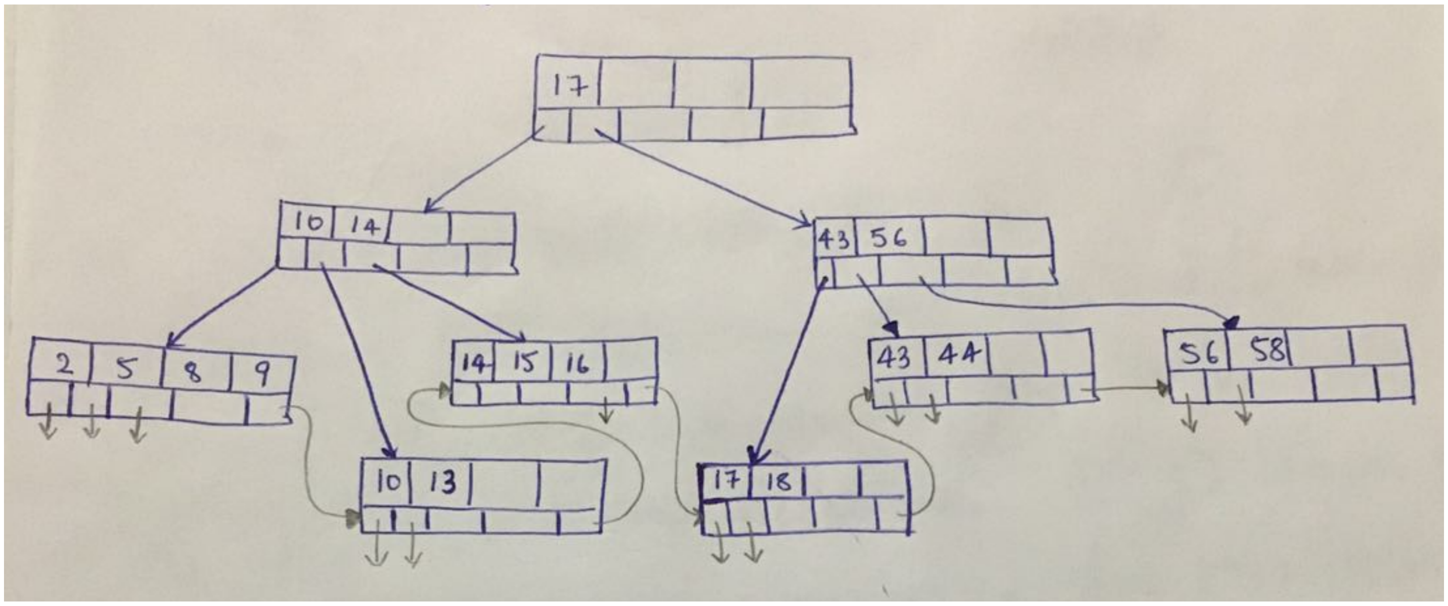
INF-551 Yamini Goyal

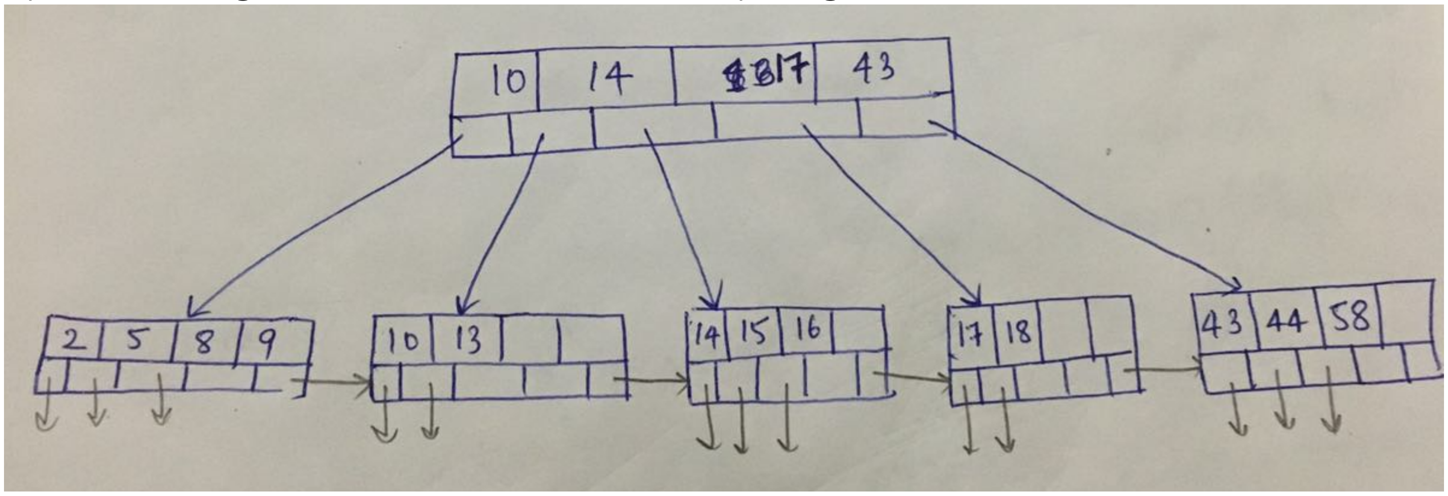
Assignment 4

Solution 1:

a) On inserting 16, B+ tree will look as below:



b) On deleting 56 from the above tree in a) we get:



c) From the root node (1st block to be read), keys are read. On seeing 10 and 17 we follow the pointer (lets name it P1) between 10 and 17. We read the block pointed by P1 (2nd block to be read) and read the keys which are 10, 13, 14, 15. Then we follow the end pointer in the 2nd block which points to its sibling. This