Exercise 4.1.1: Suppose blocks hold either three records, or ten key-pointer pairs. As a function of n, the number of records, how many blocks do we need to hold a data file and:

a) A dense index:

Solution: Blocks needed for datafile: n/3

Blocks needed for index: n/10

∴ Total blocks needed is (n/3 + n/10) = **13n/30**

b) A sparse index:

Solution: Blocks needed for datafile: n/3

Blocks needed for index: n/30

∴ Total blocks needed is (n/3 + n/30) = **11n/30**

Exercise 4.3.3: Suppose pointers are 4 bytes long, and keys are 12 bytes long. How many keys and pointers will a block of 16,384 bytes have?

Solution: (n+1) pointers * size of pointer + n keys * size of keys < size of block

$(n+1) *4 + n*12 < 16384$

$4n + 4 +12n < 16384$

$16n < 16380$

$n < 1023.75$

n = 1023

Exercise 4.3.4: What are the minimum numbers of keys and pointers in B-tree (i) interior nodes and (M) leaves, when:

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

Solution: First, Considering **B-Tree**:

For Interior nodes:

minimum number of keys= **5**

minimum number of pointers= **6**

For leaves:

minimum number of keys= **5**

minimum number of pointers= **6**

Now, Considering **B+ Tree**:

For Interior nodes:

minimum number of keys =((n+1)/2) - 1 = ((10+1)/2)-1 = 4.5 = **5**

minimum number of pointers = (n+1)/2 = (10+1)/2 = 5.5 = **6**

For leaves:

minimum number of keys= (n+1)/2 = (10+1)/2 = 5.5 = **6**

minimum number of pointers= (n+1)/2 = (10+1)/2 = 5.5 = **6**

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

Solution: First Considering **B-Tree**:

For Interior nodes:

minimum number of keys= **5**

minimum number of pointers= **6**

For leaves:

minimum number of keys= **6**

minimum number of pointers= **7**

Now, Considering **B+ Tree**:

For Interior nodes:

minimum number of keys =((n+1)/2) - 1 = ((11+1)/2)-1 = **5**

minimum number of pointers = (n+1)/2 = (11+1)/2 = **6**

For leaves:

minimum number of keys= (n+1)/2 = (11+1)/2 = **6**

minimum number of pointers= (n+1)/2 = (11+1)/2 = **6**

Exercise 4.3.5: Execute the following operations on Fig. 4.23. Describe the changes for operations that modify the tree.

a) Lookup the record with key 41.
Solution: First We will start with the root 13. Since 41 > 13, we move to the second pointer to second level node with key 23, 31 and 43. Since 31<41<43, So we go to the third pointer. Then we reach the leaf with key 31, 37 and 41. We find record with key 41.

b) Lookup the record with key 40.
Solution: We will start with the root 13. Since 40 > 13, we move to the second pointer to second level node with key 23, 31 and 43. Since 31<40<43, So we go to the third pointer. Then we reach the leaf with key 31, 37 and 41. There is no key 40. Thus, there is no record with key 40.

c) Lookup all records in the range 20 to 30.

Solution: We start with the root 13. Since both 20 and 30 >13, we move to the second pointer to the second level node with key 23, 31, and 43. Since 20 < 23, we follow the first pointer to check if there is any key is greater or equal to 20 but less than 23. We reach the leaf with key 13, 17 and 19 which are all less than 20. We also need to follow the second key to check if there is number less than 30. The keys **23** and **29** followed by the second pointer are records lies between 20 to 30.

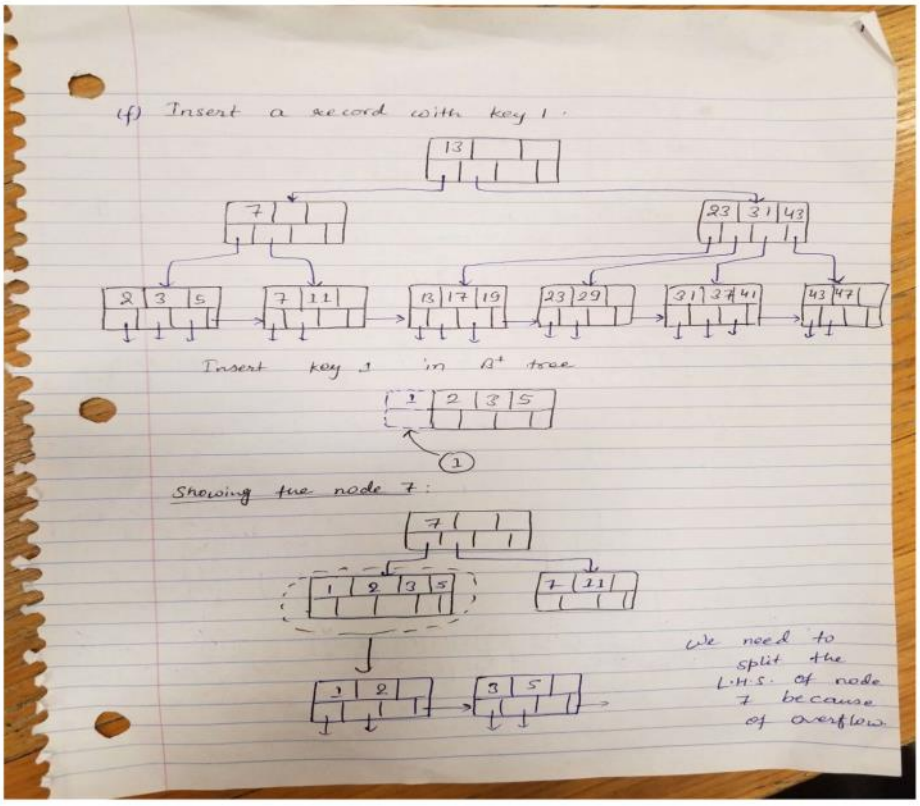
d) Lookup all records with keys less than 30.

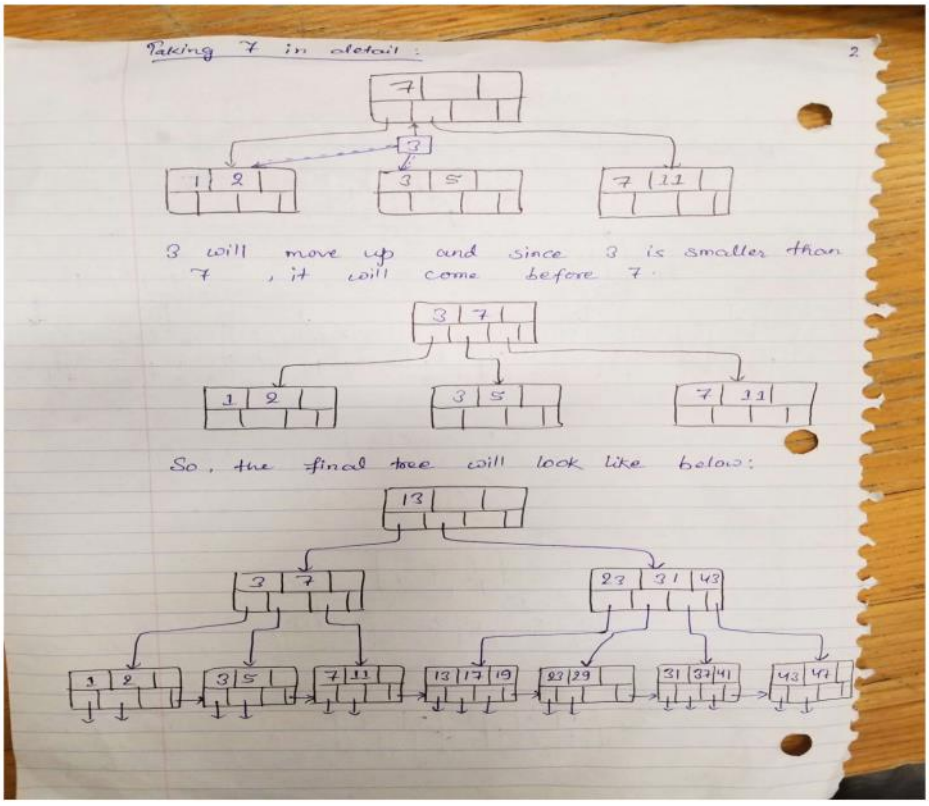
Solution: We start with the root 13. Since 30>13, we move to all the leaves in the left side of the root and find the keys 2, 3, 5, 7, and 11.
Then we move to the second pointer to second level node of root and find the keys 23, 31, and 43. Since 23<30, we move to the first pointer and find 13, 17, and 19.
Also need to go to the second key to check if it is less than 30 and find the keys 23 and 29.
Therefore, we find records with key **2, 3, 5, 7, 11, 13, 17, 19, 23** and **29**.

e) Lookup all records with keys greater than 30.

