Quadtrees

Aditya Trivedi : 190101005

Atharva Vijay Varde: 190101018

Index

2.1 Quadtrees

- Intro
- Types
- Grid Structure
- Use case

Demo

2.2 Compressed Quadtrees

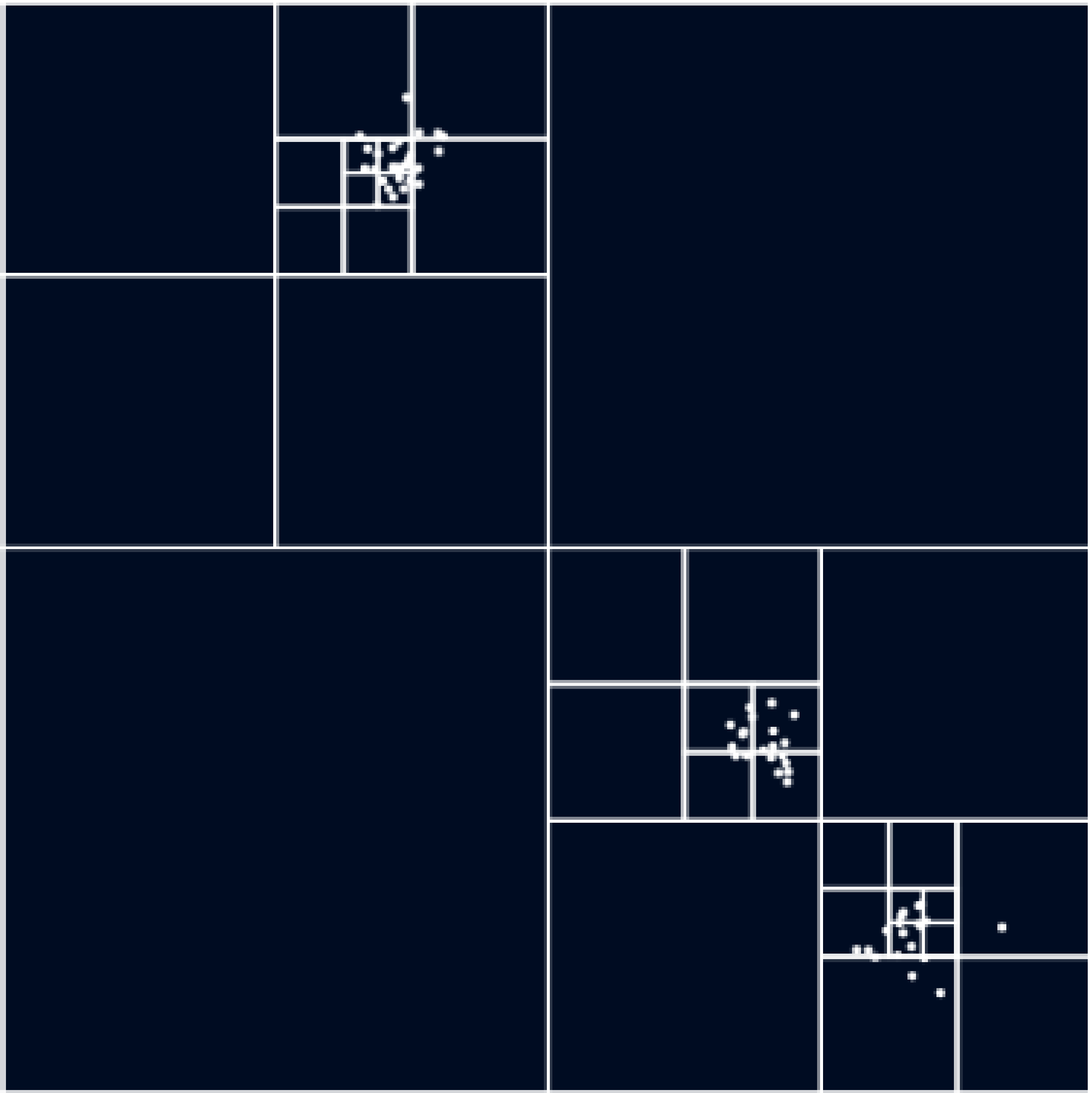
- Need
- Construction
- Finger Tree
- Searching

