null pointer, attach ✓

Delete data ✓

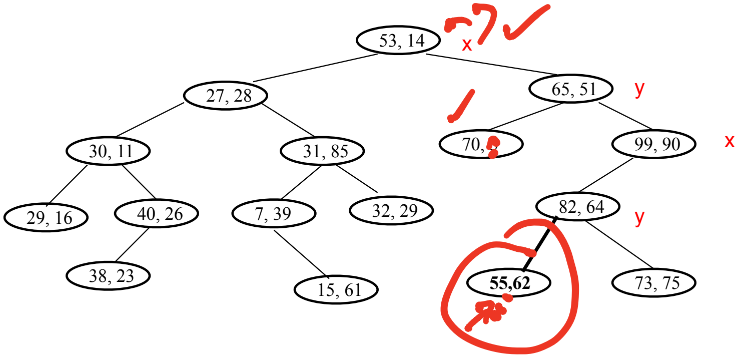
✓ Careful!

- Suppose we need to remove **p** = (a, b)
  - Find node **t** which contains **p**
  - If t is a leaf node, replace it by null
  - Otherwise, find a replacement node **r** = (c, d) – see below!
  - Replace (a, b) by (c, d)
  - Remove **r**
- Finding the replacement **r** = (c, d)
  - If **t** has a right child, use the successor*
  - Otherwise, use node with minimum value* in the left subtree
    - Move right child of that node as appropriate

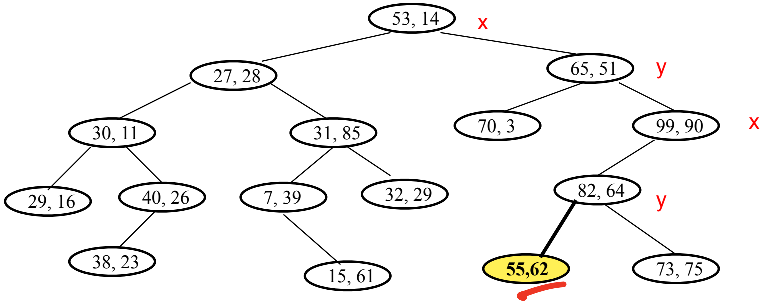
*(depending on what axis the node discriminates)

Delete data (cont’d)

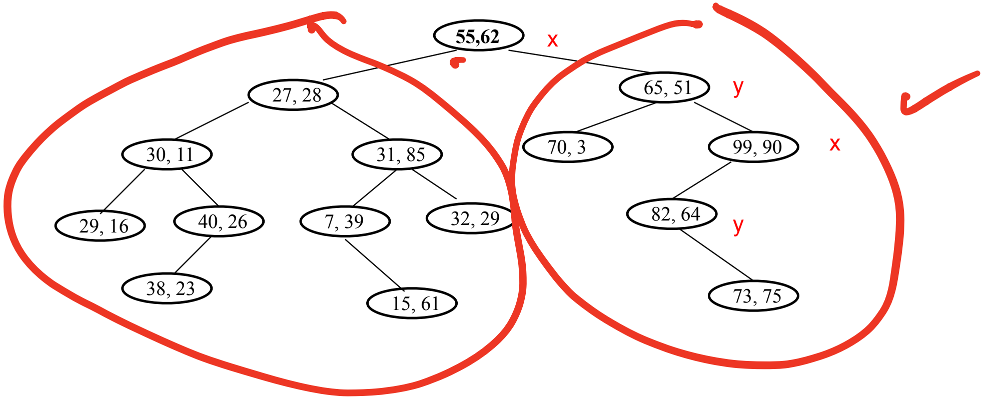
Delete (53,14)



Find min x on right

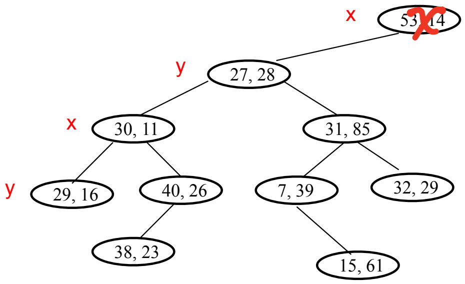


Swap it

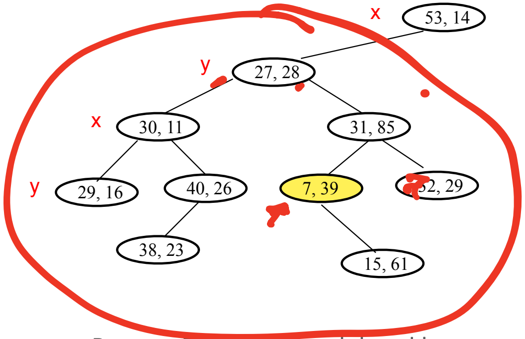


Delete data (cont'd)

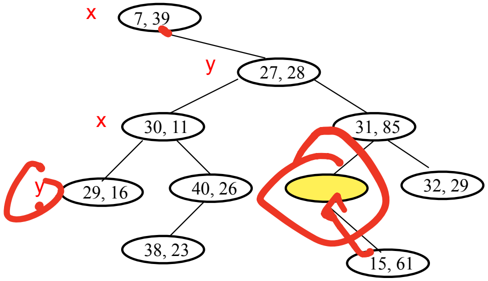
Delete (53,14)



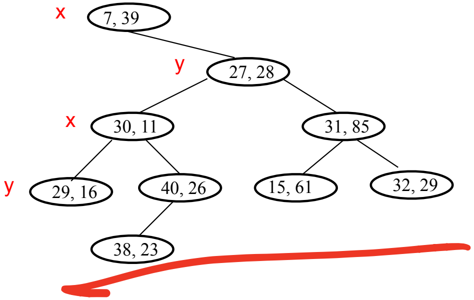
No right child, find MIN x on left



Swap with root, move left to right

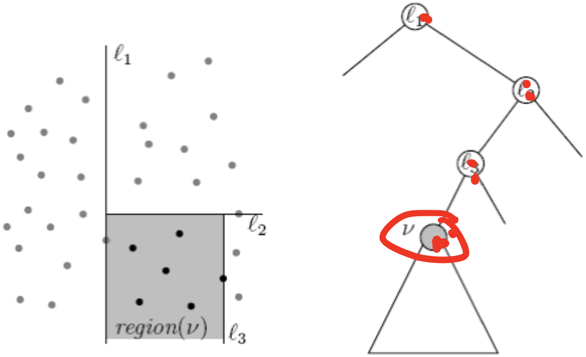


Repeat Process from deleted key



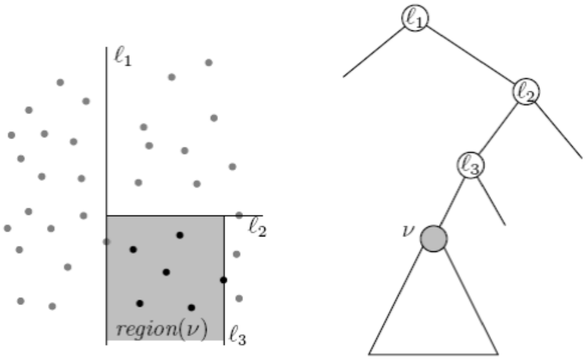
KD Tree – **Region of a node**

- The region *region(v)* corresponding to a node v is a rectangle, which is bounded by splitting lines stored at ancestors of v
- A point is stored in the subtree rooted at node v if and only if it lies in *region(v)*



KD Tree - **Region of a node** (cont’d)

- A point is stored in the subtree rooted at node v if and only if it lies in *region(v)*.



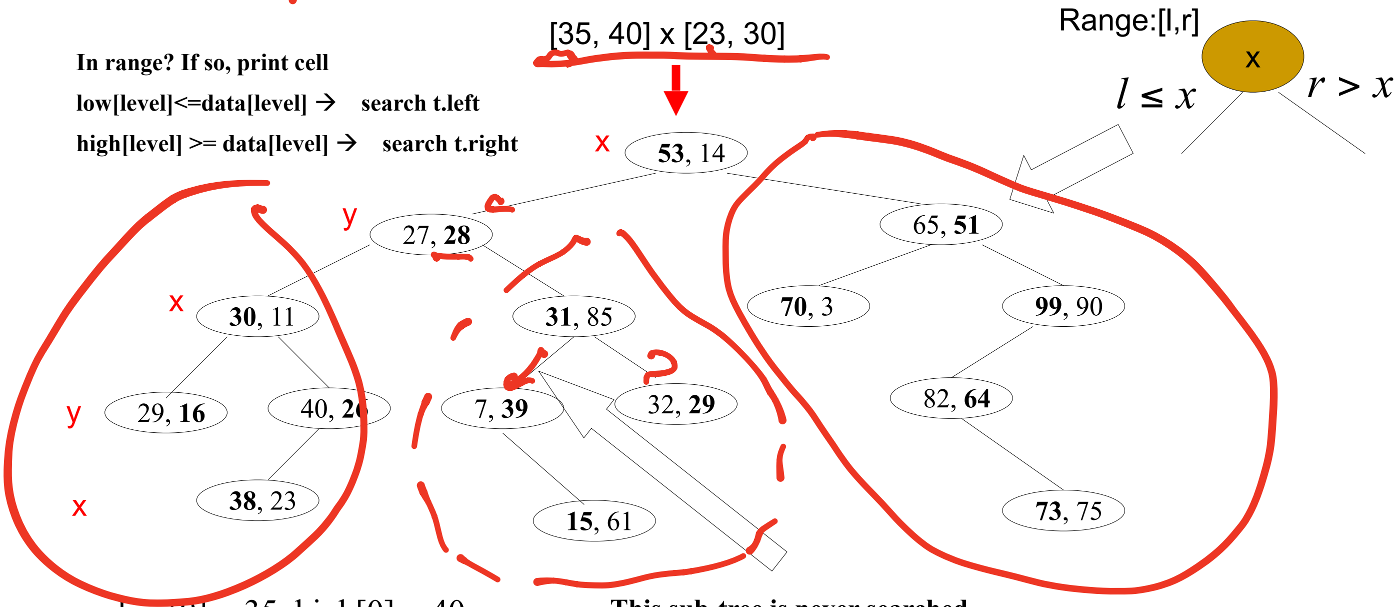
KD Tree - Range Search

[35, 40] x [23, 30]

