DEPARTMENT OF COMPUTER SCIENCE
LUDDY SCHOOL OF INFORMATICS, COMPUTING, AND ENGINEERING
INDIANA UNIVERSITY BLOOMINGTON

CSCI-B 561 Advanced Database Concepts (ADC)

ASSIGNMENT 4

HARD Deadline: April 27, 2024

Note: Total # of points = 125, total # of questions = 17

Vaishnavi Vishwas Pawar

(vpawar@iu.edu)

[Question 1] Explain what the dirty flag of a page in buffer pool management is. What complications might arise if this flag is not properly managed (5 points)?

Ans:

In buffer pool management, each memory page is assigned a binary indicator known as the dirty flag. It indicates if the page has been modified since it was last fetched from or written to disk. When a page is first loaded into the buffer pool, the filthy flag is set to false, indicating that no changes have occurred. If a transaction alters the page, the dirty flag is set to true, indicating that the modifications have not yet been saved to disk.

If the dirty flag is not properly managed, several complications can arise:

- Data Inconsistency: Without precise dirty flags, the database may write back pages that have not been modified (spending IO resources) while failing to write back altered pages, resulting in disc and buffer pool inconsistencies.
- Data Loss: If the system crashes and the dirty pages are not recognized and written back to disk, any modifications made to those pages will be lost.
- Increased I/O Operations: Improperly maintained dirty flags may result in unnecessary writes if pages are wrongly tagged as dirty. This may degrade system performance owing to excessive I/O activities.
- Concurrency Issues: If the dirty flag is not properly maintained in a concurrent environment, one transaction may overwrite the changes of another if they are both working on the same page, resulting in a loss of data integrity.
- Checkpointing and Buffer Management: Checkpointing systems rely on dirty flags to decide which pages to flush to disk. Incorrect dirty flags may result in incomplete checkpointing, compromising the recovery process and the buffer management approach, which is dependent on these checkpoints.
- Inefficient Crash Recovery: During recovery, the database uses the dirty flag to identify which pages in the log need to be restored. If dirty flags are not properly managed during recovery, the database may not be restored to a consistent condition.

[Question 2] What is the difference between page table and page directory (5 points)?

Ans:

Page Table

- In a database management system, the page table maps page IDs to copies of pages stored in buffer pool frames.
- It allows for efficient access to the pages that are currently in memory by acting as a mapping between page IDs and their corresponding copies stored in buffer pool frames.
- This structure exists in memory and does not require disk storage.
- Its contents are dynamically reassembled as pages are loaded into or evicted from the buffer pool, making it volatile and only available while the database is active.

Page Directory

- In contrast, the page directory maps page IDs to their corresponding positions in database files.
- Unlike the page table, the page directory is saved to disk because it keeps track of the relationship between page IDs and their physical locations within database files. This mapping must be permanent.
- Any changes to the page directory must be saved to disk so that the DBMS can appropriately locate and access the pages after a restart or recovery.

Differences:

- Purpose: The page table is intended to allow for quick lookup of pages in the buffer pool, resulting in more efficient data access and manipulation during DBMS operations. On the other hand, the page directory keeps track of where pages can be found on disk over time.
- Persistence: The page table is temporary and only resides in memory, whereas the page directory is persistent and saved to disk.
- Reconstruction vs Recovery: The contents of the page table are dynamically constructed and rebuilt during normal DBMS operations, changing as pages are loaded or withdrawn from the buffer pool. In contrast, the page directory is updated more deliberately because changes must be permanent and crucial for database recovery processes.
- Scope: The page table's scope is restricted to what is currently in memory (the buffer pool). The page directory contains the whole database as it resides on disk, acting as a reference for all pages, whether or not they are currently loaded into memory.

[Question 3] Explain the difference of the two buffer replacement policies, CLOCK and LRU-K, in a buffer pool. Mention an advantage and disadvantage of either policy (5 points).

Ans:

CLOCK Policy

- CLOCK retains a circular list of pages in the buffer pool, each with its reference bit.
- Every time a page is visited (read or written), the reference bit is set to 1.

- As the algorithm traverses the circular list, it provides a second chance to pages with the reference bit set by clearing the bit and bypassing them. If the reference bit is not set, the page will be evaluated for eviction.

Advantages:

- Simplicity and Low Overhead: The CLOCK policy is simple, easy to implement, and has low overhead, making it ideal for systems with limited computational resources or when simplicity is a top requirement.
- Efficiency: It is more efficient than true LRU since it only requires a small amount of checking and updating, which can be done quickly without the use of complex data structures.

Disadvantages:

- Suboptimal eviction decisions: CLOCK's less granular method of measuring time may result in suboptimal eviction judgments, particularly when the reference frequency of pages fluctuates dramatically or when access patterns fluctuate.

LRU-K Policy

- LRU-K is an improved version of the Least Recently Used (LRU) algorithm that takes into account the history of page accesses over the last K references.
- It keeps a queue of pages sorted by the timestamp of accesses, with the most recently viewed page at the front.
- The method keeps track of the times of the last K referrals to each page in the buffer pool. When an eviction is required, LRU-K selects the page with the oldest timestamp as the Kth most recent access.
- This method enables LRU-K to forecast the possibility of a page being required again in the near future based on its access history.

Advantages:

- Accuracy: LRU-K more precisely predicts which pages will be used in the near future based on previous accesses, potentially increasing cache hit rates.
- Pattern Recognition: It is more suited to identifying patterns of access over time and can distinguish between frequently and infrequently used pages, even if they were recently visited.

Disadvantages:

- Complexity: LRU-K is more sophisticated and requires more memory to keep the access history than CLOCK, resulting in slower and less scalable performance.
- Overhead: The added overhead associated with maintaining access history might be significant, especially when K or the buffer pool are large.

[Question 4] Explain the difference between locks and latches in a database management system (DBMS) (5 points).

Ans:

LOCKS:

- Locks provide data consistency and enable ACID features by restricting access to data elements such as rows, pages, or tables throughout a transaction.
- They are retained for the life of the transaction and can vary in scope depending on the size of the data element retrieved.

- Several types, including shared, exclusive, and intent locks, each serving a particular purpose in transaction management.
- Managing locks involves resolving deadlocks and ensuring efficient concurrence.
- Lock granularity improves the flexibility and speed of database operations, allowing DBMSs to optimize the balance between concurrency and data consistency. Fine-grained locks minimize contention but increase overhead, whereas coarse-grained locks simplify lock management but may reduce concurrency.
- Primarily used to prevent other transactions from accessing the same data in contradictory ways, resulting in transaction isolation.
- Typically involves more overhead due to complexity in management and the possibility of performance bottlenecks in high contention conditions.

LATCHES:

- Latches are lightweight, low-level synchronization methods that protect crucial areas of the DBMS code and memory data structures like B-trees and buffer pool pages.
- Held for a relatively brief time when a thread accesses or updates shared data structures; not usually held for the duration of a transaction.
- Ensure the physical integrity of database structures, preventing corruption due to concurrent access by many threads.
- They are designed for speed to reduce performance effects because they are often accessed.
- Because latches have crucial performance consequences, they are only designed to be held for very brief periods. Excessive latch contention can cause a bottleneck, resulting in performance deterioration, particularly under high load.
- Highlighting developments such as latch-free and lock-free algorithms demonstrates how DBMSs evolve to handle concurrency more efficiently, resulting in increased system responsiveness and scalability.
- Less sophisticated than locks, with an emphasis on reducing latency and increasing performance in database operations.

NOTE: Q5 to Q12 answers are attached after Q12 below in written format.

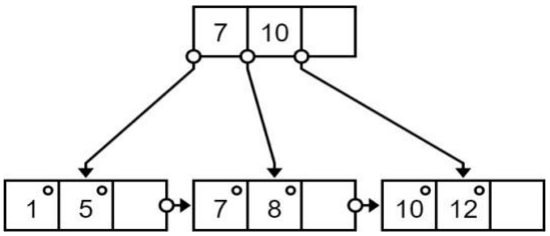
[Question 5] Suppose you have a hash table of size 10, and you are using linear probing for collision resolution. You have a set of 7 keys to insert into the hash table: 18, 29, 37, 42, 50, 53, and 62. Assume the hash function is

$$h(k) = k \bmod 10$$

Starting with an empty hash table, show the resulting hash table after inserting these keys in the given order using linear probing to resolve collisions (2 points).

[Question 6] Consider an extendible hashing scheme where each bucket can hold up to 3 keys. Insert the following keys into the hash table: 16, 4, 6, 22, 24, 10, 31, 7, 9, 20,26. Show the resulting slot array and buckets after each insertion (10 points).