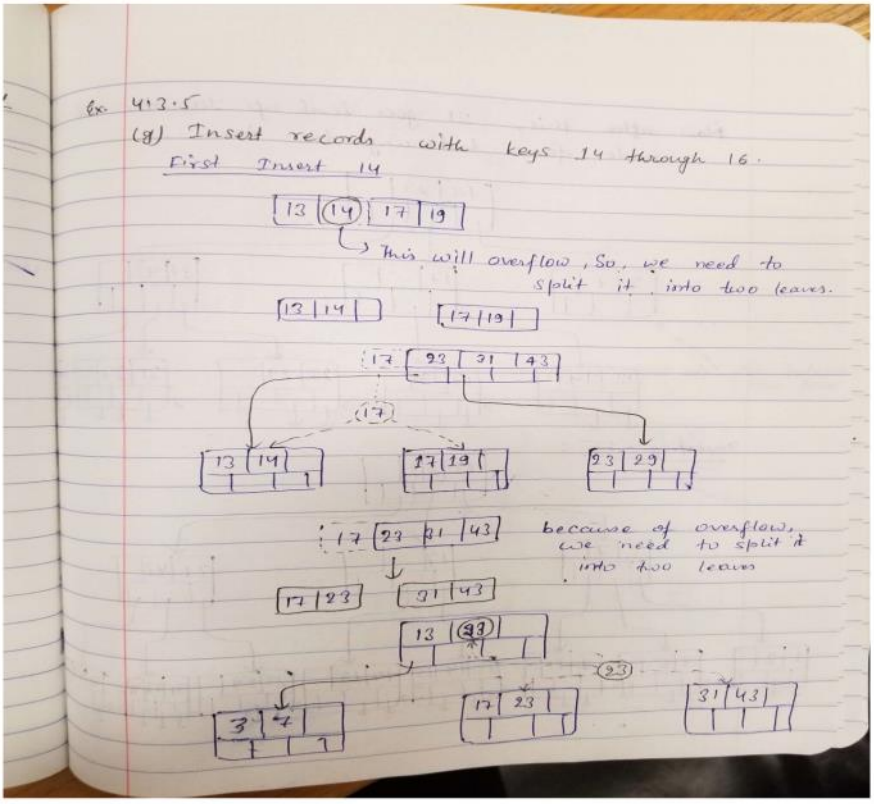
Solution: We start with root 13. Since 30>13, we need to go to the second pointer to second level node with key 23, 31 and 43. Since 23<30<31, So we need to go to the second pointer. We reach the leaf with key 23 and 29. We did not find the key greater than 30. Now 30 ≤ 31, the records in the leaves followed by the third and fourth pointers are what we need. Therefore, we find the records with keys **31, 37, 41, 43** and **47**.