block (3rd block to be read) has keys 17 and 18 which are desirable. So we now again follow the end pointer in the 3rd block which points to its sibling block (4th block to be read). 1st key in this block is 43 which is greater than 20 and hence we stop. **So, total I/O operations are 4.**

Solution 2: Given,

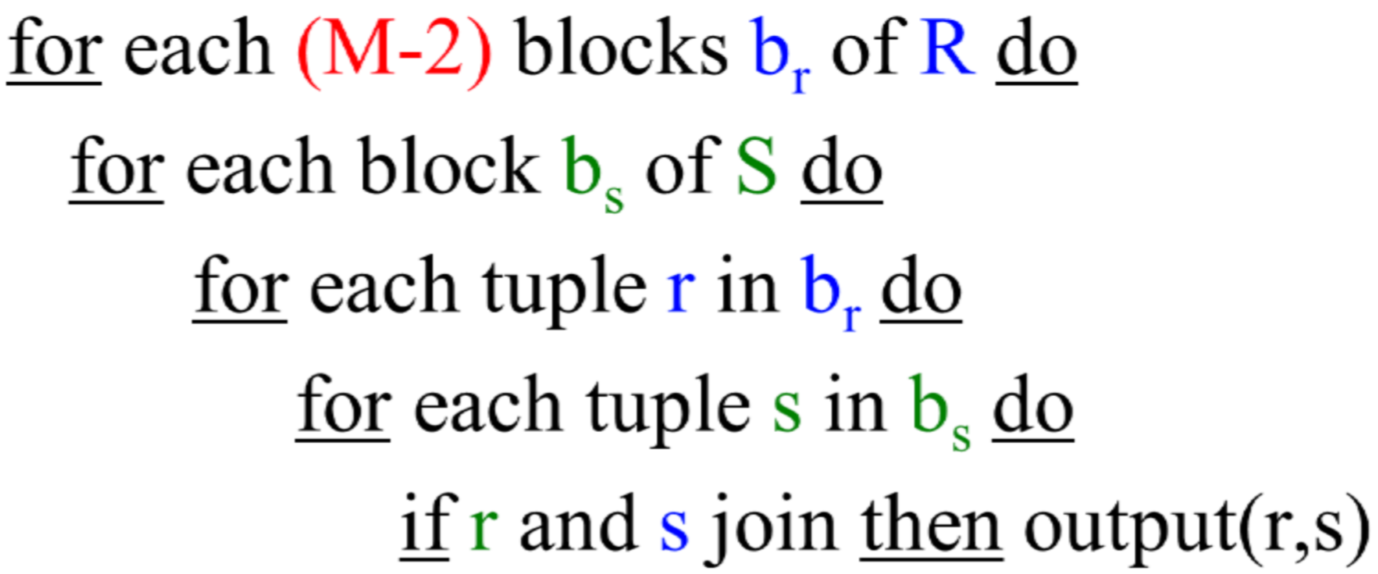
B(R) = 100 ; T(R) = 1000 ; B(S) = 200 ; T(S) = 2000 ; V(S,b) = 5 ; M = 12

a) Nested Loop Join with R as Outer Relation

Since R and S, both are clustered relations we perform Block based Nested Loop Join.

