CS3223 Tutorial 2

Gary Lim

Admin

- Lab 2 due today!
- Tutorial participation
- Telegram Group Link: https://t.me/+pnE_EW3wsl0yMmM1

Tutorial Breakdown

01 Quick Review

5 – 10 min

02 Tutorial Questions

30 – 40 min

03 Q & A / Extra Questions

5 – 10 min

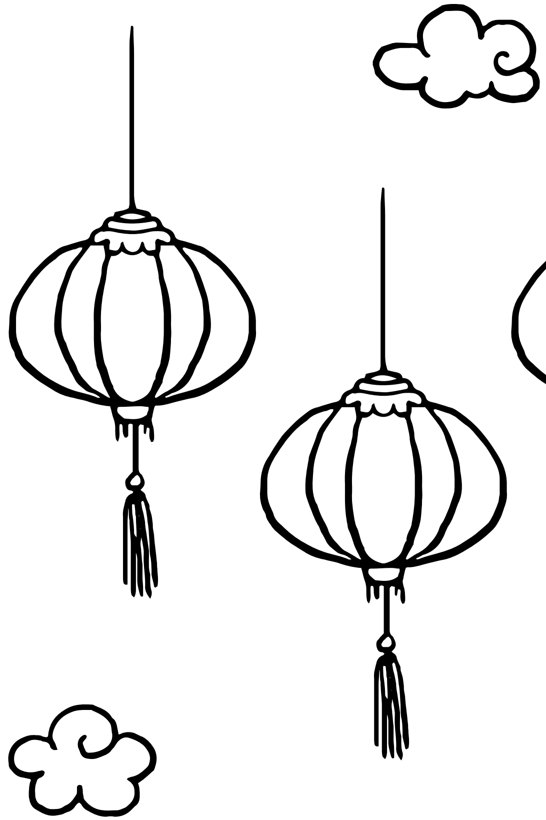
01

Review

Hash Index

02

Tutorial Questions



Q1

1. Consider a relation EMP(eno, sno, name, salary) containing 1,000,000 employee records. Suppose each page of the relation contains 10 records, and relation EMP is organized as a sorted file. Suppose sno is a candidate key, with values lying in the range 0-999,999, and that EMP is stored in sno order. Assume that each page is 4K bytes, eno and sno are each 4 bytes, and a pointer to a page is also 4 bytes. For each of the following queries, state which of the following three approaches is most likely to be the cheapest (in terms of I/O count). Justify your answer. For simplicity, assume that the data are uniformly distributed in the domain.

   - Scan the sorted file for EMP.
   - Use a (clustered) B+-tree index on sno. Assume that the nodes are 100% full, and that format 2 is used for the index.
   - Use a hash index on sno. You may assume ideal situation of no overflows, and each bucket is 70% full (and a minimum number of buckets is used). You may assume that format 1 is used for the index.

   a) Find all employees whose sno < 100,000.
   b) Find the employee whose sno = 10.
   c) Find the employees whose sno is in the range 10,000-10,010.
   d) Find the employees whose sno is not 10.

Q1

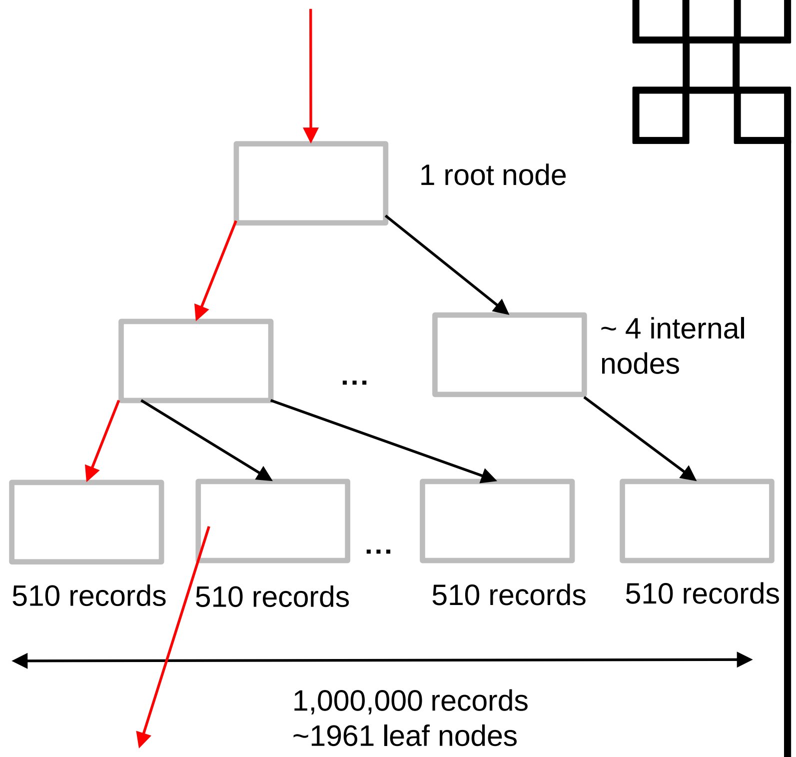
Scan Sorted File	10 records 10 records ...
Clustered B+ tree on sno. Nodes are 100% full, format 2	m = 2d, where d = order (2d + 1) * 4 + 2d * 4 <= 4096 (4 kB) d ~ 255 m = **510 keys per leaf node** Levels = $\log_{510}$(1,000,000) = log(1,000,000) / log(510) = 2.2 = **approx. 3** sno ptr ... sno ptr 4B 4B ... 4B 4B
Hash Index on sno, no overflow, bucket is 70% full, format 1	Each bucket only consists of primary data page (no overflow pages) and is 70% full Hence 7 records per page (bucket) Total buckets = 1,000,000 / 7 = 142858

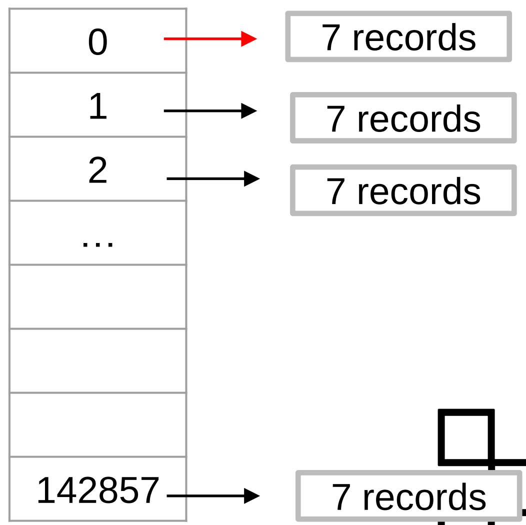
EMP(eno, sno, name, salary)

- **1,000,000** employee records.
- **10 records** per page of relation (data page)
- **sorted** on **sno**
- **sno** is a candidate key, with values lying in the range **0-999,999**
- **4K bytes** per page (B+ tree node / Hash Index primary/overflow page)
- eno and sno are each **4 bytes**
- apointer to a page is **4 bytes**
- data are uniformly distributed in the domain.

Q1

Scan Sorted File	10 records 10 records ...
Clustered B+ tree on sno. Nodes are 100% full, format 2	m = 2d, where d = order (2d + 1) * 4 + 2d * 4 <= 4096 (4 kB) d ~ 255 m = **510 keys per leaf node** Levels = $\log_{510}(1{,}000{,}000) = \log(1{,}000{,}000) / \log(510) = 2.2$ = **approx. 3** sno ptr ... sno ptr 4B 4B ... 4B 4B
Hash Index on sno, no overflow, bucket is 70% full, format 1	Each bucket only consists of primary data page (no overflow pages) and is 70% full Hence 7 records per page (bucket) Total buckets = 1,000,000 / 7 = 142858





Q1

EMP(eno, sno, name, salary)

- **1,000,000** employee records.
- **10 records** per page of relation (data page)
- **sorted** on **sno**
- **sno** is a candidate key, with values lying in the range **0-999,999**
- **4K bytes** per page (B+ tree node / Hash Index bucket)
- eno and sno are each **4 bytes**, and a
- pointer to a page is **4 bytes**
- data are uniformly distributed in the domain.

a) Find all employees whose sno < 100,000.