2.3 Dynamic Quadtrees

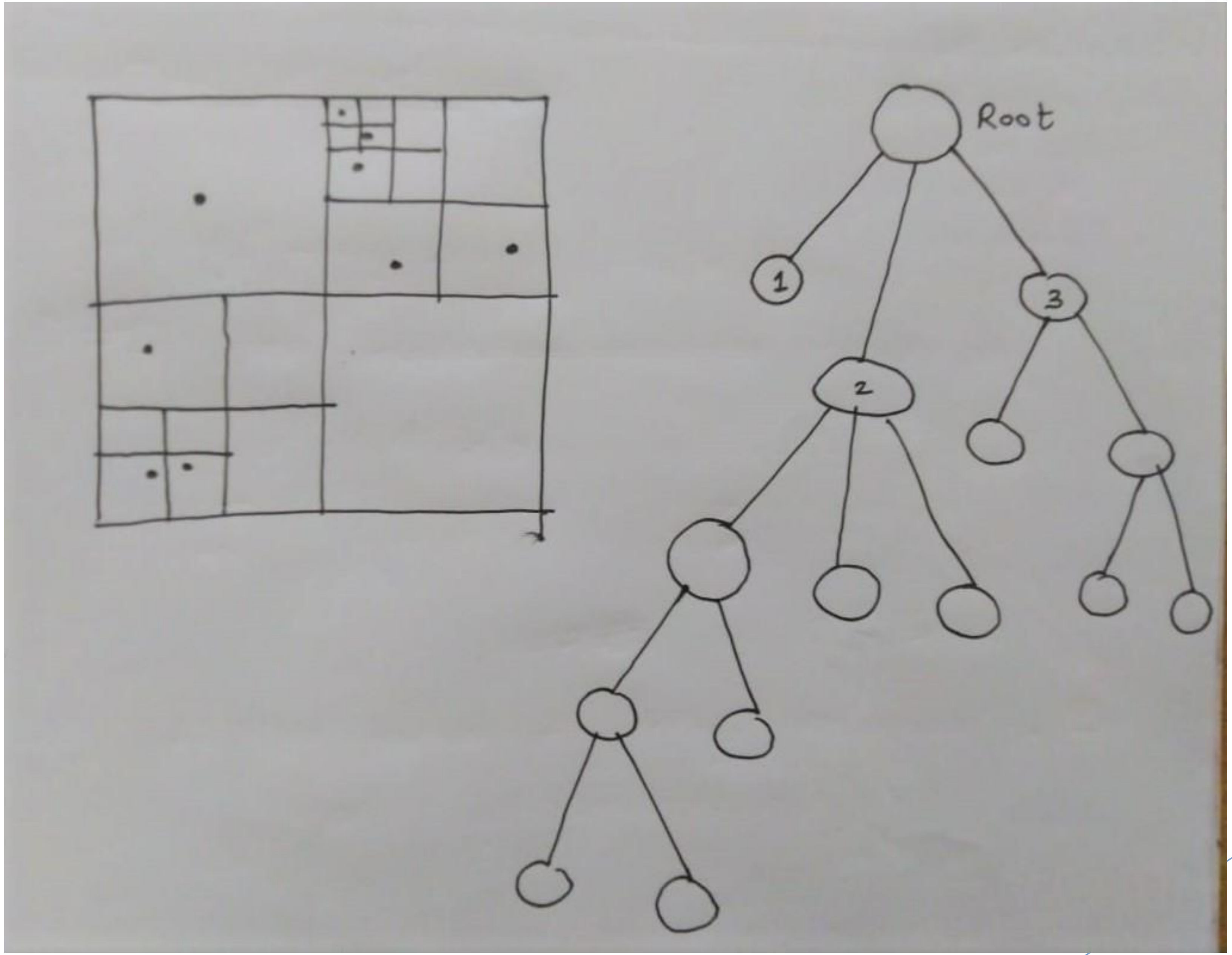
- Need
- Search/Insertion

What is a quadtree?

