

COMP 3311

DATABASE MANAGEMENT

SYSTEMS

TUTORIAL 6

INDEXING

INDEX: REVIEW

Clustering index

An index in which the records in the file are ordered (sequentially) by the search-key values. **Primary index** if search key=primary key.

Non-clustering index (secondary index)

An index in which the records in the file are not ordered by the search-key values.

Sparse Index: One index entry for many records; only applies to a clustering index.

Dense index: One index entry for each record in the file.

Single-level Index: Each index entry points to records.

Multilevel Index: An index entry points to another index entry except the bottom level points to records.

EXERCISE I

Assume that a school keeps the following file with the records of its students:

Student(studentId: 4 bytes; name: 10 bytes; deptId: 4 bytes)

where deptId is the department id to which a student belongs.

There exist 10,000 student records and 50 departments.

A page is 128 bytes; a pointer is 4 bytes.

The data file is sorted sequentially on studentId.

Record size: 18 bytes

***bf*_{Student}:** $\lfloor 128 \text{ bytes per page} / 18 \text{ bytes per record} \rfloor = 7 \text{ records/page}$

Pages needed: $\lceil 10,000 \text{ records} / 7 \text{ records per page} \rceil = 1429$

EXERCISE 1 (CONTD)

- a) Given the data file only, what is the page I/O cost of finding students in a particular department (e.g., CSE)?

Page I/O cost: 1429 page I/Os **Why?**

Explanation: The records are sorted on studentId, instead of deptId. Thus, the only way to answer the query is a file scan.

- b) How can we reduce the cost of this search?

Build an **ordered, single-level index** on deptId.

Index entry size: 8 bytes **Why?**

Explanation: Each index entry is of the form (deptId, *pointer*) and requires 4 bytes for deptId and 4 bytes for the pointer.

$bf_{deptIdindex} = \lfloor 128 \text{ bytes per page} / 8 \text{ bytes per record} \rfloor = 16$

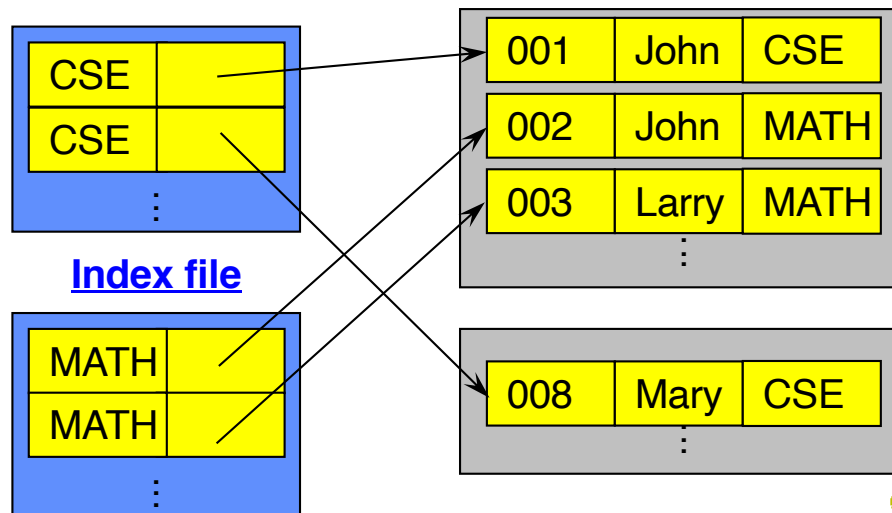
EXERCISE 1 (CONT'D)

For simplicity, we assume each department has exactly (10,000 students / 50 departments) = 200 students. We also want *an index page to contain index entries only for students in the same department*.

i. Build an ordered, single-level index.

Index pages: $\lceil 200 \text{ students per dept} / 16 \text{ index entries per page} \rceil * 50$
 departments = 650 (13 index pages per department)

Index page I/O cost: $\lceil \log_2 650 \rceil$ (to find the first index page) +
 12 additional index pages = 22 page I/Os



Data file

Pages needed:

$\lceil 10,000 \text{ records} / 7 \text{ records per page} \rceil = 1429$

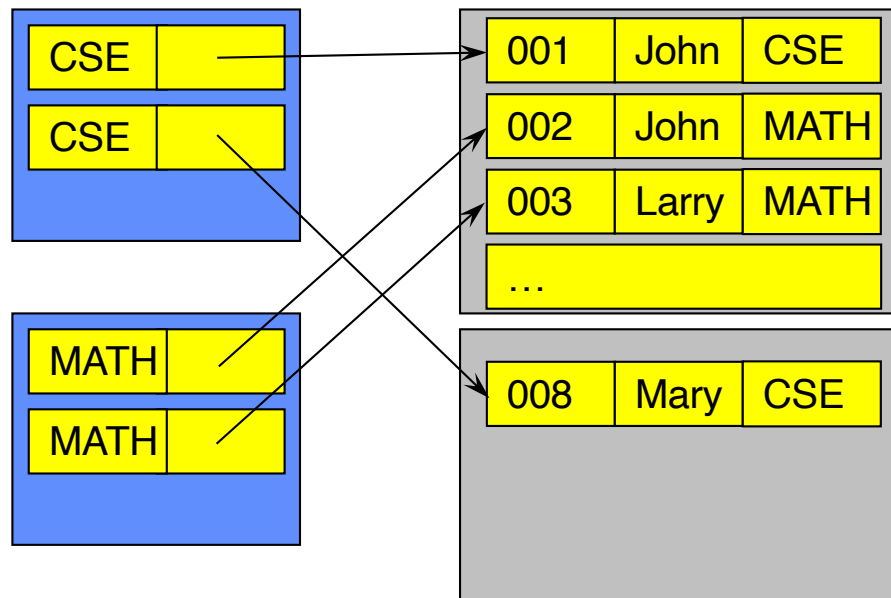
Data file page I/O cost: 200 page I/Os **Why?**

