Tutorial 7

Indexing, B^+-tree, and Hashing

Part 1: Indexing

- Speed up access to desired data.
- **Search Key** - attribute to set of attributes used to look up records in a file.

Two kinds of index

- Ordered index
- Hash Index

Index

- **Primary index (clustering index):** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
- **Secondary index (non-clustering index):** an index whose search key specifies an order different from the sequential order of the file.

- **Dense index**: one index, one record
- **Sparse Index**: one index, many records; only applicable to clustering index

- **Single level Index**: index for data
- **Multilevel Index**: index of index

Exercise 1

- 假设一所学校保存了一份包含学生记录的档案: Student (sid:4 bytes, sname: 10 bytes, dept-id: 4 bytes), dept-id is the department(院系) id where a student belongs to.
- There exist 10,000 student records and 50 departments. A page is 128 bytes and a pointer is 4 bytes. The data file is sorted sequentially on sid.
- Q1: Given the data file only, what's the cost of finding students in a particular department (e.g., CSE)?
- Q2: How to improve?

Student (sid:4 bytes, sname: 10 bytes, dept-id: 4 bytes)

Exercise 1

- Student (sid:4 bytes, sname: 10 bytes, dept-id: 4 bytes)
- There exist 10,000 student records and 50 departments. A page is 128 bytes and a pointer is 4 bytes. The data file is sorted sequentially on sid.
- Q1: Given the data file only, what's the cost of finding students in a particular department (e.g., CSE)?

  Pages required to store data file is 1429.

  Records are sorted on sid, instead of dept-id.

  Given the data file, the only way is to sequentially scan it.

  The cost is 1429.

Exercise 1

- Student (sid:4 bytes, sname: 10 bytes, dept-id: 4 bytes)
- There exist 10,000 student records and 50 departments. A page is 128 bytes and a pointer is 4 bytes. The data file is sorted sequentially on sid.
- Q2: How to improve?

  Build index on dept-id.

  The entry for an index: (dept-id, pointer), each is of size 8 bytes.

Exercise 1

- Student (sid:4 bytes, sname: 10 bytes, dept-id: 4 bytes)
- There exist 10,000 student records and 50 departments. A page is 128 bytes and a pointer is 4 bytes. The data file is sorted sequentially on sid.
- 为了简单起见，我们假设每个系正好有 200 名学生。并且在索引页面中，一个页面只能包含同一系学生的指针。
- Q2-1: Build ordered index.

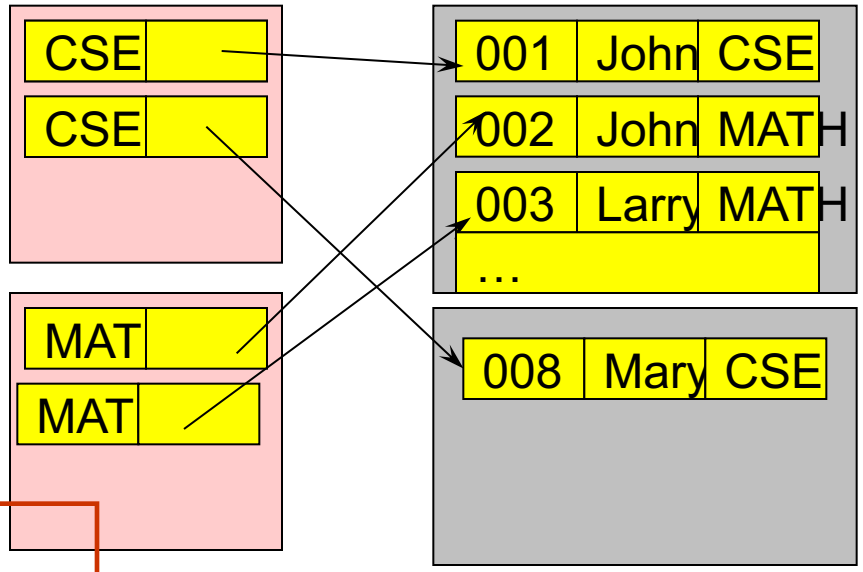
Index file

# of Pages needed:

⌈(200/⌊128/8⌋)⌉*50 = 650

Cost:

13+⌈ Log_2650 ⌉= 23



13=⌈(200/⌊128/8⌋)⌉
一个department需要的页数

Data file

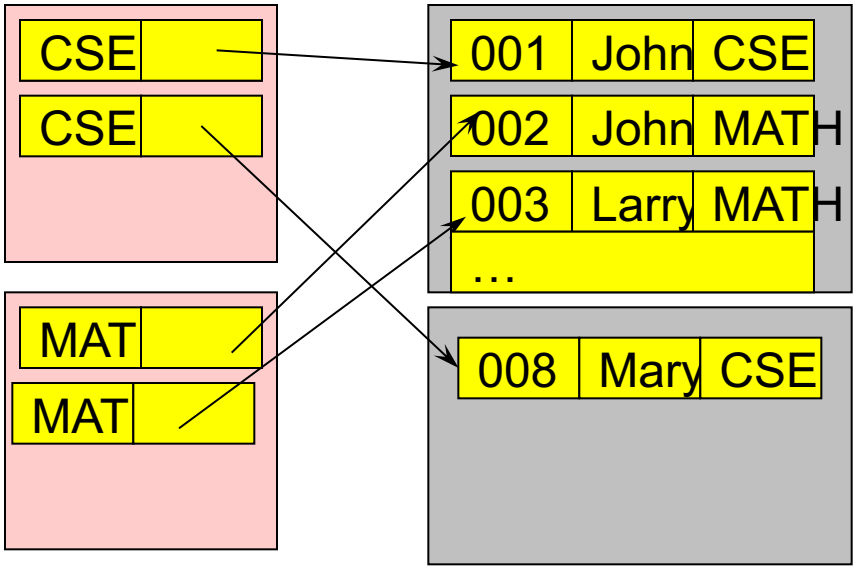
# of Pages needed:
⌈(10,000/⌊128/18 ⌋)⌉
= 1429

Cost: 200？ No

# of pages containing students in the same department.

Exercise 1

- Q2-1: Build ordered index.

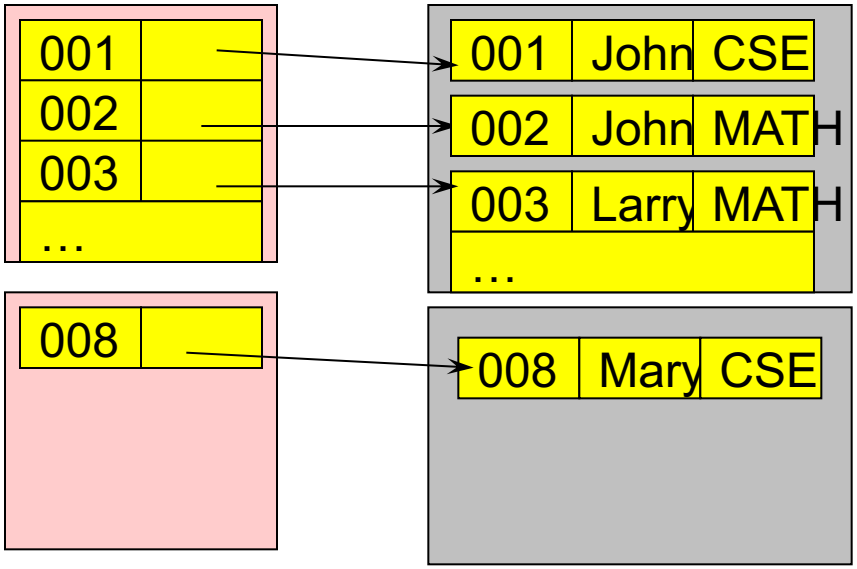


Primary index or secondary index?
Secondary index.

Dense index or sparse index?
Dense index.

Secondary index must be dense index.

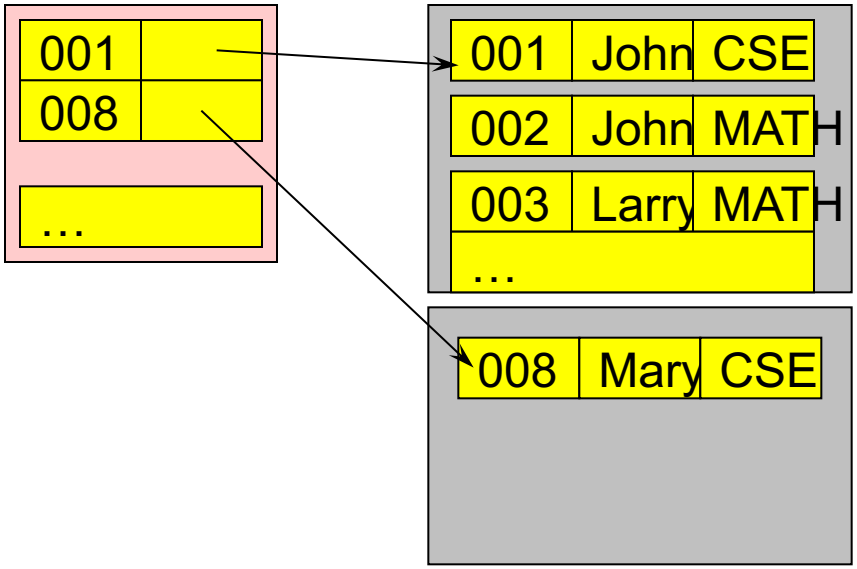
Exercise 1



Primary index or secondary index?
Primary index.

Dense index or sparse index?
Dense index.

