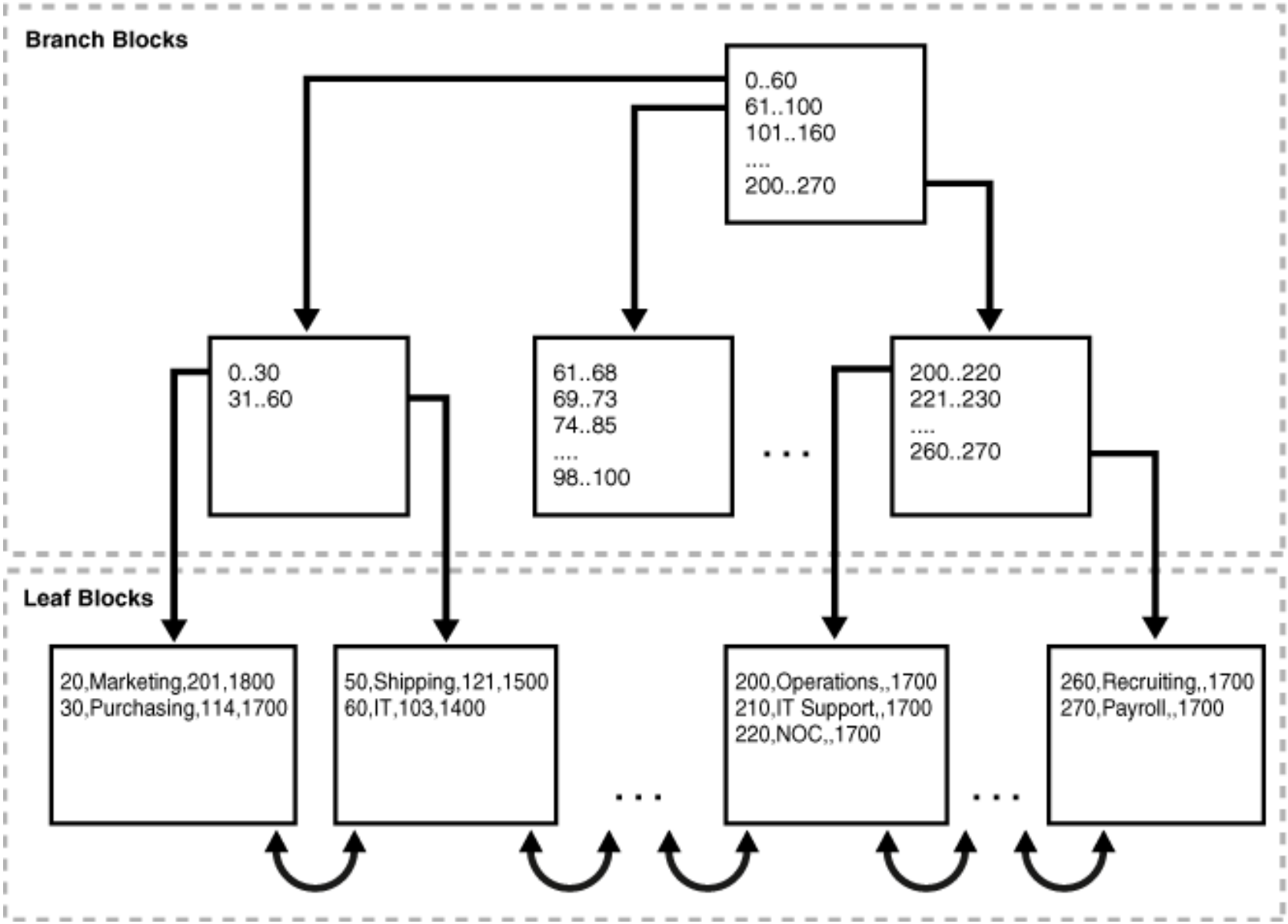
CI-5313
Arquitectura y Administración de Bases de Datos

Clase 4 – Indices (II)

Prof. Edna Ruckhaus

Láminas Prof. José Tomás Cadenas

Archivo "Clustered" - Primera alternativa



Archivo "Clustered" (costos)

- Registros con valores de clave adyacentes son almacenados físicamente "cercanos", son un cluster. Páginas 67% llenas. F – FanOut

Costo de Operaciones

- Scan completo: 1.5B*(D+bfr*C)
- Búsqueda por igualdad sobre el atributo "ordenado" (no único): $D*\log_F 1.5B+C*\log_2 bfr$
- Búsqueda por rango: el costo total es el costo de realizar la búsqueda, mas el costo de traerse el primer bloque que contiene el primer registro del rango, mas el costo de acceder todos los otros registros del rango, lo cual puede implicar mas accesos a discos, si son muchos registros

Archivo "Clustered" (costos)

- Insert: Costo de búsqueda + 1 Write:
  $D*\log_F 1.5B+C*\log_2 bfr+D$
- Delete: Igual al Insert.

Indice B+ unclustered (costos)

Costo de Operaciones

- Scan completo: B_{hoja}*(D+bfr_{hoja}*C)+B*bfr(D+C) – Se calcula B_{hoja} para hojas 67% llenas.
- Búsqueda por igualdad sobre el atributo (no único): D*$log_F B_{hoja}$ +C*$log_2 bfr_{hoja}$+D ($log_F B_{hoja}$ es # niveles)
- Búsqueda por rango: el costo total es el costo de realizar la búsqueda, mas el costo de traerse la primera hoja que contiene la primera clave del rango, mas el costo de acceder las siguientes hojas del rango,y acceder cada registro de datos lo cual puede implicar mas accesos a discos, si son muchos registros los que satisfacen la condición.

Indice B+ Unclustered (costos)

- Insert:
  - Insertar en el archivo Heap: 2D+C
  - Insertar en el indice $D*\log_F B_{hoja}+C*\log_2 bfr_{hoja}+D$
- Delete:
  - Buscar en el indice: $D*\log_F B_{hoja}+C*\log_2 bfr_{hoja}+D$
  - Escribir en indice y archivo: 2D

Ejercicio Arbol B+

Apuntador a nodo: 6 bytes, apuntador registro: 7 bytes, 30.000 registros

Suponga que el archivo no está ordenado por el campo clave CI y se quiere construir una estructura de acceso de árbol B^+ sobre CI. Calcule:

i. El orden p y p-hoja del arbol B^+
ii. El número de bloques necesarios a nivel de hoja si los bloques se llenan en un 69% aproximadamente
iii. El número de hojas necesarias si los nodos internos están llenos al 69%
iv. El número total de bloques requeridos por el árbol B^+
v. El número de bloques que se requieren acceder para buscar y recuperar un registro del árchivo, dado un valor de CI

Árbol B^+

i. Para un B +-tree de orden p, $(p * P) + ((p - 1) * V_{CI}) < B$, or $(p * 6) + ((p - 1) * 9) < 512$, lo cual da $15p < 512$, p=34
   Para nodos hoja, con apuntador a registro de datos incluidos en las hojas, $(p_{hoja} * (V_{CI}+P_R)) + P < B$, or $(p_{hoja} * (9+7)) + 6 < 512$, lo cual da $16p_{hoja} < 506$, $p_{hoja} = 31$
ii. Suponiendo que nodos 69% en promedio, el número promedio de valores de clave en un nodo hoja es $0.69*p_{hoja} = 0.69*31 = 21.39$. Redondeando a 22 valores de clave (y 22 apuntadores registro) por nodo hoja. Se tienen 30.000 registros y 30.000 valores de CI, el número de bloques hojas que se necesitan son b_1 = ceiling(30000/22) = 1364 bloques.

Árbol B+

iii. Número de niveles:
Fan-out fout promedio = ceiling(0.69*p) = ceiling(0.69*34) = ceiling(23.46) = 24 Número de bloques segundo nivel b_2 = ceiling(b_1 / fout) = ceiling(1364/24) = 57 bloques

Número de bloques 3er nivel b_3 = ceiling(b_2 /fout) = ceiling(57/24)= 3
Número de bloques 4º nivel b_4 = ceiling(b_3 /fout) = ceiling(3/24) = 1
x = 4 niveles (contando el nvel hoja).