Explanation: The file is not ordered by deptId. Thus, for each student we may need to access a page.

EXERCISE I (CONTD)

ii. Build an ordered, single-level index.

The index constructed on deptId as shown below is:



Primary index or secondary index?
Secondary index.

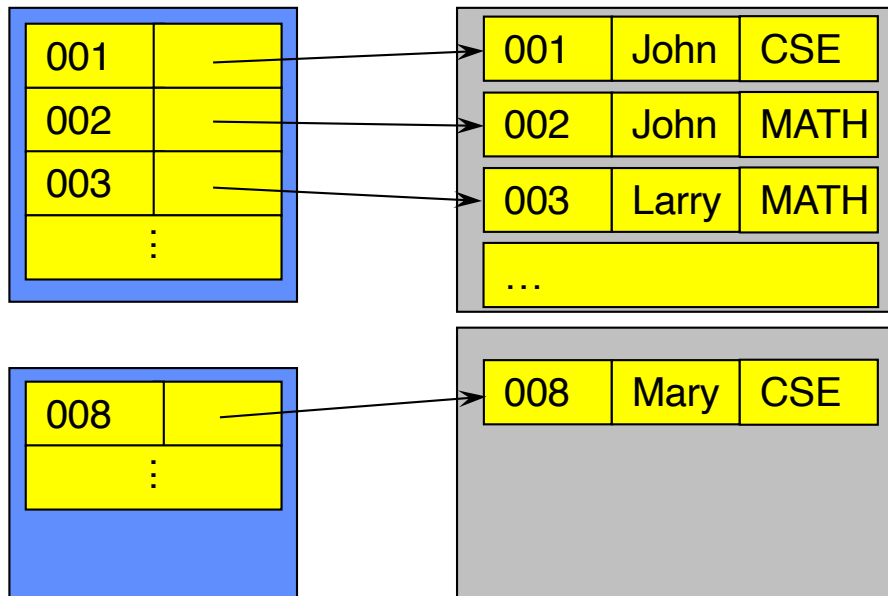
Dense index or sparse index?
Dense index. Why?

Explanation: A secondary index must be a dense index.

EXERCISE I (CONTD)

iii. Build an ordered, single-level index.

An index constructed on studentId as shown below is:



Primary index or secondary index?

Primary index.

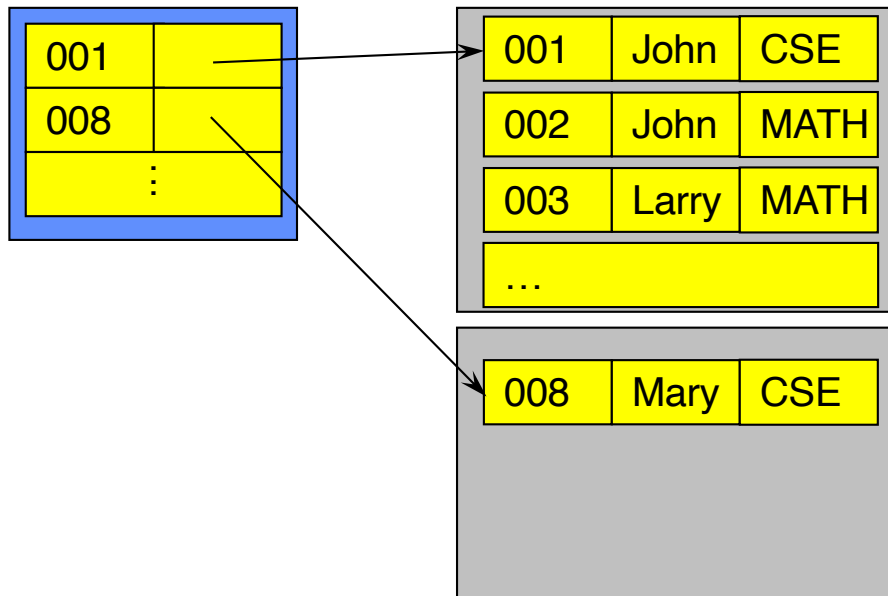
Dense index or sparse index?

Dense index.

EXERCISE I (CONTD)

iv. Build an ordered, single-level index.

An index constructed on studentId as shown below is:



Primary index or secondary index?

Primary index.

Dense index or sparse index?

Sparse index.