Exercise 1



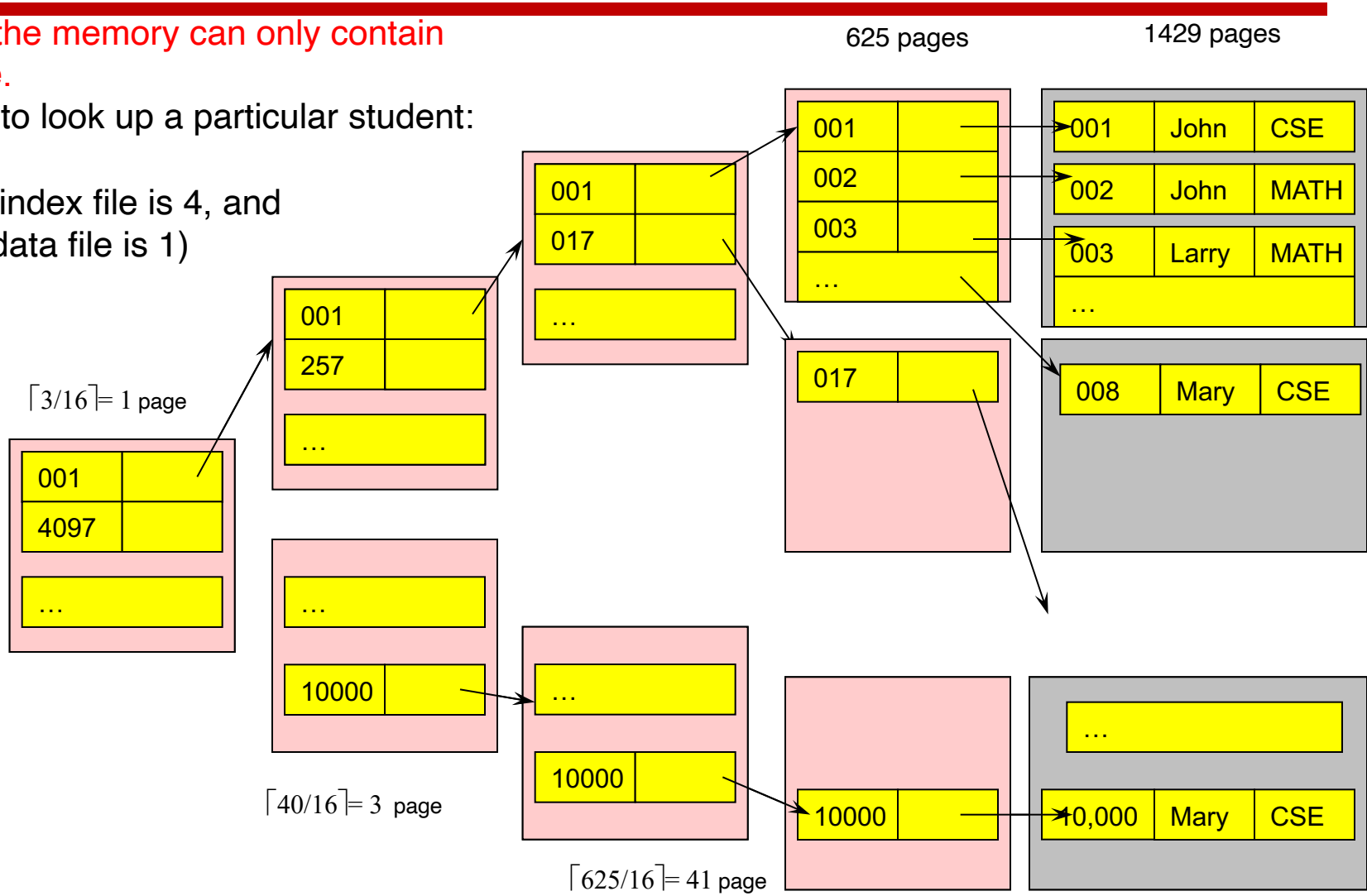
Primary index or secondary index?
Primary index.

Dense index or sparse index?
Sparse index.

Exercise 1

Assume the memory can only contain one page.

The cost to look up a particular student:
4 + 1 = 5
(Cost on index file is 4, and cost on data file is 1)



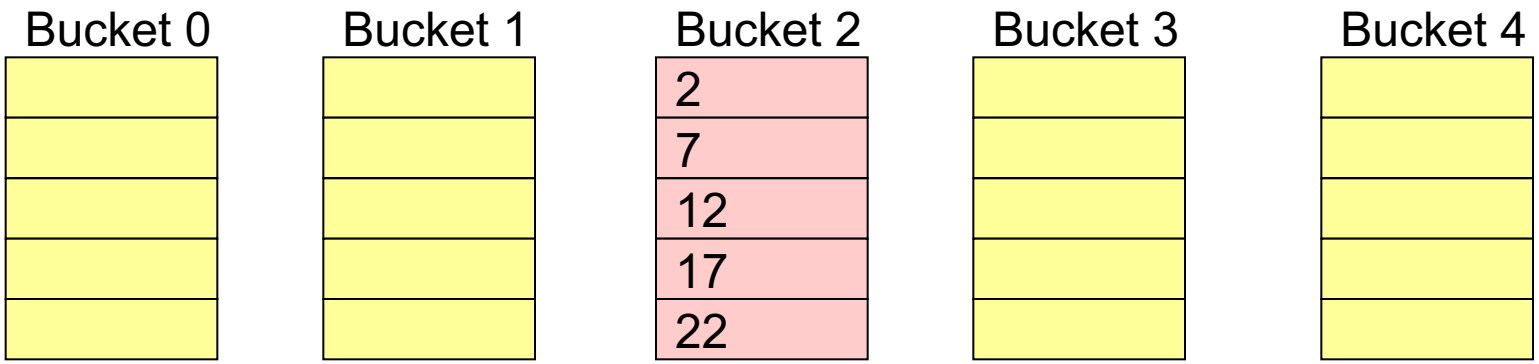
一个page可以装 128/8 = 16 个索引

Hash Index

- Hashing can be used not only for data file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices.

