Sheet-6

Problem 1

(a) Query: All Employees per city.

This query requires a join between Store and Employee relation. Employee and Store table is in BCNF since, for Employee table, Name, Salary and SID depends on EID which is the superkey. For Store table, City and EID depends on SID. For the above query, we can simply put city column to Employee table to make denormalization. The denormalized schema would be:

⇒ Employee [EID, Name, Salary, SID, City]

⇒ We denormalized the table to avoid join operation for performance gain. Downside of this denormalization

would be, we have redundant data for list of employees for city

⇒ To keep data consistent, for every store input, we need to enter corresponding store data in employee table also with city information.

⇒ Normal form of original schema is BCNF

⇒ Normal form of new schema is 2NF

⇒ Query after denormalization:

```
Select * from employees
order by city
```

Query: Price of all item in given invoice IID = 5

This query requires a join between Item and Article table. Following decisions are made :-

⇒ To denormalize the table Item and Article to find the Price, we can simply put Price in Item table and the denormalized schema will be

ITEM (<u>IID</u>, <u>SID</u>, <u>AID</u>, Count, Price)

⇒ Downside of this denormalization would be, we need to insert redundant data to keep both

Article and Item consistent.

⇒ Query after denormalization would be

Select Price from ITEM where IID=5

⇒ Normal form of original table is BCNF

⇒ Normal form of new table is 1NF, since, we inserted Price, which depends on AID, and AID is a partial key, so there exists partial FD.

(b)

(i) We can remove DISTINCT here since select * will essentially retrieve all rows.

(ii) As we are retrieving data only for EID 2002, we can simply omit GROUPBY and HAVING clause and put it in a WHERE clause. Also, when we include WHERE, there is no need for MAX aggregate also.

(iii) We can simply remove the join column and query over only Employee table to find the SID

3.

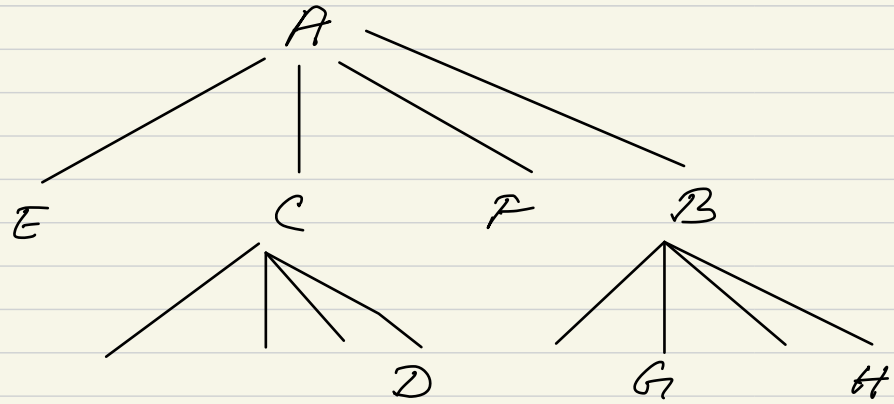
(a) Following are assumed coordinates:

name	x	y
A	35	40
B	32	13
C	45	80
D	72	64
E	7	42
F	26	31
G	83	14
H	79	5

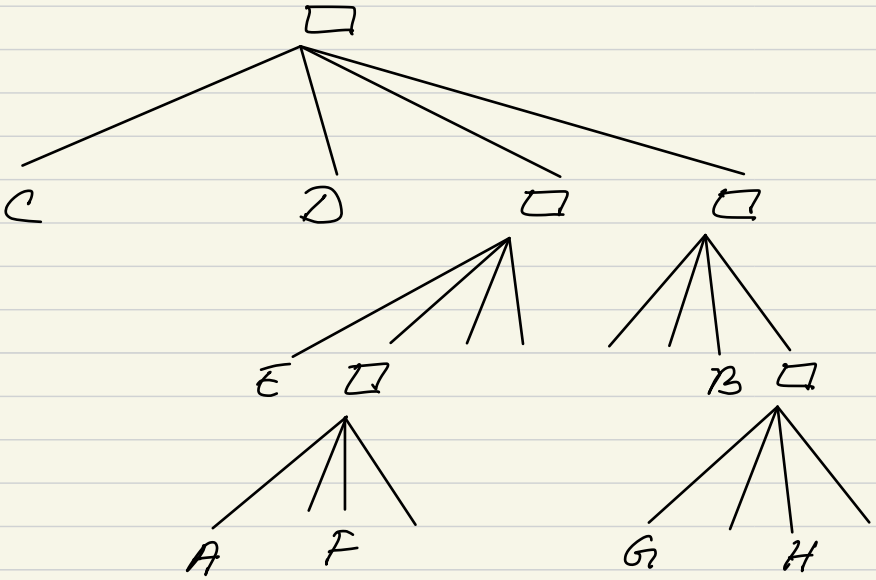
Assuming the following order to generate a Quadtree that has less empty leaf nodes than PR Quadtree based on above coordinates is

