Database Applications (15-415)

DBMS Internals- Part IV
Lecture 14, March 10, 2015

Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

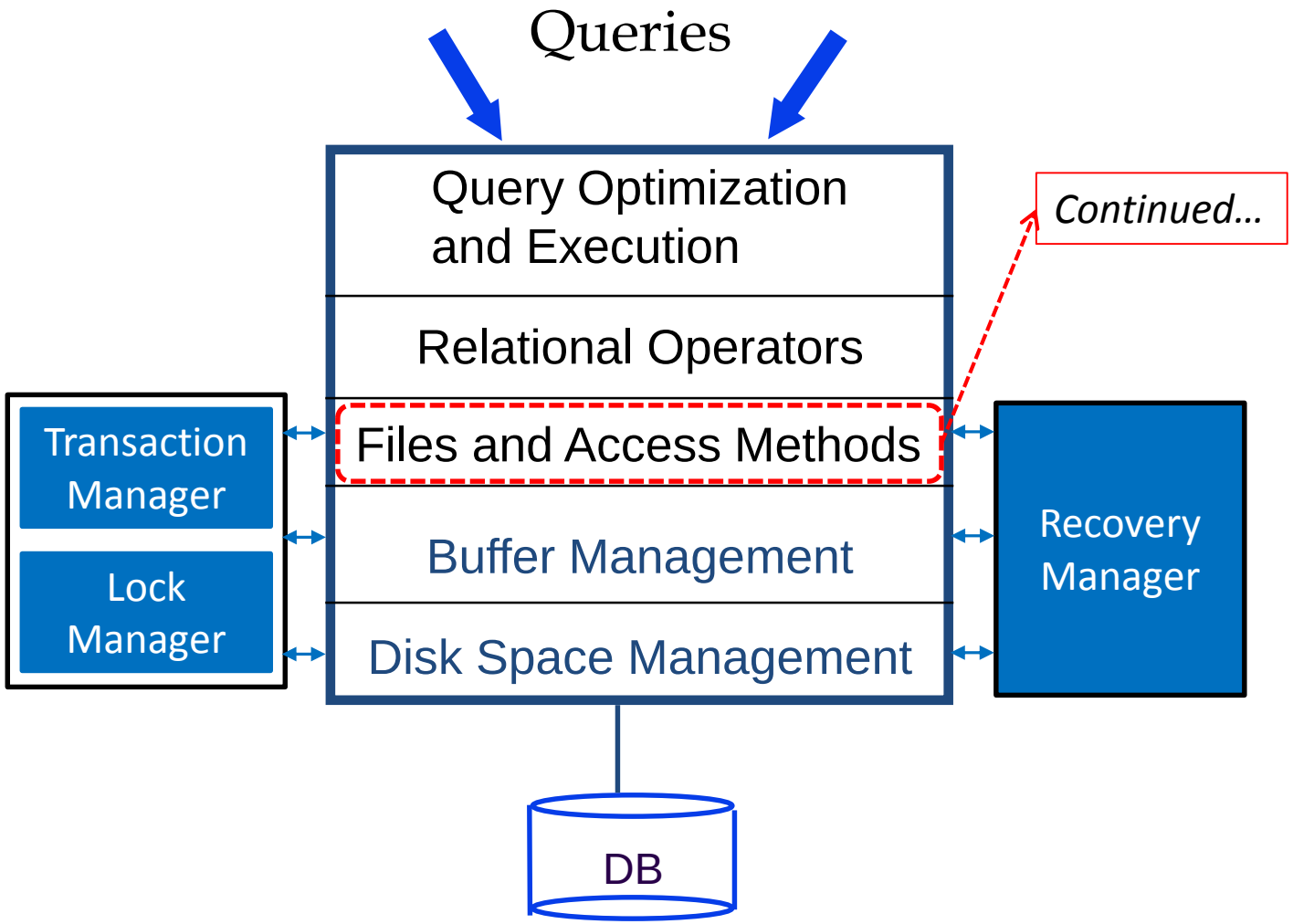
Today…

- Last Two Sessions:
  - DBMS Internals- Part III
    - Tree-based indexes: ISAM and B+ trees
  - Data Warehousing/ Data Mining (by Prof. Christos Faloutsos)

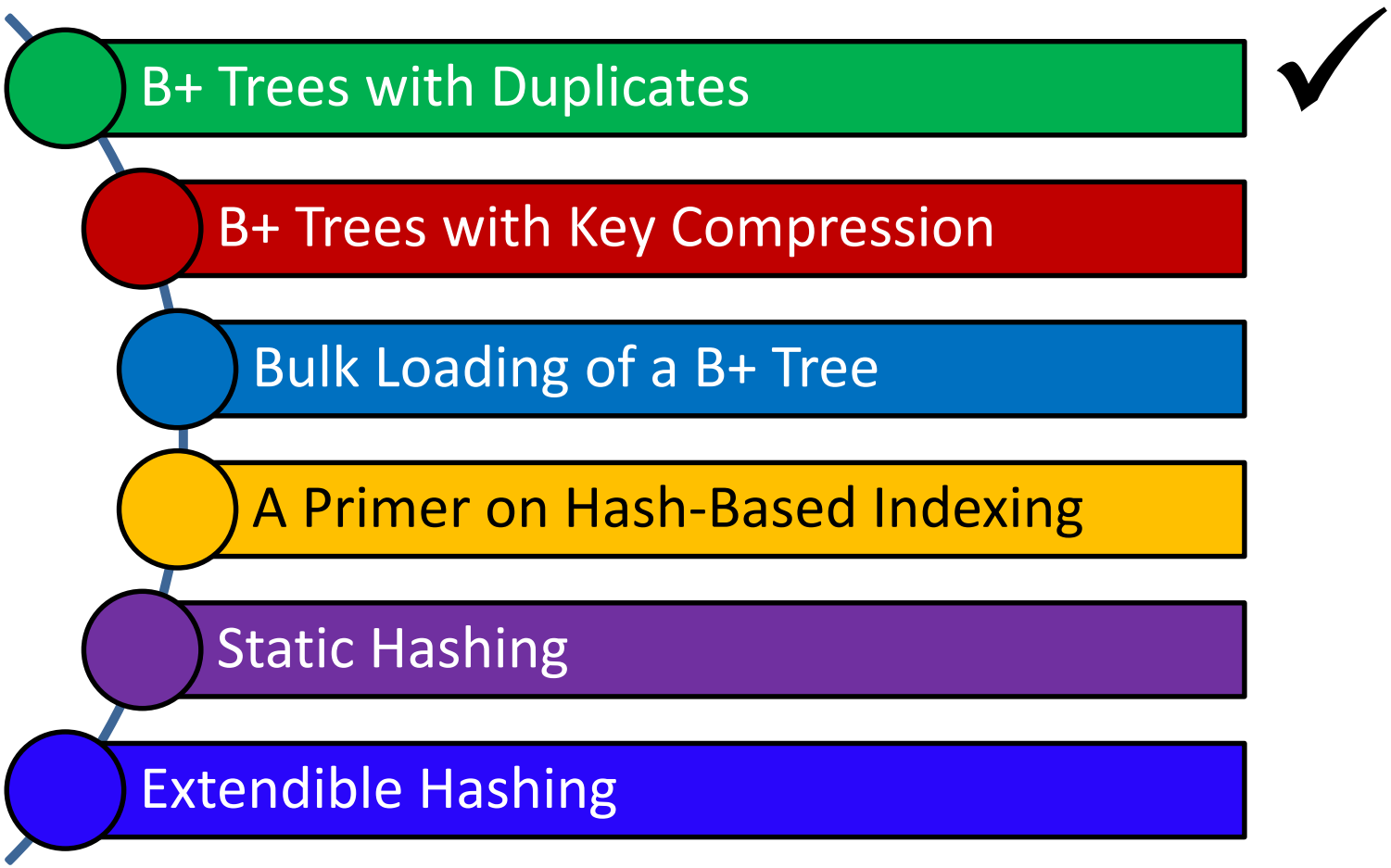
- Today's Session:
  - DBMS Internals- Part IV
    - Tree-based (B+ tree- cont'd) and Hash-based indexes

- Announcements:
  - P1 grades are out
  - Midterm grades are out
  - Mid-course grades are out
  - PS3 will be posted online by tomorrow

DBMS Layers


Queries
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
Continued…
DB

Outline



B+ Trees With Duplicates

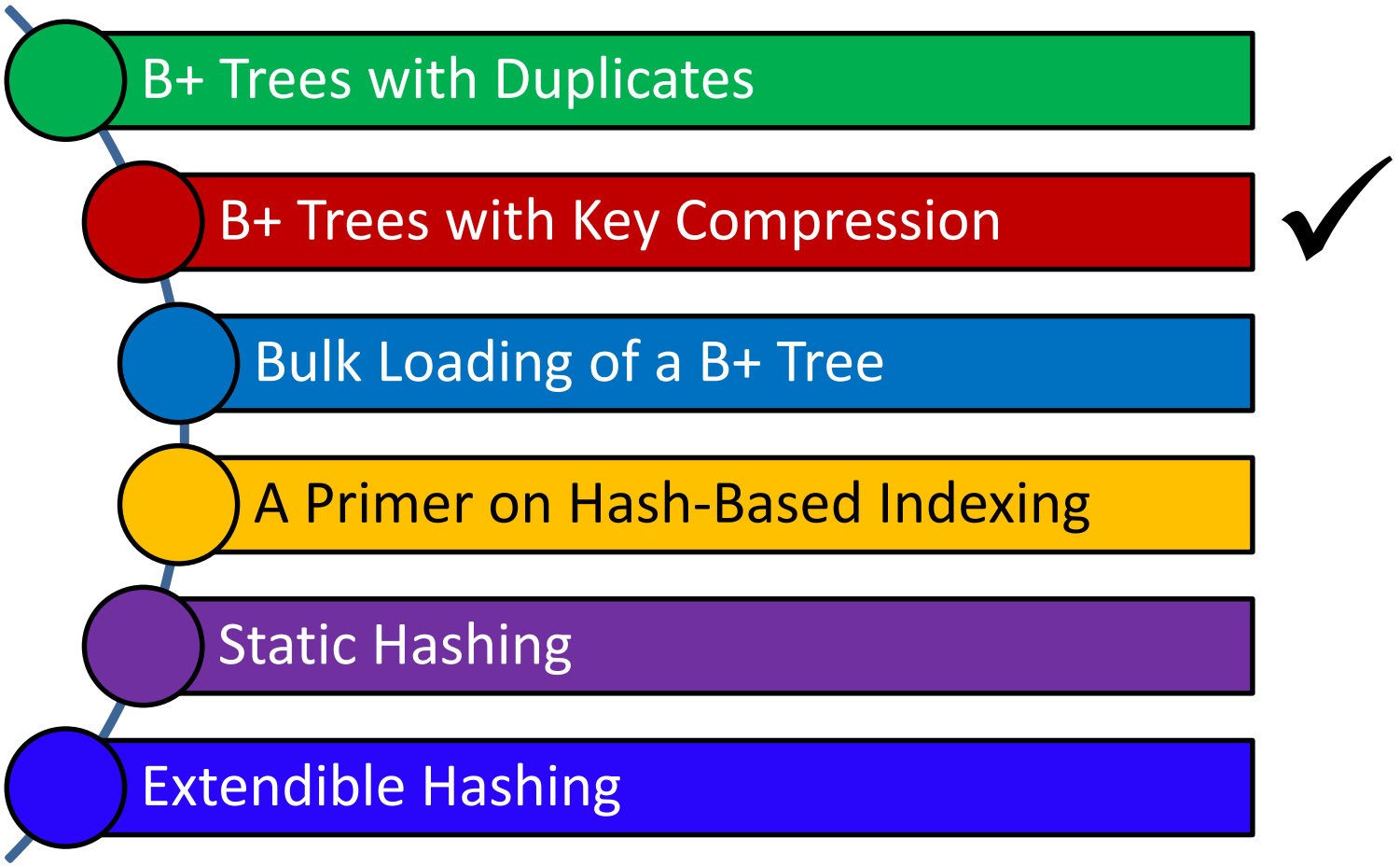
- Thus far, we only discussed unique indices (no *duplicate keys- i.e., several data entries with the same key value*)

- How can we handle duplicate keys?
  1. Use overflow pages to keep all entries of a given key value on a single leaf page (natural for ISAM)

  2. Treat duplicates like any other entries
     - Several leaf pages will contain entries of a given key value
     - How to search/delete?

  3. Make the *rid* value part of the search key

Outline



The Height of a B+ Tree

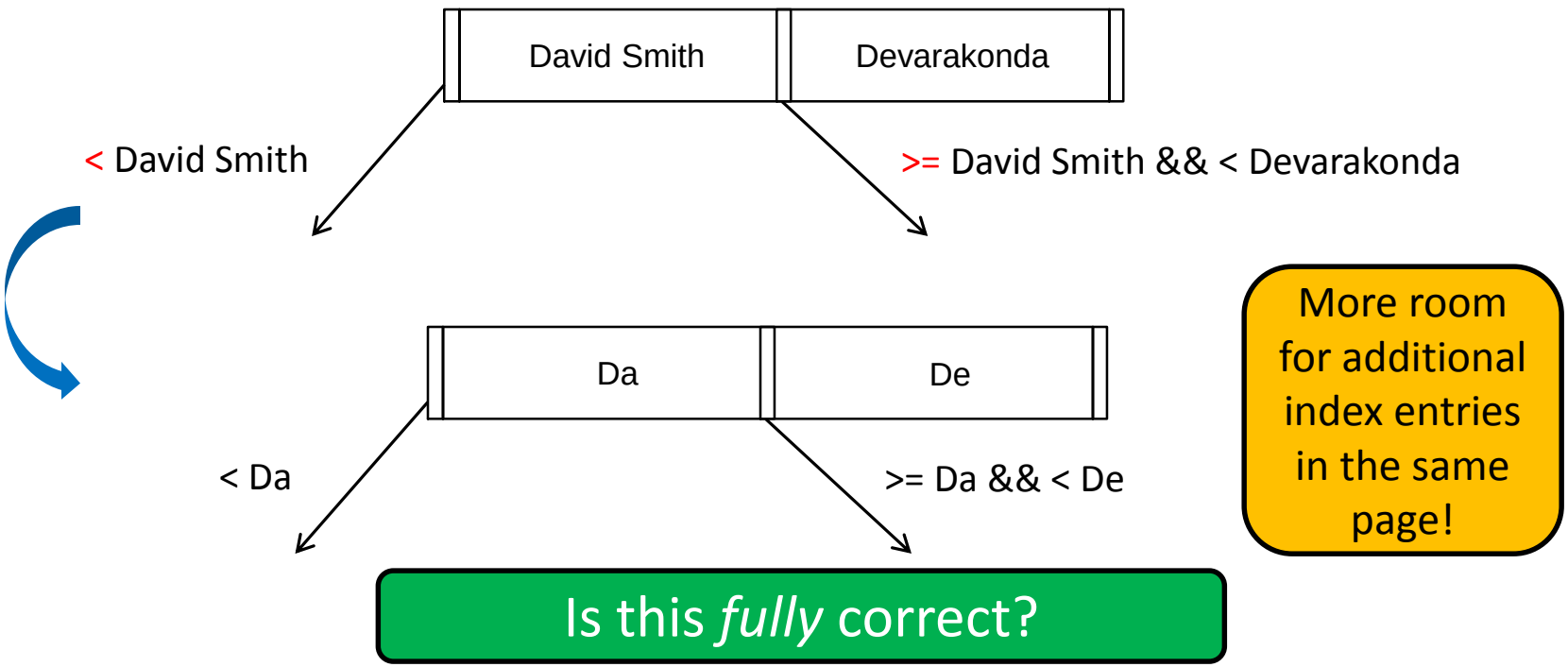
- What are the factors that define the height of a B+ tree?
  - Number of data entries
  - The order of occupancy
- The order of occupancy dictates the *fan-out* of the tree
- The height of the tree is proportional to $log_{fan\text{-}out}$ *(# of DEs)*
- What is the number of disk I/Os to retrieve a data entry?
  - $log_{fan\text{-}out}$ *(# of DEs)*
- How to minimize the height?
  - Maximize the fan-out

Towards Maximizing the Fan-Out?

