20. Indexing – Hashing

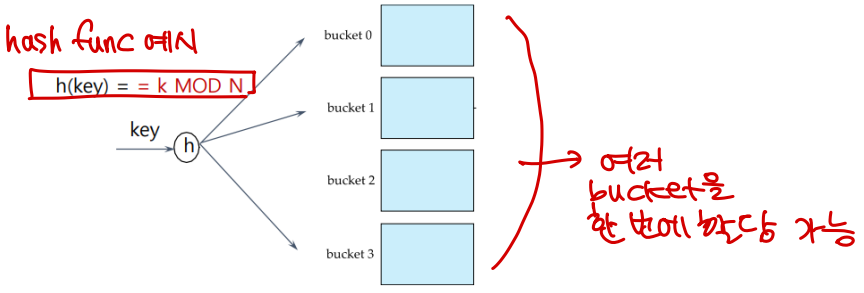
▼ Hashing

- bucket : data를 담는 공간 → data가 저장되는 단위 (보통 block 크기, 4KB)
- Hash function : bucket을 쭉 탐색해야 함 → data가 어디에 저장될지 결정.
  - h(hash func) : search-key 집합으로부터 bucket address 집합 매칭
    → data 위치를 알 수 있다 (bucket 번호를 알 수 있다!)
- 다른 serach key가 같은 bucket에 mapping 될 수 있음
  ⇒ 또한 bucket은 sequential하게 탐색 가능
- hash index : record pointer가 있는 entry 저장되어 있는 bucket 번호? 위치?
- hash-file-organization → bucket에 record 저장되어 있는 구조

▼ Static Hashing

*data 찾을 때 순서
① hash function
② bucket 위치 확인

bucket의 개수가 fixed



→ bucket이 sequentially하게 할당되어 있으며 할당된 공간은 절대 free x (재할당 불가)

▼ Handling of Bucket overflows

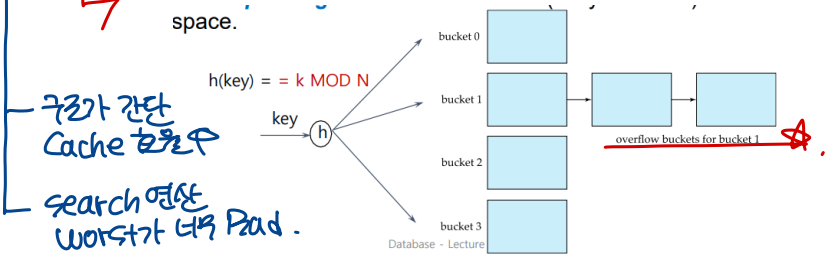
- Bucket overflow 발생 이유
  1. insufficient buckets : bucket이 너무 작거나 소수임
  2. hash function 오류(skew in distribution of records)
     - skew 발생 이유

1. multiple records : 같은 search-key를 가짐 ⇒ 미리 막을 방법이 없음
2. 선택된 hash function이 너무 단순하거나 non-uniform한 분포를 가짐