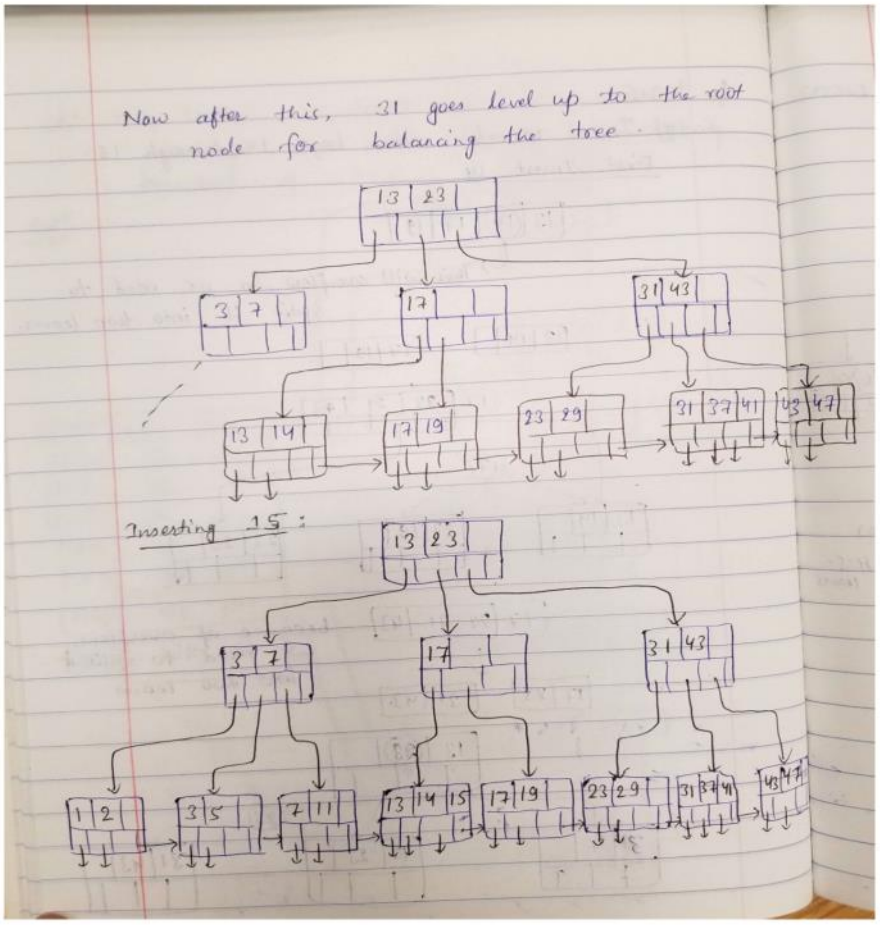
f) Insert a record with key 1

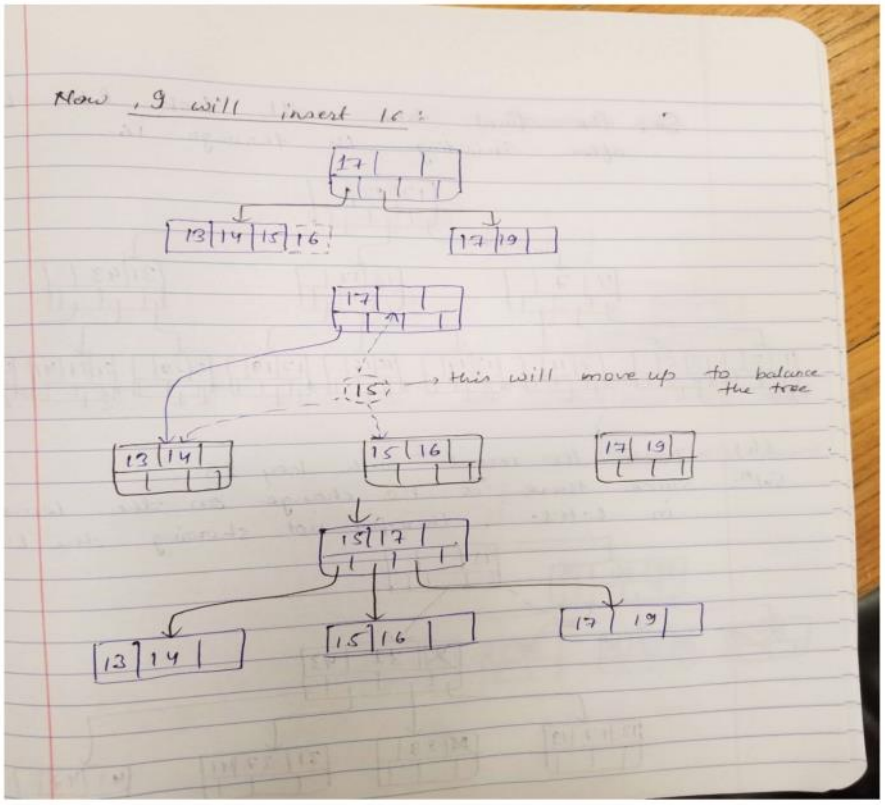


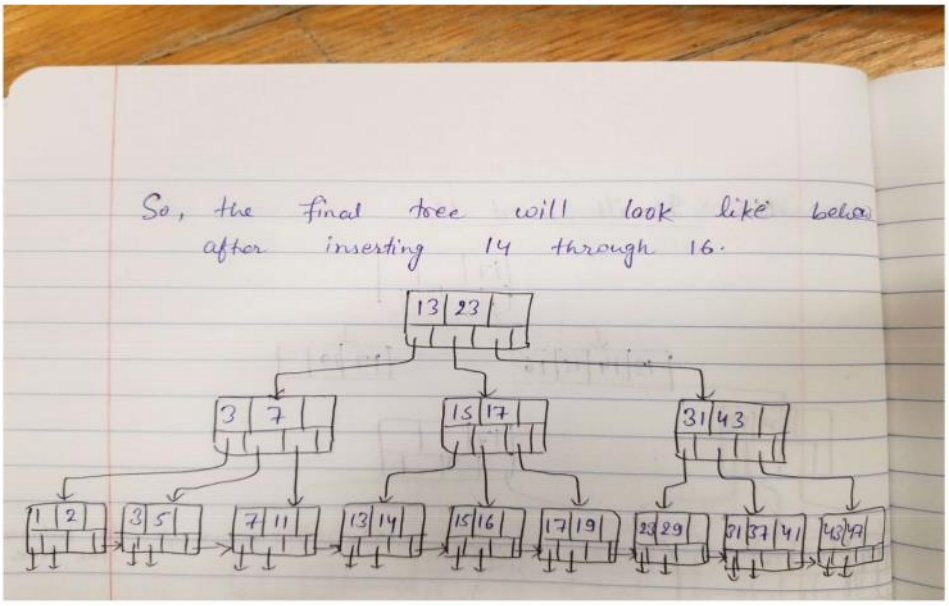


g) Insert records with keys 14 through 16

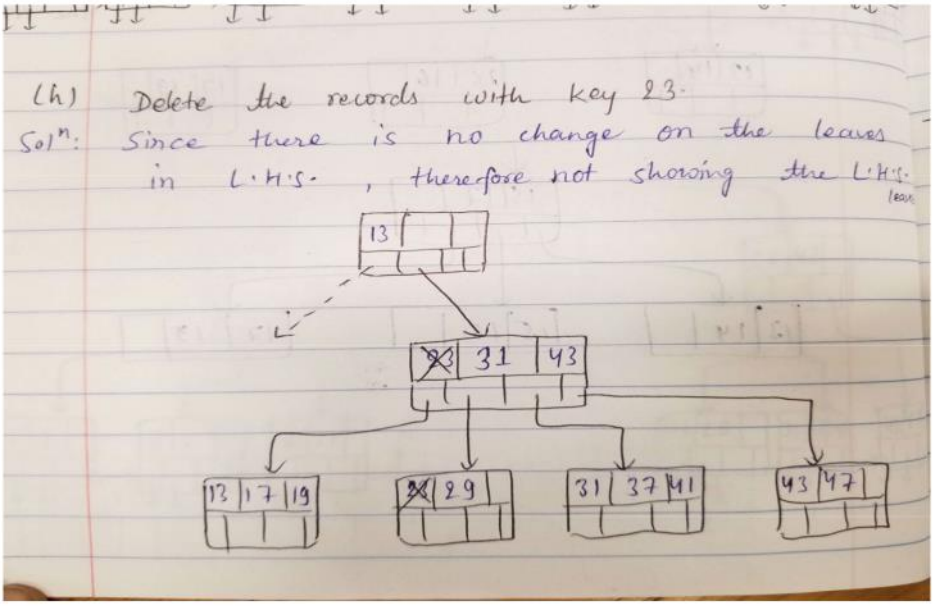


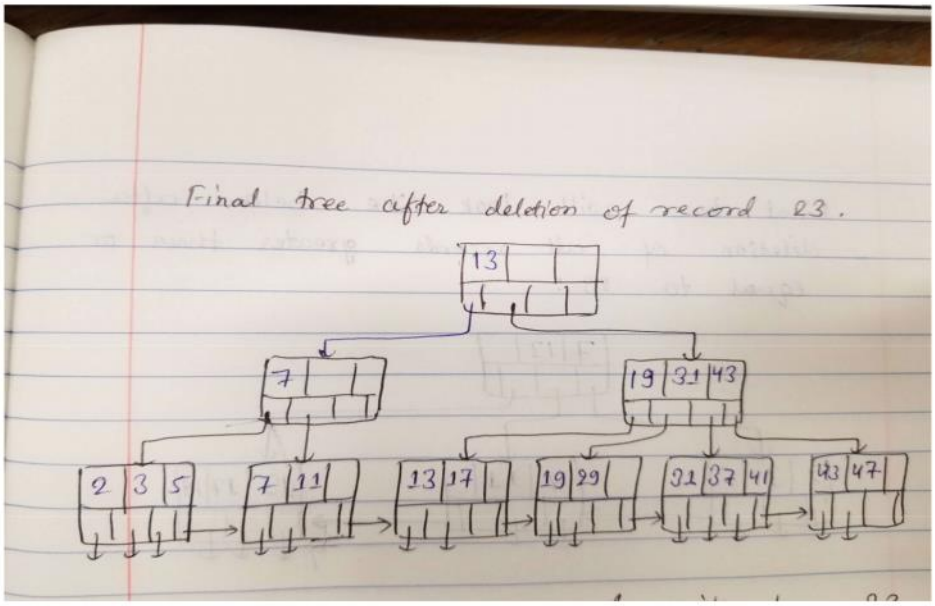

Now after this, 31 goes level up to the root node for balancing the tree.
Inserting 15 :