- Data structure - used to represent 2 dimensional spatial information in the form of a tree.
- Every node - represents a region of the space, stores information about the region.
- Every internal node - has exactly 4 children - 4 quadrants (along X and Y axes).
- 1D analogue - segment tree, extended in 3D as octatrees.



- Defining property – Node (current region) has further children iff region has 'interesting' properties.
- Point Quadtrees - built over data comprising of several data points scattered in a 2D region. Common query – locating number of points in a specified area on the plane.
- Here - 'interesting property' - is the presence of points. Therefore as long as the space contains more than one point - split into 4 parts corresponding to 4 child nodes.
- When space contains - 1 point - node is a leaf.


Root
1
2
3

Other quadtrees

- Region Quadtrees : Built over data corresponding to the space, eg. Image Compression.
- Images - stored as data belonging to each individual pixel.
- An efficient way to compress - quadtree.
- Split the current region until the color property in the region is 'sufficiently uniform'.
- Thus, most quadtrees built over real life data will be skewed.

Grid structure of Quadtrees

- Let the root of the quadtree be v_1. It represents the entire space over which the points are spread. The root is at depth 0.
- Now, consider a node at depth i. This node represents a square of side length 2^{-i}, in a grid of all squares of the same size.
- Thus, every node v in the quadtree can be defined by a triplet $<l(v), x, y>$ where l(v) is the level of the node and x and y are the coordinates of the bottom left point in the region.

Two functions of Point Quadtrees
A: Answering Range Queries

```
query(range, found) {
  if (!found) {
    found = [];
  }
  if (!this.boundary.intersects(range)) {
    return;
  } else {
    for (let p of this.points) {
      if (range.contains(p)) {
        found.push(p);
      }
    }
    if (this.divided) {
      this.northwest.query(range, found);
      this.northeast.query(range, found);
      this.southwest.query(range, found);
      this.southeast.query(range, found);
    }
  }
  return found;
}
```

```
node.intersects(range){
  return !(
    range.x - range.w > node.x + node.w ||
    range.x + range.w < node.x - node.w ||
    range.y - range.h > node.y + node.h ||
    range.y + range.h < node.y - node.h
  );
}
```

B : Fast Point Location
(Locating the node corresponding to given point)

- Main issue - level of the node at which the point is present - unknown.
- Thus, binary search all possible levels.
- Note: Hash table used for each level to store all points present at this level to their corresponding node.
- With this, QTGetNode runs in O(1) time.

QTFastPntLocInner(T, q, lo, hi).
  $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
  $v \leftarrow$ **QTGetNode**(T, q, mid)
  if $v = null$ **then**
    return **QTFastPntLocInner**$(T, q, lo, mid - 1)$.
  $w \leftarrow$ **Child**(v, q)
    //w is child of v containing the point q.
  If $w = null$ **then**
    return v
  return **QTFastPntLocInner**$(T, q, mid + 1, hi)$

Time complexity for B

- Let the height of the Quadtree be H.
- Since we are binary searching over the levels, we consider at most log H levels.
- Furthermore, the operations we do at each level are :

1. Checking if any node at that level contains our point: O(1)
2. Checking if the node we get has a further child : O(1)

- Thus the overall time complexity is O(logH). If H is well maintained and in the order of log(N), our operation's complexity is O(log log N).