Hashing: Hash Functions

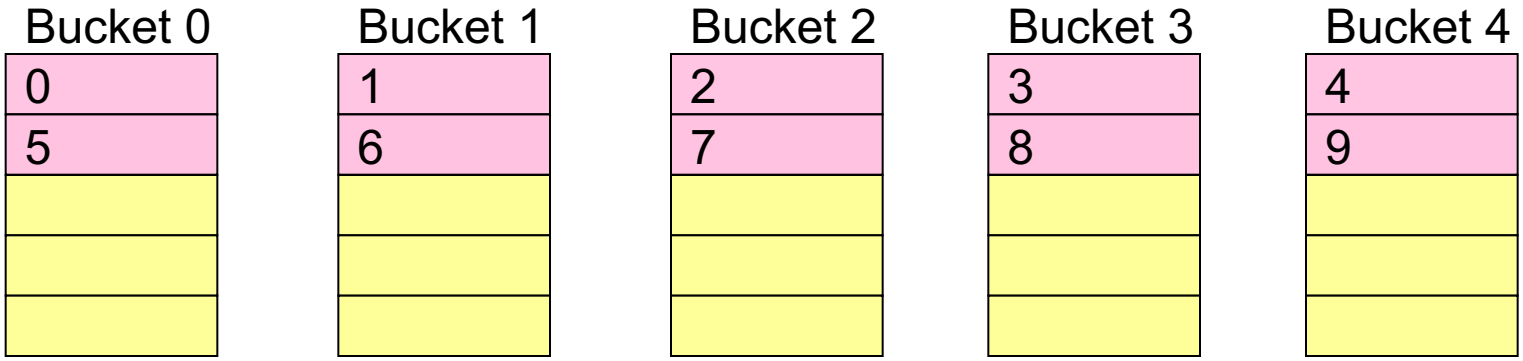
- **Worst case**
  - Hash function maps all search-key values to the same bucket
  - Access time proportional to the number of search-key values
- Data:2,7,12,17,22…
- Hash fuction F(x)=x mod 5



Hashing: Hash Functions

- **Ideal hash function**
  - Random
  - Each bucket receives same number of records

- Data:0,1,2,3,4,5,6,7,8,9…
- Hash function F(x)=x mod 5



Handling of Bucket Overflows

- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets*.

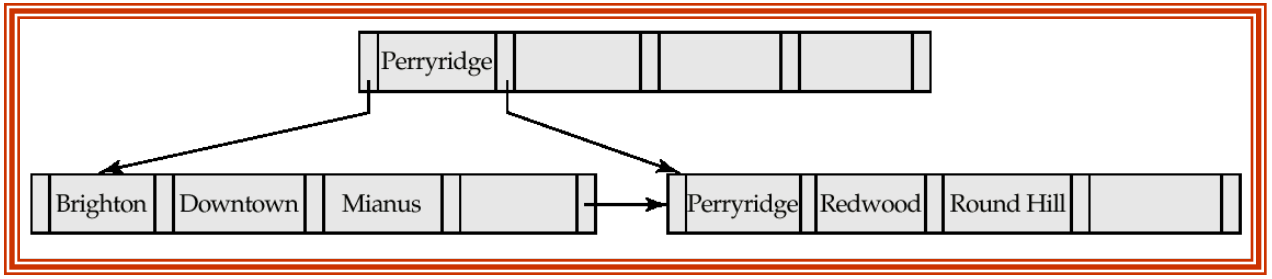
- Overflow chaining – 溢出桶在链表中链接在一起。长链会降低性能，因为查询必须读取链中的所有存储桶。

Part 2 B^+-tree and Hashing

- B^+-tree
- Dynamic Hashing

Review: B+-tree

- Balanced tree
- Use pointer, no need for sequential storage
- 树的深度是对数(log)级别
- Space overhead (空间额外开销，存储树索引)
- Insertion and deletion overhead. However, can be handled in logarithmic time. (插入删除的开销，由维护树结构导致，但可以控制在对数时间内)