In range? If so, print cell

low[level]<=data[level] → search t.left

high[level] >= data[level] → search t.right



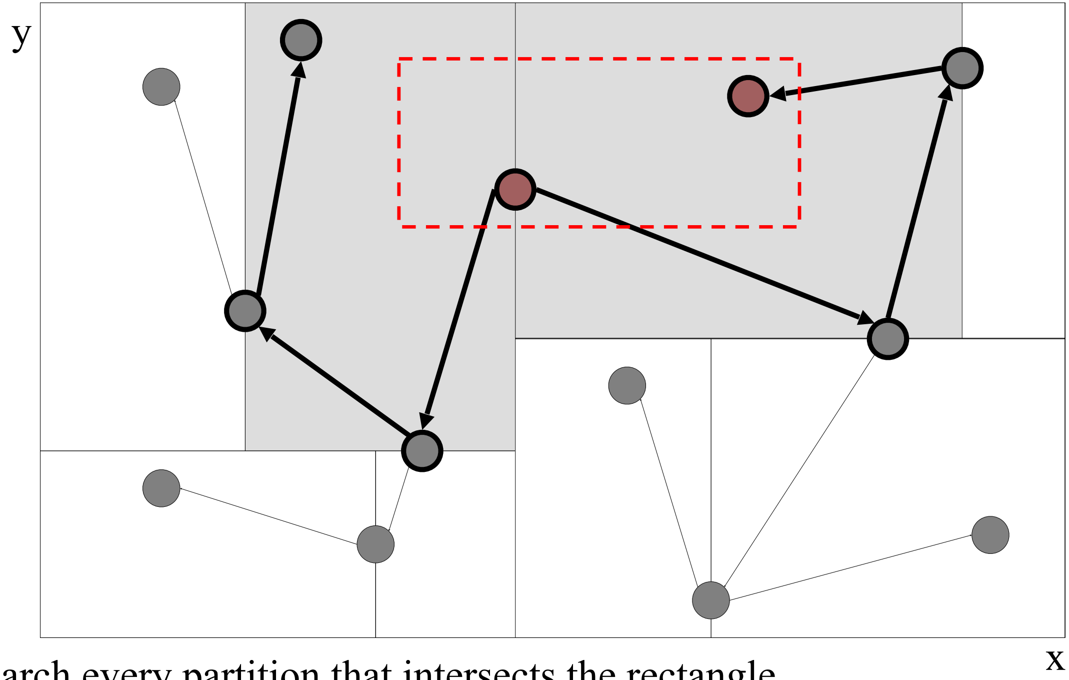
low[0] = 35, high[0] = 40;

low[1] = 23, high[1] = 30;

This sub-tree is never searched.

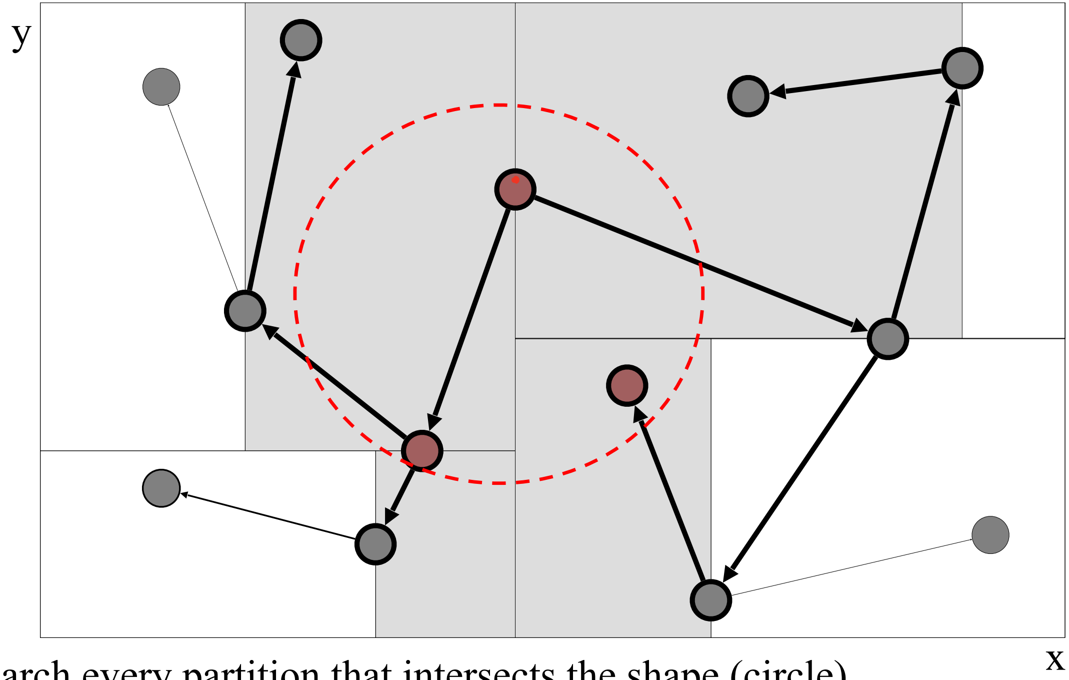
Searching is "preorder". Efficiency is obtained by "pruning" subtrees from the search.

2-D Range Querying in 2-D Trees



Search every partition that intersects the rectangle.
Check whether each node (including leaves) falls into the range.

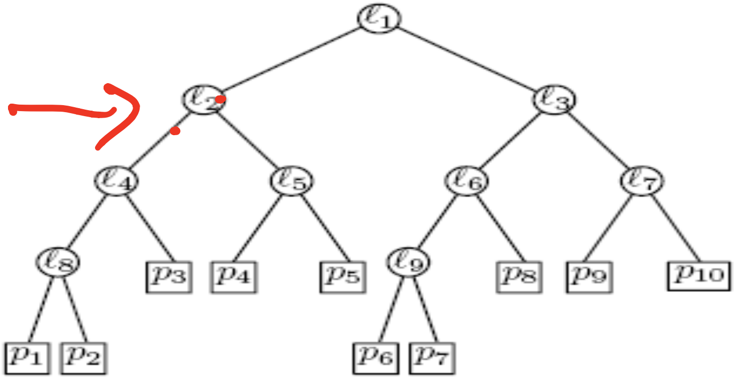
Other Shapes for Range Querying



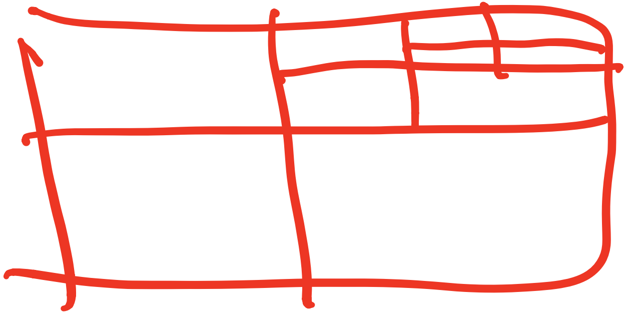
Search every partition that intersects the shape (circle).
Check whether each node (including leaves) falls into the shape.

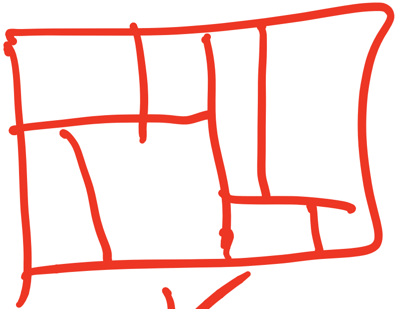
KD Tree Variation

- Data stored at leaves only
  - Navigation keys inside
  - Looks like quadtree, but with adaptive boundaries, balance
  - Similar to B vs B+ trees



QT





2DT.

2-10

Quad Trees vs. *k*-D Trees

$O(n^{1-1/d})$

$d \to 10+$ $O(n)$

- *k*-D Trees
  - Density balanced trees
  - Number of nodes is O(n) where *n* is the number of points
  - Height of the tree is O(log n) *with batch insertion*
  - Supports insert, delete, find, nearest neighbor, range queries
- Quad Trees
  - Number of nodes is O(n(1+ log(Δ/n))) where *n* is the number of points and Δ is the ratio of the width (or height) of the key space and the smallest distance between two points
  - Height of the tree is O(log n + log Δ)
  - Supports insert, delete, find, nearest neighbor, range queries

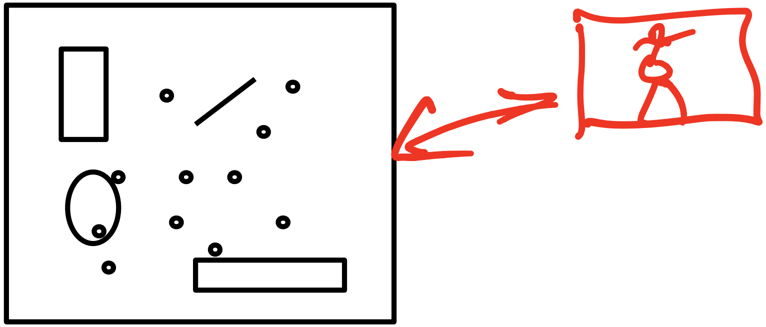
$10^9 \approx$ 27 levels! 10,000 ...