Demo :quadtree.herokuapp.com

Source Code : git.io/qtree

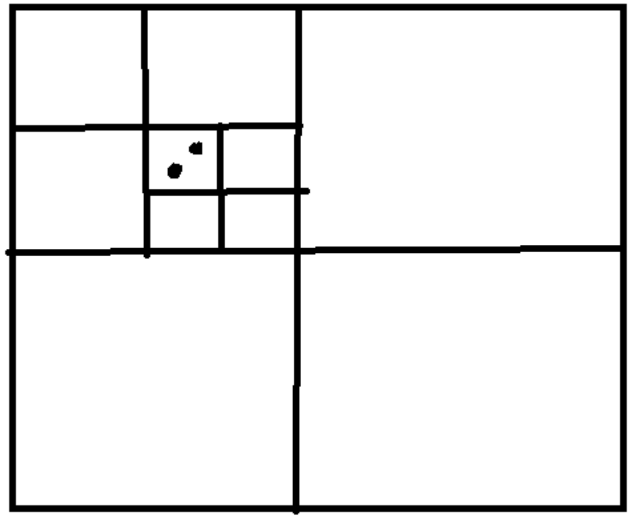
2.2: Compressed Quadtrees

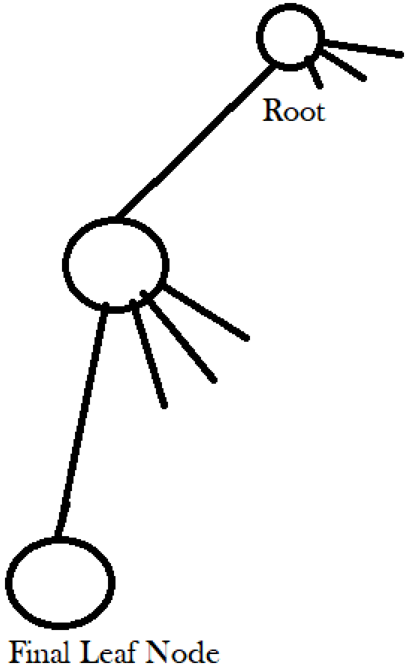
Need for Compressed Quadtrees

- A quadtree may contain a lot of useless nodes.
- Consider a case where two points are very close to each other, and there are no other points in a large region surrounding them.
- Then, the quadtree node containing these points keeps splitting. But only one of the 4 split children is useful.
- This results in the formation of long chains in the Quadtree, which result in both an increase in the number of nodes as well as height.

Compressed Quadtree: Such long chains are reduced to a single "compressed node" and a final child.

Situation mentioned on previous slide :
An unneccessary intermediate node is created, along with many null ptrs.





