Databases

Lecture 10

The Physical Structure of Databases (III)

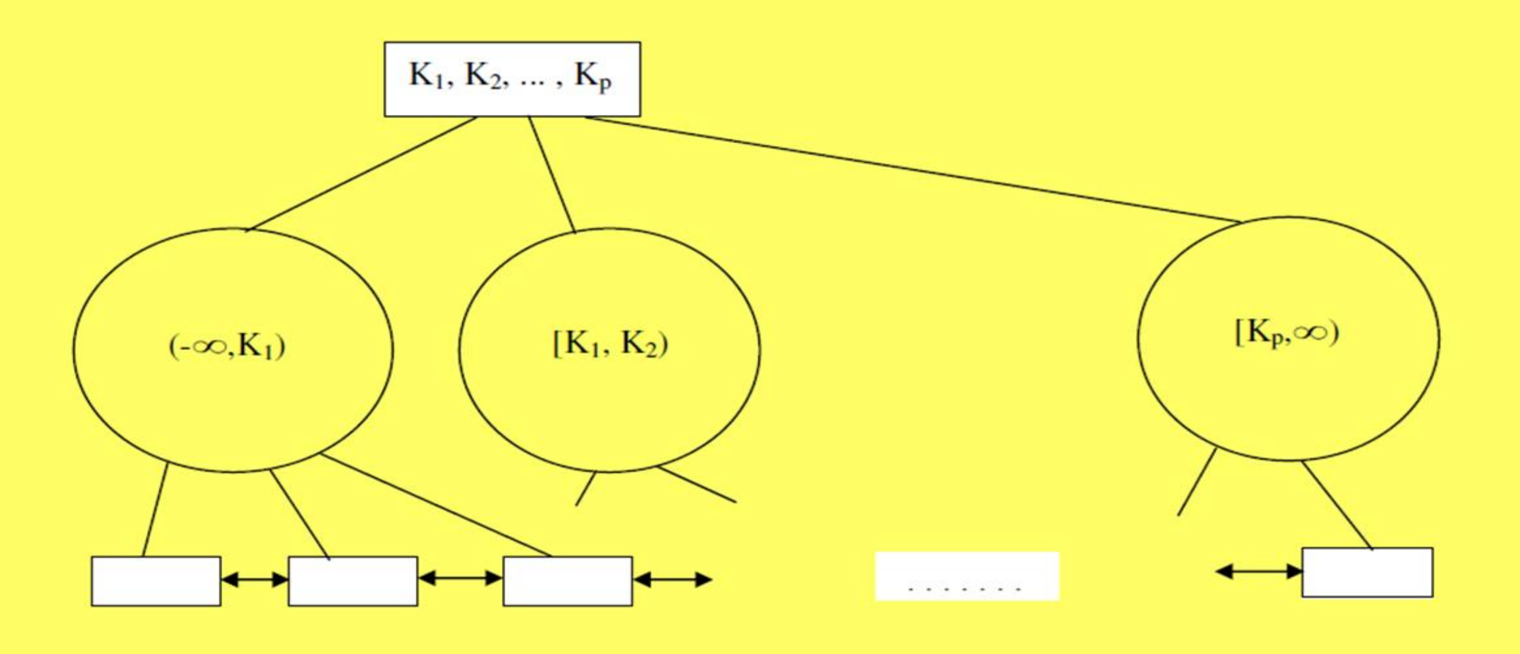
- Indexes. Tree-Structured Indexing. Hash-Based Indexing -