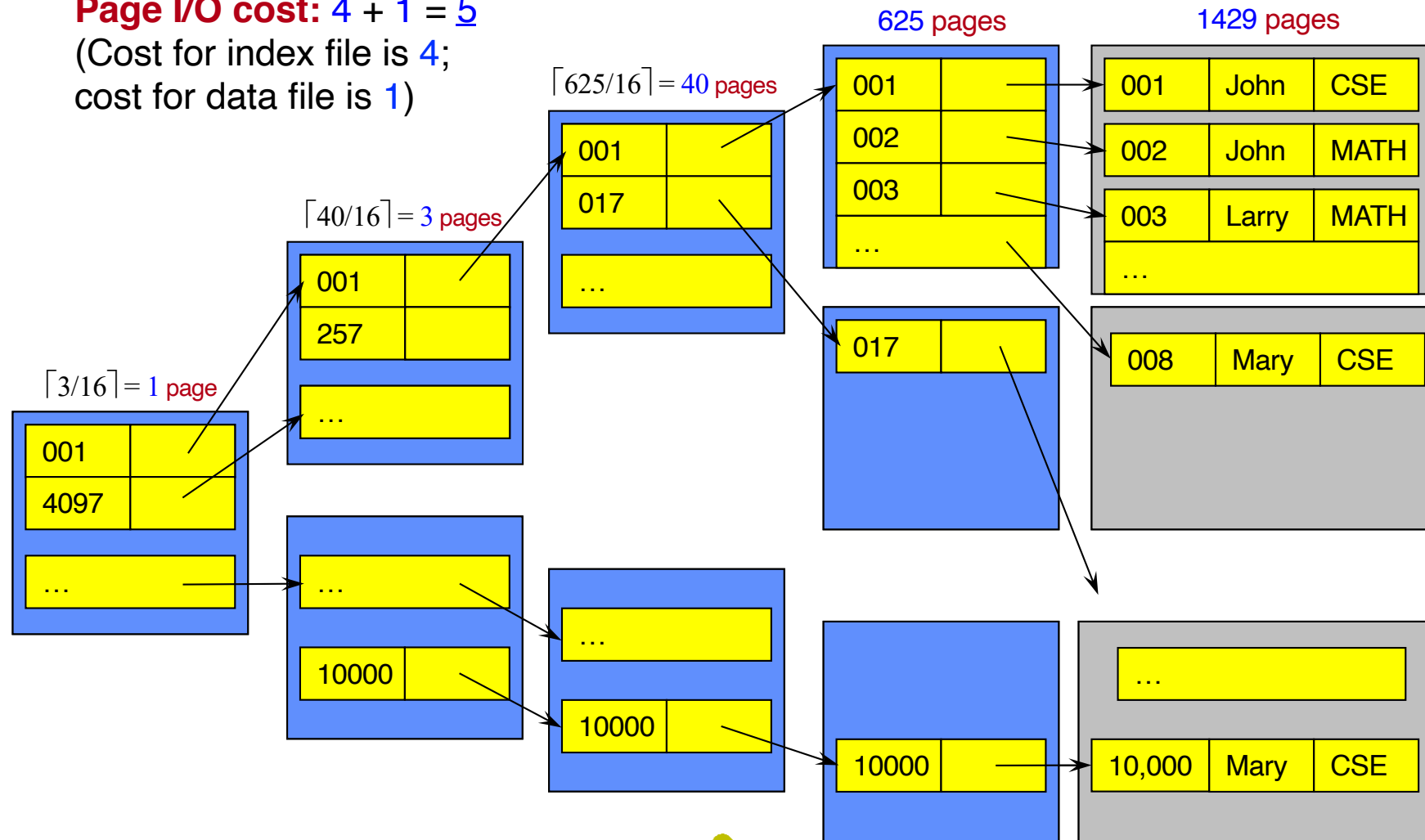
EXERCISE 1 (CONTD)

Student records: 10,000
 Departments: 50
 Page size: 128 bytes
 Pointer size: 4 bytes
 Record size: 18 bytes
 $bf_{Student}$: 7 records/page
 $bf_{deptIdindex}$: 16 entries/page
 Pages: 1429

- c) What is the page I/O cost to look up a particular student using this index on studentId?

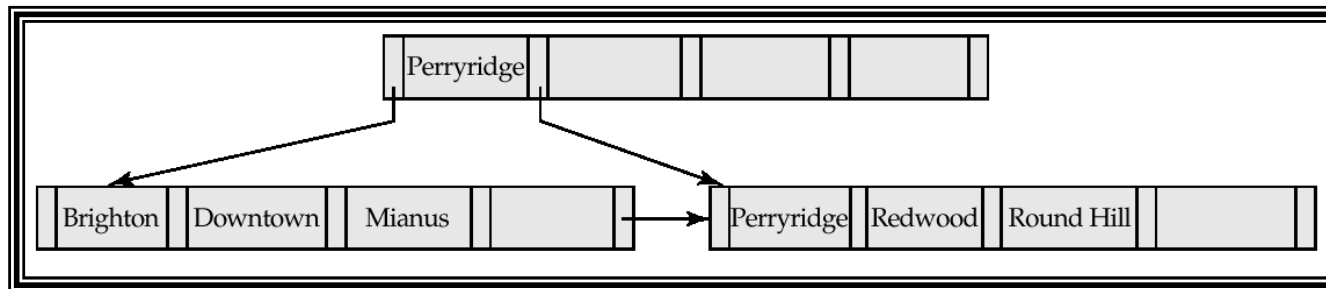
Page I/O cost: $4 + 1 = 5$

(Cost for index file is 4;
 cost for data file is 1)