Scan Sorted File	10 records per page, 100,000 records in total **10,000 I/O**
Clustered B+ tree on sno. Nodes are 100% full, format 2	**10,000 I/O + additional I/O** to access B+ tree nodes (each node is a page)
Hash Index on sno, no overflow, bucket is 70% full, format 1	**142,858 I/O** by scanning all

Q1

EMP(eno, sno, name, salary)

- **1,000,000** employee records.
- **10 records** per page
- **sorted** on **sno**
- **sno** is a candidate key, with values lying in the range **0-999,999**
- **4K bytes** per page
- eno and sno are each **4 bytes**, and a
- pointer to a page is **4 bytes**
- data are uniformly distributed in the domain.

b) Find the employee whose sno = 10.

Scan Sorted File	If we have access to the first page of the file, and it is chained, then we need 2 pages at most. But consider the general case where sno can be anything, e.g. sno = 429580
Clustered B+ tree on sno. Nodes are 100% full, format 2	**4 I/O.** One I/O to retrieve the index, two I/O to reach leaf node, one I/O to retrieve data page
Hash Index on sno, no overflow, bucket is 70% full, format 1	**2 I/O.** One I/O to retrieve the index, the other to access the primary page

Q1

EMP(eno, sno, name, salary)

- **1,000,000** employee records.
- **10 records** per page
- **sorted** on **sno**
- **sno** is a candidate key, with values lying in the range **0-999,999**
- **4K bytes** per page
- eno and sno are each **4 bytes**, and a
- pointer to a page is **4 bytes**
- data are uniformly distributed in the domain.

c) Find the employees whose sno is in the range 10,000-10,010.

Scan Sorted File	In the best case, we scan till sno = 10,000 **1001 I/O**
Clustered B+ tree on sno. Nodes are 100% full, format 2	4 + 1 = **5 I/O**. Four I/O to get to the data page consisting 10,000 to 10,009, then use the pointer to the next page to retrieve 10,010
Hash Index on sno, no overflow, bucket is 70% full, format 1	**11 I/O**. 11 for each of the data entries, assuming they are hashed to different buckets

Q1

EMP(eno, sno, name, salary)

- **1,000,000** employee records.
- **10 records** per page
- **sorted** on **sno**
- **sno** is a candidate key, with values lying in the range **0-999,999**
- **4K bytes** per page
- eno and sno are each **4 bytes**, and a
- pointer to a page is **4 bytes**
- data are uniformly distributed in the domain.

d) Find the employees whose sno is not 10.

Scan Sorted File	**100,000 I/O**
Clustered B+ tree on sno. Nodes are 100% full, format 2	**100,000 I/O + additional 3 I/O** to access B+ tree nodes
Hash Index on sno, no overflow, bucket is 70% full, format 1	**142,858 I/O** by scanning all

Q2

Consider a relation Book(Bookid: integer, Author: string, …) where **Bookid** is the key, and its values are assigned in **increasing order** starting from 1. Suppose Book has 6 million records of 200 bytes each. Suppose we have to perform 10,000 single-record accesses, and 100 range queries of 0.005% of the file, with Bookid as the search key.

300 records per range query

Which of the above two methods is better for the application? Under what circumstance will the "loser" outperform the "winner"?

Hash Index

Load factor of 70%, bucket size of 4096 bytes, Format 1, no overflow buckets

Records per bucket = bucket size / record size * 0.7
= floor(4096 / 200) * 0.7 = 14
Number of buckets = total records / records per bucket = 6000000 / 14 = 428,572 buckets

10,000 single-record accesses = 10,000 I/O
100 range queries = 100 * 300 = 30,000 I/O
Total = 40,000 I/O

Size per page = **4096 bytes**
Records per page = 20 * 0.7 = **14**
Pages required for 300 records = **22**
22 entries can span across **2-3 leaf nodes**

B+ Tree Index

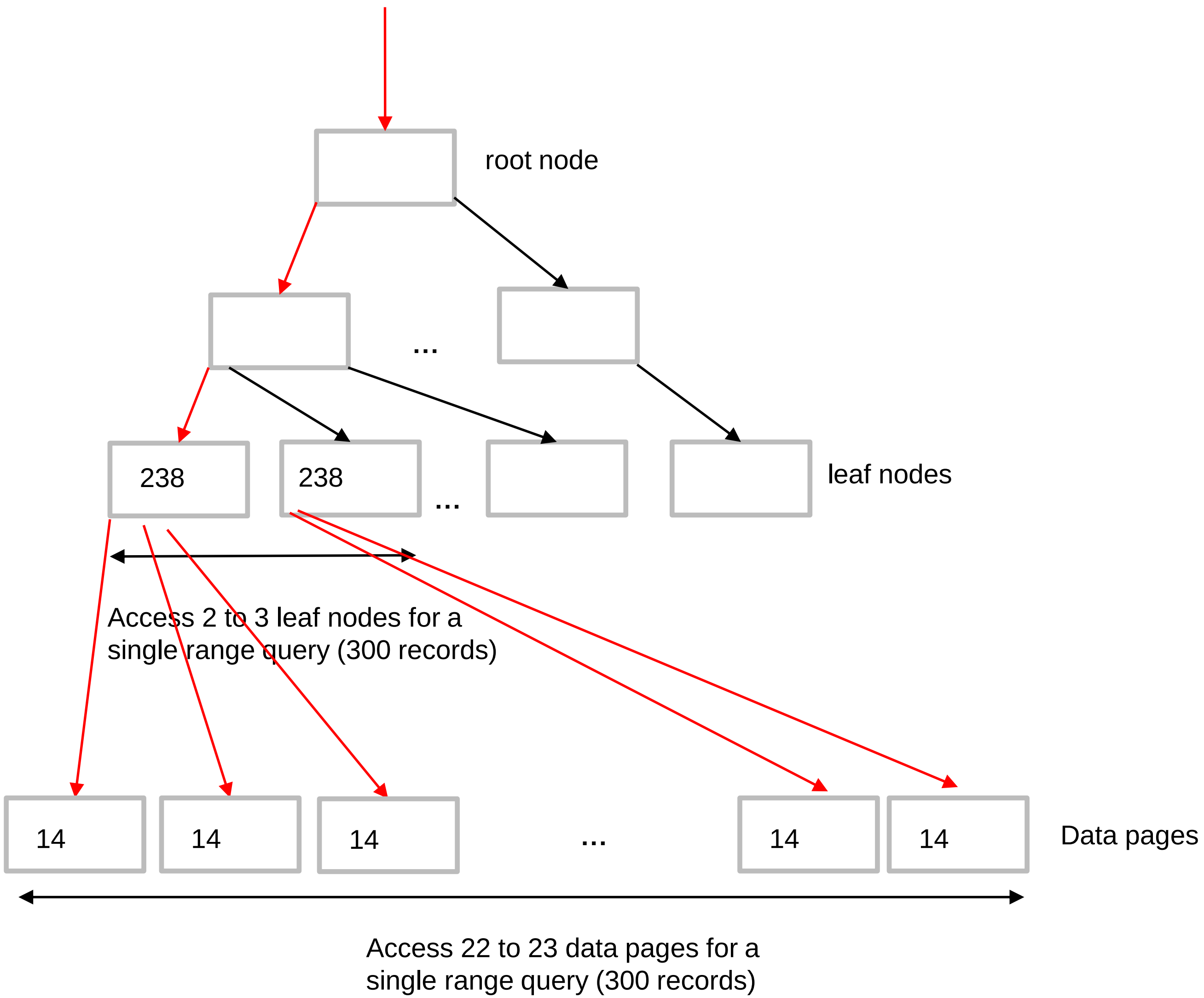
Nodes are 70% full, node has a size of 4096 bytes, key is 8 bytes each, address is 4 bytes each

m = 2d
(2d + 1) * 4 + 2d * 8 <= 4096
d = 170 (round down)
m = **340 records per leaf node**
If 70% full, **238 records per leaf node**
$\log_{238}(6000000)$ = **3 levels**