B+ trees

Quadtrees, kd Trees Good for Points

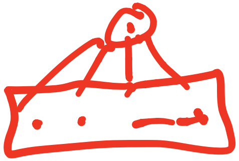
- What about shapes?

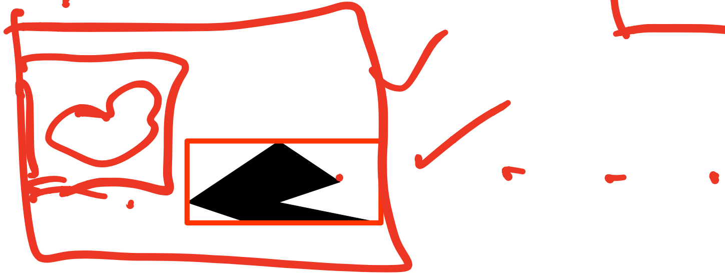
- Problem of Interest:
- Given a collection of geometric objects (points, lines, polygons, ...)
- organize them on disk, to answer efficiently spatial queries (range, nn, etc)



R-tree

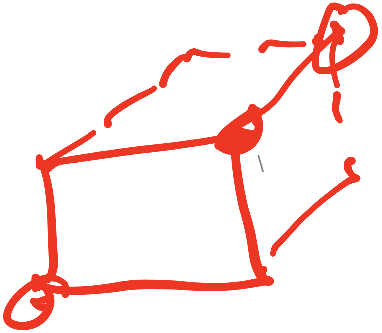
- In multidimensional space, there is no unique ordering! Not possible to use B+-trees☹
- [Guttman 84] R-tree!
- Group objects close in space in the same node
  - => guaranteed page utilization
  - => easy insertion/split algorithms.
    - (only deal with Minimum Bounding Rectangles - **MBR**s)

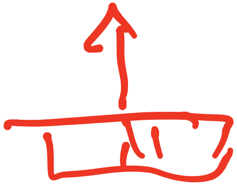




R-tree

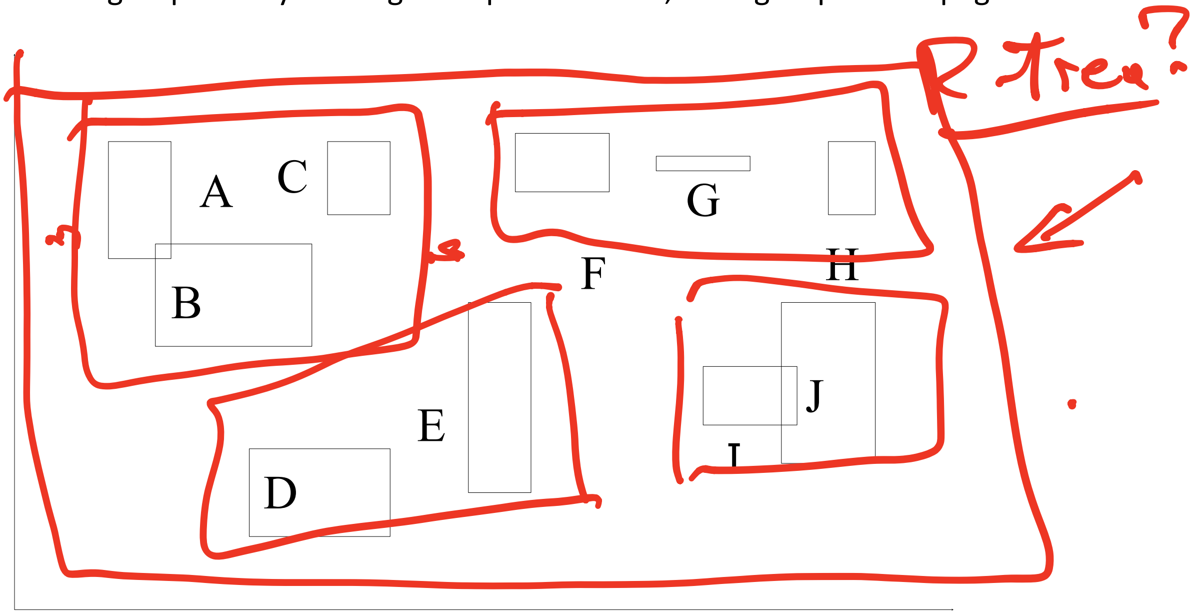
- A multi-way external memory tree
- Keys: n-dimensional rectangles, (2 points)
- Index nodes and data (leaf) nodes
- All leaf nodes appear on the same level
  - Leaf node index entries: (I, tuple_id)
  - Non-leaf node entry: (I, child_ptr)
- Every node contains between m and M entries
  - *m* ≤ *M*/2 is the minimum entries per node.
- The root node has at least 2 entries (children)





Example

eg., w/ fanout 4: group nearby rectangles to parent MBRs; each group -> disk page



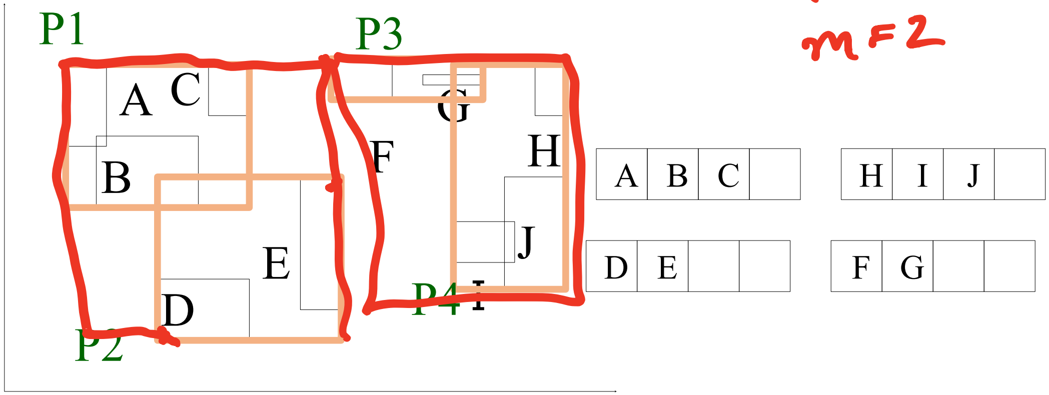
Example

- R trees grow like B+ trees
  - Bottom up

    eg., w/ fanout 4: group nearby rectangles to parent MBRs; each group -> disk page

M = 4

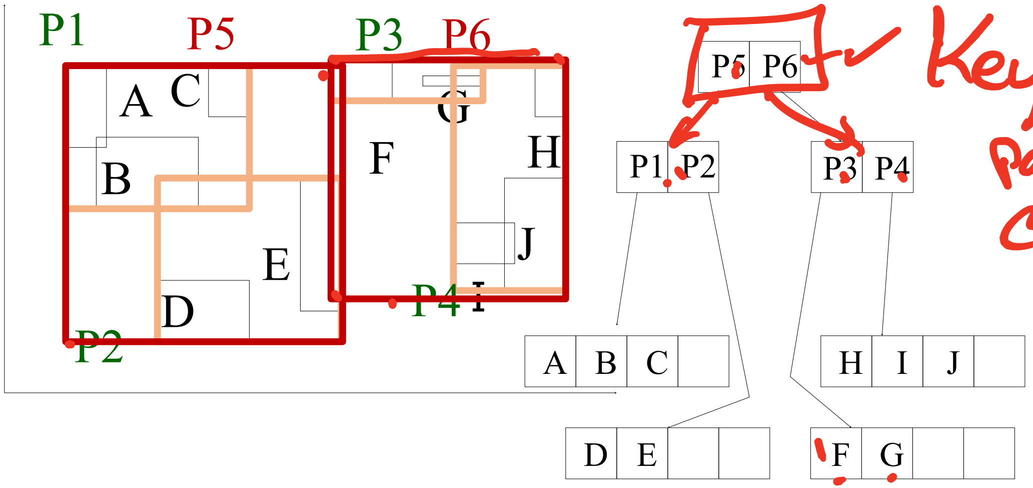
m = 2



Example

- R trees grow like B+ trees
  - Bottom up

    eg., w/ fanout 4: group nearby rectangles to parent MBRs; each group -> disk page



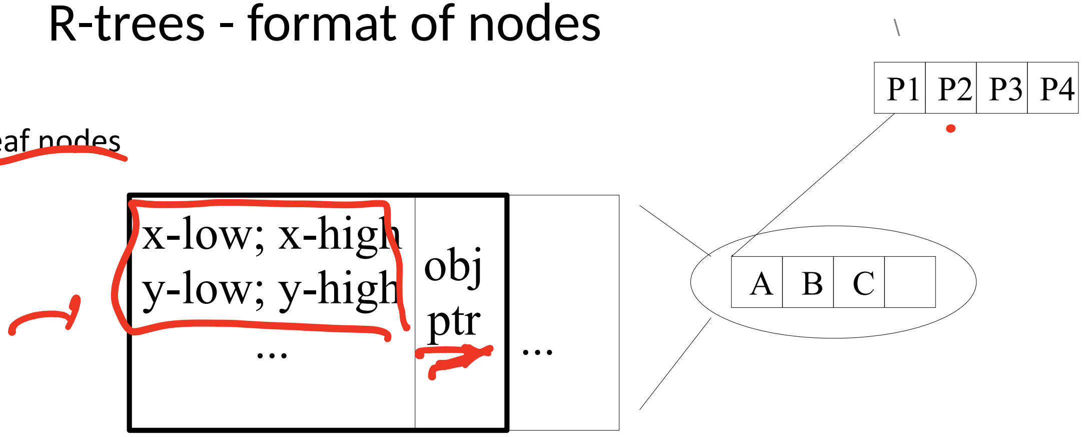
Keys are pairs of coordinates

B+ n-1 Keys ⇒ n ch

n rect. for n children

R-trees - format of nodes

- {(MBR; obj_ptr)} for leaf nodes



x-low; x-high
y-low; y-high
...

