Data Organization and Processing

Bitmaps & Hashing

(NDBI007)

David Hoksza, Škoda Petr
http://siret.cz/hoksza

Overview

- Bitmaps
- Hashing
  - Definitions
  - Motivation and basic idea
  - Static hashing
  - Dynamic hashing

Bitmaps

- Indexing of attributes with **small domains**
  - sex {*M, F*}, month {*1-12*}, state {*active, inactive*}, income level {*low, medium, high*}

- For each value of the domain a vector of bits is stored telling **which objects share given property → array of bits**
  - size of the bitmap equals the number of records and each record is therefore related to exactly one position in the bit string
  - when a record has a given value the corresponding bit in the corresponding bitmap is turned on
  - **querying** using **bitwise logical operations**

Bitmaps – example

Employees (gender)									
	Janice	Michael	Sharon	David	Kevin	John	Mary	Terry	Jill
Male	0	1	0	1	1	1	0	1	0
Female	1	0	1	0	0	0	1	0	1

Employees (department)									
	Janice	Michael	Sharon	David	Kevin	John	Mary	Terry	Jill
Design	0	1	1	0	0	1	1	1	0
Research	0	0	0	1	1	0	1	0	1
Marketing	1	0	0	0	0	1	0	0	0

Bitmaps queries (1)

- One attribute queries
  - Which employees work in research?

Employees (department)									
	Janice	Michael	Sharon	David	Kevin	John	Mary	Terry	Jill
Design	0	1	1	0	0	1	1	1	0
Research	0	0	0	1	1	0	1	0	1
Marketing	1	0	0	0	0	1	0	0	0

  - Which employees work in both design and marketing? (x AND y)

Employees (department)									
	Janice	Michael	Sharon	David	Kevin	John	Mary	Terry	Jill
Design	0	1	1	0	0	1	1	1	0
Research	0	0	0	1	1	0	1	0	1
Marketing	1	0	0	0	0	1	0	0	0

  - Which employees work in research and not in design? (x AND NOT y)

Employees (department)									
	Janice	Michael	Sharon	David	Kevin	John	Mary	Terry	Jill
Design	0	1	1	0	0	1	1	1	0
Research	0	0	0	1	1	0	1	0	1
Marketing	1	0	0	0	0	1	0	0	0

Bitmaps queries (2)

- Multi attribute queries
  - Which female employees work in research?
- Basically any boolean queries (AND, OR, NOT) can be effectively answered using bitmap indexes

Employees (department)									
	Janice	Michael	Sharon	David	Kevin	John	Mary	Terry	Jill
Design	0	1	1	0	0	1	1	1	0
Research	0	0	0	1	1	0	1	0	1
Marketing	1	0	0	0	0	1	0	0	0

Employees (gender)									
	Janice	Michael	Sharon	David	Kevin	John	Mary	Terry	Jill
Male	0	1	0	1	1	1	0	1	0
Female	1	0	1	0	0	0	1	0	1

Bitmaps – space complexity

- A file with **2^{19} (524,288) records → bitmap** for one value has **64 KiB (2^{16} B)**
- If the **domain** of the bitmap attribute has **10 values** then only **640 KiB** are needed to store the index
- If size of the **record** is **1KB** then the size of the **primary file** is about **524 MB** (in comparison to 640 KB of the index)
- Bitmap indexes
  - grow linearly with the database size
  - can be read by large blocks
  - can be compressed

Hashing

- **Hashing** is a technique capable of accessing a record in **(external) memory** in **O(1)** time **by using hash functions to map search keys onto (physical or logical) addresses**
  - keys mapped into **buckets** *(kapsy)*
- Also known as
  - direct accessing, randomizing
- **Hash function** is a mapping from the **query space** to the **address space**
  - the query space is the space of all possible values of the query key
  - we can consider relative address space only (absolute addresses can be easily calculated)
- **function h: $K^* \rightarrow \{0, 1, \ldots M-1\}$**
  - the interval $\{0, 1, \ldots M-1\}$ is called **address space**
  - $h(k)$ determines the address of a record with a key k

Hash function requirements

- Hash function should be
  - **uniform**
    - each bucket should contain keys from all parts of the key space
  - **random**
    - each bucket should be equally filled regardless of the key values distribution
    - the worst hash function would map all the search keys onto the same address → the search deteriorates into sequential scan
      - the result should be dependent on all bits of the key
  - **deterministic**
    - the resulting value is dependent only on the input values
  - **fast**
    - it should take only few instructions to compute the resulting value of the hash function
  - …

Typical hash functions (1)

- **Trivial**
  - the **numerical representation of the key** represents the relative (or absolute) **address**
  - advantages
    - fast
    - is perfect
  - disadvantages
    - usable only for relatively small domains
    - commonly not uniform, neither random
- examples
  - 32-bit integer values – such key can directly represent the bucket index
  - **short strings** such as codes of countries
    - 26 letters → 3-letter codes can be uniquely mapped into $26^3 = 17576$-long array
    - generally if a domain consists of **fixed-length sequences of size L** of attributes with a small **domains of size D** (e.g., short words, zip code, …) then it can be used to **address D^L objects**

Typical hash functions (2)

- **Modulo**
  - $\boldsymbol{h(k) = k \bmod M}$
  - M is advised to be a prime number
  - highly dependent on the value of M
    - $M = 16 \rightarrow$ bins are $0000 \ldots 1111$
      - value of $h(k)$ is dependent solely on the 4 **low-order** (least important) **bits** of the key → these bits can be poorly distributed which can lead to poor distribution of the results → lots of collisions
  - $\boldsymbol{h(k) = (a \times k + b) \bmod M}$
    - gives good results for $a \bmod M \neq 0$
- **Binning**
  - $\boldsymbol{h(k) = k/M}$
  - if we have domain range and *M=10* $\boldsymbol{< 0; 1000 >}$ then values $\boldsymbol{< 0; 99 >}$ will go to the first slot
  - can be seen as an inverse to modulo since it looks at the high-order bits
  - dependent on the high-order bits → if the distribution of the high-order bits is poorly distributed so will be the results

Typical hash functions (3)

- **Mid-Square**
  - squares the key value, and then takes the middle $\boldsymbol{r}$ bits of the result, giving a value in the range $< \mathbf{0}; \mathbf{2}^{\boldsymbol{r}-\mathbf{1}} >$
  - good to use with integers
  - is not dependent on the distribution of low- or high-order bits – all bits contribute to the final value (try to compute 4567*4567 by hand to verify)

- $r = 2, k = 4567 \rightarrow 4567^2 = 20857489 \rightarrow h(k) = 57$

$$
\begin{array}{r}
4567 \\
4567 \\
\hline
31969 \\
27402 \\
22835 \\
18268 \\
\hline
20857489
\end{array}
$$

Internal vs. external hashing

Internal hashing

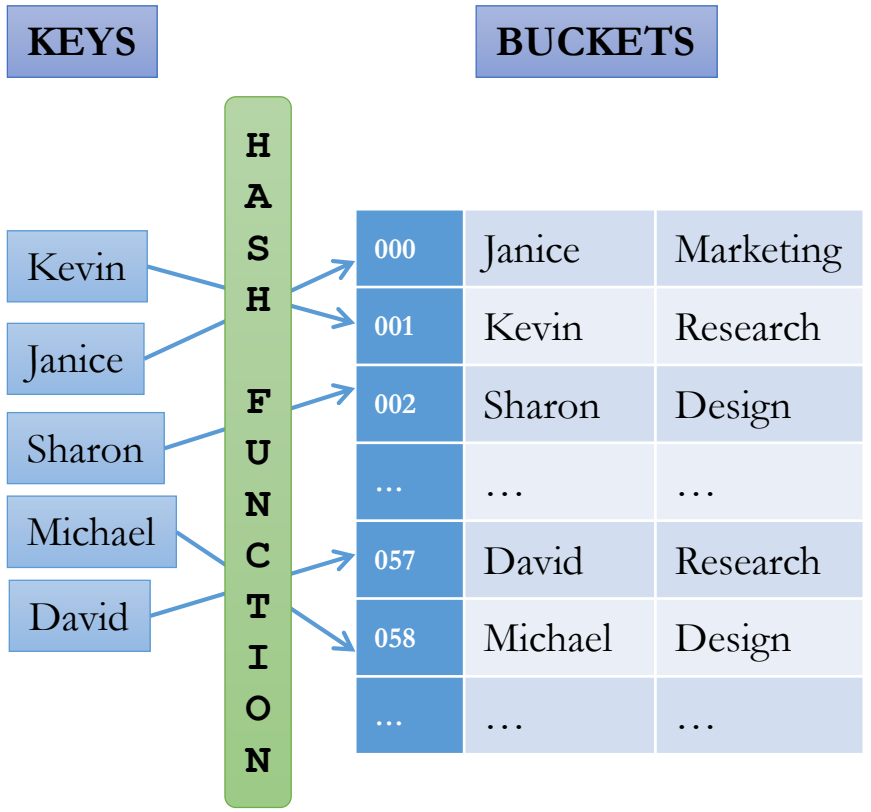
- In-memory hashing
  - Hashing structure fits in main memory
  - Each bucket contains one record
  - Limited space
- Algorithms differ in collision handling
  - separate chaining
  - open addressing
  - cuckoo hashing

External hashing

- Hashing structure does not fit into the main memory
- Efficiency counted in number of accessed blocks
  - Each block can accommodate given number of records
- Algorithms
  - static
    - Cormack, Larson & Kalja
  - dynamic
    - Fagin, Linear hashing (Litwin), Spiral storage

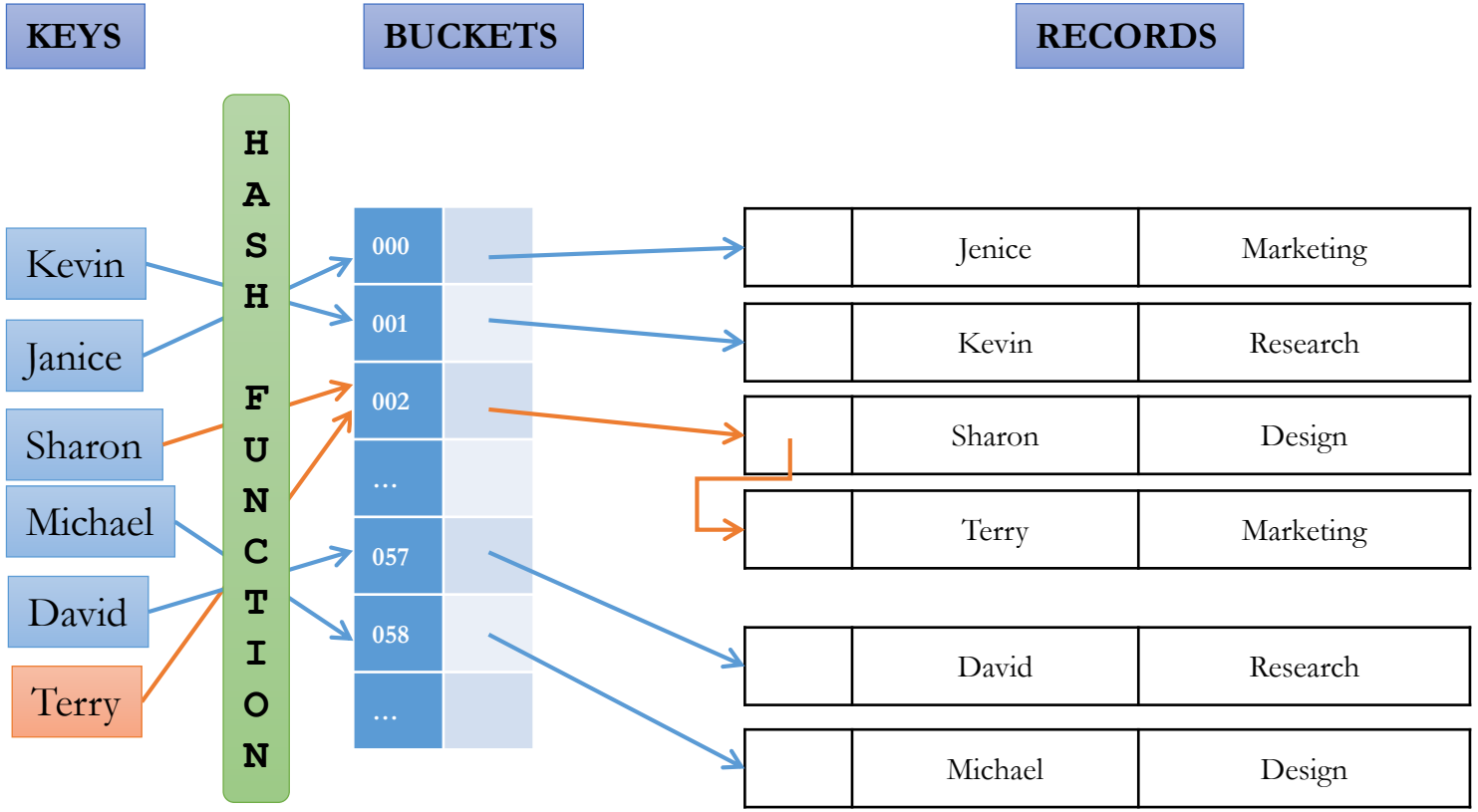
Hash tables/maps

- In-memory data structure
- Basically **associative array**
- Hash table utilizes a **hash function** (map) to **match** the **keys** with their **associated values**
  - ideally one key – one slot
    - if **multiple keys** are mapped to the **same position** → **collision**
- Hash tables vary in **collision handling**
  - $O(1)$ search complexity without collisions
  - separate chaining/hashing (*izolované řetězení*)
  - open addressing (*otevřené adresování*)
  - coalesced chaining/hashing (*spojované řetězení*)
  - cuckoo hashing (*kukačkové hashování*)