Se puede usar la fórmula:
$x = \text{ceiling}(\log_{fout}(b_1)) + 1 = \text{ceiling}(\log_{24} 1364) + 1 = 3 + 1 = 4$ niveles

iv. Número total de bloques árbol $b_i = b_1 + b_2 + b_3 + b_4 = 1364 + 57 + 3 + 1 = 1425$ blocks

v. Número de bloques para acceder al registro = x + 1 = 4 + 1 = 5

Hash-based Indexes

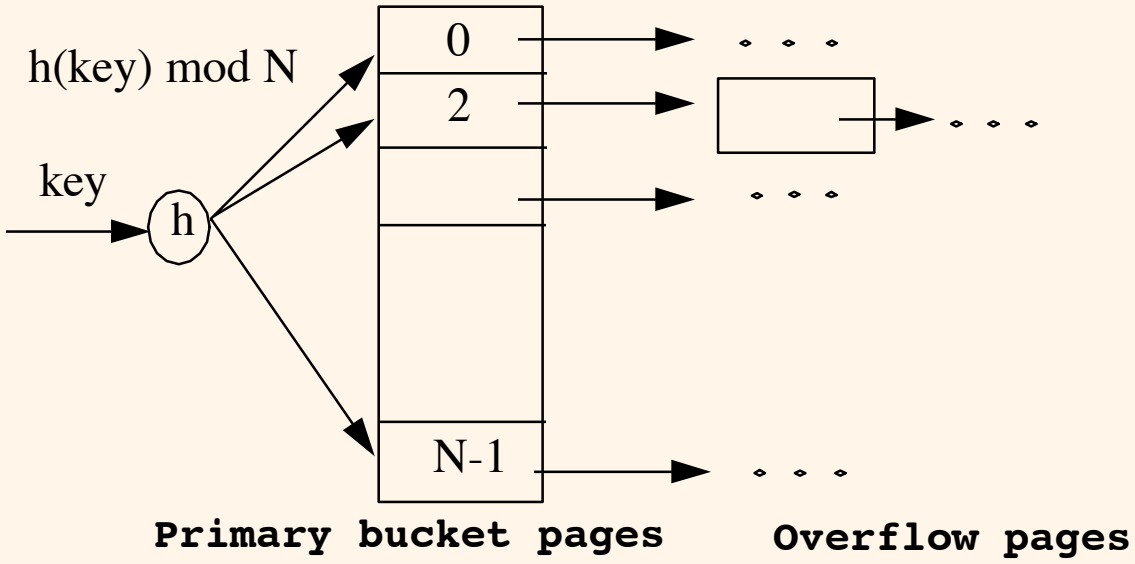
- Hash-based indexes are best for equality selections.
- Cannot support range searches.
- Static and dynamic hashing techniques exist;

Static ≈ Hashed Files

- Hashing for disk files is called External Hashing
- The file blocks are divided into M equal-sized buckets, numbered $bucket_0$, $bucket_1$, ..., $bucket_{M-1}$.
- Typically, a bucket corresponds to one (or a fixed number of) disk block.
- One of the file fields is designated to be the hash key of the file.
- The record with hash key value K is stored in bucket i, where i=h(K), and h is the hashing function.
- Collisions occur when a new record hashes to a bucket that is already full. An overflow file is kept for storing such records.

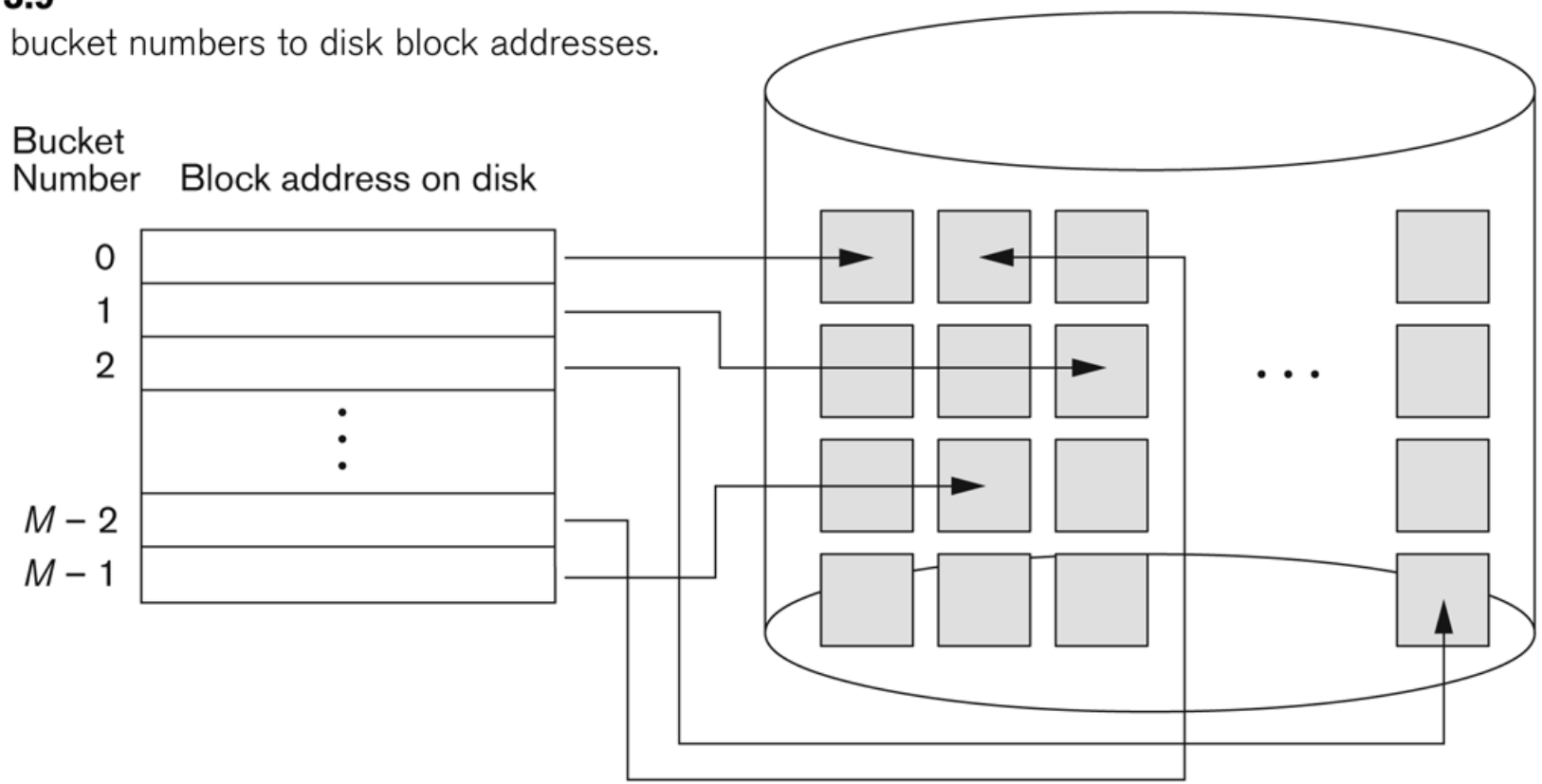
Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod M = bucket to which data entry with key *k* belongs. (M = # of buckets)



Hashed Files (contd.)

Figure 13.9
Matching bucket numbers to disk block addresses.



Archivo Hash (costos)

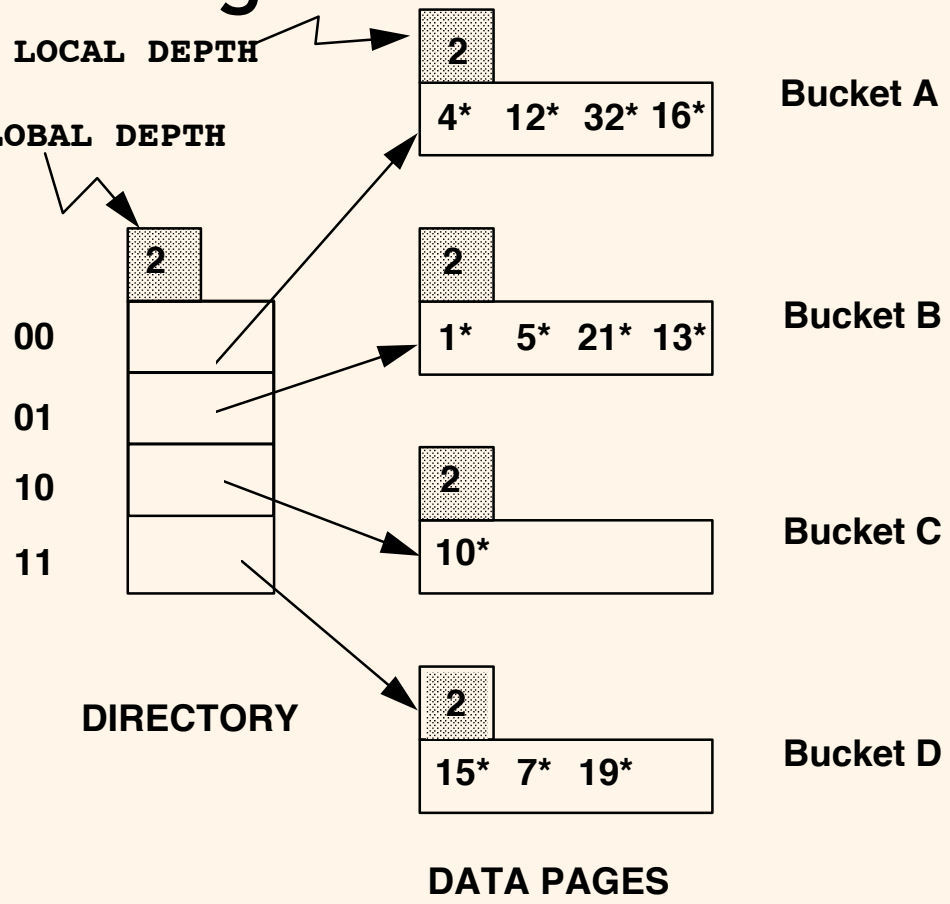
- Se supone 80% de ocupación de bloques , por lo que el tamaño total es 1.25 el tamaño original de los datos
- Scan completo: 1.25*B*(D + bfr*C)
- Búsqueda por igualdad: H+D+C*bfr/2, si el campo búsqueda es la clave de hash con la cual se organizó el archivo. H tiempo para calcular la función Hash
- Búsqueda por rango: 1.25*B*(D + bfr*C)
- Insert: Costo de la búsqueda del bloque donde insertar +D+C
- Delete: costo de la búsqueda del registro a eliminar +D+C

