CSE 4/560

Fall 2024

Databases and Query Languages Homework -3

Sahithya Arveti Nagaraju (Person number – 50559752 UBITName - sarvetin)

Problem 1

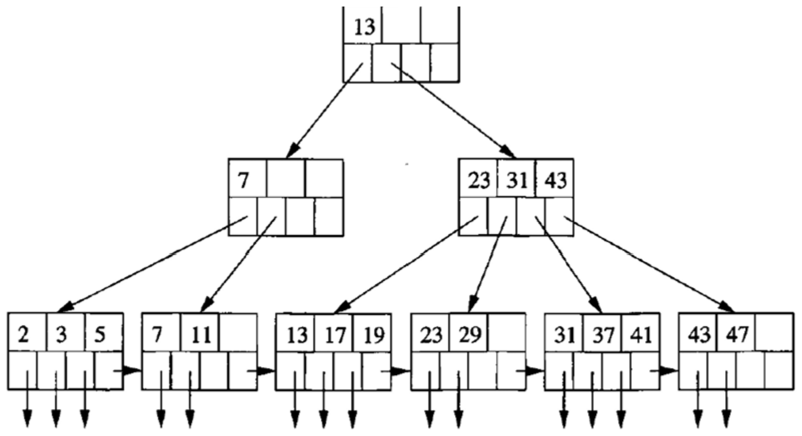


Figure 14.13: A B-tree

Execute the following operations on Fig. 14.13. Describe the changes for operations that modify the tree.

a) Lookup the record with key 41.

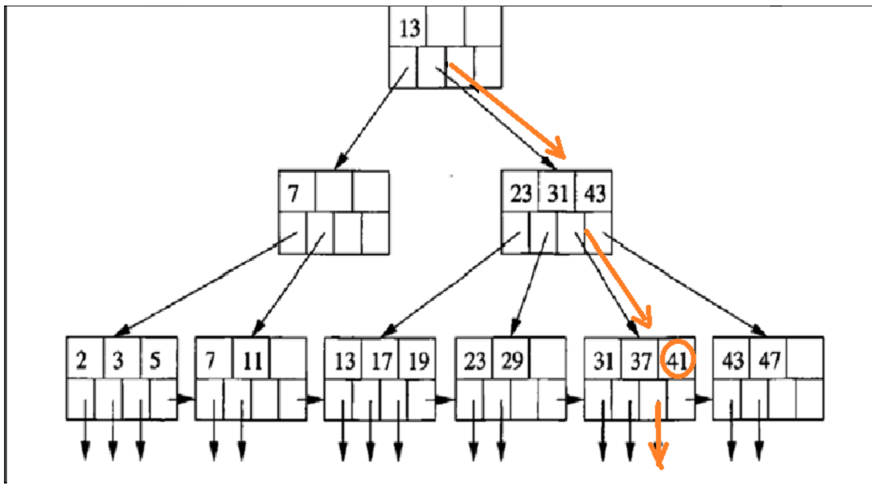
Ans: To locate the key 41, we need to traverse the B-tree as below,

- Start at root 13. 41 is grater the 13 so move to right side.
- In the subtree (with nodes 23, 31, 43) 41 is between 31 and 43 31<41<43 follow the pointer between 31 and 43.
- The leaf node contains keys 31, 37, 41. There is 41 in this node. The record with key 41 is found.

The record with key 41 is located in the leaf node containing keys **31, 37, 41**. Since this is a lookup operation, the tree will not be modified.

