Databases

Lecture 10

Hash-Based Indexing

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

* hash functions

- relation R – key K
- records R stored in a file

  F = $\{r_1, r_2, \ldots, r_n\}$

  $K_i = \Pi_K(r_i)$, i=1, ..., n
- algorithm to determine the answer to the query: $K = K_0$
  - sequential search
  - examine an index

* hash functions

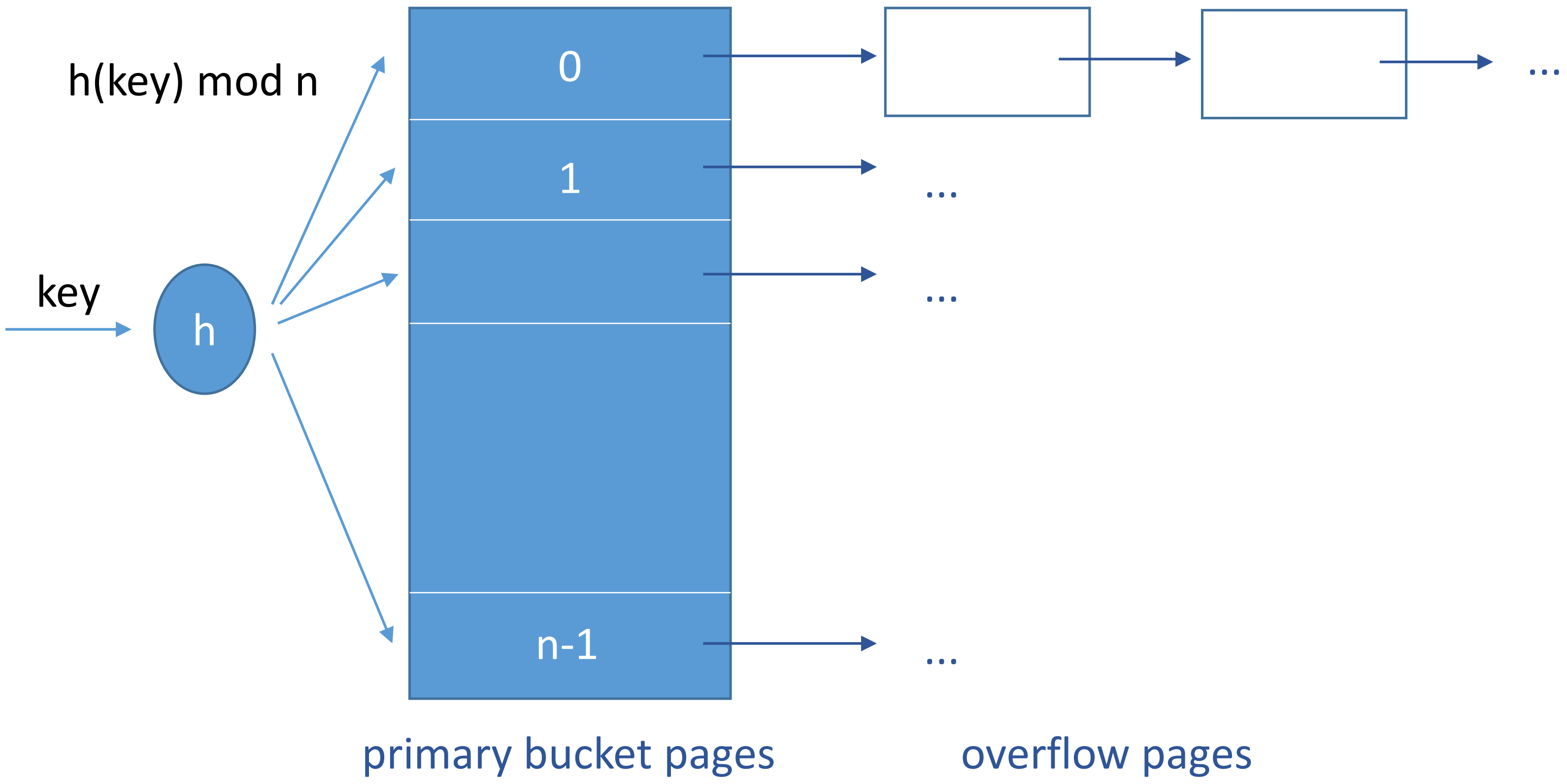
- define $h : \{K_1, K_2, ..., K_n\} \rightarrow A$,
  A = set of addresses that can store F
  let A = {0, 1, ..., m-1}, i.e., there are m locations that can store n records
- $h(K_i)$ = address where record r_i will be stored, i=1, ..., n
- injectivity requirement
  - $h(K_i) \neq h(K_j), i \neq j$
  - dynamic collections - difficult
- efficiency – collisions are allowed
  - $h(K_i) = h(K_j), i \neq j$
  - r_i and r_j – synonyms
  - $h(K_i)$ - start address for search op.