Static Hashing

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance.
  - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

Extendible Hashing

Example

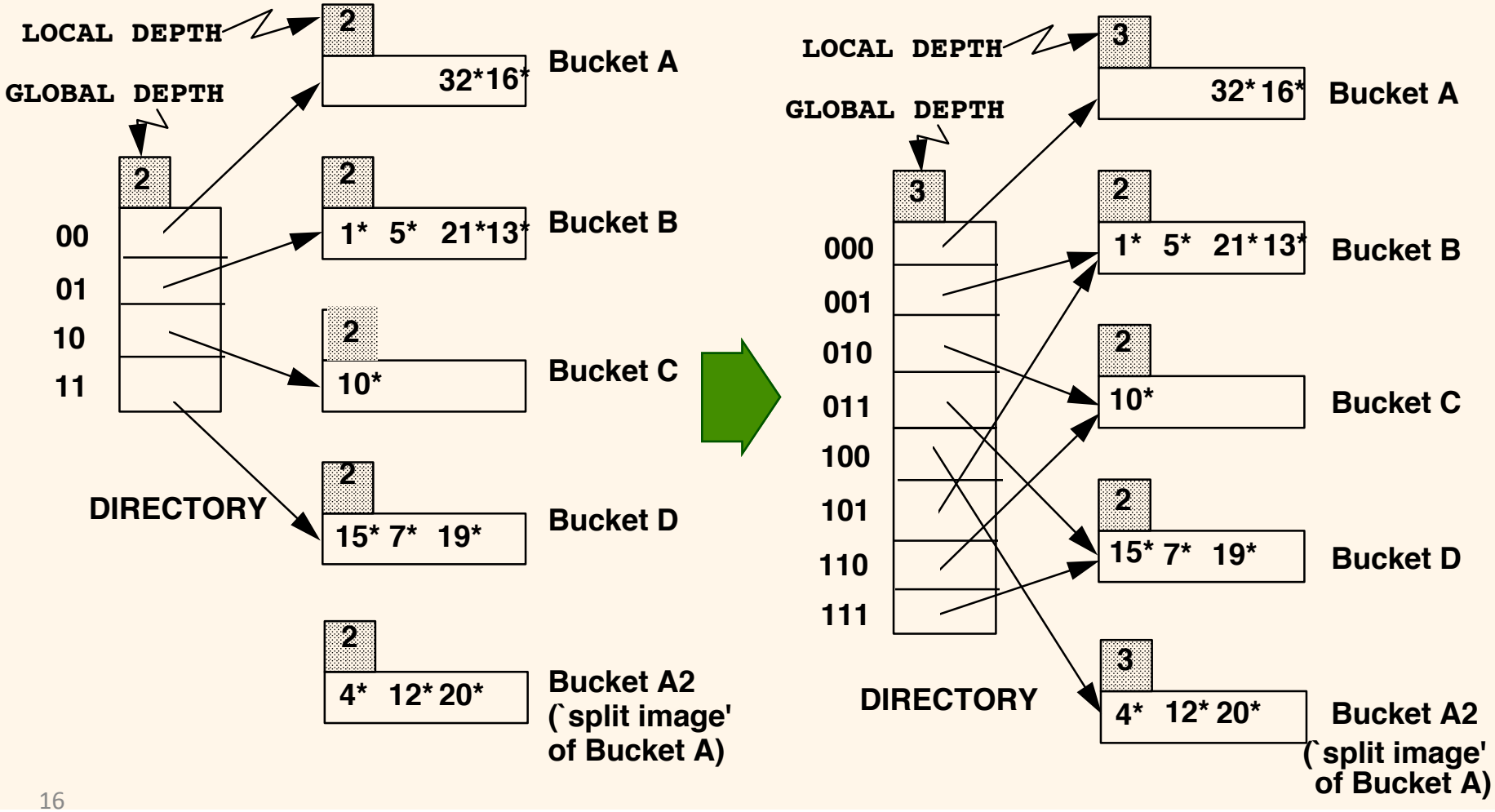


- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

- **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).
- *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Extendible Hashing

Insert h(r)=20 (Causes Doubling)



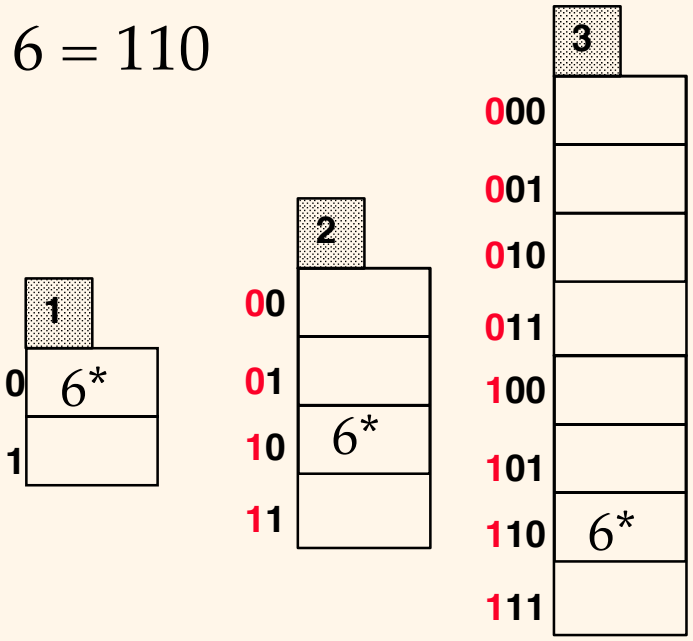
Extendible Hashing

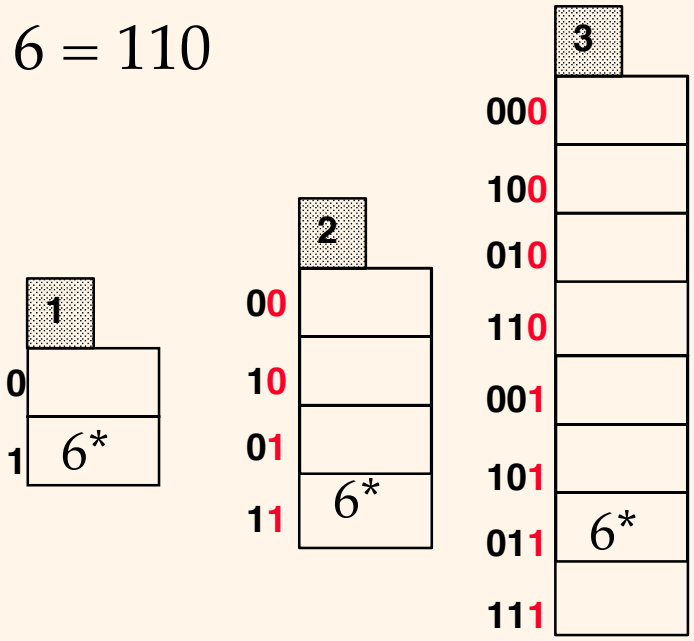
- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **3** bits needed to tell which.
  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

Extendible Hashing

Why use least significant bits in directory?

➳ Allows for doubling via copying!





Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems!
- **Delete**: If removal of data entry makes bucket empty, can be merged with `split image'. If each directory element points to same bucket as its split image, can halve directory.