Lookup operation:

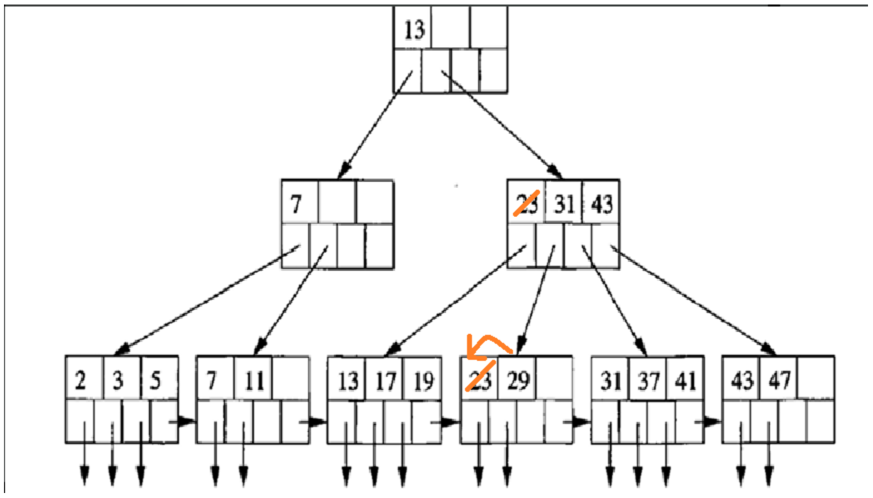
Select * from R where k == 41;

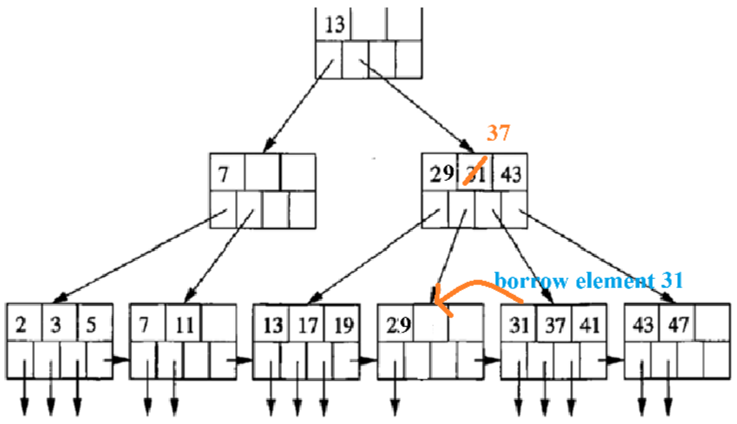


b) Delete the record with key 23.

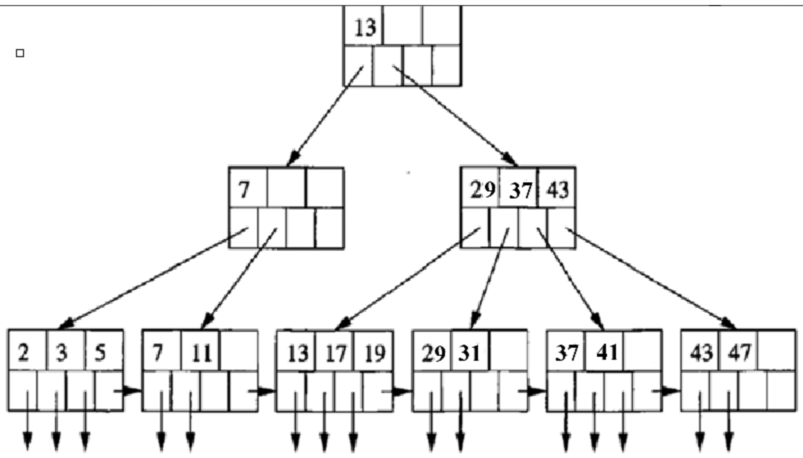
Ans:

- Start at root 13. 23 > 13 so move to right subtree.
- In intermediate level, node (23, 31, 43) key 23 is found and it the smallest key in that node.
- 23 is in intermediate node and in leaf node (23, 29).
- To delete it in a B-tree, we need to redistribute the keys.
- Remove key 23 from leaf node (23, 29). Since this node still satisfies the minimum key constraint (minimum of 1 key per node) no merging is required.
- Delete key in intermediate node as well and replace it with 29. Therefore, intermediate node will be (29, 31, 43) after removing 23.
- After removing leaf node (29) violates leaf constraint of minimum number of nodes for leaf node (that is 2). So, borrowed an element i.e. 31 from next node and changed the intermediate nodes (29, 31, 43) accordingly.