* hash functions

- functions that hash an integer value
  - division method
    - h(K) = K (mod m) => range in [0.. m-1]
    - e.g., 1618 % 89 = 16
    - prime numbers - found to work best for m
  - multiplication method
    - $\mathrm{h(K)} = [\mathrm{m} * \{\mathrm{Z} * \mathrm{K}\}]$
    - good results for $Z = \frac{\sqrt{5}-1}{2} = 0.61803$... or 1 - $Z = 0.38196$...
    - e.g., $[99 * \{0.61803 * 1618\}]$ = 96

* 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
- 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
        - $h(val) = a*val + b$
        - $h(val) \bmod n$ to identify bucket, for buckets numbered 0..n-1

* static hashing

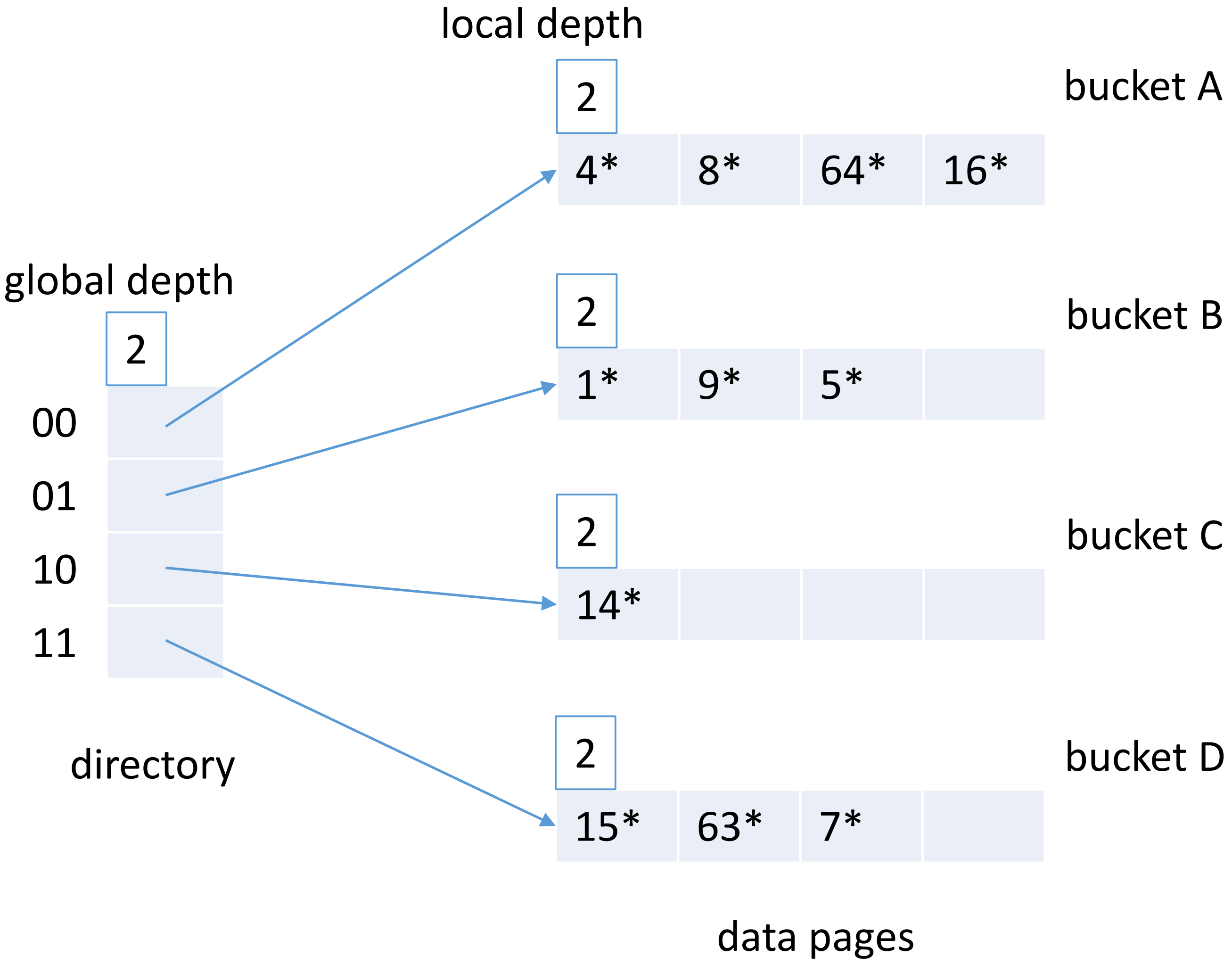
- number of buckets known when the file is created
- ideally
  - search = 1 I/O
  - I/D = 2 I/Os
- file grows a lot => overflow chains; long chains can significantly affect performance
  - tackle overflow chains
    - initially, pages - 80% full
    - create a new file with more buckets
- file shrinks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