Hash Lineal

- Sin necesidad de usar un directorio aumenta o disminuye dinámicamente el número de buckets dispuestos para el archivo. Cuando hay una colisión que obliga a usar el área de overflow en cualquier bucket, se comienza a subdividir un bucket de los originales.
  - La primera vez que hay una colisión se subdivide el bucket 0, creándose un bucket M y repartiendo los registros residentes en el 0 entre él mismo y el nuevo M.
  - El siguiente uso de overflow hace que se subdivida el siguiente bucket, es decir, el 1 y se crea el M+1.
  - Así se sigue linealmente si hace falta. En cada subdivisión se utiliza una nueva función de hash para redirigir los registros al bucket que está siendo dividido y al nuevo bucket creado en la subdivisión

Linear Hashing

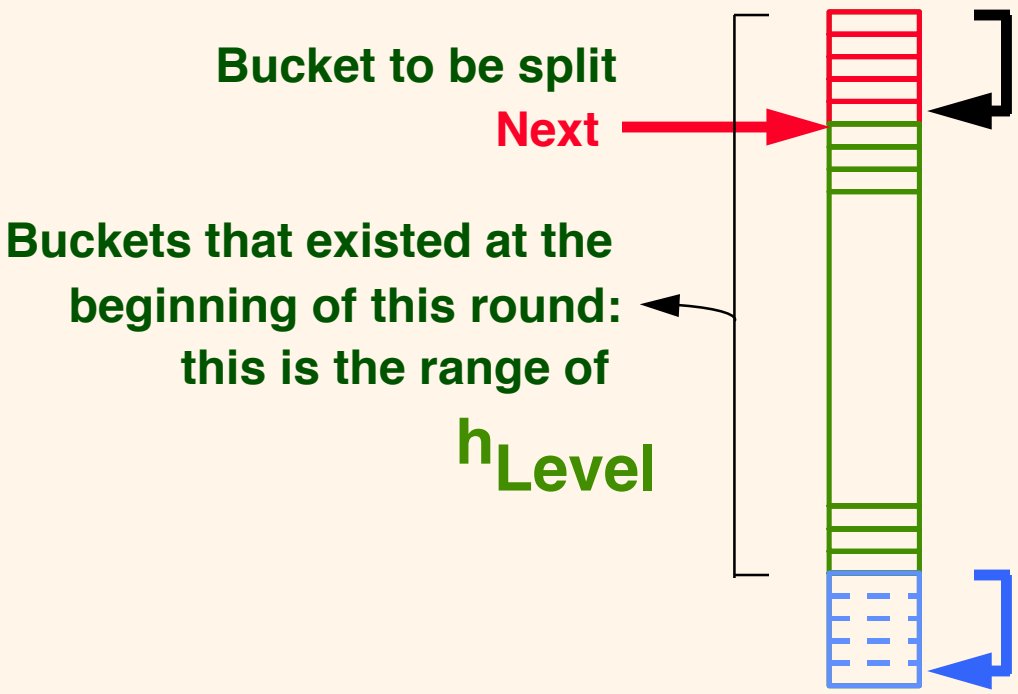
- This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- LH handles the problem of long overflow chains without using a directory, and handles duplicates.
- *Idea*: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$; N = initial # buckets
  - **h** is some hash function (range is *not* 0 to N-1)
  - If $N = 2^{d0}$, for some *d0*, $\mathbf{h}_i$ consists of applying **h** and looking at the last *di* bits, where $di = d0 + i$.
  - $\mathbf{h}_{i+1}$ doubles the range of $\mathbf{h}_i$ (similar to directory doubling)

Linear Hashing

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
  - Current round number is *Level*.
  - **Search:** To find bucket for data entry r, find $\mathbf{h}_{Level}(r)$:
    - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R', r belongs here.
    - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.

Linear Hashing

- In the middle of a round.



Buckets split in this round:
If h_{Level} (search key value) is in this range, must use $h_{Level+1}$ (search key value) to decide if entry is in `split image' bucket.

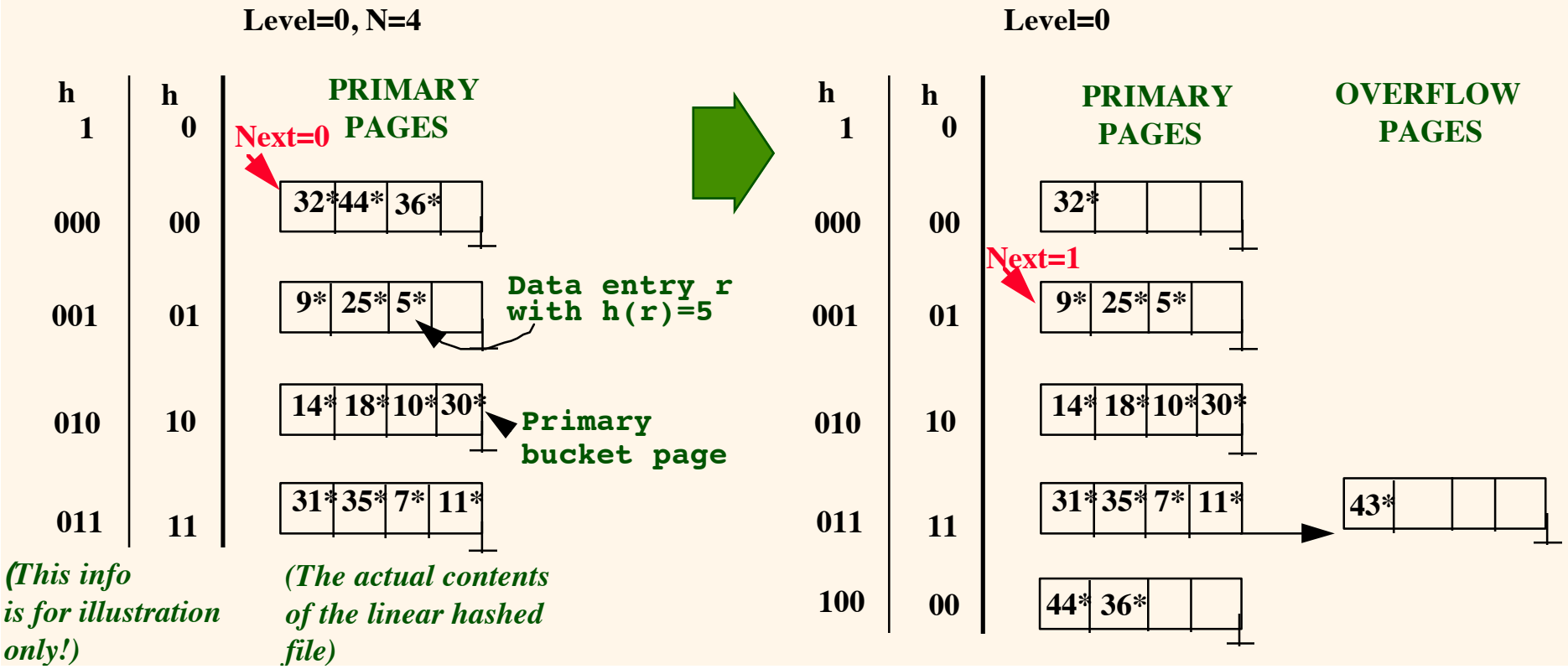
`split image' buckets: created (through splitting of other buckets) in this round

