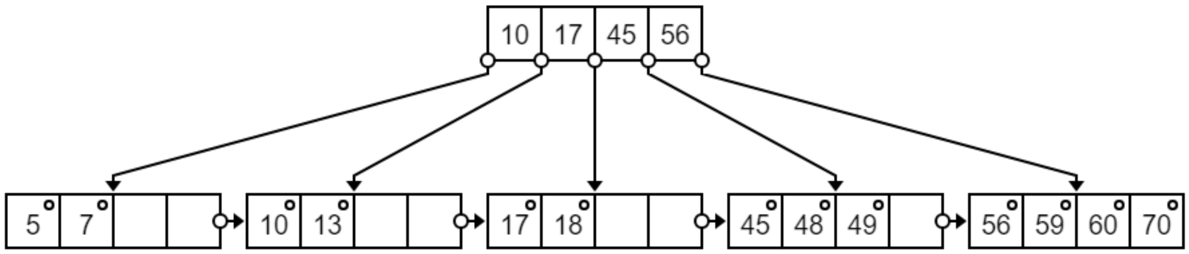
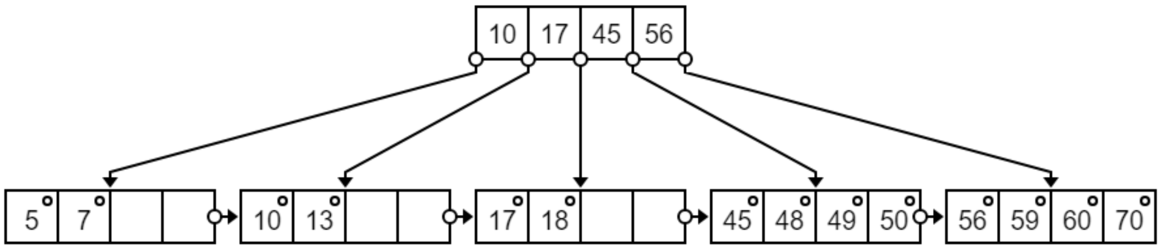
Homework -4

1. **B+ tree for search key age:**
   a. Query: age>=10 and age<=50
   Steps:
   Step 1: Read the root node. Check for the value age>=10 & move to the second block in the second row. [1 block I/O]
   Step 2: Start reading from the second block (from key=10) and keep reading till you reach the key=56 in 4th block, but since 56 is greater than 50 it exceeds the condition and stops. [4 blocks I/O]
   So, total we require 5 blocks I/O's for this process.

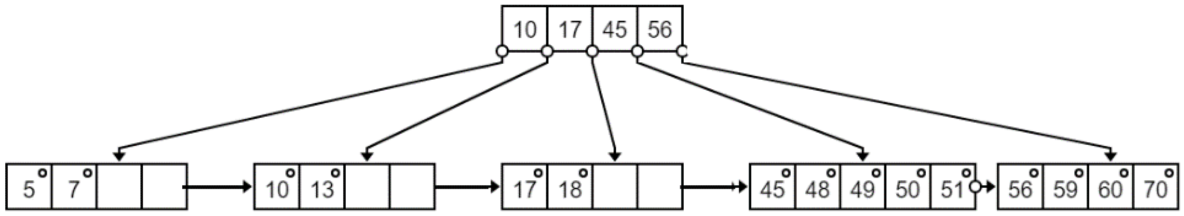
   b. Inserting 49:



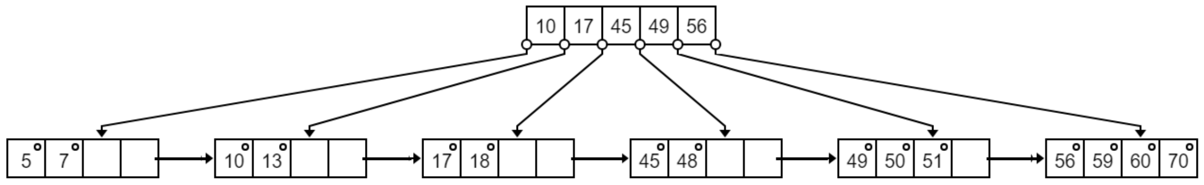
Inserting 50:



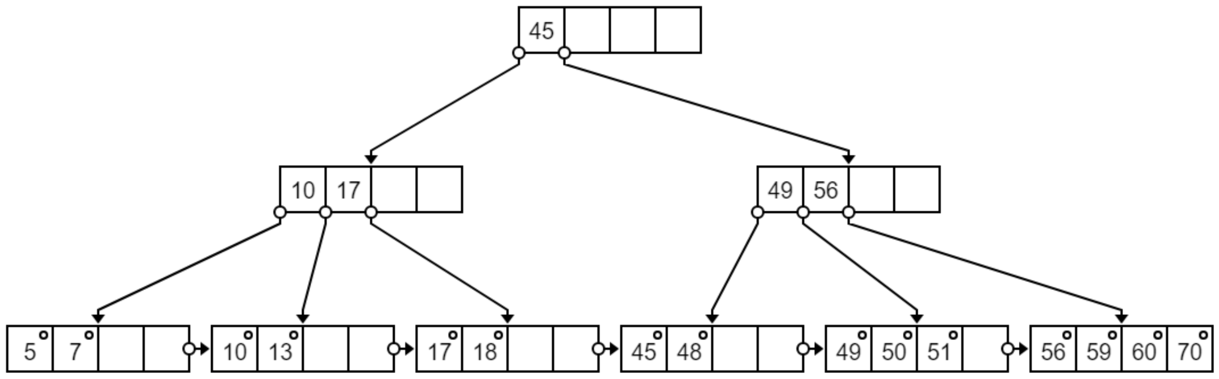
Inserting 51:



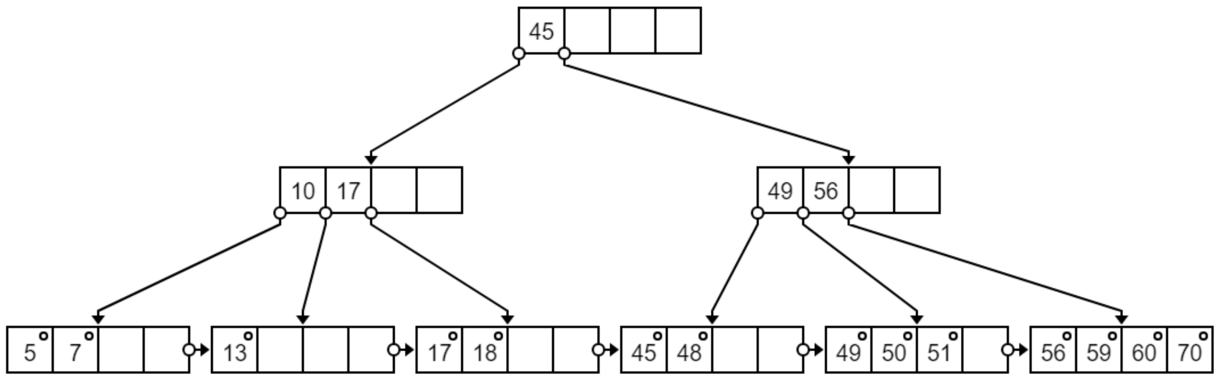
When we try to insert 51 into the 4th node in second row, there will be an overload, since number of keys (i.e. 5) is greater than 2d (i.e. 4). So, to balance the tree we need to split the 4th node to get the following:



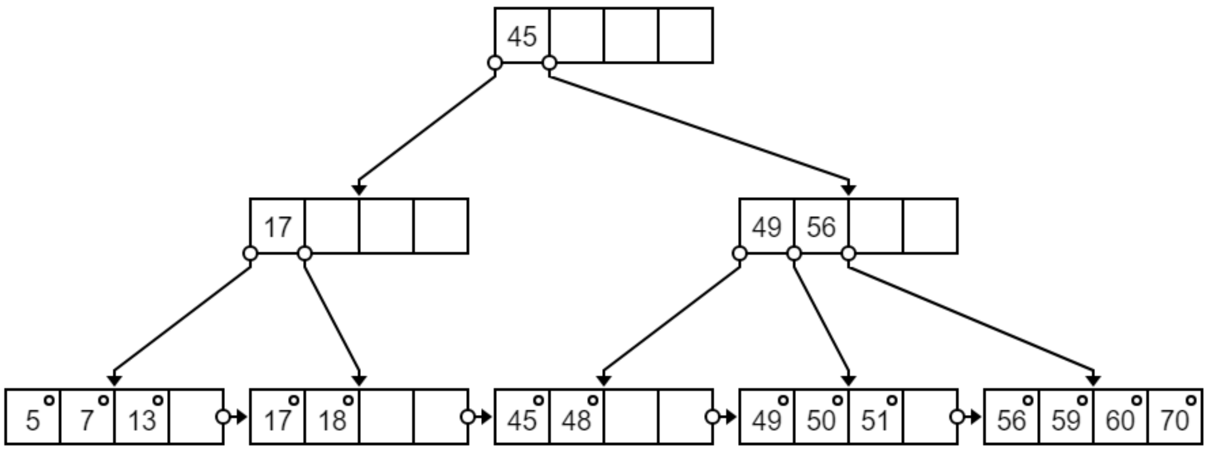
Now there are 5 keys in the root node, which is greater than 2d (that is 4). So, we need to split it to get the following:



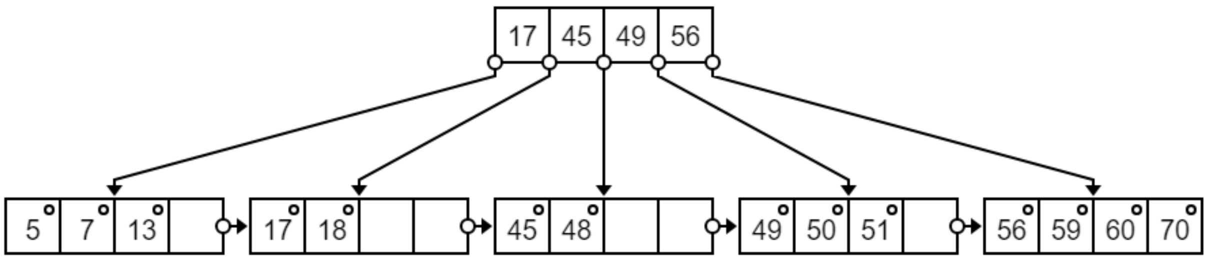
c. **Deleting 10:**



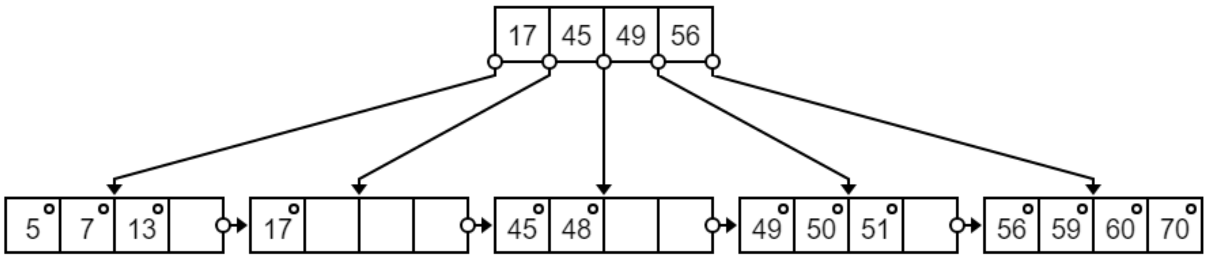
Now we can see that in the 2nd node in third row, the number of keys (=1) is less than d (=2). So, to balance the tree we need to merge this node with the immediate left node. Also, we need to delete 10 from the 1st node in second row to obtain the following:



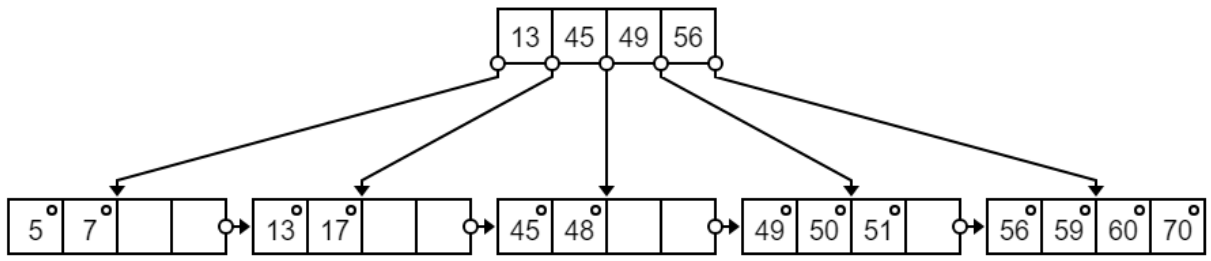
Now we can see that in the 1st node of second row, the number of keys (=1) is less than d (=2). So, to balance the tree we need to reduce the tree by bringing the parent node down as follows:



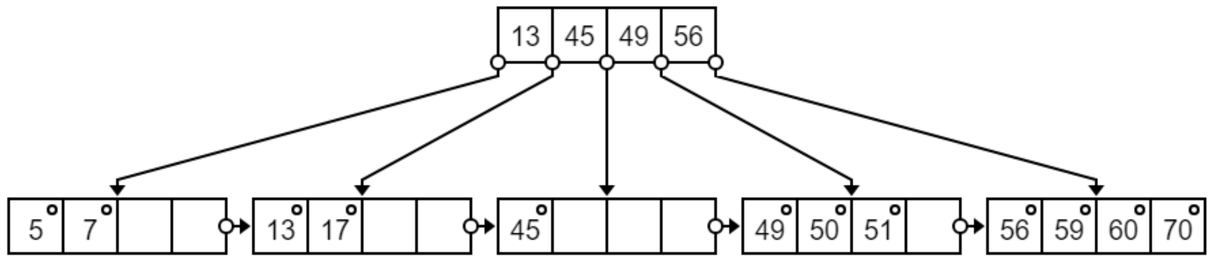
Deleting 18:



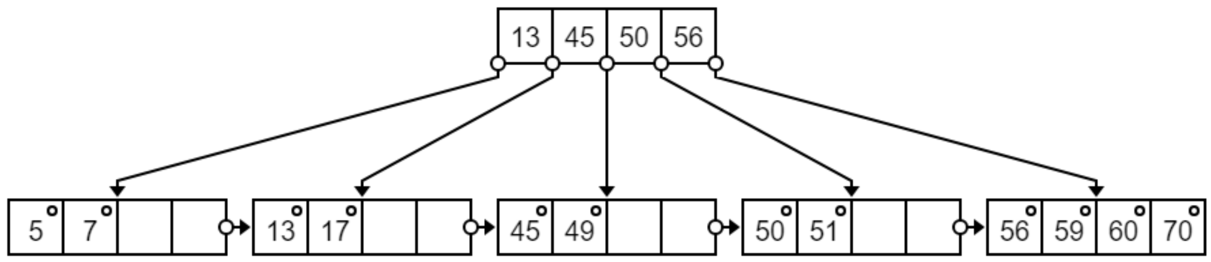
Now we can see that the second node in the 2nd row has number of keys (=1) less than d (=2). So, to balance the tree we need to move one key from the immediate left node. Also, we need to update the root node key=17 to key=13 to obtain the following:



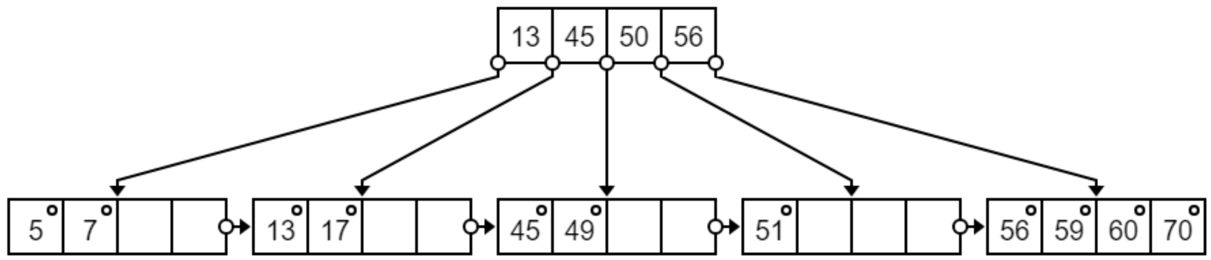
Deleting 48:



Now we can see that the third node in the 2nd row has number of keys (=1) less than d (=2). So, to balance the tree we need to move one key from the immediate right node. Also, we need to update the root node key=49 to key=50 to obtain the following:

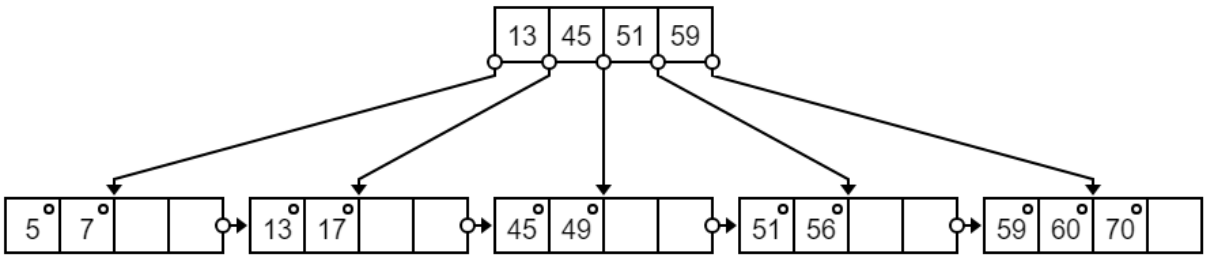


Deleting 50:

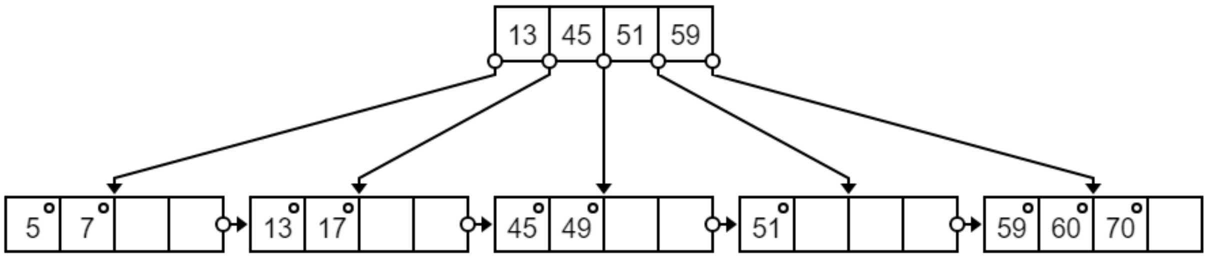


Now we can see that the fourth node in the 2nd row has number of keys (=1) less than d (=2). So, to balance the tree we need to move one key from the immediate right node.

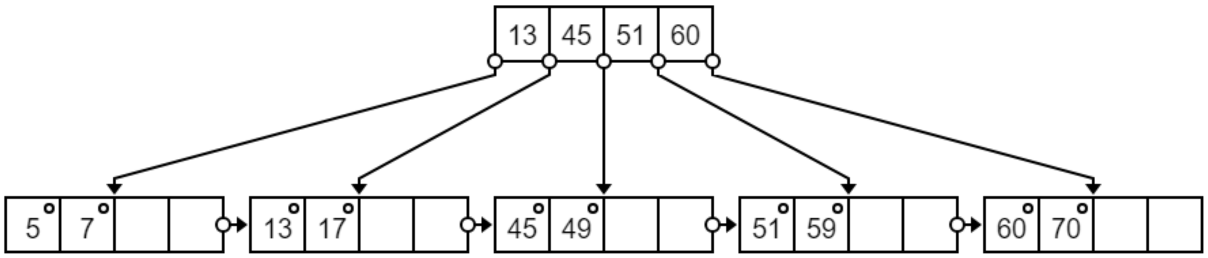
Also, we need to update the root node key=50 to key=51 and key=56 to key=59 obtain the following:



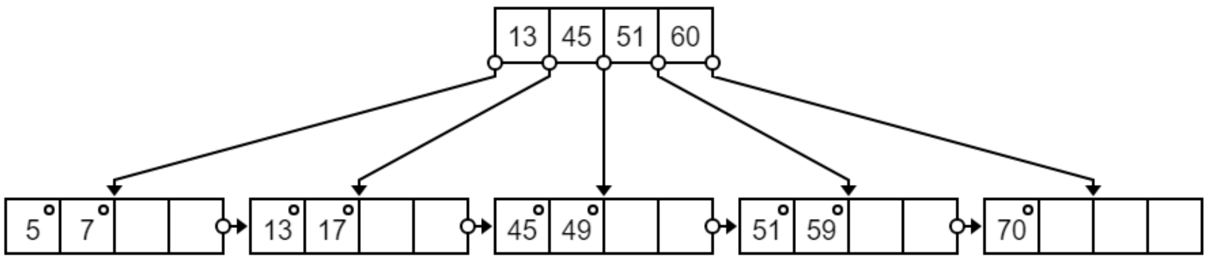
Deleting 56:



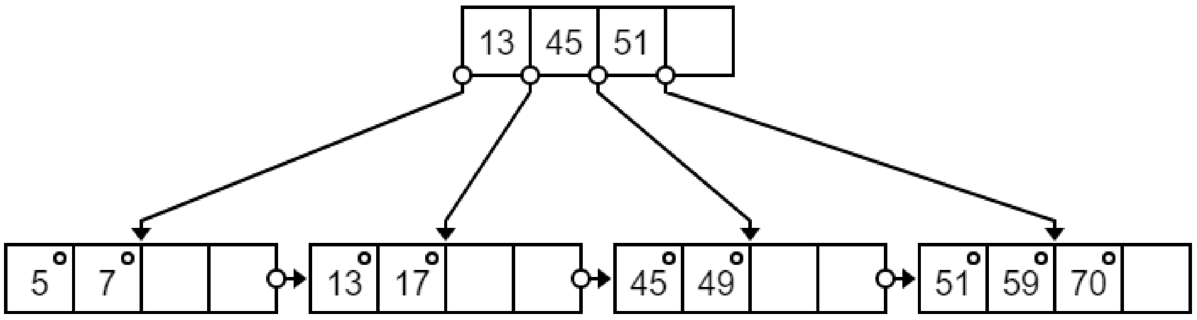
Now we can see that the fourth node in the 2nd row has number of keys (=1) less than d (=2). So, to balance the tree we need to move one key from the immediate right node. Also, we need to update the root node key=59 to key=60 obtain the following:



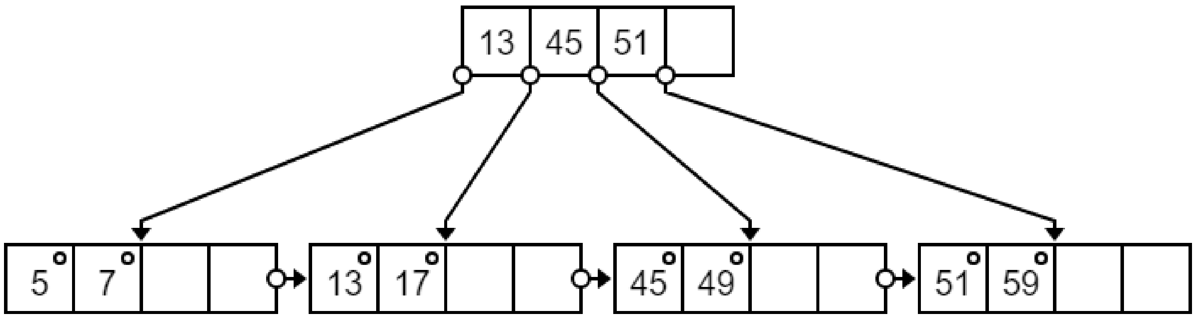
Deleting 60:



Now we can see that the 5th node in the 2nd row has number of keys (=1) less than d (=2). So, to balance the tree we need to merge this node with the immediate left node. Also, we need to delete the key=60 in root node to obtain the following:



<u>Deleting 70:</u>



2. <u>Join algorithms:</u>

a. Nested-loop join with R as the outer relation:

Algorithm:

```
for each 100 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 $b_s$ so
        if r and s join then output(r,s)
```

<u>Input:</u> M-2 = 100 blocks of R at a time and 1 block of S for each M-2 blocks of R.

<u>Output:</u> Joined tuples of R & S.

<u>Cost:</u> Read R once –> cost = B(R) = 20000

Outer loop runs B(R)/(M-2) times, and each time need to read S –> cost = B(R)B(S)/(M-2) = 20000*8000/100 = 16,00,000

Total cost = B(R) + B(R)B(S)/(M-2) = 16,20,000 I/O’s

b. Nested-loop join with S as the outer relation:

```
for each 100 blocks $b_s$ of S do
   for each block $b_r$ of R do
      for each tuple s in $b_s$ so
         for each tuple r in $b_r$ do
            if s and r join then output(s,r)
```

