Tutorial 7

Data Indexing

B+ Tree Review

- A balanced tree
- Each node can have at most *m* key fields and *m+1* pointer fields
- Half-full must be satisfied (except root node):
- m is even and m=2d
  - Leaf node half full: at least d entries
  - Non-leaf node half full: at least d entries
- m is odd and m = 2d+1
  - Leaf node half full: at least d+1 entries
  - Non-leaf node half full: at least d entries (i.e., d+1 pointers)

Show the tree after insertions

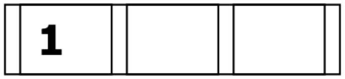
- Suppose each B+-tree node can hold up to 4 pointers and 3 keys.
- m=3 (odd), d=1
- Half-full (for odd m value)
  - Leaf node, at least 2 (d+1) entries
  - Non-leaf nodes, at least 2 (d+1) pointers (1 entry)
- Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

B+ tree Index Insertion

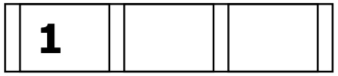
- Step 1: Descend to the leaf node where the key fits
- Step 2:
  - ❑ (Case 1): If the node has an empty space, insert the key into the node.
  - ❑ (Case 2) If the node is already full, split it into two nodes by the middle key value, distributing the keys evenly between the two nodes, so each node is half full.
    - ▪ (Case 2a) If the node is a leaf, take a copy of the middle key value, and repeat step 2 to insert it into the parent node.
    - ▪ (Case 2b) If the node is a non-leaf, exclude the middle key value during the split and repeat step 2 to insert this excluded value into the parent node.

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

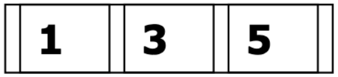
- Insert 1



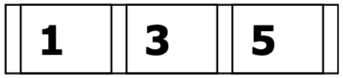
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



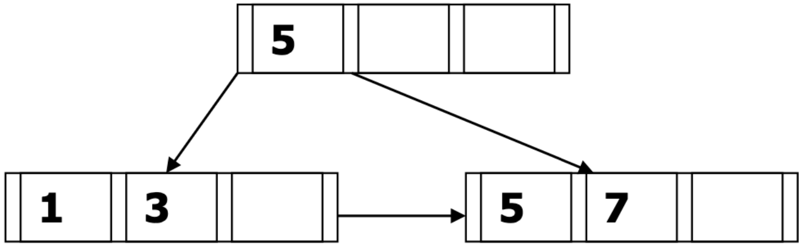
- Insert 3, 5



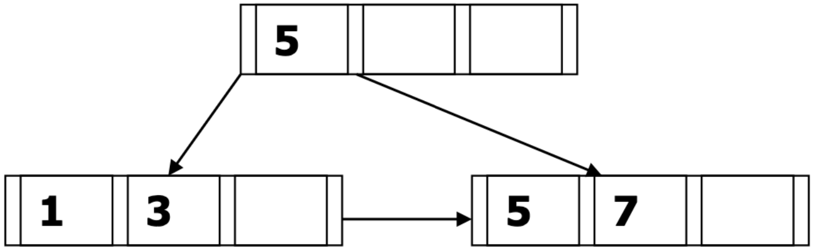
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



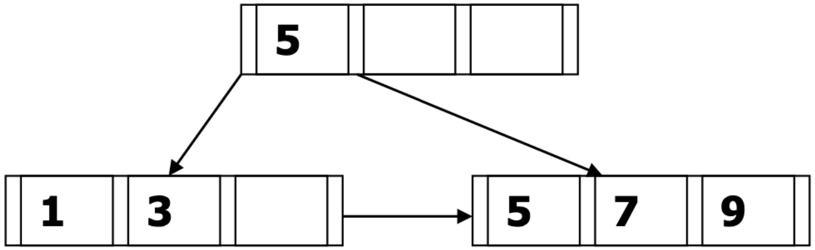
- Insert 7



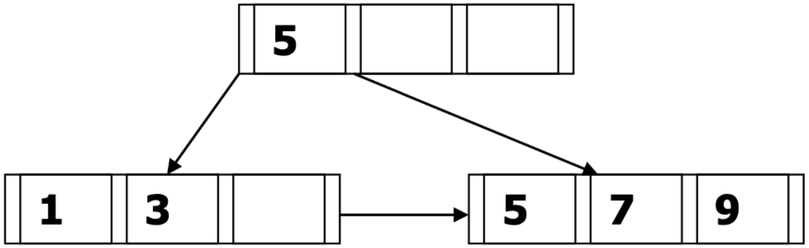
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



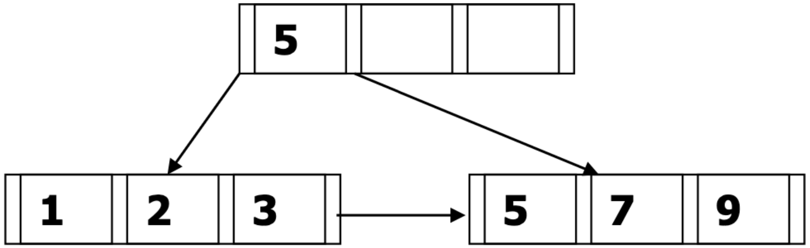
- Insert 9



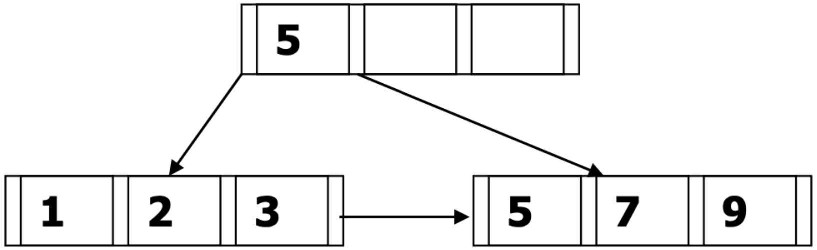
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