* static hashing

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

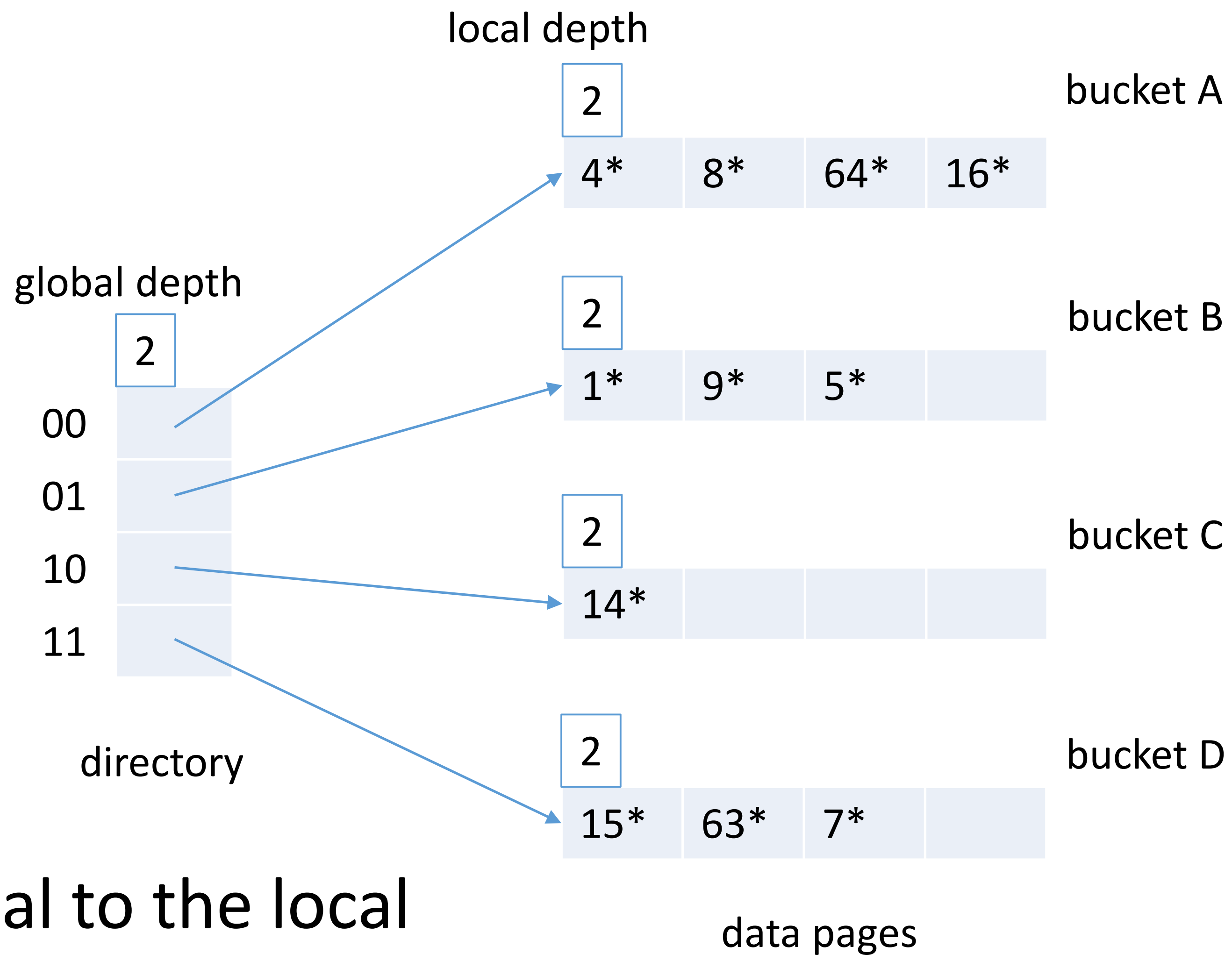
* extendible hashing

- 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
- $h(K) = (\ldots a_2a_1a_0)_2$
- $nr = a_1a_0$, i.e., last 2 bits in $(\ldots a_2a_1a_0)_2$, 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
    - e.g., 4 buckets => gd = 2
      - 8 buckets => gd = 3, etc
- initially, the global depth is equal to the local depth of every bucket