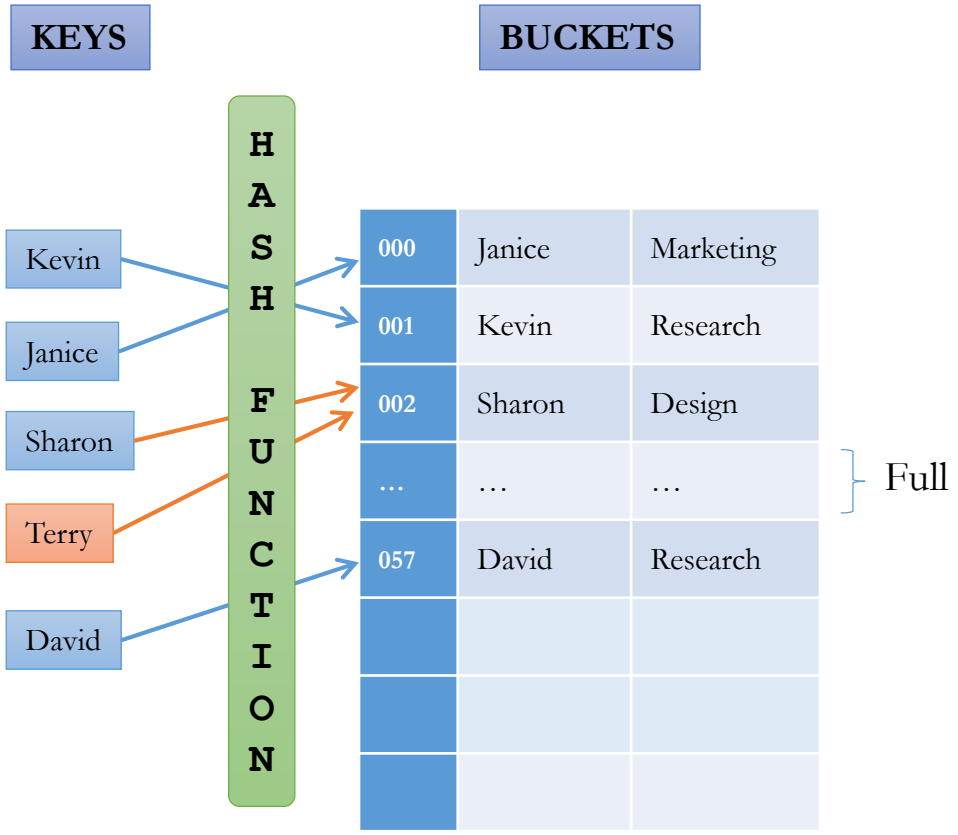
Separate chaining

- Buckets contain **links to chains** of collided records



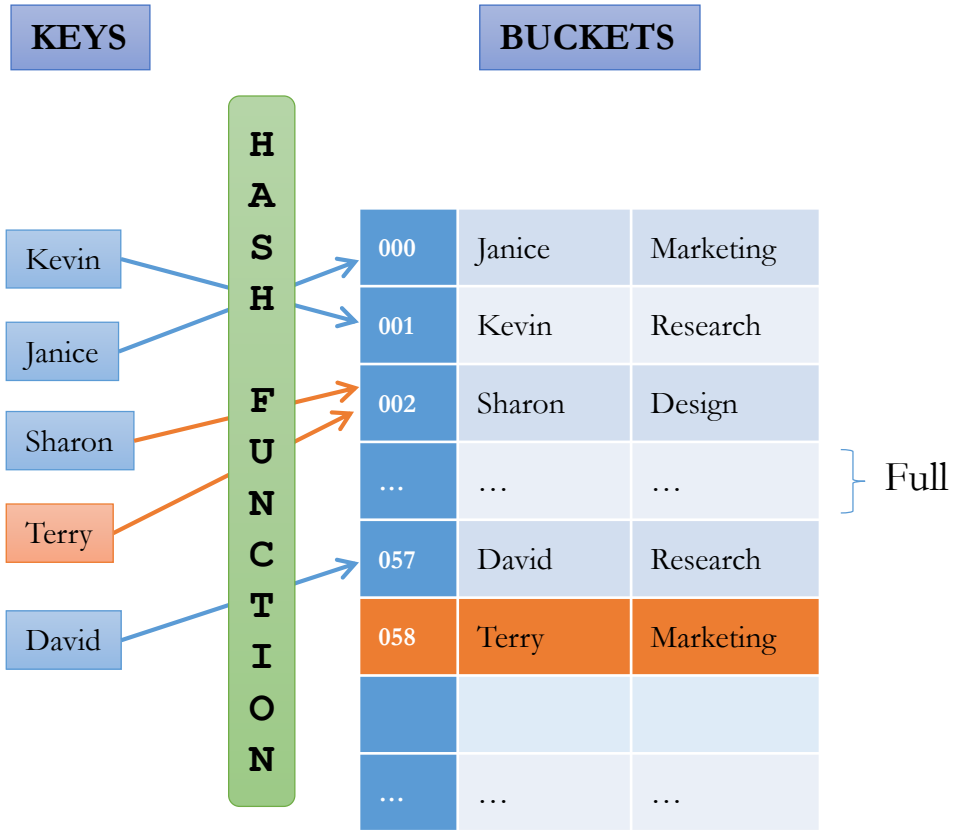
Open addressing

- Collided record is inserted into the **next free bucket** (basic version)
- Searching for a record with key ***K***
  1. compute the **address *A*** from the query key *K* using the hash function
  2. if **no record** is present **at *A*** the searched record is **not in the table**
  3. otherwise **scan** (see the following slides) the table until either record with key ***K* is found (record found)** or an **empty slot** is encountered **(record not present)**



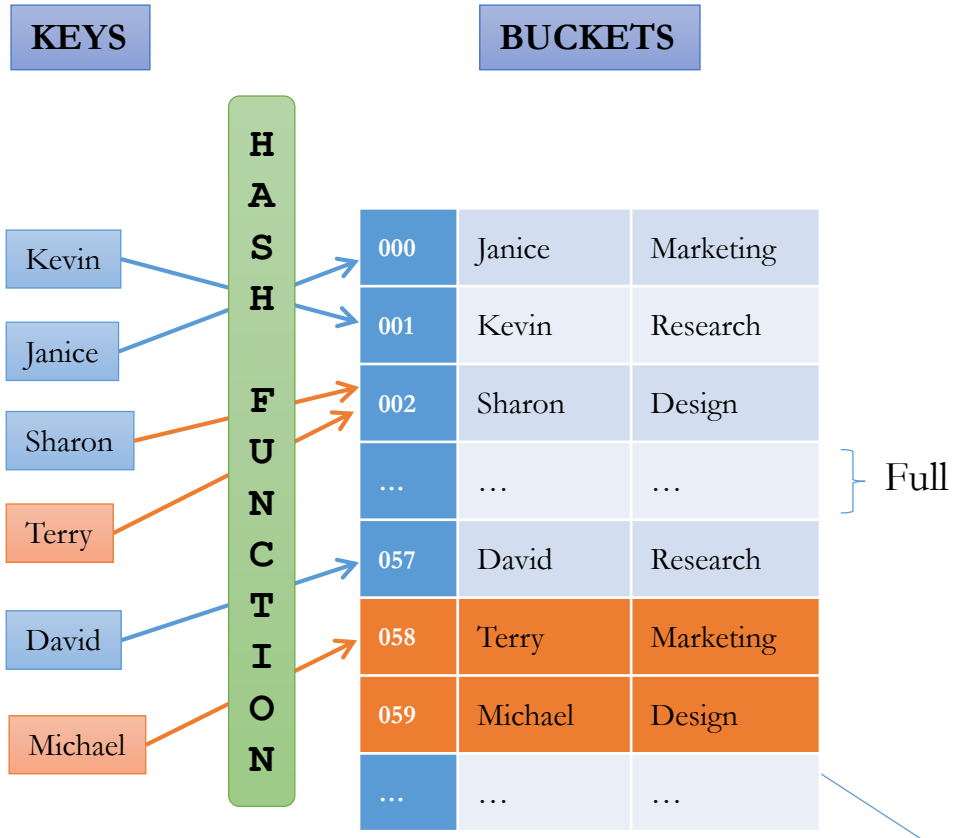
Open addressing

- Collided record is inserted into the **next free bucket** (basic version)
- Searching for a record with key ***K***
  1. compute the **address *A*** from the query key *K* using the hash function
  2. if **no record** is present **at *A*** the searched record is **not in the table**
  3. otherwise **scan** (see the following slides) the table until either record with key ***K* is found (record found)** or an **empty slot** is encountered **(record not present)**



Open addressing

- Collided record is inserted into the **next free bucket** (basic version)
- Searching for a record with key ***K***
  1. compute the **address *A*** from the query key *K* using the hash function
  2. if **no record** is present **at *A*** the searched record is **not in the table**
  3. otherwise **scan** (see the following slides) the table until either record with key ***K* is found (record found)** or an **empty slot** is encountered **(record not present)**



Although Michael didn't collide, now he collides with Terry who was placed at address 058 (first free address after its original target address 002). Naturally, this holds only when Michael is inserted only after Terry (otherwise, Terry would be at 059).

Open addressing – collision resolution (1)

- Where to search for the next free bucket if the home bucket is occupied?
  - **probe sequence (*sondovací sekvence*)** generated by a **probe function (*sondovací funkce*)**

```
void hashInsert(const Key& k, const Record& r)
{
  int home;                          // Home position for k
  int pos = home = h(k);             // Init probe sequence
  int i = 0;
  while (HT[pos].key() != EMPTYKEY){
    i++
    pos = (home + p(k, i)) % M;   // probe function
    if (k == HT[pos].key()) {
      cout << "Duplicates not allowed\n";
      return;
    }
  }
  HT[pos] = r;
}
```

- The function should also keep track of whether it did not get into a cycle
- Search function implementation follows the same principle as the insert function presented here

Open addressing – collision resolution (2)

- **Primary clustering**
  - when sequentially scanning for a **next free slot, the probe sequences can collide** and thus cause **clustering**
  - **optimal probe function** should provide each slot with an **equal probability** of receiving a record
    - not true for the scanning probe function
    - **a free slot after a long chain** has a **high probability of receiving** a record
    - moreover, small clusters tend to chain, forming big clusters
- **Linear probing**
  - i-th probe: $\boldsymbol{p(k,i) = c_1 i}$
  - if c_1 is 2 and M is even then the hash table will be effectively divided into two disjoint sets → possible performance degradation when one group is favored by the hash function
  - $\boldsymbol{c}$ **and** $\boldsymbol{M}$ **should share no factors**($c = 1,3,7,9 \; - \; M = 10$)
  - **probe chains can collide** and thus contribute to clustering, e.g., $\mathrm{c} = 2, k_1 = 3, k_2 = 5$

Open addressing – collision resolution (3)

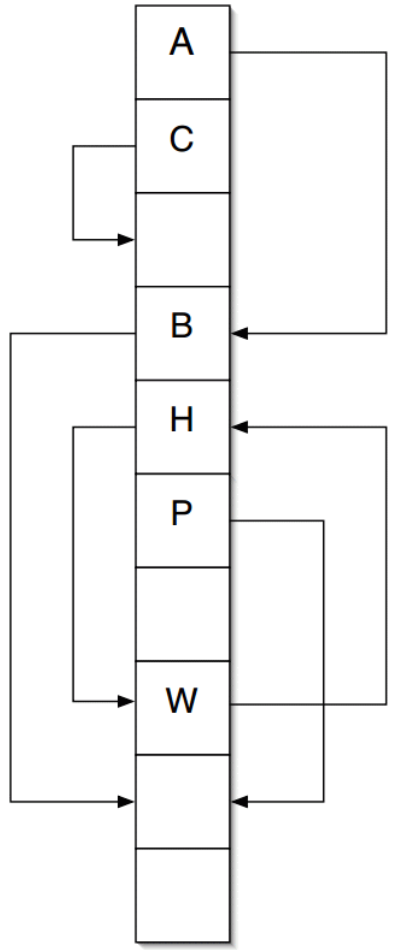
- **Quadratic probing**
  - i-th probe: $\boldsymbol{p(k,i) = (c_1 i + c_2 i^2)}$
  - **wrong choice of constants** can prevent from visiting every slot
    - $c_1 = 0$, $\boldsymbol{c_2 = 1}$, M = 105
      - length of any probe = 23
  - there exists a **fitting choice** of the constants, e.g.:
    - $\boldsymbol{c_1 = 0}$, $\boldsymbol{c_2 = 1}$, **M = prime number**
      - at least half slots will be visited
    - $\boldsymbol{c_1 = 1/2}$, $\boldsymbol{c_2 = 1/2}$, **M = power of 2**
      - every slot will be visited
- **(Pseudo-)random probing**
  - $perm$ … a table with permutations of length M
  - i-th probe: $\boldsymbol{p(k,i) = perm[i]}$
- **Double hashing**
  - i-th probe: $\boldsymbol{p(k,i) = i * h_2(k)}$
  - the **probe sequence** is now **different for different keys**

Coalesced Chaining

- Combines separate chaining and open addressing
  - when a collision occurs, the last position in the chain is fetched and connected to the first free bucket (e.g., first free bucket from the end of the table)
  - **collided records are chained** to decrease the retrieval time (for both insert and query operations)
    - two chains never merge (as probe sequences can)

Cuckoo Hashing

- Pagh & Rodler, 2001
- **Two hash functions h_1, h_2**
  - **no overflow chains or scanning of the hash table**
  - if $T[h_1(k)]$ is full, insert the record anyway and kick the residing record into its alternative location $T[h_2(k')]$
  - lookup needs inspection of 2 positions at most
  - often implemented by 2 tables each having its own hash function

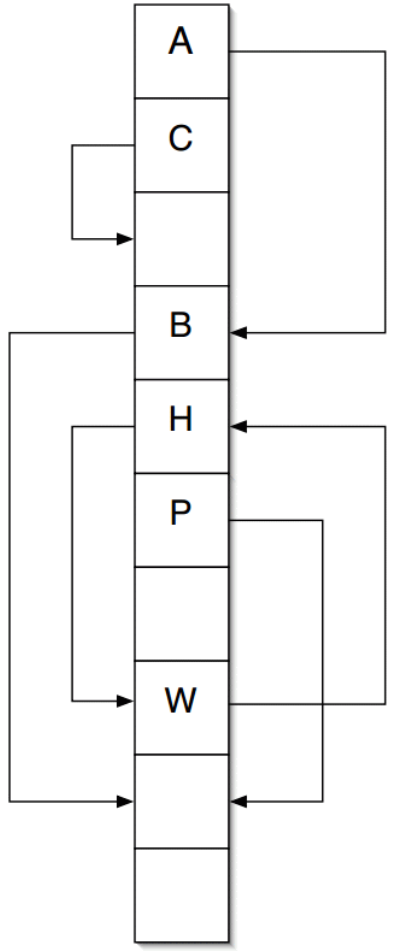



```
procedure insert(x)
  if T[h1(x)] = x or T[h2(x)] = x then return;
  pos ← h1(x);
  loop n times {
    if T[pos]= NULL then {T[pos] ← x; return; }
    x ⟷ T[pos]; //insert into T and refill x with the content of the current position
    if pos = h1(x) then pos ← h2(x) else pos ← h1(x);
  }
  rehash(); insert(x);
```

Cuckoo Graph

- Insert Z
  - $h_1(Z) = h_i(W), h_2(Z) = h_j(A)$

- The graph shows the insertion "chain"
  - $Z \rightarrow W$
  - $W \rightarrow H$
  - $H \rightarrow Z$
  - $Z \rightarrow A$
  - $A \rightarrow B$
  - $B \rightarrow empty$



External Hashing

Static

- Hash function maps keys into a **fixed number of addresses/pages**
- Static hashing allows to add records, but not to extend the address space without the need to rebuild the whole index
- Good for more or less **static databases**

Dynamic

- Allows **dynamic growth of the address space** based on the size of the database
- Allows the hash function to be modified dynamically
- Good for **databases that grow and shrink in size**

Cormack

- **Perfect hashing**
  - → no overflow policy required
- Requires
  - additional O(N) space (**directory**)
  - Requires a **set of hash functions**
    - one function is used as the **initial hash function**
    - **another function** is used **to hash collided records** into **continuous space**

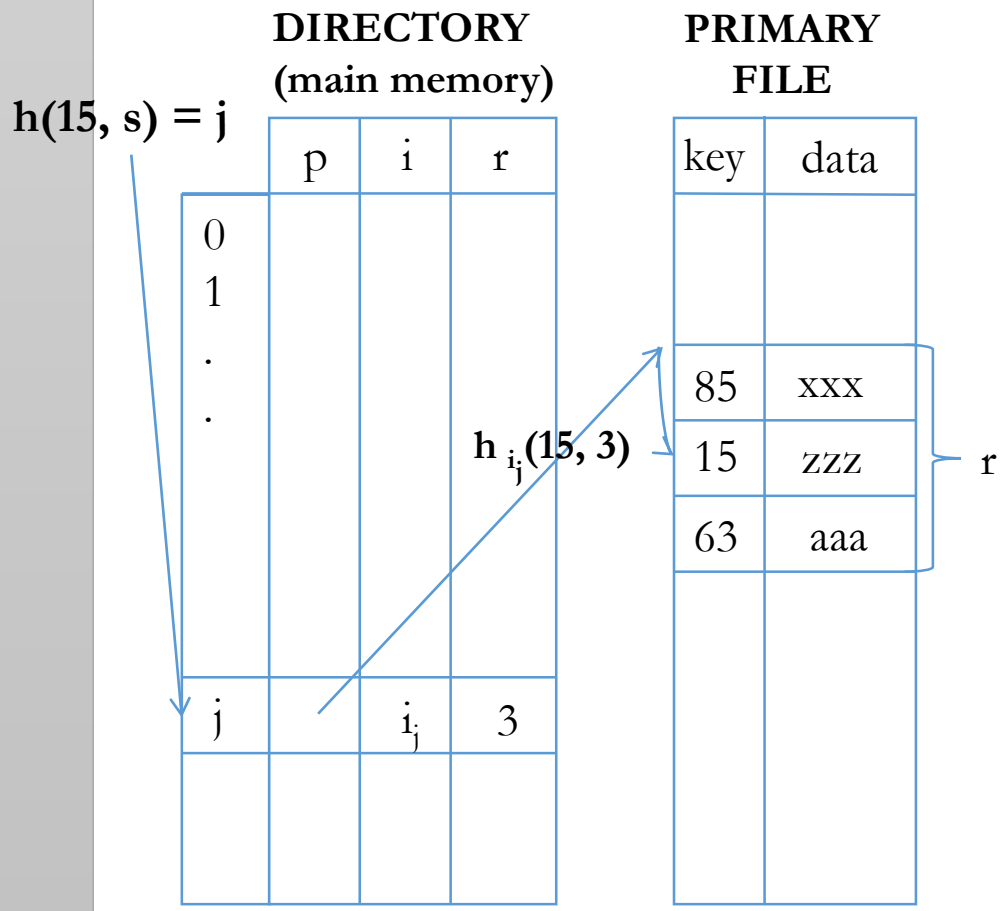
- **Presumes low number of collisions** in which case it is possible **to find a perfect hashing function** for the collided records in a **reasonable time**