[Question 7] Recall that in an M-way B+ tree, every node other than the root is at least half-full, i.e., $M/2 - 1 \leq$ #keys $\leq M - 1$. Consider the following B+ tree.

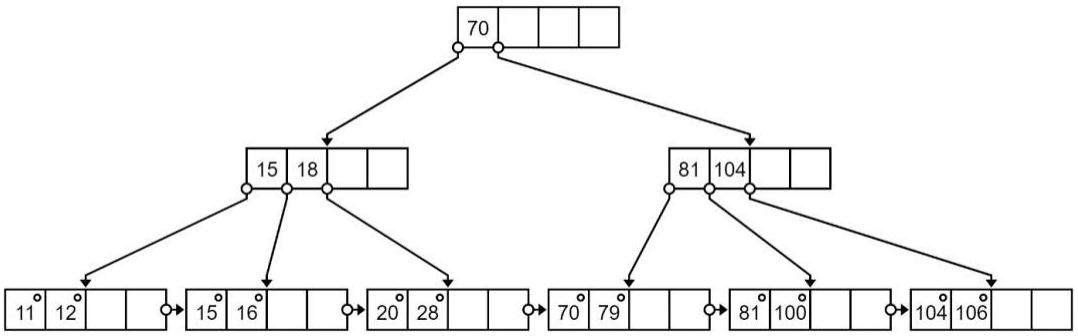


Let us assume that the following keys are inserted in the given order. Please show your tree after each insertion step and explain how you reach it.

a) Insert 24
b) Insert 15
c) Insert 16
d) Insert 17

(2 points each, 8 in total)

[Question 8] Consider the following B+ tree.

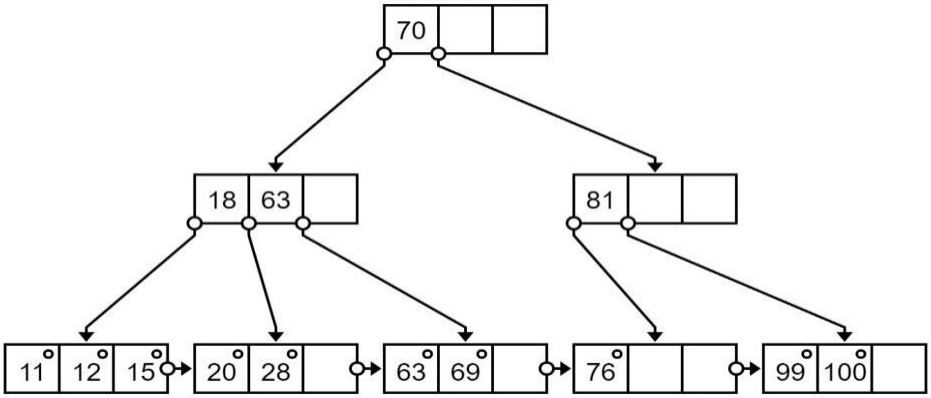


Let us assume that the following keys are deleted in the given order. Please show your tree after each deletion step and explain how you reach it.

a) Delete 104
b) Delete 12

(2 points each, 4 in total)

[Question 9] Consider the following B+ tree.



Explain how the latch crabbing/coupling protocol will work on this tree when

a) Finding 28
b) Inserting 77
c) Inserting 16

(2 points each, 6 in total)

[Question 10] We have a table with 2 million pages (N = 2,000,000 pages), and we want to sort it using external merge sort. Assume that the DBMS is not using double buffering or blocked I/O, and that it uses quicksort for in-memory sorting. Let B denote the number of buffers (Note: Write steps for every calculation you make along with the correct formulas; if steps are missing, points could be deducted).

a) Assume that the DBMS has 15 buffers. How many sorted runs are generated? Note that the final sorted file does not count towards the sorted run count (5 points).
b) Again, assume that the DBMS has 15 buffers. How many passes does the DBMS need to perform in order to sort the file (5 points)?
c) Again, assume that the DBMS has 15 buffers. How many pages does each sorted run have after the fourth pass (i.e. Note: this is Pass #3 if you start counting from Pass #0) (5 points)?

[Question 11] We have a database file with 1.5 million pages (N = 1,500,000 pages), and we want to sort it using external merge sort. Assume that the DBMS is not using double buffering or blocked I/O, and that it uses quicksort for in-memory sorting. Let B denote the number of buffers (Note: Write steps for every calculation you make along with the correct formulas; if steps are missing, points could be deducted).

a) Assume that the DBMS has 30 buffers. What is the total I/O cost to sort the file (5 points)?
b) What is the smallest number of buffers B such that the DBMS can sort the target file using only three passes (5 points)?
c) Suppose the DBMS has 200 buffers. What is the largest database file (expressed in terms of the number of pages) that can be sorted with external merge sort using three passes (5 points)?

[Question 12] We have a database with a buffer size of 102 buffers and two tables, R and S, with following properties:

- R: 1,000,000 pages and 100,000,000 tuples total
- S: 100,000 pages and 10,000,000 tuples total

Compute the total I/O cost and total time cost (at 0.2 ms/IO) of the following join algorithms on R and S.

a. Naïve Nested Loop Join with R as outer table and S as inner table (2 point).
b. Naïve Nested Loop Join with S as outer table and R as inner table (2 point).
c. Block Nested Loop with S as outer table and R as inner table (2 point).
d. Sort-Merge Join (2 point).

Which of the above join algorithms should be preferred for joining the tables R and S (2 point)?

Below are the answers to Q5 to Q12 attached

Vaishnavi Pawar
(vpawar@iu.edu)

M T W T F S S
Page No.:
Date:
YOUVA

Q5.

Hash table size 10

$h(k) = k \bmod 10$

1] Key 18
$h(18) = 18 \bmod 10 = 8$ (Placed 18 in slot 8)

2] Key 29
$h(29) = 29 \bmod 10 = 9$ (Placed 29 in slot 9)

3] Key 37
$h(37) = 37 \bmod 10 = 7$ (Placed 37 in slot 7)

4] Key 42
$h(42) = 42 \bmod 10 = 2$ (Placed 42 in slot 2)

5] Key 50
$h(50) = 50 \bmod 10 = 0$ (Placed 50 in slot 0)

6] Key 53
$h(53) = 53 \bmod 10 = 3$ (Placed 53 in slot 3)

7] Key 62
$h(62) = 62 \bmod 10 = 2$

Slot 2 is already occupied by 42, Try slot 3, occupied by 53. The next slot is empty, hence place 62 in slot 4.

0	1	2	3	4	5	6	7	8	9
50		42	53	62			37	18	29

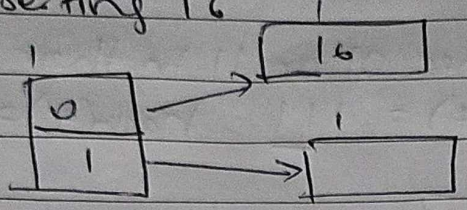
Q6.

Binary forms of given number:

16 - 10000
4 - 00100
6 - 00110
22 - 10110
24 - 11000
10 - 01010
31 - 11111
7 - 00111
9 - 01001
20 - 10100
26 - 11010

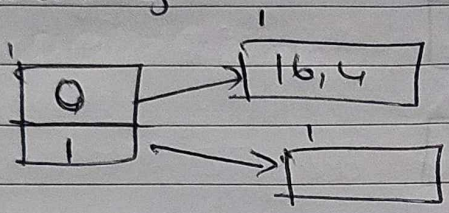
Initially,
Global depth = 1
Local depth = 1