Sabina S. CS

B+ tree

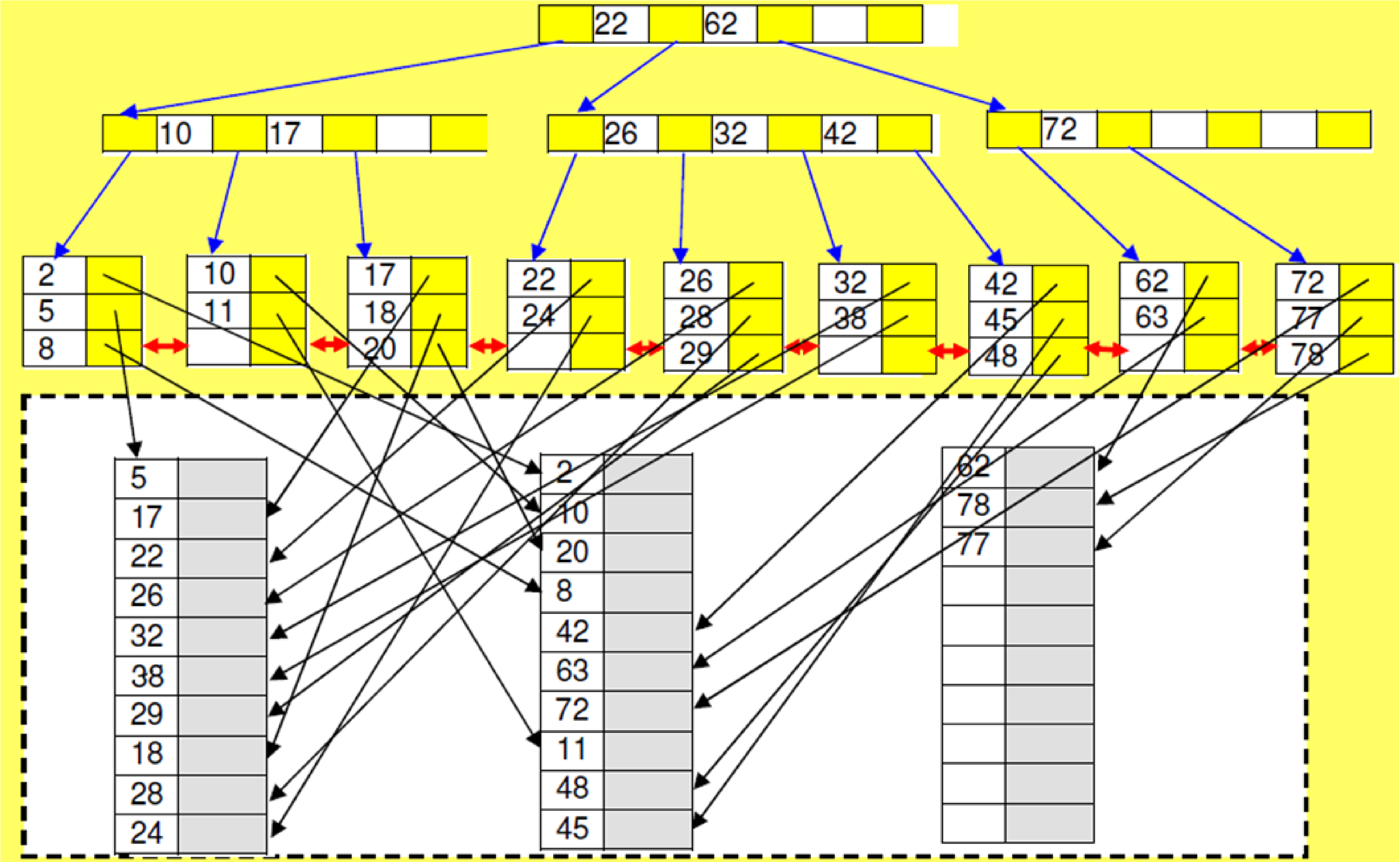
B+ tree

- B-tree variant
- last level contains all values (key values and the records' addresses)
- some key values can also appear in non-terminal nodes, without the records' addresses; their purpose is to separate values from terminal nodes (guide the search)
- terminal nodes are maintained in a doubly linked list (data can be easily scanned)



B+ tree

- example



B+ tree

- storing a B+ tree
  - B-tree methods
- operations (algorithms)
  - B-tree

B+ tree - in practice

- concept of *order* - relaxed, replaced by a physical space criterion (for instance, nodes should be at least half-full)
- terminal / non-terminal nodes - different numbers of entries; usually, inner nodes can store more entries than terminal ones
- variable-length search key => variable-length entries => variable number of entries / page
- if alternative 3 is used (<k, rid_list>) => variable-length entries (in the presence of duplicates), even if attributes are of fixed length

B+ tree - in practice

* prefix key compression

- larger key size => less index entries fit on a page, i.e., less children / index page => larger B+ tree height
- keys in index entries - just direct the search => often, they can be compressed
- adjacent index entries with search key values: *Meteiut*, *Mircqkjt*, *Morqwkj*
- compress key values: *Me, Mi,* etc
- what if the subtree also contains *Micfgjh*? => need to store *Mir* (instead of *Mi*)
- it's not enough to analyze neighbor index entries *Meteiut* and *Morqwkj*; the largest key value in *Mircqkjt*'s left subtree and the smallest key value in its right subtree must also be examined
- inserts / deletes - modified correspondingly

B+ tree - in practice

- values found in practice
  - order – 200
  - fill factor (node) – 67%
  - fan-out – 133
  - capacity
    - height 4: 133^4 = 312,900,721
    - height 3: 133^3 = 2,352,637
- top levels can often be kept in the BP
  - 1st level – 1 page (8KB)
  - 2nd level – 133 pages (approx. 1MB)
  - 3rd level – 133^2 =17689 pages (approx. 133 MB)

B+ tree - benefits

- balanced index => uniform search time
- rarely more than 3-5 levels, the top levels can be kept in main memory => only a few I/O operations are needed to search for a record
- widely used in DBMSs
- ideal for range selections, good for equality selections as well

