Hashing

CMPT 606

Read Chapter 18 and Chapter 17 (pages 611 to 621)

Outline

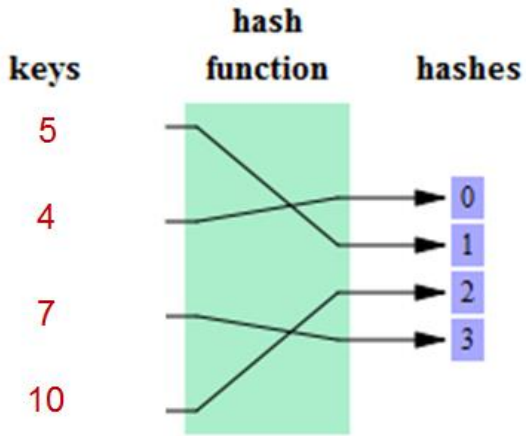
1. Extensible Hashing
2. Linear Hashing

Hash Indexes

Back

Hash Tables in Main Memory

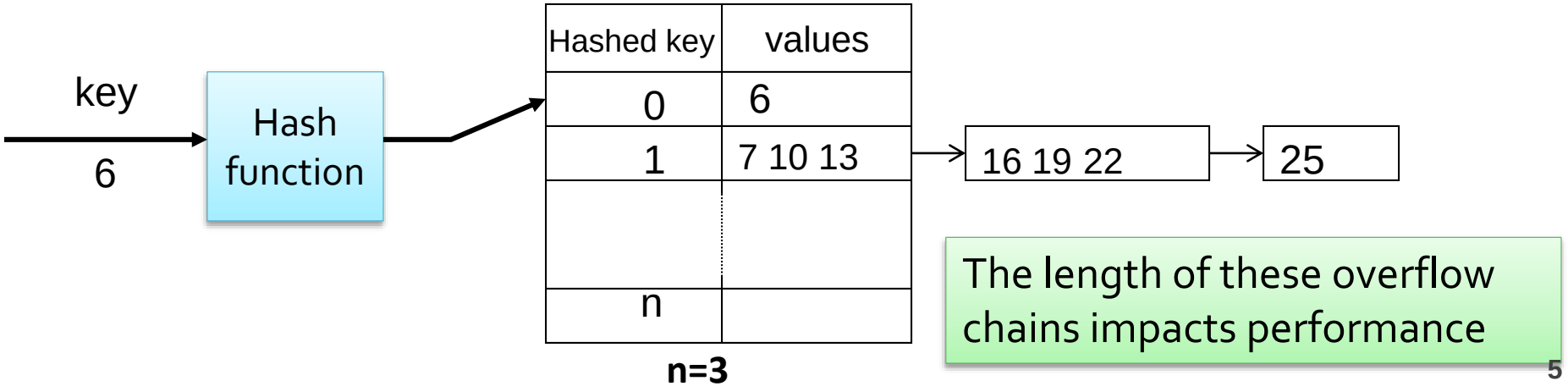
- In a hash table a hash function maps search key values to array elements
  - The array can either contain the data objects, or
  - Linked lists containing data objects (often called ***buckets***)
- Hash functions generate a value between 0 and *B*-1
  - Where *B* is the number of buckets
  - A record with **search key *K*** is **stored in bucket *h*(*K*)**



- **A typical hash functions is a bit representation of (the Search Key Value modulus the Number of Buckets)**
- **Hash indexes do not support range lookup**

Hash Index

- A hash index stores **key-value pairs** based on a *hash function => an array of pointers to buckets*
- The hash function often is ***K* mod *B***
  - Where ***K*** is the key value and ***B*** is the number of buckets
- Collision is resolved by placing new values in an **overflow bucket**



Static Hashing vs. Dynamic Hashing

- In static hashing the number of buckets is fixed
  - If the file grows most buckets will have overflow chains, reducing efficiency