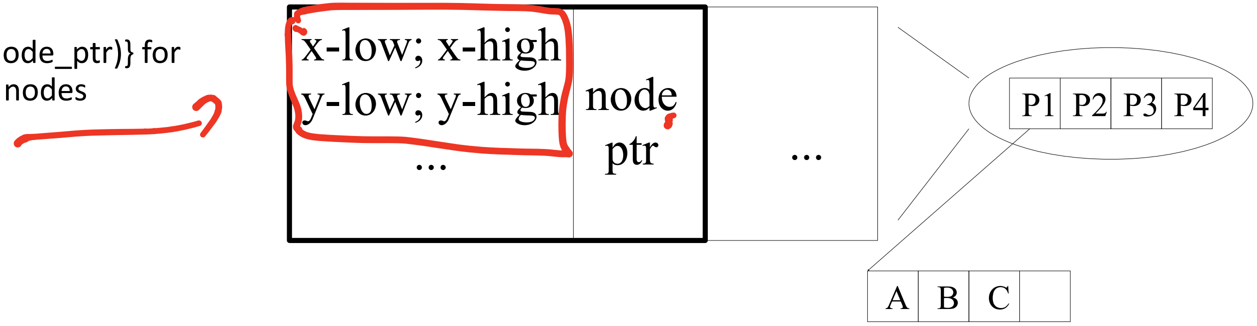
obj
ptr

...

A B C

P1 P2 P3 P4

- {(MBR; node_ptr)} for non-leaf nodes



x-low; x-high
y-low; y-high
...

node
ptr

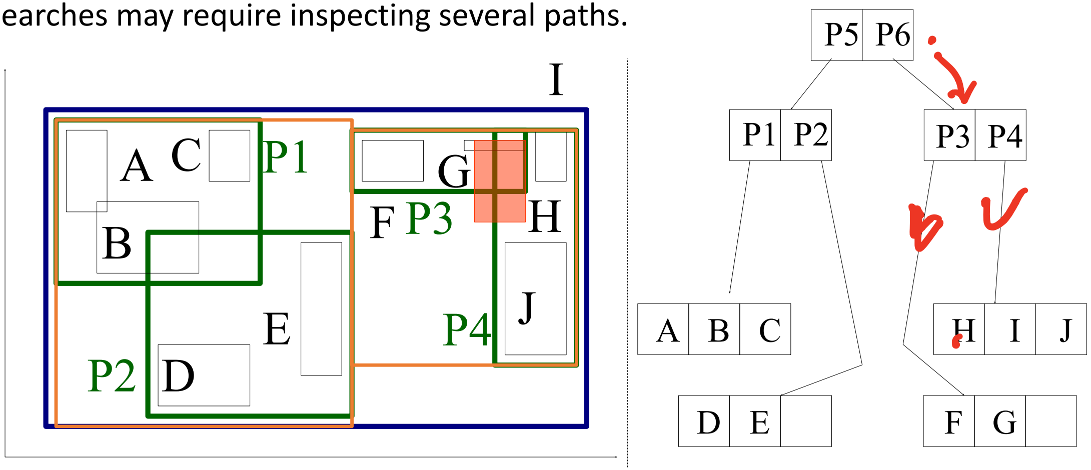
...

P1 P2 P3 P4

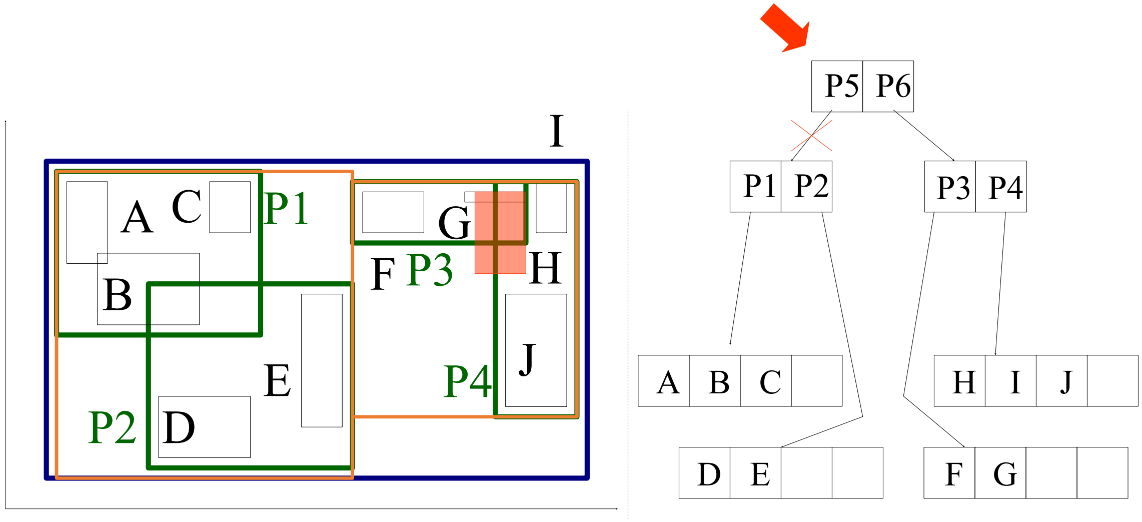
A B C

R-trees: Search

- Given a search rectangle S ...
  - Start at root and locate all child nodes whose rectangle I intersects S (via linear search).
  - Search the subtrees of those child nodes.
  - When you get to the leaves, return entries whose rectangles intersect S.
  - Searches may require inspecting several paths.



R-trees:Search

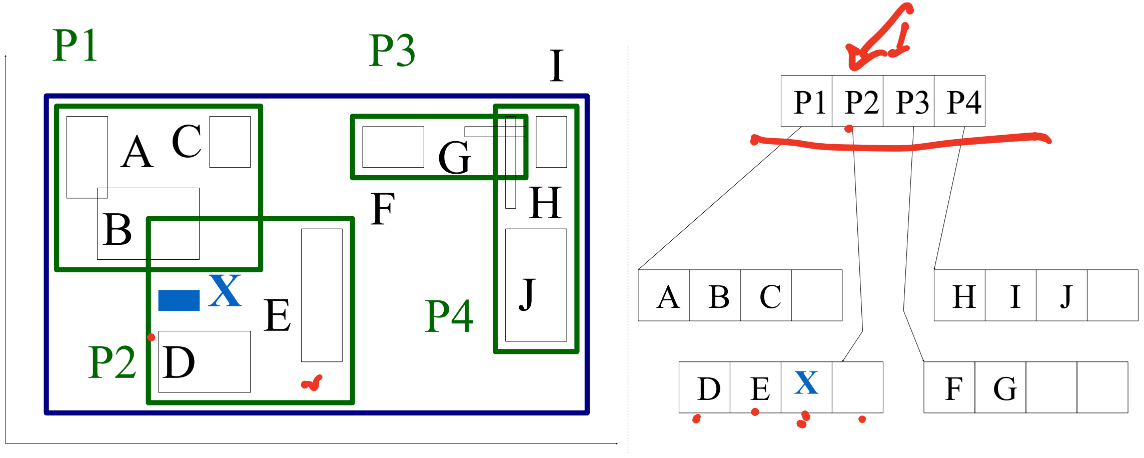


R-trees: Search

- Main points:
  - every parent node completely covers its ‘children’
  - nodes in the same level may overlap!
  - a child MBR may be covered by more than one parent - it is stored under ONLY ONE of them
  - a point query may follow multiple branches.
  - works for higher dimensions

R-trees:Insertion

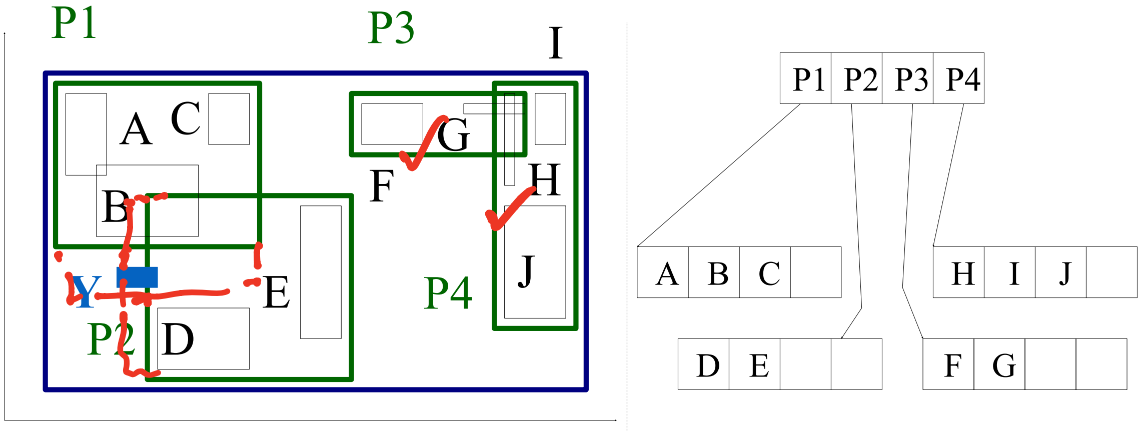
- Insert X: Start from the leaves. Which one?
  - Start at root
  - Go down the tree by choosing child whose rectangle needs the least enlargement to include X (Δ area or perimeter…) In case of a tie, choose child with smallest area
  - Least enlargement: increase in area or perimeter…a choice!