Cormack - ACCESS

```
typedef struct {
  int pos, i, r;
} dLine;
```

```
typedef struct{
  KEY_TYPE key;
  DATA data;
} pfLine;
```

```
void ACCESS(dir[], pfLine pf[], KEY_TYPE k, int
&pfPos, bool &found){
  int s = dir::size();
  int j = h(k,s);
  if (dir[j].r == 0) found = false;
  else {
    int t = dir[j].i;
    pfPos = dir[j].pos + h_t(k, dir[j].r);
    if (pf[pfPos].key != k) found = FALSE;
    else found = TRUE;
 }}
```



Cormack - INSERT

```
bool INSERT(dLine dir[], pfLine pf[], pfLine rec)
{
  int s = dir::size();
  int j = h(rec.key,s);
  if (dir[j].r == 0)
  {
    int posNew = FREE(pf, 1);
    pf[posNew] = rec;
    dir[j].p = y; dir[j].i = -1; dir[j].r = 1;
  }
  else
  {
    int r = dir[j].r, p = dir[j].p;
    if (CONTAINS(pf, p, r, rec.key)) return FALSE;
    /*find a hash function with index m, such that h_m(rec.key, r+1) != h_m(pf[p].key, r+1) != h_m(pf[p+1].key, r+1) != … != h_m(pf[p+r-1].key, r+1)*/
    int m = FIND_HASH_F(pf, p, r, rec.key);
    int posNew = FREE(r+1); //allocate space for colliding records and the new record
    for (int i = 0; i < r; i++)
    {//copy the existing colliding records into new space
      pf[posNew+h_m(pf[p+i].key, r+1)] = pf[p+i];
      ERASE(pf, p+i); //frees memory
    }
    dir[j].p = newPos; dir[j].i = m; dir[j].r = r+1;
  }
}
```

Demo

Larson & Kalja (1)

- **Perfect hashing** introduced in 1984 by Larson and Kalja
- Uses **two sets of hash functions**
  - $h_i(k), i \in \{1, \ldots, M\}$ generates a sequence of page addresses where a record with a key k could be inserted
  - $s_i(k), i \in \{1, \ldots, M\}$ generates a sequence of d-bit long strings called **signatures**
  - pages have assigned d-bit long strings called **separators** and comparison of a page signature and the record signature for that page determines whether the record can be found or inserted there

Larson & Kalja (2)

- A record with a key k can be inserted into a page $h_i(k)$ only if $s_i(k)$ is smaller than the separator of that page
- Records are sorted in the page in increasing values of their signatures
- Page signature is the lowest signature of all the records which could not fit into that page (overflown records)
- The initial value of the separator is 2^{d-1}
  - signatures cannot take this value
  - until the first overflow of page p, every record can be inserted in p
- During the INSERT operations, more records can be pushed out of the page → INSERT cascade

Larson & Kalja - example

- Insert a record with key '*ab*'
- $h_1(ab) = 10, h_2(ab) = 46,$
- $s_1(ab) = 1011, s_2(ab) = 0101$
- $b = 3$

10	46	95	116
od-0100	ef-0100	kl-0100	op-0010
	gh-1000	mn-1001	
	ij-1000		
sep: 1000	sep: 1001	sep: 1111	sep: 1000

- The target page for *ab* is therefore 46 → pushes out records with keys *gh* and *ij* → new page separator
- $h_j(gh) = 46, h_{j+1}(gh) = 95, s_j(gh) = 1000, s_{j+1}(gh) = 1011$
- $h_j(ij) = 46, h_{j+1}(ij) = 116, s_j(ij) = 1000, s_{j+1}(ij) = 0100$

10	46	95	116
od-0100	ef-0100	kl-0100	op-0010
	ab-0101	mn-1001	**ij-0100**
		gh-1011	
sep: 1000	sep: **1000**	sep: 1111	sep: 1000

Larson & Kalja - ACCESS

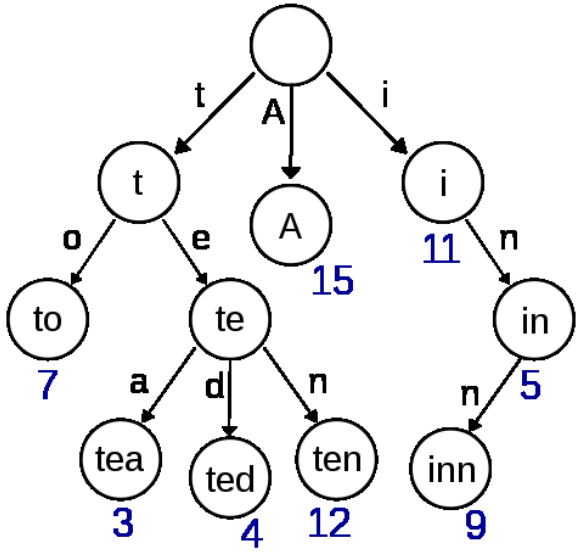
```
void ACCESS(int sep[], KEY_TYPE k, PAGE_TYPE &page, bool &found)
{
  int m = sep::size();
  for (int i = 0; i < m; i++)
  {
    int adr = h_i(k), sign = s_i(k);
    if (sign < sep[adr])
    {
      GET_PAGE(adr, page);
      found = SEARCH_PAGE(page, k);
      return;
    }
  }
  found = FALSE;
}
```

Demo

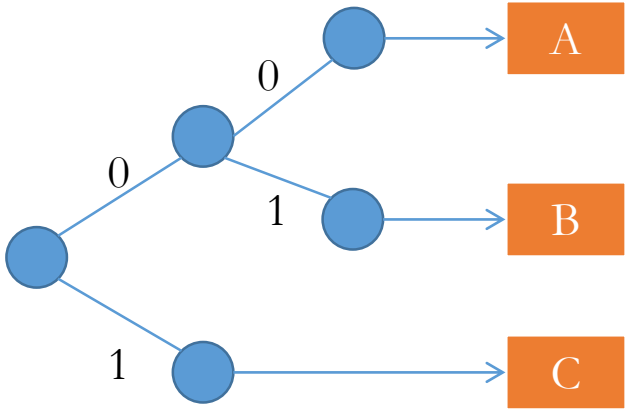
Dynamic (extendible) hashing

- Static hashing structures or standard hashing table structure have a **fixed maximum size**
- **Chaining methods lose** the expected **constant-time** operations
- **Maximum size limitations should be avoided** while retaining the advantages of constant-time find, insert, and delete operations

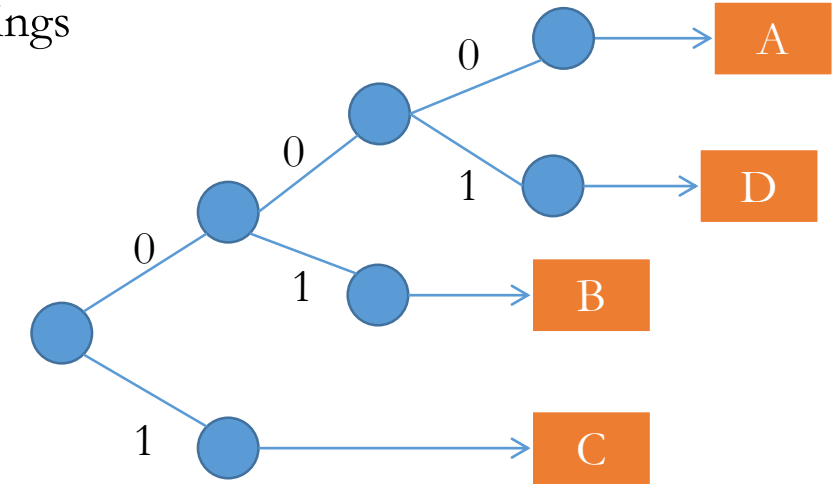
Tries as the basis for dynamic hashing



- **Trie = prefix tree**
  - **branching** pattern determined **not by the entire key** but only **by pa** of it
  - all the **descendants** of a node have a **common prefix**
  - we will work with **binary number/string keys**
    - other key types can be converted to binary strings



more records (A overflows)



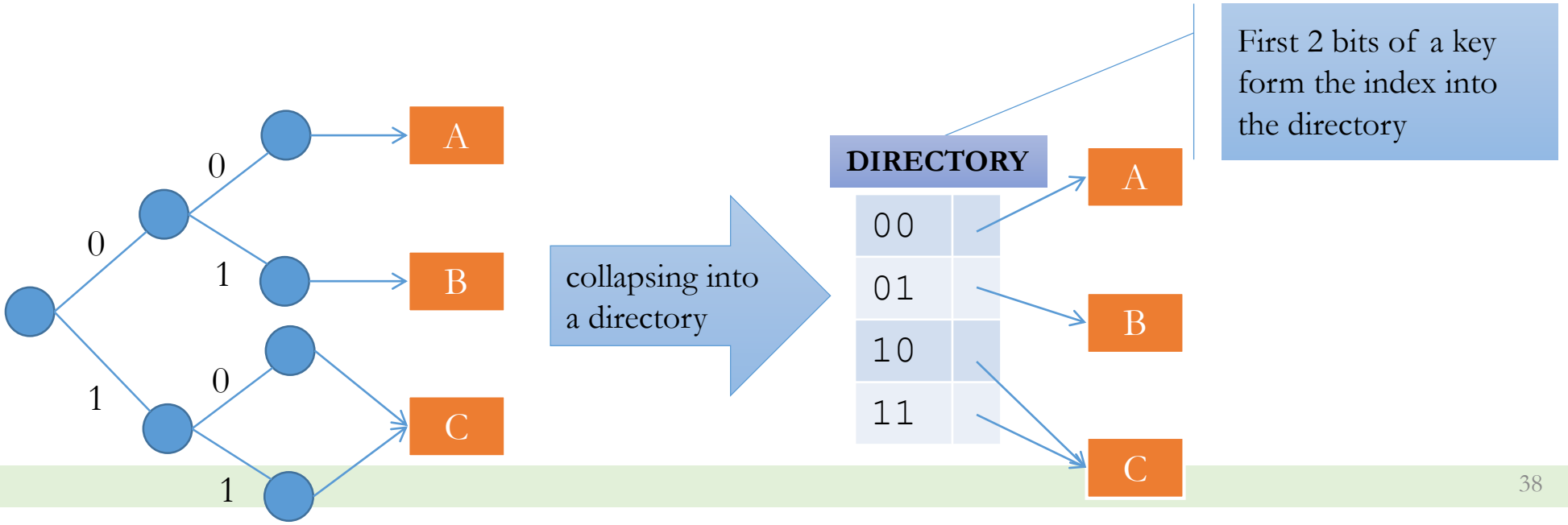
A … records with (hashed) keys starting with 00
B … records with (hashed) keys starting with 01
C … records with (hashed) keys starting with 11

A … records with (hashed) keys starting with 000
D … records with (hashed) keys starting with 001
B … records with (hashed) keys starting with 01
C … records with (hashed) keys starting with 11

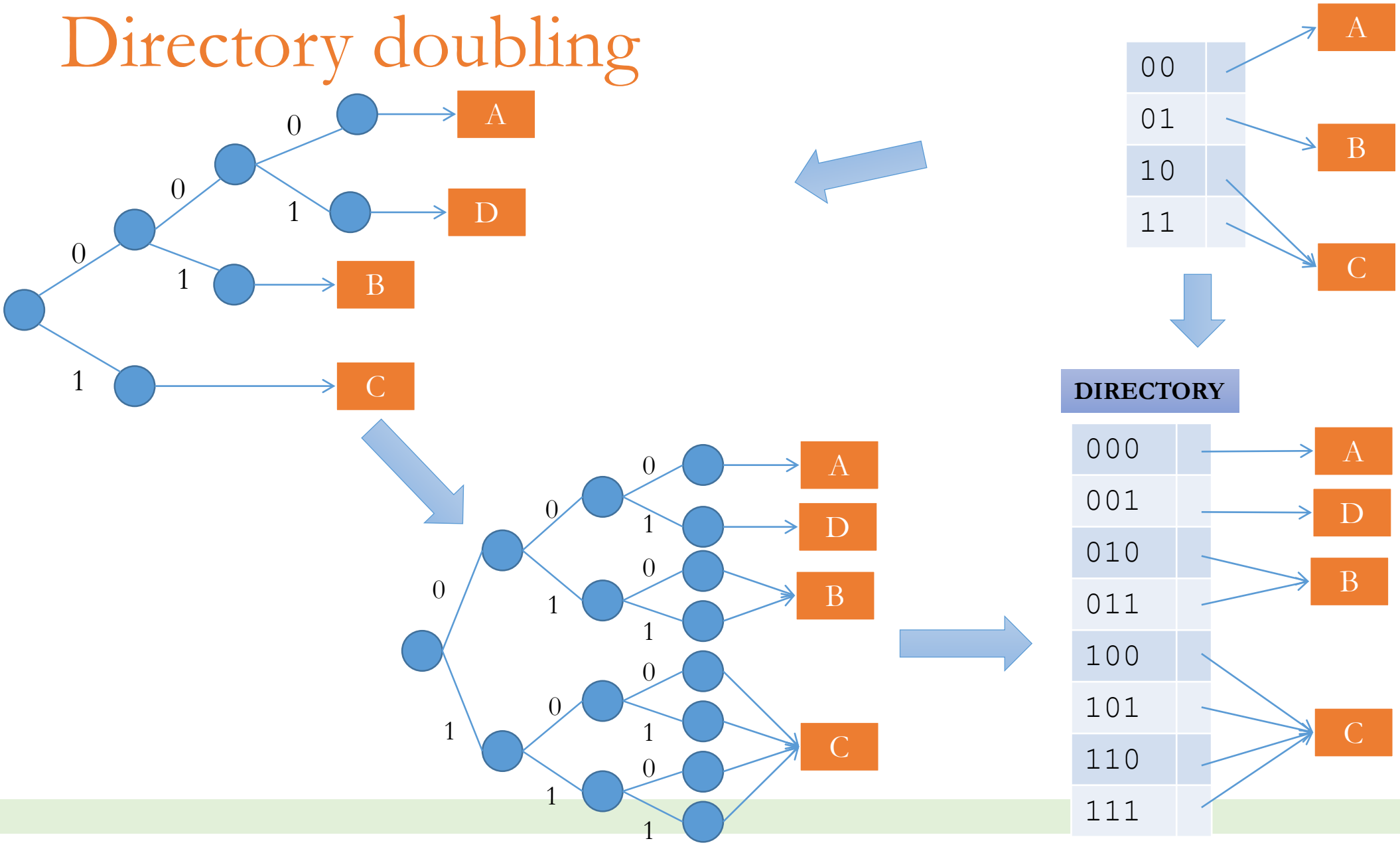
Collapsing a trie into a directory

- Possible problems with tries
  - longer access times in case of a skewed key distribution
  - balancing trie by choosing different order of bits
    - building minimum depth trie is NP-complete