- What does an index entry contain?
  - A search key
  - A page pointer
- Hence, the size of an index entry depends primarily on the size of the search key value!
- What if the search key values are very long?
  - Not many index entries will fit on a page
  - Fan-out will be small
  - The height of the tree will be large

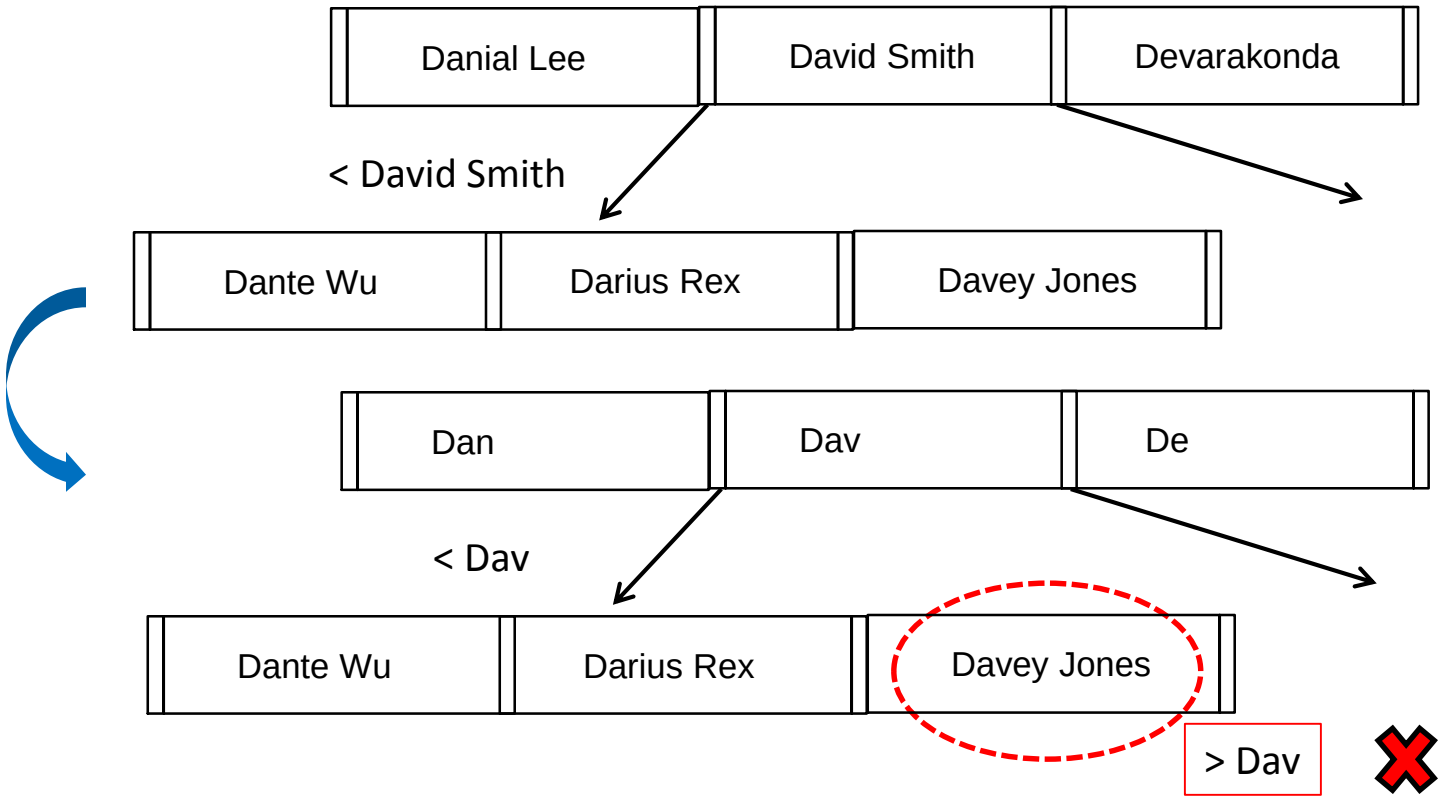
Key Compression: A Way to Maximize the Fan-Out

- How can we reduce the size of search key values?
  - Apply key compression, especially that keys are only used to direct traffic to the appropriate leaves



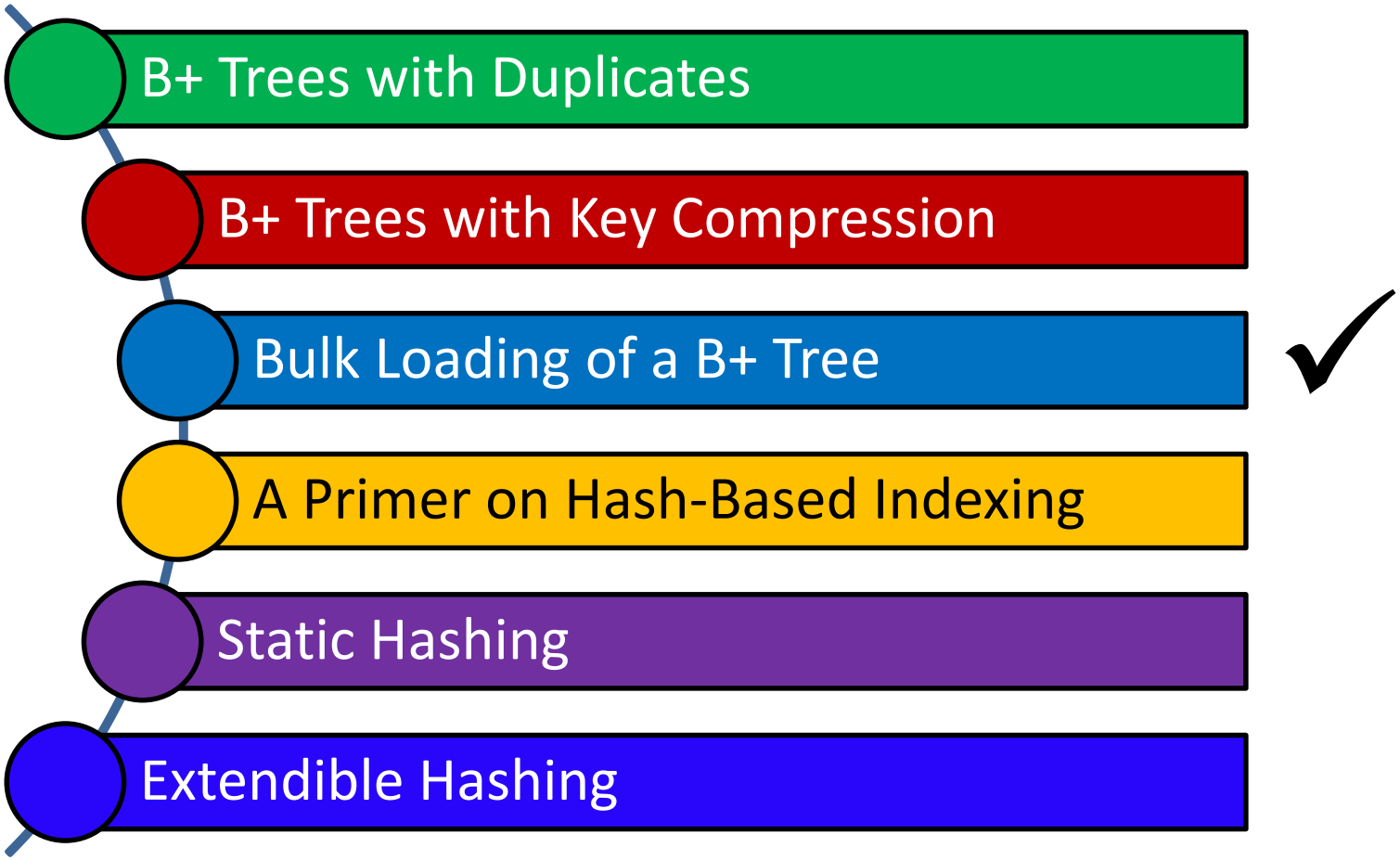
Key Compression: A Way to Maximize the Fan-Out (Cont'd)

- What about the following example?



To ensure correct semantics, the **largest key value in the left sub-tree** and the **smallest key value in the right sub-tree** must be examined!

Outline



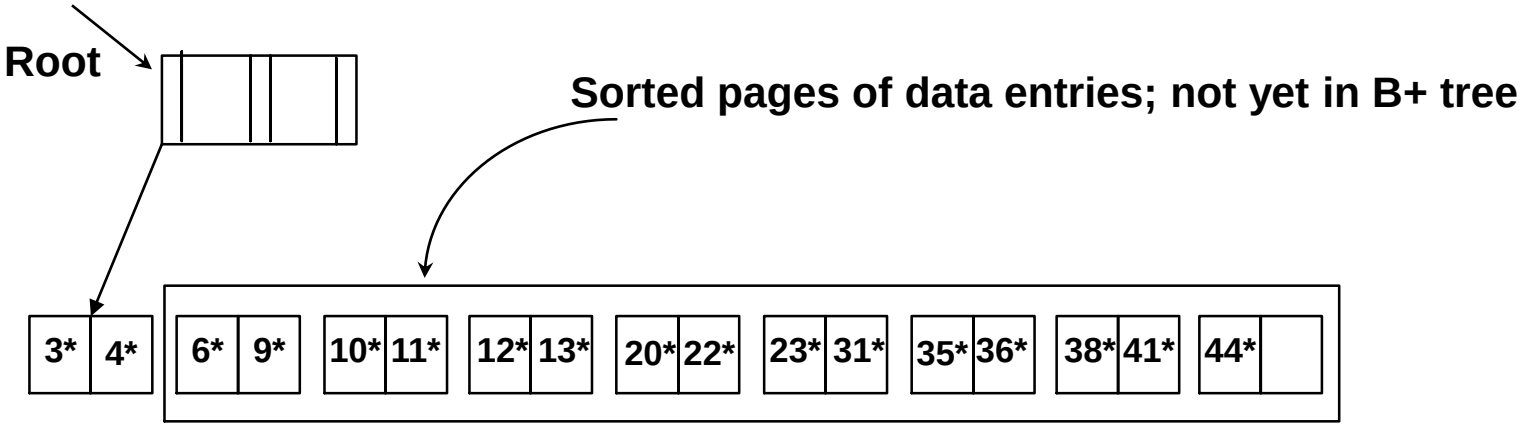
B+ Tree: Bulk Loading

- Assume a collection of data records with an *existing* B+ tree index on it
  - How to add a new record to it?
    - Use the B+ tree insert() function

- What if we have a collection of data records for which we want to create a B+ tree index? (i.e., we want to *bulk load* the B+ tree)
  - Starting with an empty tree and using the insert() function for each data record, *one at a time*, is expensive!
    - This is because for each entry we would require starting again from the root and going down to the appropriate leaf page

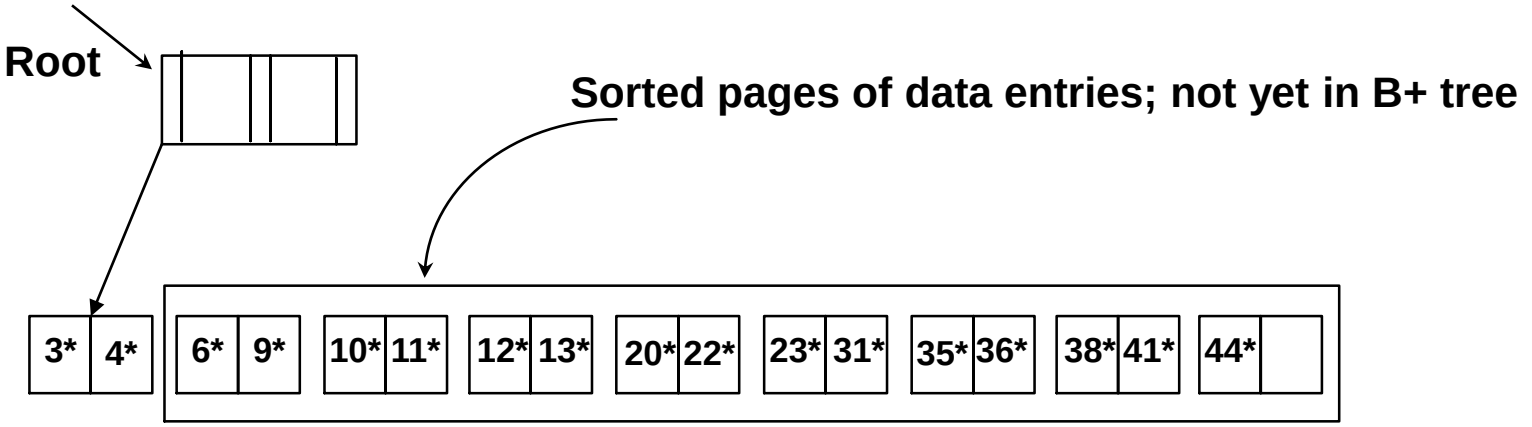
B+ Tree: Bulk Loading

- What to do?
  - *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page



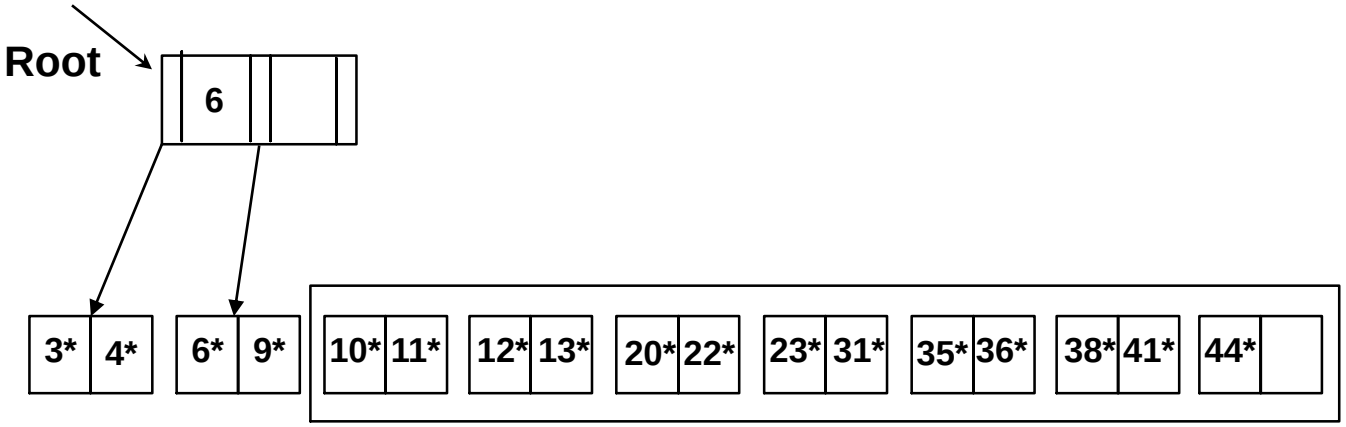
B+ Tree: Bulk Loading

- What to do?
  - Add one entry to the root page for each subsequent page of the sorted data entries *(i.e., <lowest key value on page, pointer to the page>)*