+1 to access data page

10,000 single-record accesses = 10,000 * (3+1) = 40,000 I/O
100 range queries = 100 * (3+1+22) = 2600 I/O
Total = 42,700 I/O

Q2


root node
...
238
238
...
leaf nodes
Access 2 to 3 leaf nodes for a
single range query (300 records)
14
14
14
...
14
14
Data pages
Access 22 to 23 data pages for a
single range query (300 records)

Q2

Discussion

- Winner is hash index (based on our assumptions)

When will B+ tree index (loser) outperform hash index (winner)?

- number of range queries increases or the range increases, or the number of single-record queries reduces

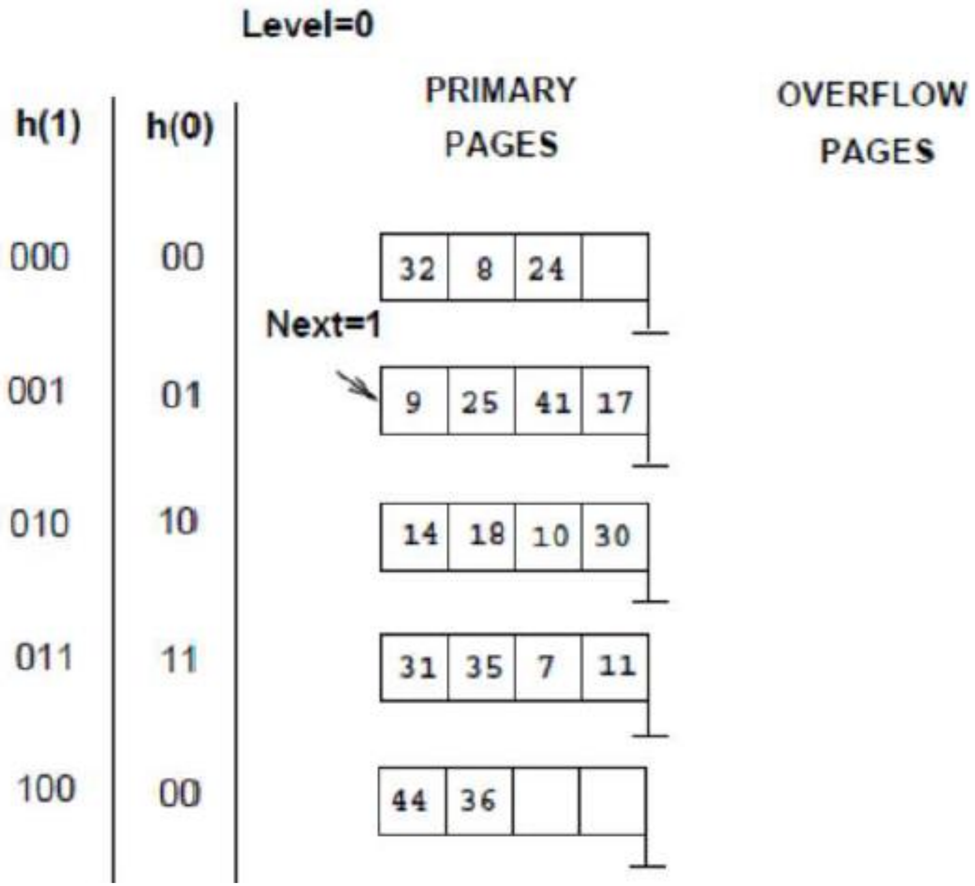
When will hash index (winner) outperform B+ tree index (winner)?

- The other way round (more single record retrieval)

Q3

Consider the Linear Hashing index shown in the figure below. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

a) What can you say about the last entry that was inserted into the index if you know that there have been no deletions from this index so far?



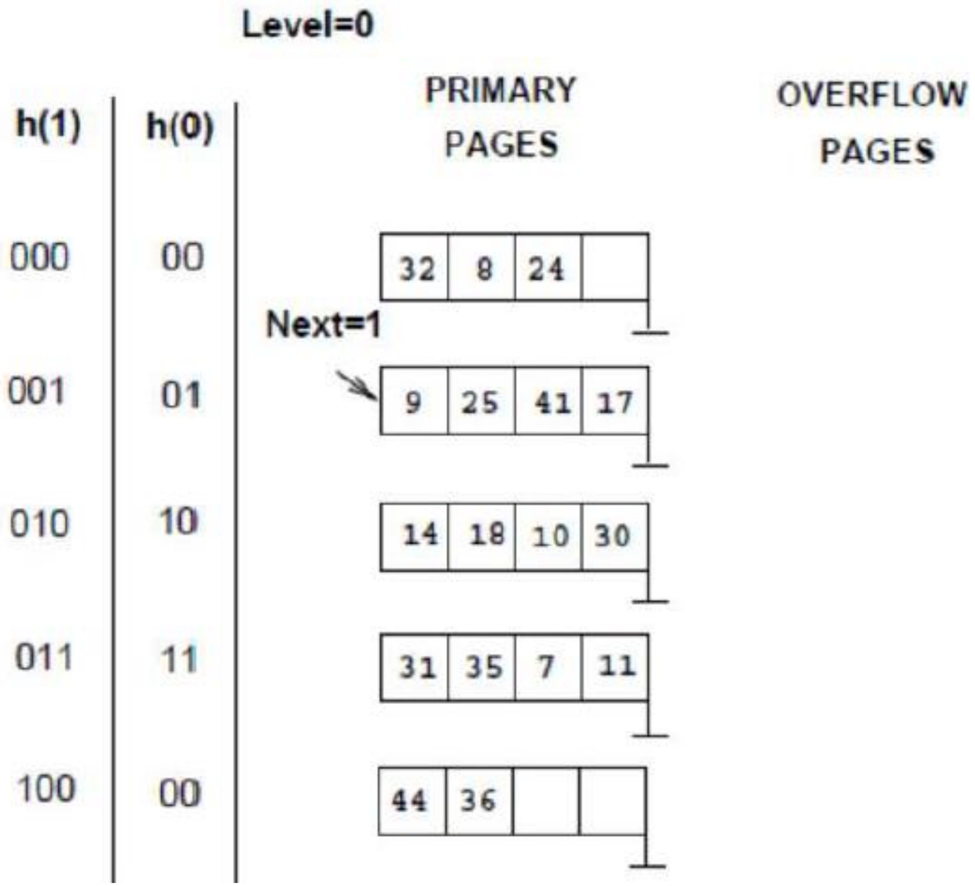
If the last item that was inserted had a hashcode h0(keyvalue) = 00 then it caused a split (as there are no overflow pages for any of the other buckets), otherwise, any value could have been inserted.

Q3

Consider the Linear Hashing index shown in the figure below. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

b) Show the index after inserting an entry with hash value 4, and 15.