Linear Hashing

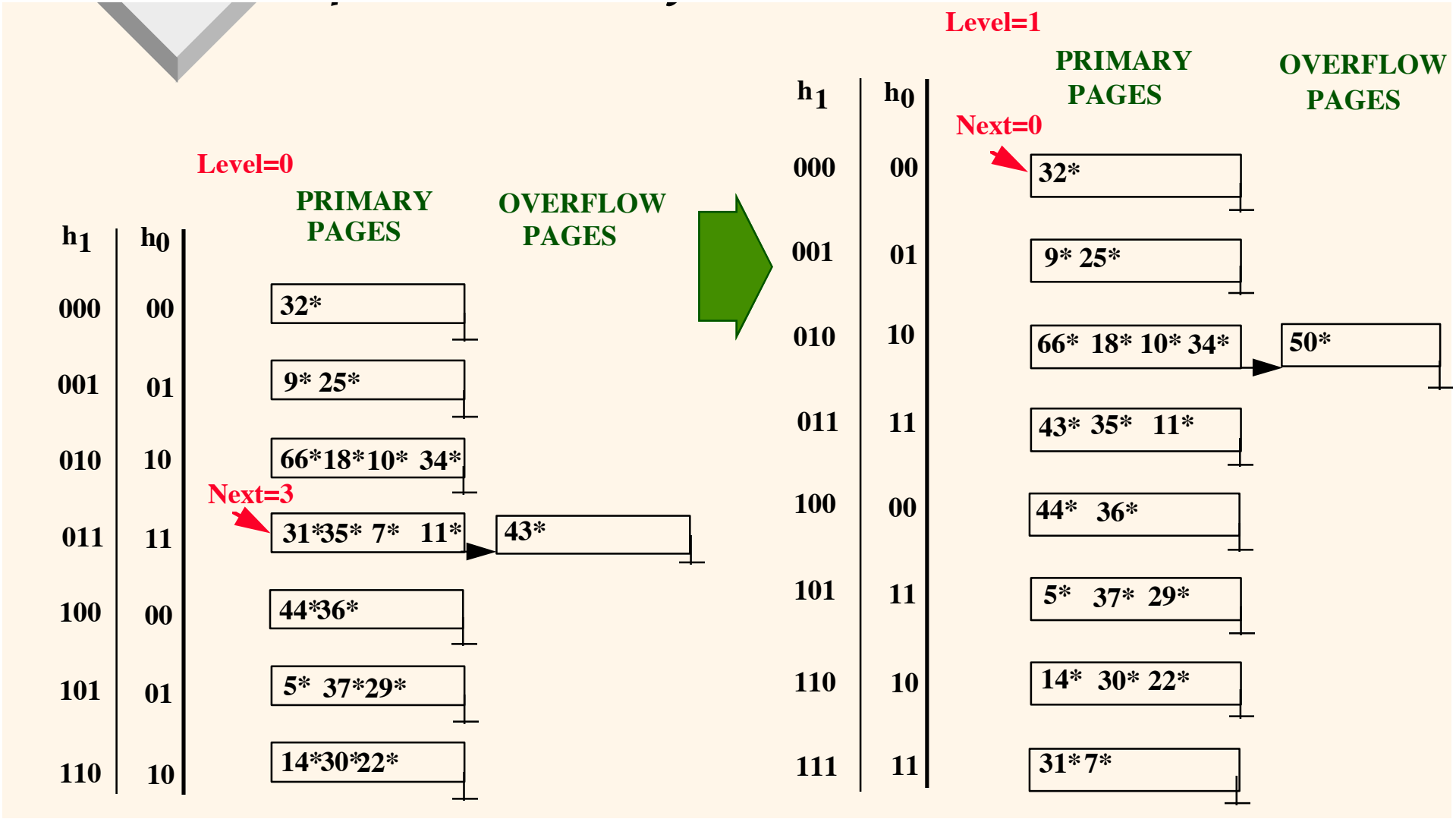
- **Insert**: Find bucket by applying $\mathbf{h}_{Level}$ / $\mathbf{h}_{Level+1}$:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (*Maybe*) Split *Next* bucket and increment *Next*.
- Can choose any criterion to `trigger' split.
- Since buckets are split round-robin, long overflow chains don't develop!
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.

Linear Hashing

- On split, $\mathbf{h}_{Level+1}$ is used to re-distribute entries.



Linear Hashing



Ejercicio Archivo Hash Estático

- Un archivo de Partes con NoParte tiene una clave hash e incluye los siguientes registros con valores en NoParte: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428,3943, 4750, 6975, 4981, 9208. El archivo usa 8 buckets, numerados del 0 al 7. Cada bucket está en un bloque de disco y contiene dos registros
- Cargar estos registros en el archivo en el orden dado utilizando la función de hash h(K)=K mod 8.
- Calcular el número de bloques promedio accedidos para una recuperación aleatoria sobre el No. Parte

Ejercicio Archivo Hash Estático

Valor No. Parte (K)	h(K) No. Bucket
2369	1
3760	0
4692	4
4871	7
5659	3
1821	5
1074	2
7115	3
1620	4
2428	4 (overflow)
3943	7
4750	6
6975	7 (overflow)
4981	5
9208	0

Número de bloques promedio accedidos para una recuperación aleatoria sobre el No. Parte = 17/15 = 1.133

Ejercicio Hashing Extensible

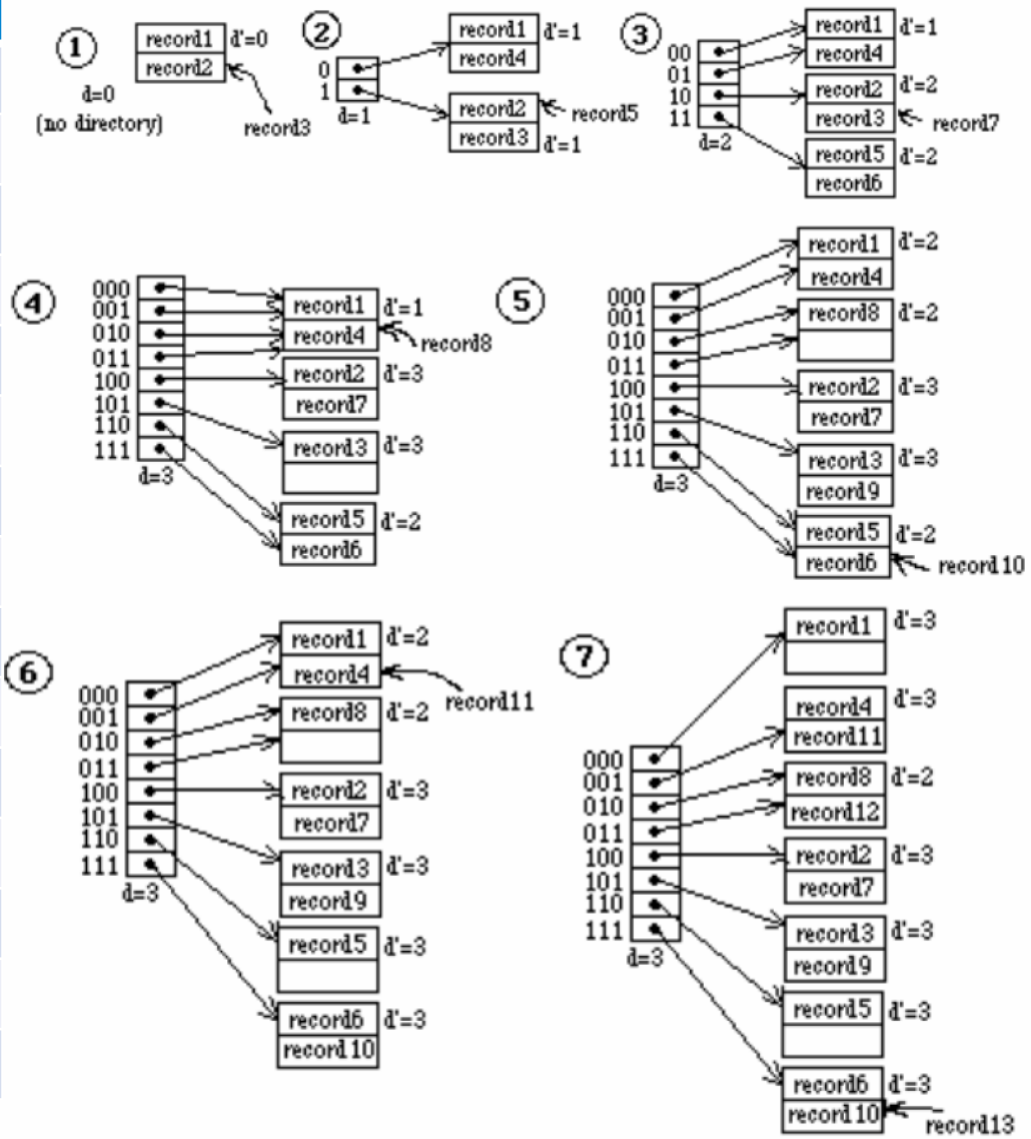
- Cargar los registros del ejercicio anterior en un archivo hash basado en hashing extensible.
- Mostrar la estructura del directorio en cada paso y las profundidades locales y globales.
- Usar la función hash h(K)= K mod 32

Ejercicio Hashing Extensible

Reg. No.	Valor No. Parte (K)	h(K)	h(K) binario
1	2369	1	00001
2	3760	16	10000
3	4692	20	10100
4	4871	7	00111
5	5659	27	11011
6	1821	29	11101
7	1074	18	10010