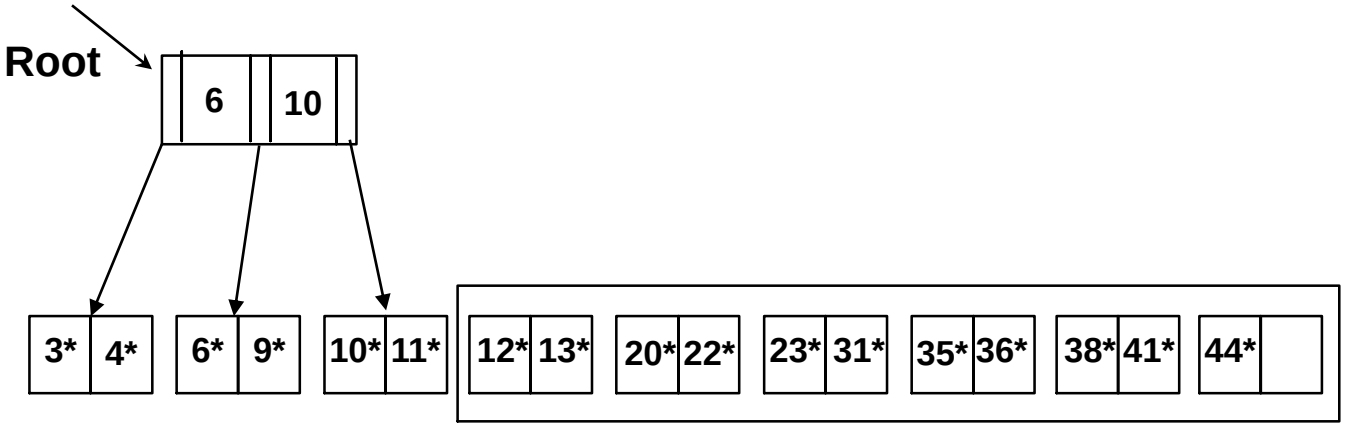
B+ Tree: Bulk Loading

- What to do?
  - Add one entry to the root page for each subsequent page of the sorted data entries *(i.e., <lowest key value on page, pointer to the page>)*



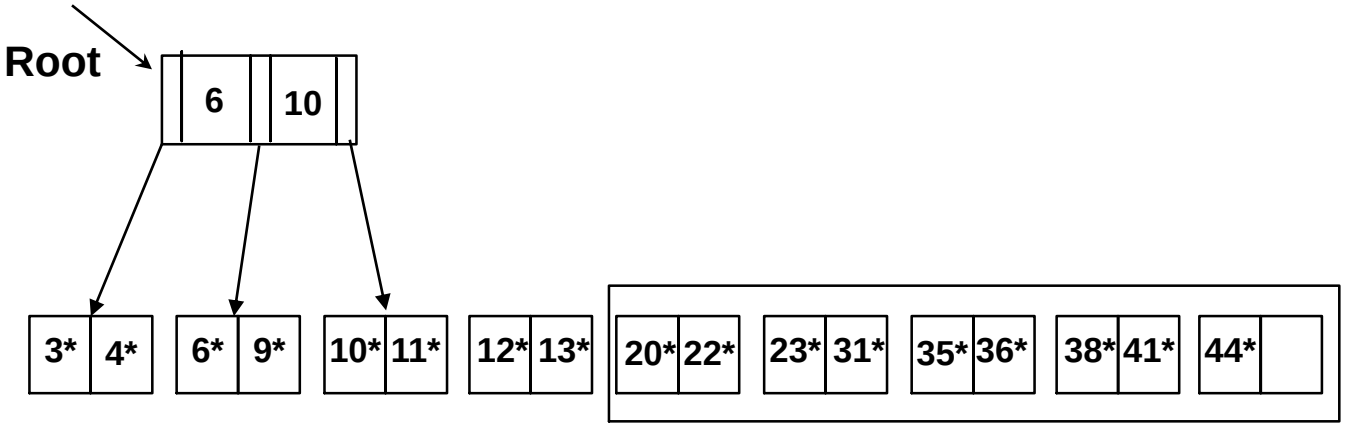
B+ Tree: Bulk Loading

- What to do?
  - Add one entry to the root page for each subsequent page of the sorted data entries *(i.e., <lowest key value on page, pointer to the page>)*



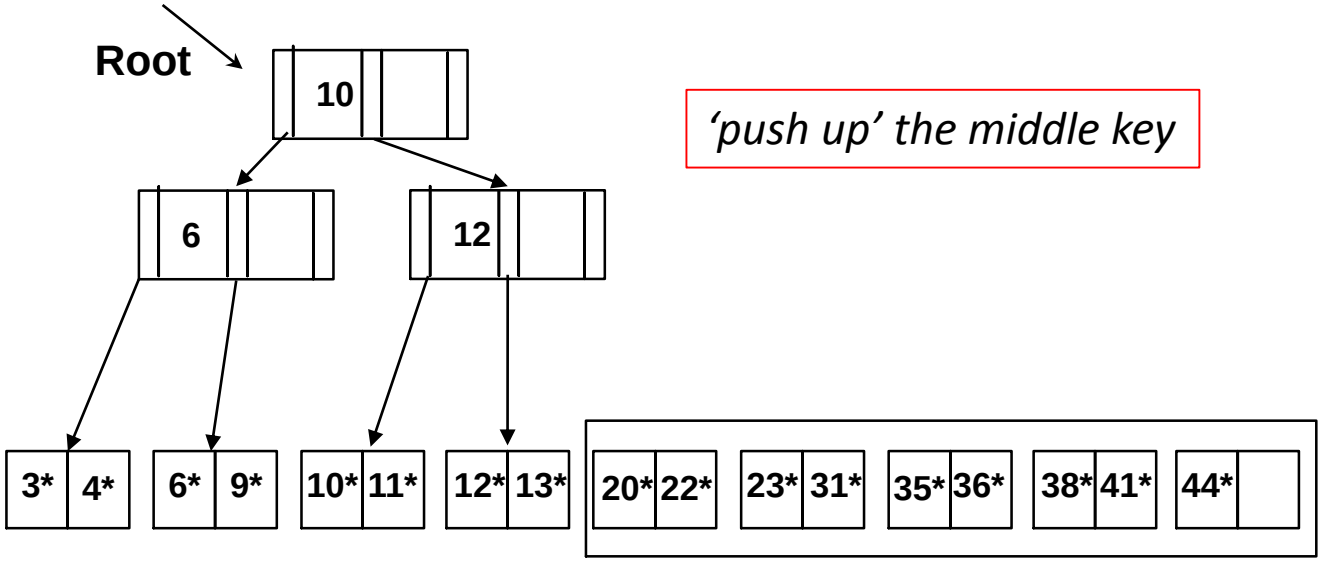
B+ Tree: Bulk Loading

- What to do?
  - Split the root and create a new root page



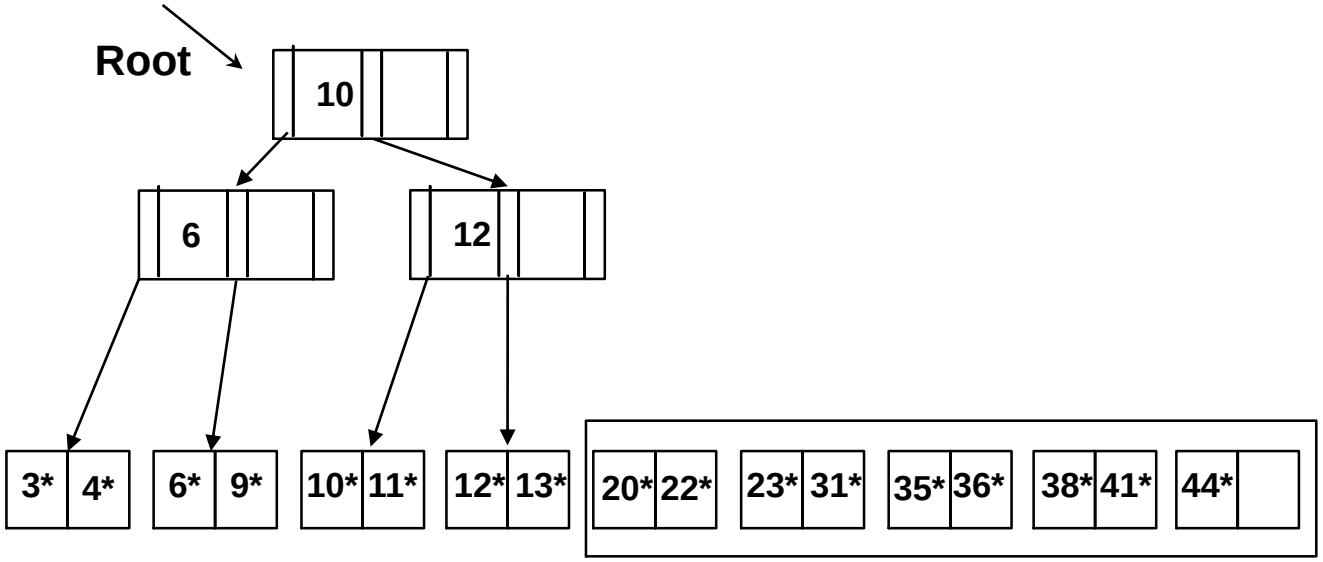
B+ Tree: Bulk Loading

- What to do?
  - Split the root and create a new root page



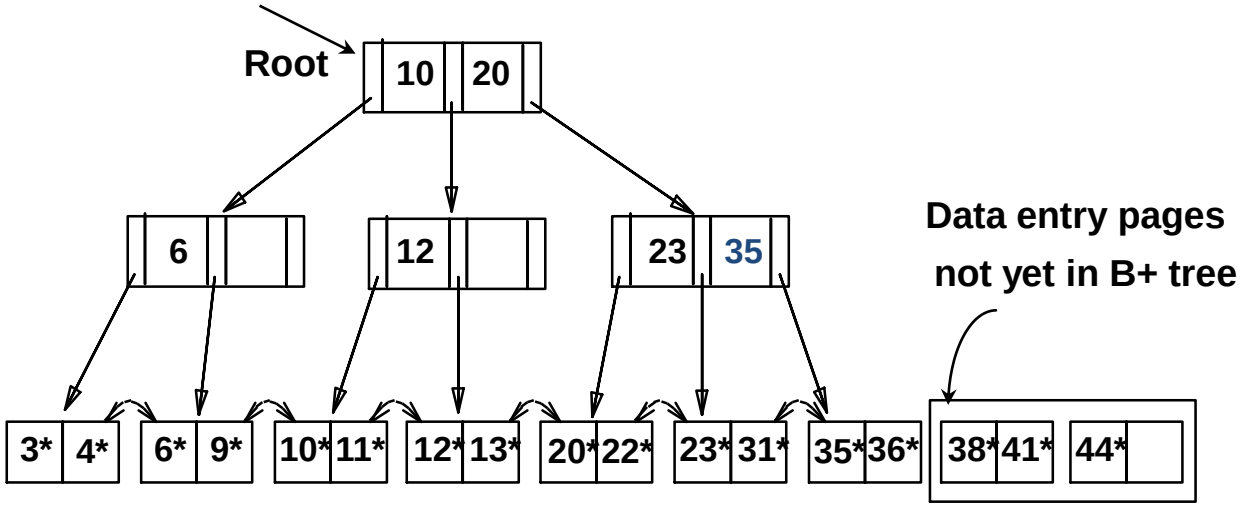
B+ Tree: Bulk Loading

- What to do?
  - Continue by inserting entries into the right-most index page just above the leaf page; split when fills up



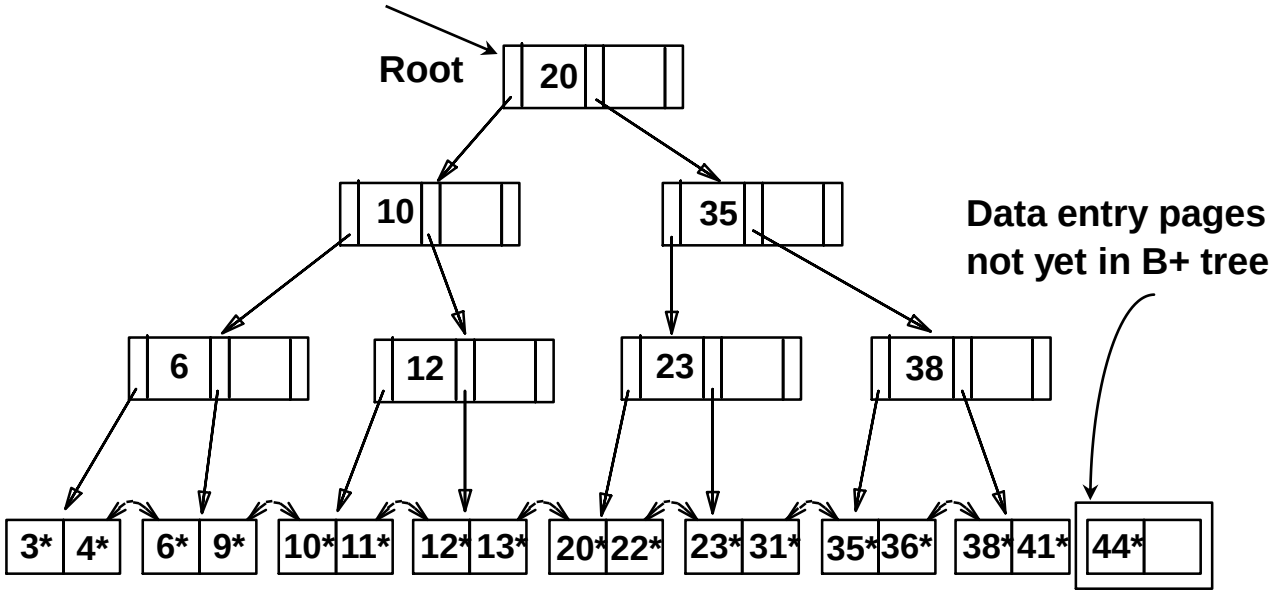
B+ Tree: Bulk Loading

- What to do?
  - Continue by inserting entries into the right-most index page just above the leaf page; split when fills up