Binary representation
4 – 100
15 – 1111

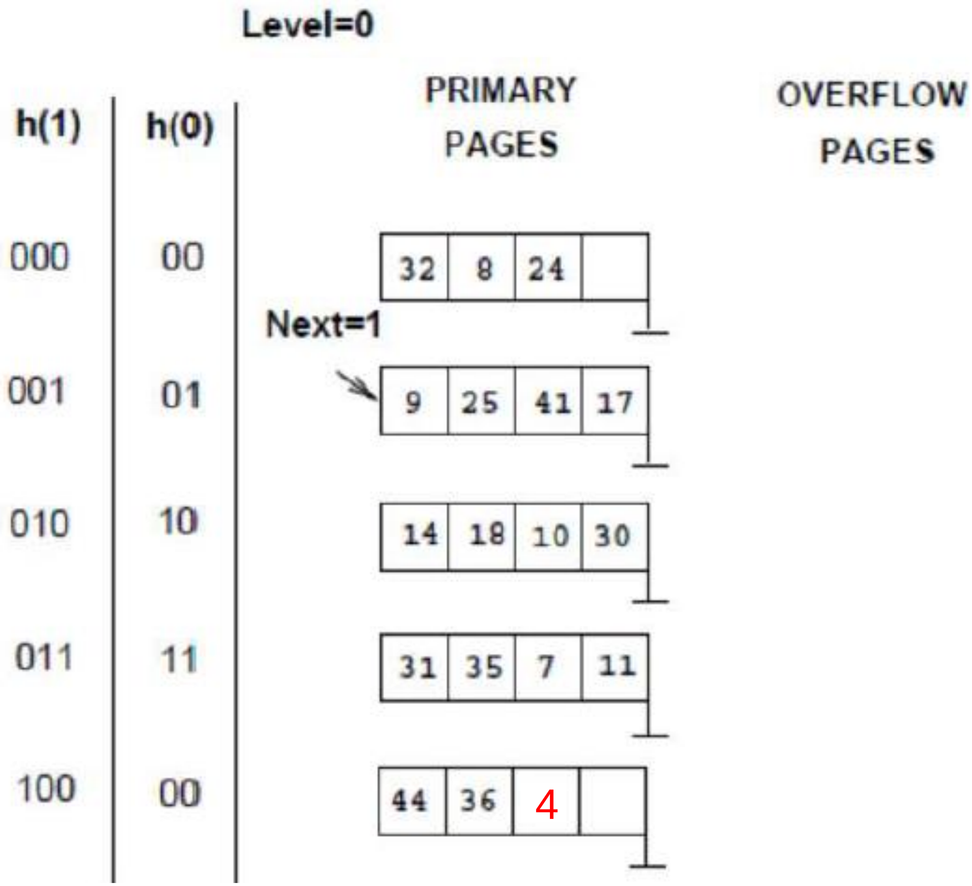


Q3

Consider the Linear Hashing index shown in the figure below. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

b) Show the index after inserting an entry with hash value 4, and 15.

Binary representation
4 – 100
15 – 1111

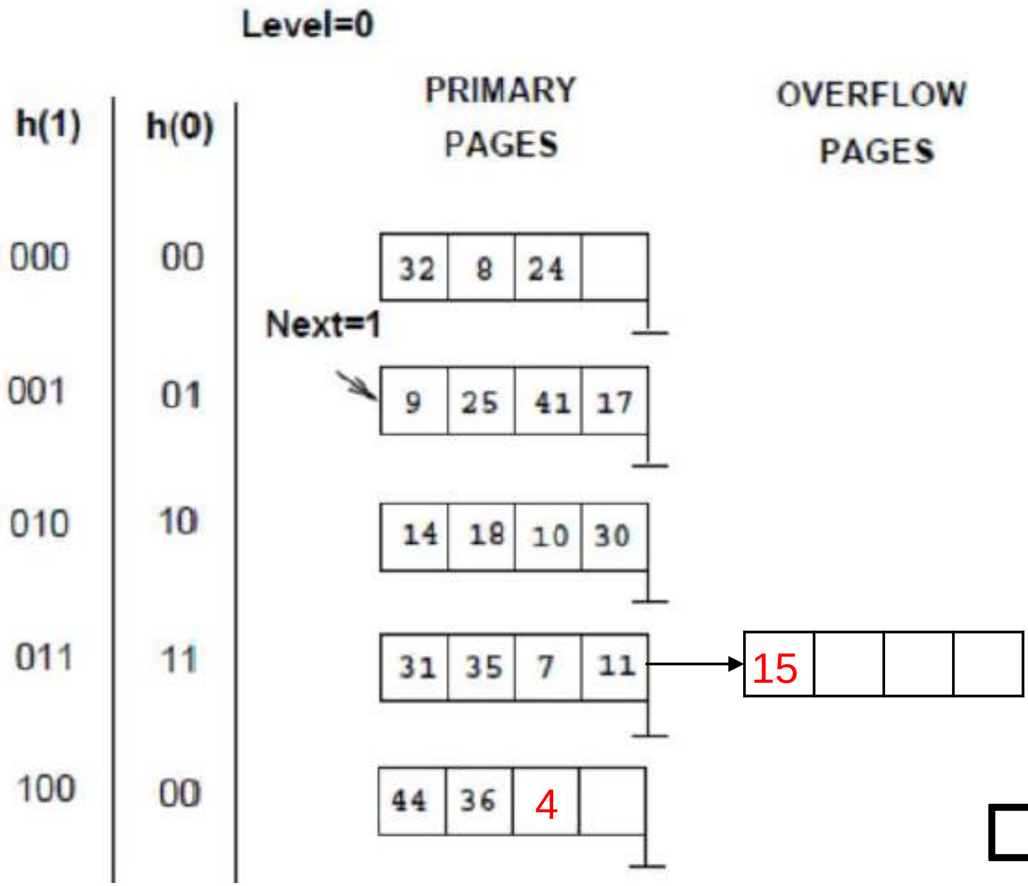


Q3

Consider the Linear Hashing index shown in the figure below. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

b) Show the index after inserting an entry with hash value 4, and 15.