M=4 ,m=2

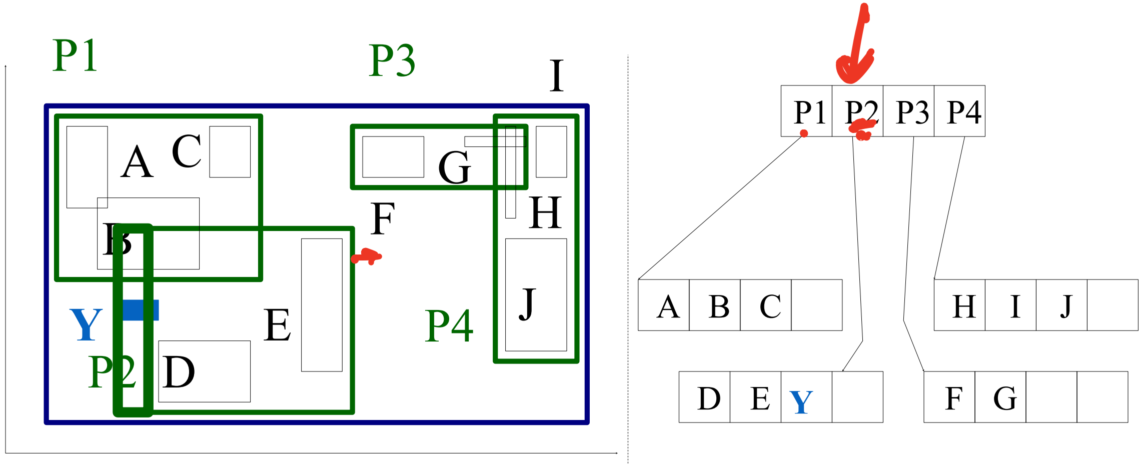
R-trees:Insertion

Insert Y



R-trees:Insertion

- Extend the parent MBR

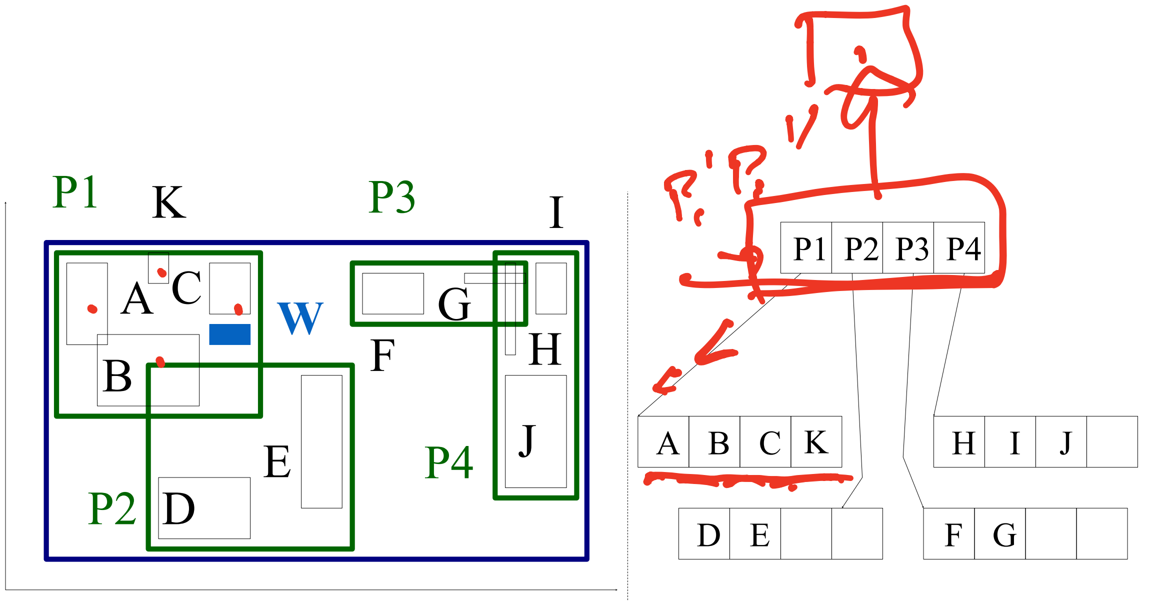


R-trees:Insertion

- How to find the next node to insert a new object Y?
  - Using ChooseLeaf: Find the entry that needs the least enlargement to include Y. Resolve ties using the area (smallest)
- Enlargement measured by change in perimeter of MBR or change in area
- Problem: Can saturate a leaf. In this case, need to ***split***
  - When you split, you readjust MBR in parent to correspond to remaining objects in each of the new nodes.
  - May need to recursively split parent…

R-trees:Insertion

- If node is full then <u>Split :</u> ex. Insert w



M=4
no room!

R-trees:Insertion

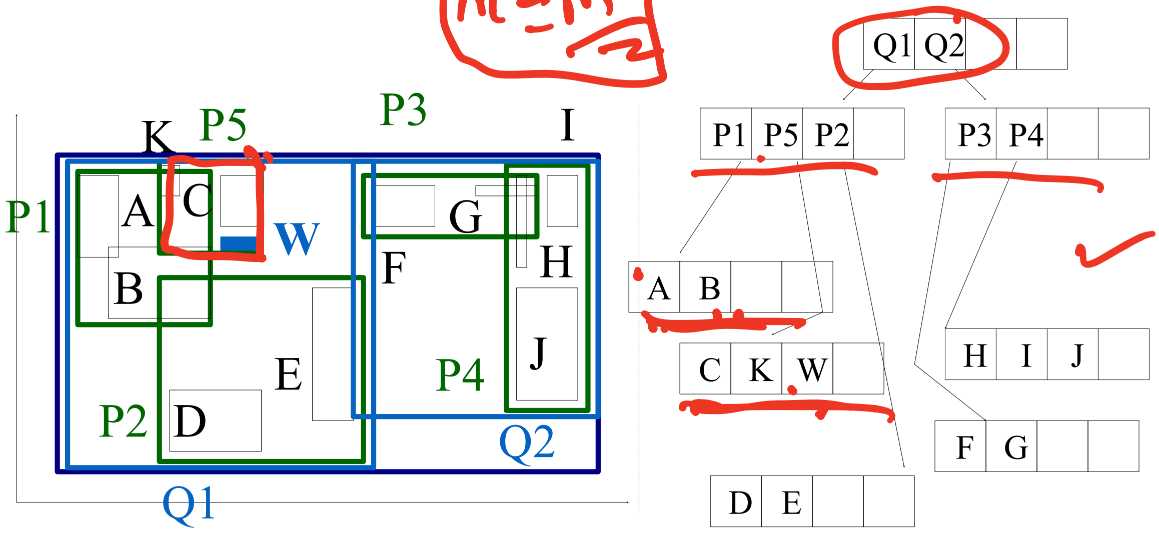
m → lower bound

M → upper bound

- If node is full then Split : ex. Insert w
- Note shrinkage of P_1

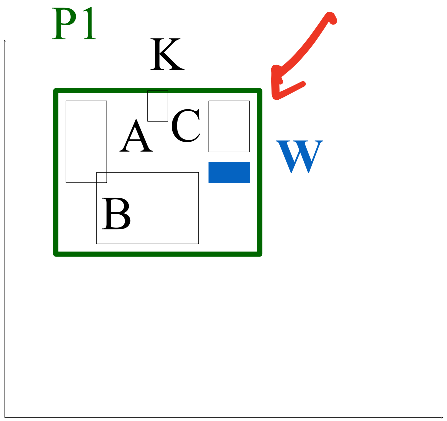
$m \leq \frac{M}{2}$

$m = 2$



R-trees:Split

- Split node P1: partition the MBRs into two groups.
- Multiple algorithms possible

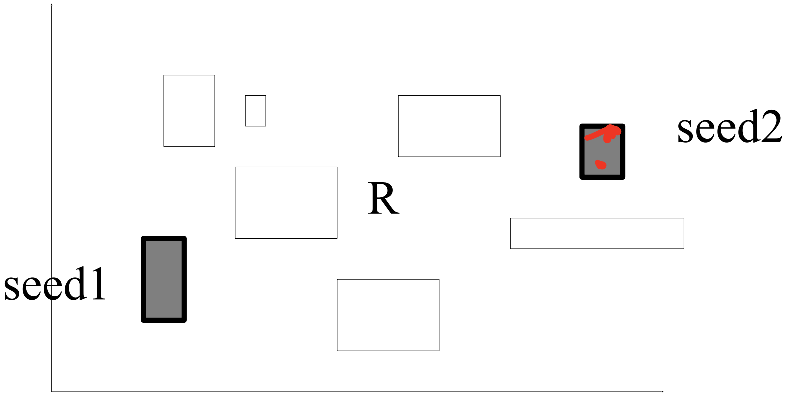


- A2: ‘linear’ split
- A3: quadratic split
- A4: exponential split:

  2^{M-1} choices

R-trees:Split

- Pick two rectangles as 'seeds'for group 1 and group 2
  - Farthest apart in dimension relative to total spread in dimension

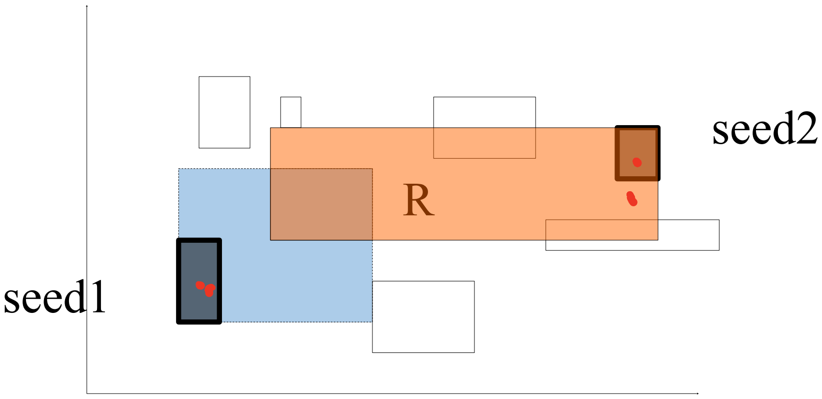


R-trees: Split

- pick two rectangles as ‘seeds’ for group 1 and group 2;
- assign each rectangle ‘R’ to the ‘closest’ ‘group’ in any order
- ‘closest’: the smallest increase in area
- Once a base rectangle has maximum number of rectangles for split, the rest are assigned to other rectangle: guarantee minimum m in both!