- **Collapsing trie**
  1. **shortening of a trie by collapsing it into a directory**
     - decreasing search time
  2. **accessing the directory using a hash function**
     - uniform hash function ensures a balanced trie

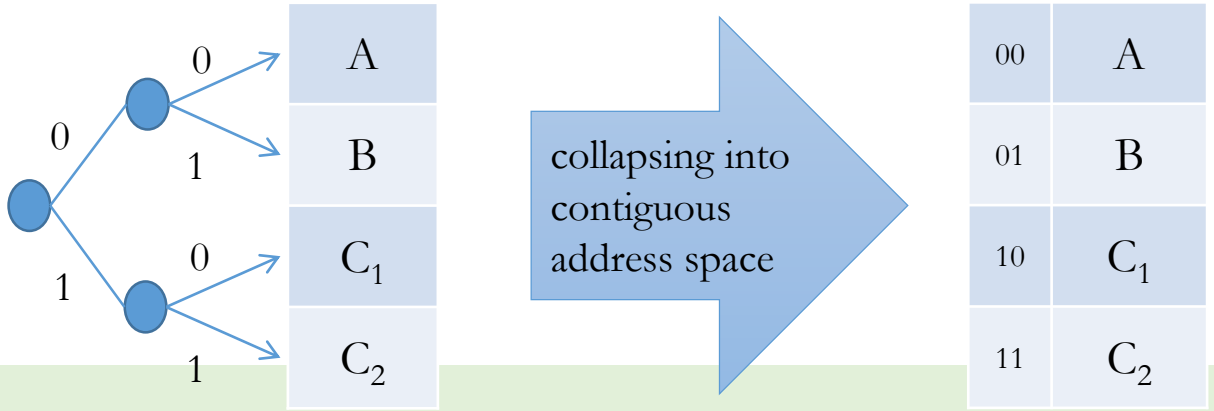


Directory doubling



Collapsing a trie without a directory

- Directory introduces a level of indirection in the addressing → **collapsing** the trie **without a directory** → directory-less schemes
  - maintaining **pages in a contiguous address space**
  - **search path** in the trie forms the **address**
    - that is, first few bits of the key represent the address where the record is going to be stored
  - decreases utilization of the pages
  - **only doubling** the directory is possible → **overflow of a page causes doubling of the address space** size and redistributing records based on the bit prefixes (suffixes) of their keys



Hashing schemes

Directory Schemes

- Pages can be scattered in the address space
- High utilization
- Can allow overflows

Directoryless schemes

- Address of the page determined by the hash function directly
- Can show poor utilization
- Must allow overflows
  - pages split in a predefined order and a page can be full earlier than it is its turn to split

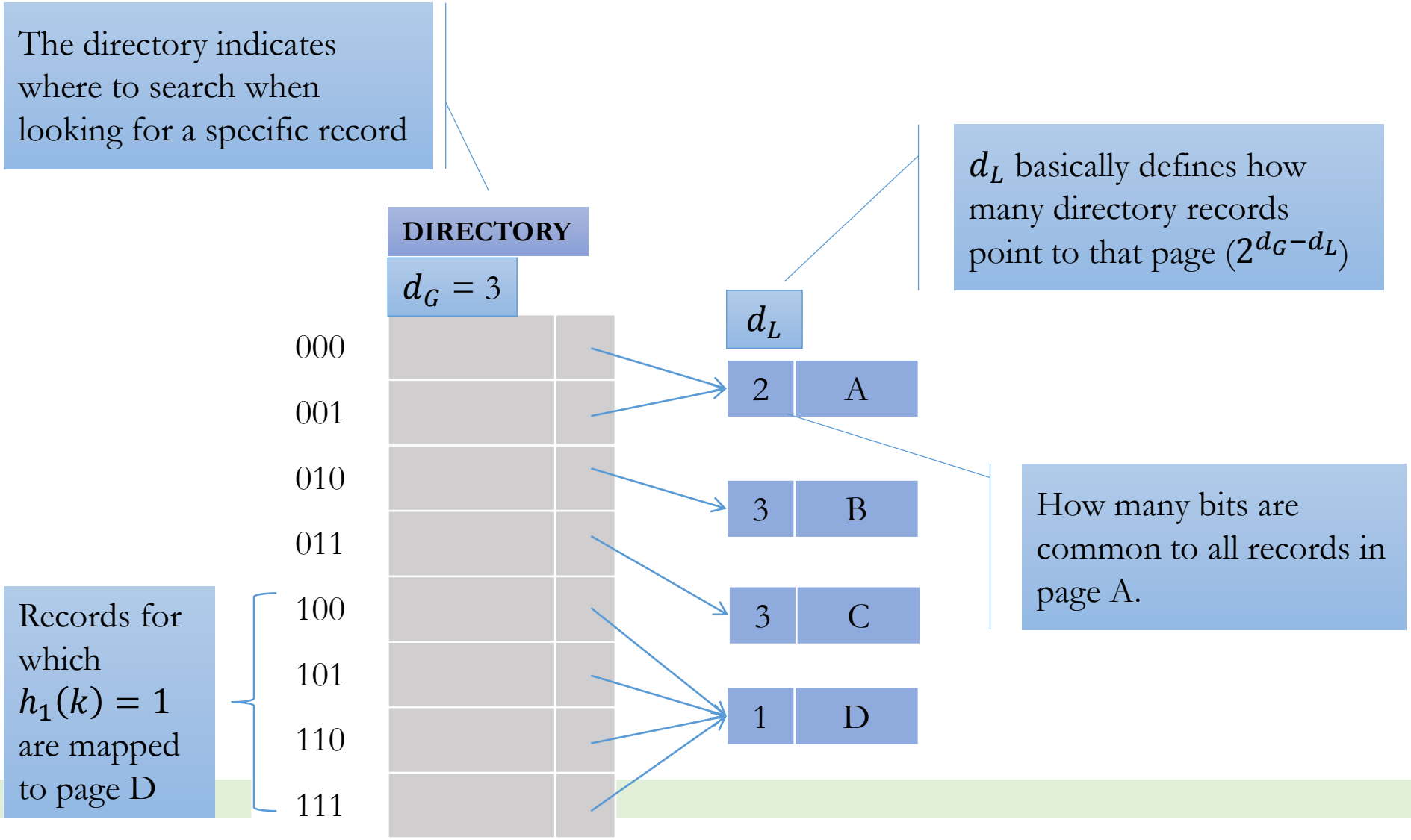
Dynamic bit-based hash function

- In order to have a dynamic hashing, a **hash function should grow/shrink its domain** according to the data
- Typically, hashing schemes use **a bit function** which generates **more bits as the file expands** and **fewer as the file contracts**
  - the hash function can be defined as a **series of functions**
    - $\boldsymbol{h_i: K \rightarrow \{0, 1, \ldots, 2^i - 1\}}$
    - $\boldsymbol{h_i(k) = h_{i-1}(k) \vee h_i(k) = h_{i-1}(k) + 2^{i-1}}$
  - let us have a function $\boldsymbol{H}$ **mapping a key** $\boldsymbol{k}$ **into a random** $\boldsymbol{m}$**-long bit pattern**
  - $h_i(k)$ then corresponds to last i bits of $H(k)$
    - $\boldsymbol{H(k) = b_m \ldots b_2 b_1 b_0 \rightarrow h_i(k) = b_{i-1} \ldots b_1 b_0}$
    - working with first i bits is equivalent – in the following algorithm we will user first few bits since the images are more compact in that way

Fagin's extendible hashing

- Fagin, 1979
- **Directory-based** scheme
  - directory contains **global depth d_G**
    - global depth expresses **maximum** number of **bits needed to tell any pair of records from different buckets apart**
  - each page contains **local depth d_L**
    - local depth expresses **the number of bits common** to all records **in a page** apart
- **Querying**
  - $h_{d_G}(H(k))$ provides **address** of the directory entry with a pointer to the bucket/page containing the record with key k
- **Overflow**
  - overflowing causes a **change in the structure of the directory** (d_G, d_L) **and the primary file**
  - does not involve static hashing overflow techniques, e. g., chaining

Fagin


The directory indicates where to search when looking for a specific record
DIRECTORY
$d_G = 3$
000
001
010
011
100
101
110
111
Records for which $h_1(k) = 1$ are mapped to page D
d_L
d_L basically defines how many directory records point to that page ($2^{d_G - d_L}$)
2
A
3
B
3
C
1
D
How many bits are common to all records in page A.

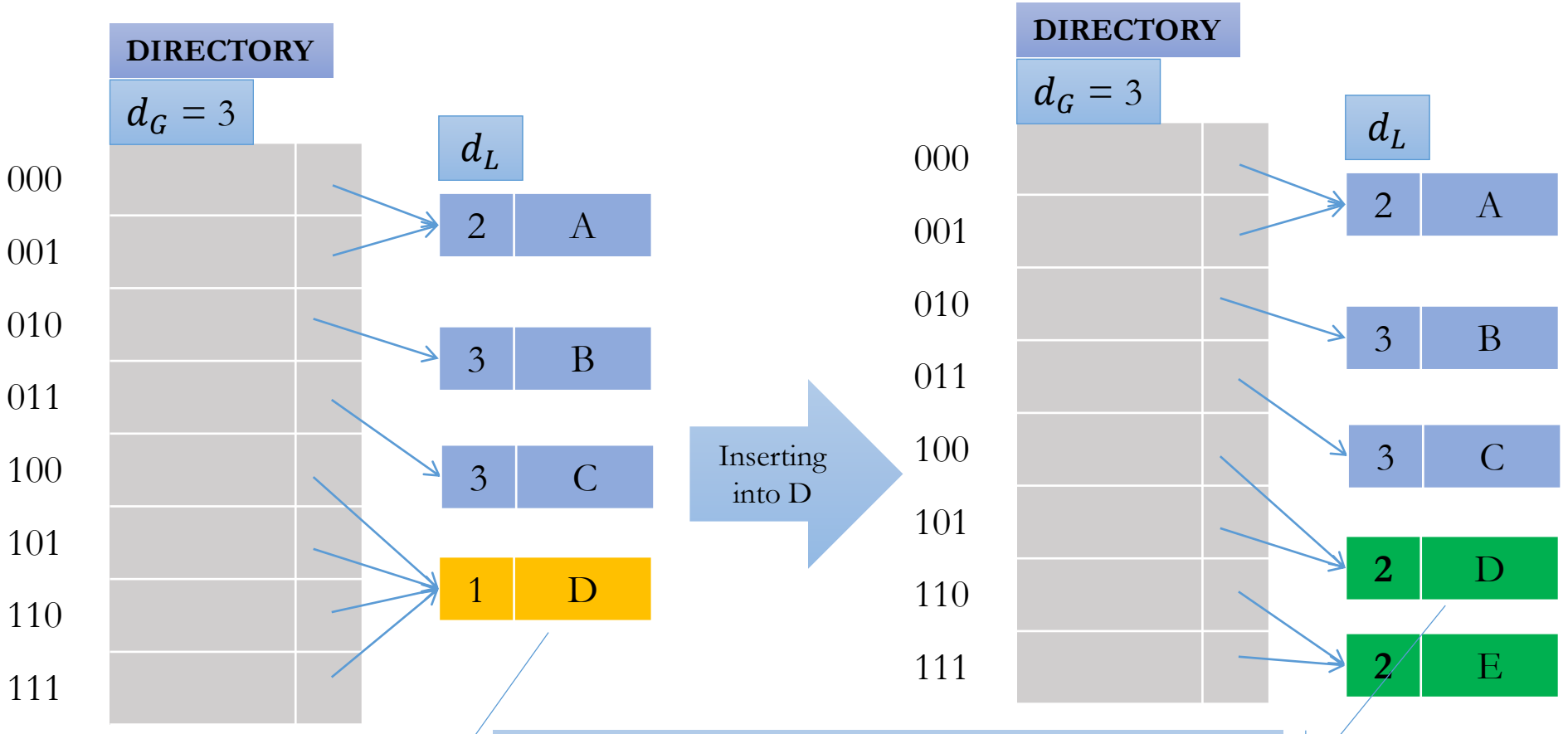
Fagin - FIND

- Finding a record with a key k

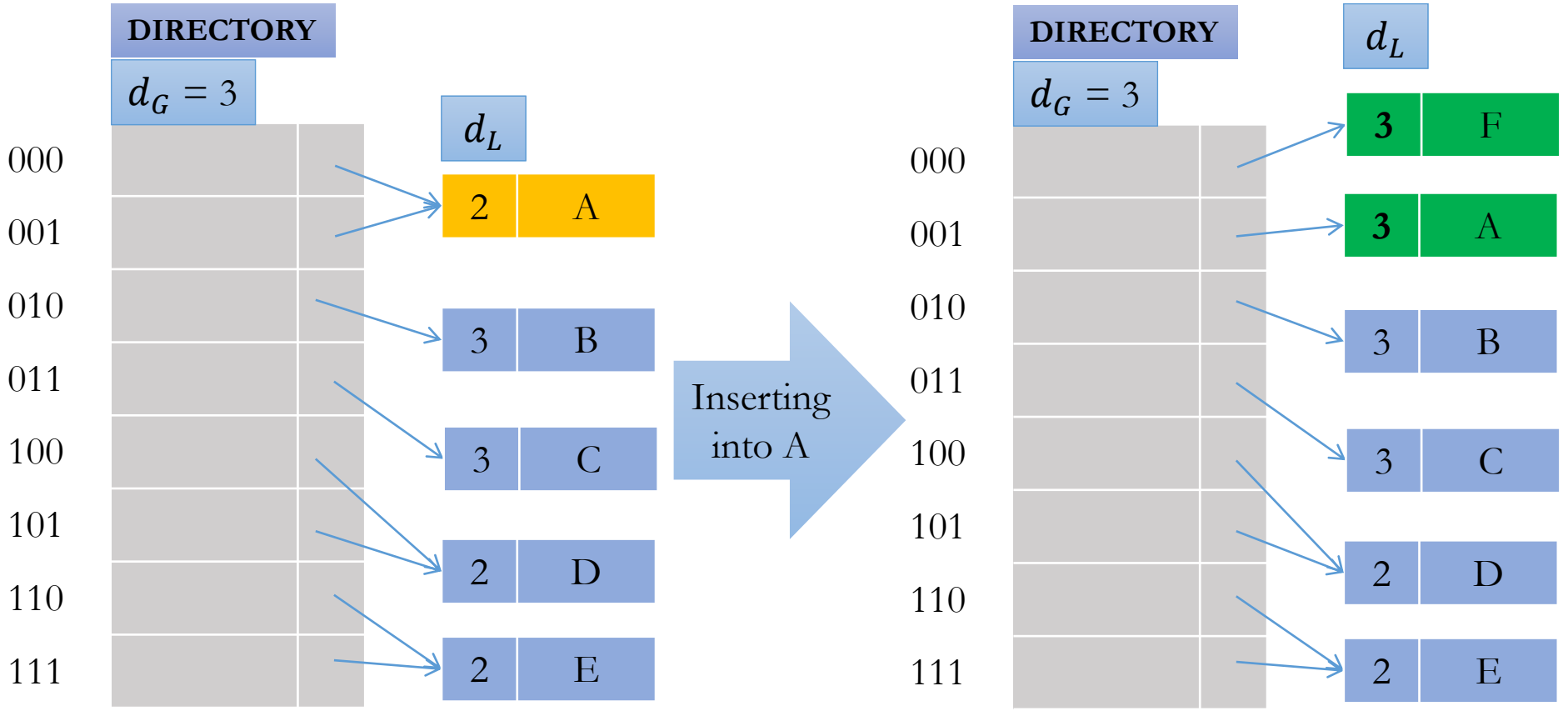
  1. **Compute $k' = H(k)$**
  2. **Compute $k'' = h_{d_G}(k')$**
  3. **Access page pointed to by the directory record with key k''**
  4. **Scan the accessed page for record with key k. If the record is not found it is not present in the file.**