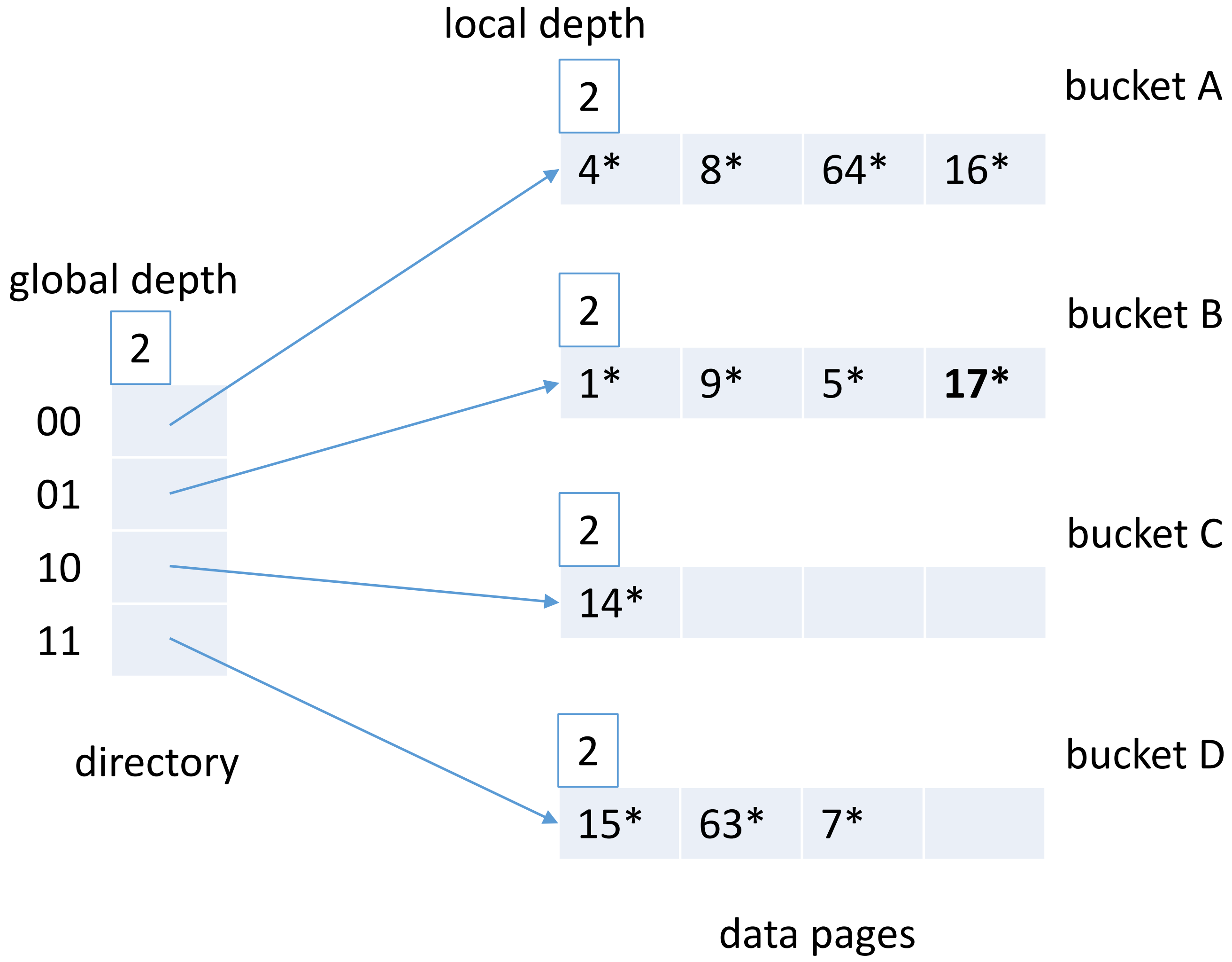


* extendible hashing

- insert entry
  - find bucket

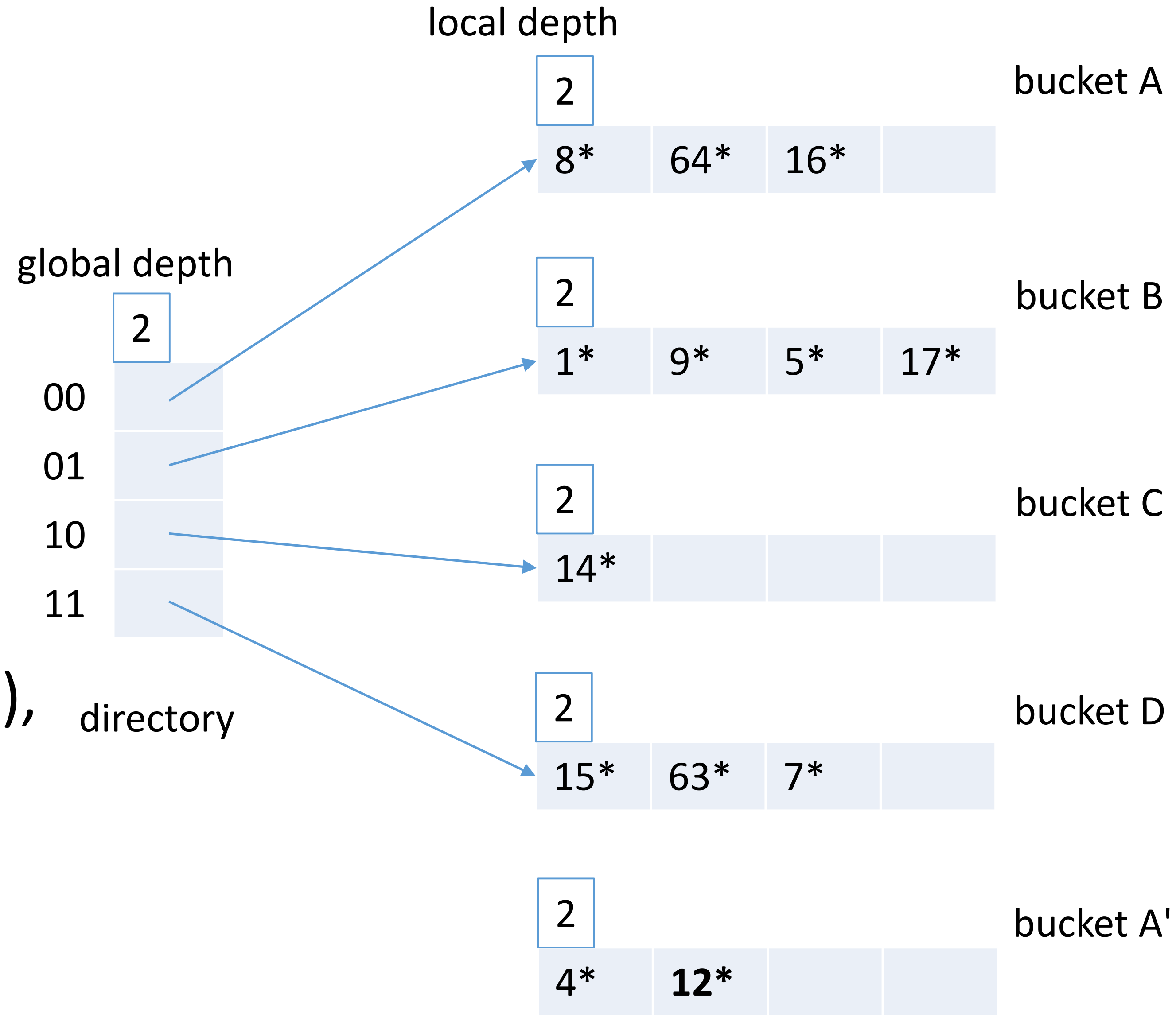
    a. bucket has free space => the new value can be added, e.g., add data entry with hash value 17 in bucket B