M=4
m=2

Max→3
mi:2



R-trees: Linear Split

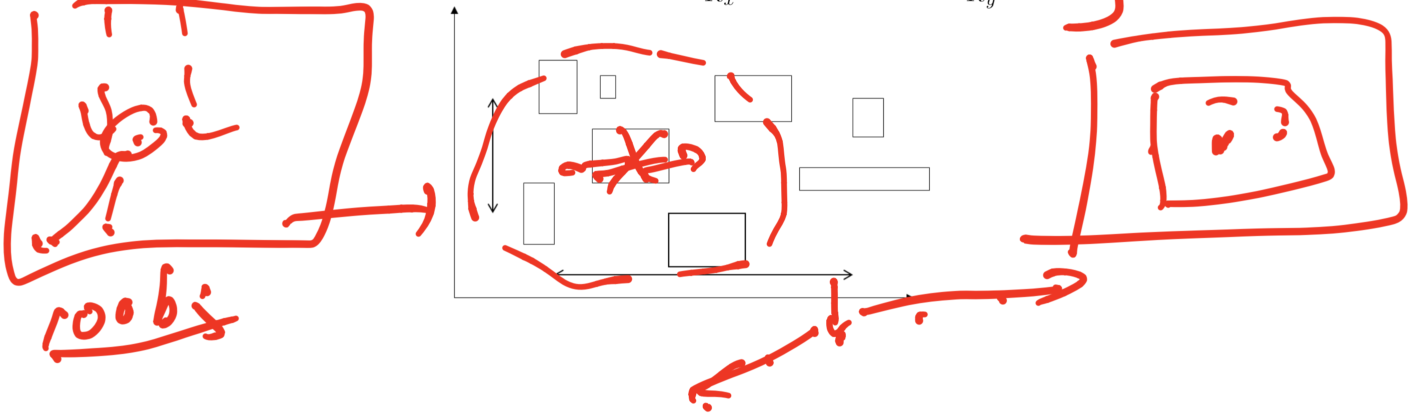
- How to pick Seeds:
  - Find the rects with the highest low and lowest high sides in each dimension
  - Normalize the separations by dividing by the width of all the rects in the corresponding dim
  - Choose the pair with the greatest normalized separation

$$\text{Rectangles } [(x_{low}^{(i)}, y_{low}^{(i)}), (x_{hi}^{(i)}, y_{hi}^{(i)})]$$

$$y_{hi} = \max_i y_{low}^{(i)}; \quad y_{low} = \min_i y_{hi}^{(i)}; \quad x_{low} = \min_i x_{hi}^{(i)}; \quad x_{hi} = \max_i x_{low}^{(i)}$$

$$R_y = \max_i y_{hi}^{(i)} - \min_i y_{low}^{(i)}; R_x = \max_i x_{hi}^{(i)} - \min_i x_{low}^{(i)}$$

$$\text{Normalized Separation: } NS(x) = \frac{x_{hi} - x_{low}}{R_x}; \quad NS(y) = \frac{y_{hi} - y_{low}}{R_y}$$



R-trees: Quadratic Split

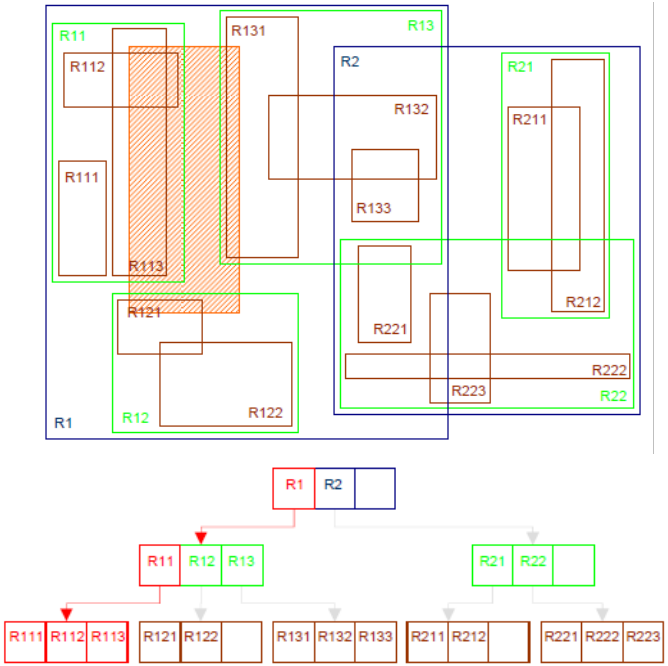
- How to pick Seeds:
  - For each pair E1 and E2, calculate the rectangle J=MBR(E1, E2) and d= J-E1-E2. Choose the pair with the largest d
- PickNext:
  - For each remaining rectangle E, calculate the area increase to include it in group d1(E) and d2(E)
  - Choose the remaining rectangle to insert with highest difference:

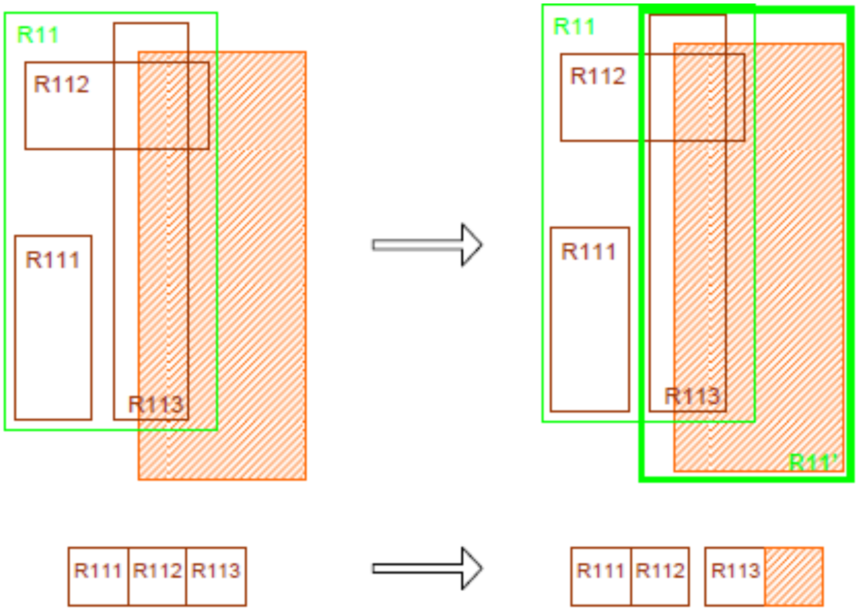
    |d1(E)-d2(E)|
  - Assign this remaining rectangle to its closest group: the one that has the smallest area increase.
  - Repeat until all rectangles are assigned, or until one group has M-m+1 entries. In the latter case, put the remaining rectangles into the other group and stop. If all rectangles have been distributed then stop.

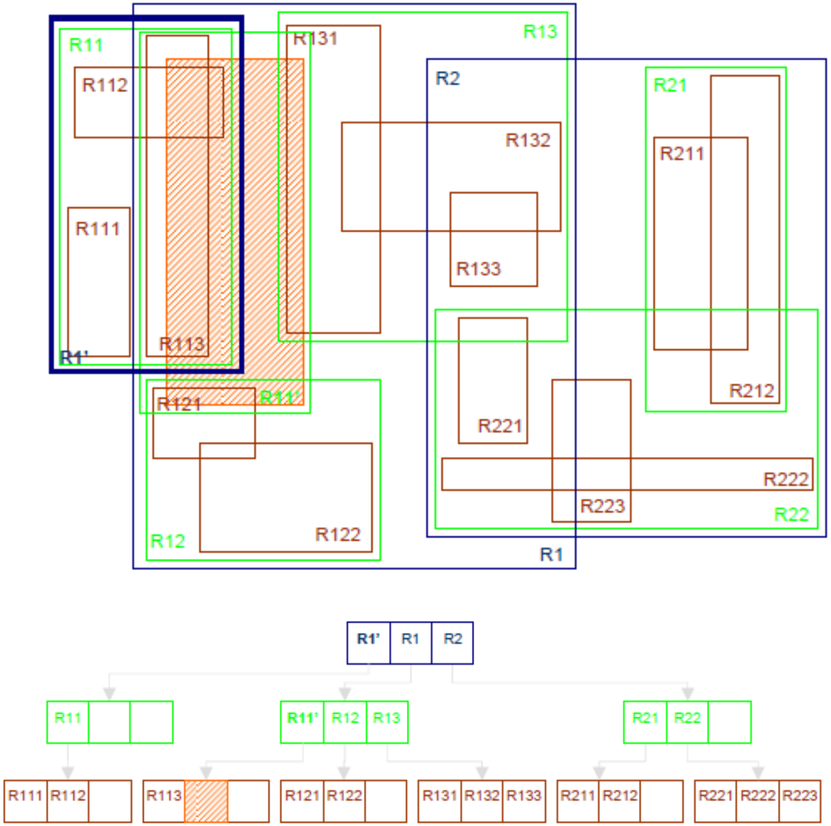
R-Trees:Deletion

- Find the leaf node that contains the entry E
- Remove E from this node
- If underflow:
  - Eliminate the node by removing the node entries and the parent entry
  - Reinsert the orphaned (other entries) into the tree using **Insert**
- Why reinsert?
  - Nodes can be merged with sibling whose area will increase the least, or entries can be redistributed.
  - Reinsertion is easier to implement.