Now, 9 will insert it :
this will move up to balance the tree

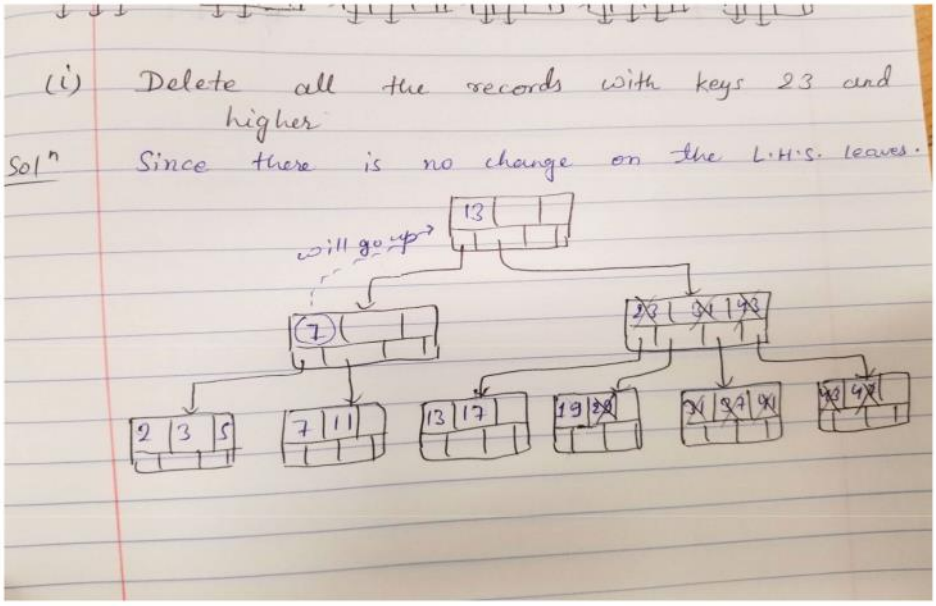


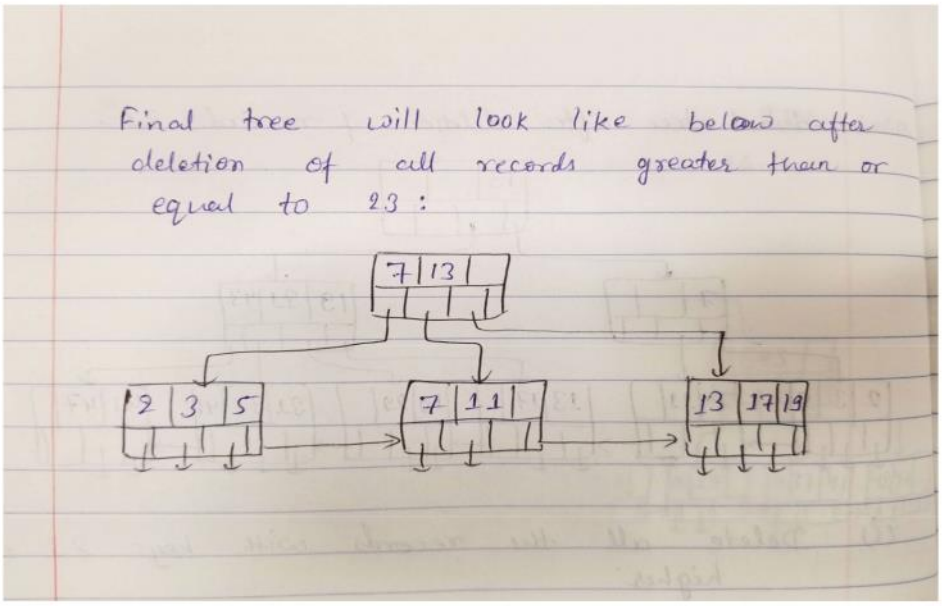
h) Delete the record with key 23





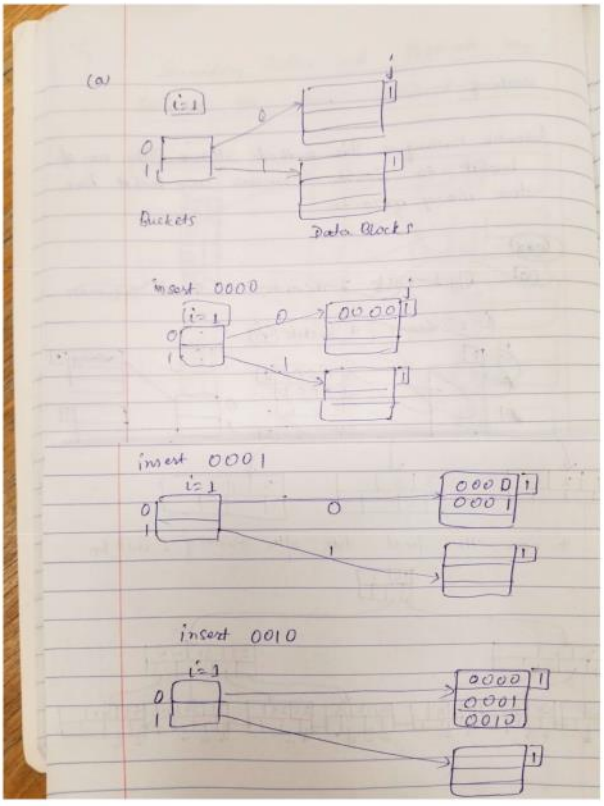
i) Delete all the records with keys 23 and higher

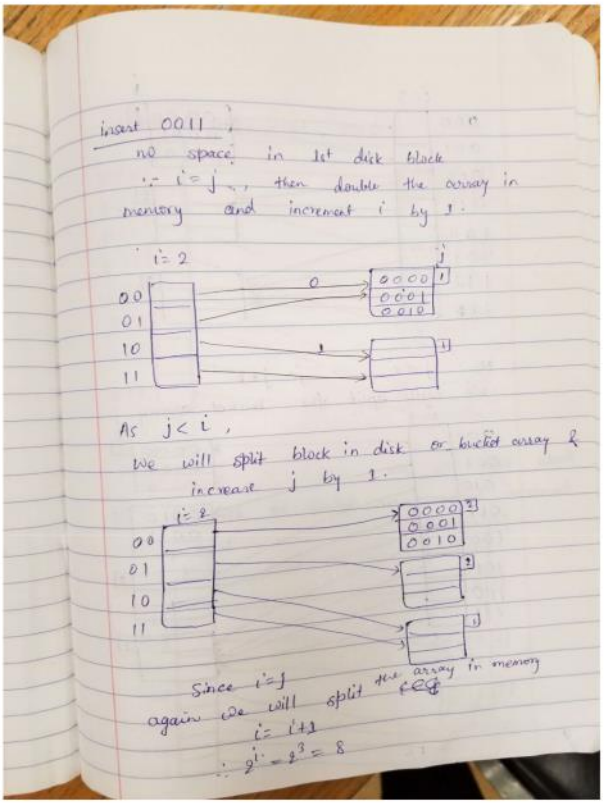


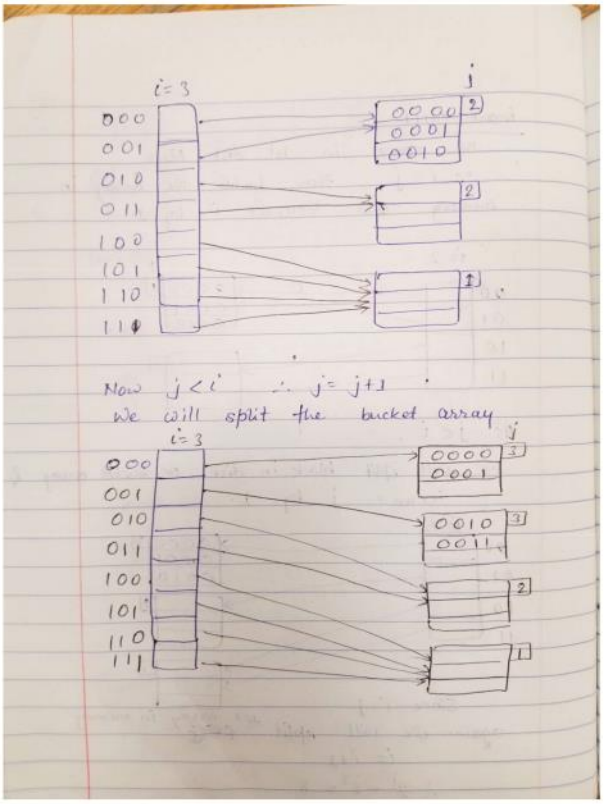


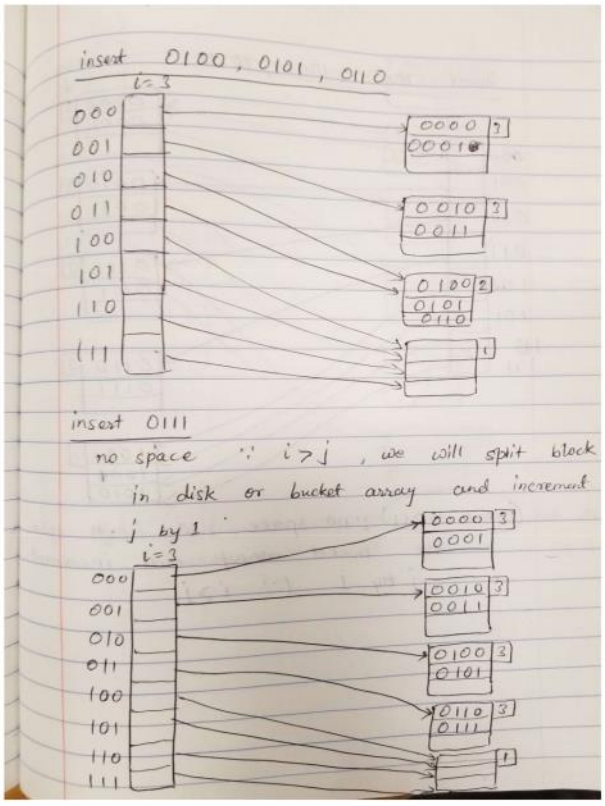
Exercise 4.4.6: Suppose keys arc hashed to four-bit sequences, as in our examples of extensible and linear hashing in this section. However, also suppose that blocks can hold three records, rather than the two-record blocks of our examples. If we start with a hash table with two empty blocks (corresponding to 0 and 1), show the organization after we insert records with keys:

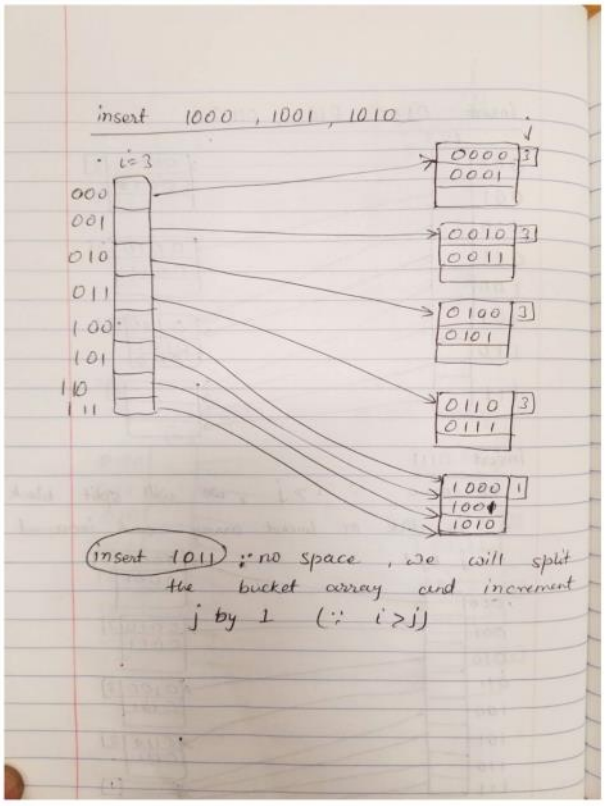
a) 0000,0001,..., 1111, and the method of hashing is extensible hashing.