B+ Tree: Bulk Loading

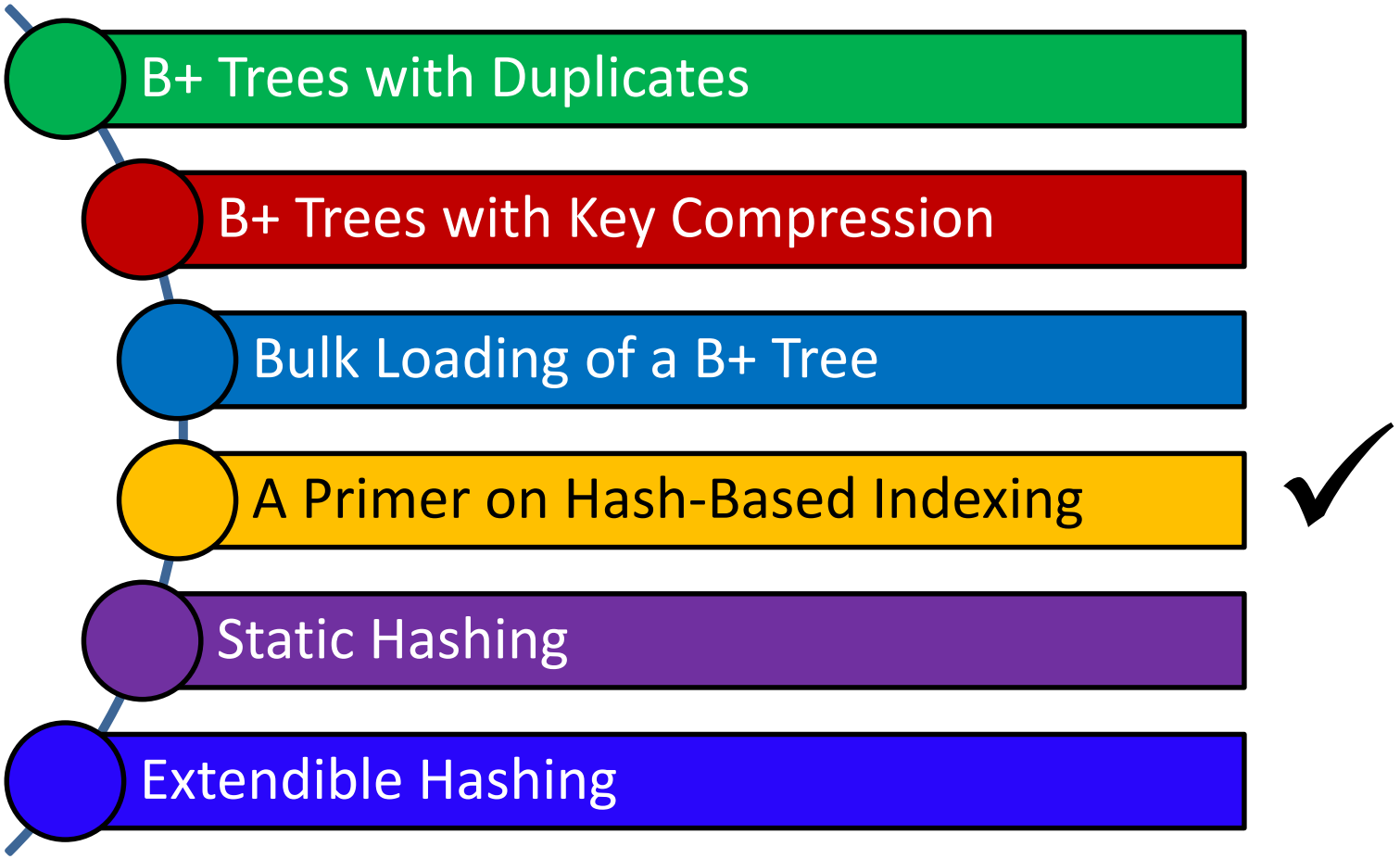
- What to do?
  - Continue by inserting entries into the right-most index page just above the leaf page; split when fills up



B+ Tree: Bulk Loading

- What is the cost of bulk loading?
  1. Creating the leaf-level entries
     - Scanning the data entries and writing out all the leaf-level entries (i.e., ***K****)
     - Hence, (***R***+***E***) I/Os, where ***R*** is the number of pages containing data entries and ***E*** is the number of pages containing ***K****entries
  2. Sorting leaf-level entries
     - 3***E*** I/Os (*when discussing sorting, we will see how*)
  3. Building the index from the sorted leaf-level entries
     - The cost of writing out all index-level pages (*will be an exercise in the recitation*)!

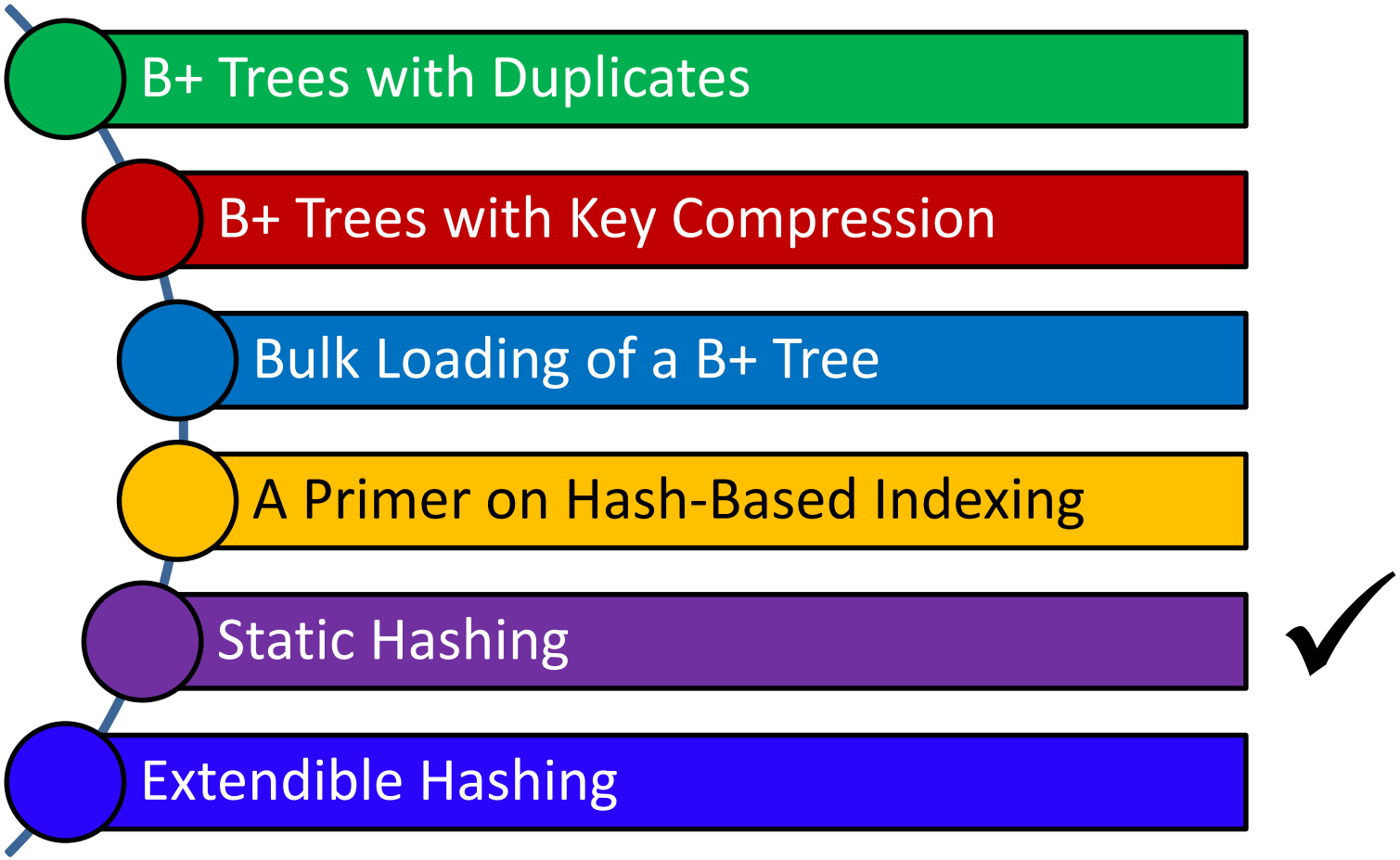
Outline



Hash-Based Indexing

- What indexing technique can we use to support *range searches* (e.g., “Find s_name where gpa >= 3.0)?
  - Tree-Based Indexing
- What about *equality selections* (e.g., “Find s_name where sid = 102”?
  - Tree-Based Indexing
  - Hash-Based Indexing (*cannot support range searches!*)
- Hash-based indexing, however, proves to be very useful in implementing relational operators (e.g., joins)

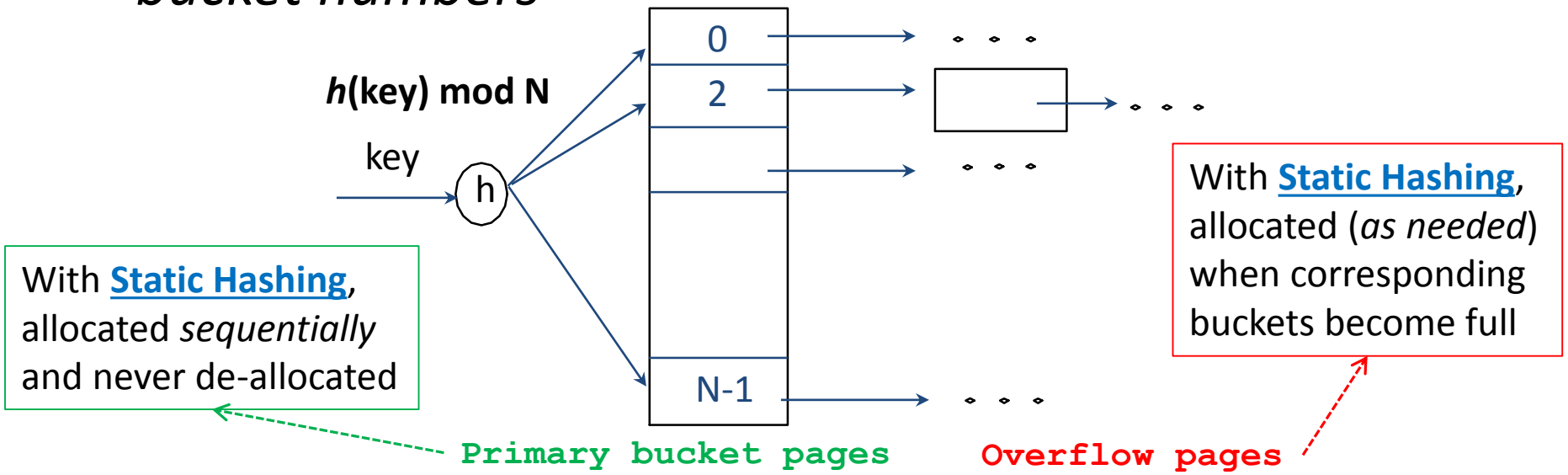
Outline



Static Hashing

- A hash structure (or table or file) is a *generalization* of the simpler notion of an ordinary array
  - In an array, an arbitrary position can be examined in O(1)

- A hash function ***h*** is used to map keys into a range of *bucket numbers*



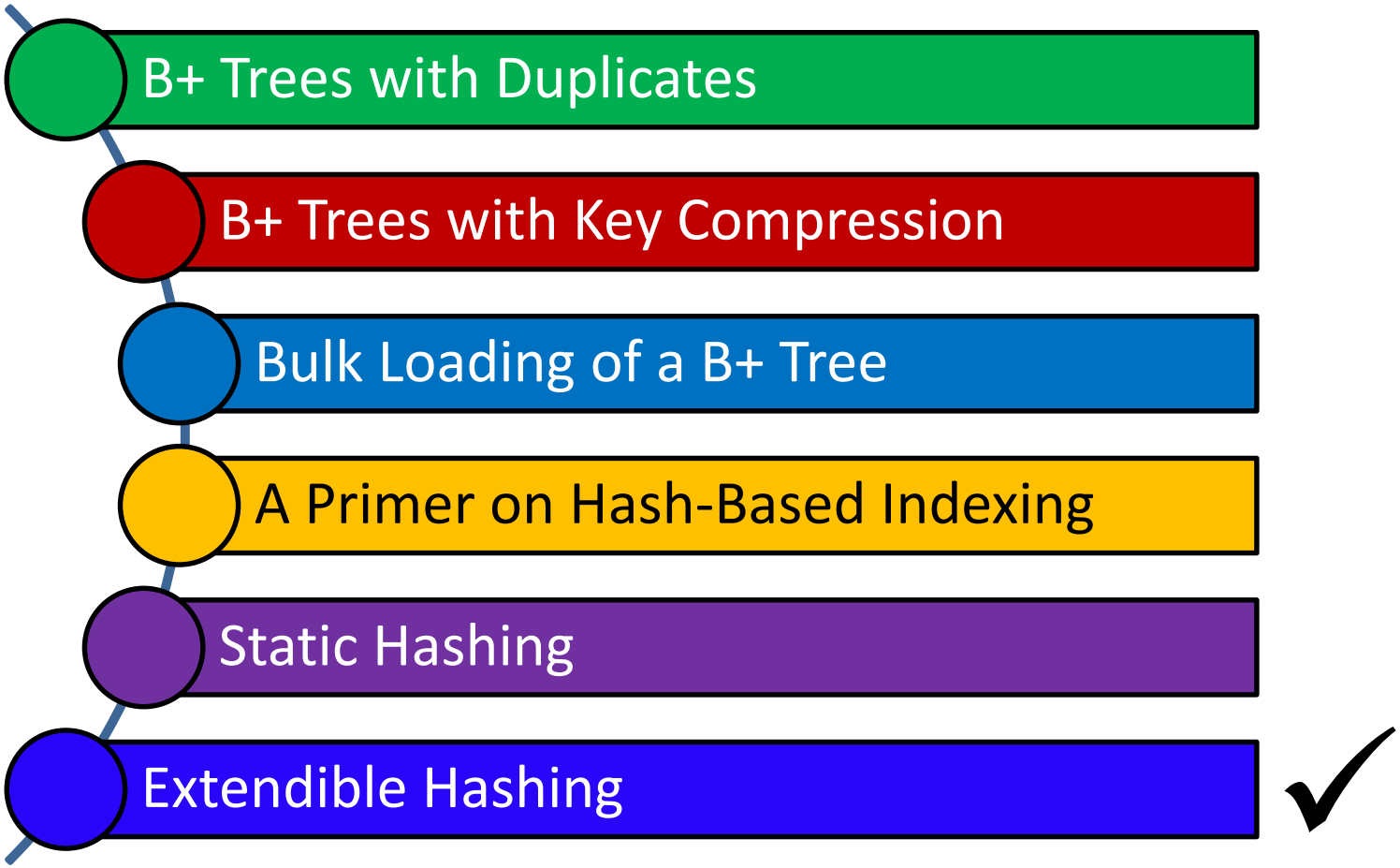
Static Hashing

- Data entries can be any of the three alternatives (**A (1)**, **A (2)** or **A (3)**- see previous lecture)
- Data entries can be *sorted* in buckets to speed up searches
- The hash function ***h*** is used to identify the bucket to which a given key belongs and subsequently *insert*, *delete* or *locate* a respective data record
  - A hash function of the form ***h***(key) = (**a** * key + **b**) works well in practice
- A search *ideally* requires 1 disk I/O, while an insertion or a deletion necessitates 2 disk I/Os

Static Hashing: Some Issues

- Similar to ISAM, the number of buckets is fixed!
  - Cannot deal with insertions and deletions gracefully
- Long overflow chains can develop easily and degrade performance!
  - Pages can be initially kept *only* 80% full
- *Dynamic* hashing techniques can be used to fix the problem
  - Extendible Hashing (EH)
  - Liner Hashing (LH)

Outline



Directory of Pointers

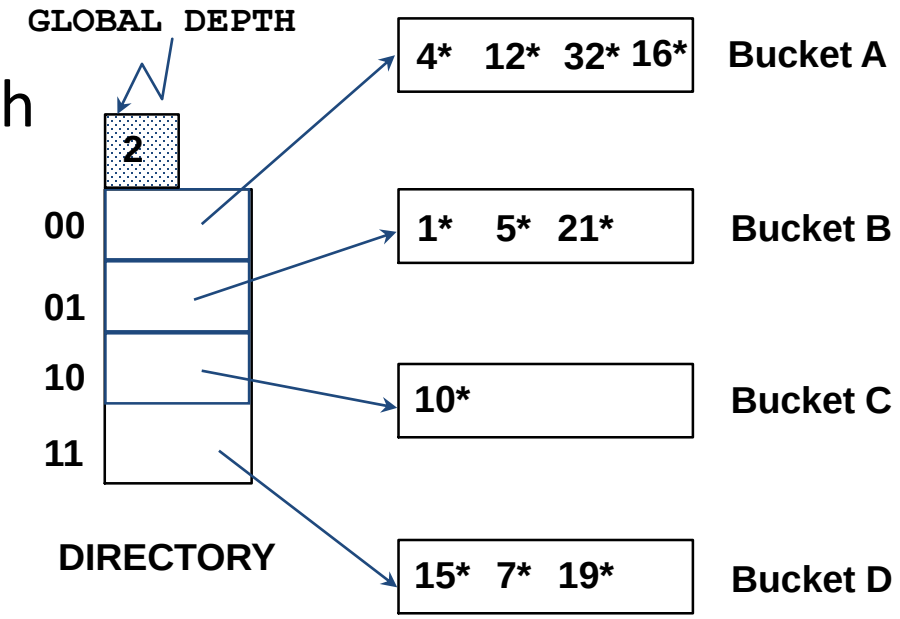
- How else (*as opposed to overflow pages*) can we add a data record to a full bucket in a *static* hash file?
  - Reorganize the table (e.g., by doubling the number of buckets and redistributing the entries across the new set of buckets)
  - But, reading and writing all pages is expensive!