Inserting 16



hash(16) = 10000
id = 0

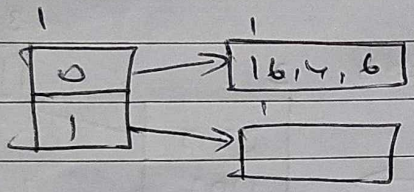
Inserting 4



hash(4) = 100

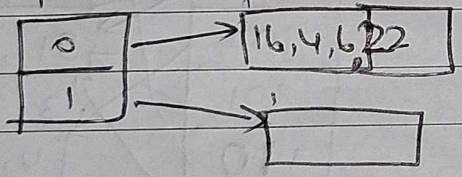
Inserting 6

hash(6) = 110

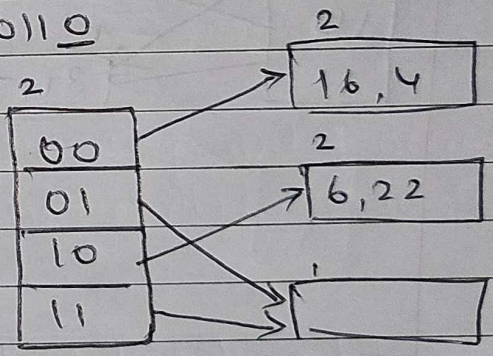


Inserting 22

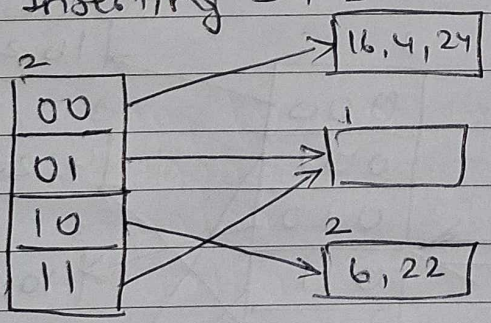
hash(22) = 10110



split

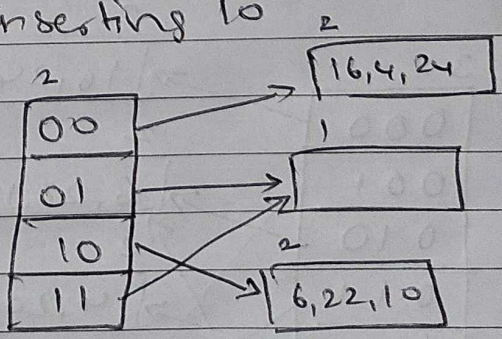


Inserting 24



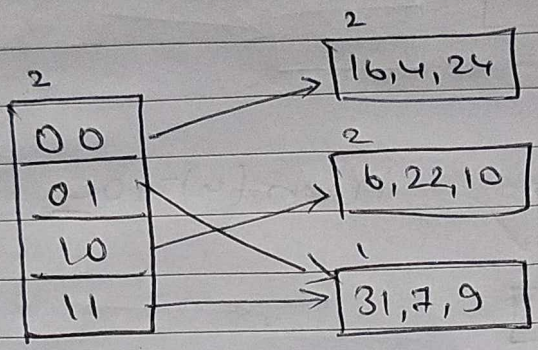
hash(24) = 11000

Inserting 10

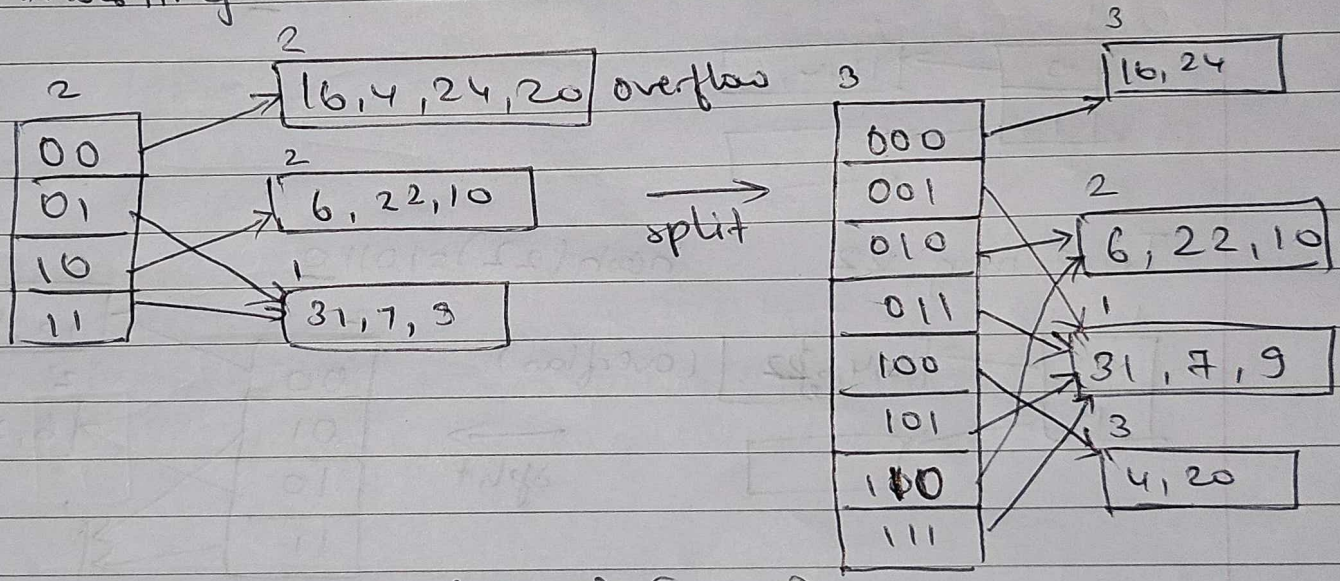


hash(10) = 1010

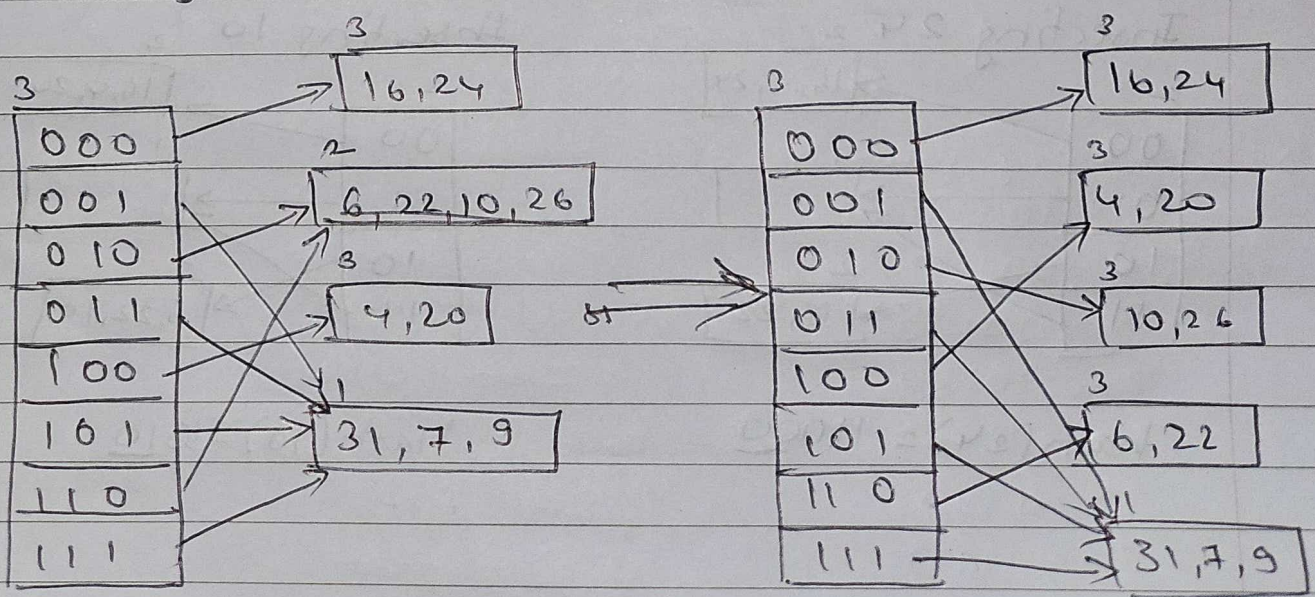
Inserting 31, 7, 9 together.
All the elements have either 01 or 11 in their LSBs
hash(31) = 11111 hash(7) = 111 hash(9) = 1001



Inserting 20 hash(20) = 10100



Inserting 26 hash(11010) (overflow)

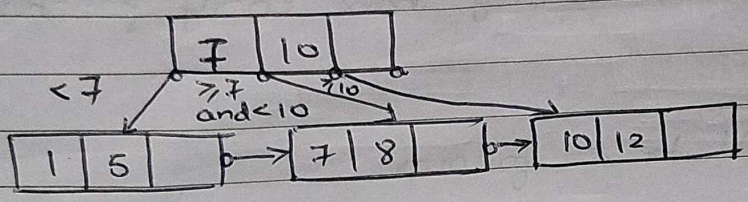


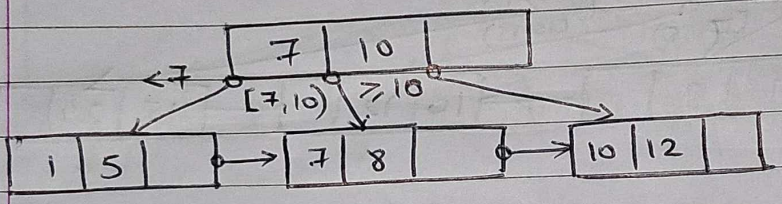
Q7. B+ Tree.

Insert.