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



* extendible hashing

- insert entry
  - b. bucket is full
    - e.g., 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)

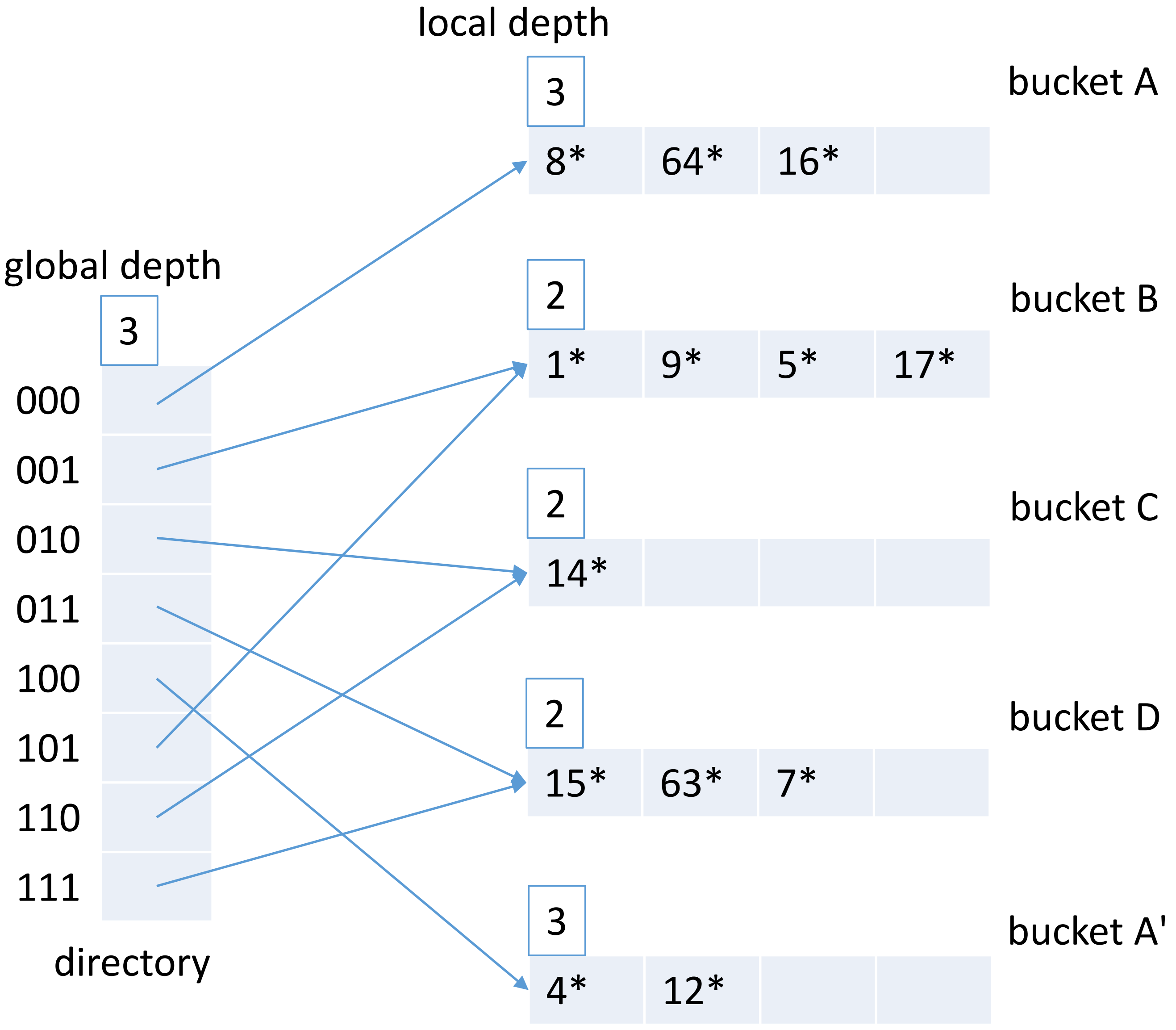


* extendible hashing

- 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

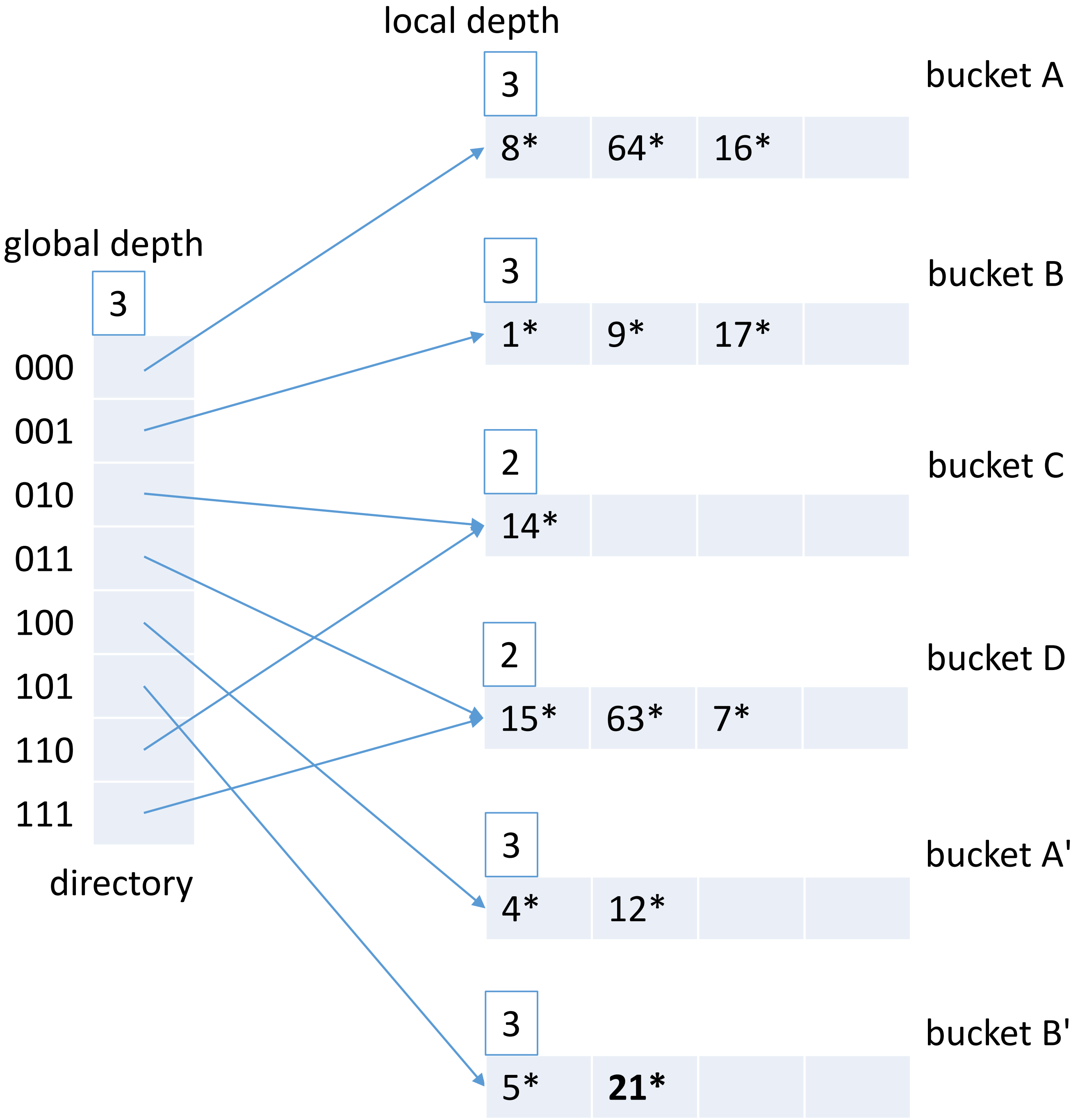
* extendible hashing

- insert entry

  b. bucket is full

  - if gd > local depth of bucket being split => directory doesn't have to be doubled
  - e.g., add 21*
  - it belongs to bucket B, which is already full, but its local depth 2 is less than 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 number of buckets in static file to avoid overflow pages
  - read entire file
  - write twice as many pages
- double extendible hashed file
  - allocate new bucket page nBk
  - write nBk and its split image
  - 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
- e.g., 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

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [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
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008