Binary representation
4 – 100
15 – 1111



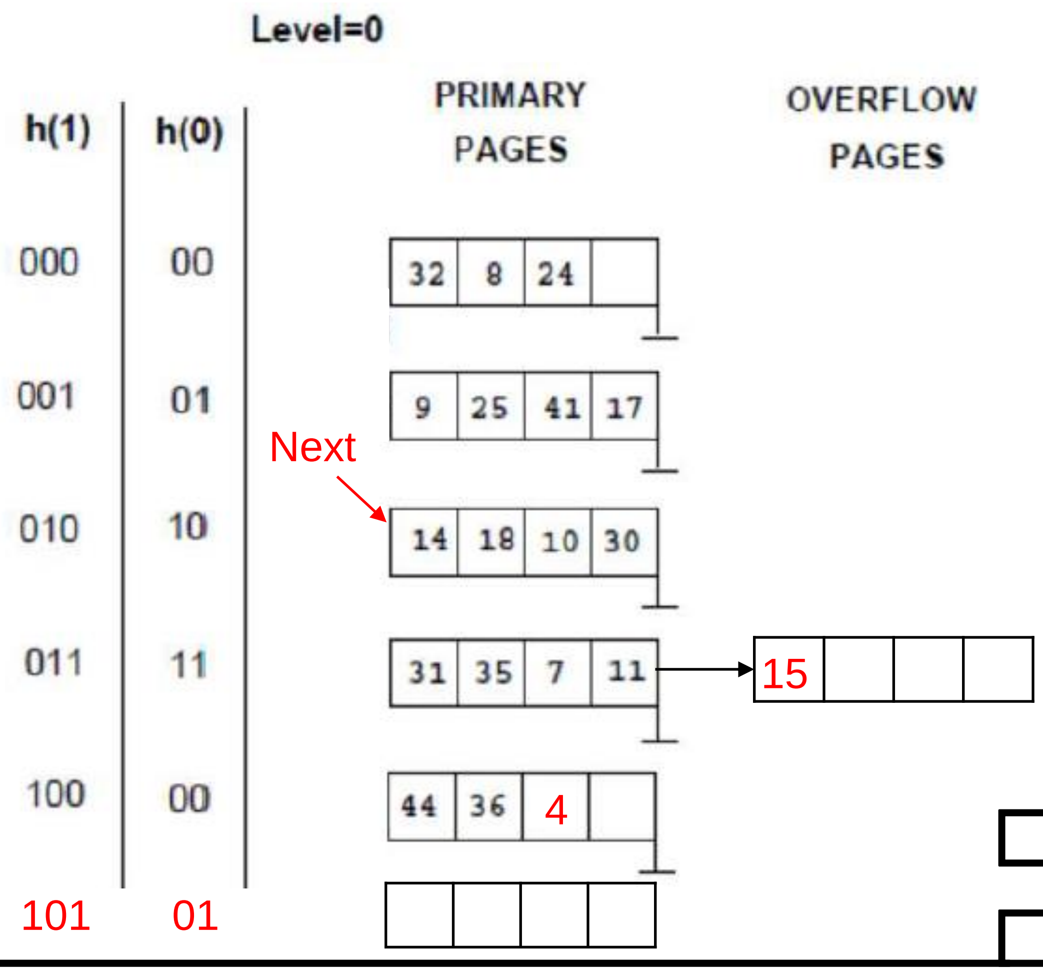
Q3

Consider the Linear Hashing index shown in the figure below. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

b) Show the index after inserting an entry with hash value 4, and 15.

Binary representation
4 – 100
15 – 1111

9 – 1**001** (8+1)
25 – 11**001** (16+8+1)
41 – 101**001** (32+8+1)
17 – 10**001** (16+1)

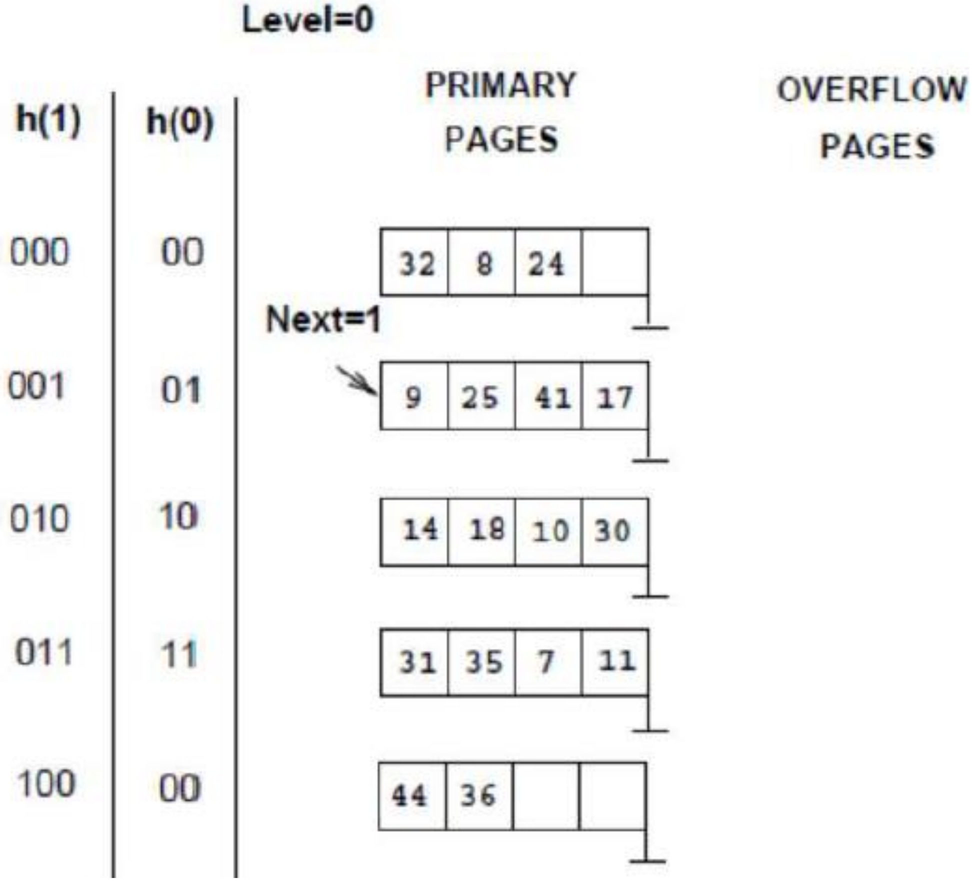


Q3

Consider the Linear Hashing index shown in the figure below. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

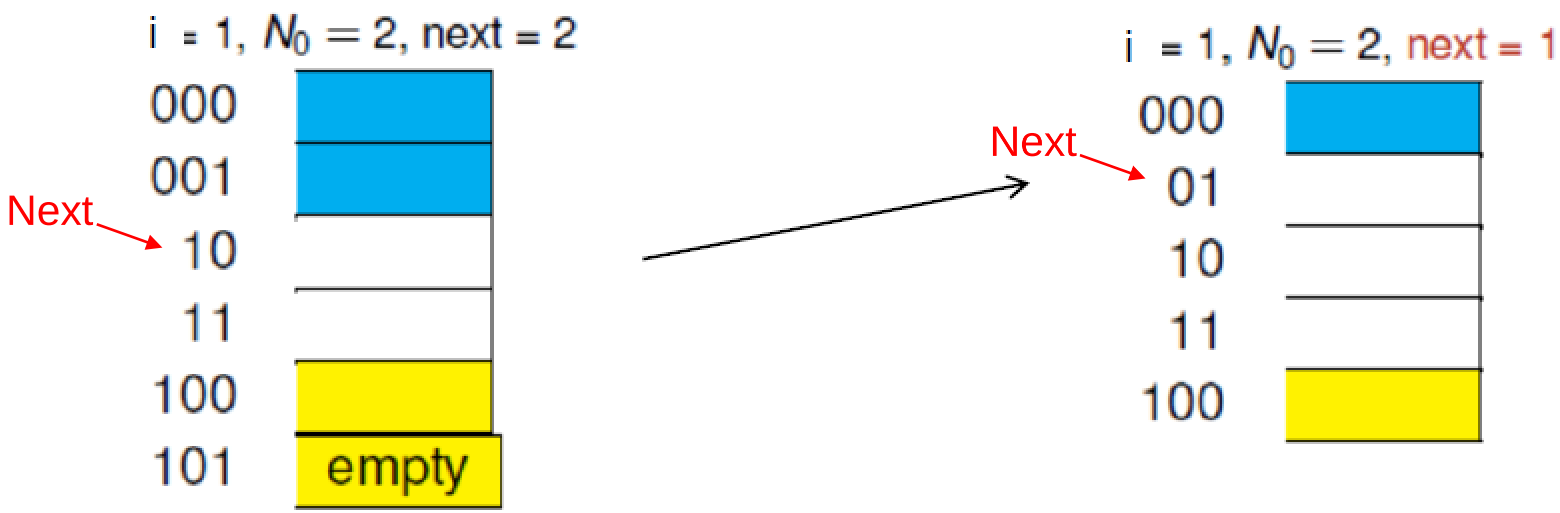
c) Show the index after deleting the entries with hash values 36 and 44 into the original tree. Assume that the full deletion algorithm is used.