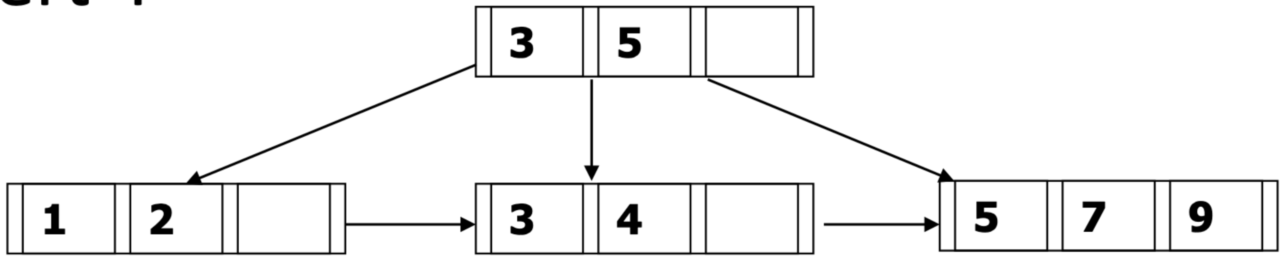
- Insert 2



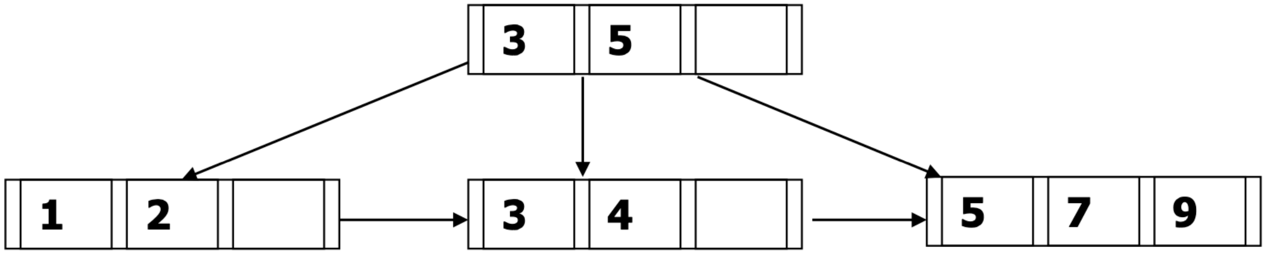
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



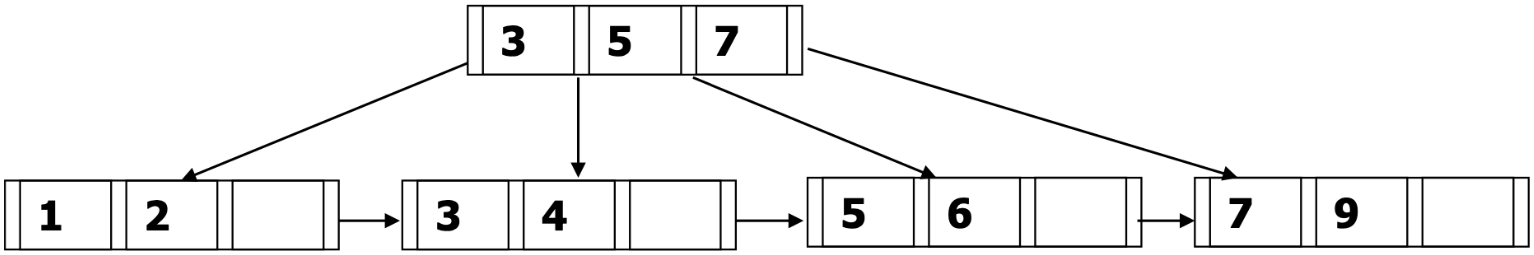
- Insert 4



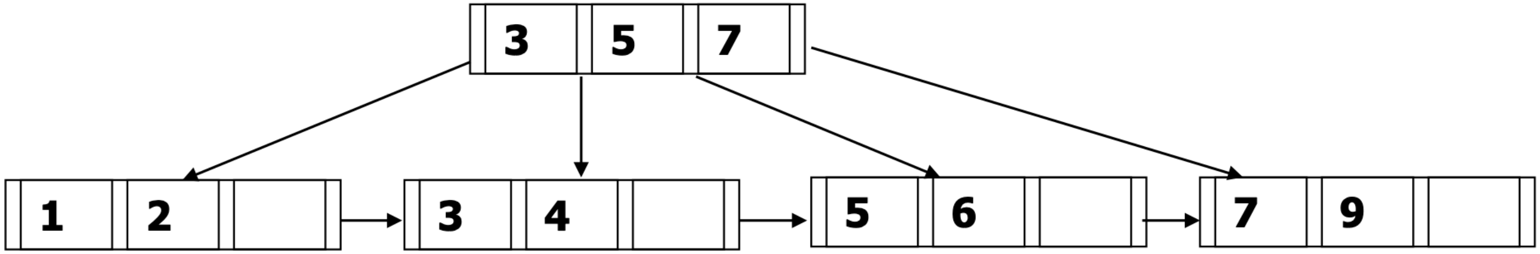
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