Insertion example

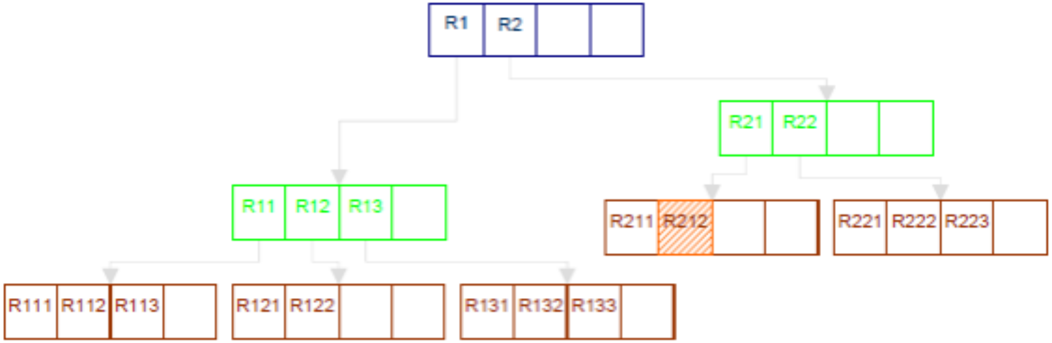



R11
R112
R111
R113
R11
R112
R111
R113
R11'
R111
R112
R113
R111
R112
R113

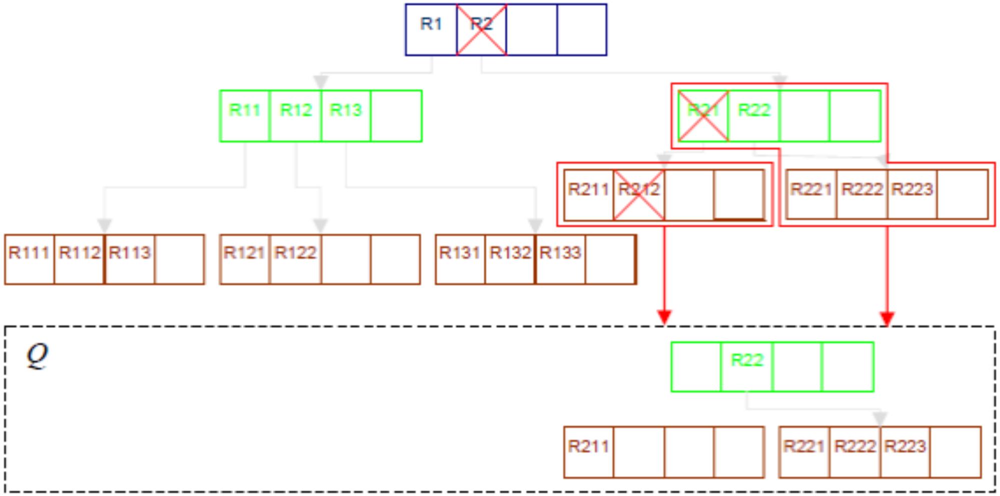

R11
R112
R111
R113
R1'
R131
R13
R2
R132
R133
R21
R211
R212
R121
R11'
R221
R222
R223
R22
R12
R122
R1
R1'
R1
R2
R11
R11'
R12
R13
R21
R22
R111
R112
R113
R121
R122
R131
R132
R133
R211
R212
R221
R222
R223

R-Trees: Deletion

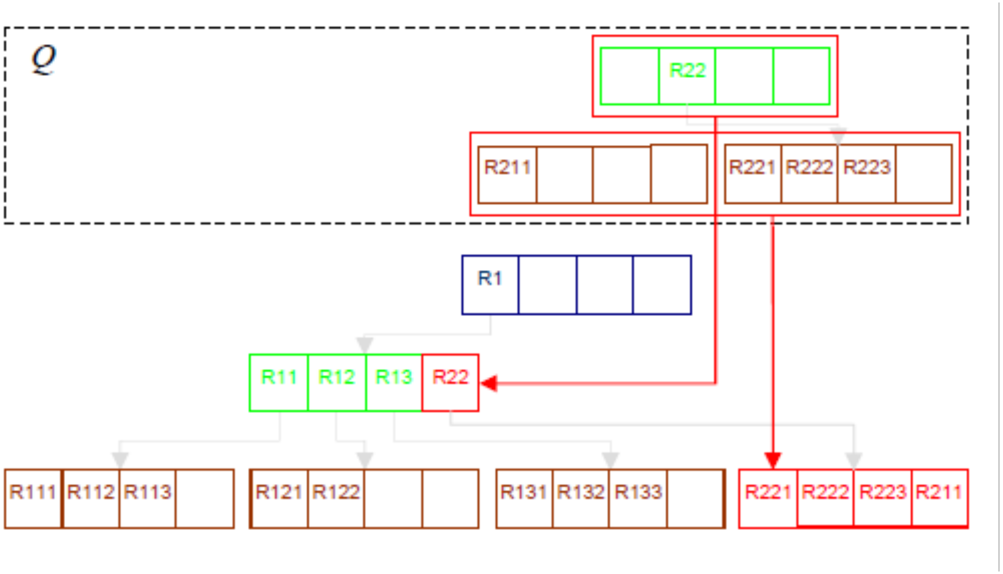
- M = 4, m = 2; Delete R212 which creates underflow in R21.
- Will delete and add R211 (orphan) to reinsert Queue
- Have underflow in R2. Will delete R2 and add intermediate node R2 to reinsert queue. Add the entries in R22 for reinsertion.