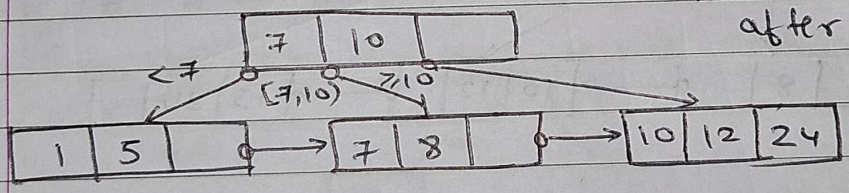
a] Insert 24.

The current B+ Tree





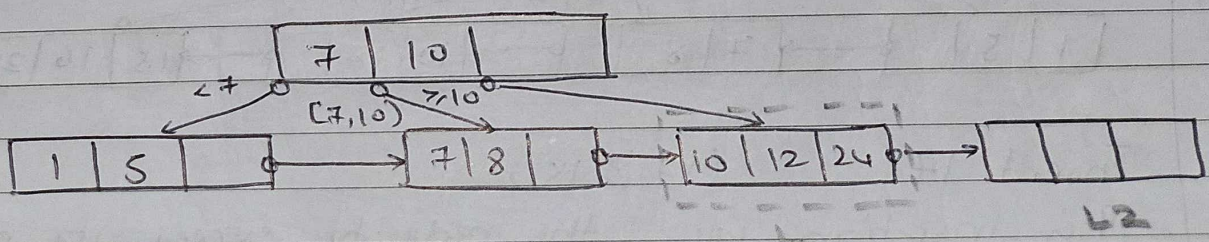
Insert 24 - can be inserted directly, as 24 ≥ 10, it will be inserted in the last leaf node after 12

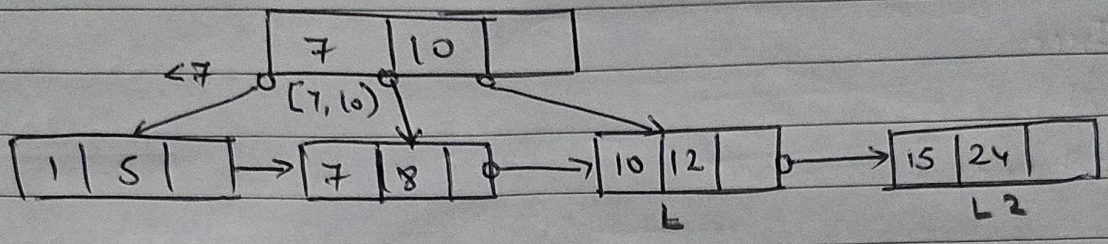


b) Insert 15

According to the sorting order as 15 ≥ 10, it will also be inserted into the last node, but as all the keys in the node are full. We can't add it there.

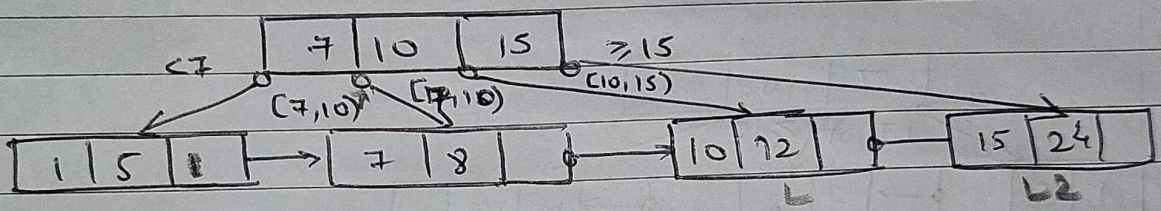
Hence, split the key into L and a new node L2.

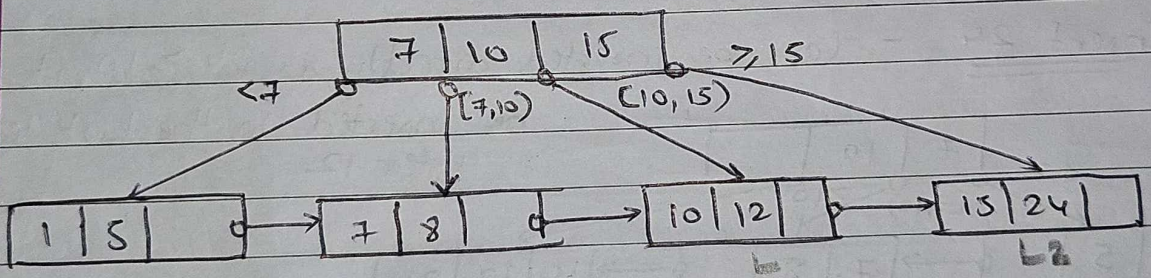




Here we will redistribute entries evenly, copy up middle key. The next step will be, inserting index entry pointing to L2 into parent of L

∴ The B+ Tree will be as follows

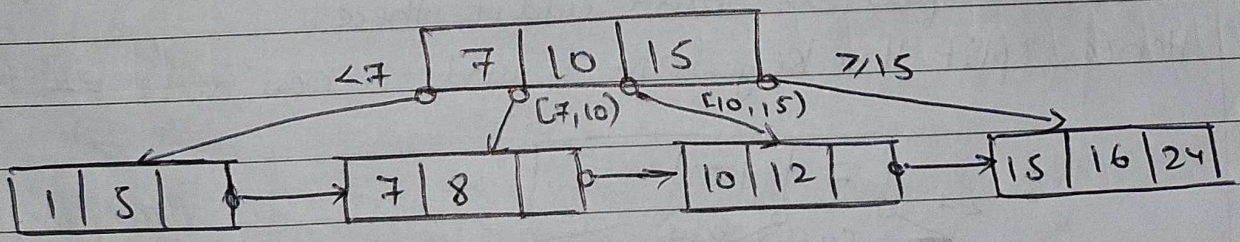




c) Insert 16

Again here the correct leaf node will be L2 ($16 \geq 15$)

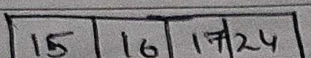
We can directly insert the data in sorted order, as it has enough space.



d) Insert 17. ($17 \geq 15$)

This insertion cause the node to exceed its maximum capacity. As the last node is already full.

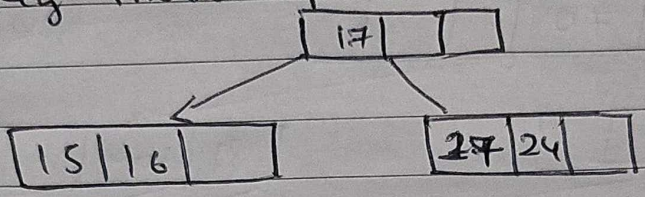
The node will look like this,



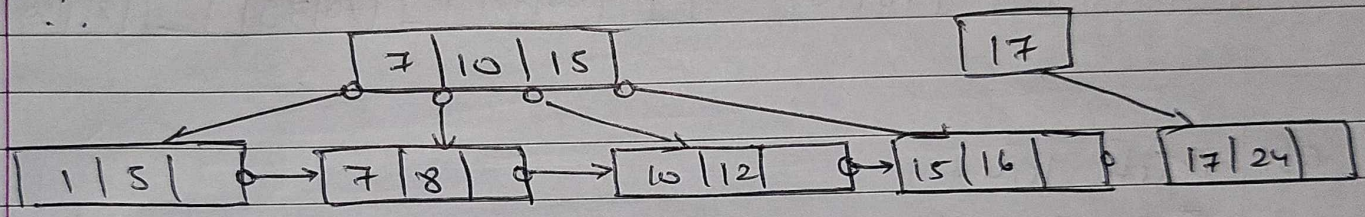
→ No space in the node as only 3 keys are allowed,

Hence split the node into two nodes, and the middle key moves up to become the new root.