REVIEW: B⁺-TREE INDEX

- A balanced tree.
- Use pointers to access records; no need for sequential storage.
- The height of the tree is in logarithmic order.
- **Space overhead** – additional space is needed for the index.
- **Insertion and deletion overhead.** However, can be handled in logarithmic time.



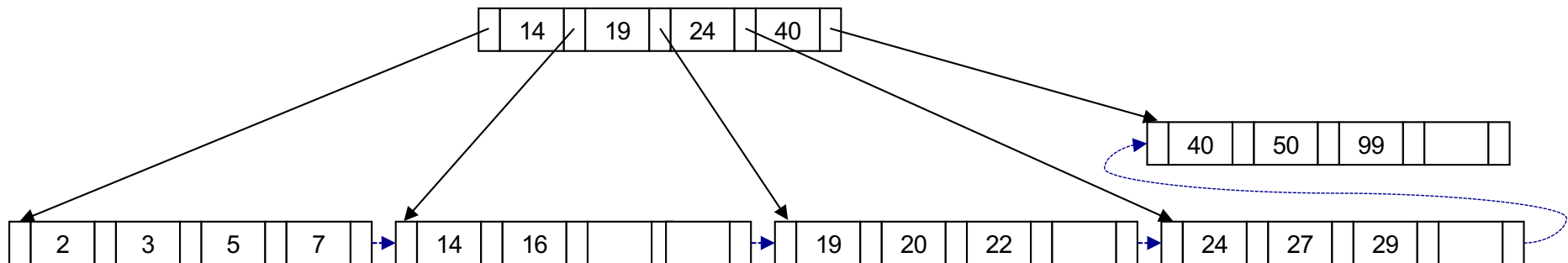
REVIEW: EXTENDABLE HASHING

- Allows the number of pages (buckets) to be modified **dynamically**.
- If **insert** of record is to a full page, **split it**.
 - If necessary, **double the directory**.
 - Before inserting, **local depth** of page = **global depth**.
 - Insert causes **local depth** > **global depth**.
- If **removal** of data entry makes page empty, **merge with its “split image”**.
 - Each directory entry points to the same page as its split image.
 - If necessary, **halve the directory**.

EXERCISE 2

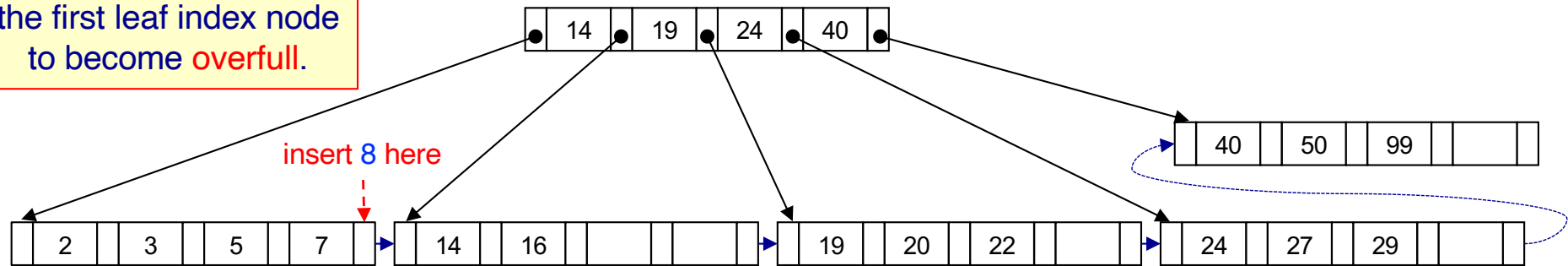
For the B⁺-tree shown below, show the tree that would result after *successively* applying the following operations in order.