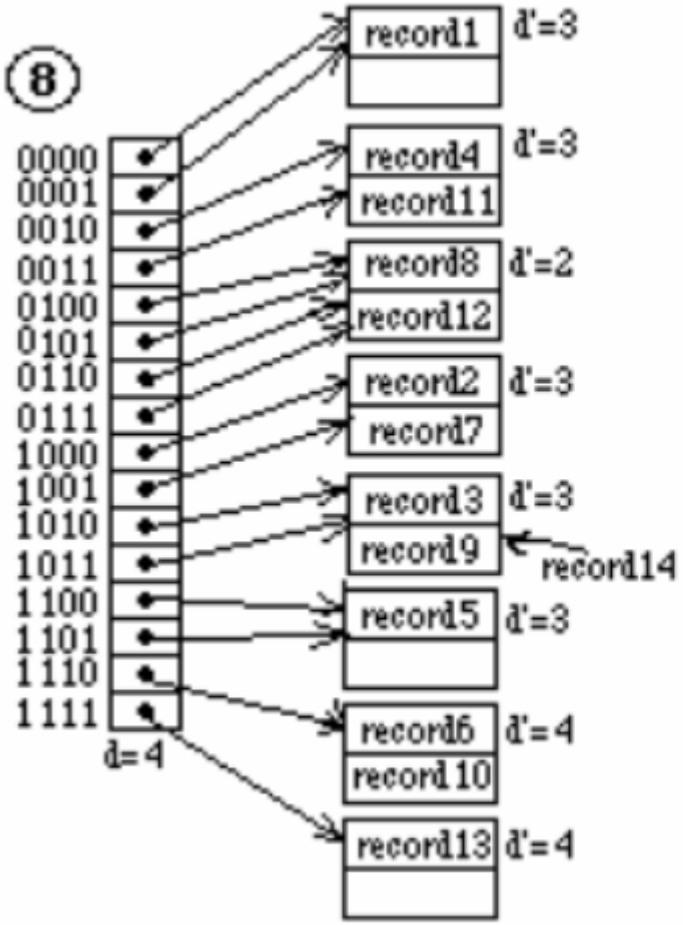
Ejercicio Hashing Extensible

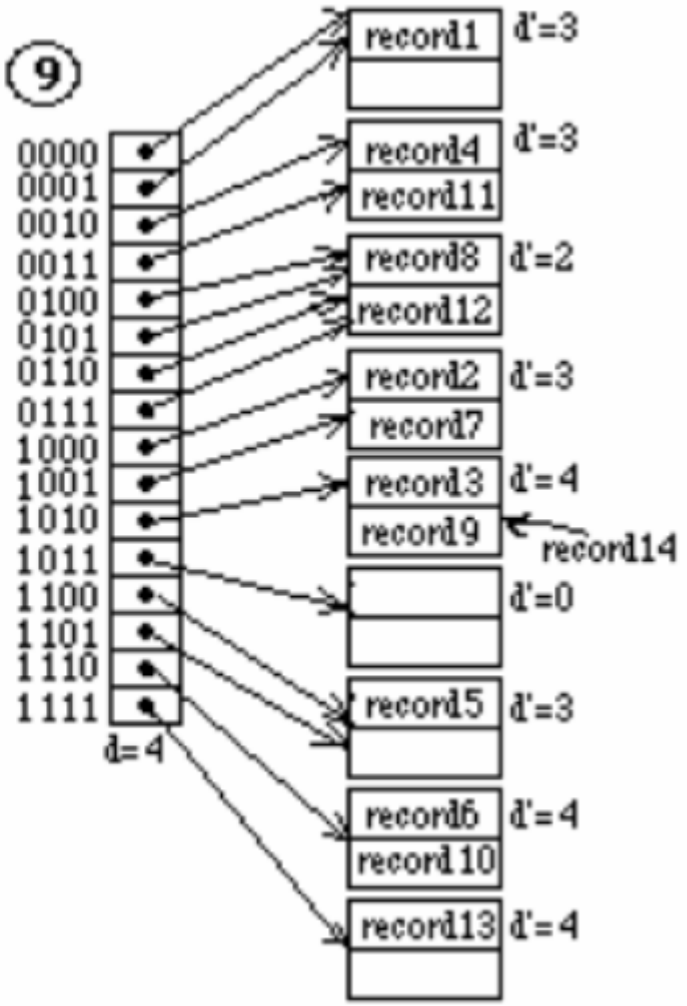
Reg. No.	Valor No. Parte (K)	h(K)	h(K) binario
1	2369	1	00001
2	3760	16	10000
3	4692	20	10100
4	4871	7	00111
5	5659	27	11011
6	1821	29	11101
7	1074	18	10010
8	7115	11	01011
9	1620	20	10100
10	2428	28	11100
11	3943	7	00111
12	4750	14	01110
13	6975	31	11111
14	4981	21	10101
15	9208	24	11000



Ejercicio Hashing Extensible

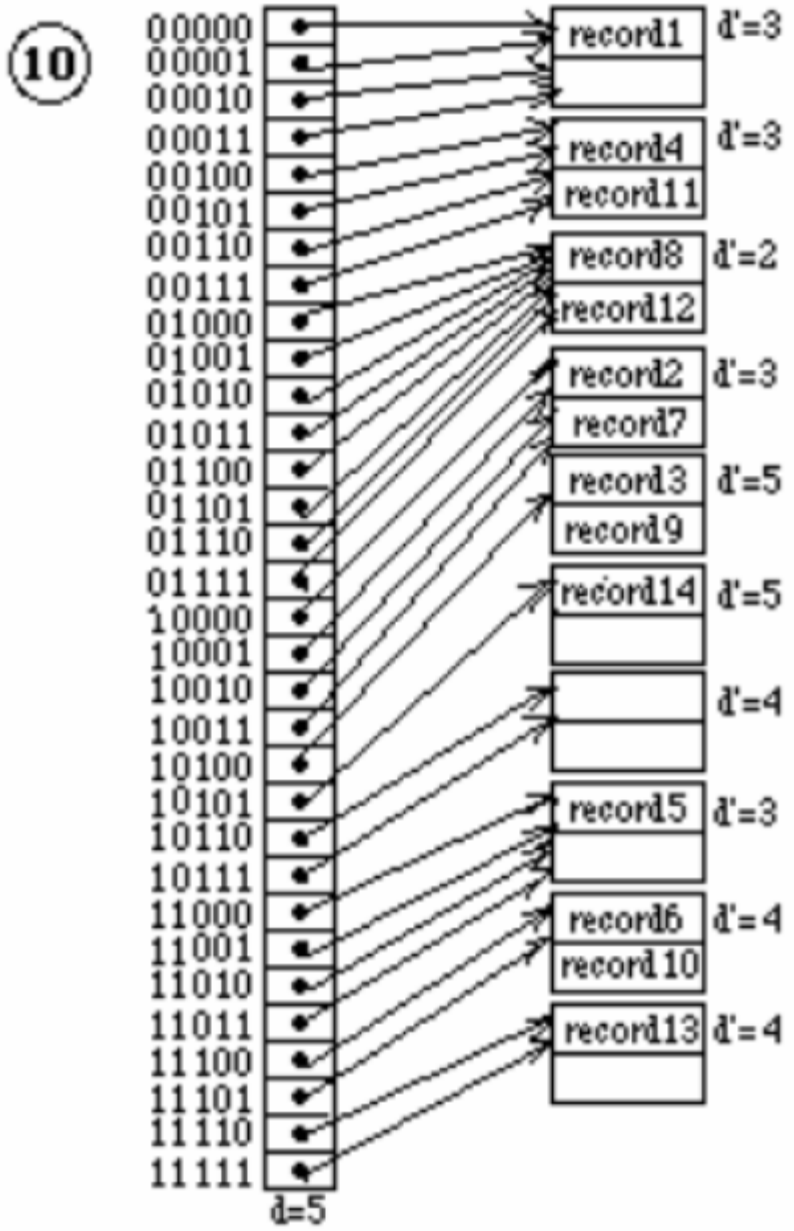
Reg. No.	h(K) binario
1	00001
2	10000
3	10100
4	00111
5	11011
6	11101
7	10010
8	01011
9	10100
10	11100
11	00111
12	01110
13	11111
14	10101
15	11000





Ejercicio Hashing Extensible

Reg. No.	Valor No. Parte (K)	h(K)	h(K) binario
1	2369	1	00001
2	3760	16	10000
3	4692	20	10100
4	4871	7	00111
5	5659	27	11011
6	1821	29	11101
7	1074	18	10010
8	7115	11	01011
9	1620	20	10100
10	2428	28	11100
11	3943	7	00111
12	4750	14	01110
13	6975	31	11111
14	4981	21	10101
15	9208	24	11000



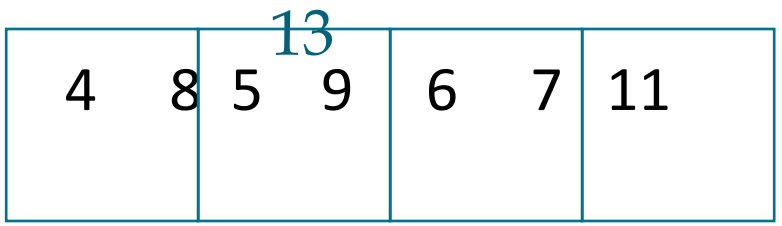
Linear Hashing: Example

Initially: h(x) = x *mod* N (N=4 here)

Assume 3 records/bucket

Insert 17 = 17 mod 4 → 1

Bucket id 0 1 2 3



Linear Hashing: Example

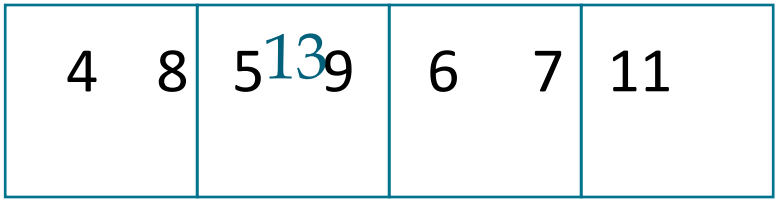
Initially: h(x) = x *mod* N (N=4 here)