Final tree after deleting key 23:

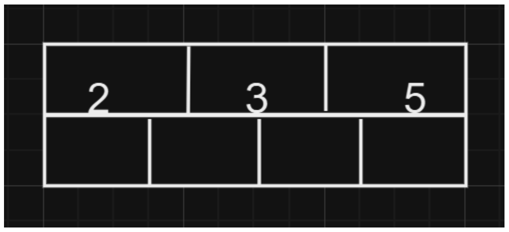


Problem 2

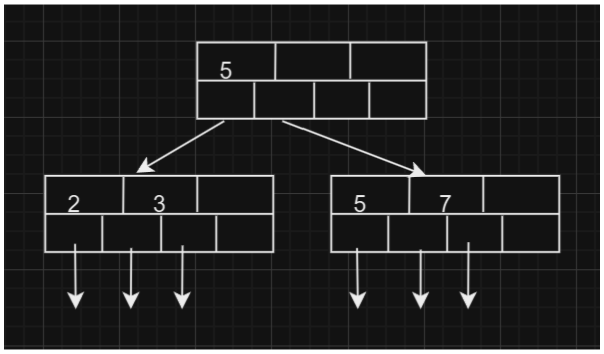
Construct a B+-tree for the following set of key values: (2, 3, 5, 7, 11, 17, 19, 23, 29, 31) Assume that the tree is initially empty and values are added in ascending order. Construct B+-trees for the cases where the number of pointers that will fit in one node is Four

Ans: A B+-tree with 4 keys a node will have 5 pointers

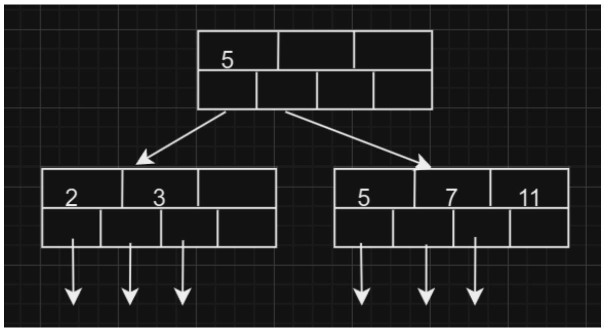
- Insert 2, 3, 5: Added to root node



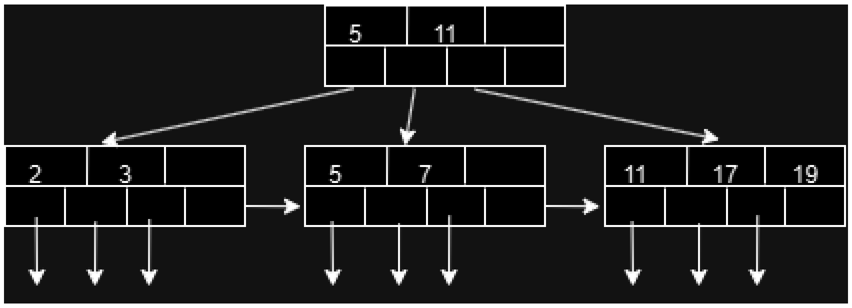
- Insert 7: causes root to exceed its capacity (4 keys). Split the root by taking 5 as middle element.