<u>Input:</u> M-2 = 100 blocks of S at a time and 1 block of R for each M-2 blocks of S.
<u>Output:</u> Joined tuples of R & S.
<u>Cost:</u> Read S once –> cost = B(S) = 8000
Outer loop runs B(S)/(M-2) times, and each time need to read R –> cost = B(S)B(R)/(M-2) = 20000*8000/100 = 16,00,000
Total cost = B(S) + B(S)B(R)/(M-2) = 16,08,000 I/O's

c. Sort-merge join:
<u>Step 1:</u> Load 100 blocks of R at a time, sort, and write it back to the disk. This will generate 200 runs of size 100 blocks each. Cost = 2*B(R) = 2*20000 = 40000 I/O's
<u>Step 2:</u> Load 100 blocks of S at a time, sort, and write it back to the disk. This will generate 80 runs of size 100 blocks each. Cost = 2*B(R) = 2*8000 = 16000 I/O's
Here we observe that runs of R + runs of S>101. So we merge runs from R further.
<u>Step 3:</u> Load 181 runs from R, sort it and write back to disk. This will generate 2 runs of size 100*100 and 81*100 blocks/pages each. Cost = 2*(181*100) = 36200 I/O's
<u>Step 4:</u> Merge the runs of R and S. Cost = B(R) + B(S) = 20000+8000 = 28000 I/O's
Total number of block I/O's = Total cost = 1,20,200 I/O's

d. Simple sort-based join:
<u>Step 1:</u> Load 100 blocks of R at a time, sort them and write them back to the disk. This will generate 200 runs of size 100 blocks each. Cost = 2*B(R) = 2*20000 = 40000 I/O's
<u>Step 2:</u> Load 101 runs of R at a time and merge them. This will generate 2 runs of size is 10100 and 9900 pages each. Cost = 2*B(R) = 2*20000 = 40000 I/O's
<u>Step 3:</u> Load 2 runs of R at a time and merge them. This will generate 1 run of size 20000 pages. Cost = 2*B(R) = 2*20000 = 40000 I/O's
<u>Step 4:</u> Load 100 blocks of S at a time, sort them and write them back to the disk. This will generate 80 runs of size 100 blocks each. Cost = 2*B(S) = 2*8000 = 16000 I/O's
<u>Step 5:</u> Load 80 runs from S at a time, sort them and write them back to the disk. This will generate 1 run of size 8000 pages. Cost = 2*B(S) = 2*8000 = 16000 I/O's
<u>Step 6:</u> Merge the run from R and S. Cost = B(R) + B(S) = 20000+8000 = 28000
Total number of block I/O's = Total cost = 1,80,000 I/O's

e. Partitioned-hash join:

<u>Step 1:</u> Hash R into buckets of size M-1 (=100) buckets using Hash Function h, and send all buckets to disk. This will generate 100 buckets of size 200 blocks each. Cost = 2*B(R) = 2*20000 = 40000 I/O's.

<u>Step 2:</u> Hash S into buckets of size M-1 (=100) buckets using Hash Function h, and send all buckets to disk. This will generate 100 bucket of size 80 blocks each. Cost = 2*B(S) = 2*8000 = 16000 I/O's.

Since the memory is not enough to hold R of size = 200 blocks/bucket, we need to hash it further.

<u>Step 3:</u> Hash each bucket of R into M-1 (=100) buckets using Hash Function h', and send all buckets to disk. This will generate 100 buckets of 2 blocks each. Cost = 2*B(R) = 2*20000 = 40000 I/O's.

<u>Step 4:</u> Join every pair of corresponding buckets of R and S. Cost = B(R) + B(S) = 20000+8000 = 28000 I/O's.

Total number of block I/O's = Total cost = 1,24,000 I/O's

f. Index join:

Given:

We know that S has a clustered index on the join attribute a.

V(S, a) = 100

<u>Step 1:</u> Iterate over M-2 (=100) blocks of R, for each tuple. Cost = B(R) = 20000 I/O's

<u>Step 2:</u> Fetch corresponding tuples from S without join tuples. Number of tuples = B(S)/V(S,a) = 8000/100 = 80 tuples

Cost = T(R)B(S)/V(S,a) = 200,000*80 = 1,60,00,000 I/O's

Total number of block I/O's = Total Cost = B(R) + T(R)B(S)/V(S,a) = 1,60,20,000 I/O's

<u>Conclusion:</u> From the Total cost calculated for each algorithm, we can see that Sort-merge algorithm is most efficient with Total cost = 1,20,200 I/O's.