- No scanning of overflow areas

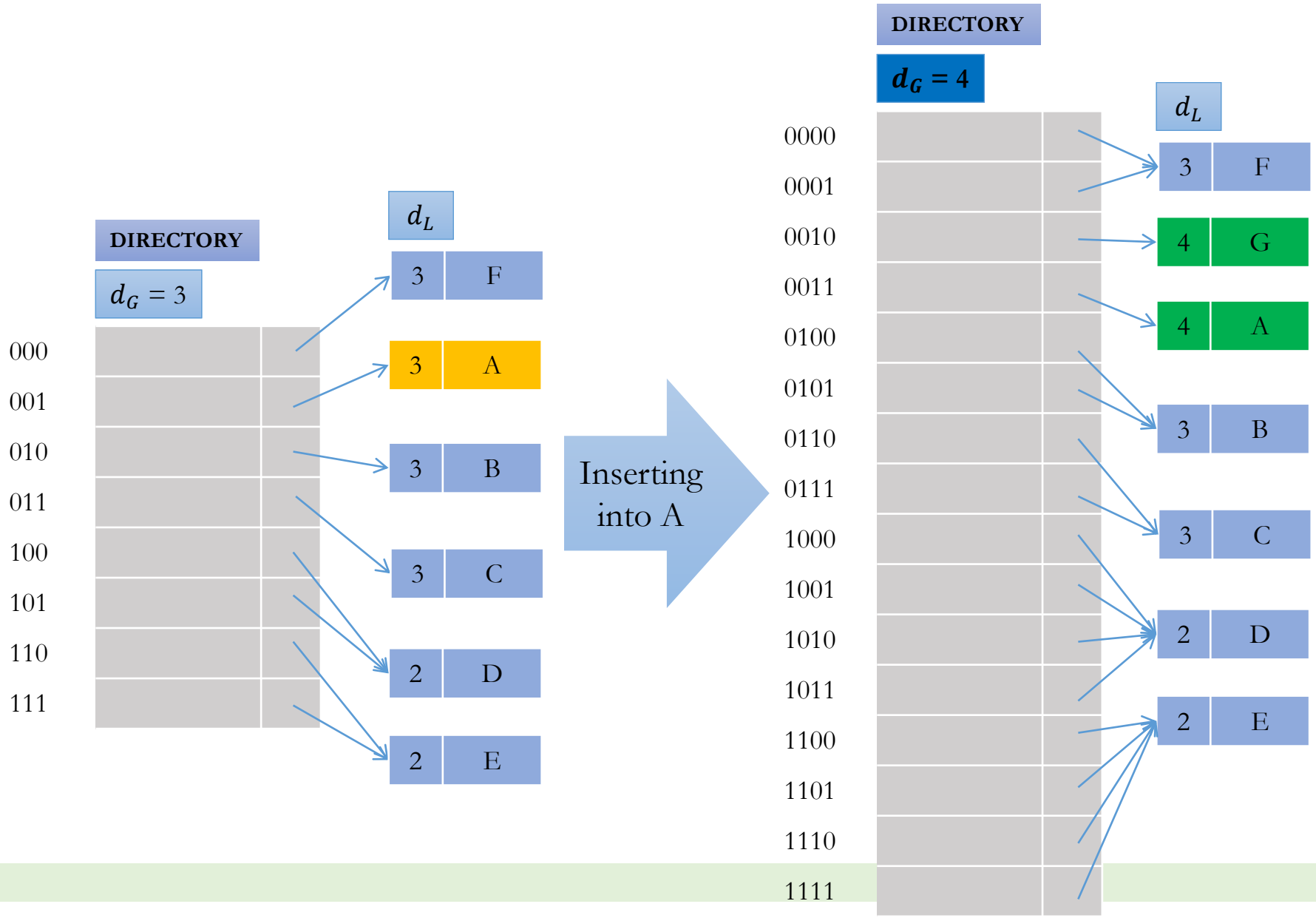
Fagin – INSERT



Fagin – INSERT (cont.)



Fagin – INSERT (cont.)



Fagin – INSERT (cont.)

- **Inserting** a record **R** with a key ***k***

1. **find a page B** where the record R should be inserted
2. if **B is not full**, **insert** R into B and **return**
3. if **B is full,** split B
4. **repeat** steps **1-2**

Fagin – Page Split (Local)

- Splitting page *(štěpení stránky)* B if $\boldsymbol{d_L(B) < d_G}$

1. allocate a new page B'
2. modify the directory pointers originally pointing to B so that, e.g., half of them having common first $d_L(B)$ bits followed by 0 point to B and rest of them point to B'
3. set local depth of both B and B' to $d_L(B) + 1$
4. reinsert all the data in B into the index based on the first $d_L(B')$ bits
5. if all records get into the same page, repeat the process

Fagin – Page Split (Global)

- Splitting page B if $\boldsymbol{d_L(B) = d_G}$

1. double the directory size
2. set $d_G = d_G + 1$
3. for each page P, set the pointers so that if P was pointed to by an entry with a bit key $\boldsymbol{x}$ ($\boldsymbol{|x| = d_G - 1}$) now it is pointed to by entries with keys starting with $\boldsymbol{x0}$ and $\boldsymbol{x1}$
4. $d_L(B) < d_G \rightarrow$ continue with local page split

Fagin – Pros & Cons

Pros

- The performance stays more or less constant with increasing number of stored records

Cons

- Another level of indirection
- The directory might not fit in the main memory
- If the block factor is low, many splits can occur leaving many pages empty

Linear hashing (1)

- Litwin, 1980, Enbody & Du, 1988
- **Directory-less scheme**
- Principal idea
  - in Fagin, doubling the directory does not lead to doubling the number of data pages but that would be the case in the basic directory-less scheme
  - let us add **one page after a pre-specified condition** (such as overflow or given number of inserts)
  - if we find ourselves in i-th step/iteration then after 2^i insertions we get into $i+1$-st iteration

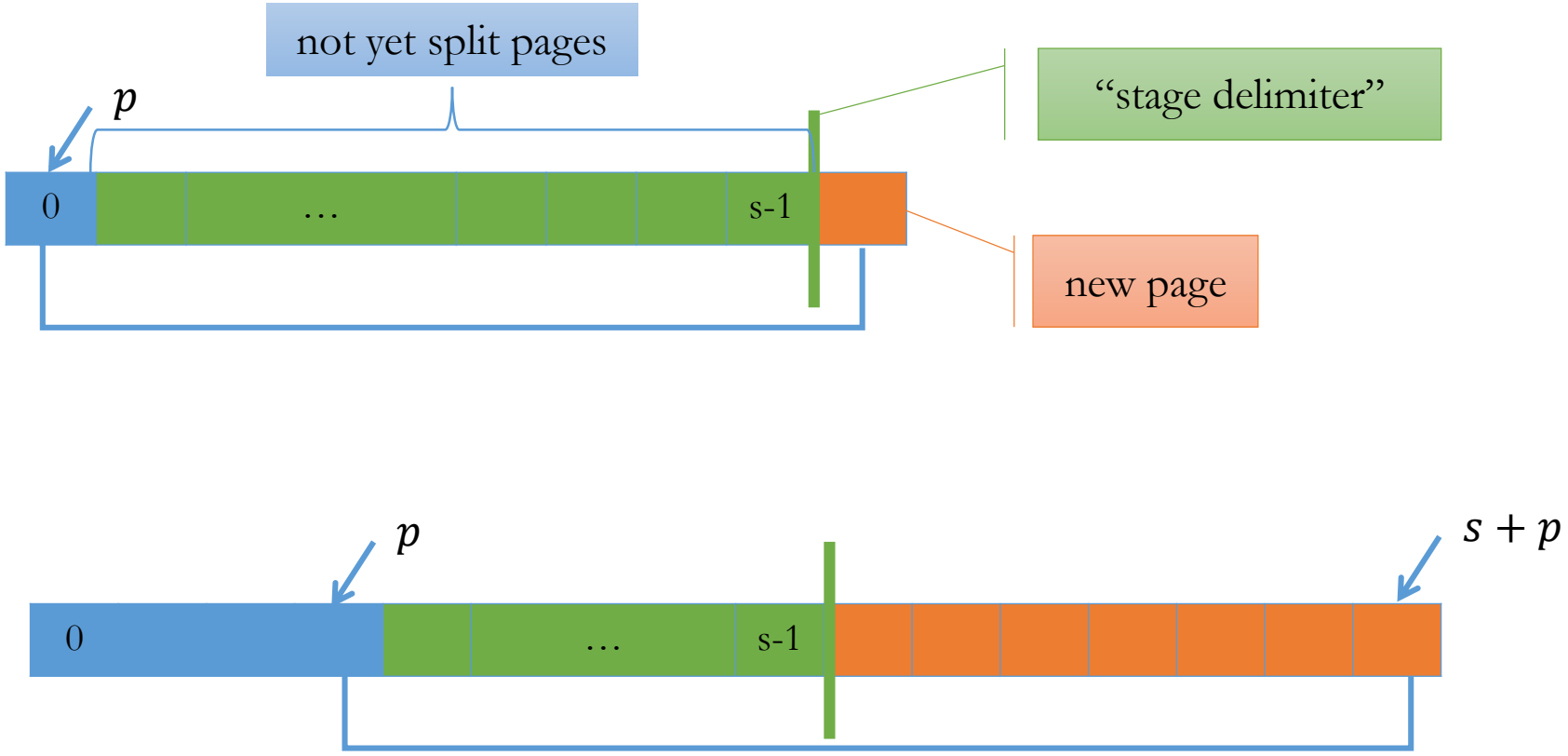
Linear hashing (2)

- **Expansion** process divided into **stages** (*fázi*)
- **Stage d starts** when the **number of pages** is $s = 2^d$ and **ends** when the **number of pages** reaches 2^{d+1}
- A (split) **pointer p** is used to point to the pages $0 \ldots 2^d$
- The **purpose of p** is to identify the **next page to be split**

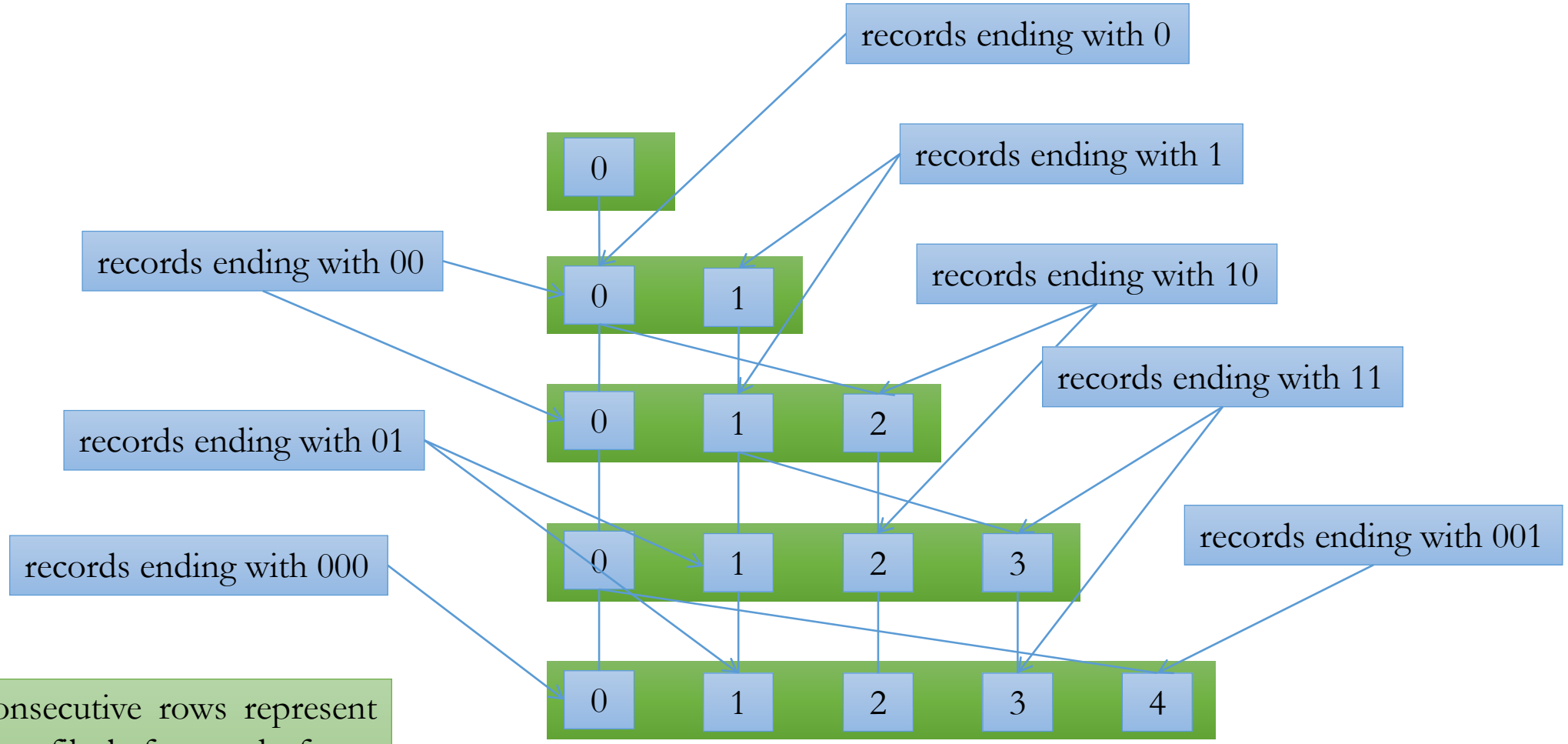
Linear hashing (3)

- At the **beginning of stage d, p points to page 0** and after every **split** operation it is **increased by 1** (moves to the next primary page)
  - if a page overflows before it is its time to split, overflow pages need to be utilized
- The **growth** of the primary file is **linear → linear hashing**
- When **splitting** the **new primary page** will be $p + s$
- **Records from page p** (and possible overflow pages) will be distributed **between pages p and $s + p$** using hash function $h_{d+1}(k)$
- At **each stage two types of hash functions**
  - for pages already split
  - for pages not yet split

Linear hashing (4)