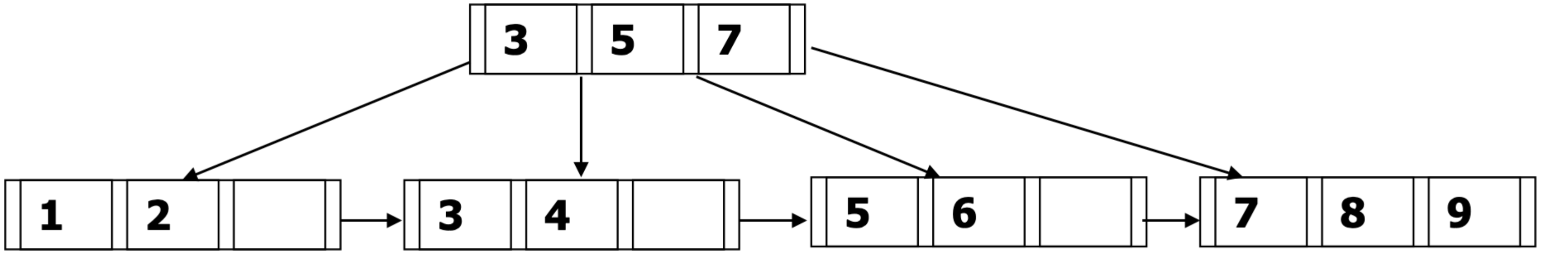
- Insert 6



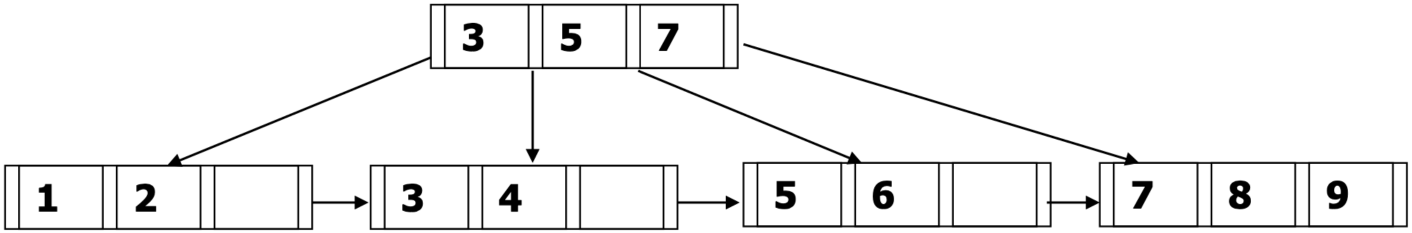
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



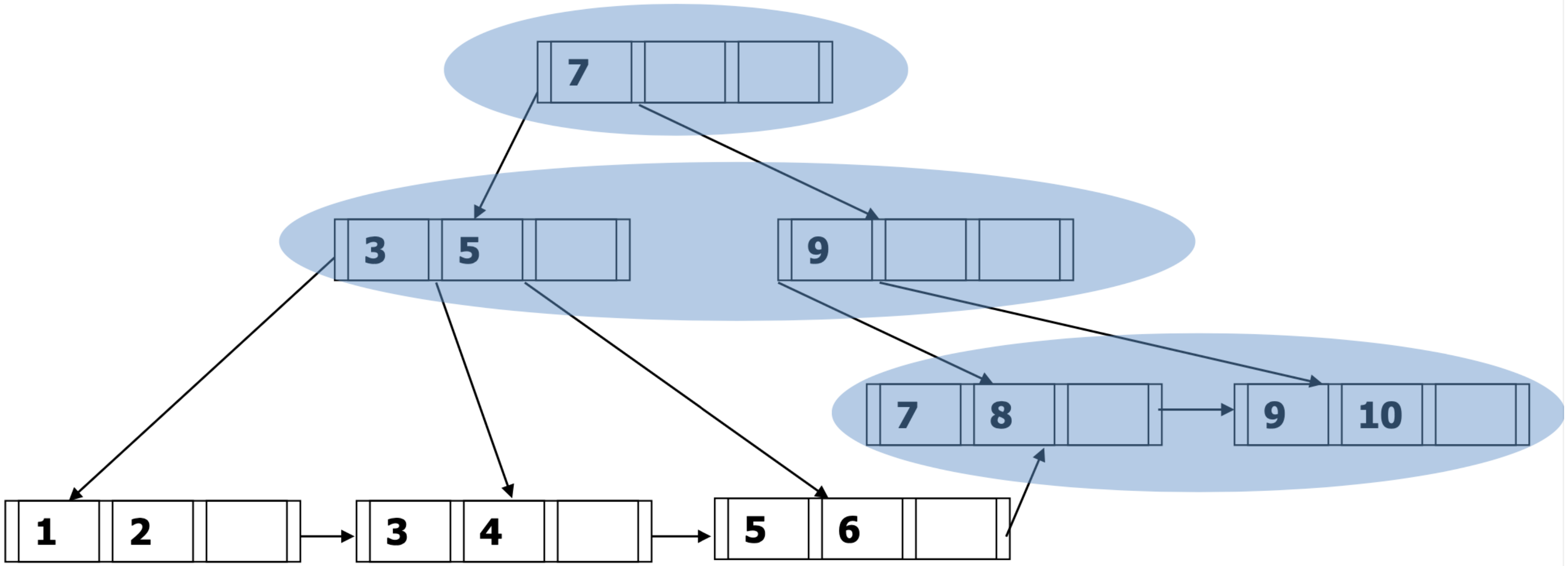
- Insert 8



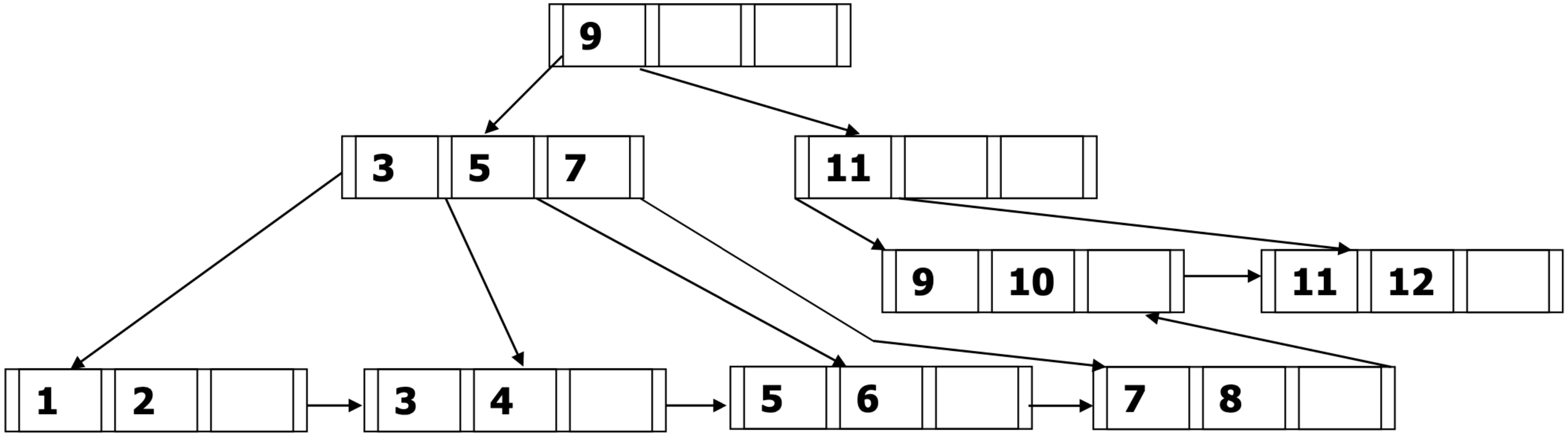
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