A, B, C, D, E, F, G, H

Quadtree : No. of empty leaf = 5



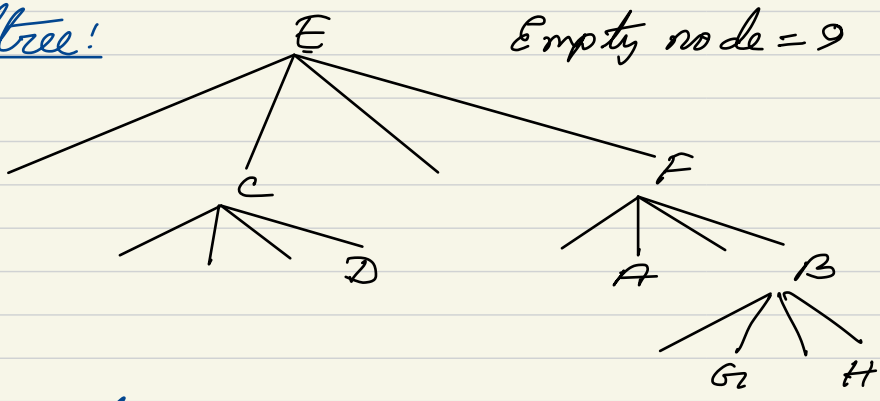
PR Quadtree No. of empty leaf = 8



Assuming the following order which will generate Quadtree with more empty leaf nodes than PR Quadtree:

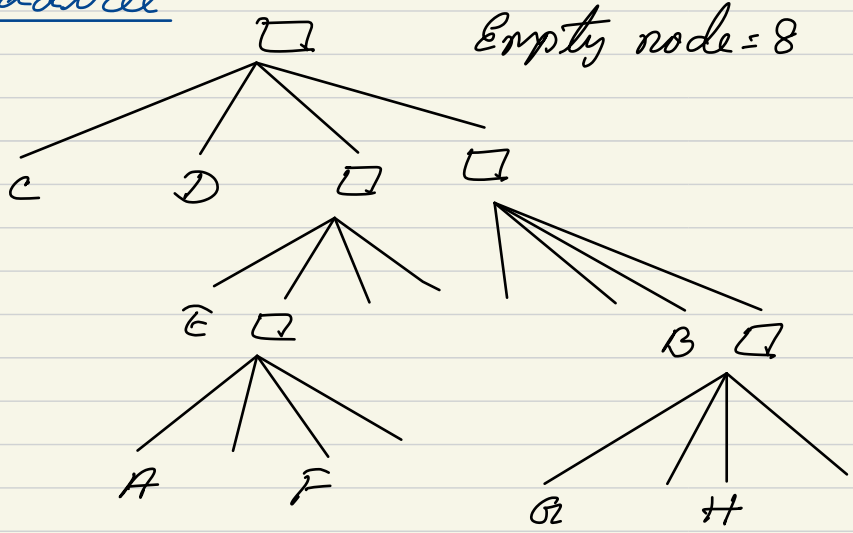
E, F, A, B, C, D, H, G

Quadtree:



Empty node = 9

PR Quadtree



Empty node = 8

(b) Pseudocode for optimized quadtree creation with minimized height is as follows:

OptimizeQuadtree (coordinate, Plane)

→ if (coordinate = nil, Return

→ else calculate (x_i, y_i) coordinate which's distance is minimum from center of plane, i.e. Median

→ if $(x_i, y_i) = 0$, Return

→ else plot (x_i, y_i)

→ $S_W = (0, x_i), (y_i, y)$

→ $S_E = (x_i, x), (y_i, y)$

→ $N_W = (0, x_i), (0, y_i)$

→ $N_E = (x_i, x), (0, y_i)$

→ Remove (x_i, y_i)

→ Recursively call

- optimizeQuadtree (coordinates, SW)
- optimizeQuadtree (coordinates, SE)
- optimizeQuadtree (coordinates, NW)
- optimizeQuadtree (coordinates, NE)

Explanation:

In the above pseudocode, we first derived the median from available dataset, then divided other data points into 4 regions, on each side of median, providing same amount of datapoints. From each of those 4 regions, we calculate the median again and go deeper. This way, the height of the tree is optimized.

(c) As we know, PR Quadtree always has 4 uniform regions, with depth k, probability of finding a particular point at depth k will be