Assumption satisfied: 100 < 200 && 100 > 12

R being the outer relation, **(M-2 = 10)** blocks of R and 1 block of S will be loaded in memory as input. For each tuple of 10 blocks of R and 1 block of S we check for join condition. If they join we output that tuple else reject. Algorithm is as below:



```
for each (M-2) blocks $b_r$ of R do
  for each block $b_s$ of S do
    for each tuple r in $b_r$ do
      for each tuple s in $b_s$ do
        if r and s join then output(r,s)
```

Total Number of I/O = B(R) + B(R) * B(S) / (M-2)
= 100 + 100 * 200 / 10
= **2100**

b) Nested Loop Join with S as Outer Relation

Since R and S, both are clustered relations we perform Block based Nested Loop Join.

Assumption: **200 > 12**

S being the outer relation, **(M-2 = 10)** blocks of S and 1 block of R will be loaded in memory as input. For each tuple of 10 blocks of R and 1 block of S we check for join condition. If they join we output that tuple else reject.

Total Number of I/O = B(S) + B(S) * B(R) / (M-2)

= 200 + 200 * 100 / 10

= **2200**

c) Sort Merge Join

Assumption: 100 + 200 <= 144 – Not satisfied. Hence a little modification from the usual case where we just sort the blocks of two relations and merge them together.

However, here. We first sort the blocks of R and S (load 10 blocks every run, generating 10 and 20 runs respectively) and perform the intra-relation merging of R and S (using 10 blocks as buffer). We load the sorted and merged relation R, S (1 for R and 2 for S) in the memory and perform the inter merging of Relation R and S.

Total Number of I/O = I/O for sorting + Inter merging R + Inter Merging of S+ Intra Merging R and S

= 2B(R) + 2B(S) + 2B(R) + 2B(S) + B(R) + B(S)

= 5B(R) + 5B(S)

= 5*100 + 5*200

= 1500

d) Simple sort based join

First sort R and S individually loading 10 blocks in the memory generation 10 and 20 runs respectively which will cost 2*B(R) + 2*B(S). Later, merge R individually loading 10 pages in memory which will cost 2*B(R). Now, merge S in 2 passes.

Pass 1: Use 10 pages of memory for input buffers, and one page as the output buffer. Merge runs of S. When output is full, write it. This step will generate 2 runs. Cost: 2*B(S).

Pass 2: Use 2 pages of memory for input buffers, and one page as the output buffer. Merge runs of S. When output is full, write it. Cost: 2*B(S).

Join runs of R and S: Use 3 pages of memory for input buffer (1 for R, 2 for S) and one page as the output buffer. Merge runs of R and S at once, join tuples and S while merging. When output is full write it. This will cost another B(R) + B(S) I/O

Total Number of I/O = 5 * B(R) + 7 * B(S)
= 5*100 + 7 *200
= 1900

e) Partitioned Hash Join
Assumption: min (100,200) = 100 <= 144
First, for each relation R and S separately, using the hash key generate buckets using one pass algorithm and send it to the disk. For R, each bucket will approximately the size of 100/12 = 8 blocks and for S, 200/12 = 17 blocks. Duplicate tuples are hashed to the same bucket.

B(R) = 100 < 144, hence the buckets of B(R) can fit into the memory. Load buckets of R.

Now, tuples in partition Ri match only the tuples in partition Si. So, read in a partition of Si, and hash it using another hash function h'. Scan matching partition of Ri and search for matches.

Total I/O Cost: 3 * B(R) + 3 * B(S)
3 * 100 + 3* 200
= **900**

f) Index Join
Since S has a clustered index, all the tuples with the same value appear together and in the fewest blocks as possible.
Iterate over R, for each tuple, fetching corresponding tuple from S.
Now since R is clustered,

Total I/O Cost: B(R) + T(R) * B(S) / V(S,b)
= 100 + 1000 * 200 / 5
= **40100**

Comparison:

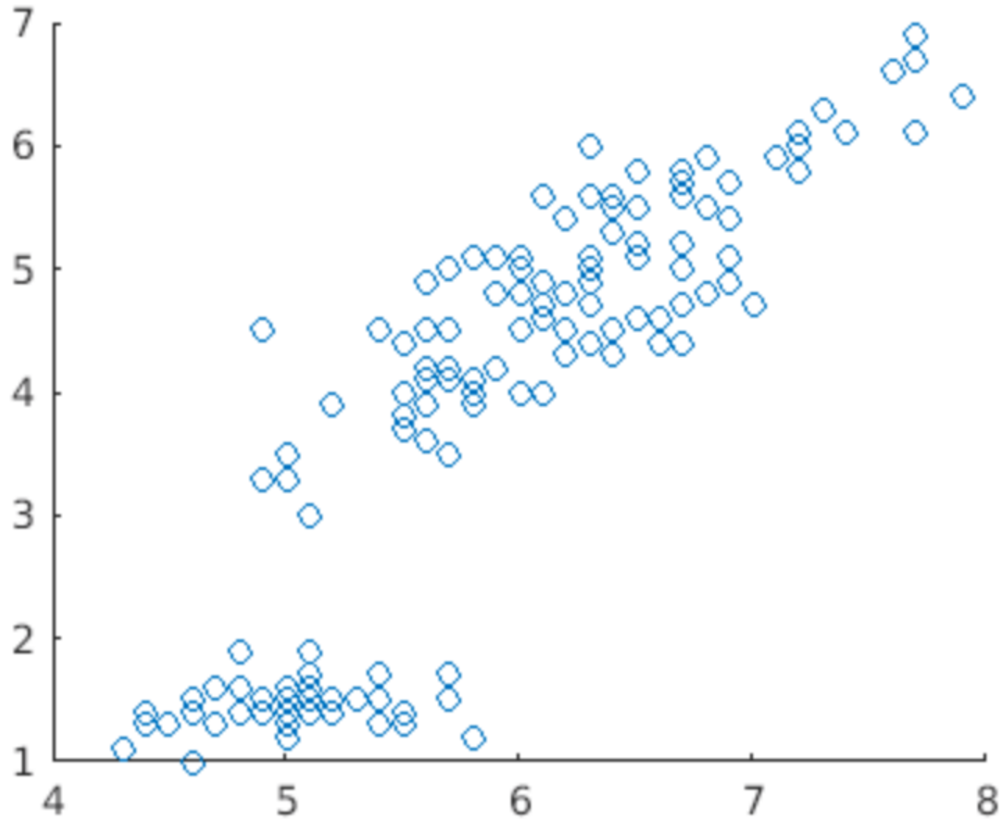
Lowest Cost: Partitioned Hash Join
Highest Cost: Index Join

Partitioned hash uses hashing function to generate the buckets such that duplicate tuples match to the same bucket. So when the merging happens, tuples in ith partition of R are looked into only ith partition of S. Hence the lower cost.

Whereas, in Index partition tuples with similar values are in the fewest blocks possible. However still all the blocks are searched in both relations for all the tuples to join the relations increasing the I/O operations. Hence the higher cost.

Solution 3:
I have used MATLAB to solve this problem. MATLAB scripts has been included in the submission folder. *Scatterplot.m* consists code for a) and b). *Histogram.m* consist of code for c)

a) Following depicts the scatterplot for the given data. Sepal length as x-axis and petal length as y-axis.

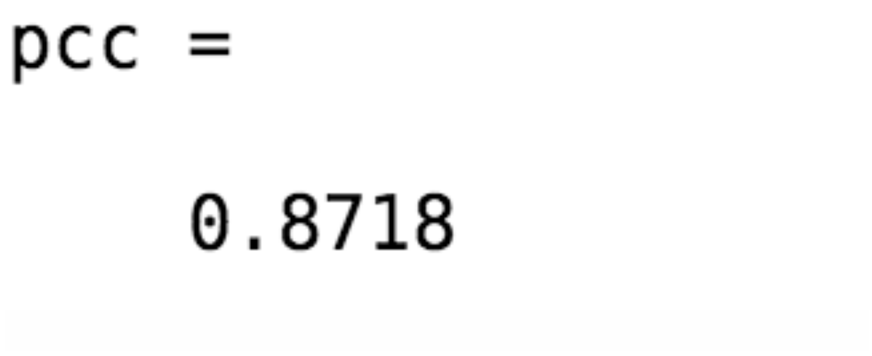


From the graph it looks like the coefficient falls between 0.6 and 1.00. Hence, sepal length and petal length seems like they are **positively correlated and dependent**.

b) The formula used in the program to calculate Pearson correlation coefficient is as follows:

$$\text{Pearson Coefficient : } \frac{\text{covariance (sepal length, petal length)}}{\text{Std dev(sepal length) * std dev(petal length)}}$$

For the given data Pearson Coefficient is as follows:



```
pcc =

    0.8718
```