- Insert 10



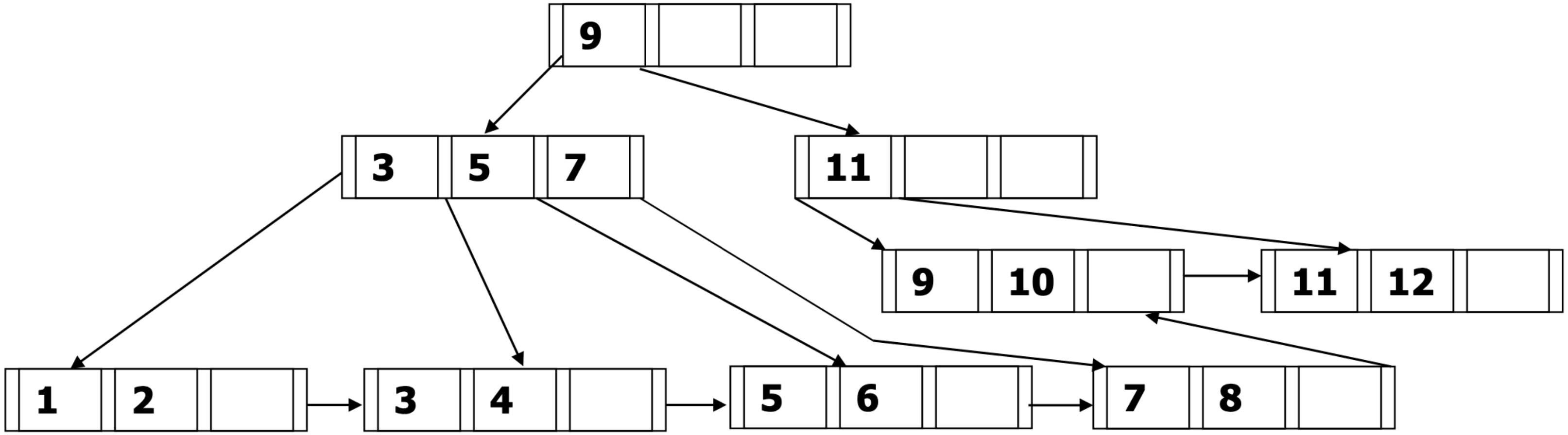
Show the tree after deletions



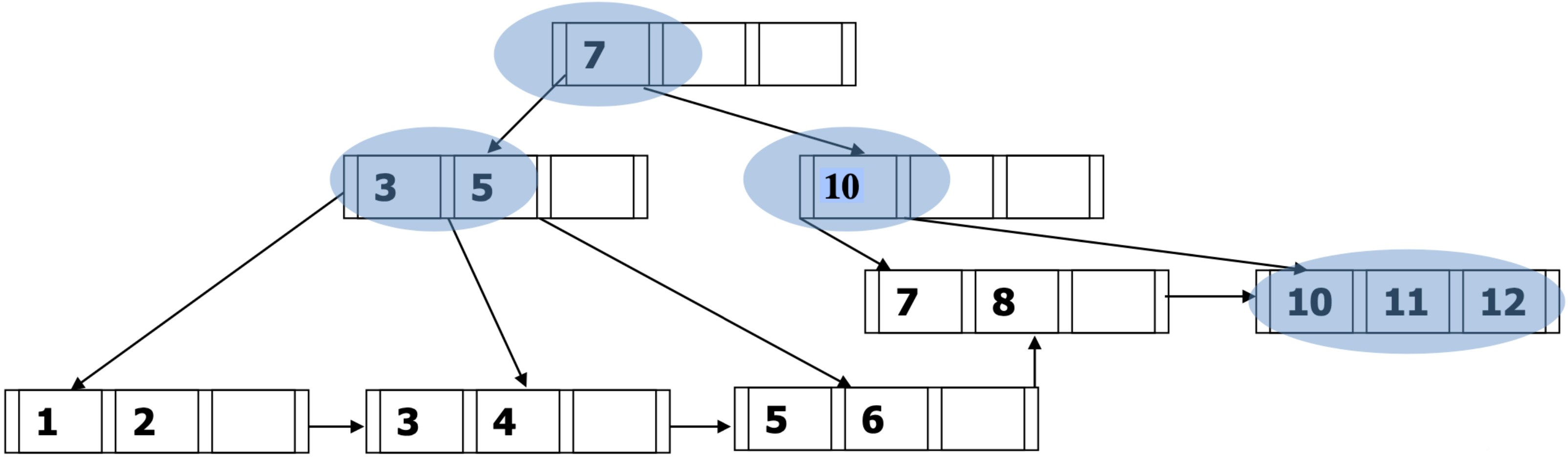
- Remove 9, 7, 8

B+ tree Index Deletion

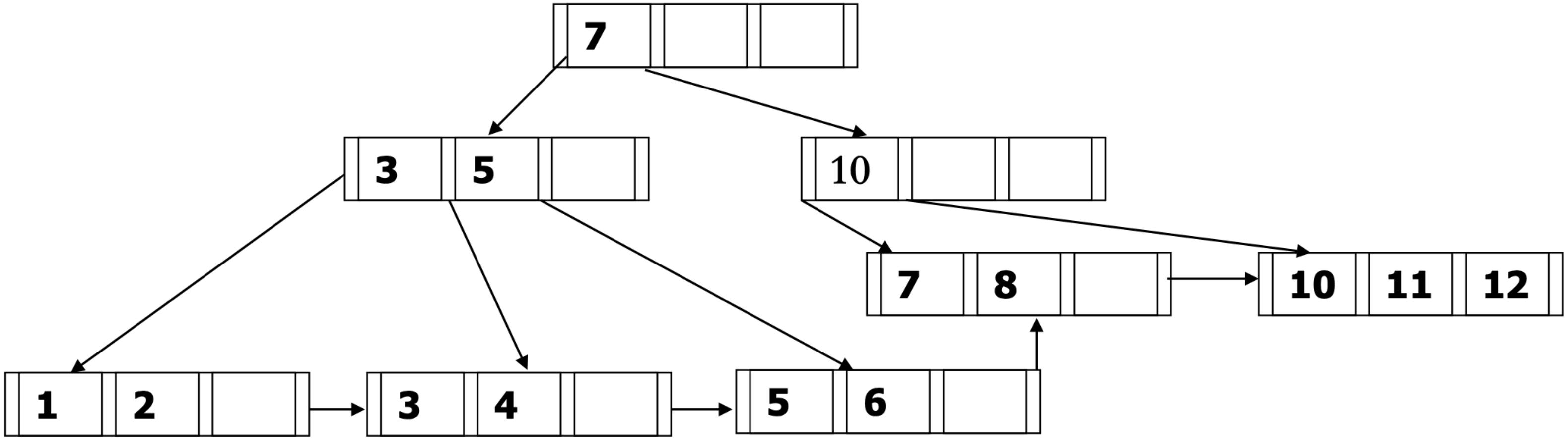
- Step1: Descend to the leaf node where the key fits
- Step 2: Remove the required key and associated reference from the node.
  - ❑ (Case 1): If the node still has half-full keys, repair the keys in parent node to reflect the change in child node if necessary and stop.
  - ❑ (Case 2): If the node is less than half-full, but left or right sibling node is more than half-full, redistribute the keys between this node and its sibling. Repair the keys in the level above to represent that these nodes now have a different “split point” between them.
  - ❑ (Case 3): If the node is less than half-full, and left and right sibling node are just half-full, merge the node with its sibling. Repeat step 2 to delete the unnecessary key in its parent.