- i. insert 8
- ii. delete 2
- iii. delete 3



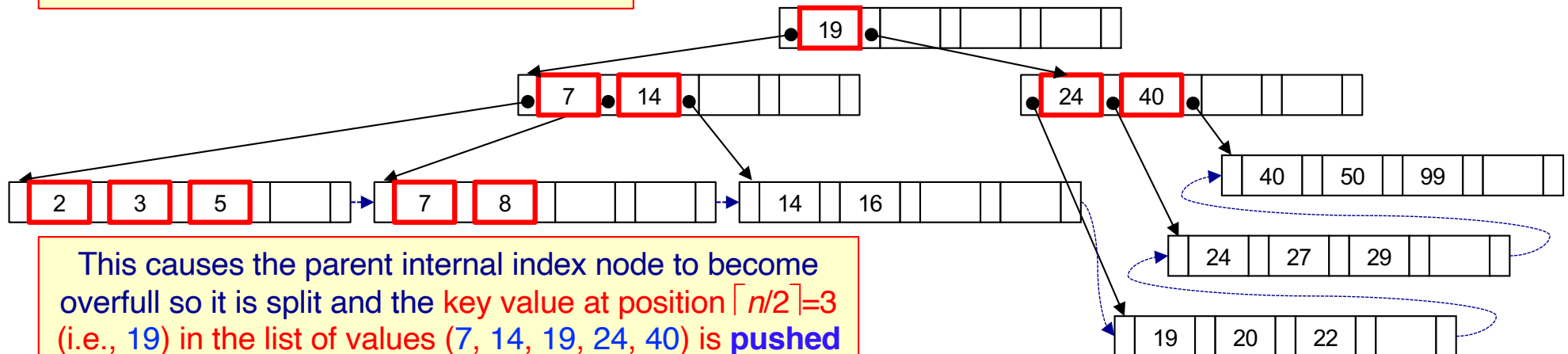
EXERCISE 2 (CONT'D)

The insertion causes the first leaf index node to become **overflow**.



The leaf index node is split and the key value at position $\lceil n/2 \rceil + 1 = 4$ (i.e., 7) of the list of values (2, 3, 5, 7, 8) is **copied up** to the parent internal index node.

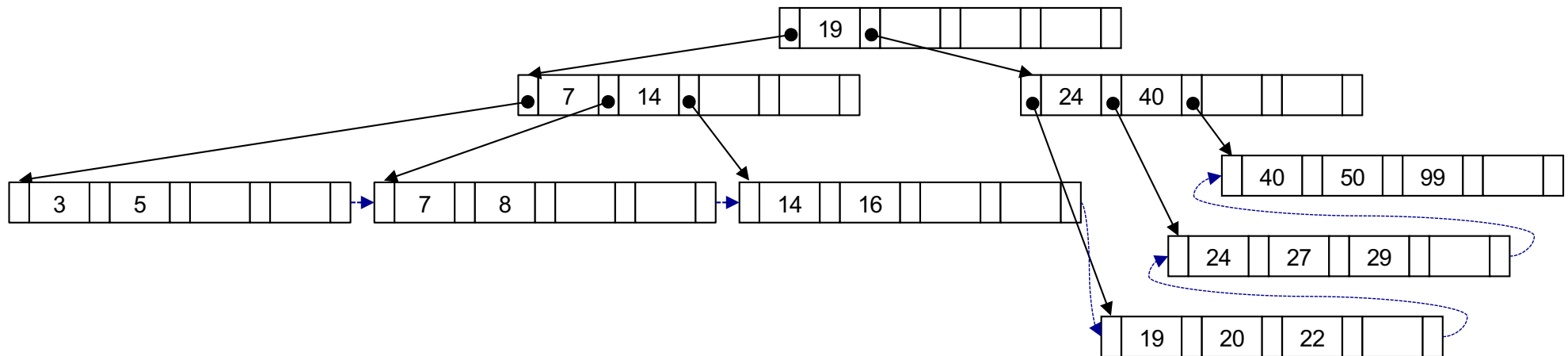
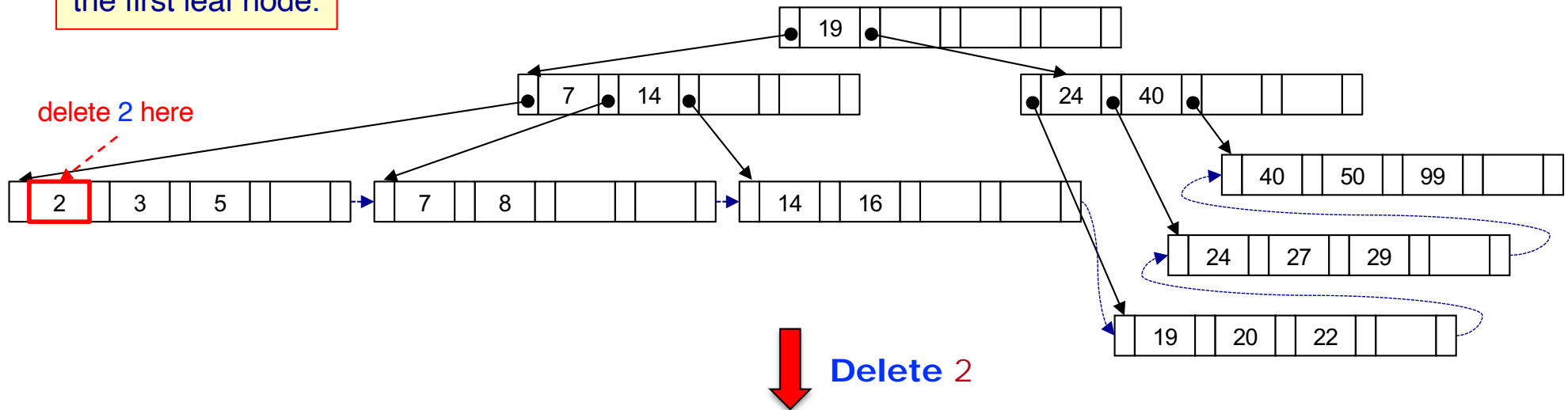
Insert 8



This causes the parent internal index node to become overfull so it is split and the key value at position $\lceil n/2 \rceil = 3$ (i.e., 19) in the list of values (7, 14, 19, 24, 40) is **pushed up** to the new parent (root) internal index node.