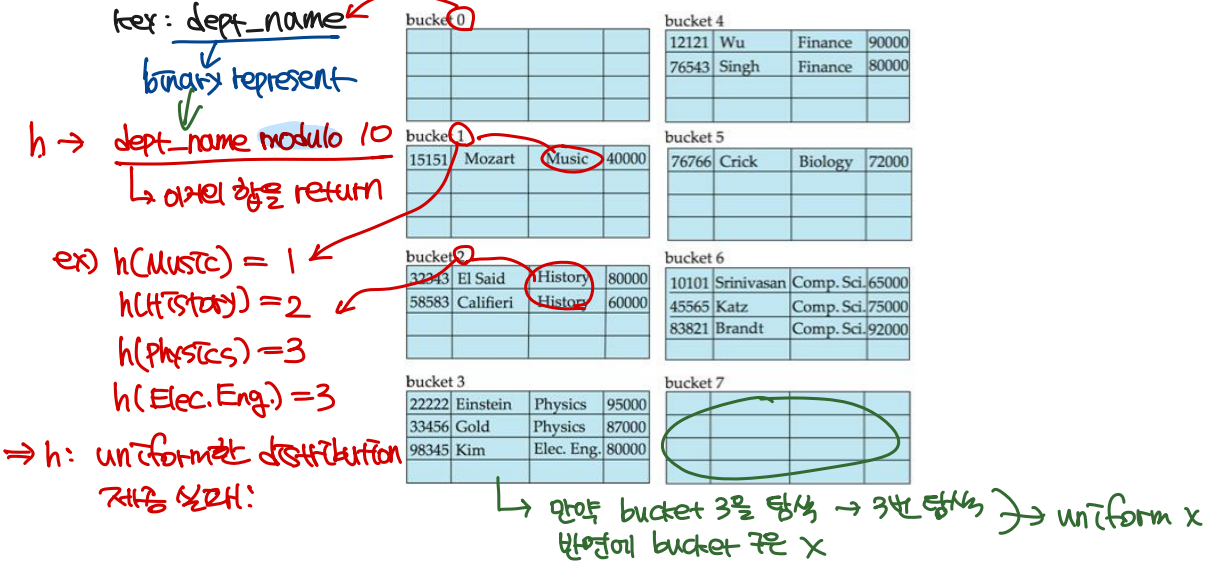
⇒ bucket overflow를 줄일 수 있지만 아에 일어나지 않게 할 수는 없음

→ overflow buckets을 사용하여 조절함

- Overflow Chaining
  - 새로운 bucket 설정 + bucket이랑 linked list로 연결
  - Closed Addressing(Closed Hashing) : overflow bucket이 linked list 안에 chained된 형태
  - Open Hashing : overflow bucket 사용하지 않음
    - ⇒ Linear Probing : space를 가지는 다음 bucket 사용(in cyclic order)



▼ Example of Hash file organization



▼ Deficiencies of Static Hashing

- static hashing → func h가 bucket address의 고정된 set에 맞게 search-key mapping

- database
  - bucket이 너무 작음 → overflow때매 성능 저하
  - bucket이 너무 큼 → underfull + diskspace 낭비

Solution

⇒ Rehasing

- 새로운 hash function으로 re-organization

  → but, expensive

⇒ better solution: **bucket수가 동적으로 수정 가능하도록!**

▼ Dynamic Hashing

- Linear Hashing : h() 수정 But, 공간 제공 X
- Extendible Hashing : h() 수정 X But " O

- Basic Idea ⇒ Bucket (dir) 2배로 증가!

  Idea Indirection의 level을 높이자!