Nodes in Quadtree

Important points about Compressed Quadtrees

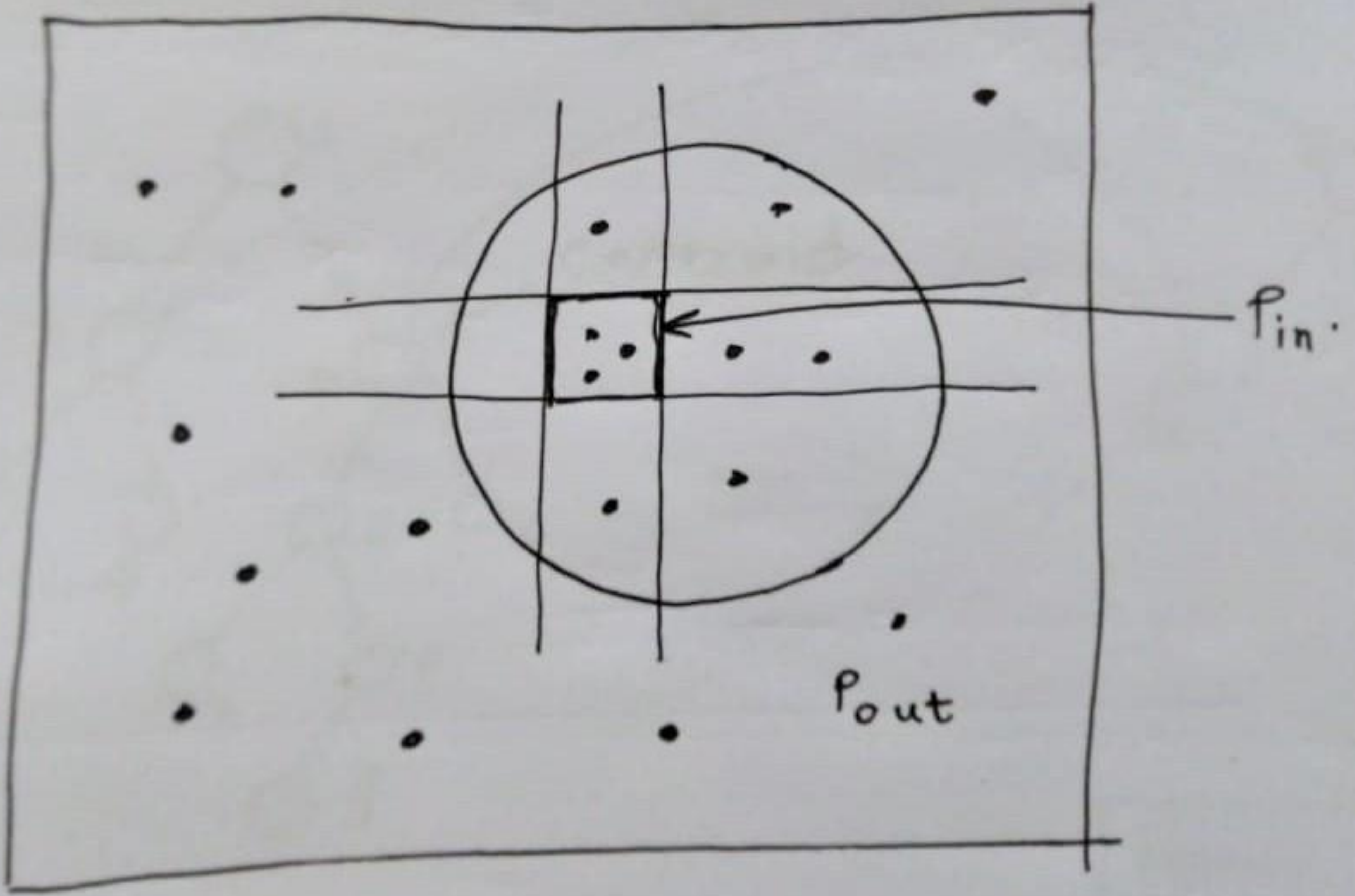
1. It has a linear number of nodes. This follows from the fact that there are at most n-1 internal nodes with degree > 1 in the compressed quadtree.
2. There are no bounds on the height of Quadtree. Height can still be O(n) in the worst case.
3. Since intermediate nodes - removed - no specific depth relation holds between parent and child. Hence - binary search technique discussed previously - no longer works.

Thus a new algorithm must be devised.

Constructing the Compressed Quadtree

- We are looking to construct the compressed quadtree in O(nlogn) time. A recursive method can be used.
- Consider that the plane has n points.
- Using the disc results from the previous PPT – it is possible to find a disc with radius r containing at least n/10 points, such that $r <= 2\, r_{opt}$.
- Now, we construct a grid of level log(r).
- Let $l = 2^{\lfloor \log(r) \rfloor}$. Consider the grid G_l .

- Now due to the relation between the size of grid squares and the radius, there exists a square in the grid which contains at least n/250 points and at most n/2 points.
- Pigeonhole principle (grid intersects with 25 squares and contains n/10 pts).
- Thus, using this square, we can break the problem into two subproblems, each of size n/k : Building T_{out} for points outside the square and T_{in} for points inside the square.
- Also, once these trees are individually calculated, the root of T_{in} can then be hung at the appropriate compressed node in T_{out} (which can be found by a point location query).
- Thus, the compressed quadtree can be computed in O(NlogN) time.