- In contrast, we can use a directory of pointers to buckets
  - Buckets number can be doubled by doubling just the directory and *splitting "only" the bucket that overflowed*
  - The *trick* lies on how the hash function can be adjusted!

Extendible Hashing

- Extendible Hashing uses a directory of pointers to buckets

- The result of applying a hash function ***h*** is treated as a *binary number* and the last ***d*** bits are interpreted as an offset into the directory

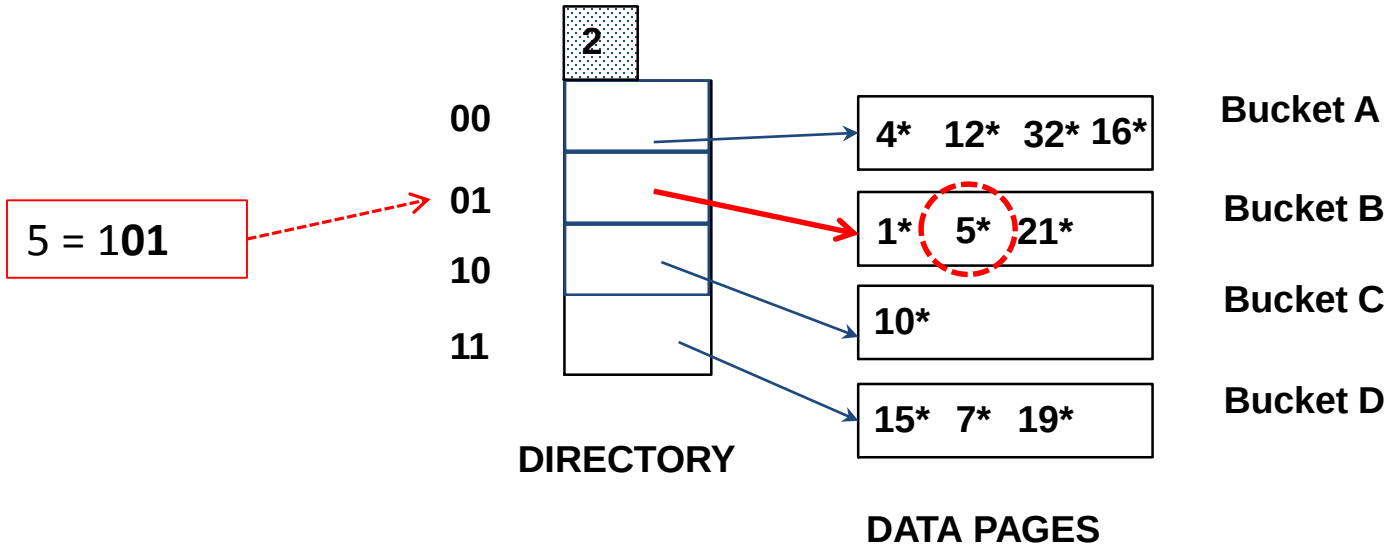


- ***d*** is referred to as the ***global depth*** of the hash file and is kept as part of the header of the file

Extendible Hashing: Searching for Entries

- To search for a data entry, apply a hash function ***h*** to the key and take the last ***d*** bits of its binary representation to get the bucket number

- Example: search for 5*



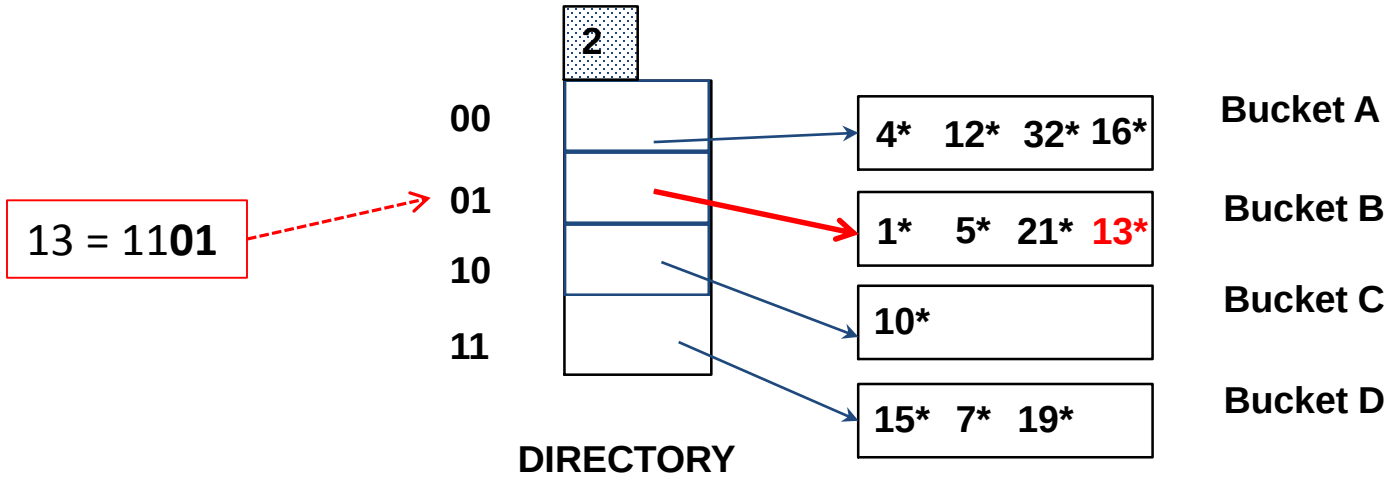
Extendible Hashing: Inserting Entries

- An entry can be inserted as follows:
  - Find the appropriate bucket (*as in search*)
  - Split the bucket *if full* and *redistribute* contents (including the new entry to be inserted) across the old bucket and its ***"split image"***
  - Double the directory *if necessary*
  - Insert the given entry

Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

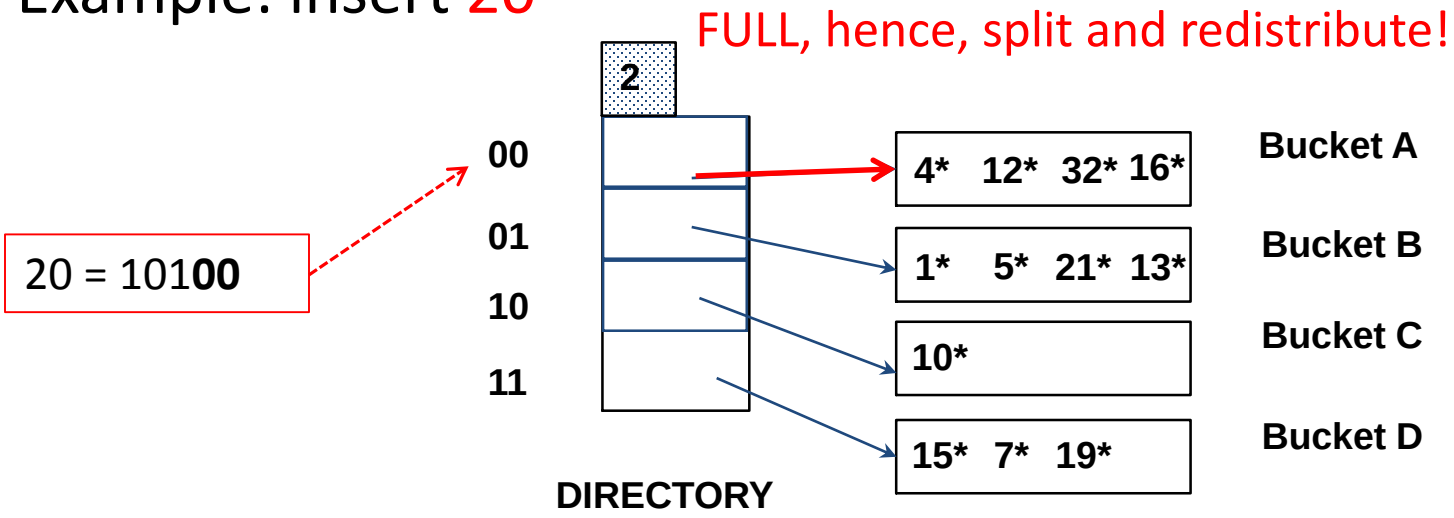
- Example: insert 13*



Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

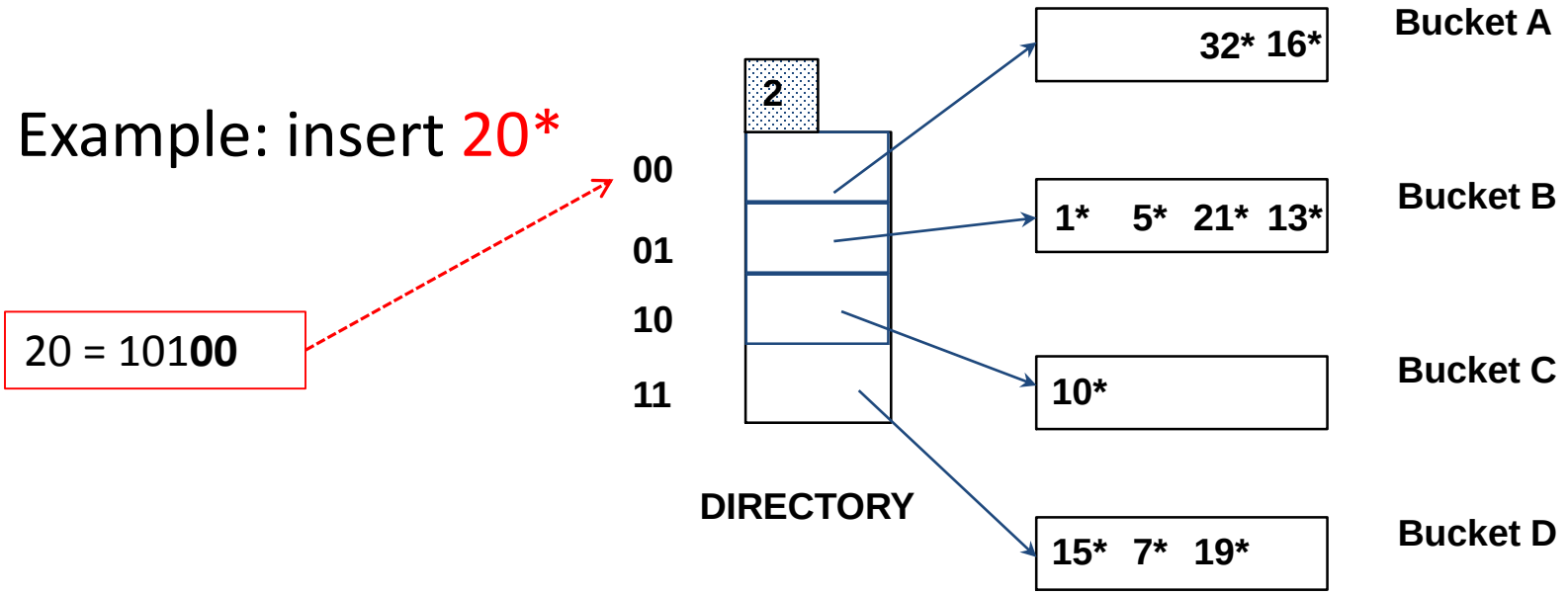
- Example: insert 20*



Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 20*

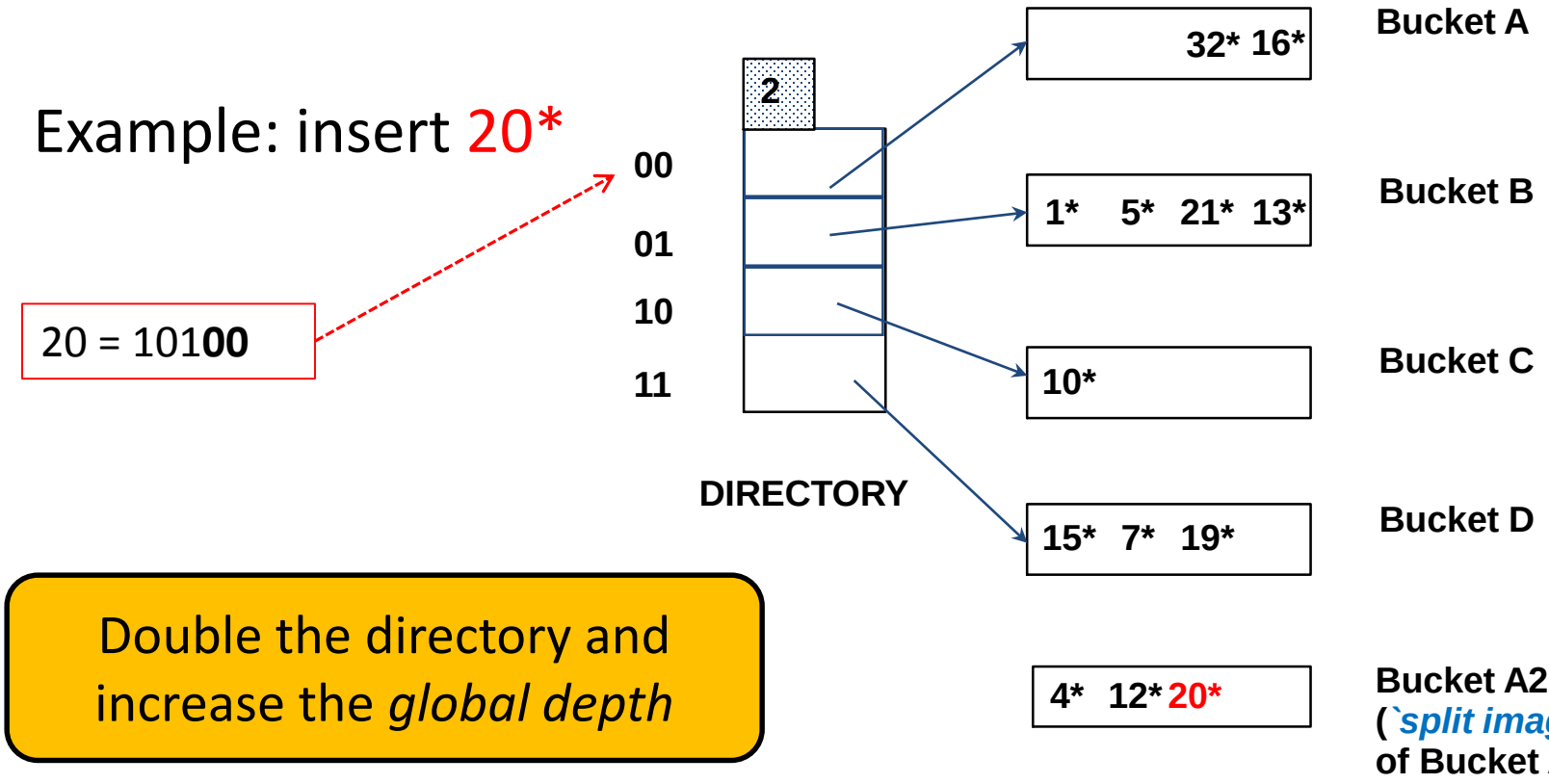


Is this enough?

Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 20*

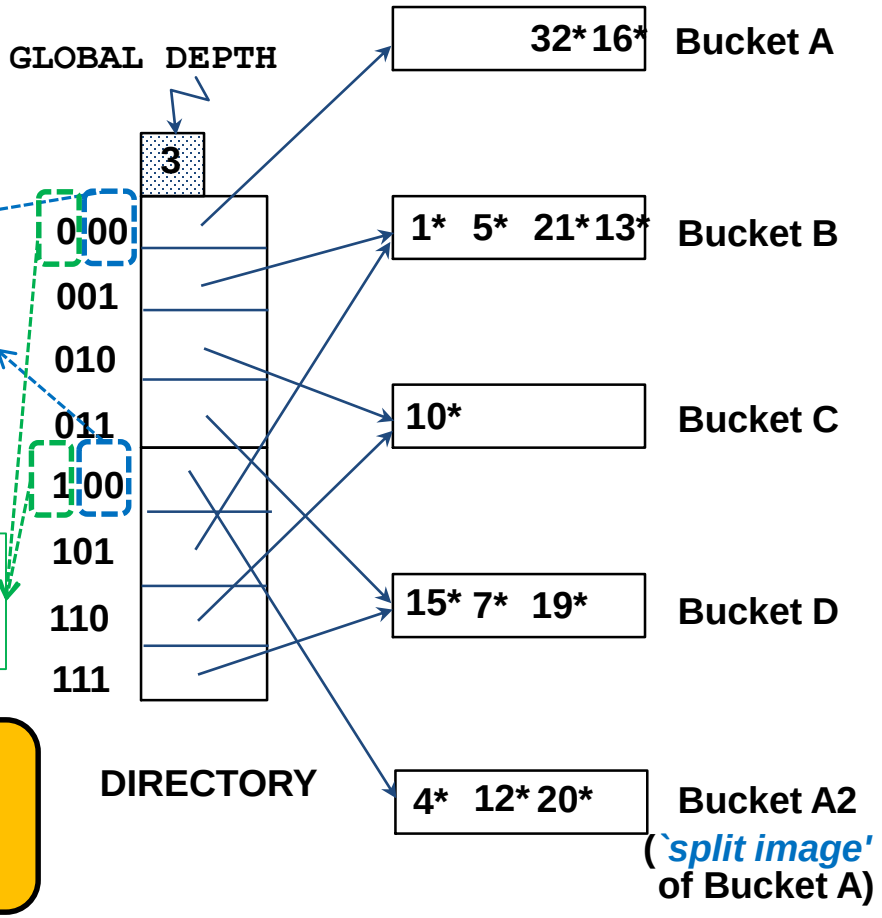


Double the directory and increase the *global depth*

Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 20*



These two bits indicate a data entry that belongs to one of these two buckets

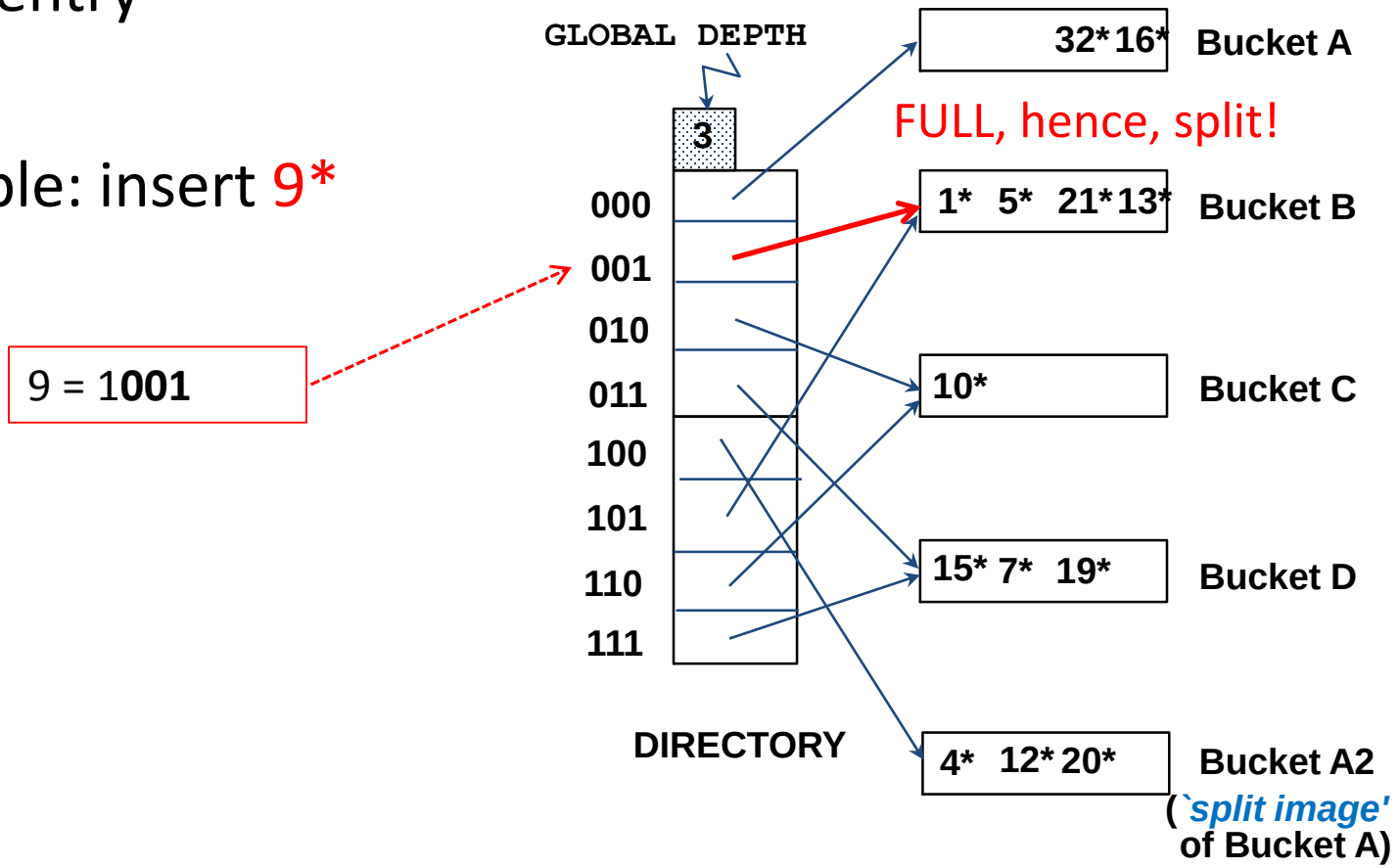
The third bit distinguishes between these two buckets!

But, is it necessary always to double the directory?

Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

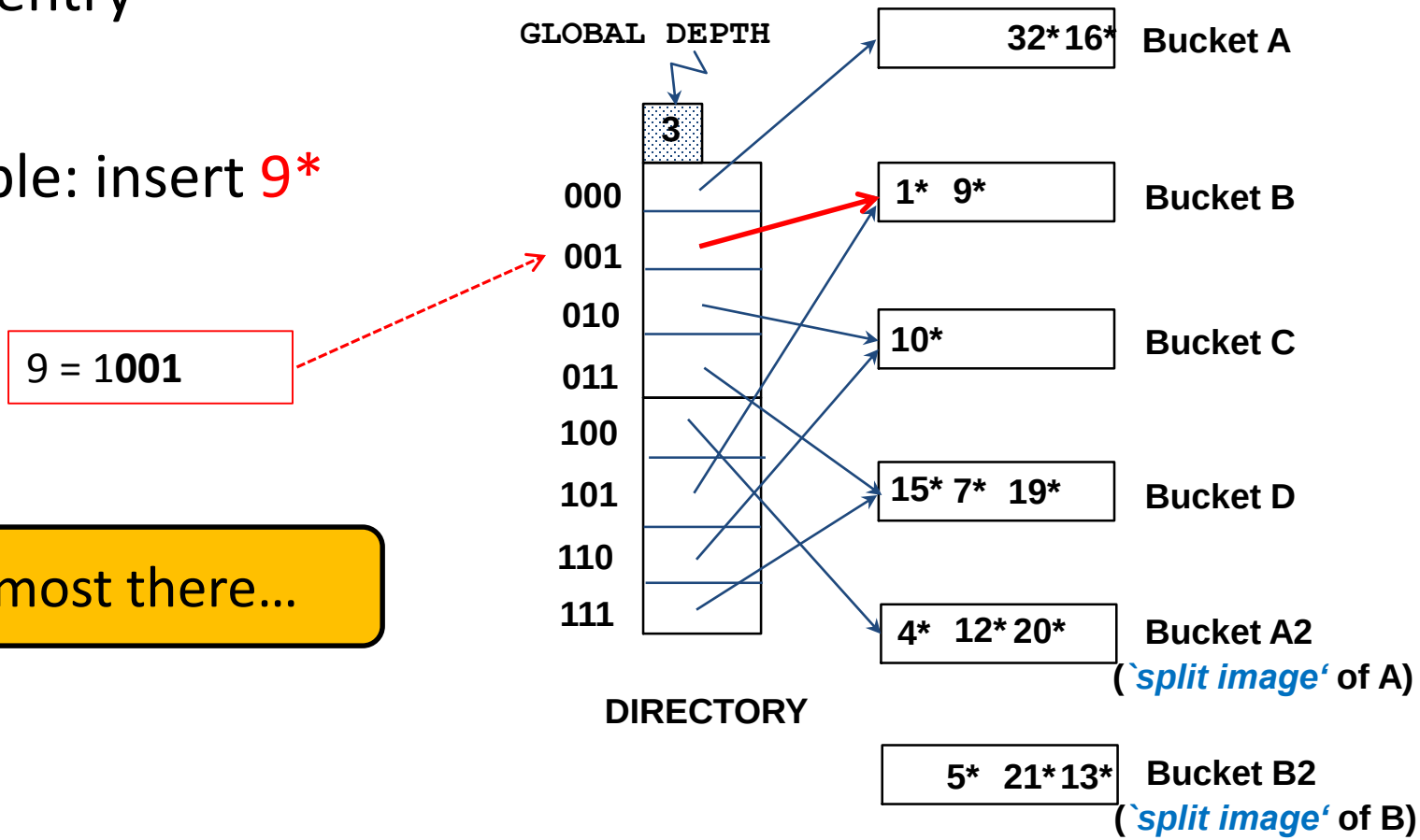
- Example: insert 9*



Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

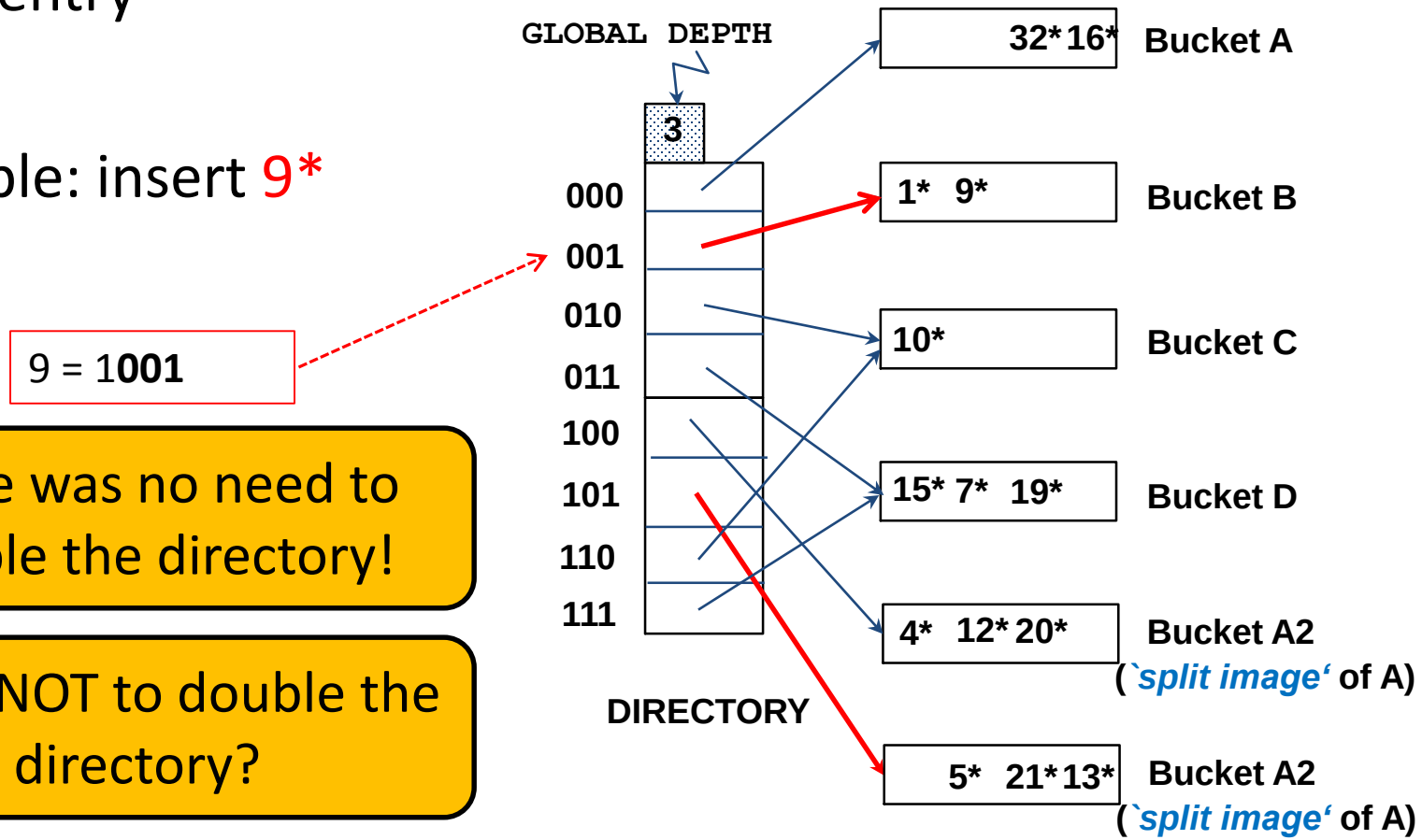
- Example: insert 9*



Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

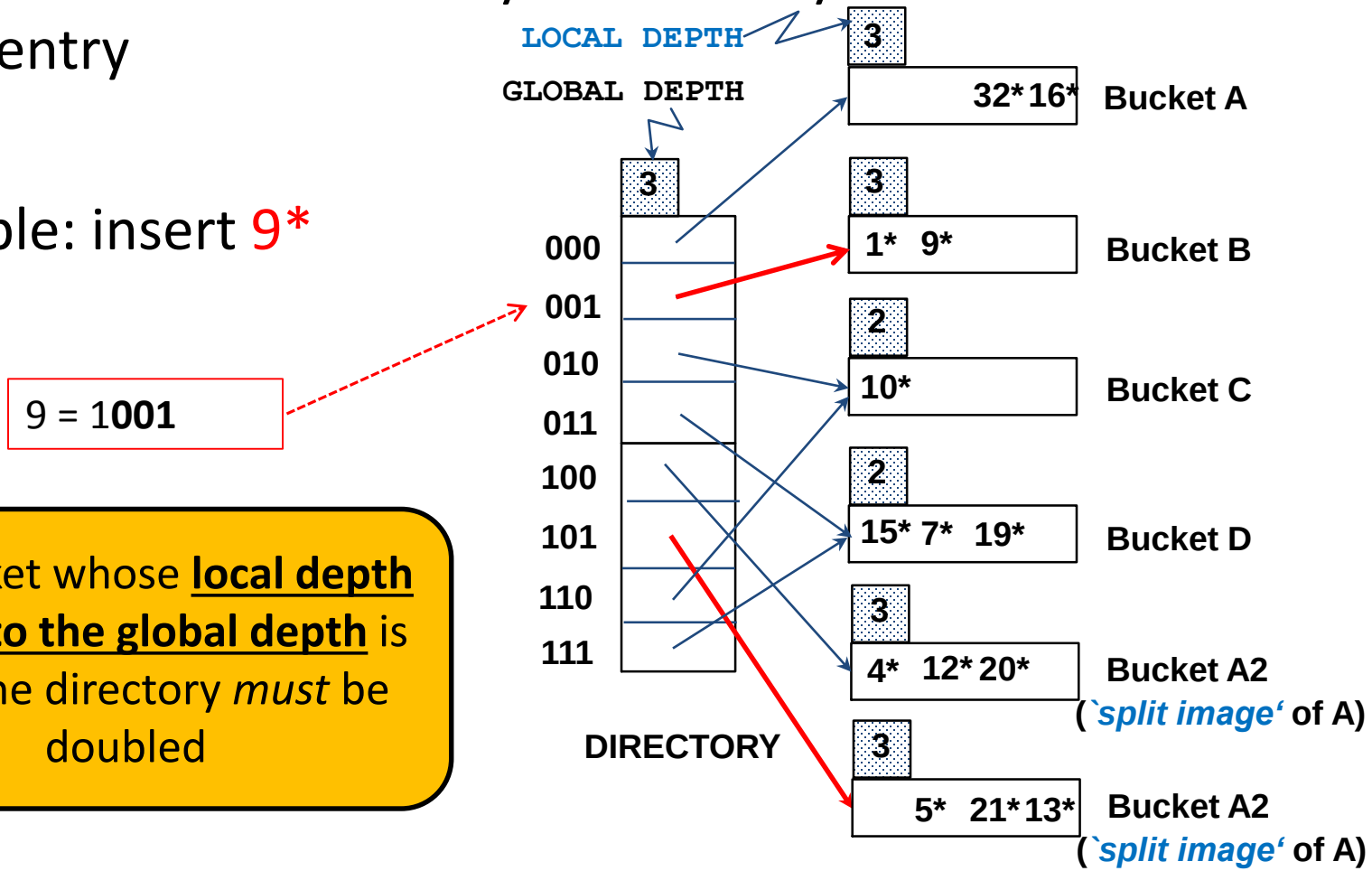
- Example: insert 9*



Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 9*

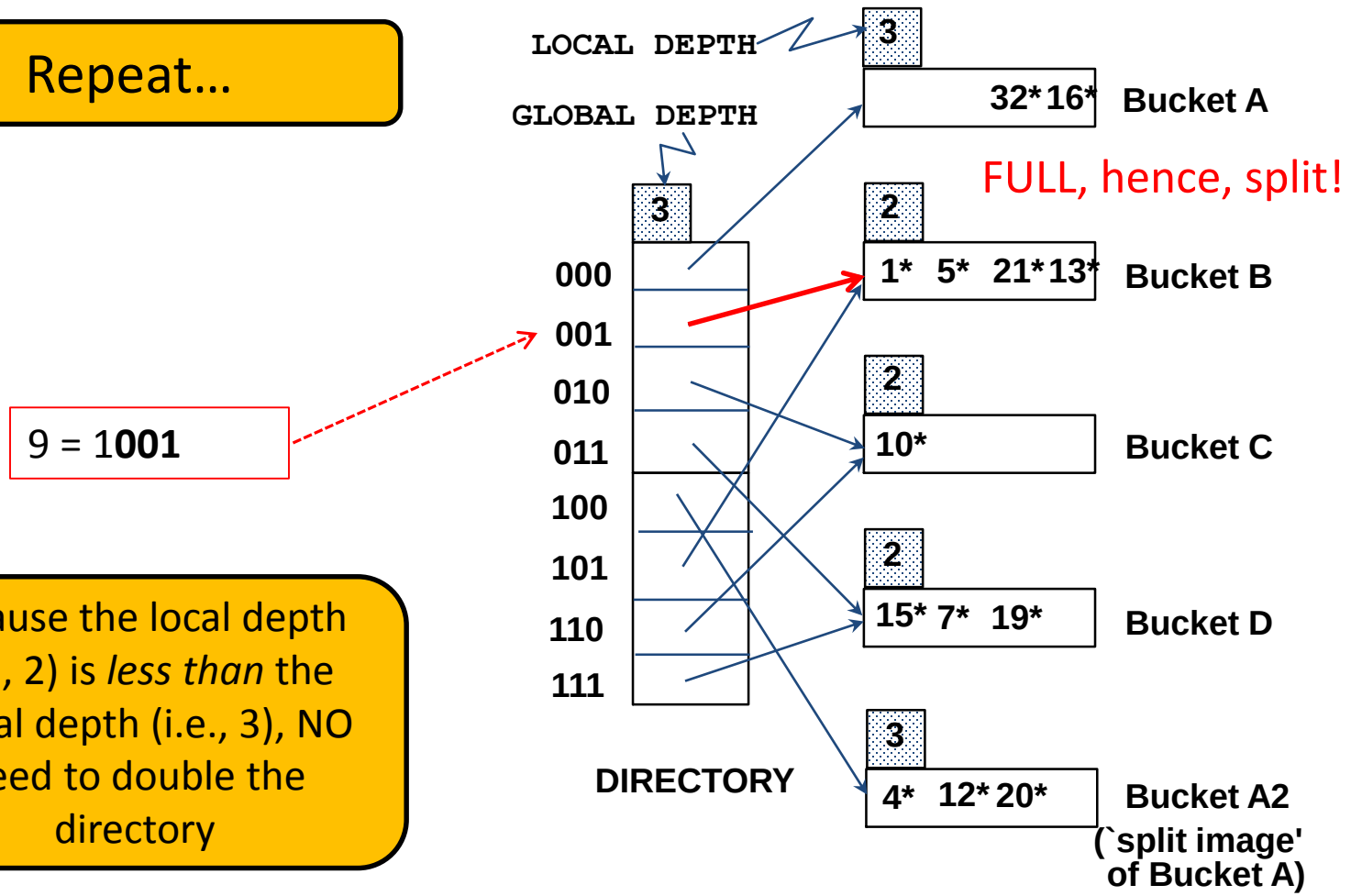


If a bucket whose **local depth equals to the global depth** is split, the directory *must* be doubled