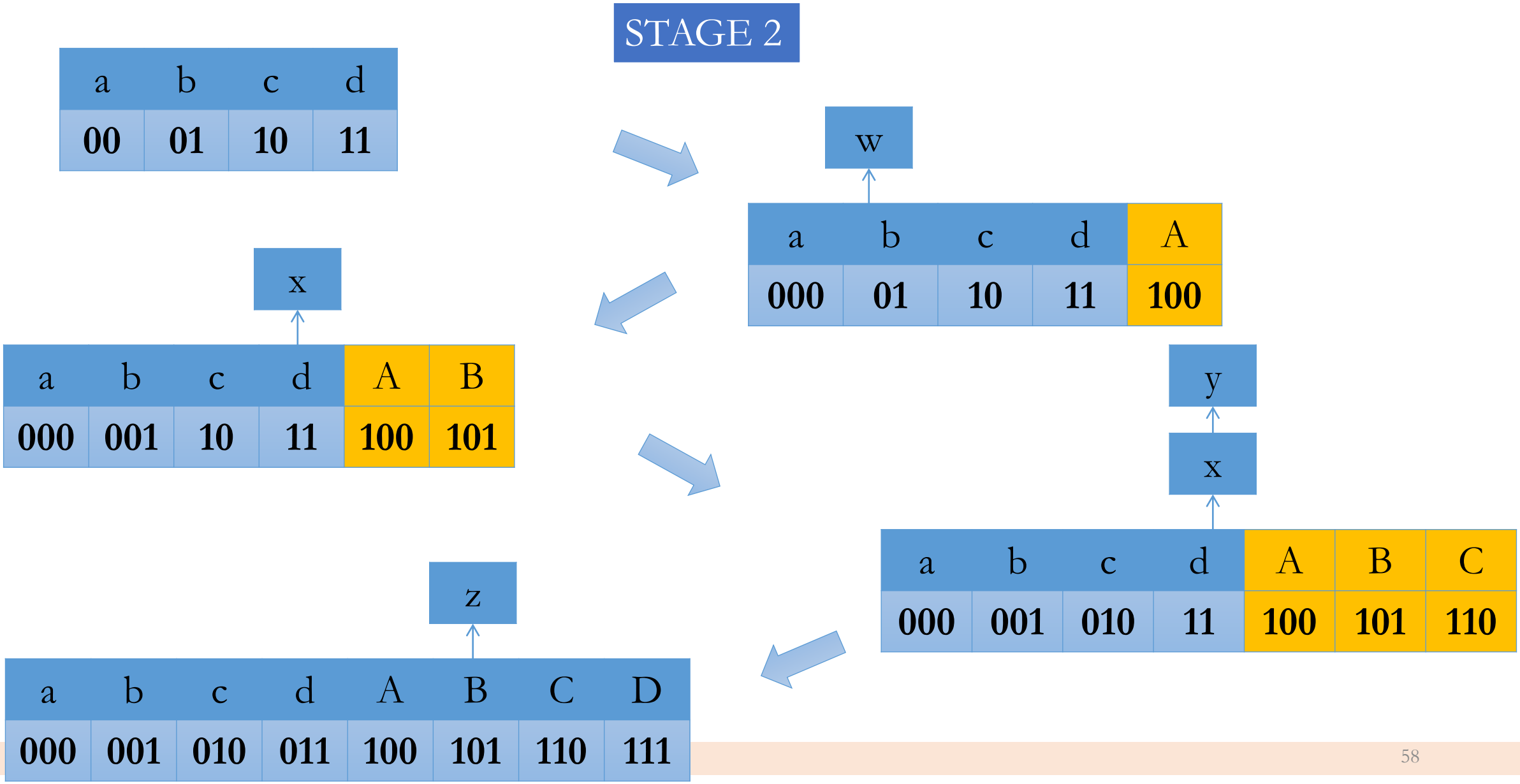


Linear hashing (5)



The consecutive rows represent the same file before and after a splitting opreations.

Linear hashing (6)



Linear hashing - addressing

- Unlike directory-based hashing, address of a record has to be computed

- Pages **left of p are already split** and therefore need **one more bit** for addressing than pages right of p

```
ADDR GetAddres(KEY k, int cnt_pages){
   d = floor(log(cnt_pages, 2));
   s = exp(2, d);
   p = cnt_pages % s;

   addr = h(k) % s;
   if (addr < p) addr = h(k) % exp(2, d + 1);

   return addr;
}
```

Splitting in linear hashing

- **Uncontrolled splitting**
  1. page pointed to by p is split after L insertions (parameter of the method)
  2. page pointed to by p is split when any page overflows
     - utilization around 60%

- **Controlled splitting**
  - splitting occurs when the utilization of page pointed to by the split pointer reaches a threshold, e.g., 80%

Overflow handling policy

- **Splitting control** has a direct effect on **how much overflow will be tolerated**

- Delayed splitting improves space utilization
  - instead of splitting a page as soon as it overflows, an **overflow page is utilized**
  - **the size of an overflow page can be different** from the size of the primary file page

- Deferring overflow **can also be applied to directory schemes**
  - especially helpful when the overflow causes directory doubling

Deferred splitting

- **Sharing overflow pages**
  - space utilization can be increased by sharing overflow pages
    - more pages share one overflow page
    - similar to having smaller overflow pages

- **Buddy pages**
  - logical pairing of pages
  - if a page overflows, the overflown records are inserted into the buddy page
  - if the buddy page needs its space or too many overflows occur the original page overflows

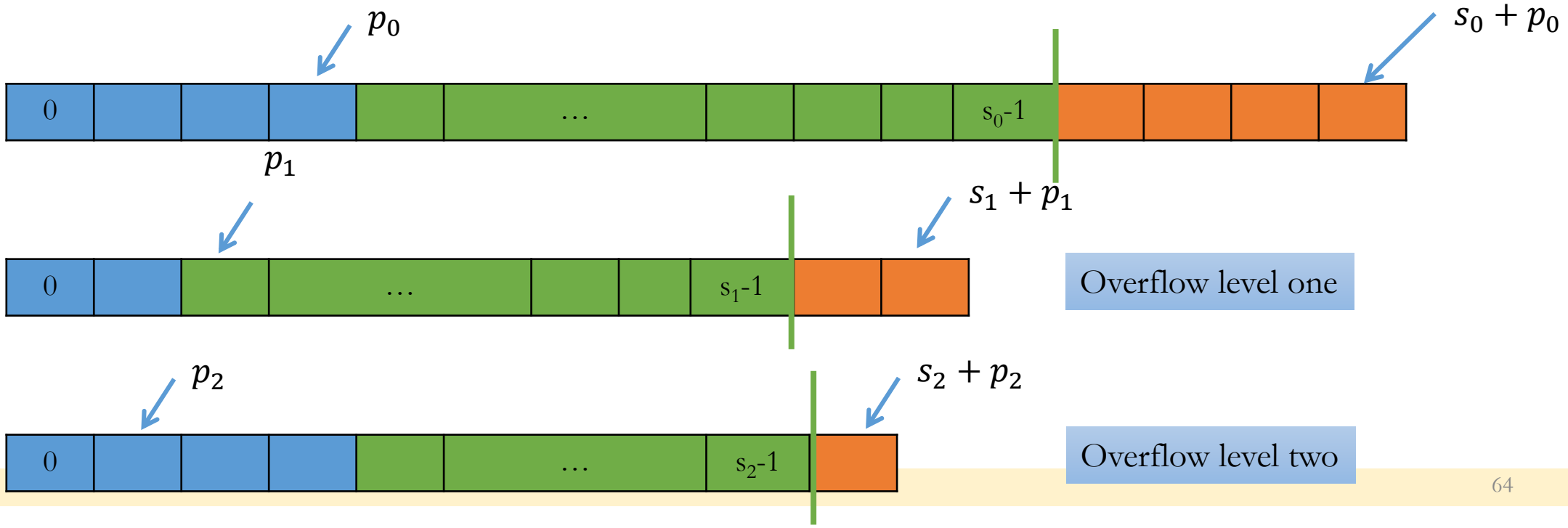
Recursive linear hashing (1)

- Ramamohanarao & Sacks-Davis, 1984
- Employs **recursive overflow handling**
  - the **overflow space** is managed as a **dynamically hashed file**
  - pages in overflow areas may themselves be overflown → **multiple levels** of dynamic files
- The **overflows are not explicitly linked** from the primary page

Recursive linear hashing (2)

- When a record overflows, no overflow page is created but the record is inserted into the 2-nd level (and possibly recursively into 3-rd and so on)

- A file at every level as a standard linear hashing file



Recursive linear hashing - splitting

- **Splitting** of a page includes **collecting** of all the relevant records from the **next level**
  - when a page at level i is split, overflown records not only from level $i + 1$ are collected but also from all the following levels $i + 2$, …
  - if the primary page still overflows, the **overflown records are put back** into the first (and possible following) level
  - decision whether to split can be controlled by the same splitting policy as in the standard linear hashing
- It has been shown that usually 3 levels are sufficient

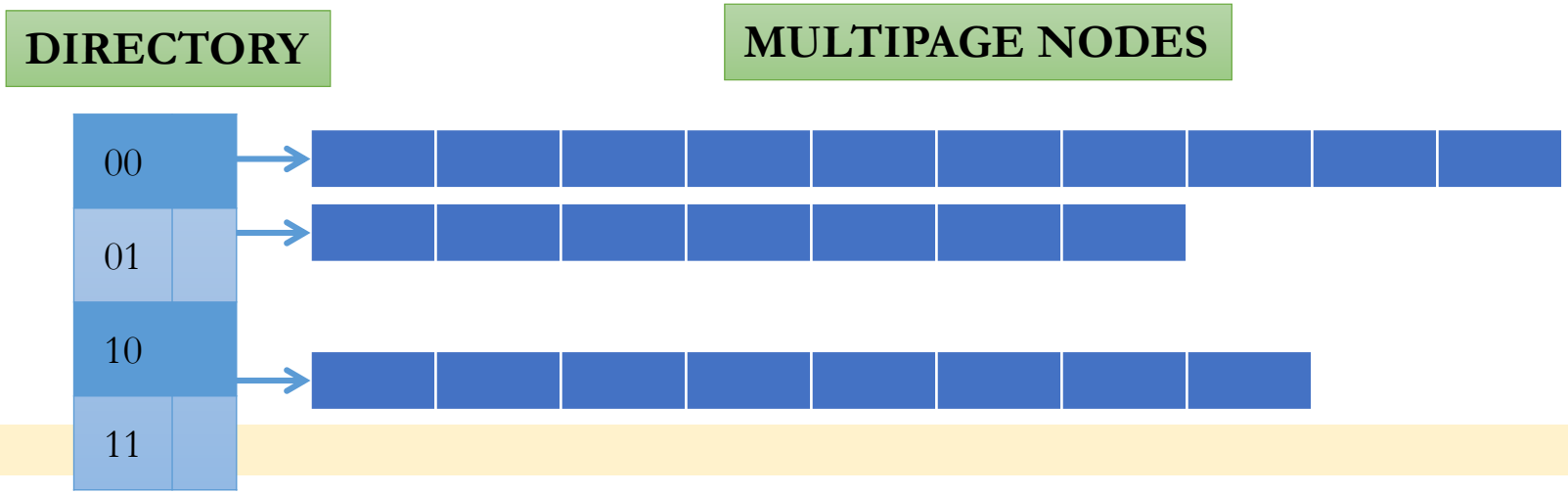
Recursive linear hashing - addressing

- Similar to linear hashing, but every level has different split pointer and number of pages
- Function ***search(p,l,k)*** searches page ***p*** in level ***l*** for record with a key ***k***

```
ADDR GetAddres(KEY k, int *cnt_pages, int cnt_levels){
   bool found = false;
   for (int level = 0; level < cnt_levels; level++) {
     d = floor(log(cnt_pages[level], 2));
     s = exp(2, d);
     p = cnt_pages % s;
     addr = h(k) % s;
     if (addr < p) addr = h(k) % exp(2, d + 1);
     if (search(addr, level, k)){
       found = true; break;
     }
   }
   if (found) return addr; else return NULL;
}
```

Multipage nodes (1)

- Lomet, 1983
- Dealing with the **possible growth of the directory** in directory schemes
- **Fixed upper limit** placed on the size of the directory
- When the limit is reached, the **nodes expand (not the directory)** forming a **multipage node**
- Access to the record needs one access to the directory and searching the multipage node



Multipage nodes (2)

- Management of multipage nodes
  - multipage nodes are stored **next to each other** → can be managed using standard file organization techniques
  - **sorted sequential files**
    - records stored in the same order they were inserted into the multipage node
  - **dynamically hashed file**
    - number of pages can be kept in the directory → the multipage node can be managed as a dynamically hashed file

Expansion techniques (1)

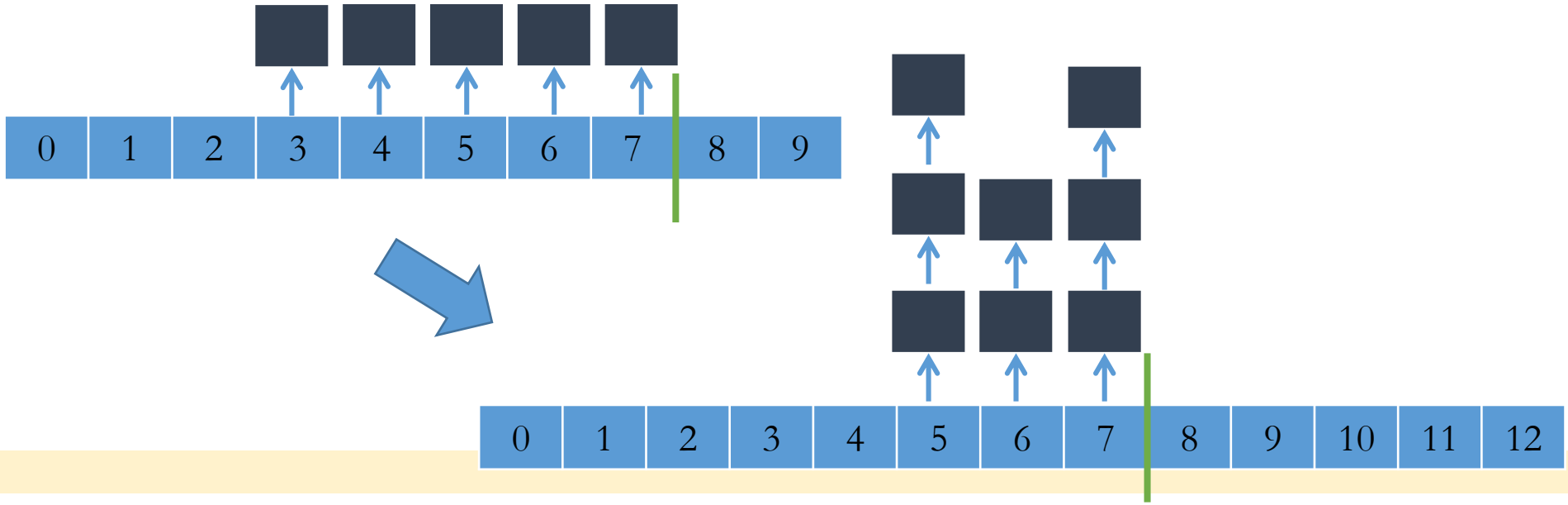
- Dynamic hashing schemes have **oscillatory performance**
  - having uniform hash function causes **all the pages to be filled more or less at the same time**
  - during **short period** many of the pages **overflow and split**
    - **utilization** goes to **N%** and then the **next moment** drops to about **0.5N%**
    - during the **splitting period** the **cost of insertion** is considerable **higher**
    - if **overflow management** techniques are employed, when the **utilization approaches 100%** the **cost of insertions and fetches increases**

Expansion techniques (2)

- Techniques **to smooth the expansion** were developed
  - **uniform distribution**
    - **linear hashing with partial expansion**
  - **non-uniform distribution**
    - **spiral storage**

Linear hashing with partial expansion (LHPE)

- Larson, 1980
- Overflow chains close to the right end of the unsplit region get too long at the end of the expansion stage
  - recently split pages are underutilized
  - pages near the right end are overutilized