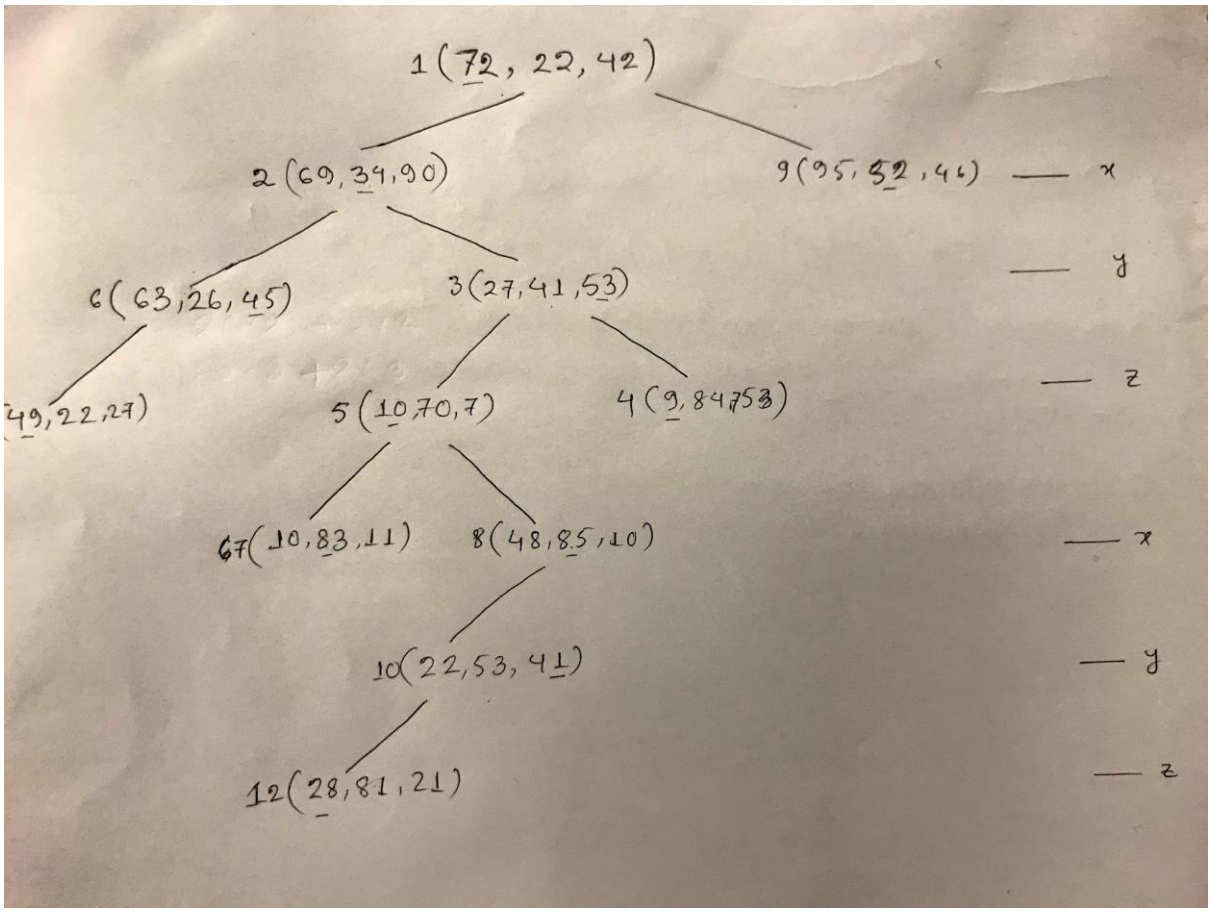
$$\frac{1}{4^k}$$

For collection of V points, the probability that none of the points lies in a given cell at depth k would be

$$\left(1 - \frac{1}{4^k}\right)^V$$

Problem 4:

a)


1(72, 22, 42)
2(69, 34, 90)
9(95, 52, 44)
x
6(63, 26, 45)
3(27, 41, 53)
y
z
49, 22, 27)
5(10, 70, 7)
4(9, 84, 75)
67(10, 83, 11)
8(48, 85, 10)
x
10(22, 53, 41)
y
12(28, 81, 21)
z

b)

Pseudocode: [1]

```
KDTree (leftNode, rightNode)
Split (value, dimention): implements KDTree
Point (v1, v2, ...): implements KDTree

insert_point(KDTree T, List<of>Point):
    // find the one with the largest variance
    var max_variance_dimension
    for all dimensions d:
        sort points by dimension d
        get two median points P1 and P2
        get variance of P1 and P2:
            avg = (P1.d.value + P2.d.value) / 2
            variance = P1.d.value ~ avg
        if variance is maximum:
            update max_variance_dimension with current dimension

    // find split point
    sort by max_variance_dimension
    get two median points P1 and P2
    split_value = (P1.max_variance_dimention.value +
P1.max_variance_dimention.value) / 2

    if T is null:
        T = new Split(split_value, max_variance_dimension)
    else if split_value <= T.leftNode.value:
        T.leftNode = insert_point(T.leftNode, List<of>Point that are lower
than split_value in max_variance_dimension)
    else if split_value > T.rightNode.value:
        T.rightNode = insert_point(T.rightNode, List<of>Point that are
larger than split_value in max_variance_dimension)

    return T
```

Explanation:

1. At first as instructed determine the split dimension.
    a. Which is the one with the largest variance, we considered the difference of a median node to the average of two median nodes. [2]
    b. Run this for all dimension and find out the maximum variance.
2. The split value is calculated as the average of two median nodes of selected dimension in previous stage.
3. Build kd tree (left and right) using recursion. [3]

Reference:

[1] https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf
[2] https://en.wikipedia.org/wiki/Variance
[3] Geometric algorithms and data structures: Prof. Suri (TU Darmstadt)
https://slideplayer.com/slide/4262586/