  - dir pointer 사용
  - bucket수 더블
  - overflowed된 bucket만 split

  ⇒ Hash function이 어떻게 고정bit에 따라 달라

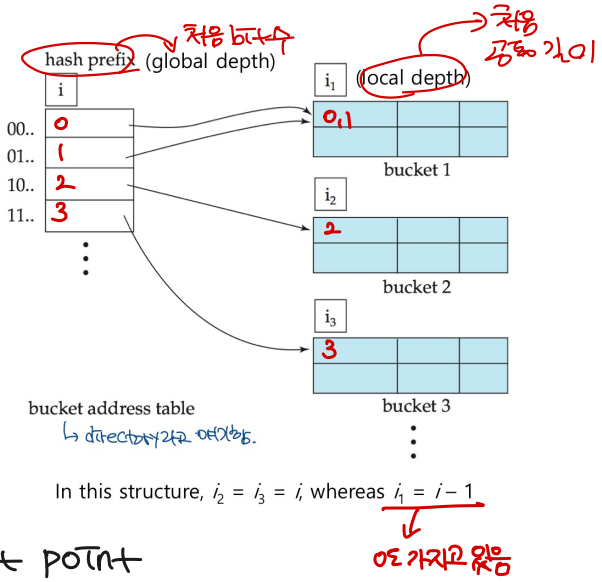
▼ Extendible Hashing

hash key의 postfix(prefix) 사용

- postfix한 길이 = i 라고 해보자

  - Dir = 2^i
  - i=0 → 최소값
  - db가 커질수록 더 커짐

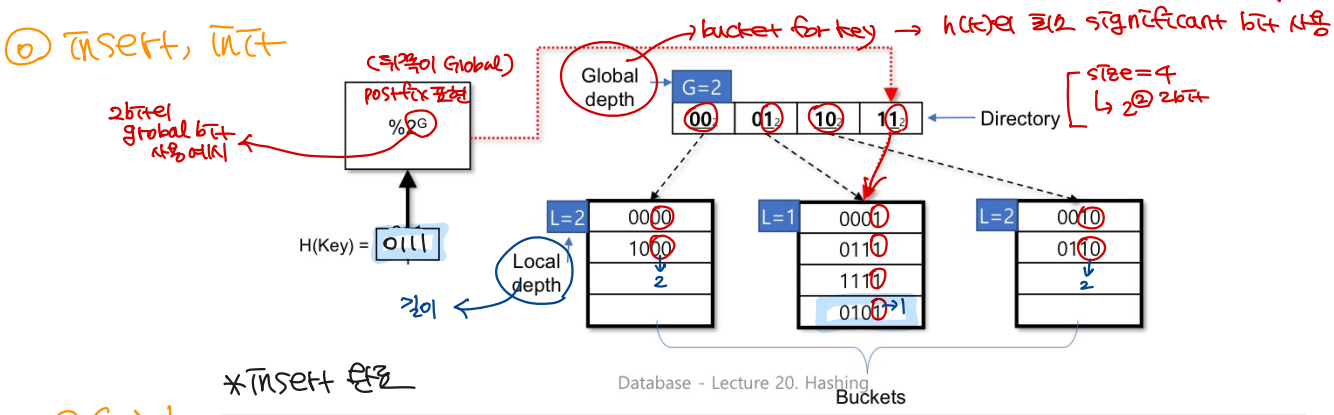
- General한 구조

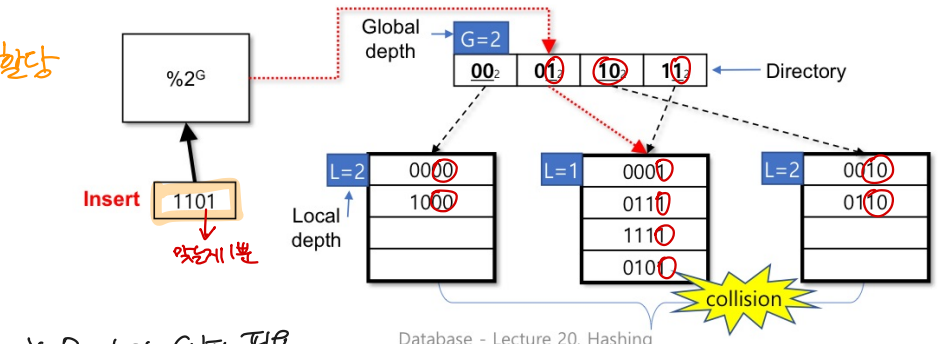


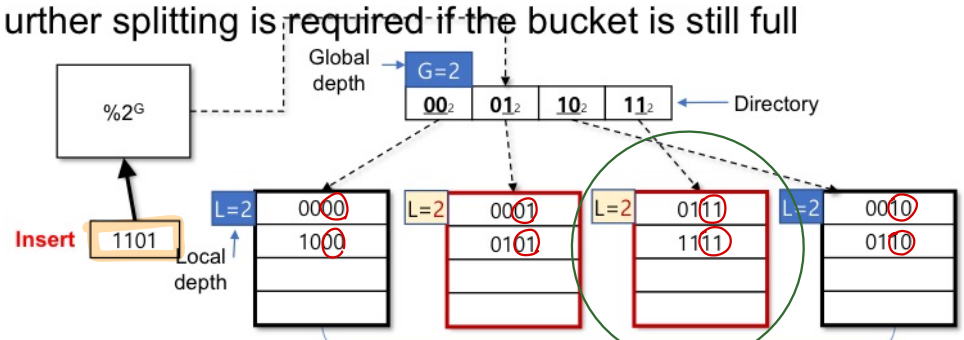
- Multiple Dir Entries : Single bucket point