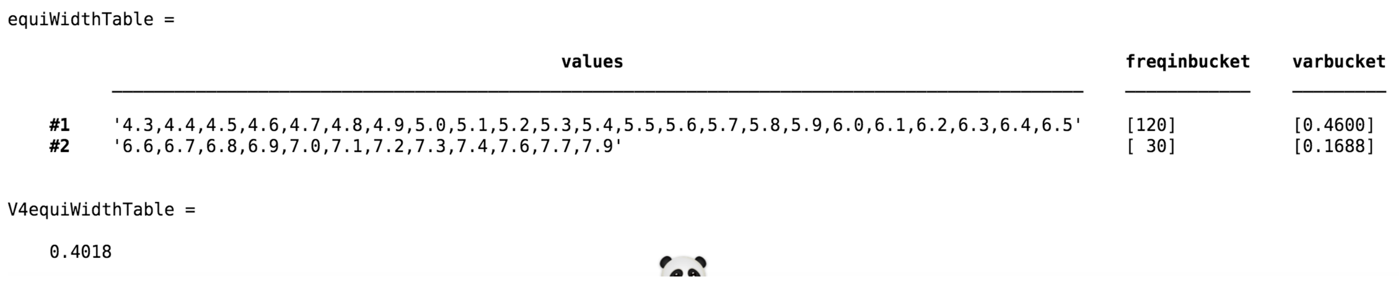
Since the coefficient is close to 1, sepal length and petal length looks **positively correlated and dependent.**

Yes, Pearson Coefficient is consistent with scatterplot results.

c)

Equi-Width

- **Values Spread:** 7.9 - 4.3 + 1 = 4.6
- **2 buckets:** 4.6 / 2 ~ 2.3
- **Buckets range:** [4.3,6.5] [6.6,7.9]



```
equiWidthTable =

                                              values                                               freqinbucket    varbucket
        ________________________________________________________________________________________   ____________    _________

    #1  '4.3,4.4,4.5,4.6,4.7,4.8,4.9,5.0,5.1,5.2,5.3,5.4,5.5,5.6,5.7,5.8,5.9,6.0,6.1,6.2,6.3,6.4,6.5'   [120]           [0.4600]
    #2  '6.6,6.7,6.8,6.9,7.0,7.1,7.2,7.3,7.4,7.6,7.7,7.9'                                               [ 30]           [0.1688]

V4equiWidthTable =

    0.4018
```

Weighted variance = 0.4018

Equi-Depth

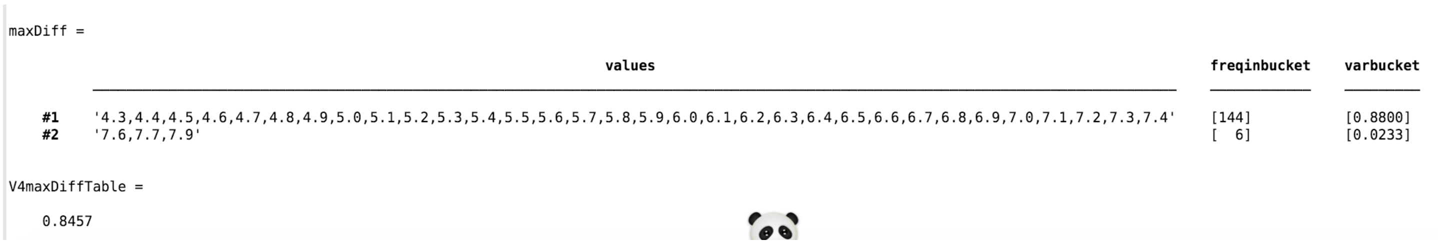
- **Total Frequencies:** 150
- **Each bucket contains 75 values**

```
equidepthTable =

                                      values                                        freqinbucket    varbucket
    ____________________________________________________________________________    ____________    _________

    #1    '4.3,4.4,4.5,4.6,4.7,4.8,4.9,5.0,5.1,5.2,5.3,5.4,5.5,5.6,5.7,5.8'                [75]            [0.2267]
    #2    '5.8,5.9,6.0,6.1,6.2,6.3,6.4,6.5,6.6,6.7,6.8,6.9,7.0,7.1,7.2,7.3,7.4,7.6,7.7,7.9'    [75]            [0.3896]

V4equidepthTable =

    0.3081
```

Weighted variance = 0.3081

Max Diff

- **Max difference in values:** 0.2



```
maxDiff =

                                                   values                                                      freqinbucket    varbucket
    ___________________________________________________________________________________________________________    ____________    _________

    #1    '4.3,4.4,4.5,4.6,4.7,4.8,4.9,5.0,5.1,5.2,5.3,5.4,5.5,5.6,5.7,5.8,5.9,6.0,6.1,6.2,6.3,6.4,6.5,6.6,6.7,6.8,6.9,7.0,7.1,7.2,7.3,7.4'    [144]    [0.8800]
    #2    '7.6,7.7,7.9'                                                                                            [  6]    [0.0233]

V4maxDiffTable =

    0.8457
```

Weighted variance = 0.8457

Weighted variance is lowest for Equi Depth. Therefore, EqiDepth method produces the best histogram in this case.