Hence,

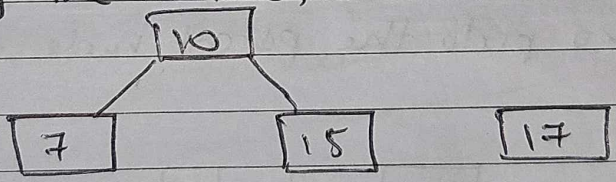


∴

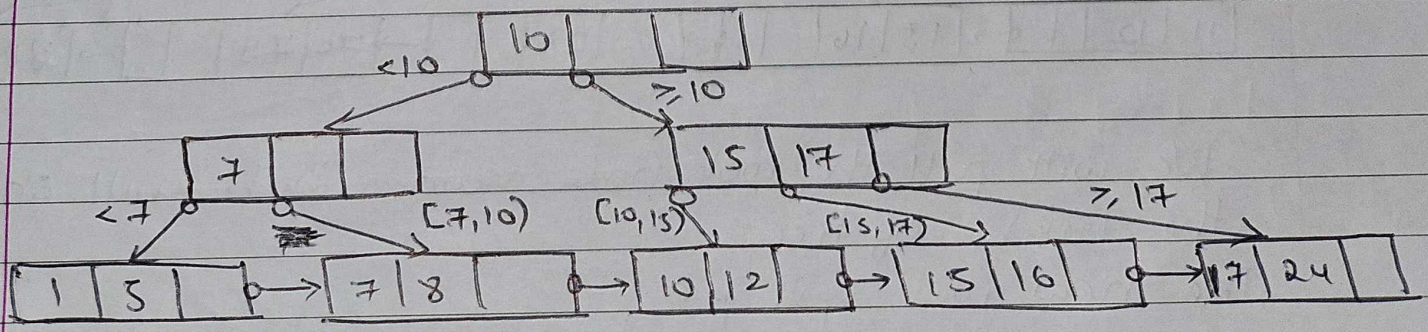


But, the node created is an orphan node. No parent node points to it. We can't insert 17 into the root node, as it is already full.

By splitting the root ~~a~~, we can insert 17.



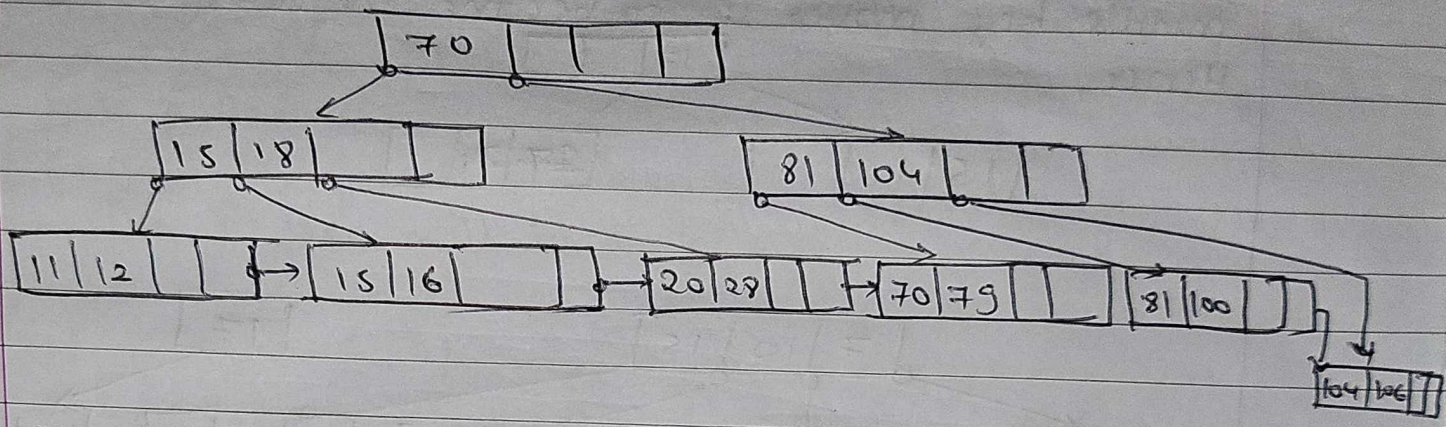
Next, after the ~~splitti~~ splitting the old root, point ~~to~~ the split nodes from the new root



Hence, all insertion completed.

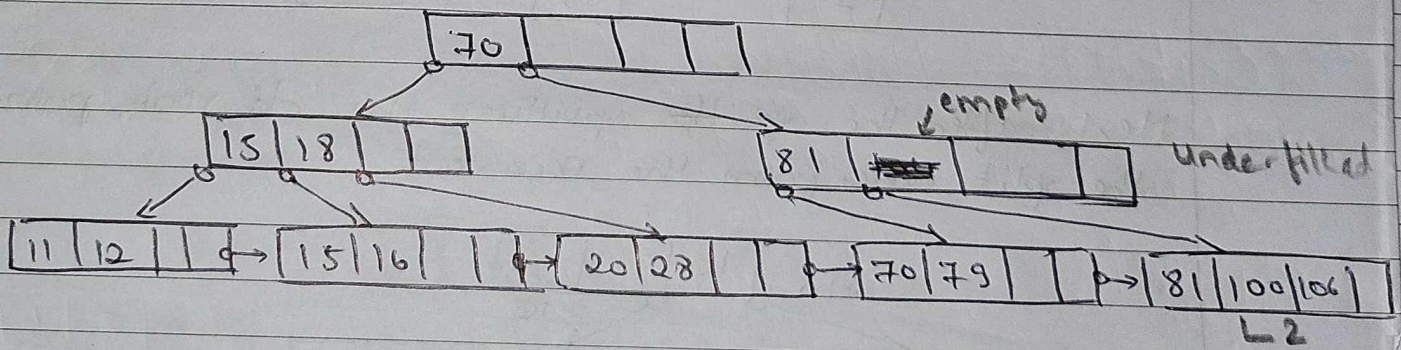
Q8. B+ Tree
Deletion

The given tree is

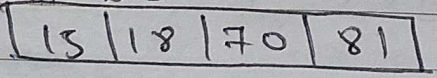


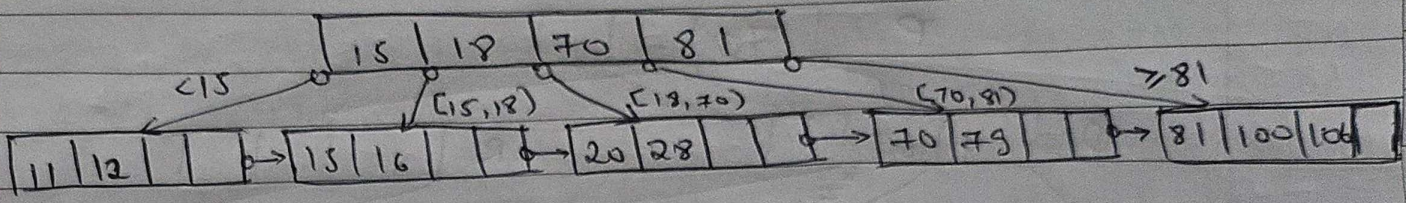
a) Delete 104.
104 is in the last Node, Deleting 104 will result the node to be underfilled. There are no "rich" neighbors to borrow from too.
Hence after deleting we will merge 106 with a sibling.

But that also puts the parent node of L2 to be under filled too



∴ We will pull the root node down, which will shrunk the height of tree.
The new root will be

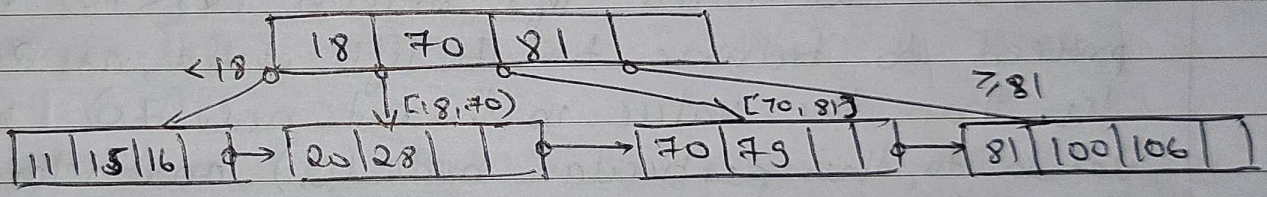




b) Delete 12.

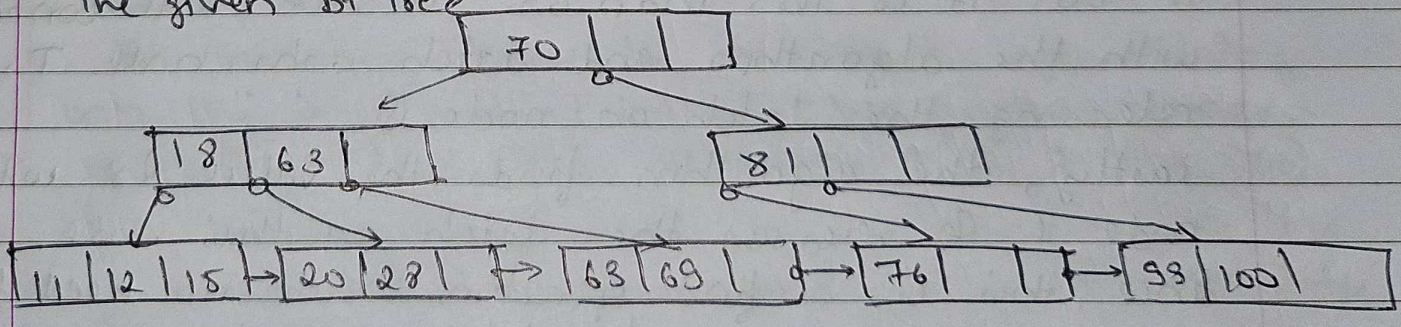
If we delete 12, leaf node L1 will be underfilled, we can redistribute from the sibling. However sibling [15,16) also has minimum number of keys (2 keys for M=4) so redistribution is not possible.