Delete 36 and 44 from the last bucket



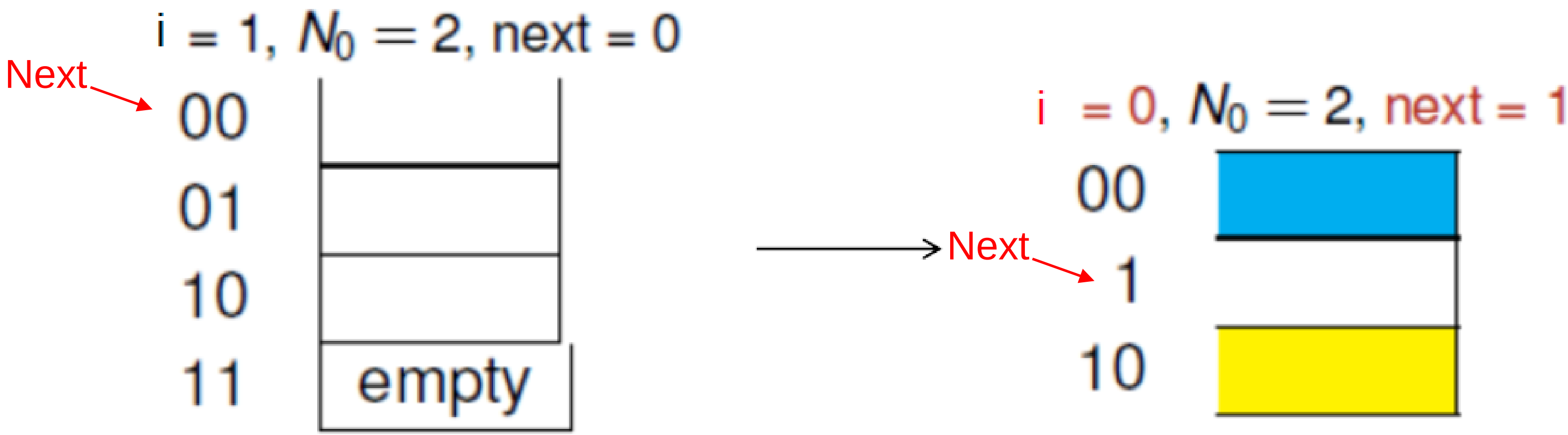
Linear Hashing: Deletion

- Locate bucket and delete entry
- If the last bucket $B_{Ni + next - 1}$ becomes empty, it can be removed
- Case 1: If next > 0
  - Decrement next



Linear Hashing: Deletion

- Case 2: If (next = 0) and (i > 0)
  - Update next to point to the last bucket in previous level $B_{ni/2-1}$
  - Decrement level



Q3

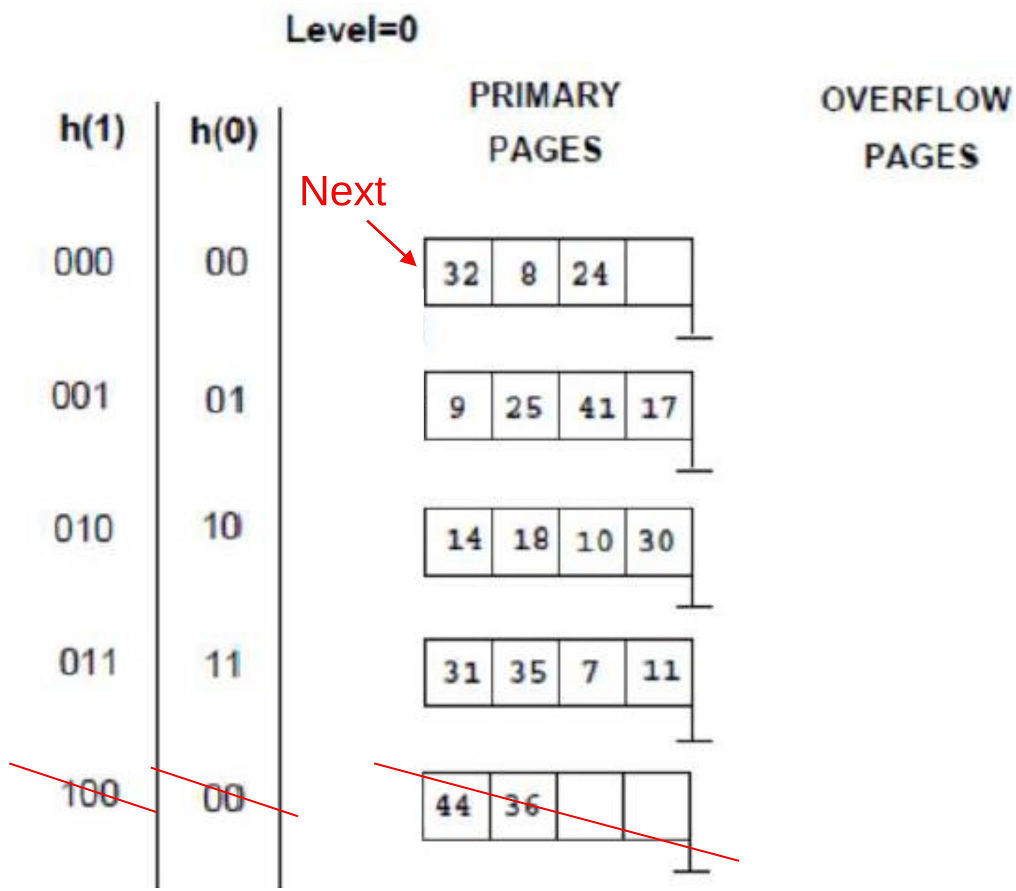
Consider the Linear Hashing index shown in the figure below. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

c) Show the index after deleting the entries with hash values 36 and 44 into the original tree. Assume that the full deletion algorithm is used.

Delete 36 and 44 from the last bucket

Delete the last bucket

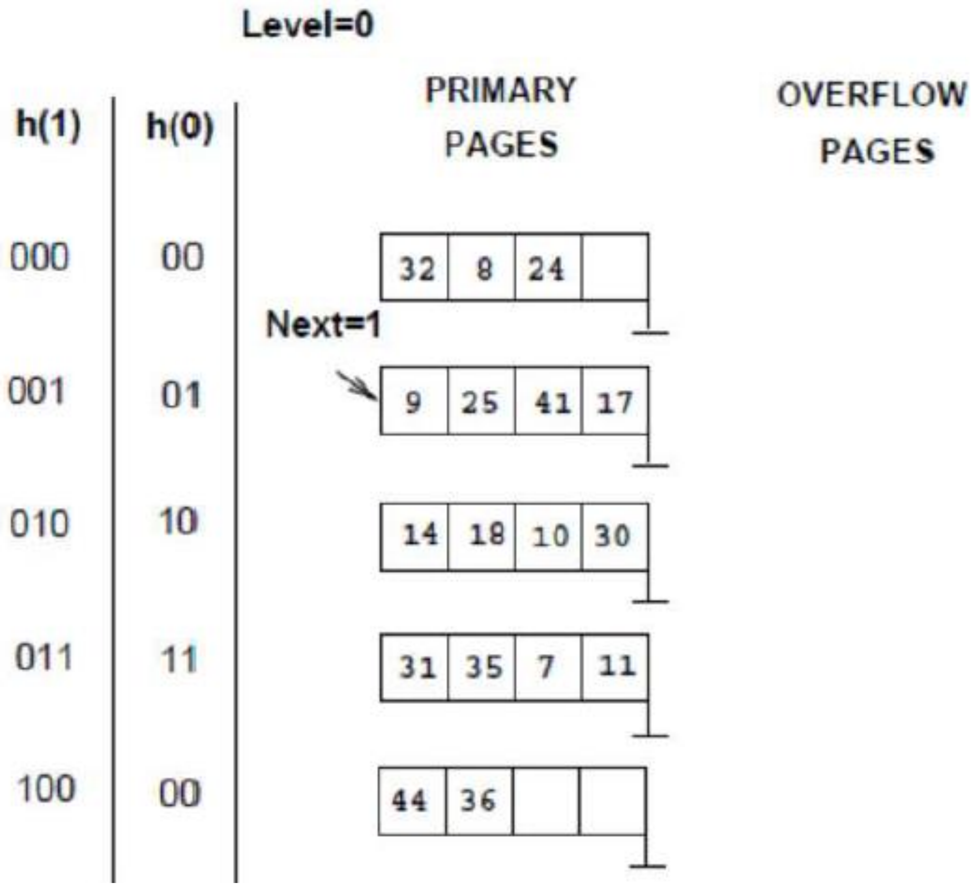
Decrement next pointer



Q3

Consider the Linear Hashing index shown in the figure below. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

d) Find a list of entries whose insertion into the original index would lead to a bucket with two overflow pages. Use as few entries as possible to accomplish this.