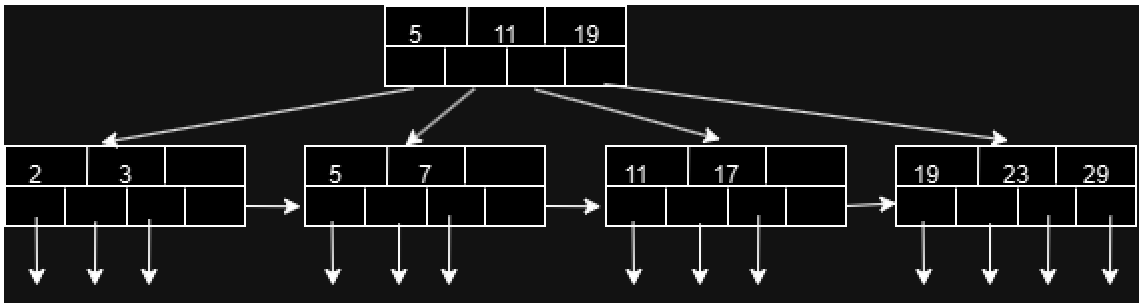
- Insert 11: Added to the node (5, 7)



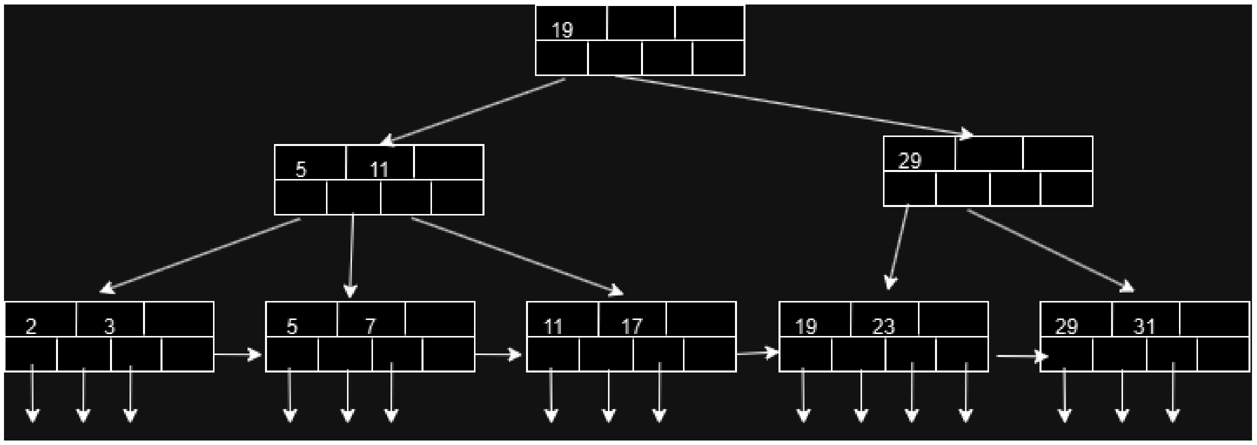
- Insert 17 and 19: causes node (5, 7, 11) to exceed its capacity. Split this node and keep 11 in root node. Then, added 19 to node (11, 17)



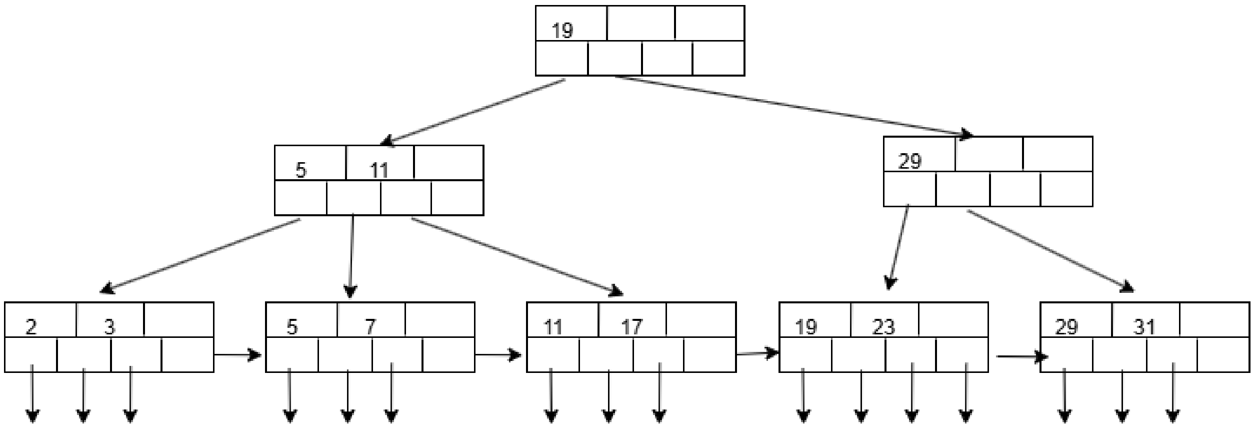
- Insert 23 and 29: causes node (11, 17, 19) to exceed its capacity. Split this node and keep 19 in root node. Then, added 29 to node (19, 23)