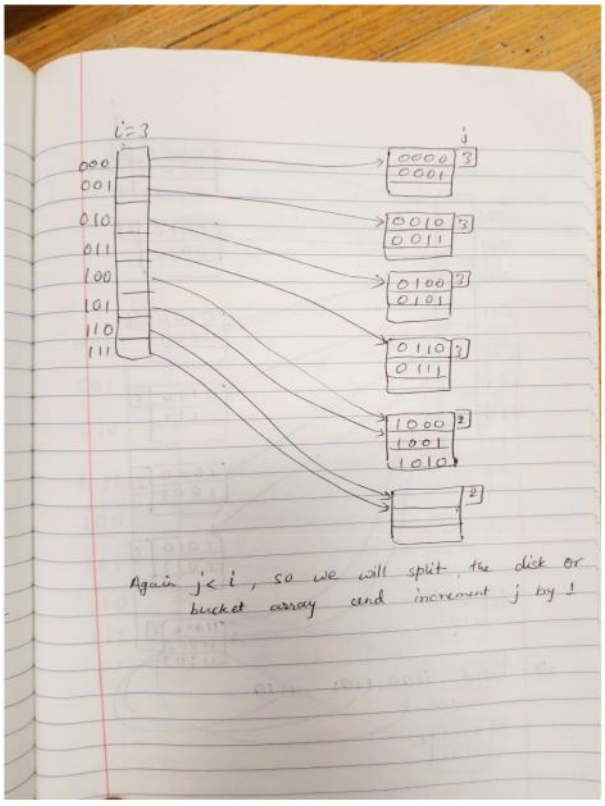


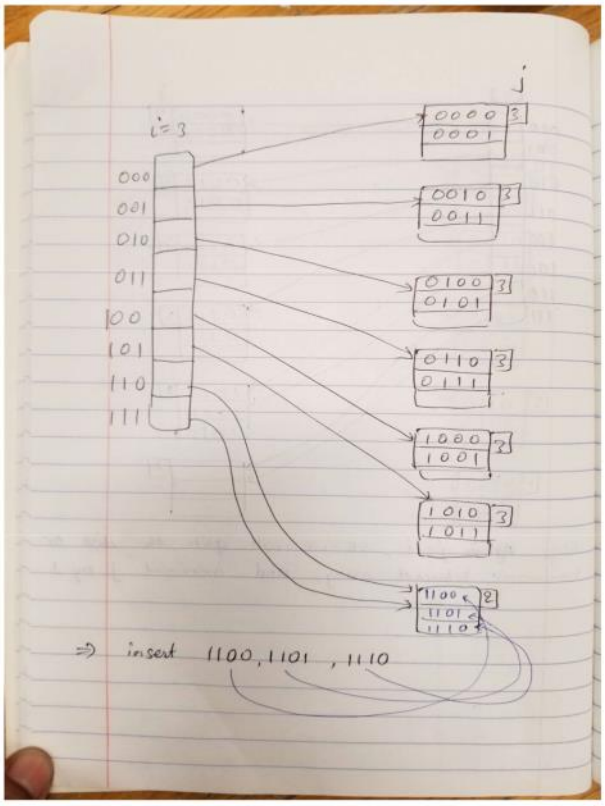

insert 0011
no space in 1st disk block
∴ i = j , then double the array in memory and increment i by 1.
i = 2
j
00
01
10
11
0000
0001
0010
As j < i ,
We will split block in disk or bucket array & increase j by 1.
i = 2
Since i = j
again we will split the array in memory
i = i+1
∴ 2^i = 2^3 = 8

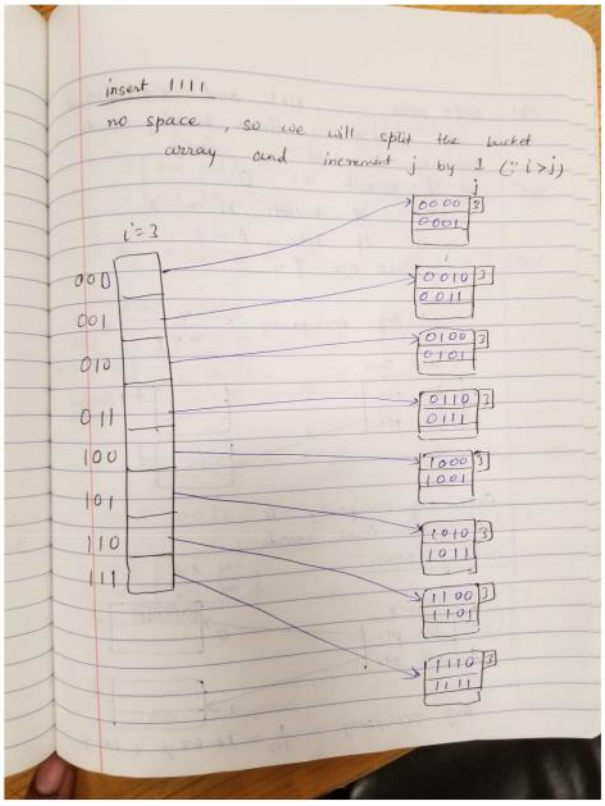

i = 3
j
000
001
010
011
100
101
110
111
0000
0001
0010
Now j < i ∴ j = j+1
We will split the bucket array
i = 3
j
0000
0001
0010
0011


insert 0100, 0101, 0110
i=3
000
001
010
011
100
101
110
111
0000
0001
0010
0011
0100
0101
0110
insert 0111
no space ∵ i>j, we will split block in disk or bucket array and increment j by 1
i=3
000
001
010
011
100
101
110
111
0000
0001
0010
0011
0100
0101
0110
0111

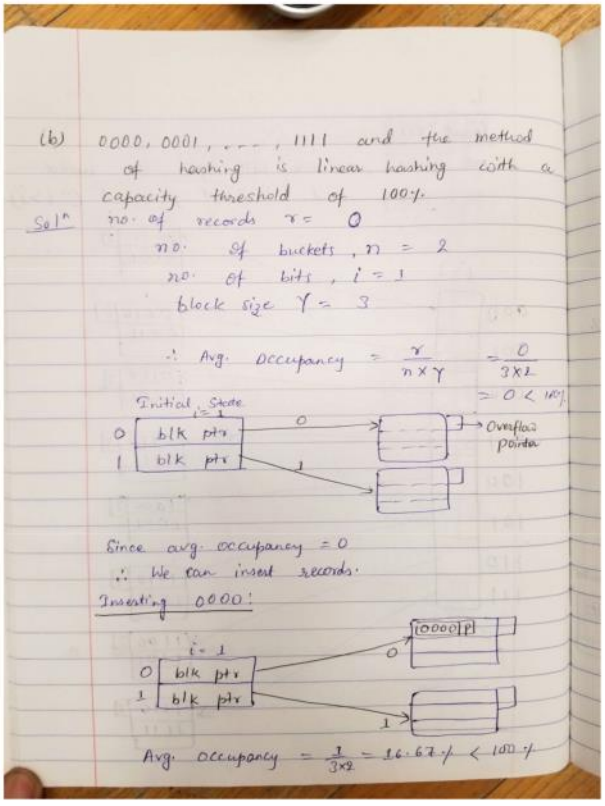

insert 1000, 1001, 1010
i=3
000
001
010
011
100
101
110
111
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
insert 1011 ∵ no space, we will split the bucket array and increment j by 1 (∵ i>j)