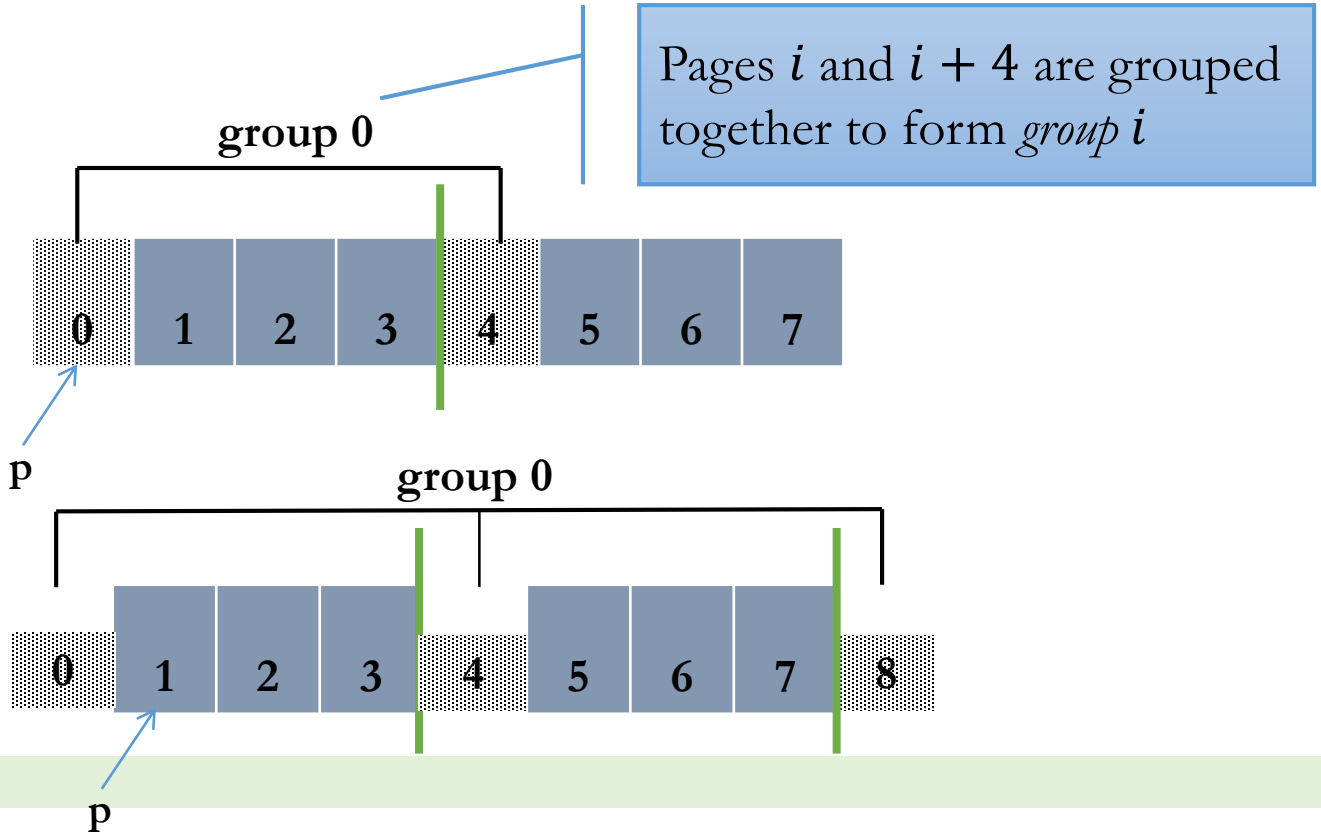
LHPE – principal idea

- Linear hashing
  - splitting after L insertions
  - s pages
- In linear hashing page $s-1$ splits after sL insertions
- In LHPE we distinguish between partial and full expansion
- During **one full expansion** the file expands in **more partial stages** and in **each partial stage all the pages are split**
  - If the number of the **partial expansion stages** is **2,** the pages $\boldsymbol{s-1}$ **splits after** $\boldsymbol{sL/2}$ **insertions** → shorter overflow chains

LHPE - expansion (1)

- $d = 3, s = 2^d = 8, g = 2\ (pages\ in\ group)$

First partial expansion



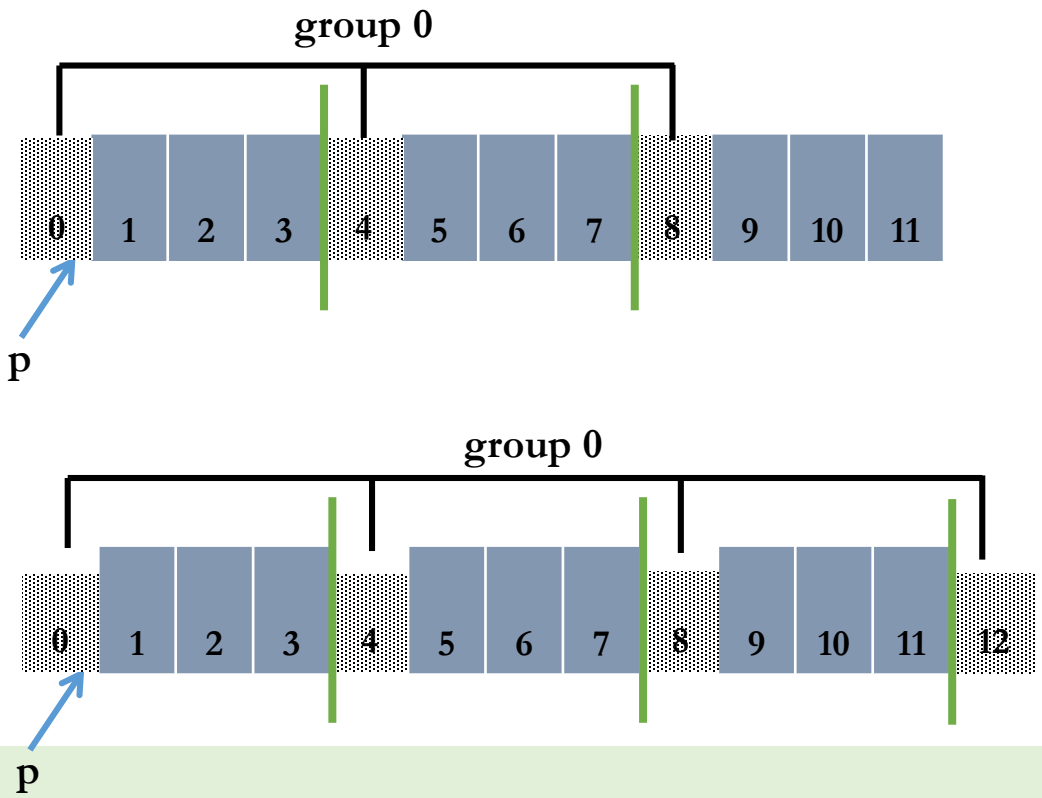
Splitting a page

1. New page is added
   - page 8
2. Records from the pages in given group are spread across that group and the new page
   - pages 0, 4 and new page 8

- If b is the blocking factor and the pages are full then then utilization after the split operation is $2/3b$

LHPE - expansion (2)

Second partial expansion



Situation after first partial expansion

- we have visited each page in the original file (p passed through all the groups)
- p returns to the page/group 0
- group 0 consists of pages 0,4,8
- we are halfway to the full doubling but we have already visited all the pages

Second partial expansion

- next page added will be 12
- If b is the blocking factor and the pages were full than utilization after the split operation is about $3/4b$
- after this partial expansion, there will be 16 pages, the file will be doubled and one full expansion is over
- size of each group will shrink to 2 again

LHPE - addressing

- To **address a page** we identify the respective **group** and compute the **offset within that group**
- When we are in d-th full expansion, the gap between pages of the same group is 2^{d-1}

$$\boldsymbol{addr(k, d, n, p) = group + 2^{d-1} * offset}$$

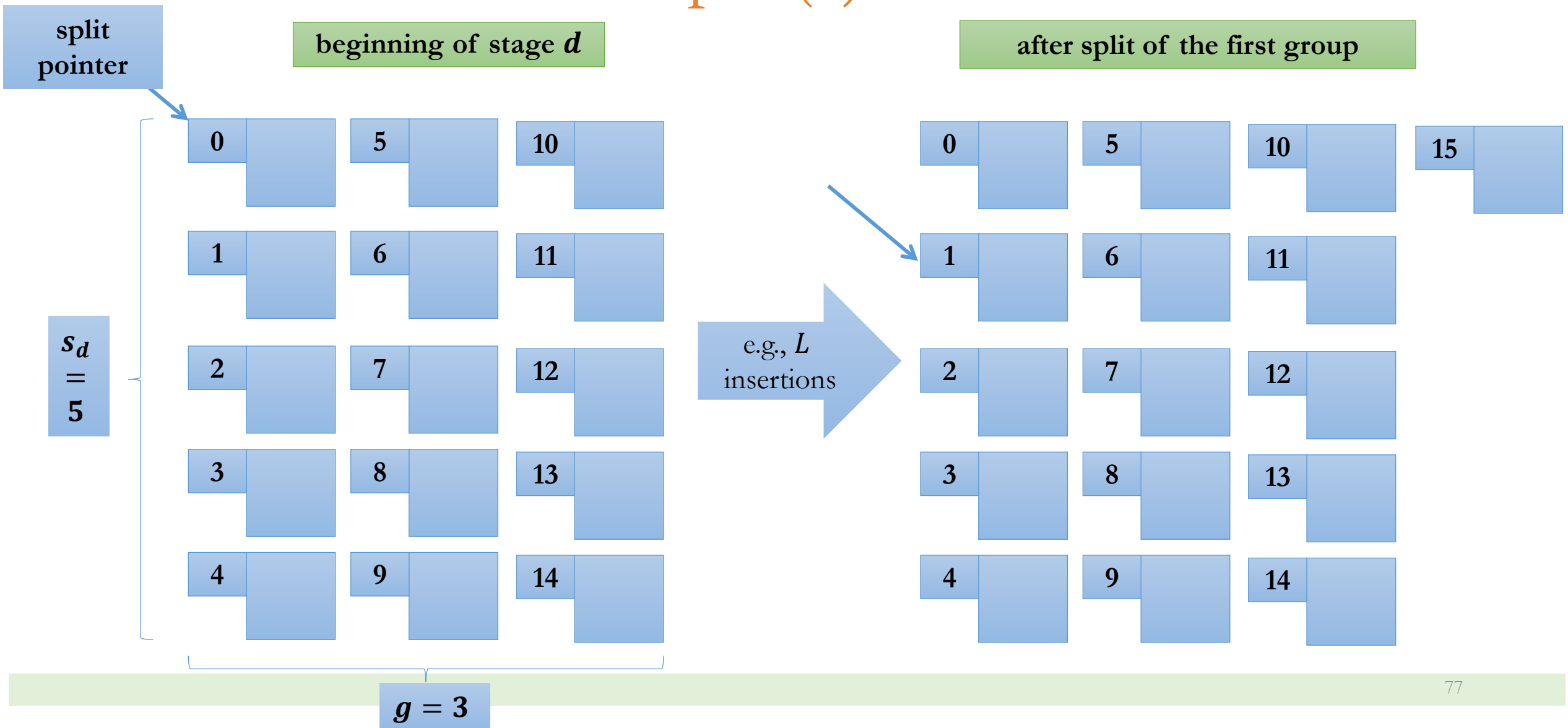
- k … key, d … full expansion, n … partial expansion, p … split pointer
- group determined as in linear hashing
- offset determined by another hash function mapping into the size of the group
- Simplified algorithm:

```
ADDR GetAddres(KEY k, int d, int n, int p){
  if (h_d(k)>=p) return h_d(k) + 2^(d-1) * i_n(k);
  else return h_(d+1)(k) + 2^(d-1) * i_(n-1)(k);
}
```

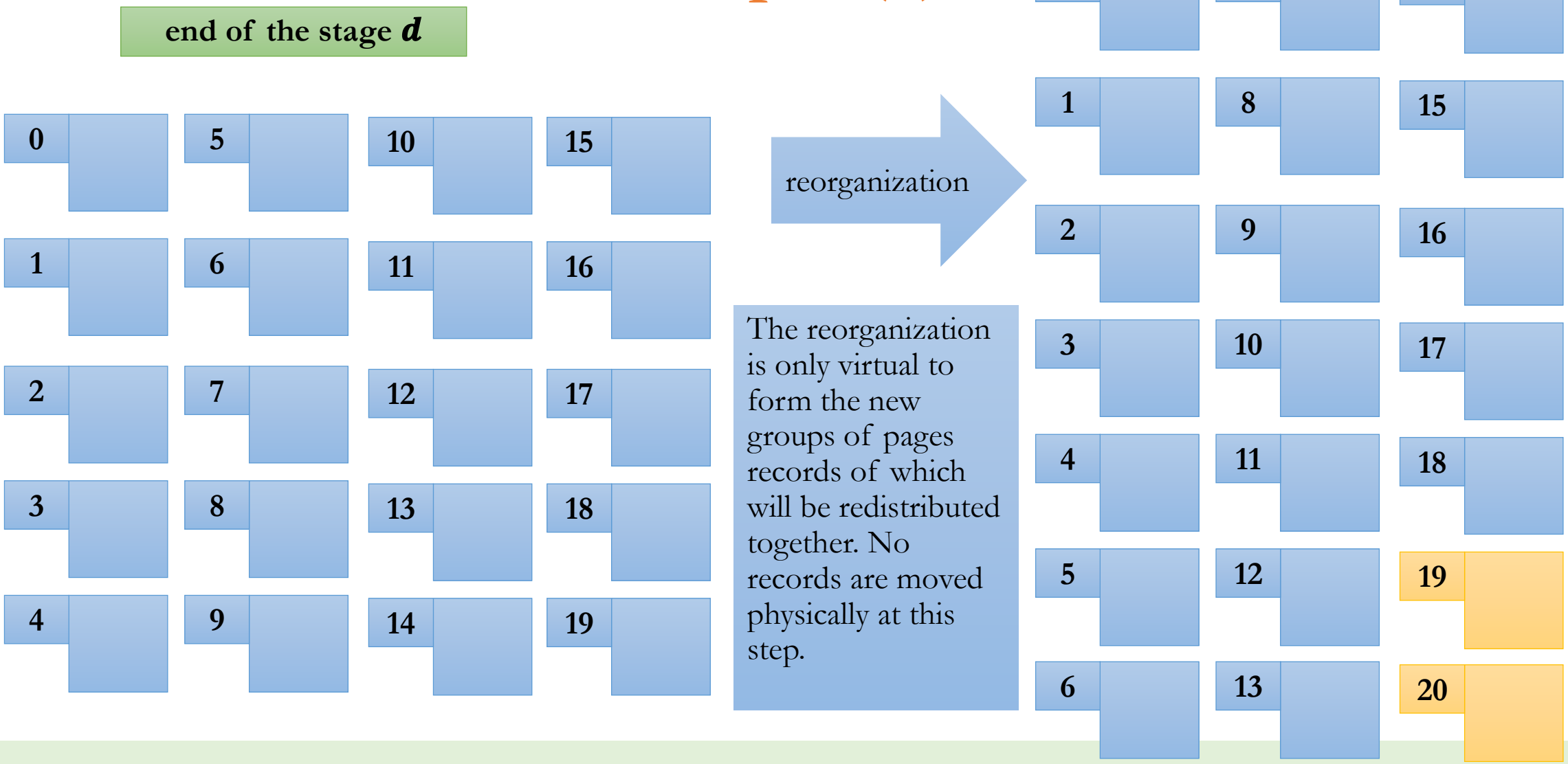
LHPE-RL

- Ramamohanarao & Lloyd, 1982
  - the proposed scheme does not have a name, let us call it **LHPE-RL**
  - *(skupinové štěpení stránek)*
- **Simplified** version of **LHPE**
  - **partial expansion in LHPE** corresponds to **full expansion in LHPE-RL**
  - pages of the primary file (having $\boldsymbol{p_d}$ **pages**) at **stage** $\boldsymbol{d}$ are **grouped into** $\boldsymbol{s_d = p_d/g}$ **groups** (**each** having $\boldsymbol{g}$ **pages**)
  - when a predefined condition is met (e.g., after L insertions), a **new page** is inserted at the end of the primary file and records in **pages in the group pointed to by the split pointer are redistributed** between those pages and the new page (being the new member of the group)
  - when the **last group is redistributed**, the file is (virtually) **reorganized** (stage $\boldsymbol{d+1}$) so that all the pages are again sorted into $\boldsymbol{s_{d+1} = p_{d+1}/g}$ pages ($\boldsymbol{p_{d+1} = \lceil s_d * (g+1)/g \rceil * g}$)

LHPE – RL – example (1)



LHPE – RL – example (2)



LHPE-RL – addressing (1)

- The file undergoes **series of redistributions and splits** and so do the records in the pages

- The method assumes **one initial hashing function h_0** and series of **independent hashing functions $h_i: K \rightarrow \{0 \ldots g\}$** being used when splitting to **identify the offset** of records **in each group**