Q3

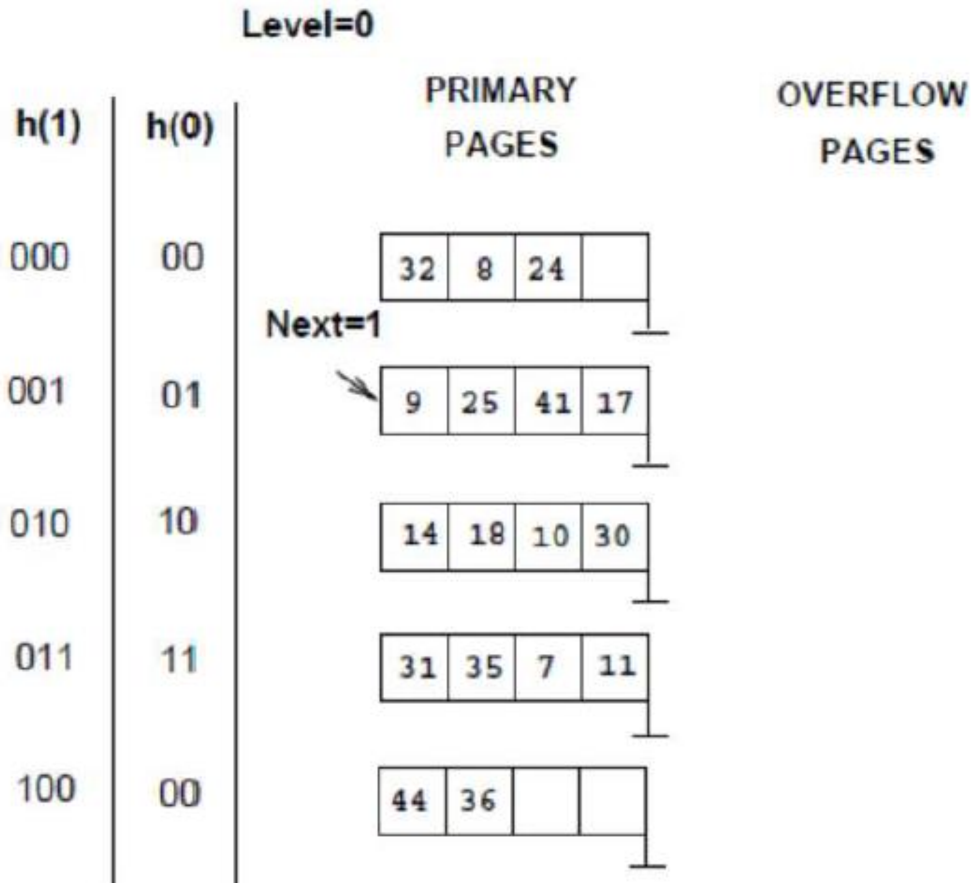
Consider the Linear Hashing index shown in the figure below. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

d) Find a list of entries whose insertion into the original index would lead to a bucket with two overflow pages. Use as few entries as possible to accomplish this.

All ending with 111
63, 127, 255, 511, 1023
The first insertion causes a split and causes an update of Next to 2. The insertion of 1023 causes a subsequent split and Next is updated to 3 which points to this bucket. This overflow chain will not be redistributed until three more insertions (a total of 8 entries) are made.

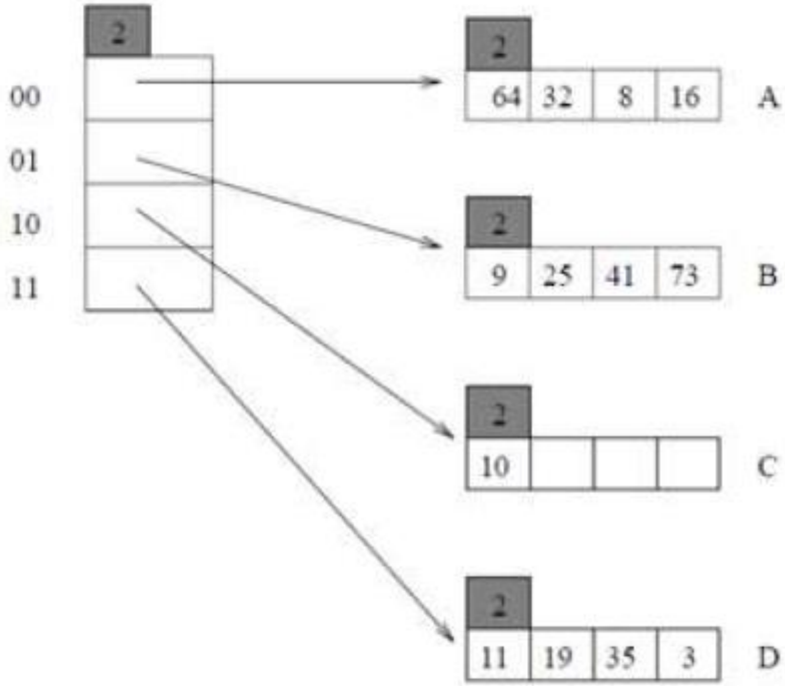
OR
Keep inserting **-001 entries** will cause bucker 01 to overflow into 2 pages