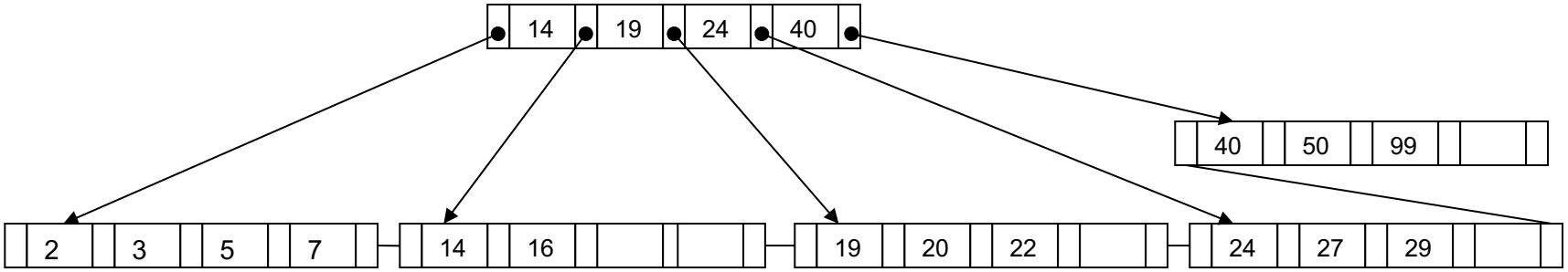


Review: Extendible Hashing

- Allow the number of buckets to be modified dynamically
- Insert: If bucket is full, split it
- If necessary, double the directory
  - Before inserting, local depth of bucket = global depth
  - Insert causes local depth > global depth
- Removal of data entry makes bucket empty
  - Merge "split image"
- Each directory element points to same bucket as its split image
  - Halve directory

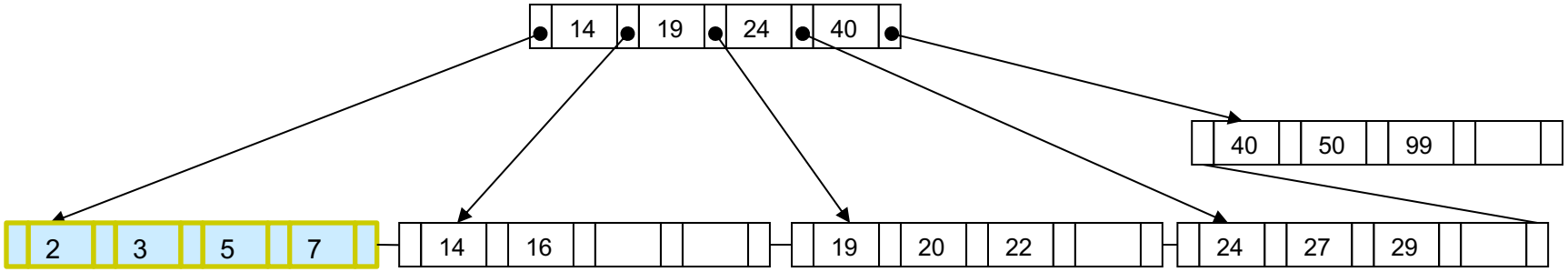
Exercise 2: B$^+$-tree

- Given a B$^+$-tree:

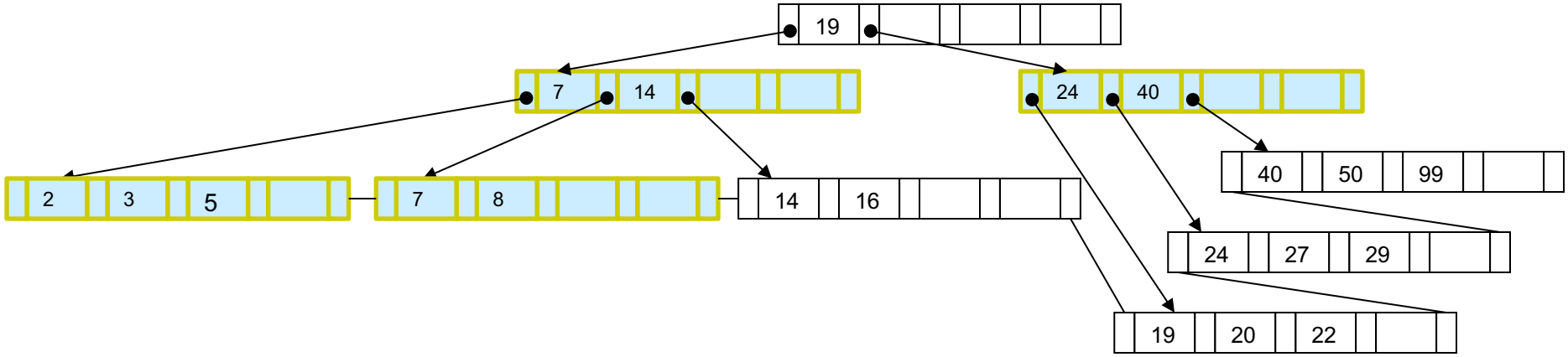


- Draw the tree after
  1. Inserting 8
  2. Deleting 2
  3. Deleting 3

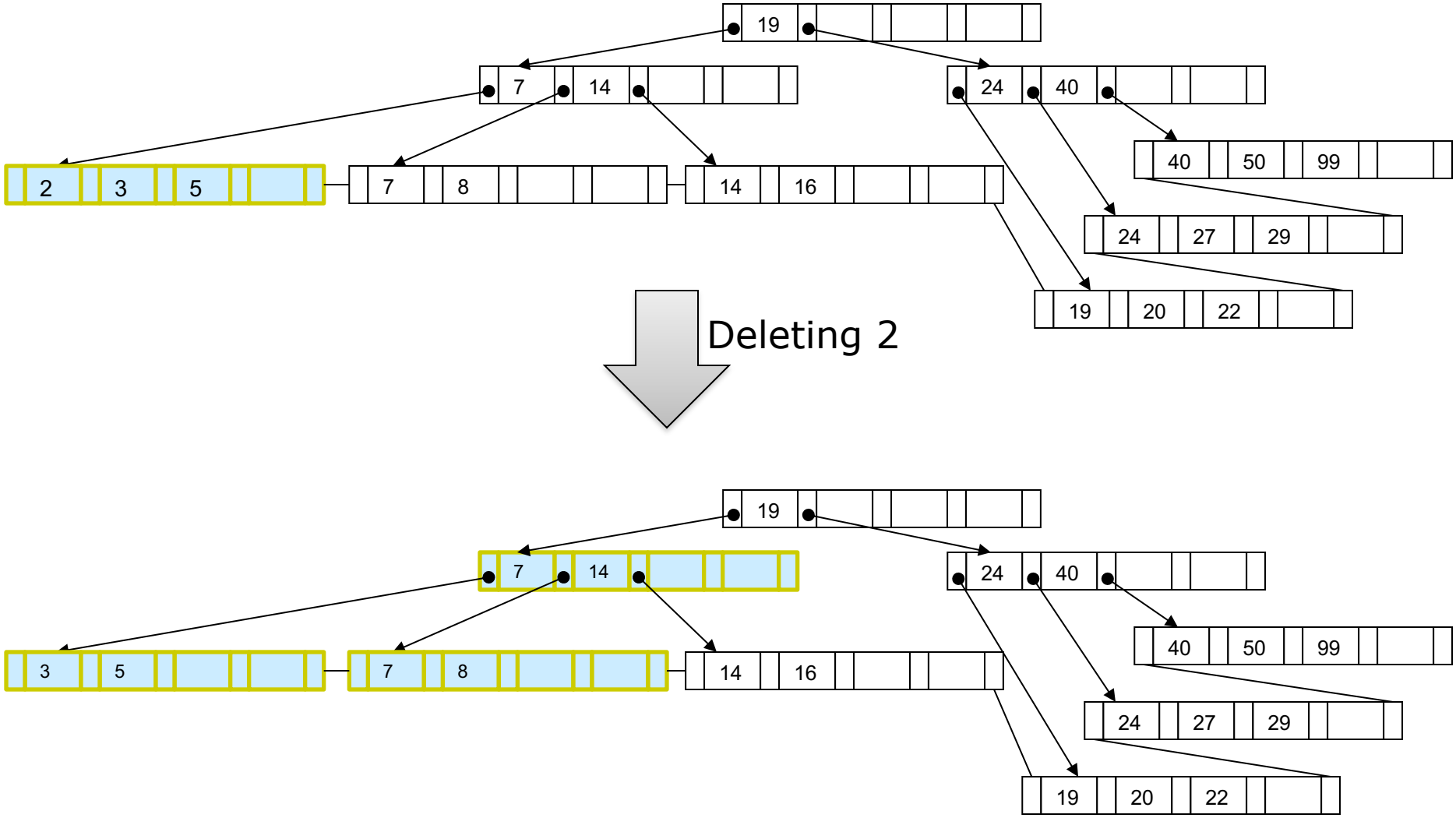
Exercise 2: B+-tree (1/3)