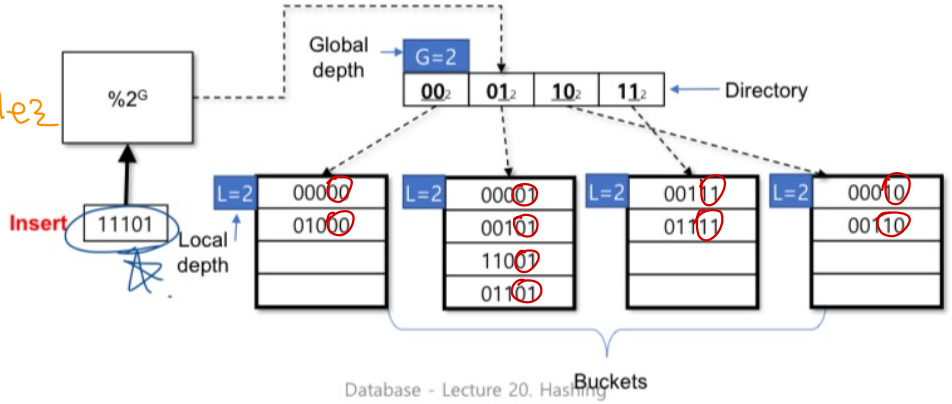
⇒ 실제 bucket 수 $< 2^i$

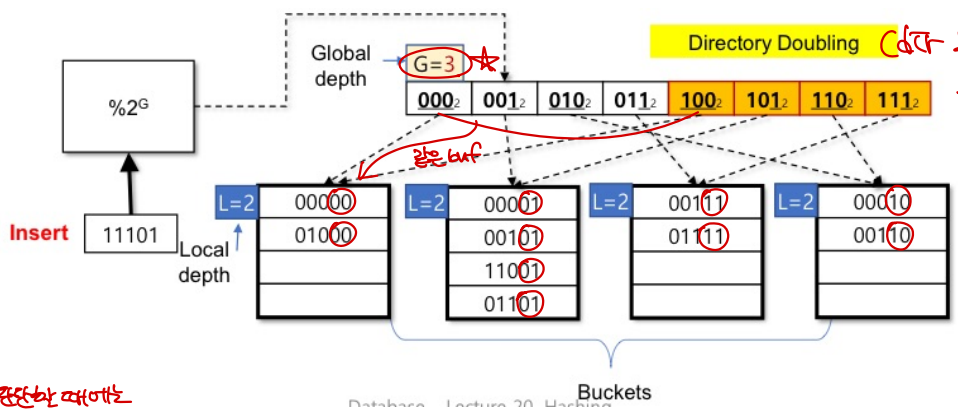
(동적e로 변경될 수 있음)


Global depth
G=2
00 01 10 11
Directory
L=2 0000 1000
L=1 0001 0111 1111 0101
L=2 0010 0110
H(Key) = 0111
Local depth
Buckets


Global depth
G=2
Directory
%2^G
Insert 1101
Local depth
L=2 0000 1000
L=1 0001 0111 1111 0101
L=2 0010 0110
collision
Buckets


urther splitting is required if the bucket is still full
Global depth
G=2
Directory
%2^G
Insert 1101
Local depth
L=2 0000 1000
L=2 0001 0101
L=2 0111 1111
L=2 0010 0110
Buckets