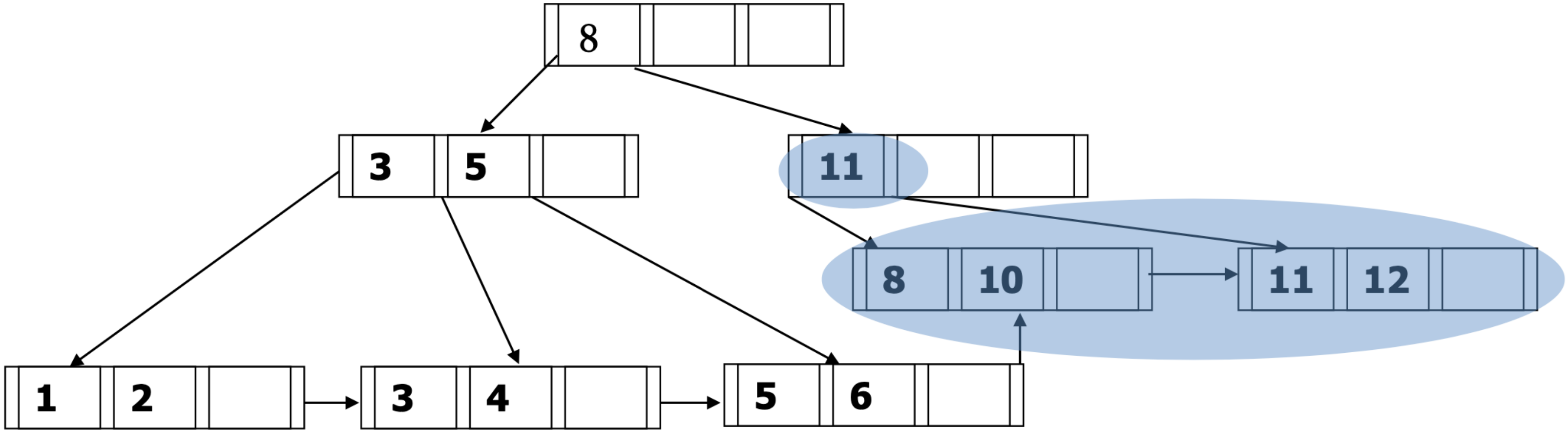
- After removing 9



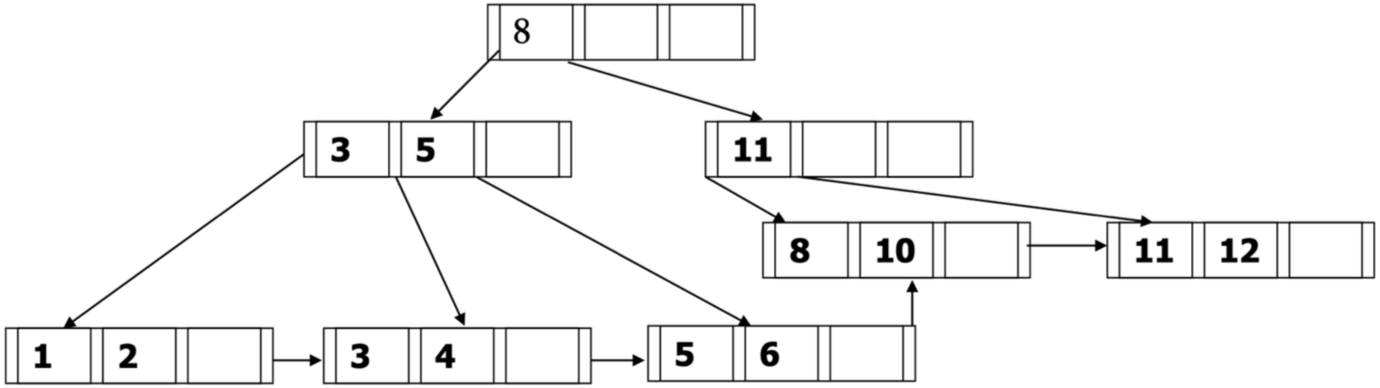
Remove 9, 7, 8



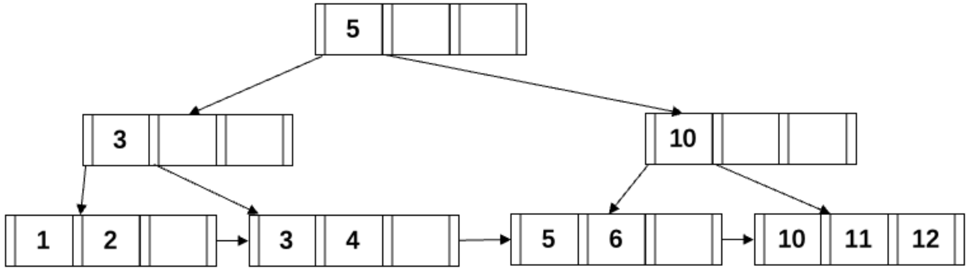
- After removing 7



Remove 9, 7, 8



- After removing 8



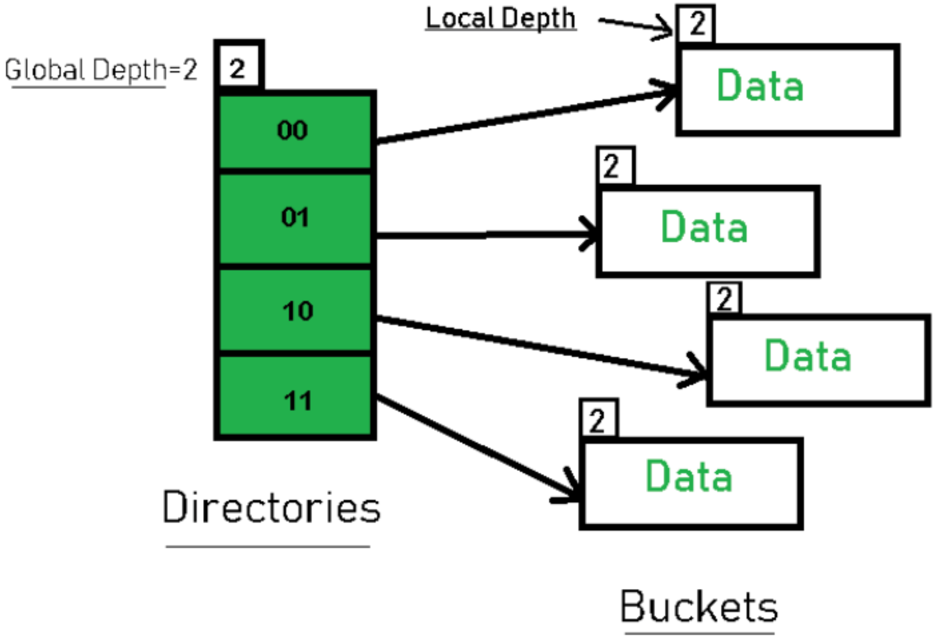
Example of Extendible Hashing

let us consider an example of hashing the following elements:

16,4,6,22,24,10,31,7,9,20,26.

- **Bucket Size:** 3
- **Hash Function:** Suppose the global depth is X. Then the Hash Function returns X LSBs.

Extendible Hashing



Extendible Hashing

- **Step 1**: Convert data into binary format