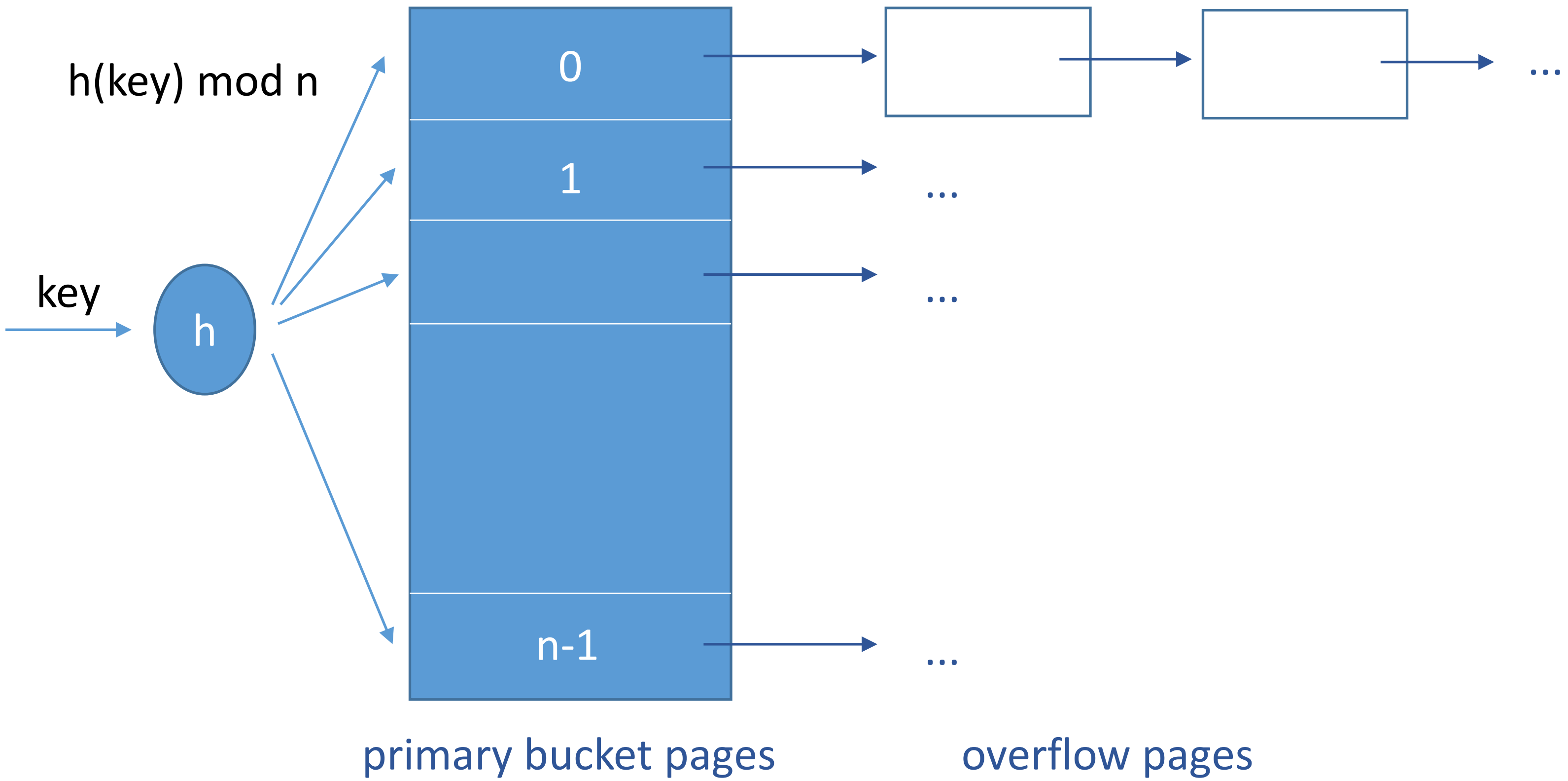
Hash-Based Indexing

- hashing function
  - maps search key values into a range of bucket numbers
- hashed file
  - search key (field(s) of the file)
  - records grouped into *buckets*
  - determine record r's bucket
    - apply hash function to search key
  - quick location of records with given search key value
    - example: file hashed on *EmployeeName*
      - Find employee *Popescu.*
- ideal for equality selections

Static Hashing

* static hashing

- buckets 0 to n-1
- bucket
  - one primary page
  - possibly extra overflow pages
- data entries in buckets
  - a1/a2/a3



* static hashing
- search for a data entry
  - apply hashing function to identify the bucket
  - search the bucket
  - possible optimization
    - entries sorted by search key

* static hashing
- add a data entry
  - apply hashing function to identify the bucket
  - add the entry to the bucket
  - if there is no space in the bucket:
    - allocate an overflow page
    - add the data entry to the page
    - add the overflow page to the bucket's overflow chain

* static hashing

- delete a data entry
    - apply hashing function to identify the bucket
    - search the bucket to locate the data entry
    - remove the entry from the bucket
    - if the data entry is the last one on its overflow page:
        - remove the overflow page from its overflow chain
        - add the page to a free pages list

* static hashing

- good hashing function
  - few empty buckets
  - few records in the same bucket
  - i.e., key values are uniformly distributed over the set of buckets
  - good function in practice
    - *h(val) = a*val + b*
    - *h(val) mod n* to identify bucket, for buckets numbered 0..n-1

* static hashing

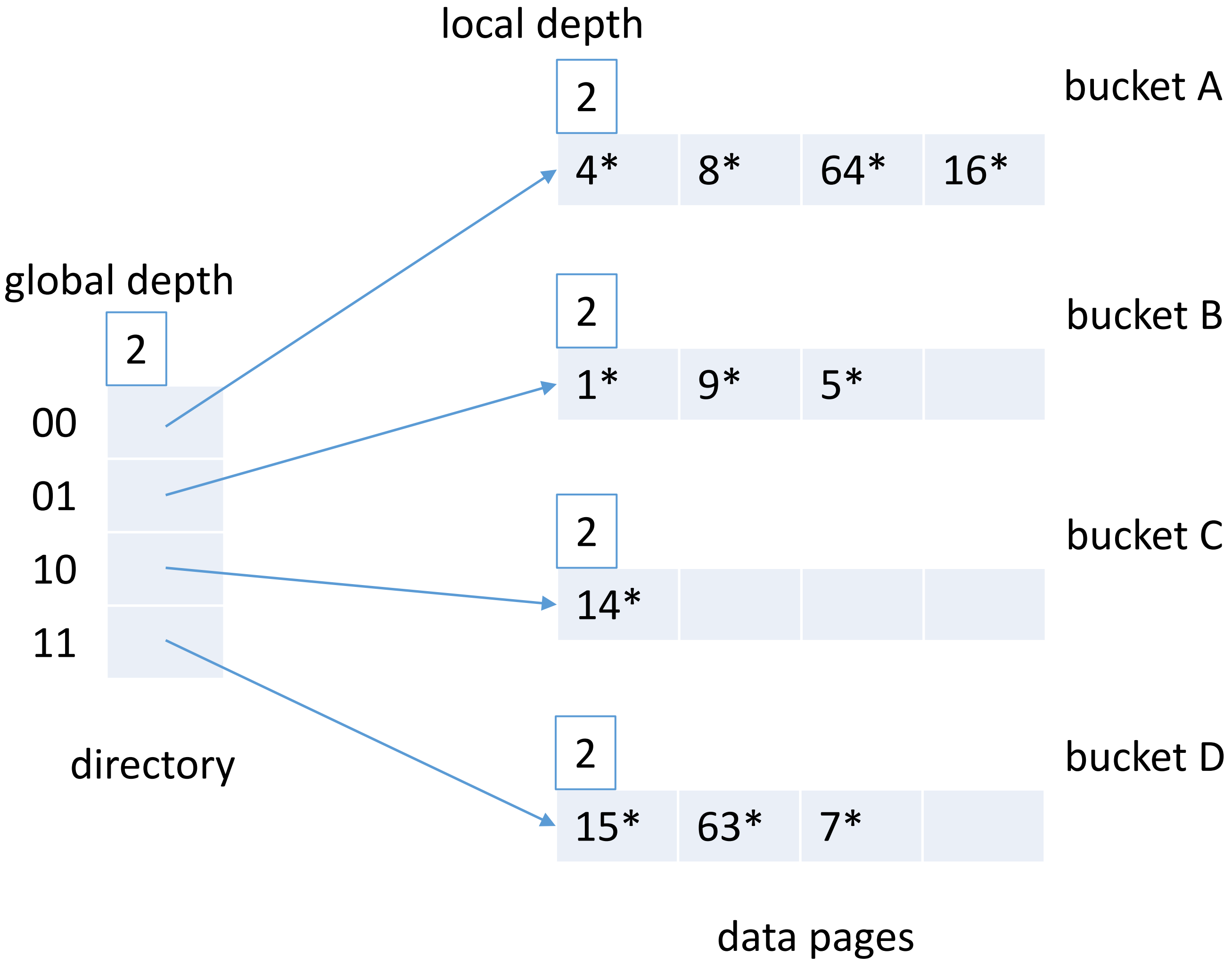
- number of buckets known when the file is created
- ideally
  - search: 1 I/O
  - insert / delete: 2 I/Os
- file grows a lot => overflow chains; long chains can significantly affect performance
  - tackle overflow chains
    - initially, pages - 80% full
    - create a new file with more buckets
- file shrinks => wasted space