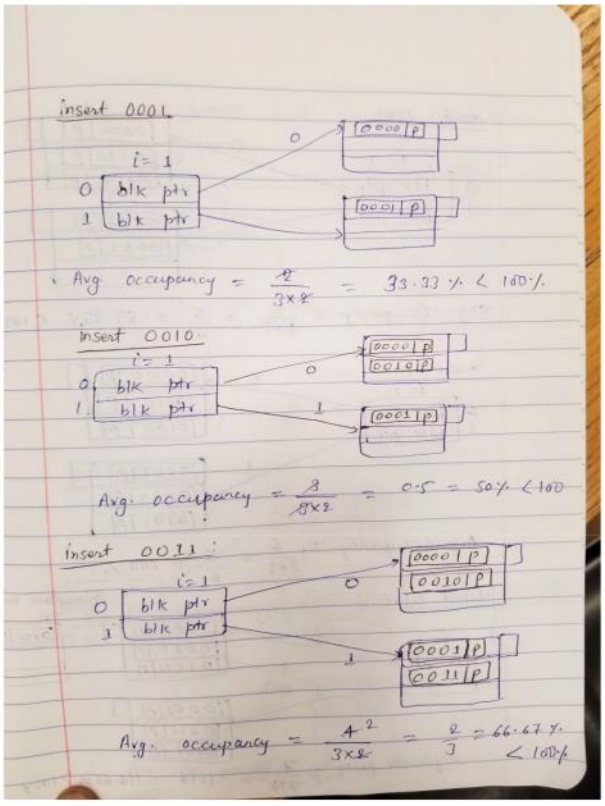

i=3
j
000
001
010
011
100
101
110
111
0000
0001
3
0010
0011
3
0100
0101
3
0110
0111
3
1000
1001
1010
2
2
Again j<i, so we will split the disk or bucket array and increment j by 1

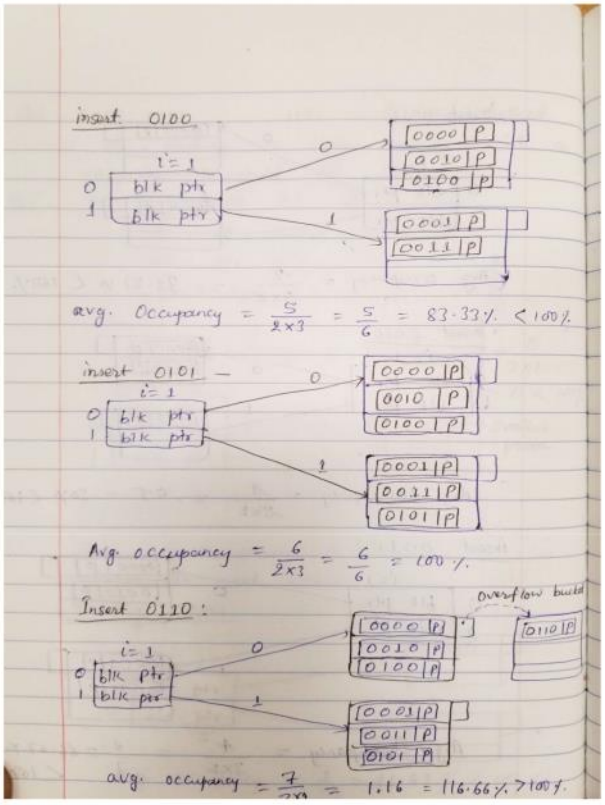

j
i=3
000
001
010
011
100
101
110
111
0000
0001
3
0010
0011
3
0100
0101
3
0110
0111
3
1000
1001
2
1010
1011
3
1100
1101
1110
2
=> insert 1100, 1101, 1110

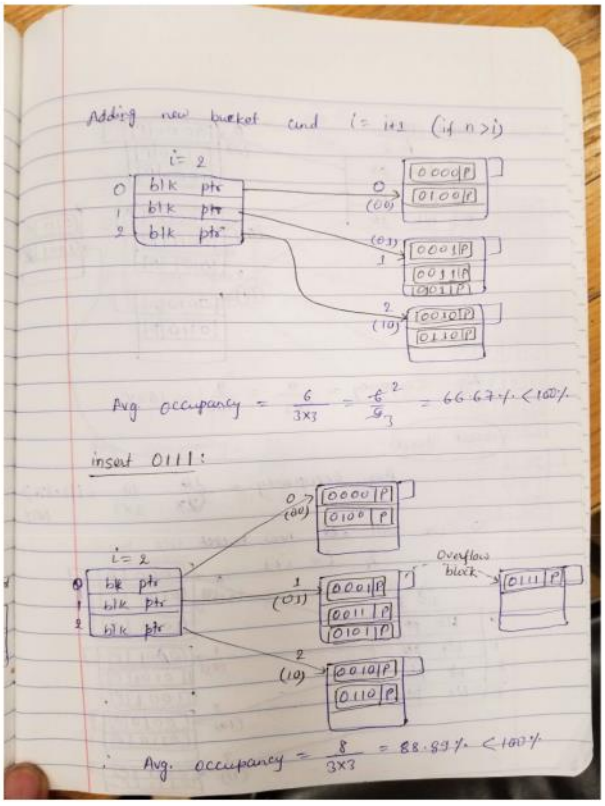


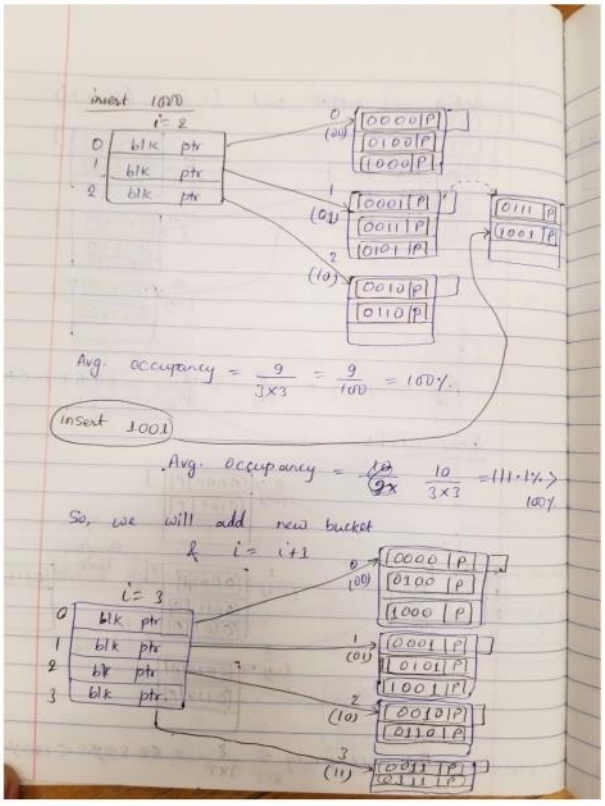
b) 0000, 0001,..., 1111, and the method of hashing is linear hashing with a capacity threshold of 100%.