Assume 3 records/bucket

Insert 17 = 17 mod 4 ⟶ 1

Overflow for Bucket 1

Bucket id 0 1 2 3

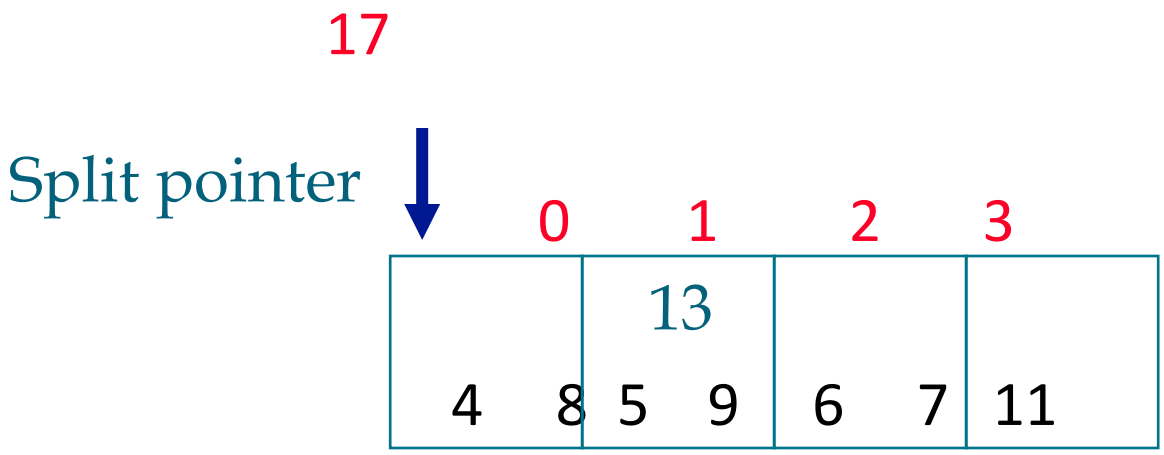


Split bucket 0, anyway!!

Linear Hashing: Example

To split bucket 0, use another function h1(x):

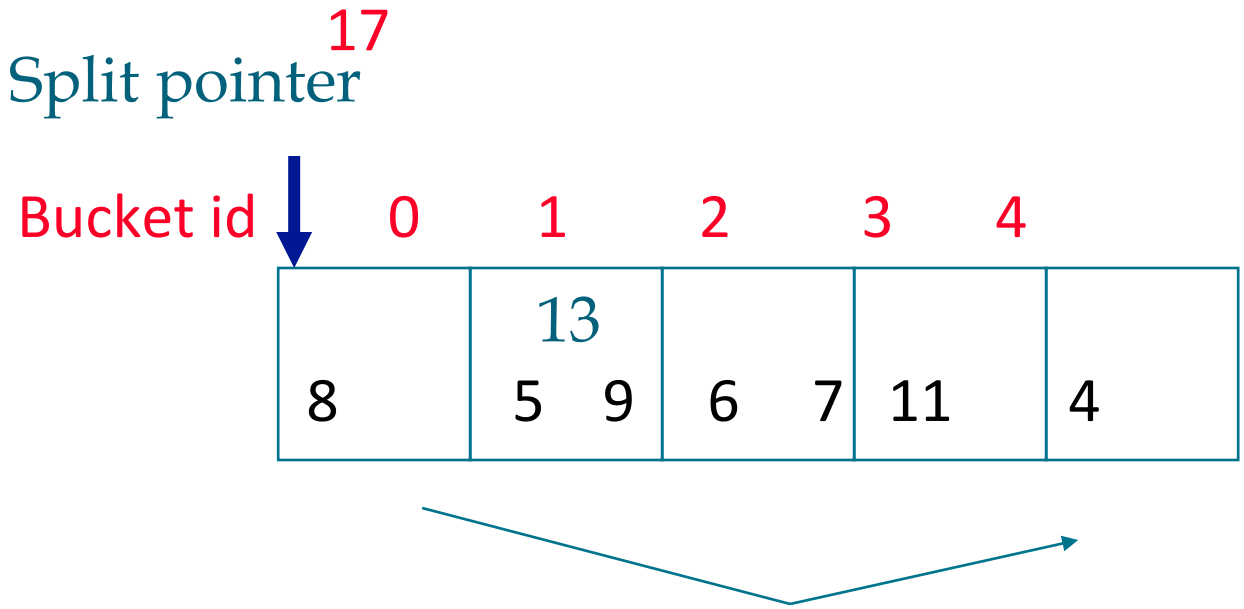
h0(x) = x mod N , h1(x) = x mod (2*N)



Linear Hashing: Example

To split bucket 0, use another function h1(x):

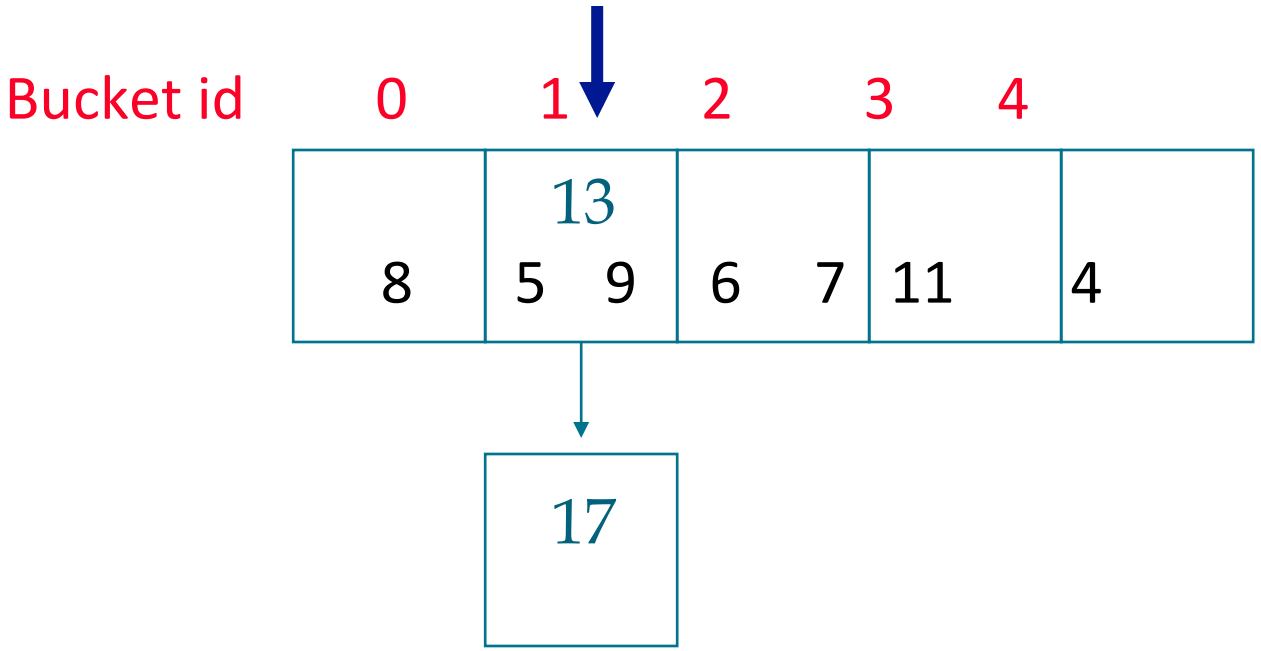
h0(x) = x mod N , h1(x) = x mod (2*N)



Linear Hashing: Example

To split bucket 0, use another function h1(x):