OR
any other valid answers

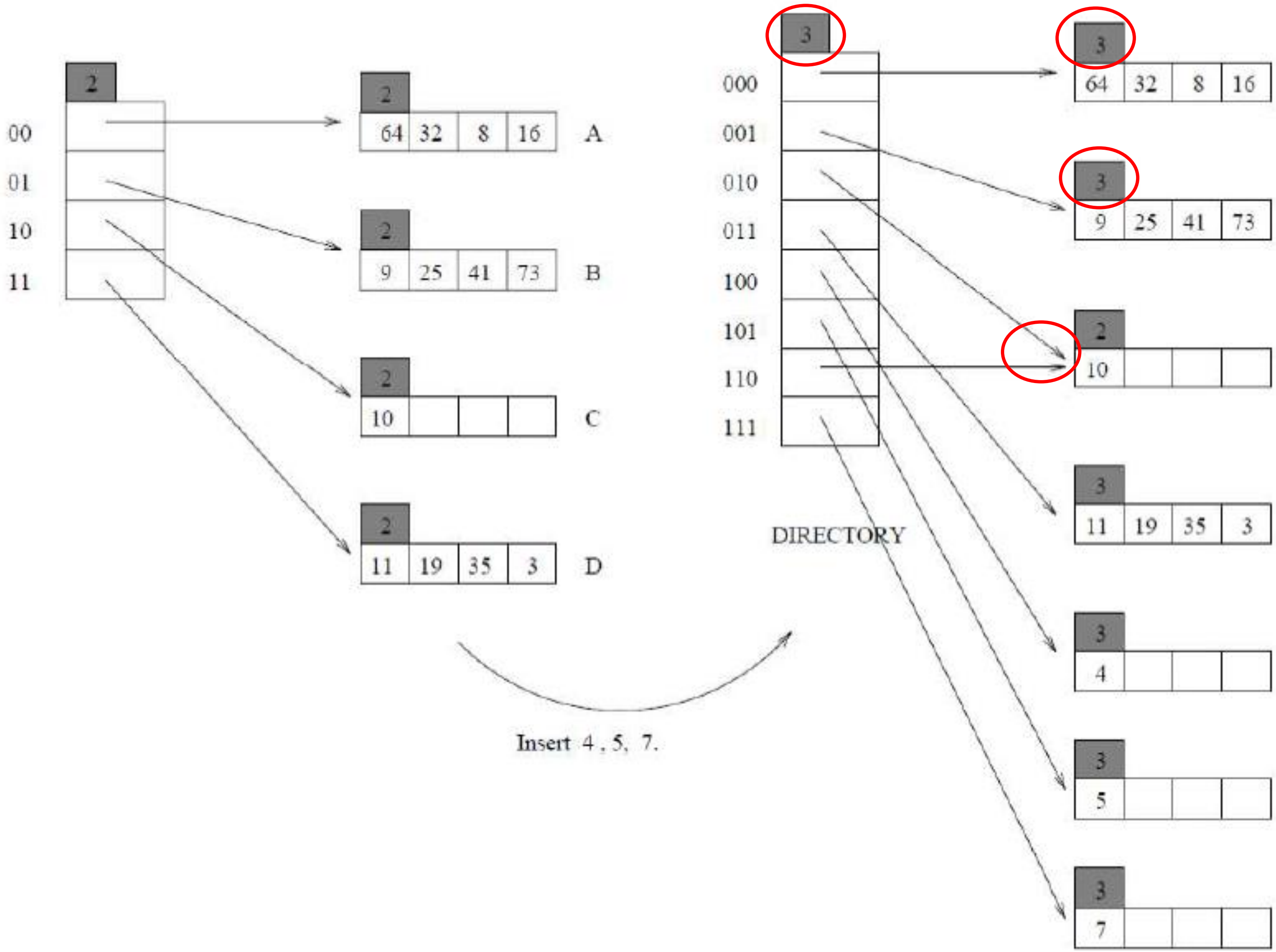


Q4

Consider the extendible hash table below. Insert 4, 5 and 7. Show the resultant hash table? Now, delete 7, 5, and 4. What is the resultant hash table?

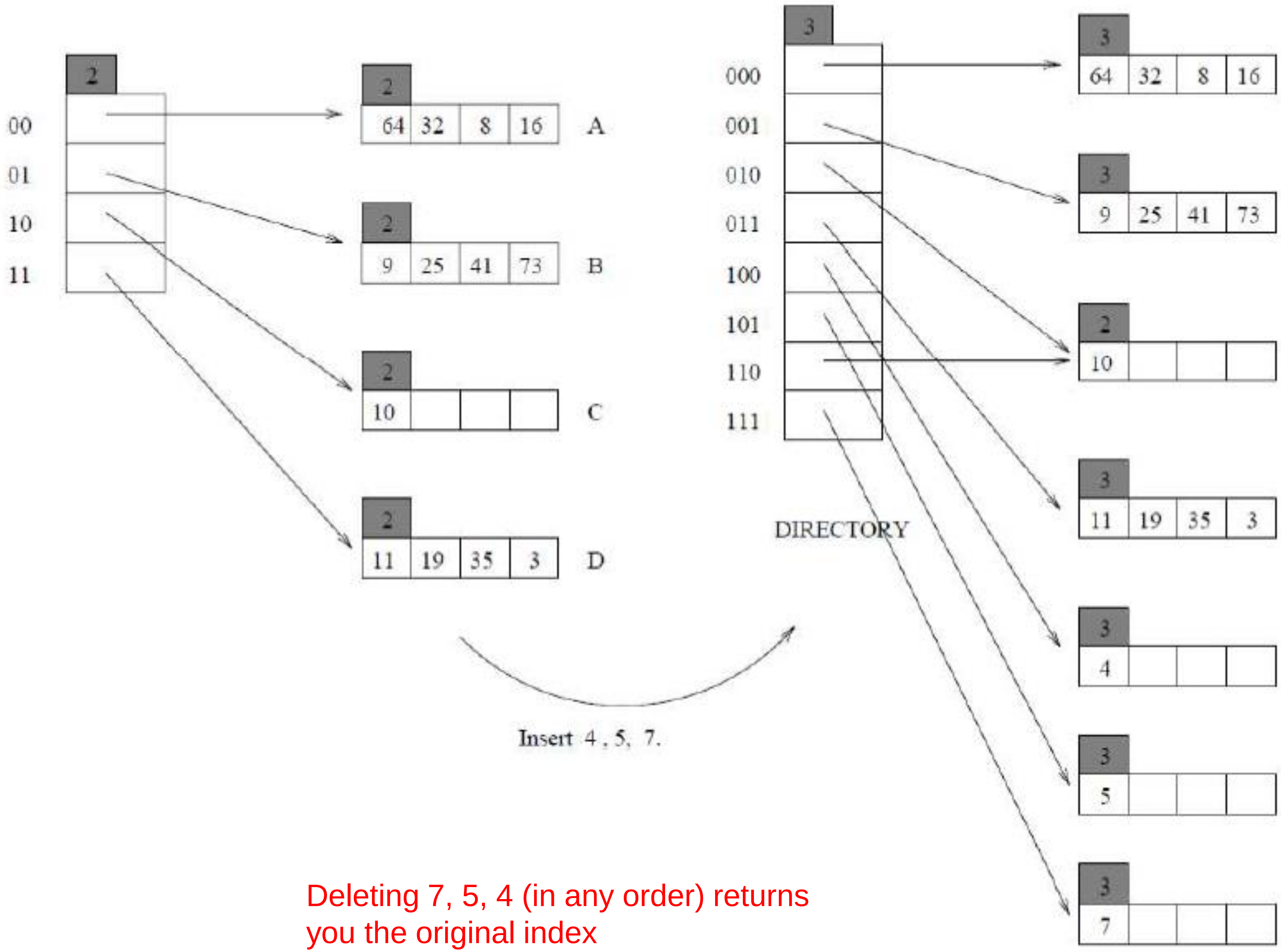


Q4



2
00
01
10
11
2
64
32
8
16
A
2
9
25
41
73
B
2
10
C
2
11
19
35
3
D
Insert 4 , 5, 7.
3
000
001
010
011
100
101
110
111
DIRECTORY
3
64
32
8
16
3
9
25
41
73
2
10
3
11
19
35
3
3
4
3
5
3
7

Q4



Deleting 7, 5, 4 (in any order) returns you the original index

03 ❀ Extra Qs

Extra

1. Is Linear Hashing order independent?

Ans:

- An index is **order dependent;** inserting the same set of keys in a different ordering result in different structure.
- Here, we look at the structure of the hash index. It is order dependent. Counter example: 4 buckets initially. Insert a1-a8 into the buckets. Case 1: insert a1-a3 into bucket 1; then insert a4-a8 to bucket 4. Case 2: insert a4-a8 to bucket 4 first; then insert a1-a3. We will have two different structures (due to different occurrence of overflow pages and bucket splitting)

Extra

2. Suppose we use a linear hashing scheme, but there are pointers to records from outside. These pointers prevent us from moving records between blocks (which is what LH does – redistribute records whenever there is a split). Suggest ways that we could modify the structure to allow pointers from outside?

2022: The year of the tiger