- To **identify a position** of a record in the file the **sequence of splits and redistributions has to be "replayed"**

LHPE-RL – addressing (2)

1. At **stage 1**, a record with **key k** is inserted into a page determined by **initial hashing function $h_0(k)$**

2. When the **split pointer** gets to the **group where k resides**, the records are redistributed using hash function $h_1(k)$ (mapping to the space $< 0; g - 1 >$ and thus the **record moves into page $p_1 = h_0(k)$ % $s_1 + h_1(k) * s_1$**

   - $h_0(k)$ % s_1
     - id of the group
   - $h_1(k)$
     - offset within the group
   - $h_1(k) * s_1$
     - pages of one group are s_1 pages apart → $h_1(k) * s_1$ is the offset in the linear address space

LHPE-RL – addressing (3)

3. After the **redistribution** (stage = 2), page $\boldsymbol{p_1}$ **gets into** group $\boldsymbol{g_2 = p_1 \% s_2}$

4. When the **split pointer reaches** $\boldsymbol{g_2}$, $\boldsymbol{h_2}$ **is used** to get the new addresses for records in pages in g_2 (and therefore also page p_1 where the record with key k resides) $\rightarrow \boldsymbol{p_2 = g_2 + h_2(k) * s_2}$

5. The process **iterates until** the last stage $\boldsymbol{d_L}$ is reached

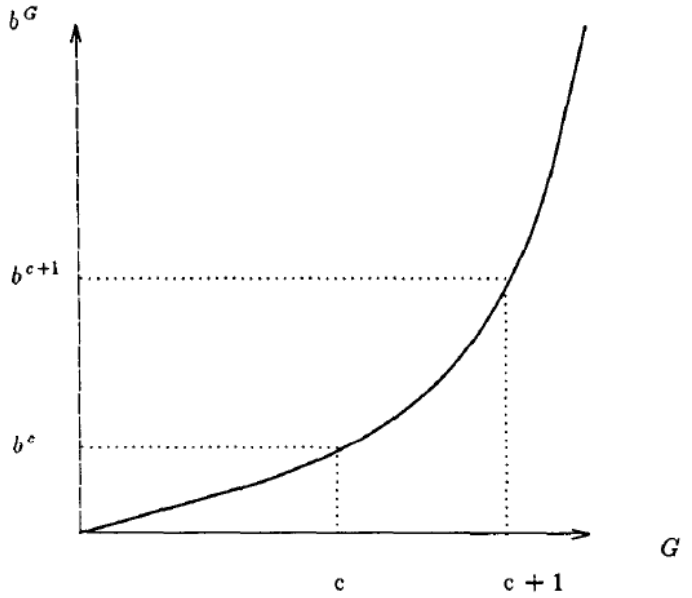
6. If $\boldsymbol{g_L}$ is **greater or equal than the split pointer position** the desired page is $\boldsymbol{p_{L-1}}$, **otherwise** we need moreover to compute $\boldsymbol{p_L}$ using h_L
   - this step corresponds to the situation in linear hashing when we differentiate between page addresses before and after the split pointer

Spiral storage

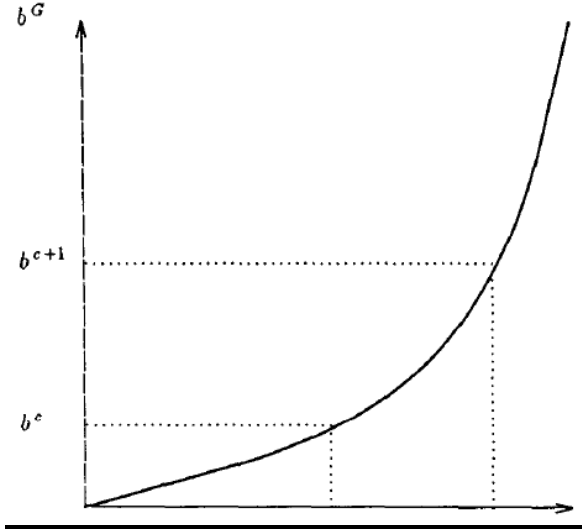
- Martin, 1979; Mullin, 1985
- Directory-less scheme
- In linear hashing, **pages on the left end** of the primary file tend to be **underutilized** whereas **pages on the right** end are **overutilized**
- If the keys had **exponential distribution**, pages on the left end would get more records and thus **balance the inequality** of the pages caused by different time of splitting

Spiral storage – idea (1)

- Spiral storage **takes keys which have originally uniform distribution** and gives them **an exponential distribution** and then assigns logical addresses
- Unlike most of the schemes, spiral storage does not expand by adding pages on the right side of the primary file
- **During expansion, pages are both deleted and added**
  - growth secured **by adding more pages than deleting**
  - records from the deleted pages are spread across the new pages



Spiral storage – idea (2)



- Keys are uniformly distributed between c and c+1 using

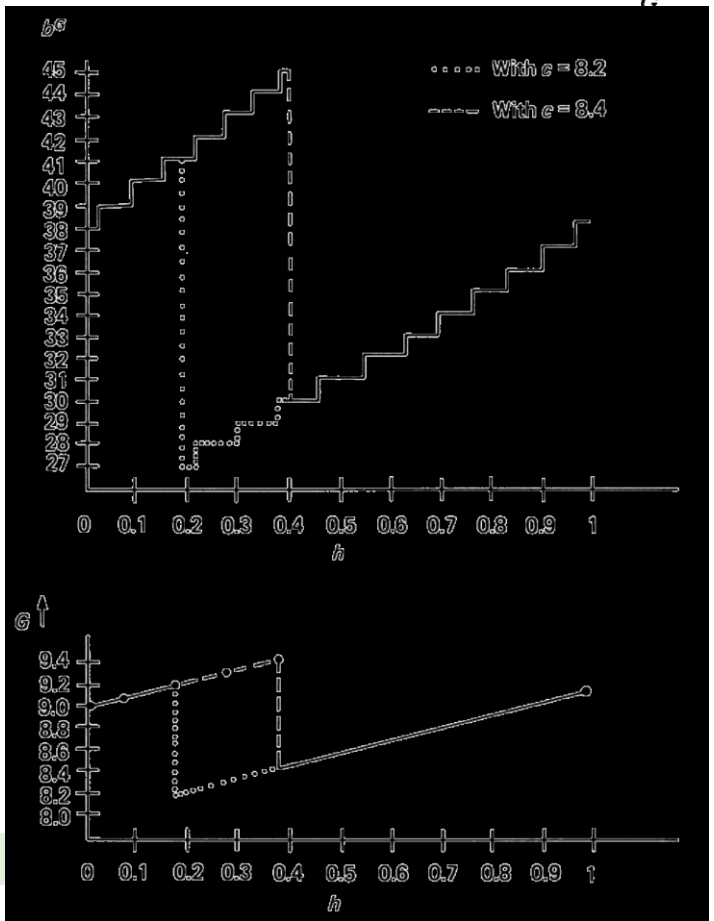
$$G(hash(k)) = \lceil c - hash(k) \rceil + hash(k),$$
$$G(hash(k)) < 0; 1 > \rightarrow < c; c+1 >, \qquad 0 \leq hash(k) \leq 1$$

- Logical address is then $\lfloor b^{G(hash(k))} \rfloor$

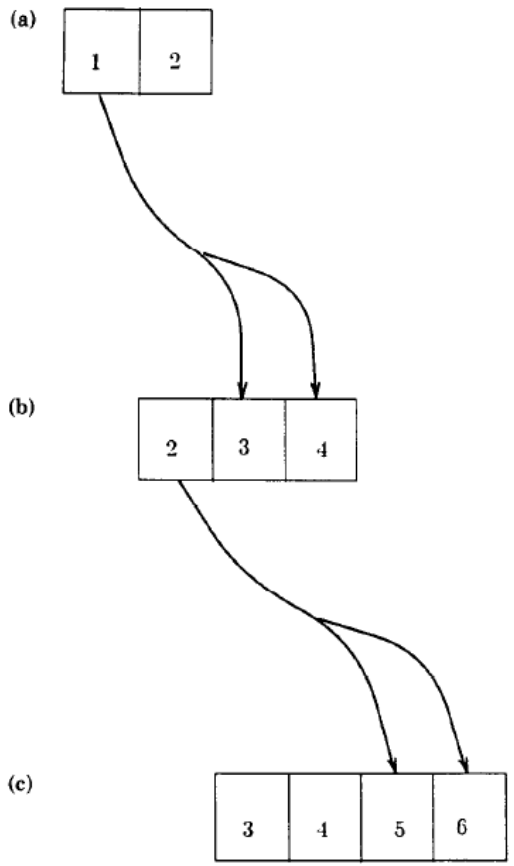
- When $< c; c+1 >$ moves to the right, size of $< b^c; b^{c+1} >$ increases and vice versa

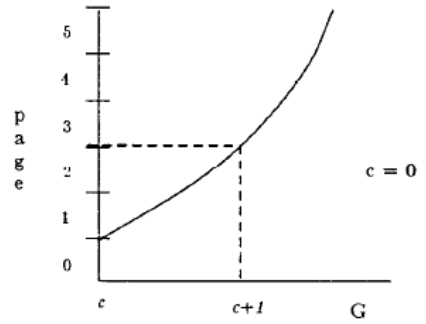
- During expansion, new c is picked in such a way to eliminate the current first page

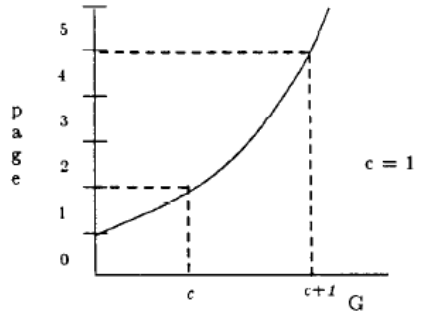
$$new\_c = \log_b(first + 1)$$

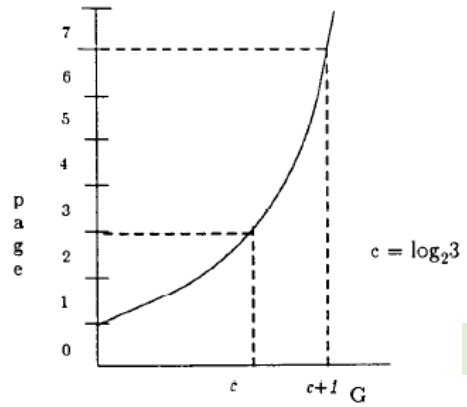


Spiral storage - algorithm









$$new\_c = \log_2(1+1) = 1$$

Page 1 deleted and pages 3 and 4 added since interval

$< 1; 2 >$ maps to $< 2^1; 2^2 >$.

$$new\_c = \log_2(2+1)$$

Page 2 deleted and pages 5 and 6 added since interval

$< \log_2 3 ; \log_2 3 + 1 >$ maps to $< 2^{\log_2 3}; \lfloor 2^{\log_2 3 + 1} \rfloor > = < 3; 6 >$.

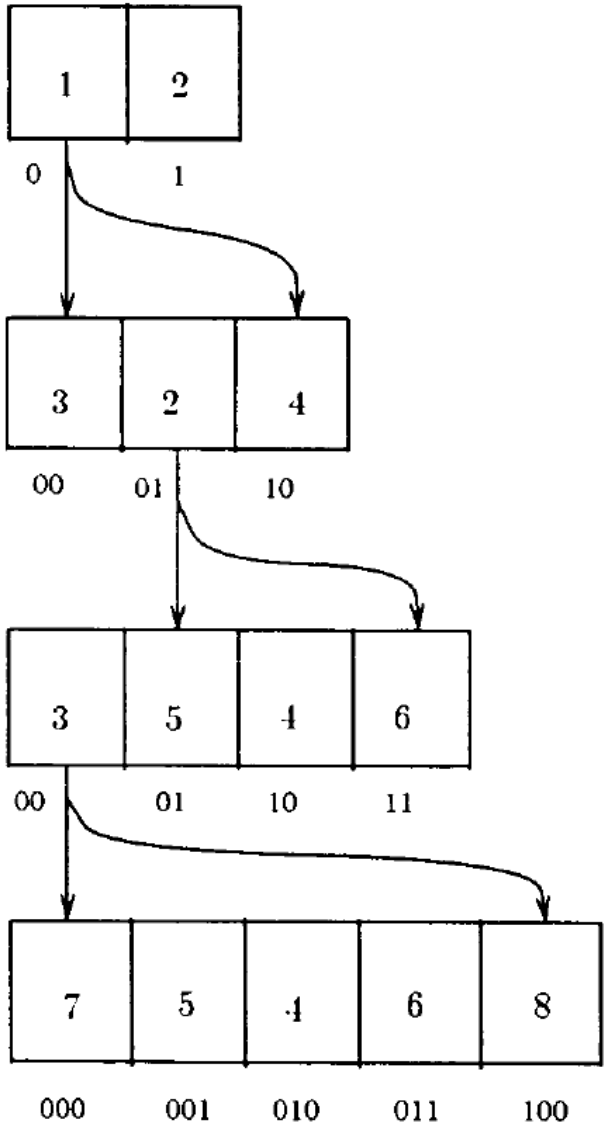
Spiral storage – mapping (1)

- The file should not drift through the address space → need for **mapping from the logical address space to the actual address space** → reusing of pages

- To translate a logical page into the actual physical page one has to **trace back the sequence of logical address modifications to the same physical page**

- Main idea
  - **first page in the list of added pages is always reused while the rest of them are the new ones**
  - if $b = 2$ then every odd page is reused

Spiral storage – mapping (2)



For b = 2

```
int actual_page(int logical_page)
{
  high = floor((1+logical_page)/2);
  low = floor(logical_page/2);
  if (low < high)
    return actual_page(low);
  else
    return logical_page - low;
}
```

Even pages at any moment are not results of a reuse. When removing a page, the page is reused/relabeled.

Membership testers (MT)

- Answers membership queries for a set: **is a query key contained in a set?**
  - weaker than preceding structures
- Suitability
  - **checking only for existence and not the content itself**
  - each (external memory) bucket can have a MT associated
- MT should be small with respect to the entire set size
- first motivation came from spell checking – testing whether a word is correct
- MT can return **false positives**
  - there should not be many of them
  - **no false negatives!**
    - if the MT returns false, we do not have to verify that claim

Exact membership testers

- Let the set come from a finite universe U containing u elements
- The tester has to represent 2^u subsets $\rightarrow$ at least $u = \log_2 2^u$ bits needed for encoding
- If we restrict ourselves to a fixed subset of n elements we need to represent only $log_2\binom{u}{n}$ bits and $log\binom{u}{n} \doteq n \log u, n \ll u$
  - several membership testers of almost that size have been proposed, e.g., in Carter et al. (1978), Brodnik and Munro (1999)

Approximate membership testers (1)

- Originally proposed by Bloom, 1970 – **Bloom filter**

- **Bit string** of length $\boldsymbol{b}$
- $\boldsymbol{k}$ **hash functions** $h_i : U \rightarrow \{1, \ldots, b\}$

- $\boldsymbol{x \in X}$: $\boldsymbol{set\ the\ bits\ \ corresponding\ to\ \ h_1(x), h_2(x), \ldots, h_k(x)\ to\ 1}$

Approximate membership testers (2)

- In order for $\boldsymbol{y}$ **to belong to** $\boldsymbol{X}$ it has to be **true that** $\boldsymbol{h_1(y) = h_2(y) = \cdots = h_k(y) = 1}$
  - can return false positives
  - under the **uniform hashing assumption** ($h_i(x)$ are independent and uniformly distributed), $|X| = n$, $b = (\log_2 e)kn$ bits $\rightarrow$ the error rate is upper bounded by 2^{-k} (Bloom (1970), Carter et al. (1978) and Mullin (1983))

- **Deletion not available** since that might modify other inserted values