So, we merge 11 with sibling [15,16) to form [11,15,16] and after that ~~the~~ we need to remove the entry pointing to 12.



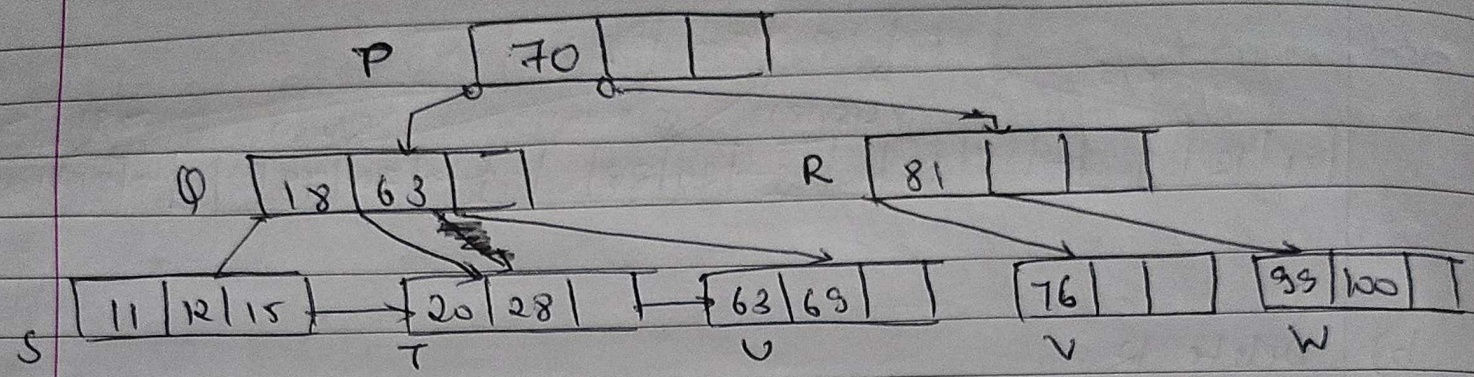
Hence, the deletion is completed.

Q9. The given B+ Tree



i) Finding 28

Latch crabbing is a technique used in B+ tree operations to ensure that the tree remains consistent and avoid deadlocks during concurrent access by multiple threads or processes.



1] Finding 28.

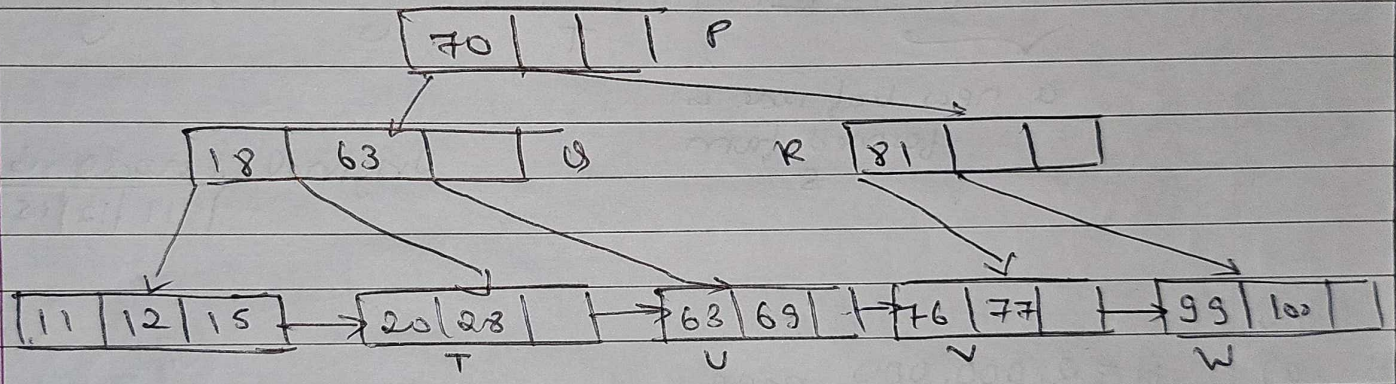
When we try to search value 28, we initially start from the root. Here I have named each node, root node being P and other nodes Q, R, S, T, U, V, W respectively.
We start with the left branch (28<70) & proceed to traverse to its left child, as 28 is less than the value in the root node (70). We latch to this particular node (Q) and release the latch from the root node P, as its a read operation.
Further as we traverse to find 28, 28 is greater than 18 but it is less than 63. So, we will continue with the algorithm and latch onto node T, while releasing the latch on node Q.
Lastly, the algorithm finds the value 28 within node T & releases the latch on this node.
Since, this is a find operation, there's no need to modify the tree. and the latches ~~are~~ are all released.

2) Inserting 77

When trying to insert 77, we can start ~~by~~ initially by latching onto the root node P, proceeding further ~~with~~

with the right child node R as 17 is less than 70. After latching to node R, we observe that there is open space for additional entries even if its child node splits. This allows us to release the latch from root node P. ~~f~~

Proceeding forward with the traversal, we move to node V, as the value 77 is less than 81. We release the latch on R node and latch onto V node, inserting value 77 into the leaf node V. Finally we can release the latch on node V, after the insertion.



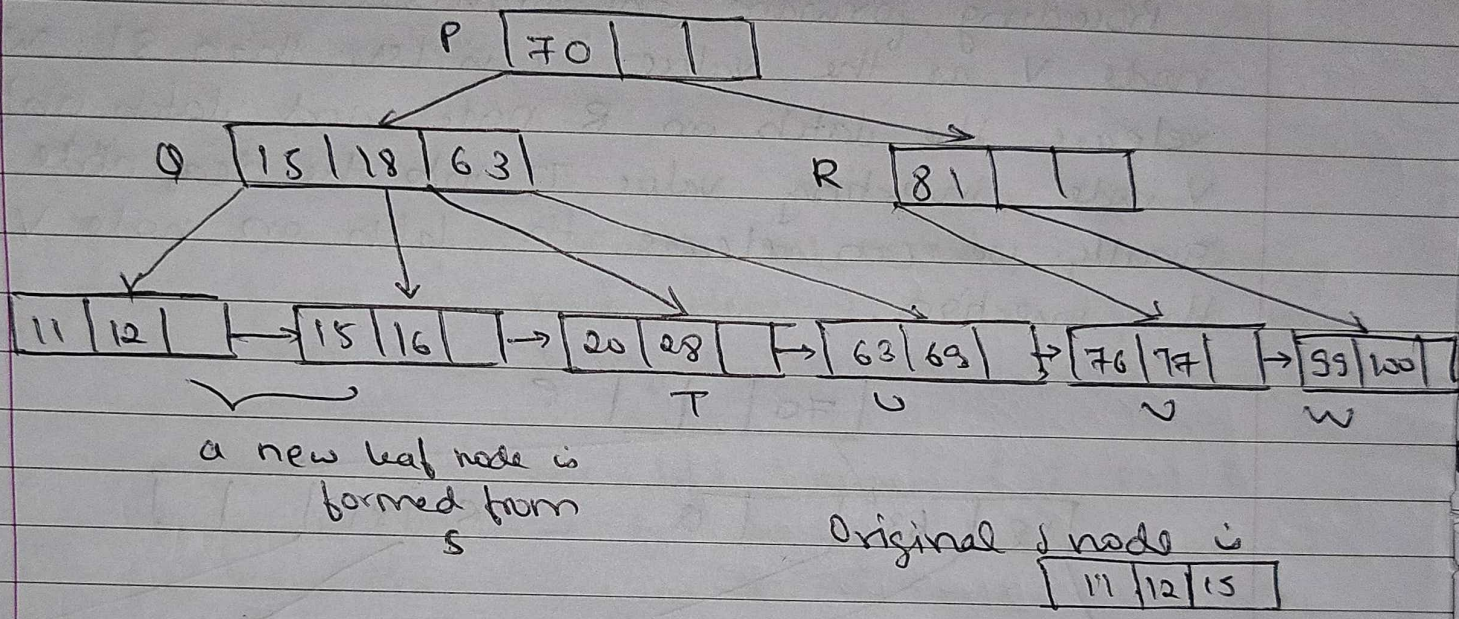
3] Inserting 16

For inserting value 16, we start by latching onto the root node P initially. We further proceed to its left child Q as 16 is less than 70 (16<70). After latching to Q node, by observing the B+ Tree, we notice that there is space for additional entries even if its child node splits, which allows us to release the latch from the root node P.