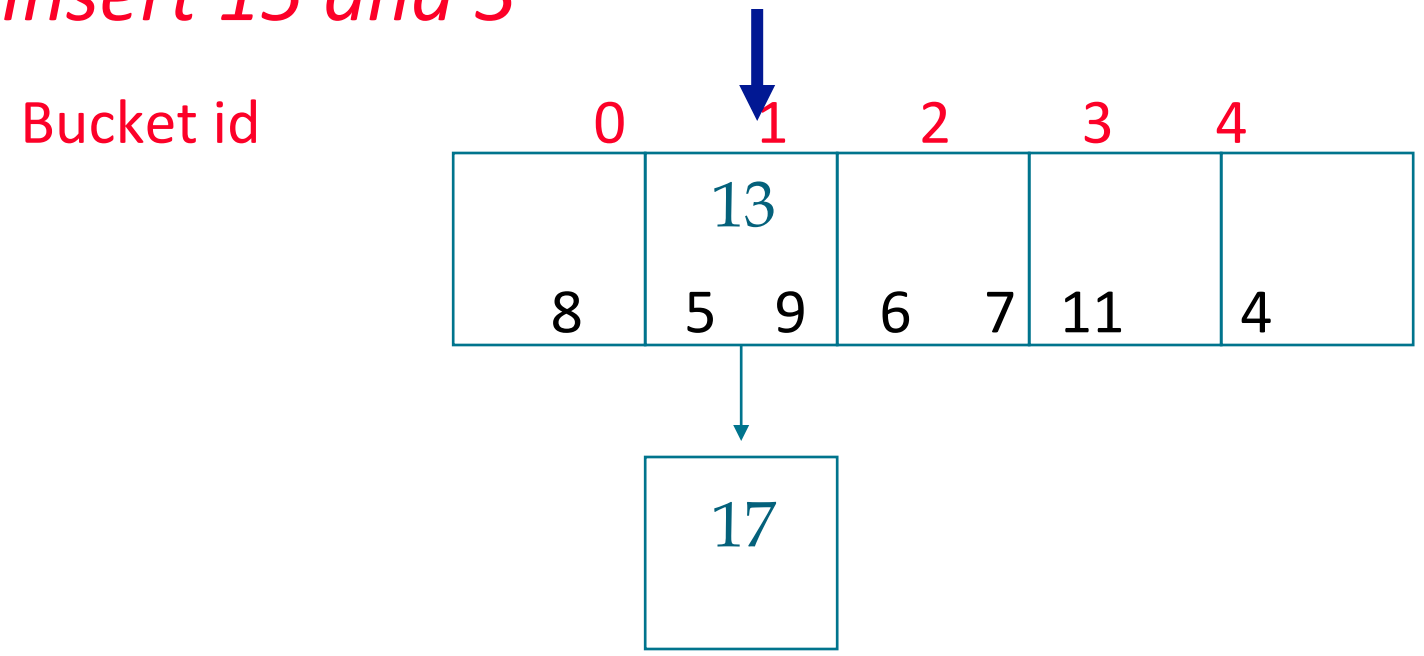
h0(x) = x mod N , h1(x) = x mod (2*N)



Linear Hashing: Example

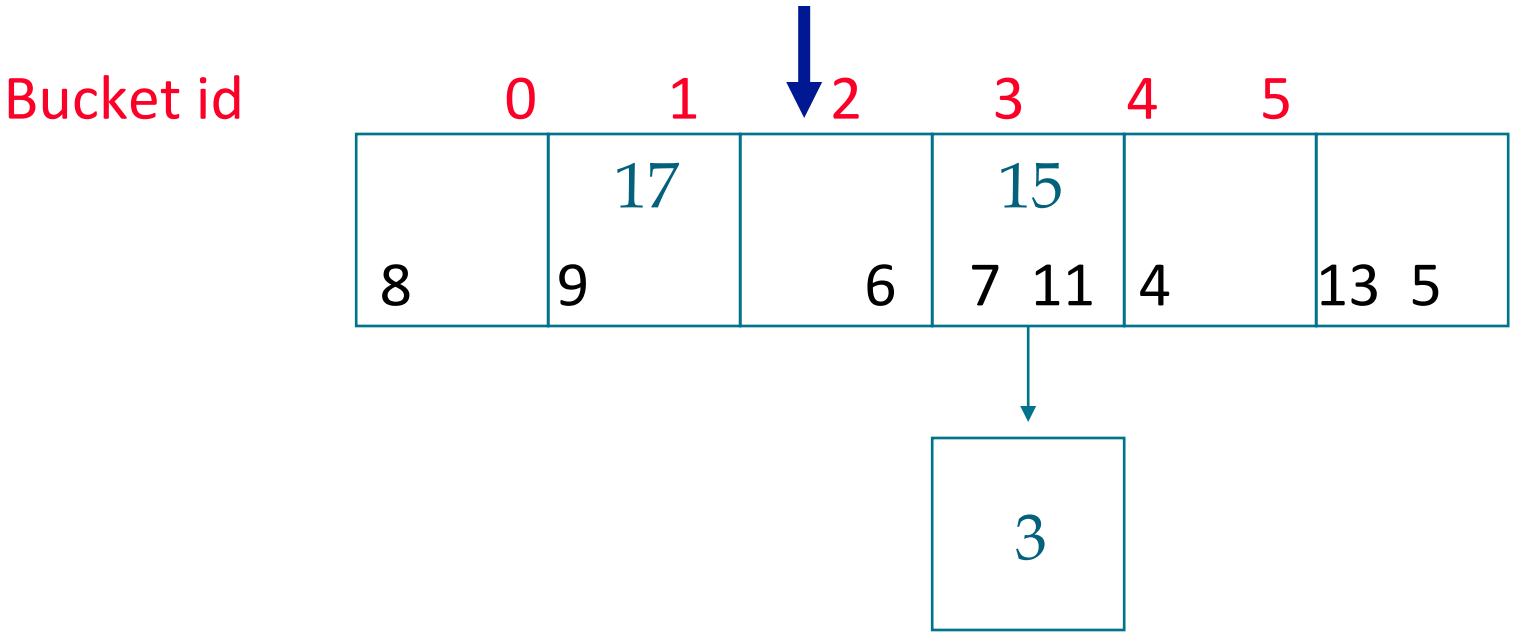
$h0(x) = x \bmod N$, $h1(x) = x \bmod (2*N)$

Insert 15 and 3



Linear Hashing: Example

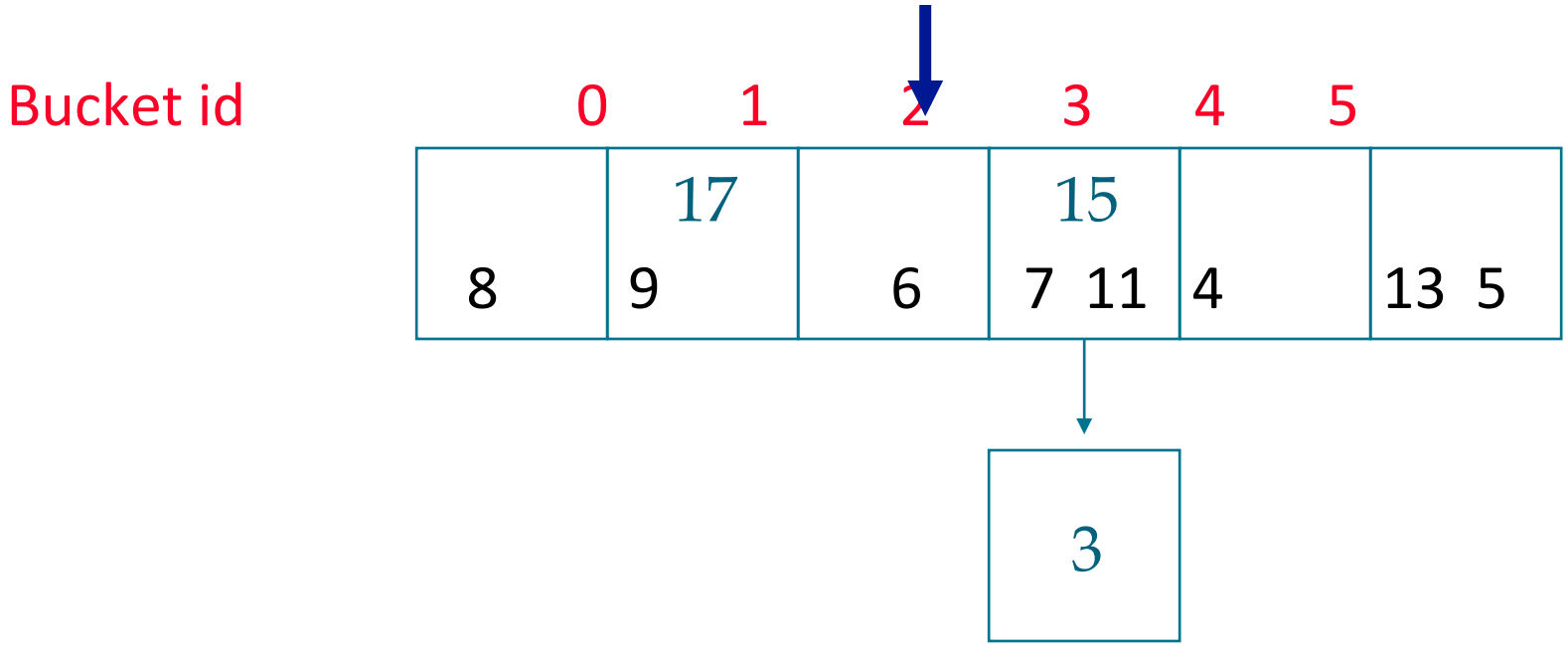
h0(x) = x mod N , h1(x) = x mod (2*N)



Linear Hashing: Example

h0(x) = x mod N (*for the un-split buckets*)

h1(x) = x mod (2*N) (*for the split ones*)



Linear Hashing: Search

Algorithm for Search:

```
Search(k)
1    b = h0(k)
2    if b < split-pointer then
3        b = h1(k)
4    read bucket b and search there
```