- Insert 31: causes node (19, 23, 29) to overflow, split that node and keep 29 middle key in root node. However, the root node is also full therefore split the root node by keeping 19 in root.



Final B+-tree:



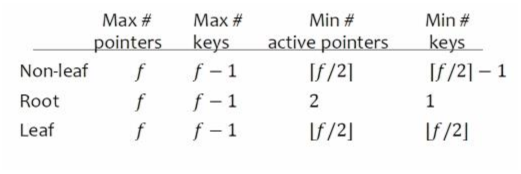
Problem 3

What are the minimum numbers of keys and pointers in B-tree (i) interior/interim nodes and (ii) leaves, when:

a) n = 10; i.e., a block holds 10 keys and 11 pointers.

b) n = 11; i.e., a block holds 11 keys and 12 pointers.

Ans:



	Max # pointers	Max # keys	Min # active pointers	Min # keys
Non-leaf	f	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f-1$	2	1
Leaf	f	$f-1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

a) n = 10, f = 11
   i. interior/interim nodes: minimum number of keys – ceil(11/2) - 1 = 5. Minimum number of pointers – ceil(11/2) = 6
   ii. leaves: minimum number of keys – ceil(11/2) = 5. Minimum number of pointers – ceil(11/2) = 5

b) n = 11, f = 12
   i. interior/interim nodes: minimum number of keys – ceil(12/2) - 1 = 5. Minimum number of pointers – ceil(12/2) = 6
   ii. leaves: minimum number of keys – ceil(12/2) = 6. Minimum number of pointers – ceil(12/2) = 6

Problem 4

Consider the Extendible Hashing index shown in Figure 11.1. Assume The Hash function is Mod 8. Answer the following questions about this index:

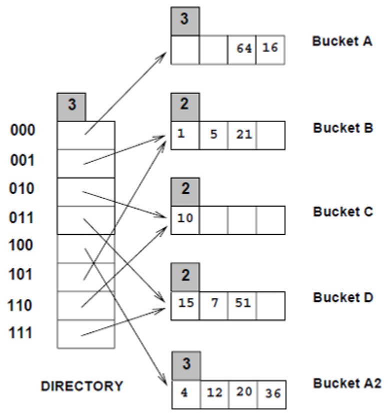


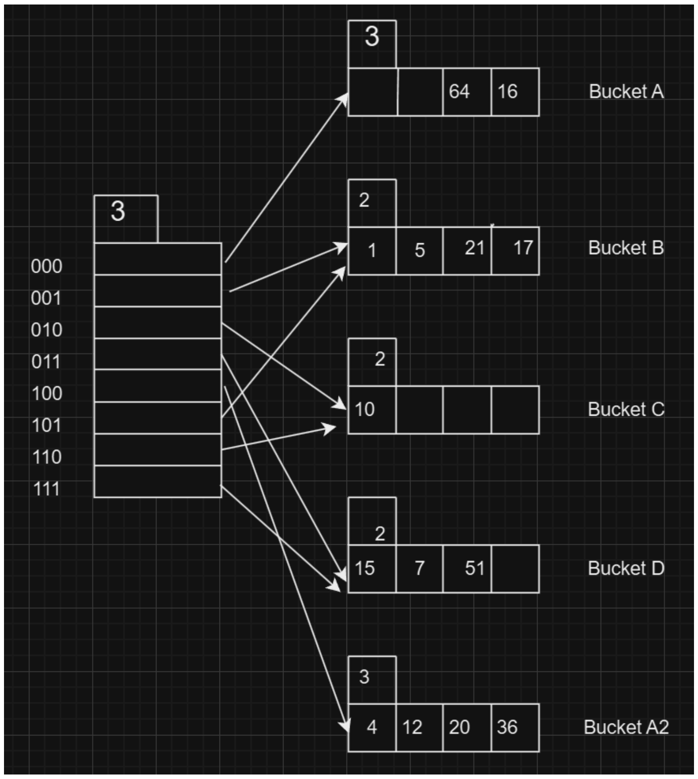
Figure 11.1 Figure for Exercise 11.1

a. Show the index after inserting entries with key values 17 and 69

b. Show the index after inserting an entry with key 68.

Ans:

a. Show the index after inserting an entry with key values 17 and 69
   Inserting 17: 17 mod 8 = 1 (001) This key will be added to Bucket B.

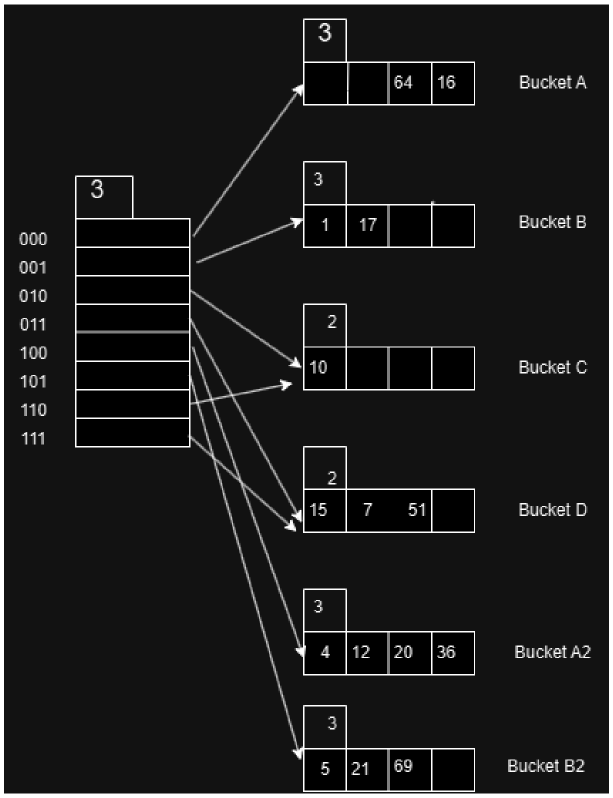


Inserting 69: 69 mod 8 = 5 (101) implies key should be added to Bucket B but this will lead to overflow. So, split the bucket B and change the depth of Bucket B to 3. After that redistribute the keys in Bucket B:

- 1 mod 8 = 1 (001) Bucket B
- 5 mod 8 = 5 (101) Bucket B2
- 21 mod 8 = 5 (101) Bucket B2
- 17 mod 8 = 1 (001) Bucket B
- 69 mod 8 = 5 (101) Bucket B2

Result After Redistribution:
Bucket B: [1,17]
New Bucket B2: [5, 21, 69]

Below is how it looks after inserting 69.