* static hashing
- main problem
    - fixed number of buckets
- solutions
    - periodic rehash
    - dynamic hashing

Extendible Hashing

* extendible hashing

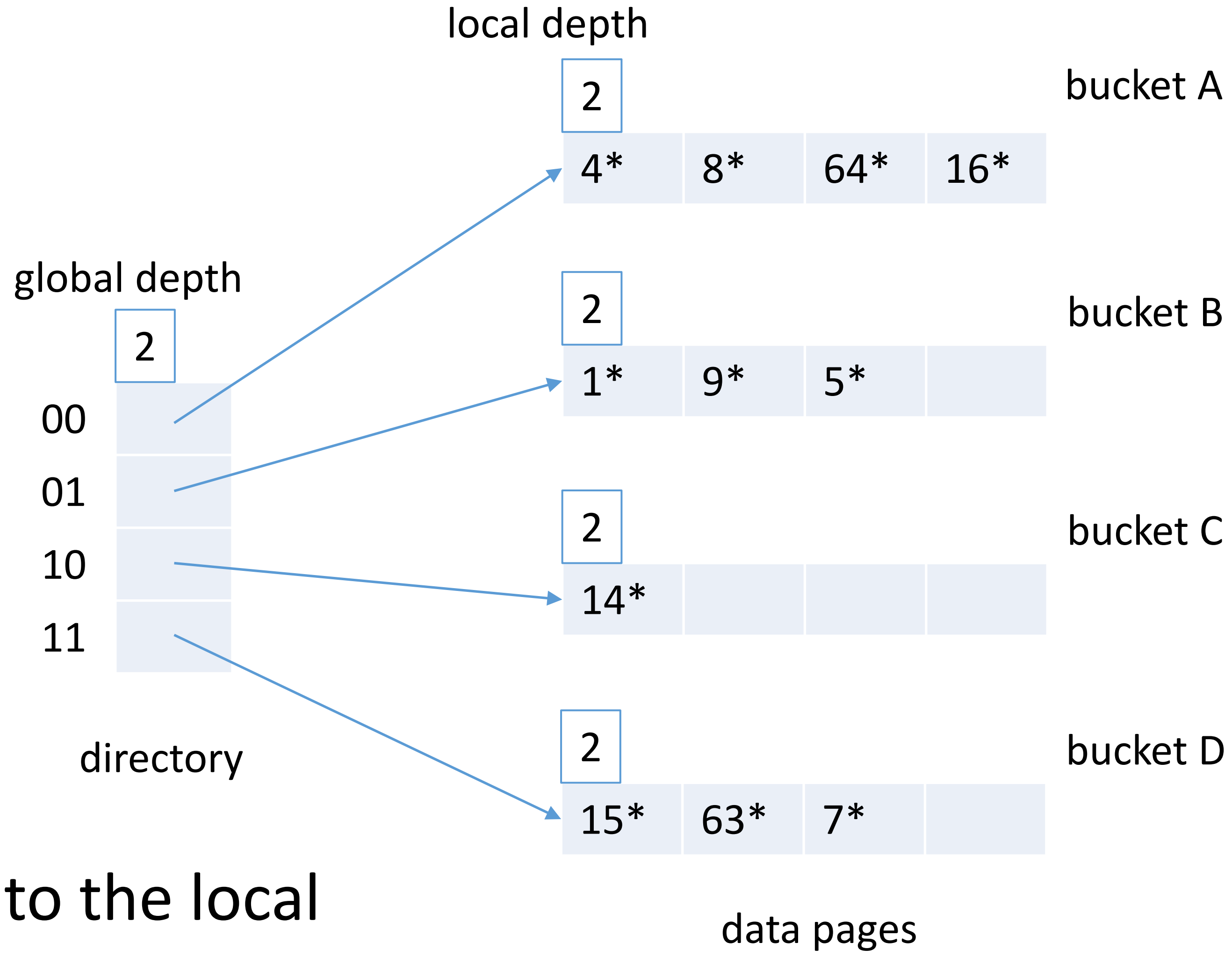
- dynamic hashing technique
- directory of pointers to buckets
- double the size of the number of buckets
    - double the directory
    - split overflowing bucket



- directory: array of 4 elements
- directory element: pointer to bucket
- entry r with key value K
- $h(K) = (\ldots a_2 a_1 a_0)_2$
- $nr = a_1 a_0$, i.e., last 2 bits in $(\ldots a_2 a_1 a_0)_2$, nr between 0 and 3
- directory[nr]: pointer to desired bucket

* extendible hashing

- global depth *gd* of hashed file
    - number of bits at the end of hashed value interpreted as an offset into the directory
    - kept in the header
    - depends on the size of the directory
        - 4 buckets => gd = 2
        - 8 buckets => gd = 3
- initially, the global depth is equal to the local depth of every bucket