EXERCISE 2 (CONT'D)

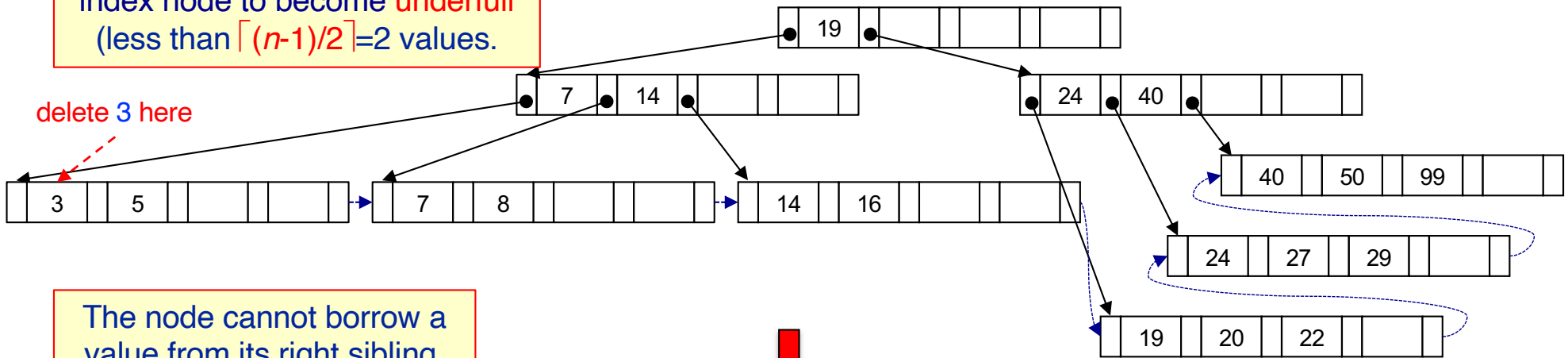
2 is deleted from
the first leaf node.



EXERCISE 2 (CONT'D)

The deletion causes the first leaf index node to become **underfull** (less than $\lceil (n-1)/2 \rceil = 2$ values).

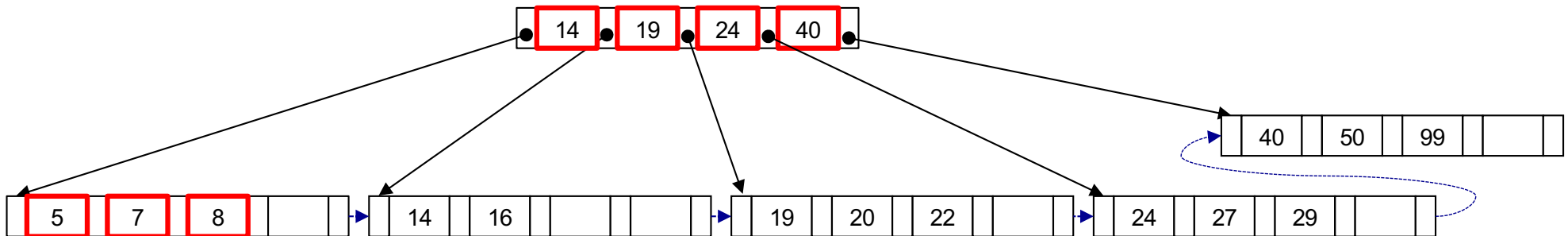
delete 3 here



The node cannot borrow a value from its right sibling, so it **must be merged** with it.



Delete 3



This causes the parent internal index node to now have only 2 pointers, but it needs 3. Therefore, it **is merged** with its sibling and the index values adjusted.