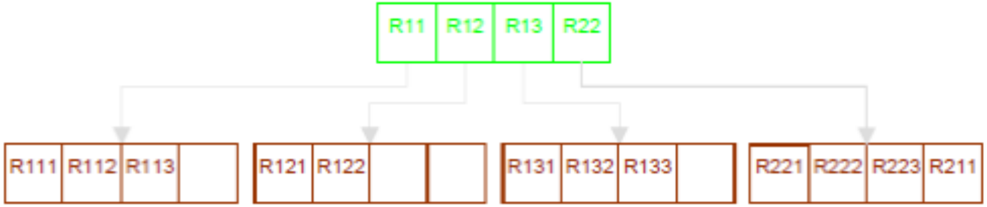
R-Trees: Deletion



R-Trees: Deletion



R-Trees: Deletion



R11 R12 R13 R22

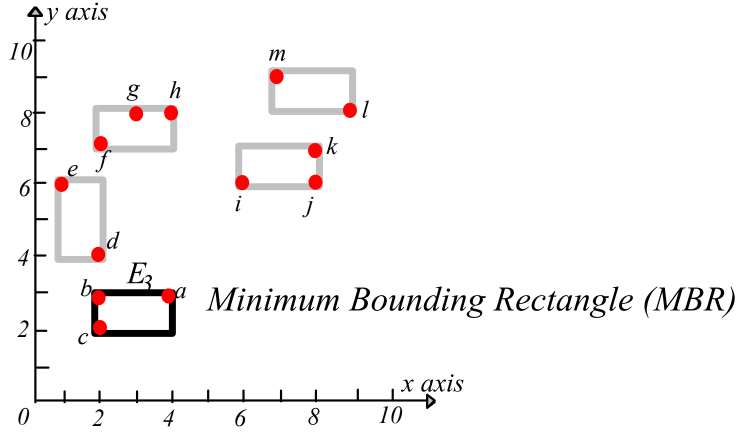
R111 R112 R113

R121 R122

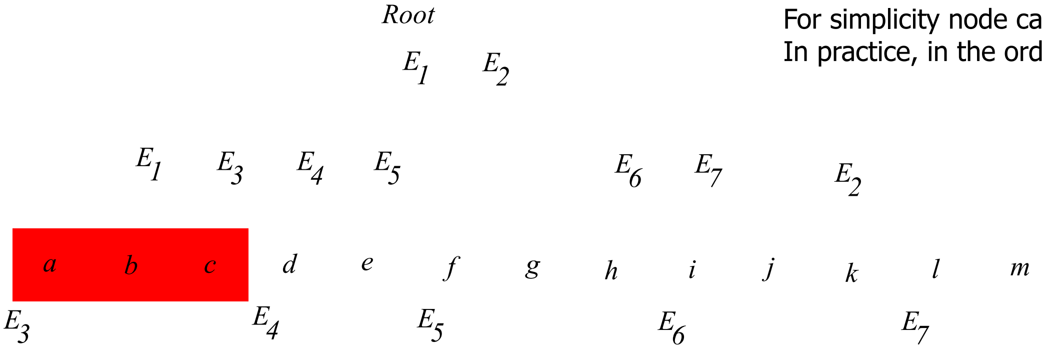
R131 R132 R133

R221 R222 R223 R211

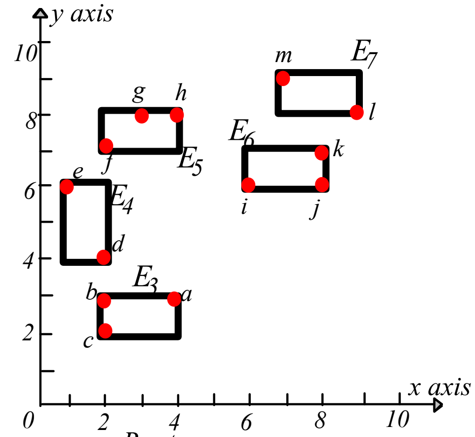
R-trees with point data

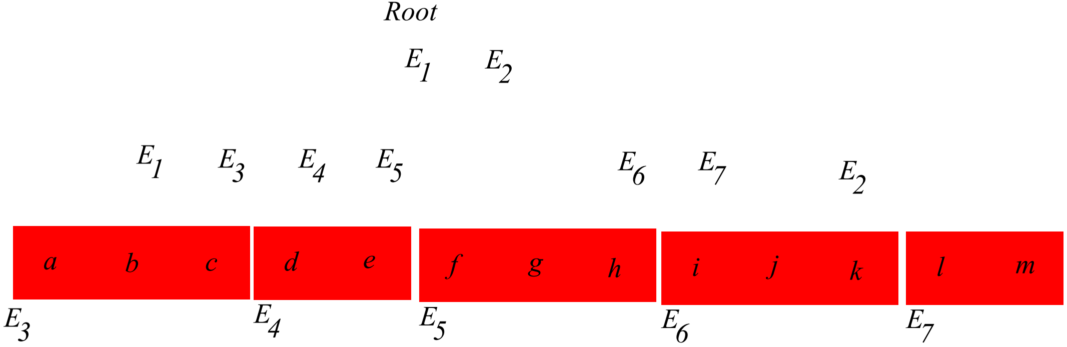


For simplicity node capacity = 3
In practice, in the order of 100

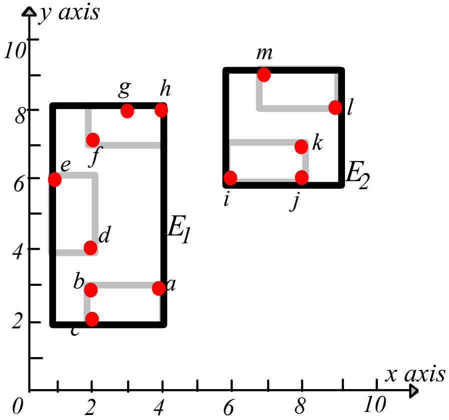


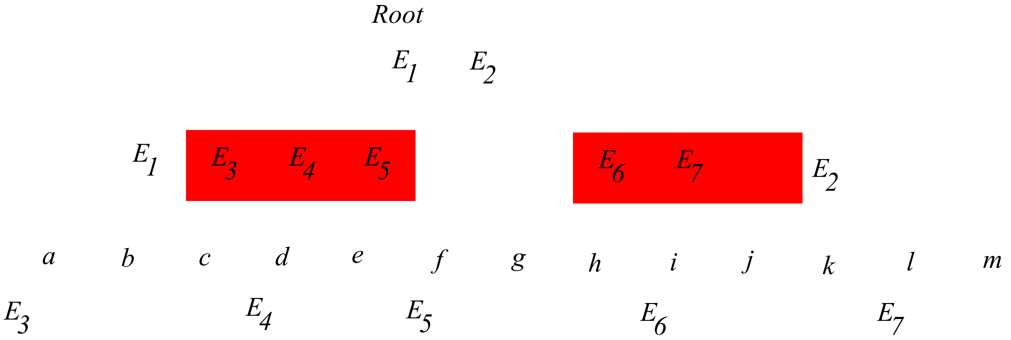
R-Tree, Leaf Nodes





R-Tree – Intermediate Nodes




Root

E_1 E_2

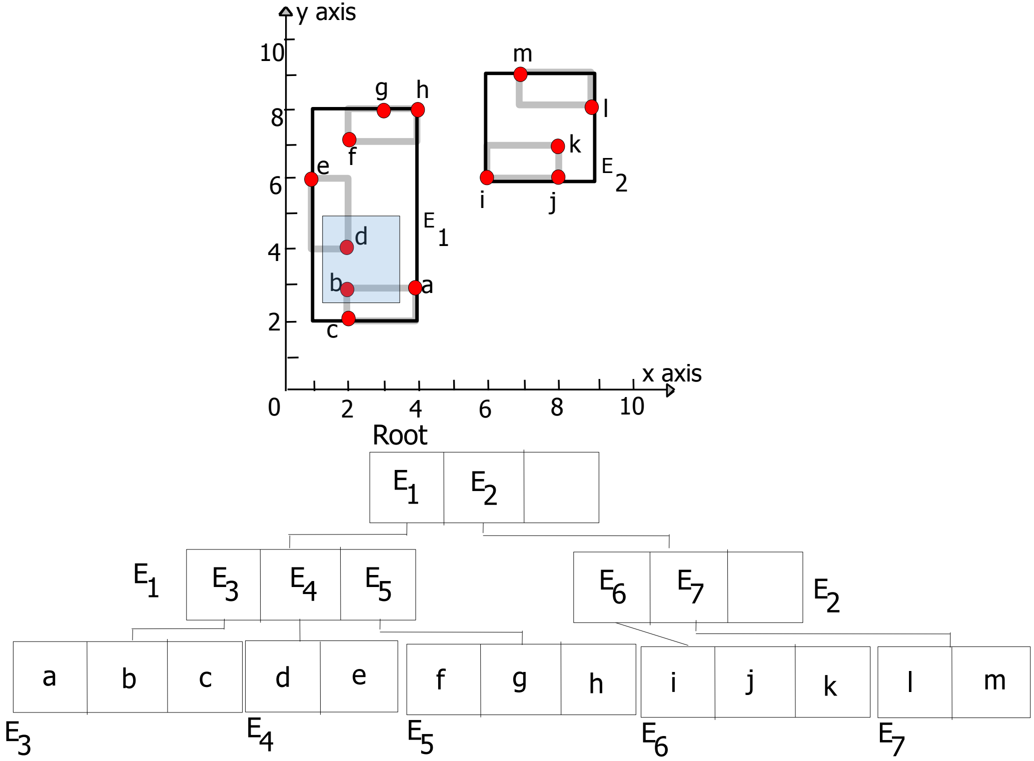
E_1 E_3 E_4 E_5

E_6 E_7 E_2

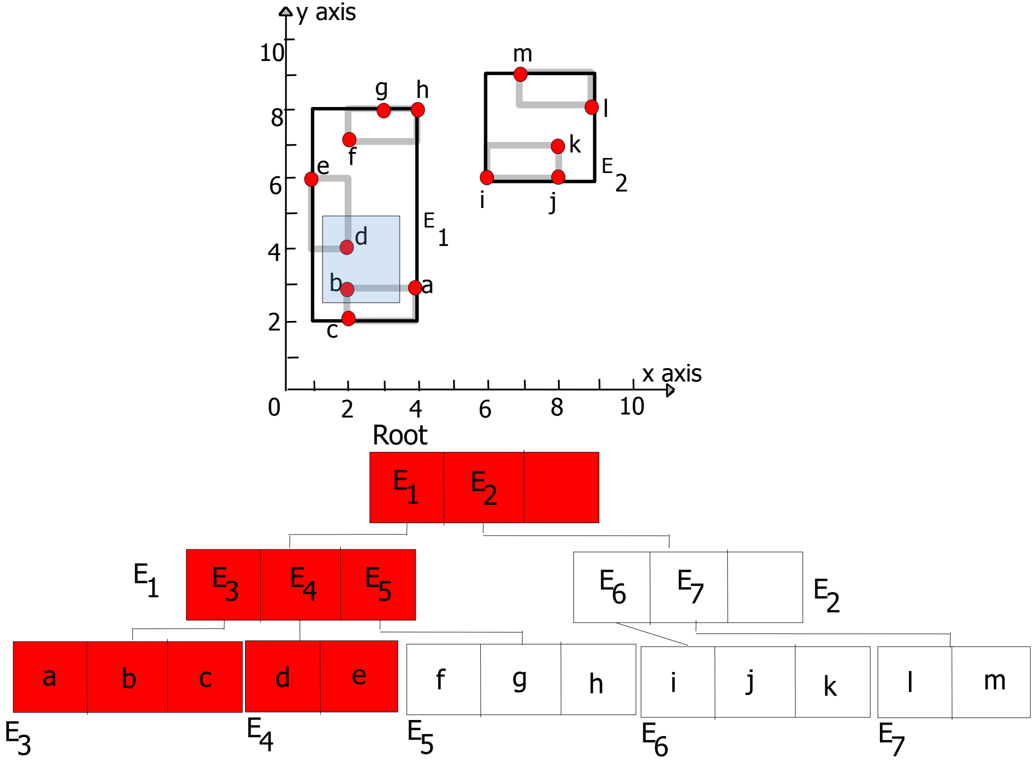
a *b* *c* *d* *e* *f* *g* *h* *i* *j* *k* *l* *m*

E_3 E_4 E_5 E_6 E_7

R-tree, Range Query



Range Query


y axis
10
8
6
4
2
0
2
4
6
8
10
x axis
a
b
c
d
e
f
g
h
i
j
k
l
m
E 1
E 2
Root
E1
E2
E1
E3
E4
E5
E6
E7
E2
a
b
c
d
e
f
g
h
i
j
k
l
m
E3
E4
E5
E6
E7

R-trees: Variations

- R+-tree: DO not allow overlapping, so split the objects (similar to z-values)
- R*-tree: change the insertion, deletion algorithms (minimize not only area but also perimeter, forced re-insertion)