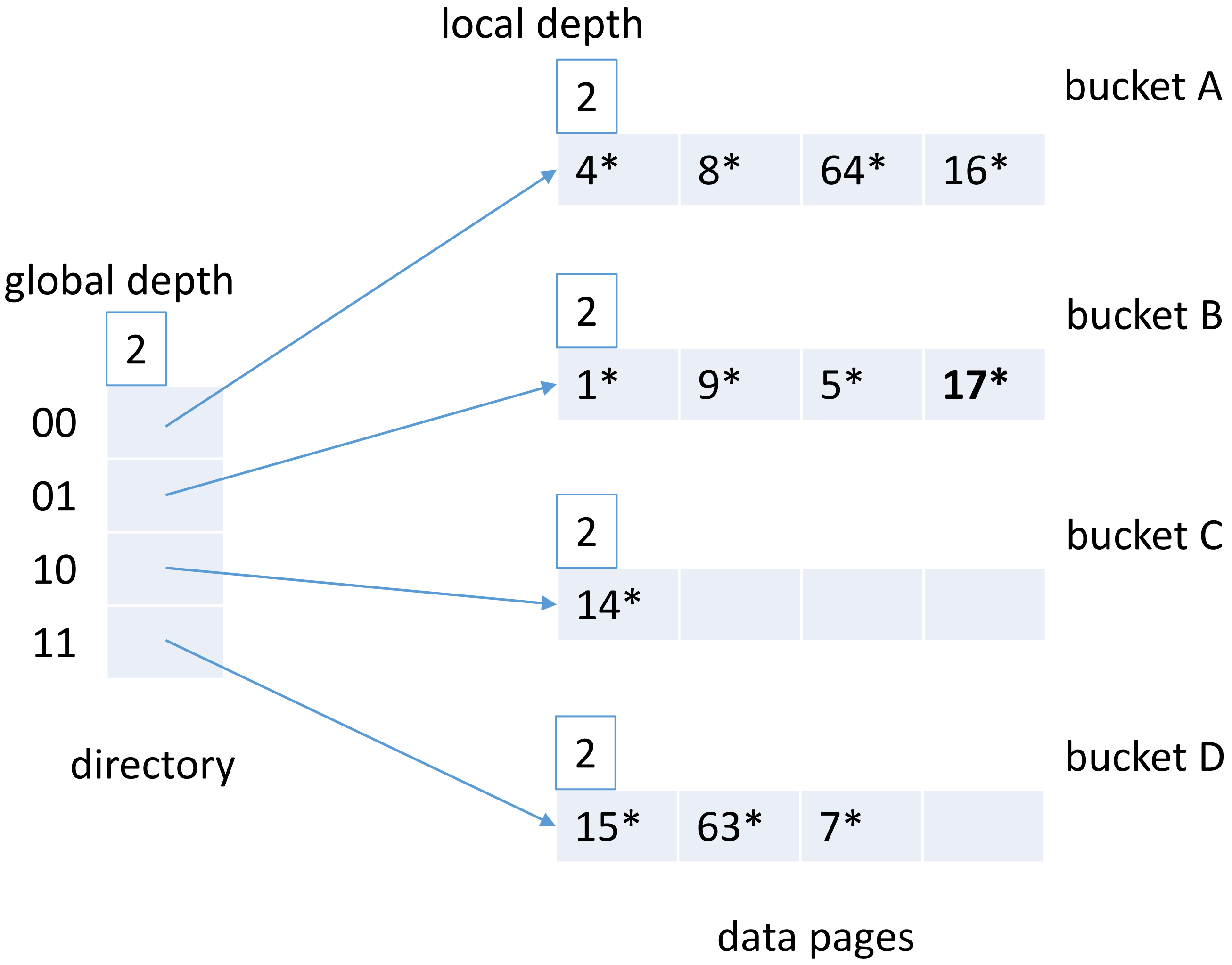
* extendible hashing

- insert entry
  - find bucket

  a. bucket has free space => the new value can be added

  - example: add data entry with hash value 17 to bucket B

obs. data entry with hash value 17 is denoted as 17*

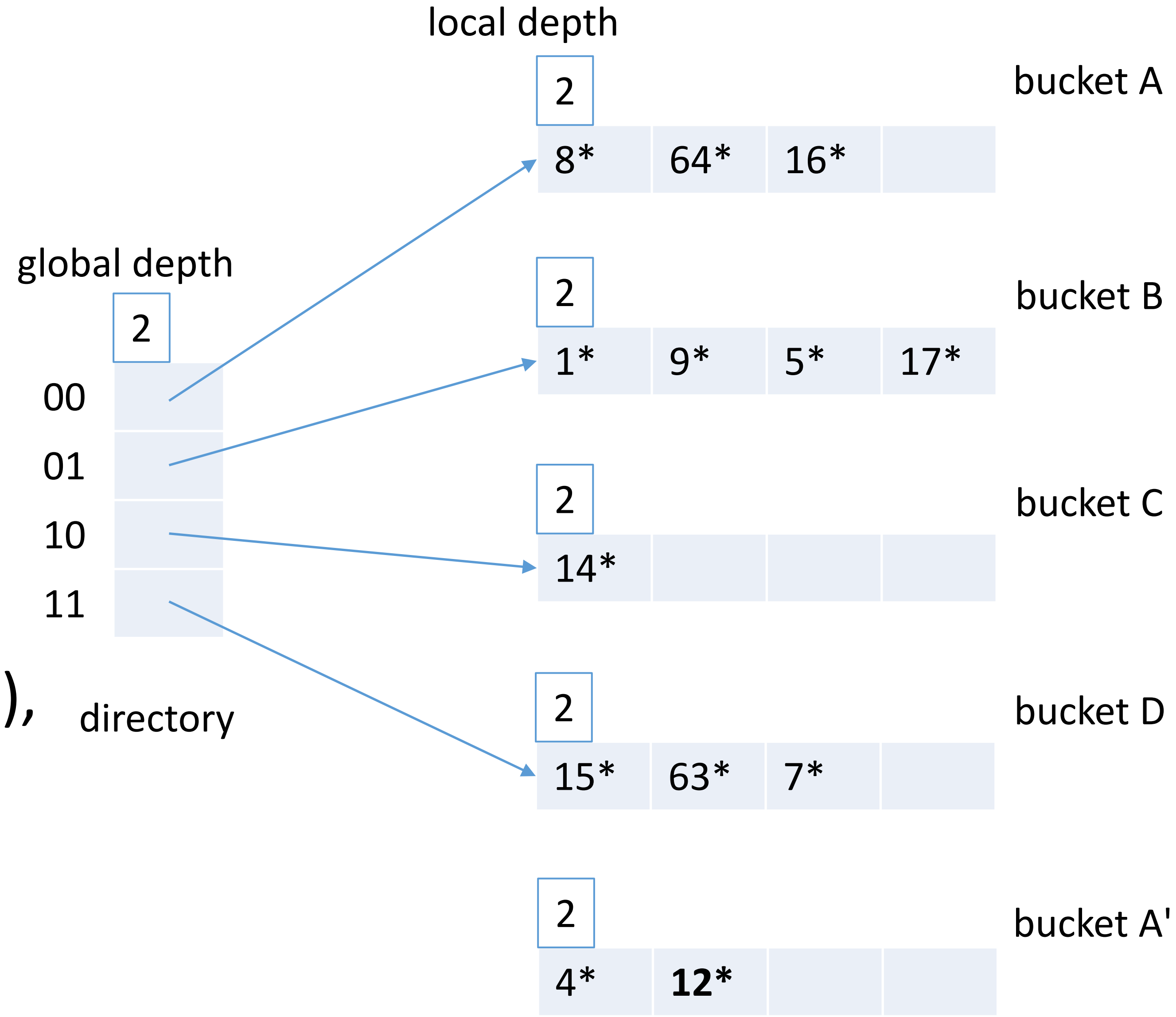


* extendible hashing

- insert entry

  b. bucket is full

  - example: add entry 12*, bucket A full
  - split bucket A
    - allocate new bucket A'
    - redistribute entries across A & A' (the split image of A), by taking into account the last 3 bits of h(K)