Global depth
G=2
Directory
%2^G
Insert 11101
Local depth
L=2 00000 01000
L=2 00001 00101 11001 01101
L=2 00111 01111
L=2 00010 00110
Buckets


Global depth
G=3
Directory Doubling
000 001 010 011 100 101 110 111
%2^G
Insert 11101
Local depth
L=2 00000 01000
L=2 00001 00101 11001 01101
L=2 00111 01111
L=2 00010 00110
Buckets

- Extendible vs. Other
  - ① 장점
    - Hash 성능이 file 크기에 비해 나빠지지 X
    - space overhead minimum
  - ② 단점
    - level의 indirection이 추가적으로 필요 (record 찾기 위함)
    - Bucket Address Table → 매우 커질 것임 (memory보다 크게!)
      - ↳ Disk에도 큰 공간 할당 X
      - ⇒ B^+tree structure 사용 → bucket address table에!
      - ⇒ But, 전체 사이즈가 엄청 커졌을 때만 효과가 있다..

▼ Linear Hashing

directory 사용하지 않고 long overflow chain 문제 해결!

⇒ 일시적인 overflow page 사용됨

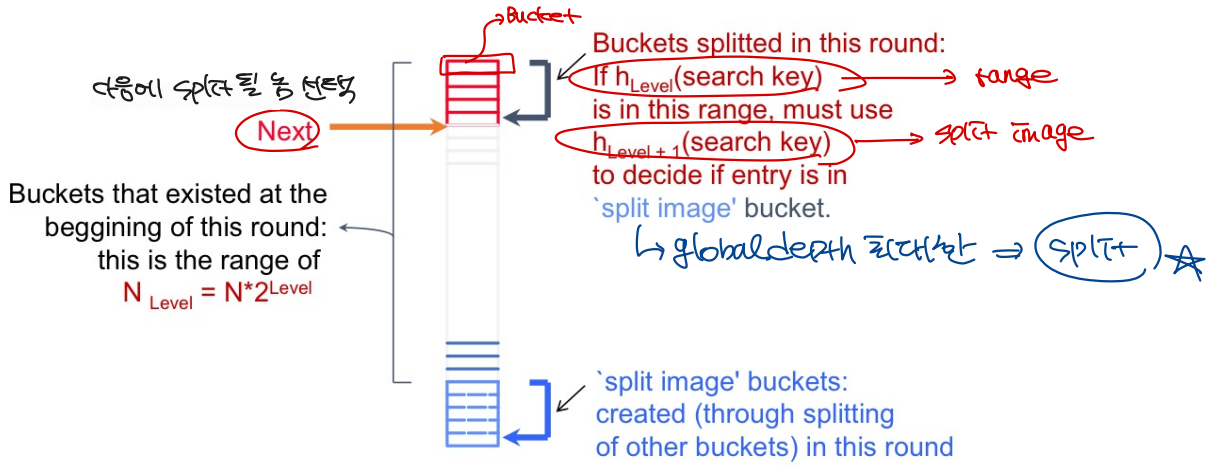
split 하기 위한 bucket ⇒ RR로 선택됨

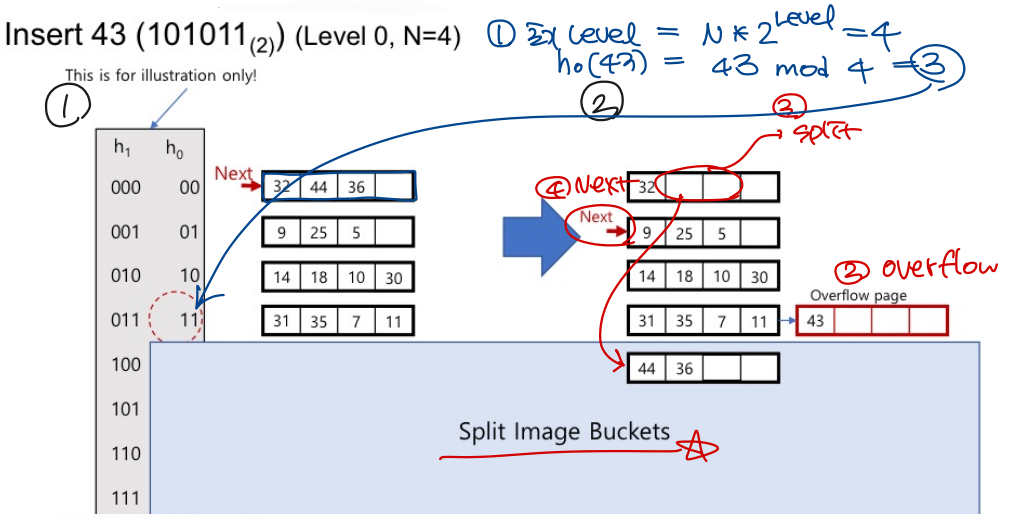
⇒ "Next"가 현재 가리키고 있는 bucket을 split한 뒤 Next pointer ++

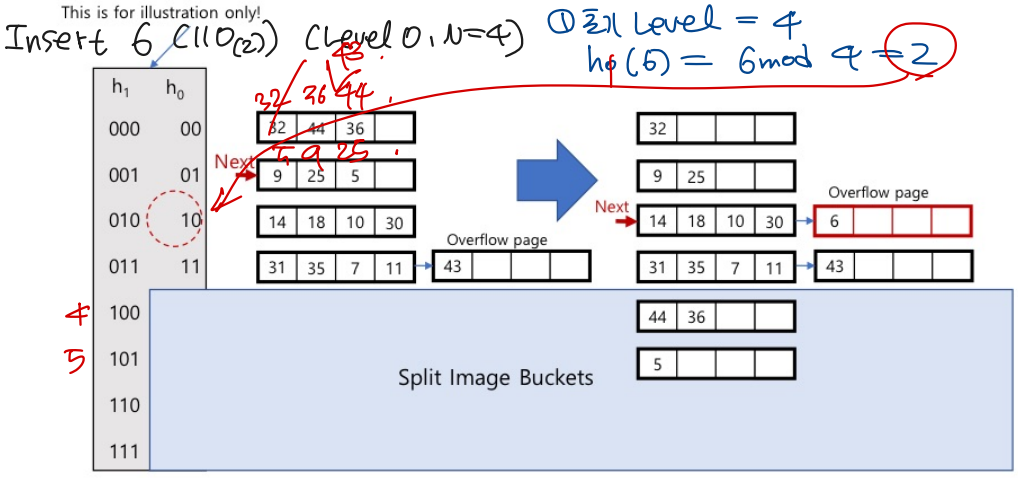
- hash function family 사용 : $h_0, h_1, h_2 \cdots$
  - $h_i(key) = (key) \bmod N$
  - N : 초기 bucket 개수 (2의 배수)
  - $h_{i+1} \leftarrow h_i$ ⟹ (dir doubling과 유사)