Extendible Hashing: Inserting Entries

- Example: insert 9*

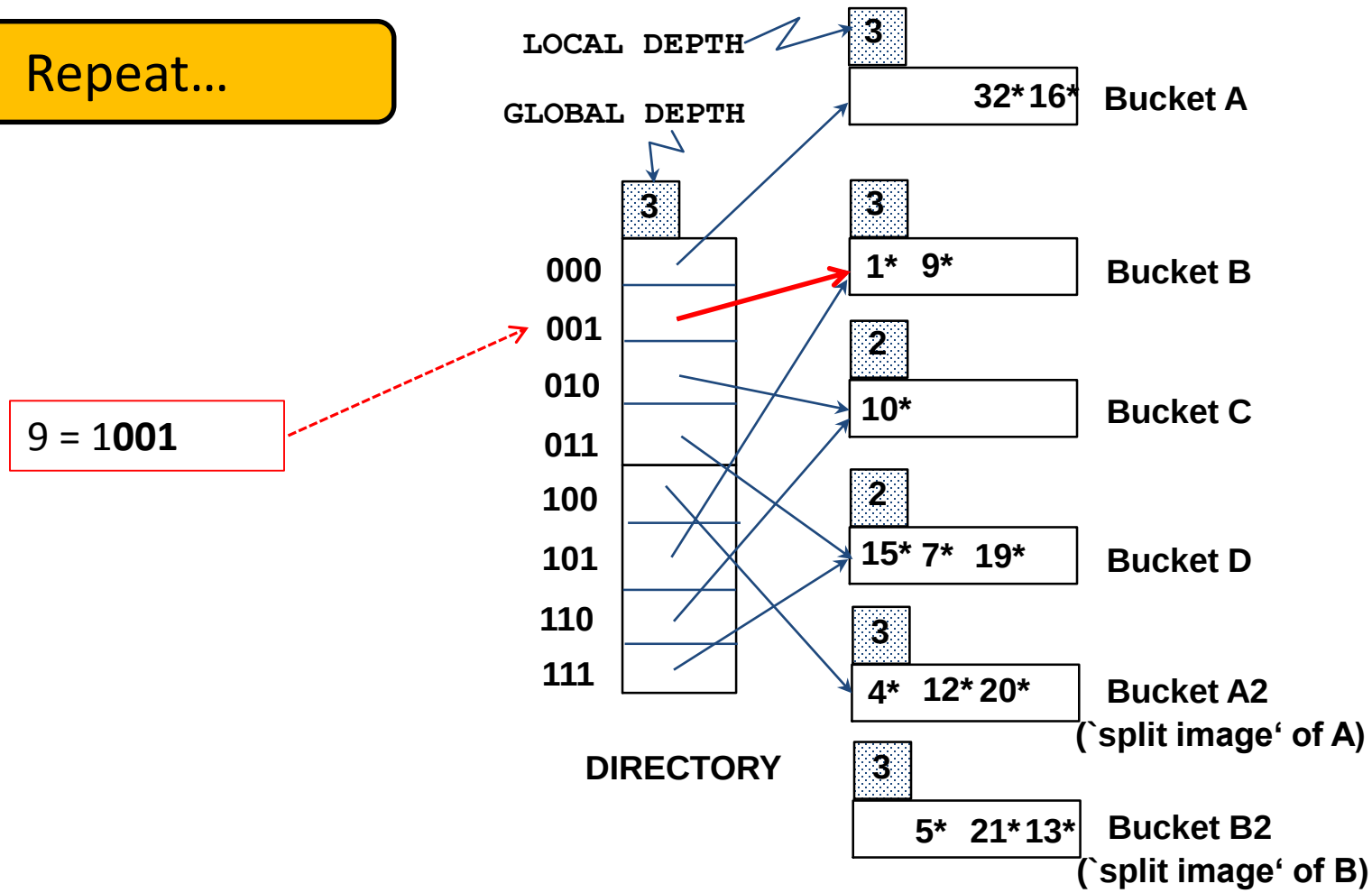
Repeat…



Extendible Hashing: Inserting Entries

- Example: insert 9*

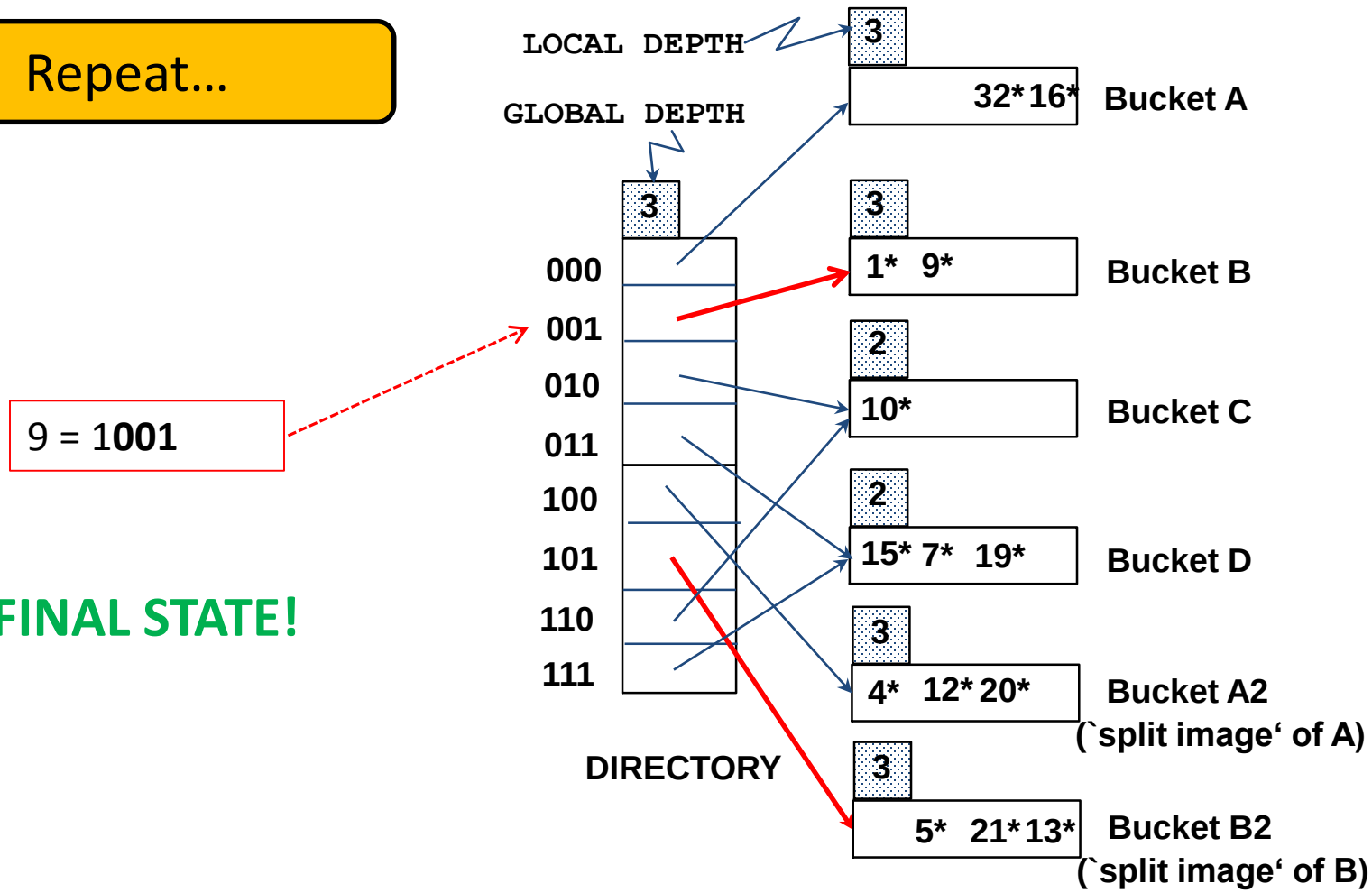
Repeat…



Extendible Hashing: Inserting Entries

- Example: insert 9*

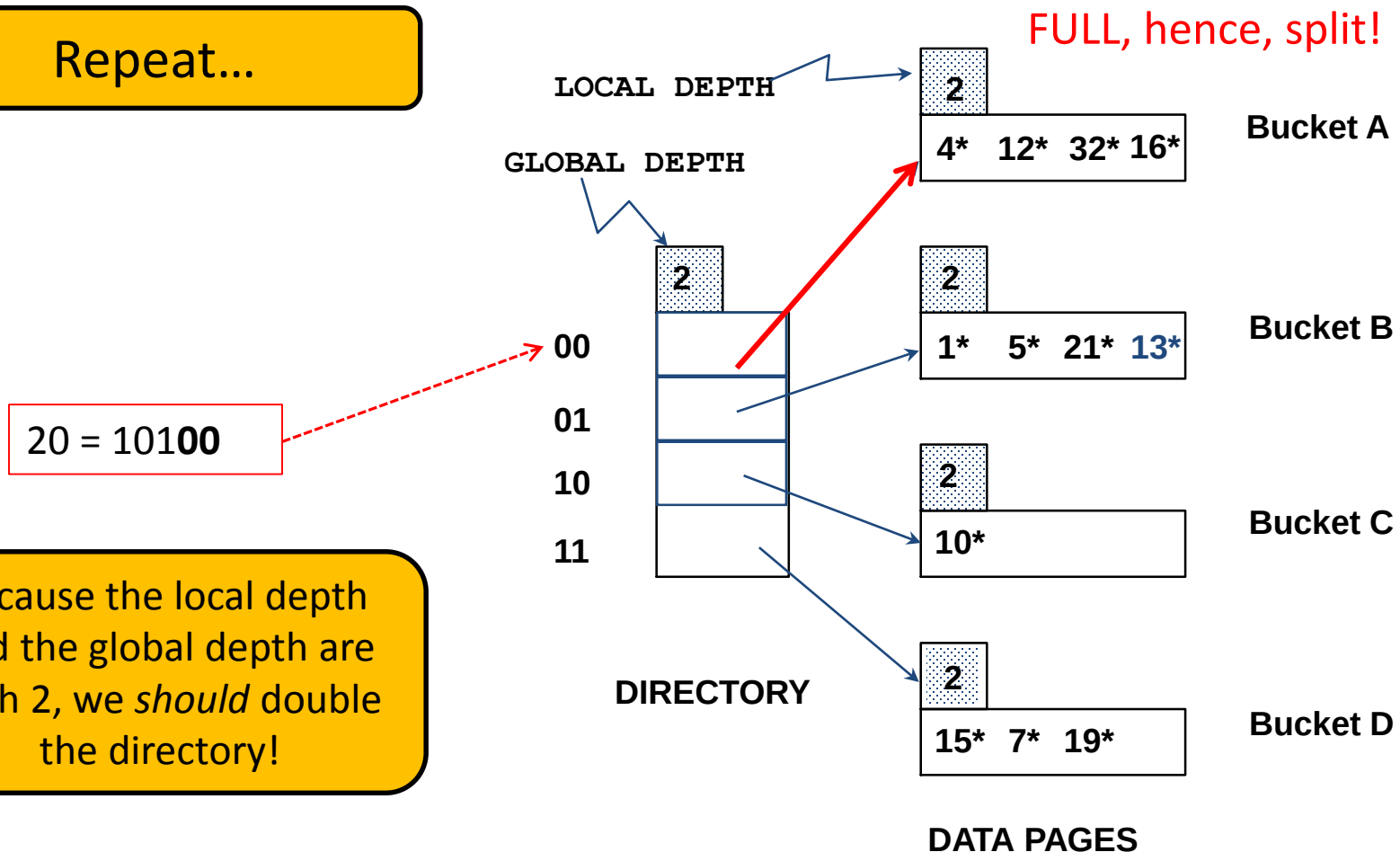
Repeat…



Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

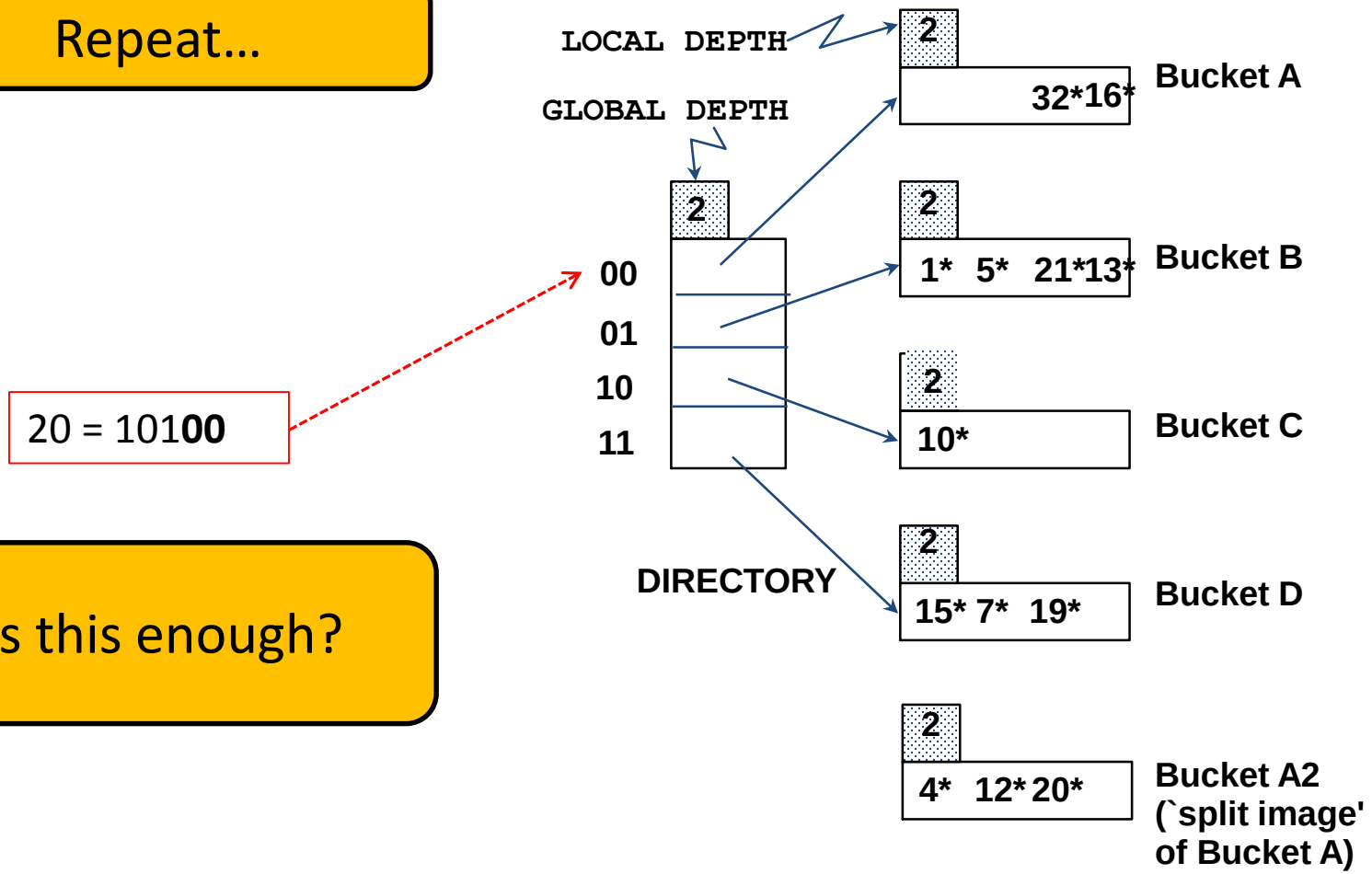


Because the local depth and the global depth are both 2, we *should* double the directory!

Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…



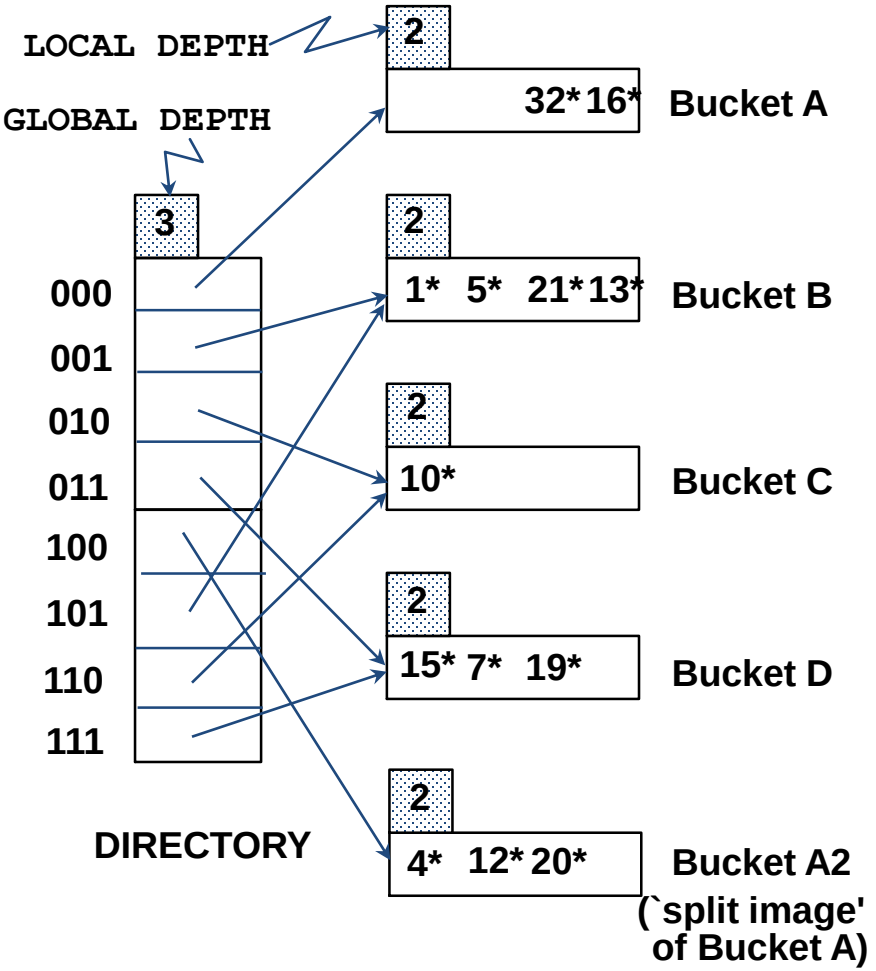
Is this enough?

Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

Is this enough?

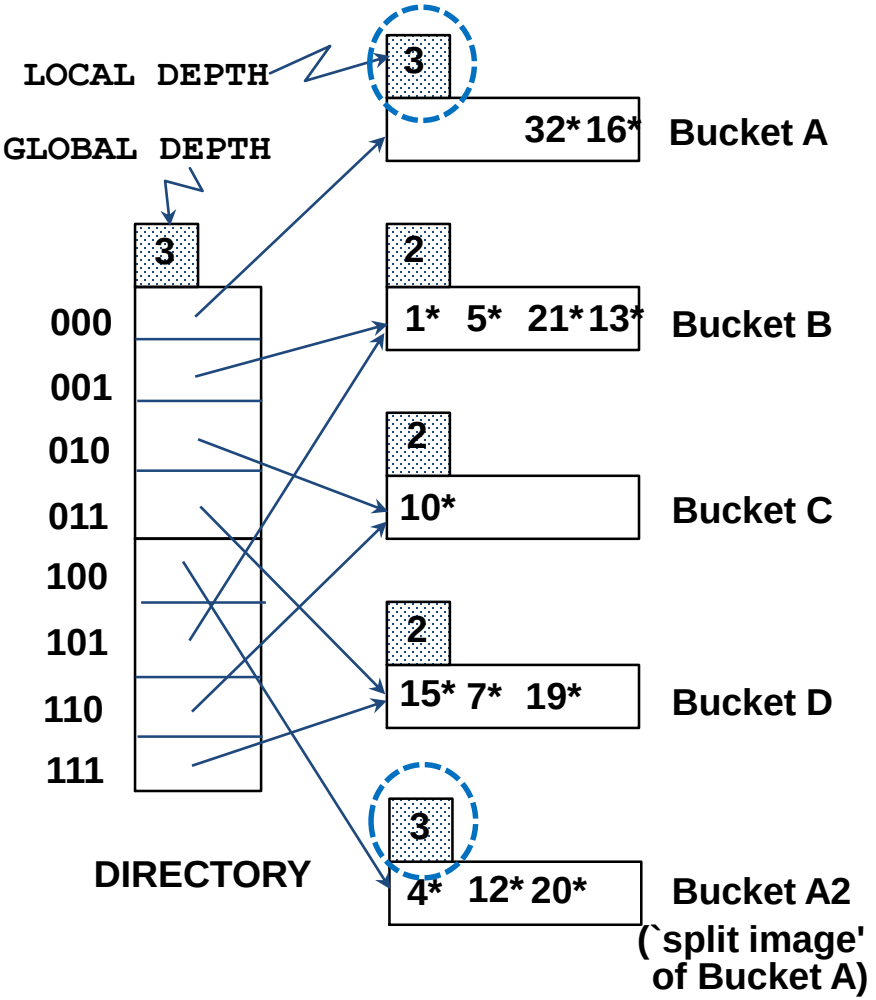


Extendible Hashing: Inserting Entries

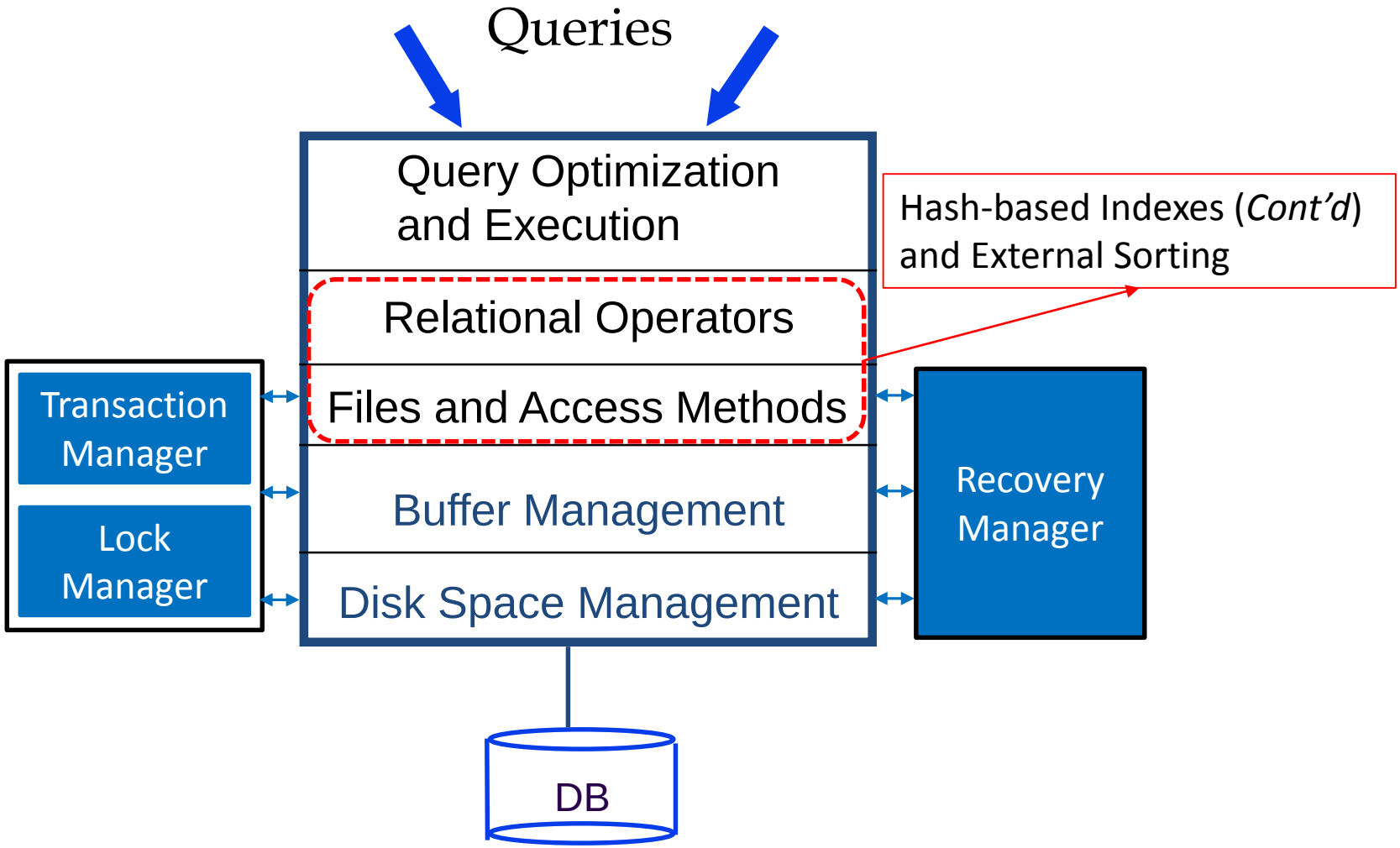
- Example: insert 20*

Repeat…

FINAL STATE!



Next Class


Queries
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
DB
Hash-based Indexes (*Cont'd*) and External Sorting