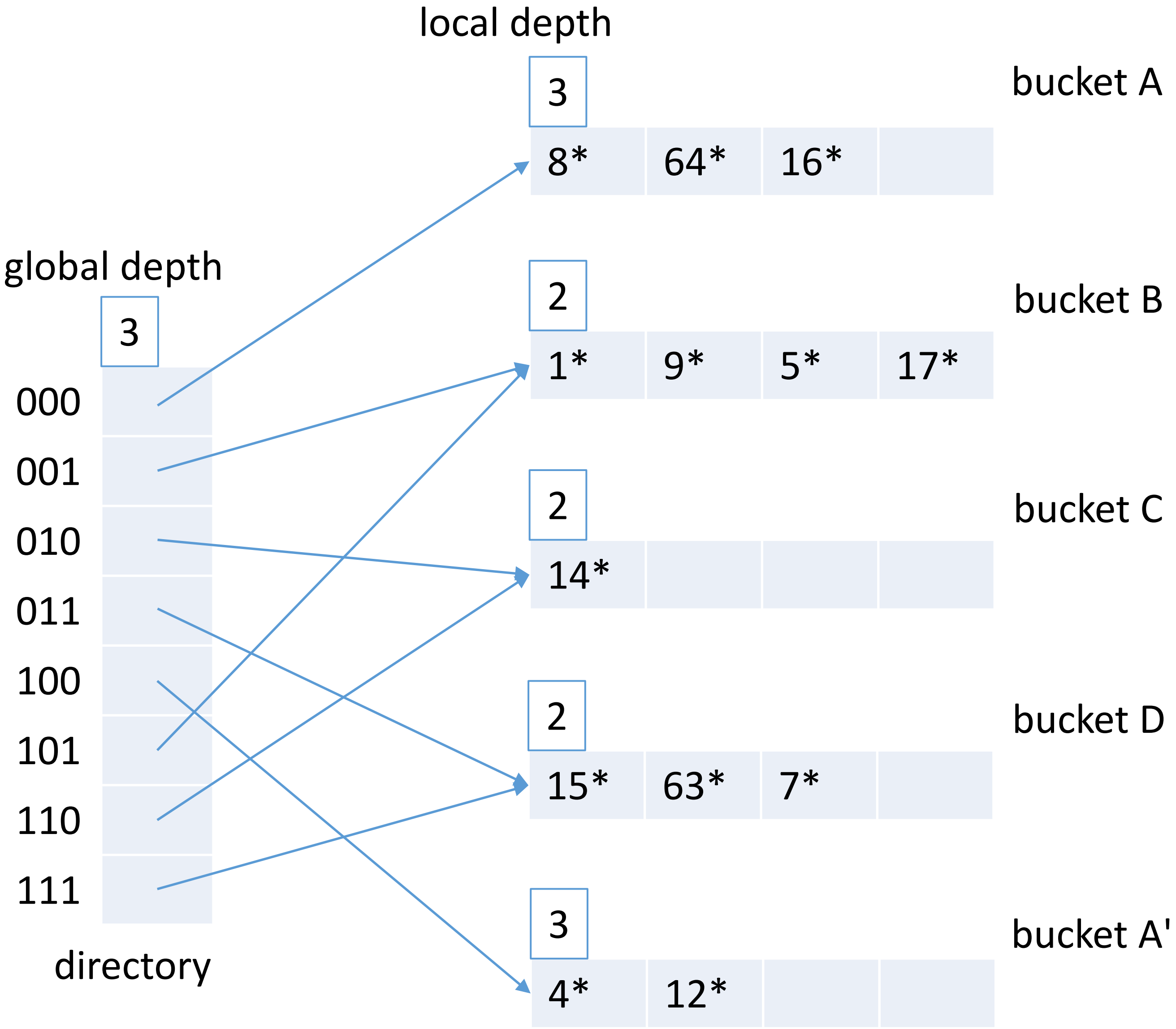


* extendible hashing

- insert entry

  b. bucket is full

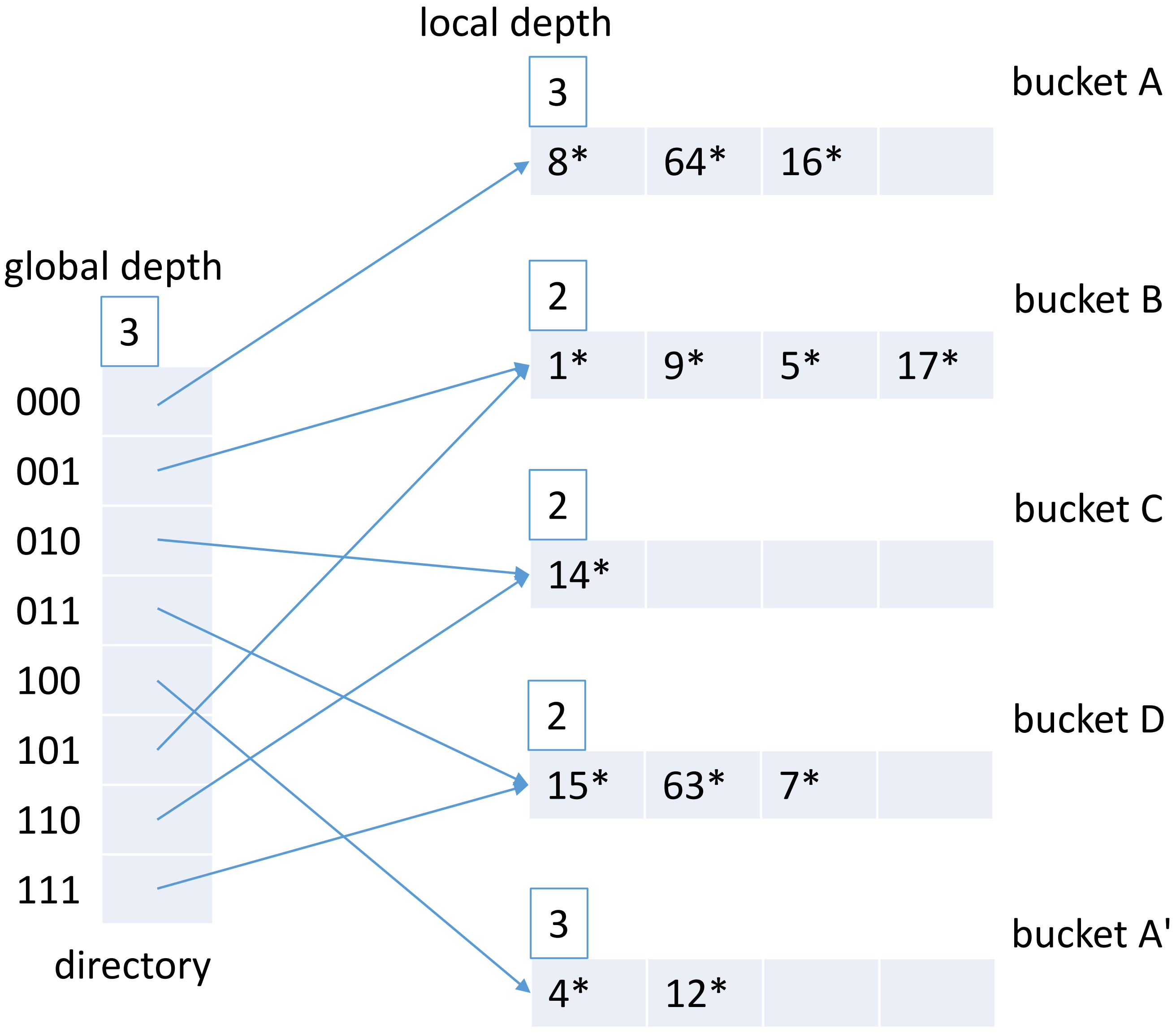
  - if gd = local depth of bucket being split => double the directory, gd++
  - 3 bits are needed to discriminate between A & A', but the directory has only enough space to store numbers that can be represented on 2 bits, so it is doubled
  - increment local depth of bucket: LD(A) = 3
  - assign new local depth to bucket's split image: LD(A') = 3