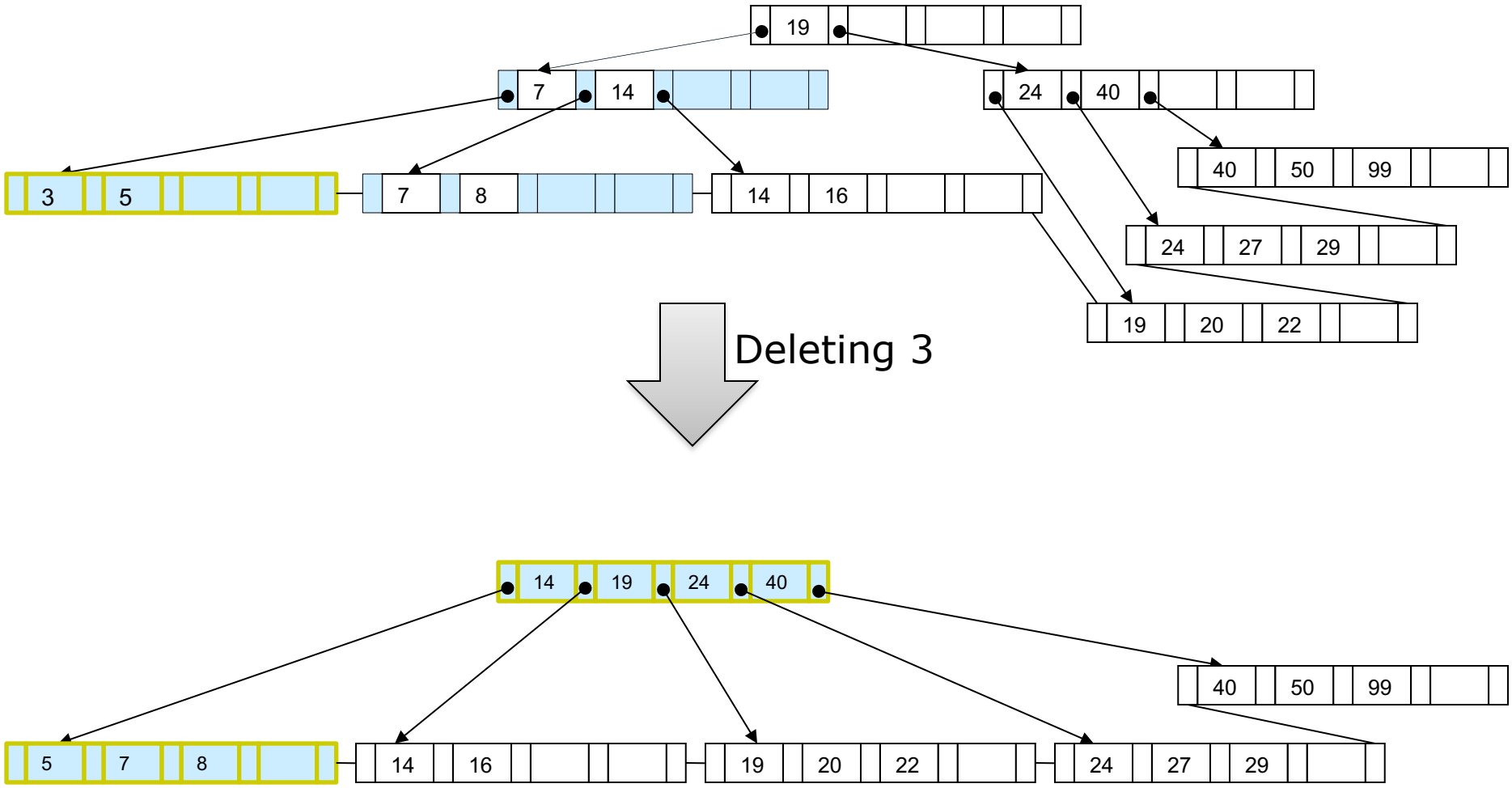
Inserting 8



Exercise 2: B+-tree (2/3)

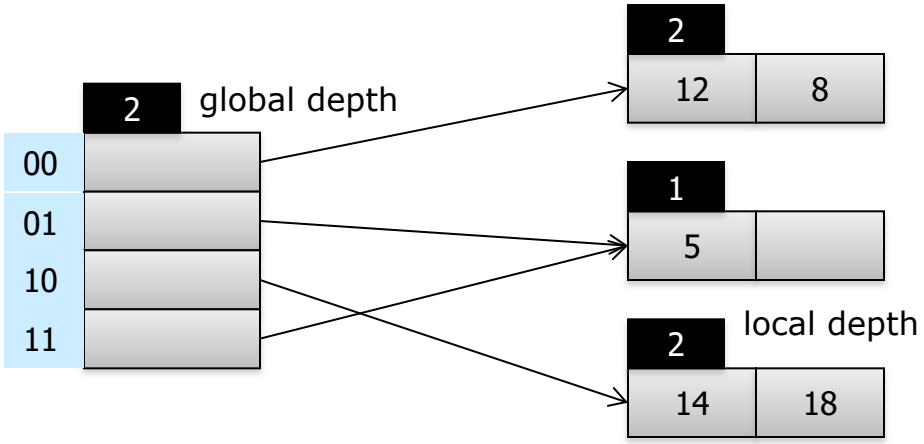


Exercise 2: B+-tree (3/3)



Exercise 3: Extendible Hashing

- Given the following directory and buckets.