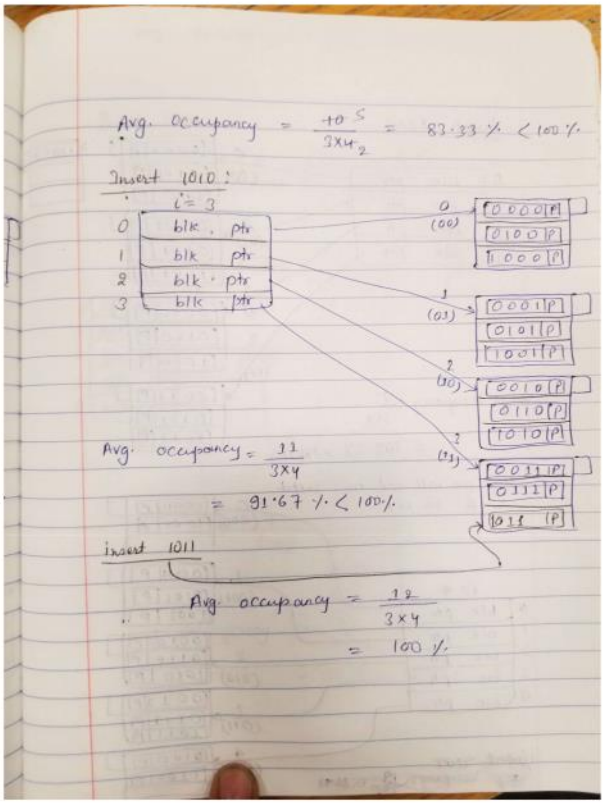


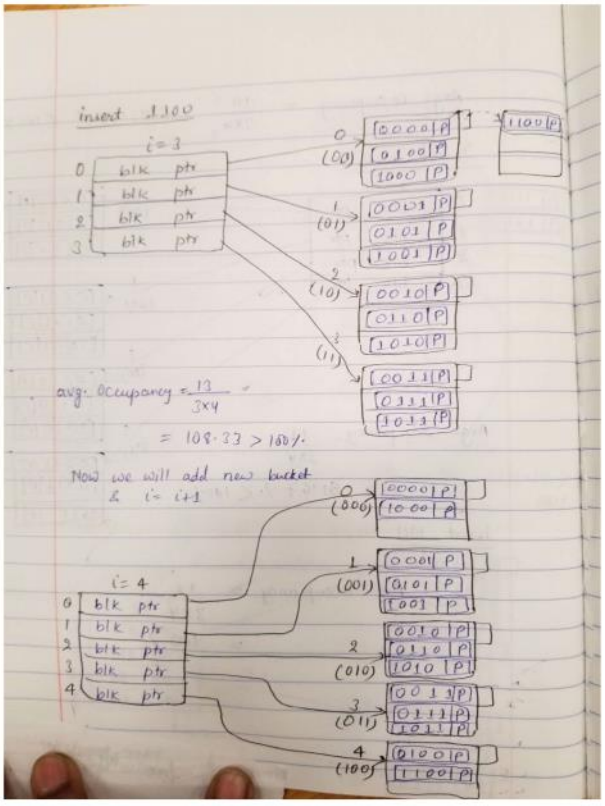

insert 0001
i = 1
0 blk ptr
1 blk ptr
0000 | p
0001 | p
Avg occupancy = 2/3×2 = 33.33 % < 100 %
insert 0010
i = 1
0 blk ptr
1 blk ptr
0000 | p
0010 | p
0001 | p
Avg. occupancy = 3/8×2 = 0.5 = 50 % < 100
insert 0011
i = 1
0 blk ptr
1 blk ptr
0000 | p
0010 | p
0001 | p
0011 | p
Avg. occupancy = 4^2/3×2 = 8/3 = 66.67 % < 100 %


insert 0100
i = 1
0 blk ptr
1 blk ptr
0000 | p
0010 | p
0100 | p
0001 | p
0011 | p
avg. Occupancy = 5/2×3 = 5/6 = 83.33 % < 100 %
insert 0101 –
i = 1
0 blk ptr
1 blk ptr
0000 | p
0010 | p
0100 | p
0001 | p
0011 | p
0101 | p
Avg. occupancy = 6/2×3 = 6/6 = 100 %
Insert 0110 :
overflow build
i = 1
0 blk ptr
1 blk ptr
0000 | p
0010 | p
0100 | p
0110 | p
0001 | p
0011 | p
0101 | p
avg. occupancy = 7/2×3 = 1.16 = 116.66 % > 100 %


Adding new bucket and i = i+1 (if n > i)
i = 2
blk ptr
blk ptr
blk ptr
0 (00)
1 (01)
2 (10)
0000 P
0100 P
0001 P
0011 P
0101 P
0010 P
0110 P
Avg. occupancy = 6/3x3 = 6/9 = 66.67% < 100%
insert 0111:
Overflow block
0111 P
Avg. occupancy = 8/3x3 = 88.89% < 100%

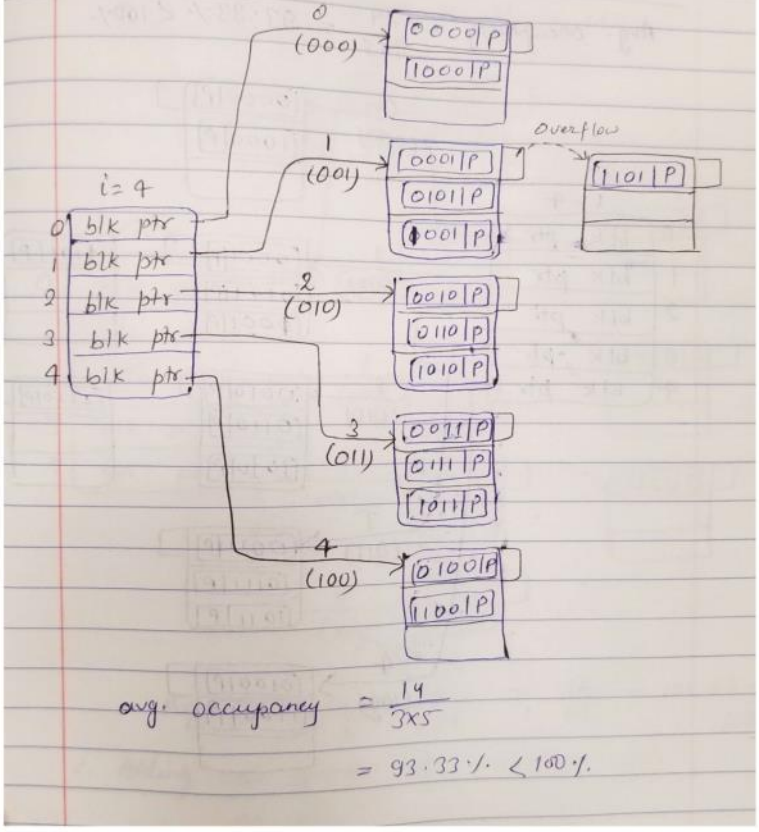

insert 1000
i = 2
blk ptr
1000 P
Avg. occupancy = 9/3x3 = 9/9 = 100%
insert 1001
1001 P
Avg. occupancy = 10/3x3 = 111.1% > 100%
So, we will add new bucket
& i = i+1
i = 3
blk ptr
3 (11)
0011 P


Avg. occupancy = 10/3x4 = 83.33 % < 100 %
Insert 1010 :
i = 3
0 blk ptr
1 blk ptr
2 blk ptr
3 blk ptr
0 (00) 0000 P 0100 P 1000 P
1 (01) 0001 P 0101 P 1001 P
2 (10) 0010 P 0110 P 1010 P
3 (11) 0011 P 0111 P 1011 P
Avg. occupancy = 11/3x4
= 91.67 % < 100 %
insert 1011
Avg. occupancy = 12/3x4
= 100 %


insert 1100
i = 3
0 blk ptr
1 blk ptr
2 blk ptr
3 blk ptr
0 (00) 0000 P 0100 P 1000 P 1100 P
1 (01) 0001 P 0101 P 1001 P
2 (10) 0010 P 0110 P 1010 P
3 (11) 0011 P 0111 P 1011 P
avg. occupancy = 13/3x4
= 108.33 > 100 %
Now we will add new bucket
& i = i+1
i = 4
0 blk ptr
1 blk ptr
2 blk ptr
3 blk ptr
4 blk ptr
0 (000) 0000 P 1000 P
1 (001) 0001 P 0101 P 1001 P
2 (010) 0010 P 0110 P 1010 P
3 (011) 0011 P 0111 P 1011 P
4 (100) 0100 P 1100 P

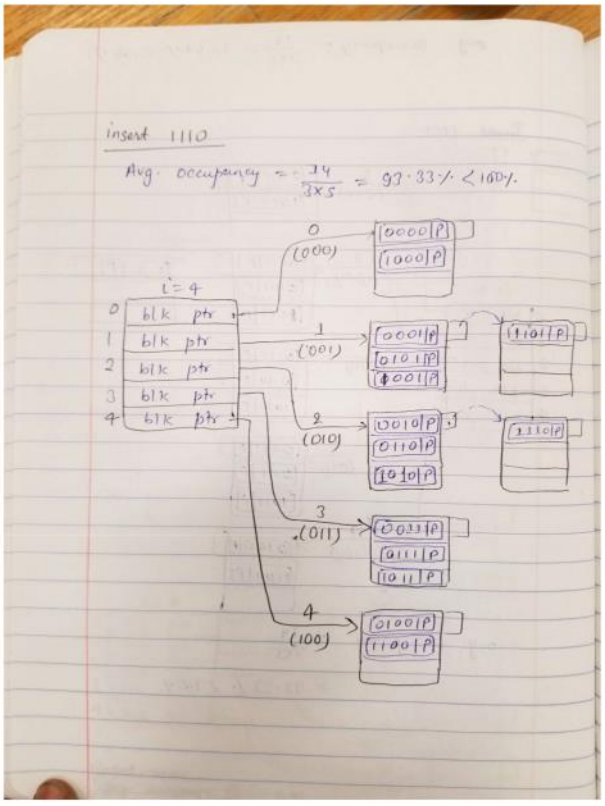
avg. occupancy = $\frac{13}{3 \times 5}$ = 86.67 % < 100 %.