Further during the traversal, we move to node S, as 16 is less than 18 (16<18). But here the leaf node S is full with values [11,12,15]. Hence there occurs an overflow & so ~~to~~ we ~~M~~ maintain the

latch on node Q. We now apply a split operation on node S, generating a new leaf node and we update node Q accordingly.

Finally, we will insert the value and remove the latches from nodes Q and S both.



Q10.

a) $N = 2,000,000$ pages

Number of sorted runs $= N/B + N'/B_{-1} + N''/B_{-1} + \ldots$

Pass 0 $= N/B = \frac{2,000,000}{15} = 133334$

We get, 133334 sorted runs of 15 pages each.

∴ Pass 0 will ~~get~~ give 133334

Pass 1 $= N'/B_{-1} = \frac{133334}{14} = 9524$

Pass 1 will give 9524 sorted runs.

Pass 2 = $N''/B-1 = \frac{9524}{14} = 681$

Pass 2 gives 681 sorted runs.

Pass 3 = $N'''/B-1 = \frac{681}{14} = 49$

Pass 3 gives 49 sorted runs.

Pass 4 = $N''''/B-1 = \frac{49}{14} = 4$

Pass 4 gives 4 sorted runs.

∴ Total Number of sorted runs = $N/B + N'/B-1 + N''/B-1 \ldots$

$= \frac{2000000}{15} + \frac{133334}{14} + \frac{9524}{14} + \frac{681}{14} + \frac{49}{14}$

$= 133334 + 9524 + 681 + 49 + 4$

$= 143592$

b] Number of Passes = $1 + (\log_{B-1}(N/B))$

$= 1 + (\log_{14}(\frac{2000000}{15}))$

$= 1 + (\log_{14} 133334)$

$\log_{14} 133334 = 4.47$

∴ Number of Passes = $1 + 5$

$= 6$

Number of Passes = 6

c] For the first pass, buffer pages will be used.
We have B = 15.
∴ For the first pass, number of pages for sorted run = 15
For all the ~~oth~~ further passes, B-1 pages will be used.

∴ Second Pass = 15 × 14
= 210 pages
Third pass = 210 × 14
= 2940 pages
Fourth pass = 2940 × 14
= 41160 pages

After 4 pass we have 41160 pages.

Q11. N = 1500000 pages
B = 30 buffers.

a] Total I/O cost = 2N × (no. of passes)
No. of passes = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
$= 1 + \lceil \log_{30-1} \left\lceil \frac{1500000}{30} \right\rceil \rceil$
$= 1 + \lceil \log_{29} 50000 \rceil$

$\log_{29} 50000$ is $3.2 \approx 4$

∴ No. of passes = 1 + 4
= 5

I/O cost = $2 \times 1500000 \times 5$

cost = 15000000

b] No. of passes = $3 = 1 + \lceil \log_{B-1}(\lceil N/B \rceil) \rceil$

For B = 50

$= 1 + \lceil \log_{B-1}\left(\frac{1500000}{50}\right) \rceil$

$= 1 + \lceil \log_{49} 30000 \rceil$

$= 1 + 3$

$= 4$

For B = 100

$= 1 + \lceil \log_{99}\left(\frac{15000000}{100}\right) \rceil$

$= 1 + \lceil \log_{99} 15000 \rceil$

$= 1 + 3 = 4$

For B = 116

$= 1 + \lceil \log_{115} \lceil \frac{1500000}{116} \rceil \rceil$

$= 1 + \lceil \log_{115}(12932) \rceil$

$= 1 + 2 = 3$

Alternative method,

$N = 1 + \log_{B-1}(\lceil N/B \rceil)$

$N - 1 = \log_{B-1}(N/B)$

$\rightarrow N \leq (B-1)^{\text{No. of Passes}}$

No. of passes = 3

$= 1500000 \leq (B-1)^3$

= 116 buffer

c) We know $P = 1 + (\log_{B-1}(N/B))$

$P = 3$

$2 = \log_3 (N/B)$

By solving above equation,

$\therefore \frac{N}{B} = (B-1)^2$

$N = B(B-1)^2$

We are asked to calculate the largest database file, we can calculate it by directly inputting the value of $B = 200$ in the above equation.

$\therefore N = 200 \times (200-1)^2$

$= 200 \times (199)^2$

$= 200 \times 39601$

$N = 7920200$

Q12. Given,

a) $M = 1000000$

$m = 1 \times 10^8$

$N = 1 \times 10^5$

Cost = M + (m . N)
= $10^6 + (10^8 \cdot 10^5)$
= $10^5 (10 + 10^8)$
= $10^5 (100000000 + 10)$
= $10^5 (1000000010)$
= 100000001000000 IOs

Rate = 0.2 ms / IO
Time = 100000001000000 × 0.2
= 20000002000000 ms
≈ 5555555.6 hrs.

∴ Time = 20000002000000 ms

b) N(S) = 100000
n = 10000000
M = 1000000

I/O Cost = N + (n × M)
= 100000 + (10000000 × 1000000)
= $10^5 + (10^7 \times 10^6)$
= $10^4 (10 + 10^9)$
= $10^4 \times 1000000010$
= 100000001 00000 IOs

Total time cost = (N + (n × M)) × 0.2
= 20000000 20000 ms

c) Total cost = N + ([N/(B-2)] * M)
= 100000 + ([100000/102-2] × 100 0000)
∵ N=100000, M=1000000

$= 100000 + (1000 \times 1000000)$

$= 100000 + 1000000000$

Total I/O Cost $= 1000100000$ IOs

Total time cost $= (N + \{[N/(B-2)] * M\}) * 0.2$

$= 200020000$ ms

d] Total cost = sort + merge

$\text{sort cost}(R) = 2M\left(1 + [\log_{B-1}(M/B)]\right)$

$\log_{B-1}(M/B) \approx 2$

$\therefore \text{cost}(R) = 2M \times (1+2)$

$= 2M \times 3$

$= 6M$

Sort cost $(R) = 6000000$ $\quad M = 1000000$

$\text{sort cost}(S) = 2N\left(1 + [\log_{B-1}(N/B)]\right)$

$= 2 \times N \times (3)$

$= 2 \times 100000 \times 3$ $\quad \therefore N = 100000$

Sort cost $(S) = 600000$

Merge cost $= M + N$

$= 1000000 + 100000$

$= 1100000$

∴ Total cost = ~~00~~ Sort + Merge

= 6000000 + 600000 + 1100000

Total cost = 7700000 IOs.

Total time = Total cost × 0.2

= 7700000 × 0.2

= 1540000 ms

All the observations, say that sort-merge join (d) has the lowest total IO cost and time cost Hence, it should be the preferred one.

∴ Sort-Merge Join algorithm should be preferred for joining the tables R and S.

[Question 13] Consider the following tables *R* and *S*

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

SELECT *R*.*, *S*.*
FROM *R* JOIN *S* ON *R*.id = *S*.id;

Join the two tables given above using Sort-Merge join algorithm along with the steps to show the positions of cursor$_R$ and cursor$_S$ at each step, and the output buffer table after each step (The table are already sorted so you may skip the sorting steps, but you need to show each step of the merging phase) (10 points).