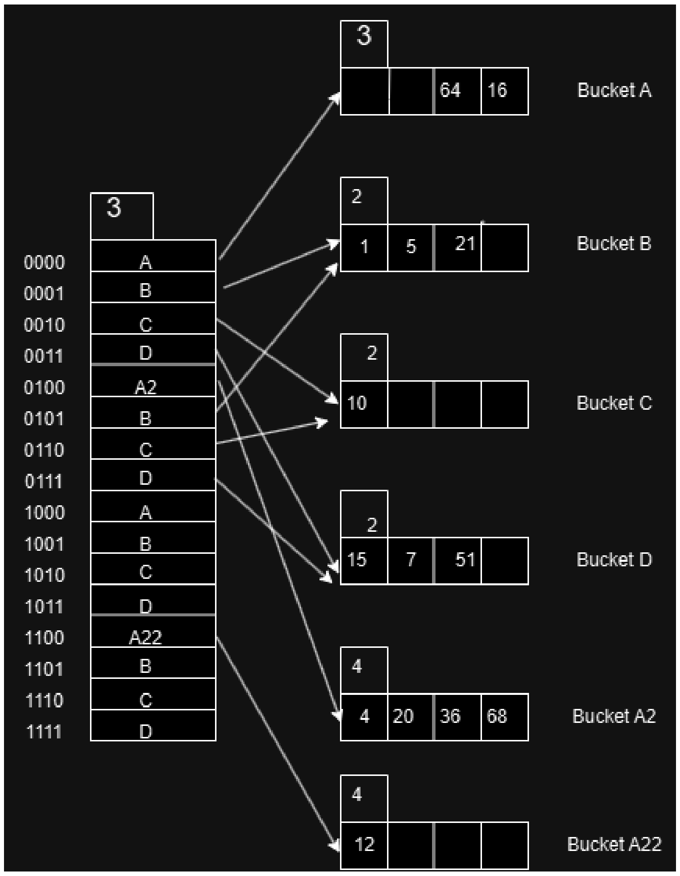


b. Show the index after inserting an entry with key 68
The hash of 68 is 68 mod 8=4 (binary: 100), which maps to Bucket A2. However, Bucket A2 is already full, causing an overflow. To resolve this:
Split Bucket A2:
Since the local depth of Bucket A2 equals the global depth, the directory is doubled.
  - The global depth increases to 4, and the hash function is updated to mod 16.

Redistribute Keys in Bucket A2:
Keys in Bucket A2 ([4,12,20,36,68]) are redistributed based on their new hash values (mod 16):

- 4 mod 16 = 4 (0100) Bucket A2
- 12 mod 16 = 12 (1100) Bucket A22
- 20 mod 16 = 4 (0100) Bucket A2
- 36 mod 16 = 4 (0100) Bucket A2
- 68 mod 16 = 4 (0100) Bucket A2

Result After Redistribution:
Bucket A2: [4,20,36,68]
New Bucket A22: [12]



Problem 5

Differentiate between Linear and Extensible Hashing

Ans:

Below are the differences between Linear and Extensible Hashing:

1. Growth Mechanism:
    - Linear Hashing: Grows by splitting one bucket at a time in a predetermined order.
    - Extensible Hashing: Grows by doubling the directory and splitting the overflowing bucket.
2. Overflow Handling:
    - Linear Hashing: Splits the bucket that overflows and redistributes its records.
    - Extensible Hashing: Only the overflowing bucket is split, and the directory is updated.

3. Directory Requirement:
    - Linear Hashing: Does not require a directory.
    - Extensible Hashing: Requires a directory to manage the buckets.
4. Complexity:
    - Linear Hashing: Simpler to implement and manage.
    - Extensible Hashing: More complex due to the need to maintain and update the directory.
5. Performance:
    - Linear Hashing: Efficient for dynamic changes but may need more overflow space.
    - Extensible Hashing: Better space utilization and efficient handling of large datasets.
6. Space Utilization:
    - Linear Hashing: May require more overflow space.
    - Extensible Hashing: Generally better space utilization.
7. Search Cost:
    - Linear Hashing: Search cost can increase if there are many overflow buckets.
    - Extensible Hashing: It has a constant search cost due to the directory.
8. Insertion Cost:
    - Linear Hashing: Insertion can be more costly if it triggers a bucket split.
    - Extensible Hashing: Insertion cost is generally lower due to efficient directory management.
9. Deletion Cost:
    - Linear Hashing: Deletion is straightforward but may leave empty buckets.
    - Extensible Hashing: Deletion is efficient but requires directory updates.
10. Scalability:
    - Linear Hashing: Scales linearly with the number of buckets.

- Extensible Hashing: Scales exponentially with the directory size.