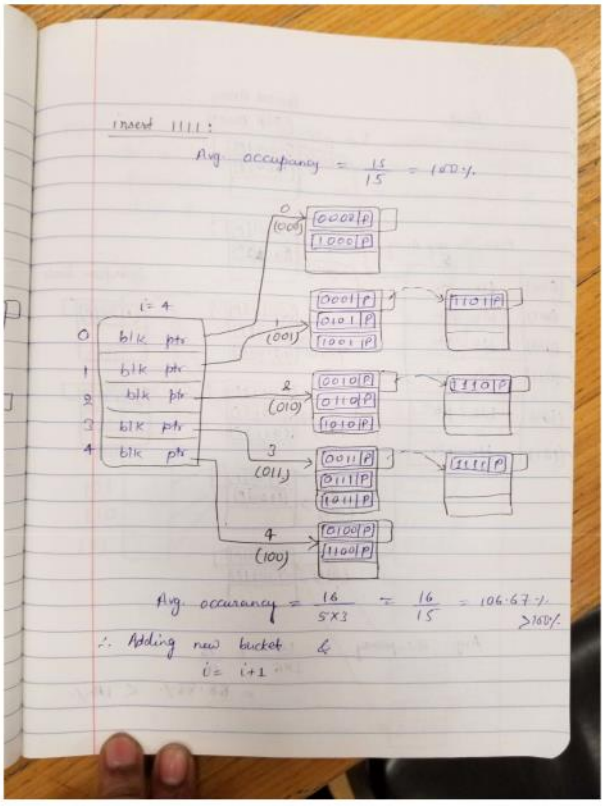
Insert 1101

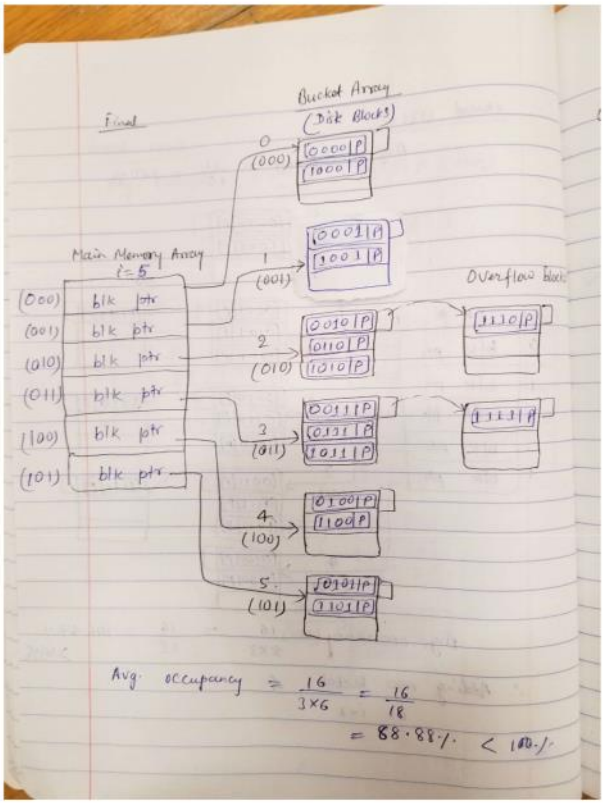


avg. occupancy = $\frac{14}{3 \times 5}$

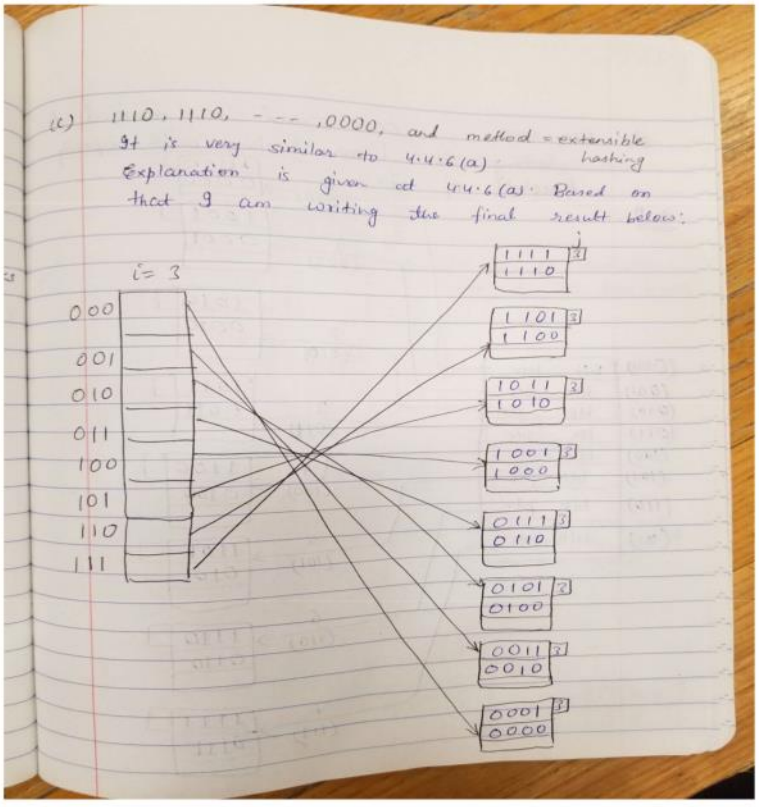
= 93.33 % < 100 %.


insert 1110
Avg. occupancy = 14/3x5 = 93.33% < 160%
i = 4
blk ptr
0 (000)
1 (001)
2 (010)
3 (011)
4 (100)
0000|P
1000|P
0001|P
0101|P
1001|P
1101|P
0010|P
0110|P
1010|P
1110|P
0011|P
0111|P
1011|P
0100|P
1100|P

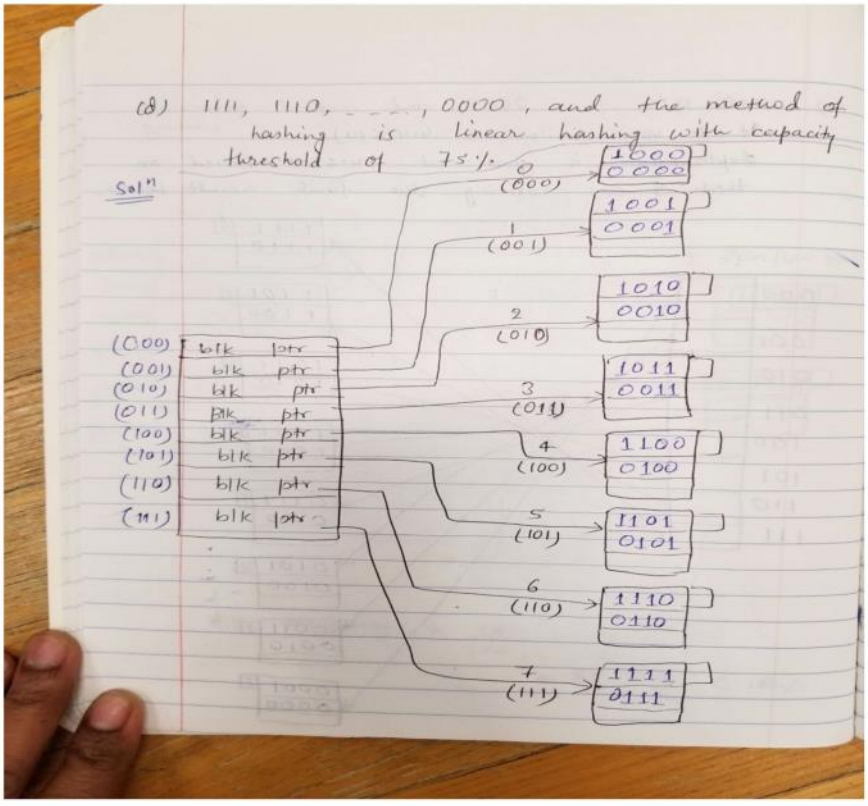

insert 1111:
Avg. occupancy = 15/15 = 100%
i = 4
blk ptr
0 (000)
1 (001)
2 (010)
3 (011)
4 (100)
0000|P
1000|P
0001|P
0101|P
1001|P
1101|P
0010|P
0110|P
1010|P
1110|P
0011|P
0111|P
1011|P
1111|P
0100|P
1100|P
Avg. occurancy = 16/5x3 = 16/15 = 106.67%
>100%
∴ Adding new bucket &
i = i+1



c) 1111,1110,..., 0000, and the method of hashing is extensible hashing.



d) 1111,1110,..., 0000, and the method of hashing is linear hashing with a capacity threshold of 75%.


(8) 1111, 1110, ..., 0000, and the method of hashing is linear hashing with capacity threshold of 75%.
Soln
(000) blk ptr
(001) blk ptr
(010) blk ptr
(011) blk ptr
(100) blk ptr
(101) blk ptr
(110) blk ptr
(111) blk ptr
0 (000) 1000 0000
1 (001) 1001 0001
2 (010) 1010 0010
3 (011) 1011 0011
4 (100) 1100 0100
5 (101) 1101 0101
6 (110) 1110 0110
7 (111) 1111 0111