ρ_{in}
ρ_{out}

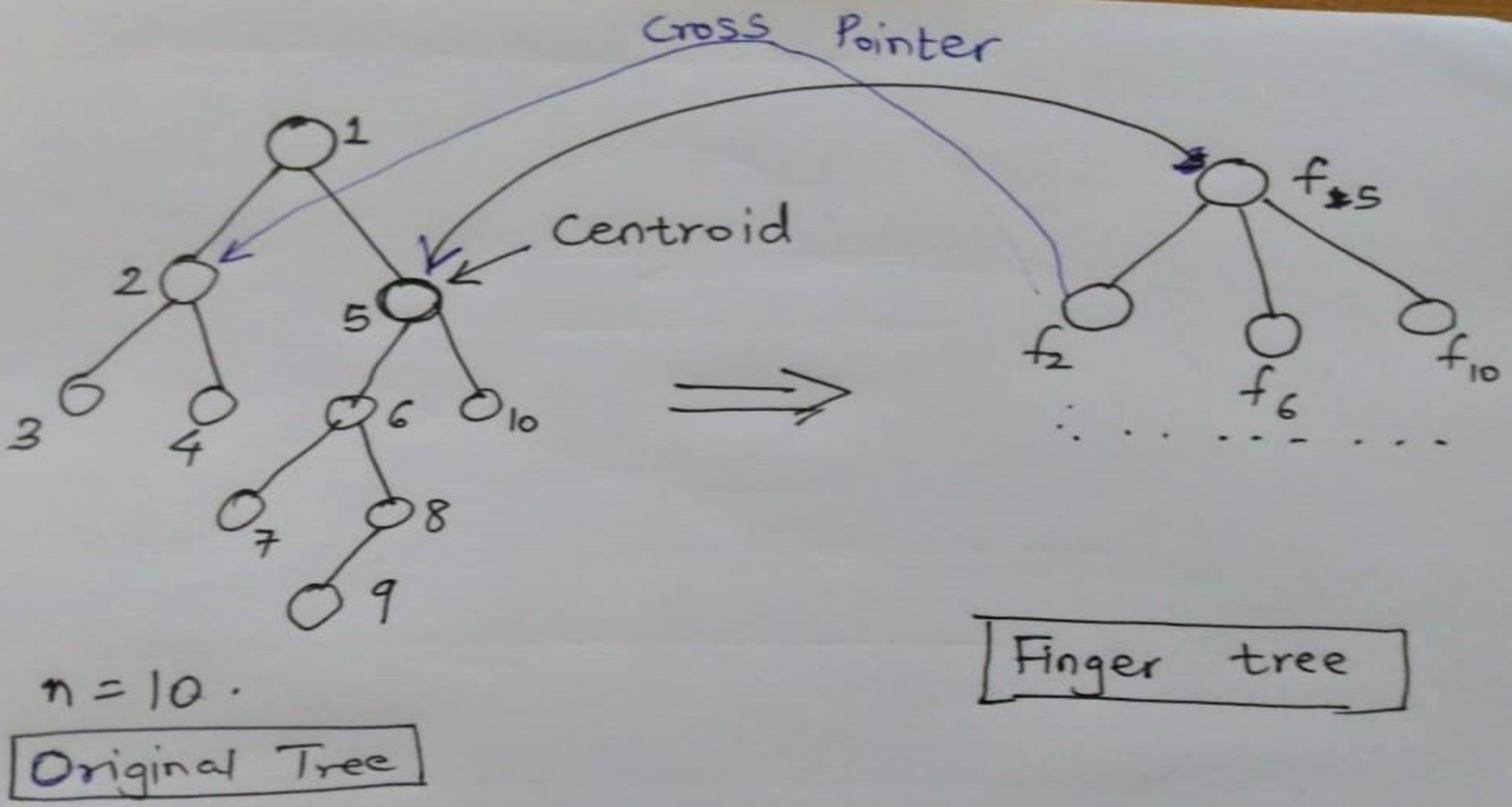
Point location in Compressed Quadtree

- Our goal is to find the node which contains our specified point.
- Simply traversal of tree might take O(n) time - no restrictions on height.
- As discussed earlier, cannot use binary search.
- An additional tree T' called the 'finger tree' will be used.
- Will contain cross pointers ("fingers") into T.
- T' - balanced

Constructing the Finger Tree T ' for Original Tree T

- Every tree has a centroid or separator node(say w) - all components created on removing this - have size <= ⌊n/2⌋.
- Root of T' - corresponds to median of T. Contains a pointer to the corresponding node in T – Let it be R_0.
- For all the new trees created, follow this procedure recursively and join these to R_0 -the root of T '.
- This new tree is balanced, and height is O(log N).

[Depth(n) <= 1+ Depth(n/2); Use Master's Theorem.]



Cross Pointer
1
2
3
4
5
6
7
8
9
10
Centroid
f_5
f_2
f_6
f_{10}
n = 10.
Original Tree
Finger tree

Searching in the Finger Tree

- For a query point q, begin traversing in T '.
- If q lies in the region represented by our current node - continue into child that contains q.
- Else go into the child in T ' - corresponds to the region outside the connected component hung on current node in T.
- Thus searches take place in O(H) of T' = O(log N).