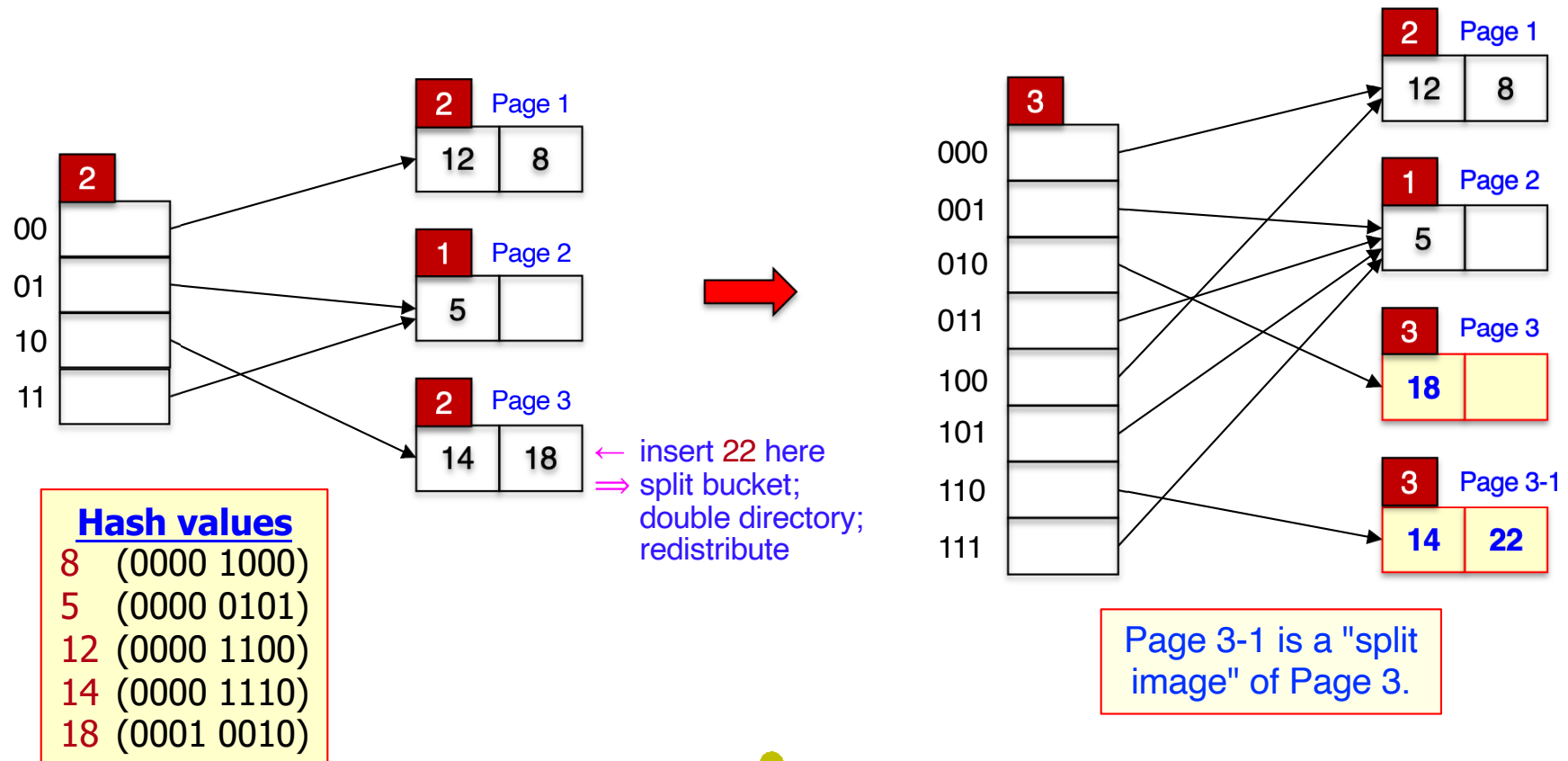
The merge of the internal index nodes causes the root index node to now have only 1 pointer, so it **is merged** with the internal index node below it and the **tree shrinks one level**.



EXERCISE 3

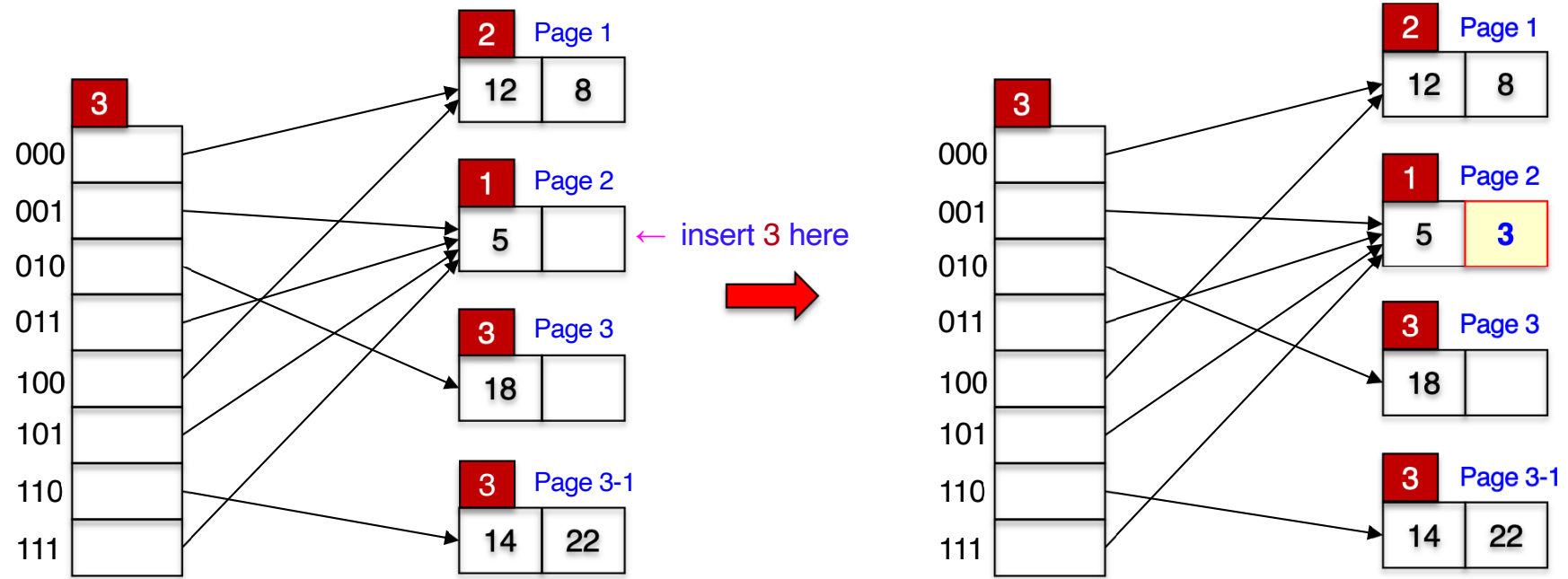
For the directory and pages shown below, use extendable hashing and show what the directory and pages would be after the following operations.