- **Step 2**: Initialize the directory and buckets
- **Step 3**: Identify the directory using hashing (LSBs)
- **Step 4**: Insertion and overflow check
- **Step 5**: Rehashing of split bucket elements
- **Step 6**: The element is successfully hashed

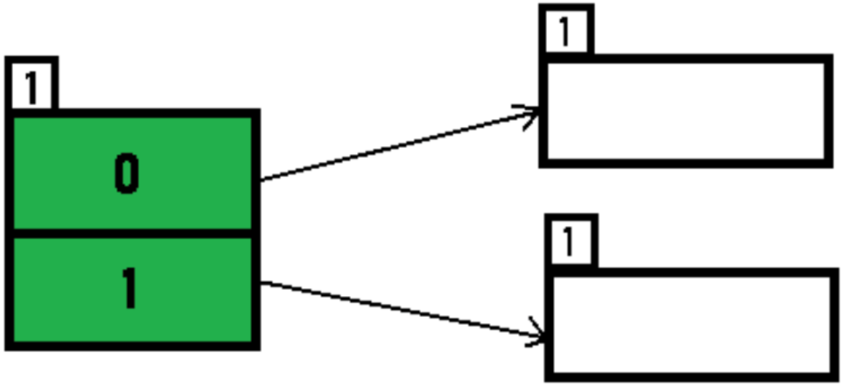
Convert data into binary format

First, calculate the binary forms of each of the given numbers

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

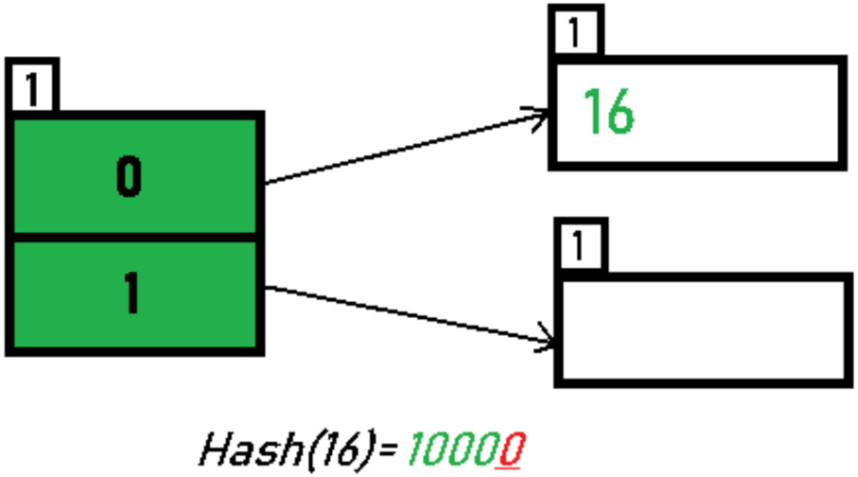
Initialize the directory and buckets

Initially, the global-depth and local-depth is always 1.
Thus, the hashing frame looks like this