2.3: Dynamic Quadtrees

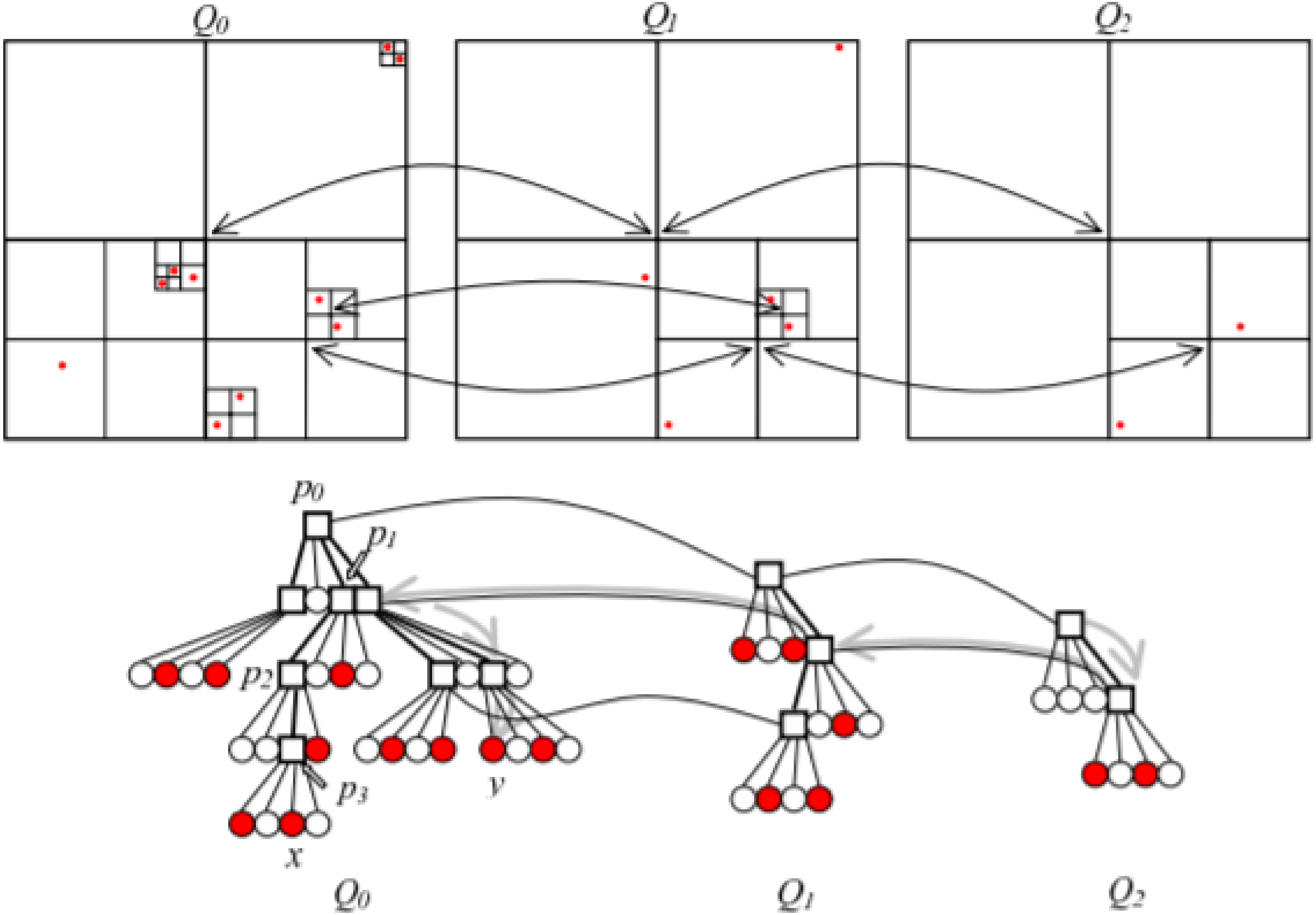
Need for Dynamic Quadtrees

- Construction of finger trees - cumbersome - if multiple points added.
- Without the finger tree - a compressed tree - complexity O(N) - for all operations.
- Thus dynamic quadtrees are an elegant, randomization based solution for carrying out searches as well as updates in O(logN) time.

Sampling sequence of points

- Create a sampling sequence of points S_m, ….S_2, S_1, such that:
  - $S_1 = P$
  - S_i is formed by picking each point of S_{i-1} by probability ½ .
  - We stop selection when the number of points goes below a prespecified k
  - $S_m <= k$ and $S_{m-1} > k$.
- We build quadtrees T_1, T_2, T_3, T_4, … on each of these sets of points.

- Each square in Quadtree T_i is connected to its corresponding square in T_{i-1}, and in T_{i+1} if it exists.
- Note that any square which is interesting in T_i is also interesting in T_{i-1}, since T_{i-1} is made out of superset of points of T_i.
- This property, along with the cross pointers is responsible for operations happening in O(logN).



Q_0
Q_1
Q_2
p_0
p_1
p_2
p_3
x
y
Q_0
Q_1
Q_2

How search/insertion takes place

- To find : Smallest square in T_1 covering location of the query point
- Start from the root of T_m, and find the minimum interesting square in T_m which contains that point, and is also interesting in T_{m-1}.
- Using pointers move to T_{m-1}, which is more detailed than T_m

(more depth).

- Continue this until T_1.
- Thus, instead of searching in the large tree T_1, we have searched in smaller trees and navigated by pointers.

Proof that the process will indeed take O(logN) time

- Within each tree - number of searching steps - constant.
- Proof: Pr(j) : Probability that j searching steps are performed.

$$E(j) = \sum_{1}^{m} jPr(j) \le \sum_{1}^{m} j\left(\frac{j+1}{2^{j+1}} + \frac{1}{2^{j+1}}\right) = \frac{1}{2}\sum_{1}^{m}\frac{j^2}{2^j} + \sum_{1}^{m}\frac{j}{2^j} \approx \frac{1}{2} \times 6.0 + 2.0 = 5.0.$$

- The expected number of levels in the dynamic quadtree is logN.
- Thus, overall O(logN).