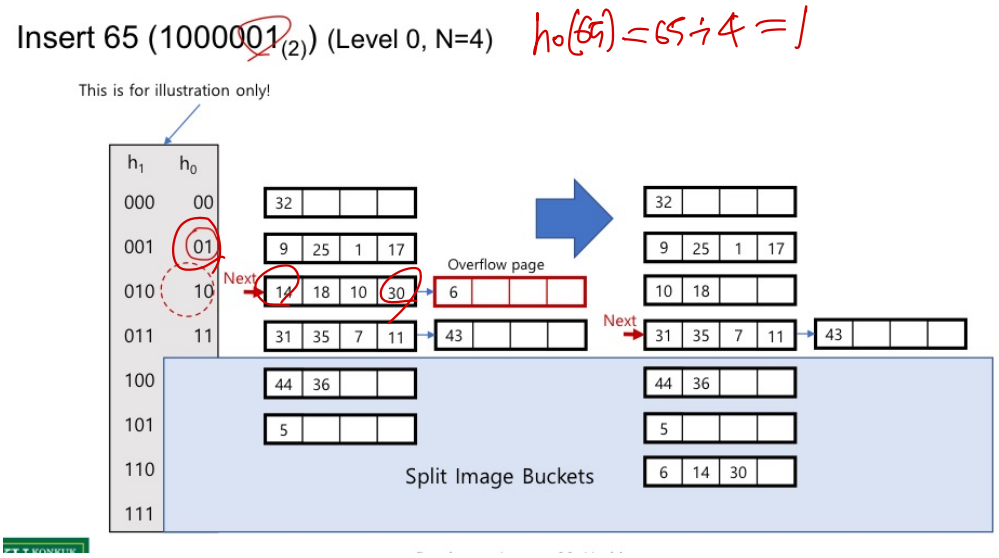
- Algorithm → round 기반
  - 초기 Level : $N \times 2^{Level}$ → h_{Level} 구함 1.
  - Bucket 0 ~ (Next−1)까지 split
  - Next ~ N_{Level}은 아직 split X
  - Next = N_{Level}
  - 다음 round : Level ++, Next = 0

- Insert → long overflow 발생 가능
  - Bucket 찾음
    - ↳ if fit → DONE
    - ↳ else
      - ↳ 일시적인 overflow page 넣고 insert data ★
      - Next bucket split → Next ++ (overflow된 곳이 아님!!)
      - re-distribute entries : h(Level+1) 사용!! ★


Bucket
다음에 split될 bucket
Next
Buckets splitted in this round:
If h_{Level}(search key) is in this range, must use
$h_{Level+1}$(search key) to decide if entry is in `split image' bucket.
range
split image
↳ global depth 증가했다 ⇒ split
Buckets that existed at the beggining of this round: this is the range of
$N_{Level} = N*2^{Level}$
`split image' buckets: created (through splitting of other buckets) in this round


Insert 43 ($101011_{(2)}$) (Level 0, N=4)
① 초기 Level = N * 2^Level = 4
h0(43) = 43 mod 4 = 3
This is for illustration only!
Next
Overflow page
Split Image Buckets
split
overflow


Insert 6 ($110_{(2)}$) (Level 0, N=4)
① 초기 Level = 4
h0(6) = 6 mod 4 = 2
This is for illustration only!
Next
Overflow page
Split Image Buckets


Insert 65 ($1000001_{(2)}$) (Level 0, N=4)
h0(65) = 65 ÷ 4 = 1
This is for illustration only!
Next
Overflow page
Split Image Buckets

• searching

i) entry r → $h_{Level}(r)$

ii) 아직 이 라운드에서 split X

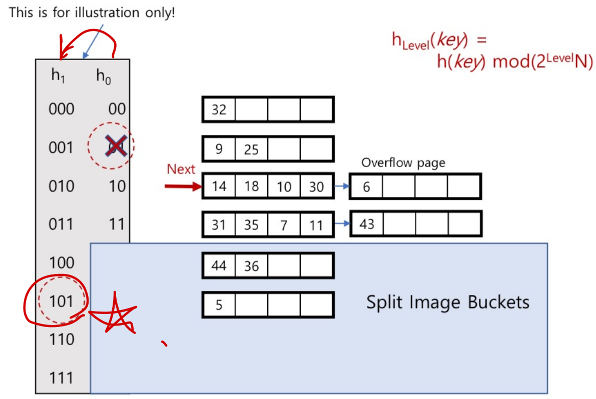
① if $h_{Level}(r) \geq$ Next → r은 이 bucket에 있음

iii) 〃 O

① $h_{Level}(r)$ 혹은 $h_{Level}(r) + N_{Level}$

② $h_{Level+1}(r)$로 찾는 것을 적용해야 함.

Search 5 ($00101_{(2)}$) (Level 0, N=4)



* re-organization → cost ↑

insertion, deletion, 상대적인 freq↑