Ans:

Step 1:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10

Step 2:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10

Step 3:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12

Step 4:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12

Step 5:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15

Step 6:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20

Step 7:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20
30	Aegon Targaryen	30	20000	2015-08-17

Step 8:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20
30	Aegon Targaryen	30	20000	2015-08-17

Step 9:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20
30	Aegon Targaryen	30	20000	2015-08-17

Step 10:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20
30	Aegon Targaryen	30	20000	2015-08-17
30	Jon Snow	30	5000	2013-08-15

Step 11:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20
30	Aegon Targaryen	30	20000	2015-08-17
30	Jon Snow	30	5000	2013-08-15
30	Jon Snow	30	10000	2014-08-20

Step 12:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20
30	Aegon Targaryen	30	20000	2015-08-17
30	Jon Snow	30	5000	2013-08-15
30	Jon Snow	30	10000	2014-08-20
30	Jon Snow	30	20000	2015-08-17

Step 13:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20
30	Aegon Targaryen	30	20000	2015-08-17
30	Jon Snow	30	5000	2013-08-15
30	Jon Snow	30	10000	2014-08-20
30	Jon Snow	30	20000	2015-08-17

Step 14:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20
30	Aegon Targaryen	30	20000	2015-08-17
30	Jon Snow	30	5000	2013-08-15
30	Jon Snow	30	10000	2014-08-20
30	Jon Snow	30	20000	2015-08-17
40	Arya Stark	40	8000	2013-10-05

Step 15:

R (id, name)

id	name
10	Rob Stark
20	Sansa Stark
30	Aegon Targaryen
30	Jon Snow
40	Arya Stark
50	Ben Stark

S (id, money, cdate)

id	money	cdate
10	10000	2013-08-10
20	5000	2013-09-12
30	5000	2013-08-15
30	10000	2014-08-20
30	20000	2015-08-17
40	8000	2013-10-05

R.id	R.name	S.id	S.money	S.cdate
10	Rob Stark	10	10000	2013-08-10
20	Sansa Stark	20	5000	2013-09-12
30	Aegon Targaryen	30	5000	2013-08-15
30	Aegon Targaryen	30	10000	2014-08-20
30	Aegon Targaryen	30	20000	2015-08-17
30	Jon Snow	30	5000	2013-08-15
30	Jon Snow	30	10000	2014-08-20
30	Jon Snow	30	20000	2015-08-17
40	Arya Stark	40	8000	2013-10-05

[Question 14]

Consider the following tables *R* and *S*:

R (id, name)

id	name
11	Rob Stark
21	Sansa Stark
12	Jon Snow
22	Arya Stark
32	Ben Stark

S (id, money, cdate)

id	money	cdate
11	10000	2013-08-10
11	5000	2013-09-12
12	5000	2013-08-15
12	10000	2014-08-20
12	20000	2015-08-17

SELECT *R*.*, *S*.*
FROM *R* JOIN *S* ON *R*.id = *S*.id;

Suppose the tables given above need to be joined on the "id" columns using a Simple Hash Join algorithm. Please build 2 hash tables, HT_R for R and HT_S for *S*, using the hash function $h(x) = x$ MOD 10 (for example, $x = 15$ yields $h(x) = h(15) = 15$ MOD 10 = 5) where MOD is the modulus operator. Assume that each hash bucket can store up to 10 values (5 points).

After building the hash tables for *R* and *S*, find the matching records and display those records in a table with all their 5 attributes (5 points).

Finally, compute the number of comparisons you made using the hash table to find the matching records using Simple Hash Join algorithm and the number of comparisons you would have to make if a Naïve Nested Loop Join was used instead (Show your calculations) (5 points).

Answer:

Below are the hash values for each ID from R and S

R (id, name)

id	name
11	Rob Stark
21	Sansa Stark
12	Jon Snow
22	Arya Stark
32	Ben Stark

HTR

11
21
12
22
32

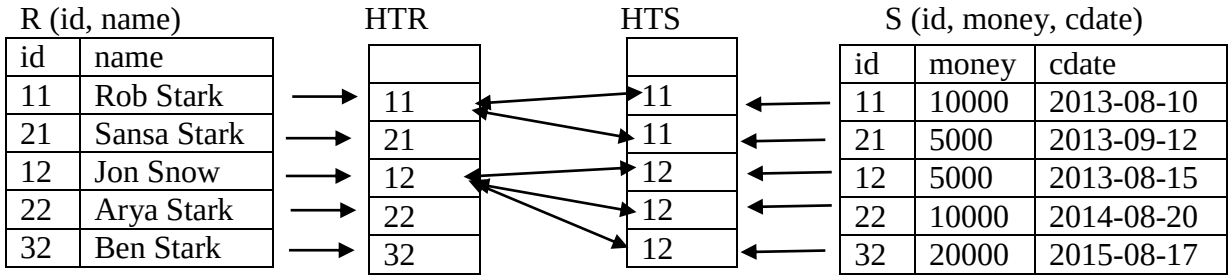
HTS

11
11
12
12
12

S (id, money, cdate)

id	money	cdate
11	10000	2013-08-10
21	5000	2013-09-12
12	5000	2013-08-15
22	10000	2014-08-20
32	20000	2015-08-17

Below is the Matching IDs mapping



Below is the output of Matching Records,

R.id	**R.name**	**S.id**	**S.money**	**S.cdate**
11	Rob Stark	11	10000	2013-08-10
11	Sansa Stark	11	5000	2013-09-12
12	Jon Snow	12	5000	2013-08-15
12	Jon Snow	12	10000	2014-08-20
12	Jon Snow	12	20000	2015-08-17

Number of Comparisons using Hash Join,

- To calculate the number of comparisons using Hash Join, consider the number of entries in buckets with hash values 1 and 2
- Assume buckets with hash values 1 and 2 for table R have XR and YR number of elements, respectively.
- Assume buckets with hash values 1 for table S have XS number of element and buckets with hash value 2 for table R is YS number of elements
- Accordingly, XR=2, XR=3, XS=2 and YS = 3
- Hence, Total Number of comparisons = (XR * XS) + (YR + YS)
  = (2*2) +(3*3)
  = 4 + 9
  = 13

  Hence, Total Number of comparisons = 13

Number of comparisons using Naïve Nested Join
= (Number of elements in HTR) * (Number of elements in HTR)
= 5 * 5
= 25
Hence, Number of comparisons using Naïve Nested Join = 25

[Question 16] What are the key benefits of the vectorization model in query processing compared to traditional row-based processing (2.5 + 2.5 = 5 points)?

Ans: Vectorization in query processing is the way of processing data in batches of rows at a time, as opposed to one row at a time as in traditional row-based processing. This architecture, commonly used in modern columnar databases and data analytics platforms, provides several significant advantages over traditional row-based processing:

- Reduced Instruction Overhead: In typical row-based processing, each row operation results in CPU instruction overhead. Vectorized processing decreases overhead by allowing a single instruction to act on numerous data items, reducing the total number of instructions executed and increasing CPU performance.
- Higher Cache Hit Rates: Vectorized processing often works with data stored in a columnar format, which organizes data by columns rather than rows. This data organization improves the locality of reference for operations that handle huge amounts of data from a single column, resulting in greater cache hit rates. Higher cache efficiency enables faster data processing, decreasing the need for slower memory accesses.
- Batch Processing: Vectorized query processing enables operations to be performed on whole batches of data at once, rather than on individual rows. This approach is more consistent with modern CPU architecture, which is designed to perform activities in parallel. Vectorization makes the best use of CPU SIMD (Single Instruction, many Data) capabilities by processing many data points at the same time.
- Reduced Data Movement: Processing data in columns requires less data movement in memory, which is advantageous because transporting data between different levels of memory hierarchy i.e. from disk to RAM, RAM to CPU caches, is frequently a bottleneck in data processing. Vectorization reduces this movement, particularly for actions that only affect a subset of columns, as is frequent in analytical queries.

[Question 17] Describe the concept of a "Halloween Problem" in database systems and explain how it is typically addressed in modern DBMS (3 + 2 = 5 points).

Ans:

- The "Halloween Problem" happens in database systems when certain update or delete actions change a row's physical position within a table. This update may cause the row to be revisited or skipped in the same process, resulting in inaccurate or unexpected outcomes.
- When a UPDATE operation changes the value of an indexable column, the row's location inside that index may change. If the row advances in the order, the procedure may process it again, potentially resulting in an infinite loop in which the row fits the criteria for update and is constantly placed ahead of the scan cursor.
- If the updated row moves backward in the index, other rows that need to be updated may be skipped since the database engine has already passed their new position in the index.
- In severe instances, particularly when an update results in a row that continues to match the operation's conditions, an infinite loop may arise. This causes the procedure to constantly update the same row without moving on to others, consuming resources and potentially causing system hangs or crashes.
- These actions might cause database integrity issues, as the data may become inconsistent. This is particularly difficult in transactional systems, where data quality and consistency are critical.

Addressing the Halloween Problem

Modern DBMSs have evolved several strategies to alleviate the Halloween Problem while maintaining data integrity and consistency:

- Query Optimization: DBMS optimizers can recognize probable Halloween scenarios and adapt the query plan to separate the read and write phases, such as employing joins or access paths that are unaffected by row order changes.
- Locking Mechanisms: Locks can be added to the rows or the range of rows can be modified to keep them from moving throughout the operation.
- Snapshot Isolation: Using snapshot isolation, transactions can work on a version of the data that existed at the beginning of the transaction, ignoring changes that occur throughout the transaction.
- Temporary Tables/Spooling: Before making updates, the database copies the impacted rows into a temporary table or buffer to ensure that modifications do not interfere with the ongoing table scan that determines which rows to update.