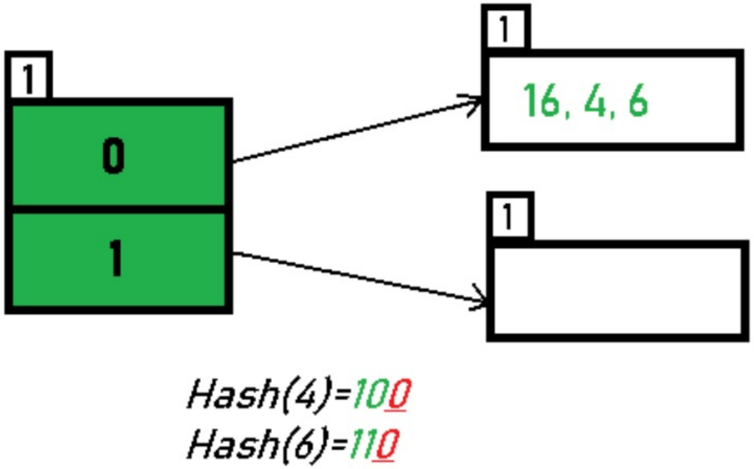
Inserting 16

The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0



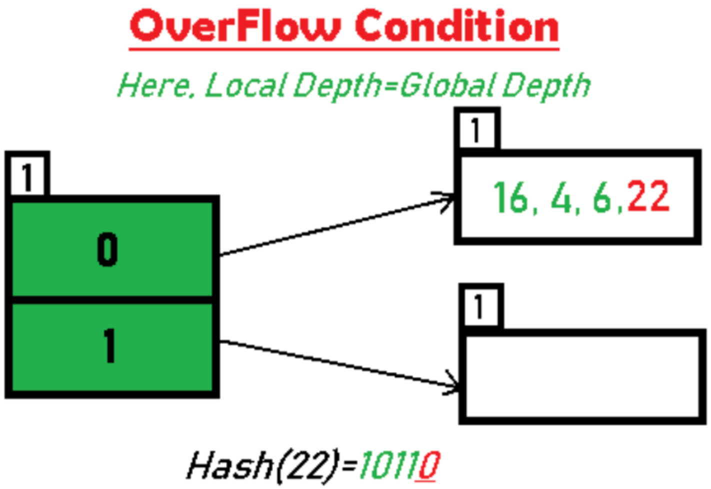
Inserting 4 and 6

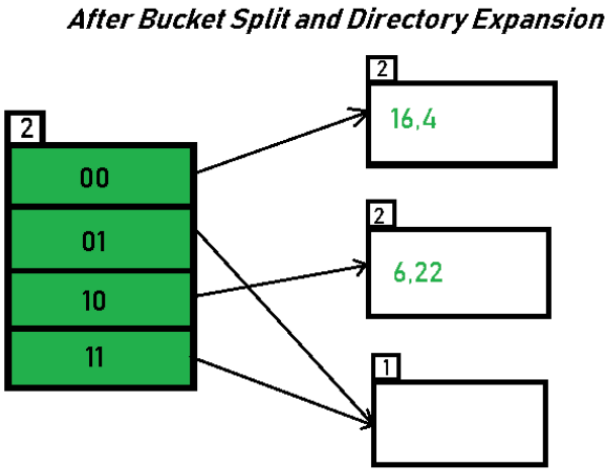
Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows



Inserting 22

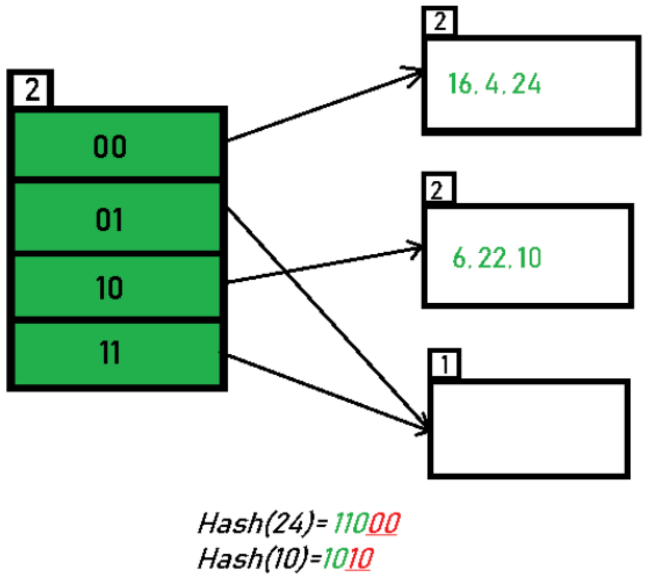
- The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.
- the bucket splits and directory expansion takes place, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]