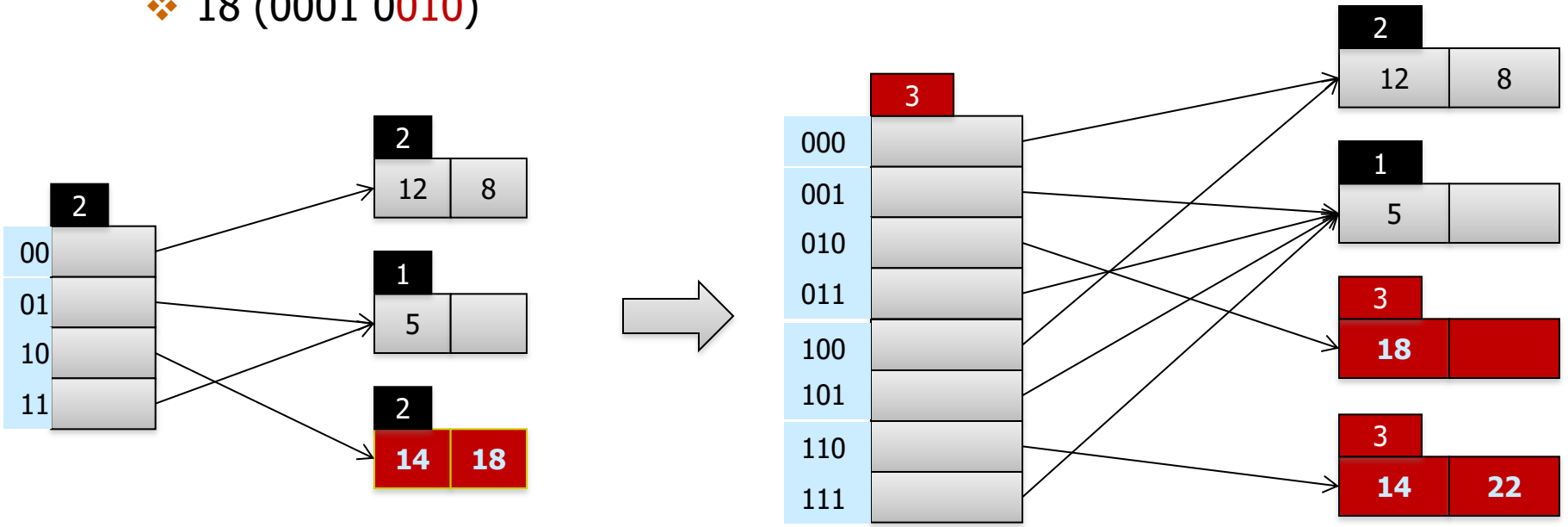


- Using extendible hashing, what will they be after:

1. Inserting 22
2. Inserting 3
3. Inserting 9

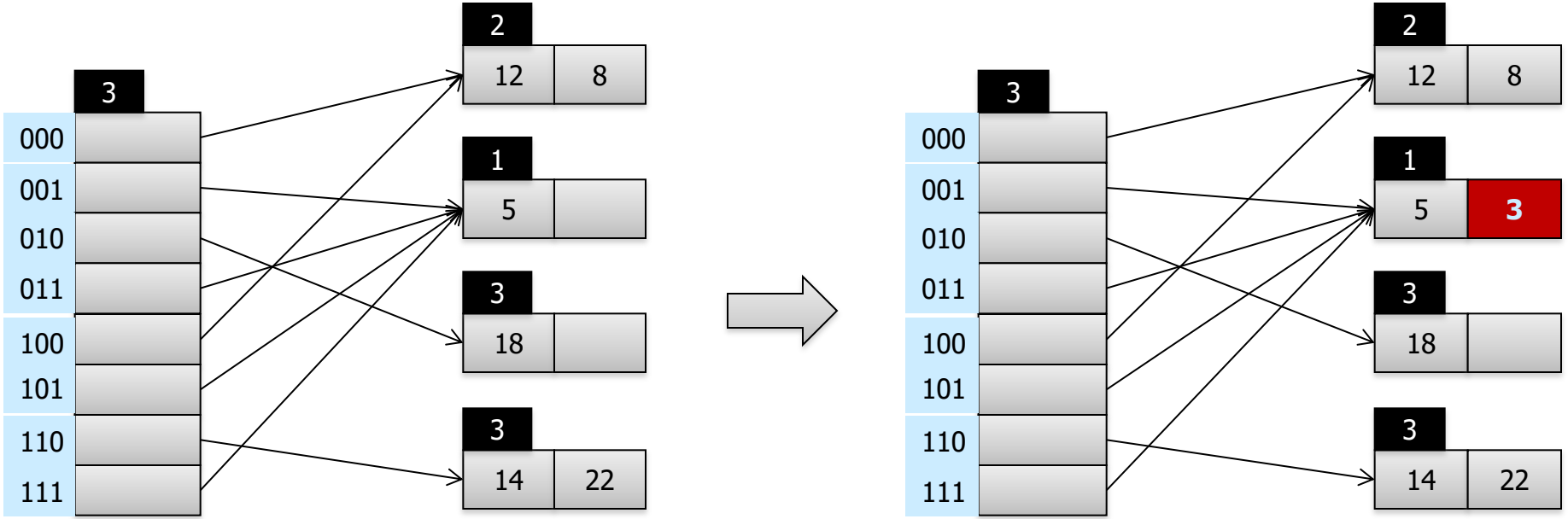
Exercise 3: Extendible Hashing (1/3)

- Inserting 22 (00010110)
  - 14 (0000 1110)
  - 18 (0001 0010)



Exercise 3: Extendible Hashing (2/3)

- Inserting 3 (0000 0011)



Exercise 3: Extendible Hashing (3/3)

- Inserting 9 (0000 1001)