Insert 22 (0001 0110)



EXERCISE 3 (CONTD)

Insert **3** (0000 0011)



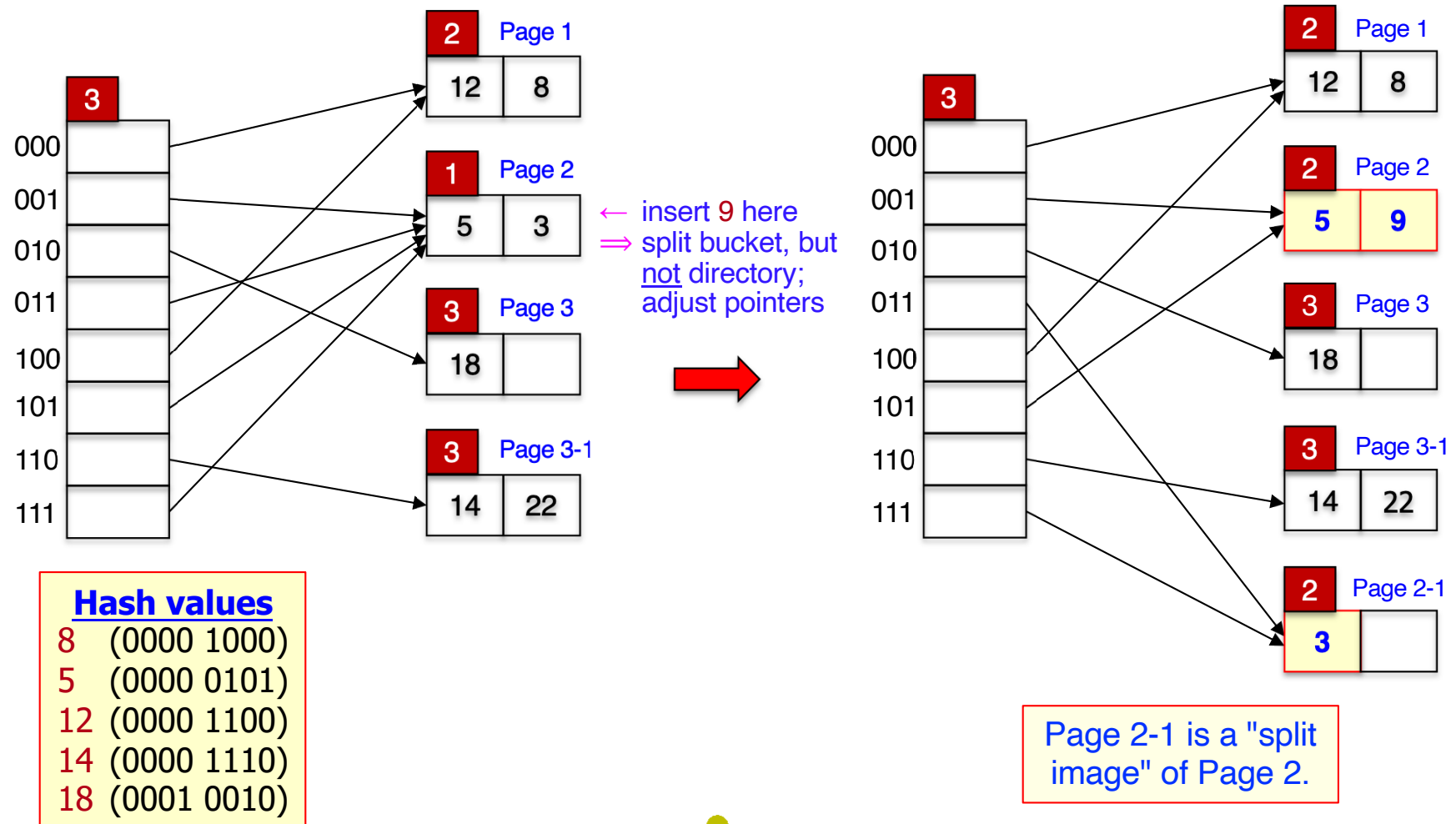
Hash values

8 (0000 1000)
 5 (0000 0101)
 12 (0000 1100)
 14 (0000 1110)
 18 (0001 0010)



EXERCISE 3 (CONTD)

Insert 9 (0000 1001)



EXERCISE 3 (CONTD)

delete 18 (0001 0010)