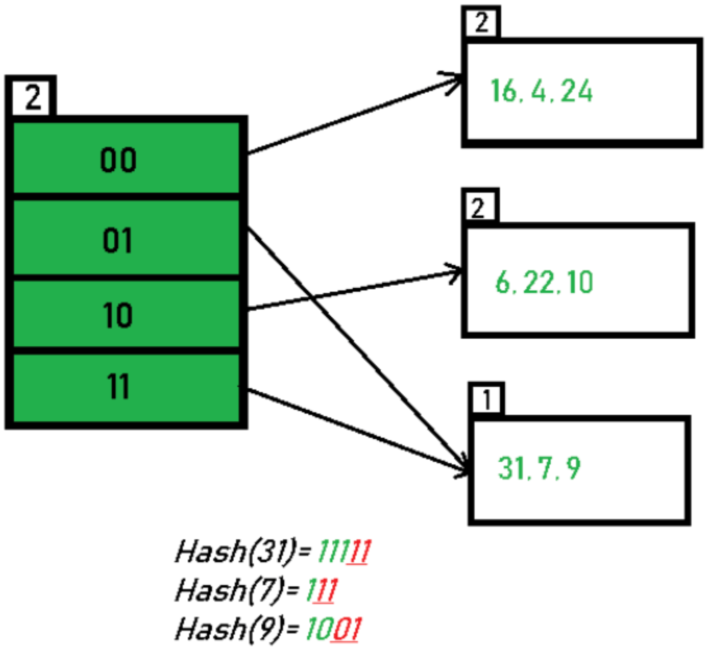
Inserting 24 and 10

24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition



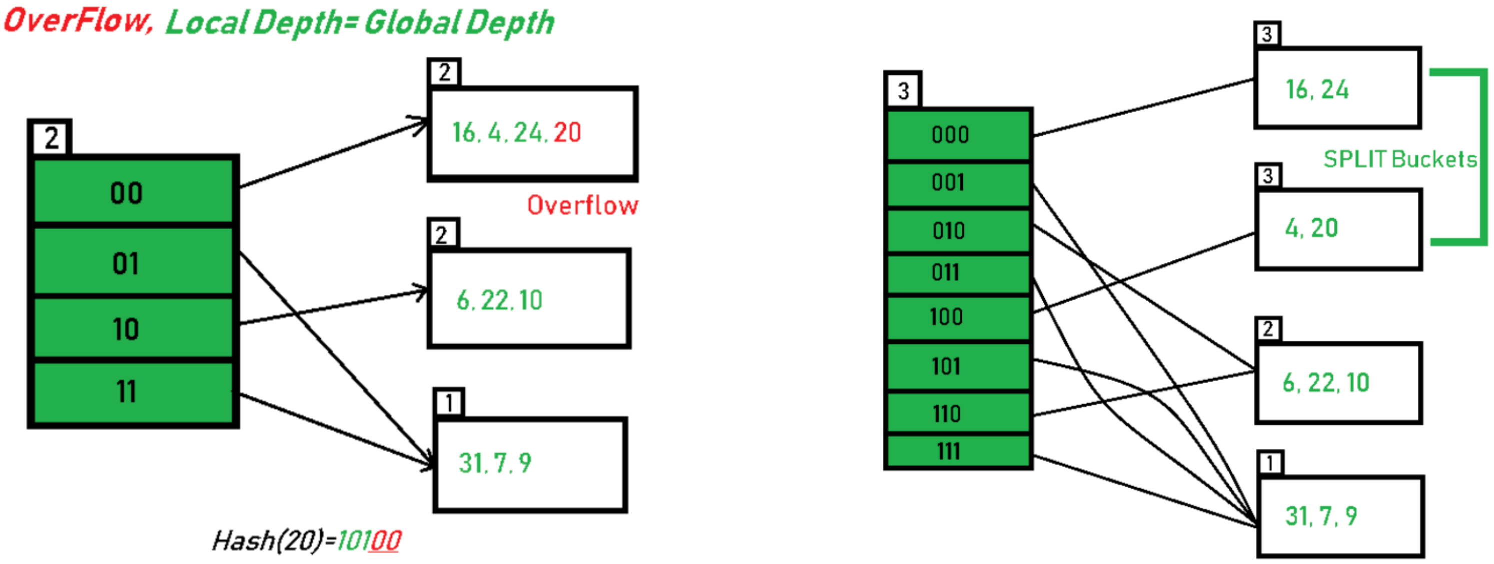
Inserting 31,7,9

All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



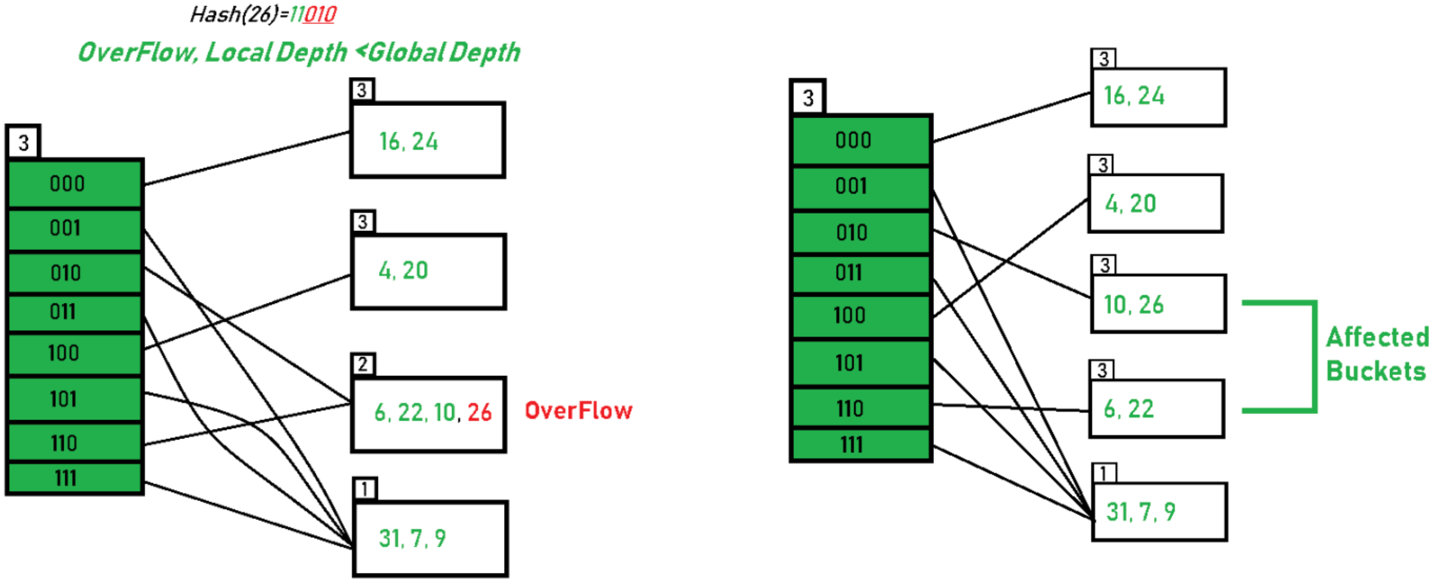
Inserting 20

Insertion of data element 20 (101**00**) will again cause the overflow problem



Inserting 26

Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010


Hash(26)=11010
OverFlow, Local Depth <Global Depth
3
000
001
010
011
100
101
110
111
3
16, 24
3
4, 20
2
6, 22, 10, 26
OverFlow
1
31, 7, 9
3
000
001
010
011
100
101
110
111
3
16, 24
3
4, 20
3
10, 26
3
6, 22
1
31, 7, 9
Affected Buckets