Sabina S. CS

* extendible hashing

- insert entry
  - b. bucket is full
    - *corresponding elements*
      - 000, 100
      - 001, 101
      - 010, 110
      - 011, 111
      - point to the same bucket, except for 000 and 100, which point to A and split image A', respectively

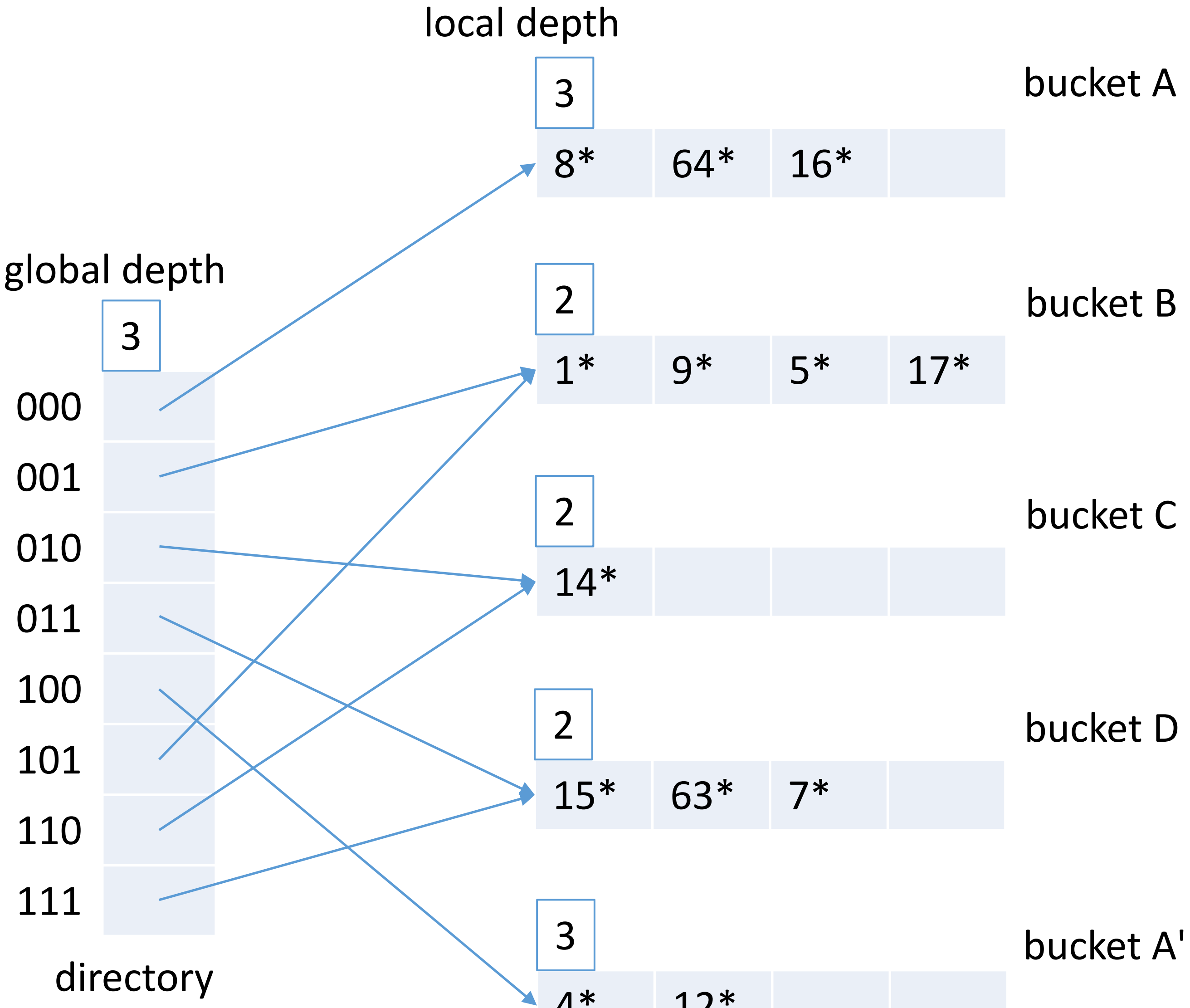


* extendible hashing

- insert entry

    b. bucket is full

    - example: add 21*
    - it belongs to bucket B, which is already full, but its local depth is 2 and gd = 3



->

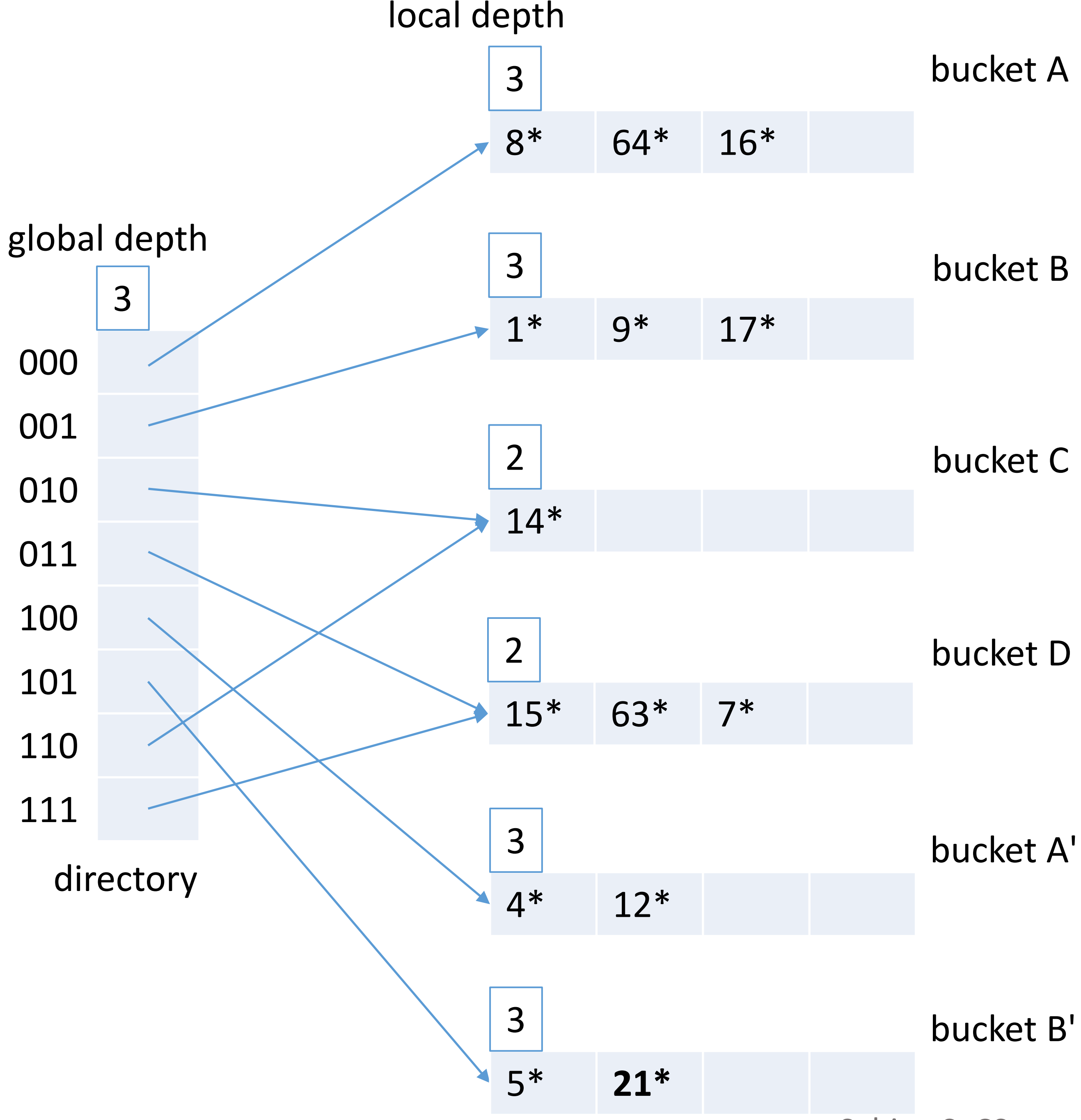
* extendible hashing

- insert entry

  b. bucket is full

  - example: add 21*
  - it belongs to bucket B, which is already full, but its local depth is 2 and gd = 3

  => split B, redistribute entries, increase local depth for B and its split image; directory isn't doubled, gd doesn't change



* extendible hashing

- search for entry with key value K_0
    - compute $h(K_0)$
    - take last gd bits to identify directory element
    - search corresponding bucket
- delete entry
    - locate & remove entry
    - if bucket is empty:
        - merge bucket with its split image, decrement local depth
    - if every directory element points to the same bucket as its split image:
        - halve the directory
        - decrement global depth

* extendible hashing

- obs 1. 2^{gd-ld} elements point to a bucket Bk with local depth ld
  - if gd=ld and bucket Bk is split => double directory
- obs 2. manage collisions - overflow pages

- double extendible hashed file
  - allocate new bucket page nBk
  - write nBk and bucket being split
  - double directory array (which should be much smaller than file, since it has 1 page-id / element)
    - if using *least significant bits* (last gd bits) => efficient operation:
      - copy directory over
      - adjust split buckets' elements

* extendible hashing

- equality selection
- if directory fits in memory:

  => 1 I/O (as for Static Hashing with no overflow chains)
- otherwise
  - 2 I/Os

- example: 100 MB file, entry = 50 bytes => 2.000.000 entries
- page size = 8 KB => approx. 160 entries / bucket

=> need 2.000.000 / 160 = 12.500 directory elements

References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009