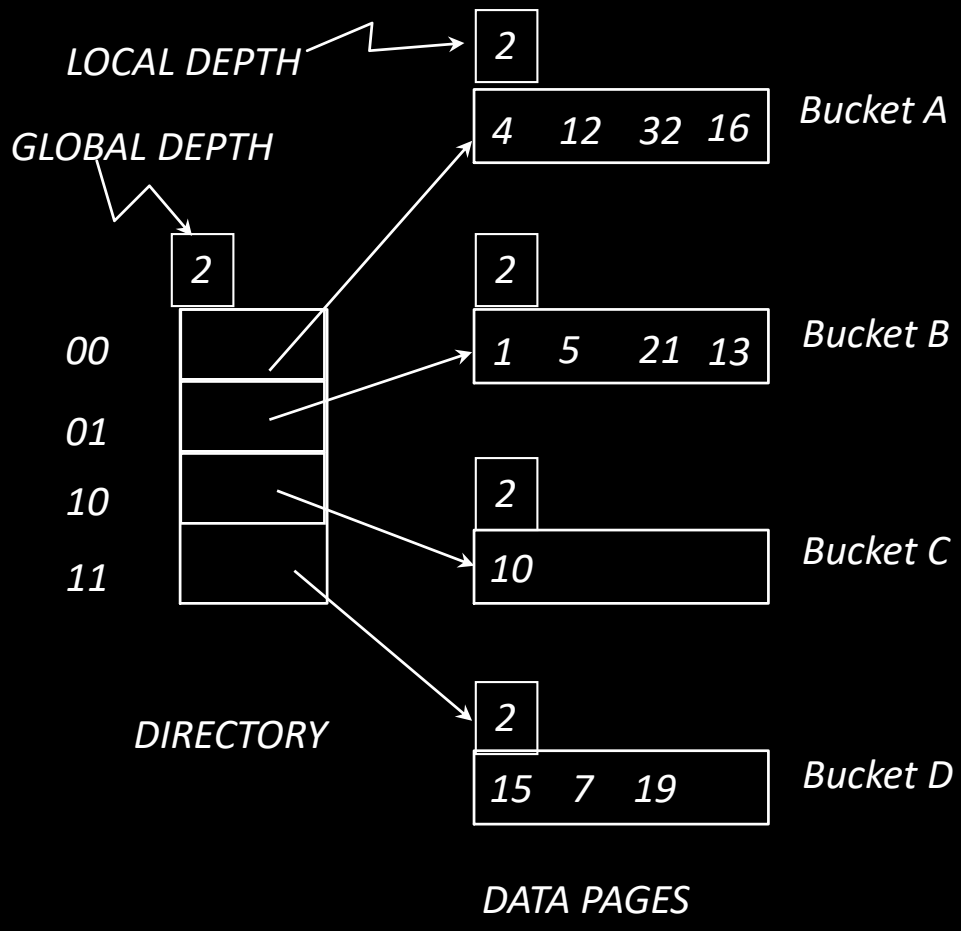
  => There should be enough buckets so that there are few overflow blocks
- Two versions of dynamic hashing that allow **dynamic File Expansion**
  - Extensible hashing
  - Linear hashing

Extensible Hashing Structure

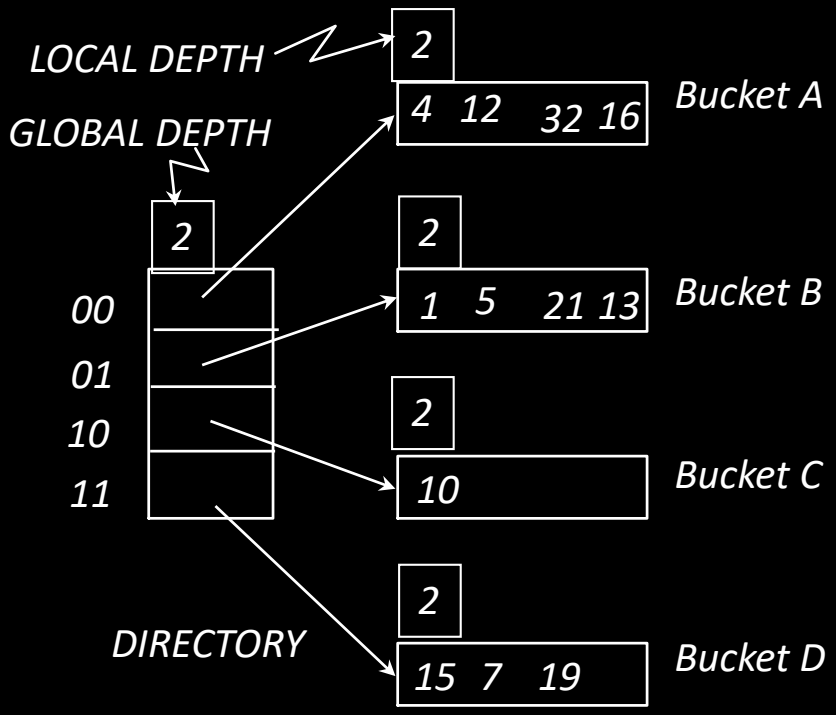
- In extendible hashing there is a **directory** to buckets
  - The directory is an array of pointers to buckets
  - As the array only contains pointers it is relatively small, so it can usually fit in memory
  - The directory size is doubled when overflow occurs
  - New buckets are only created as necessary
- The hash function computes a sequence of bits
  - The directory (and the associated buckets) uses the last ***i*** bits of the hash value
  - The directory will have 2^i entries
    - When the directory grows, $i+1$ bits of the hash value are used

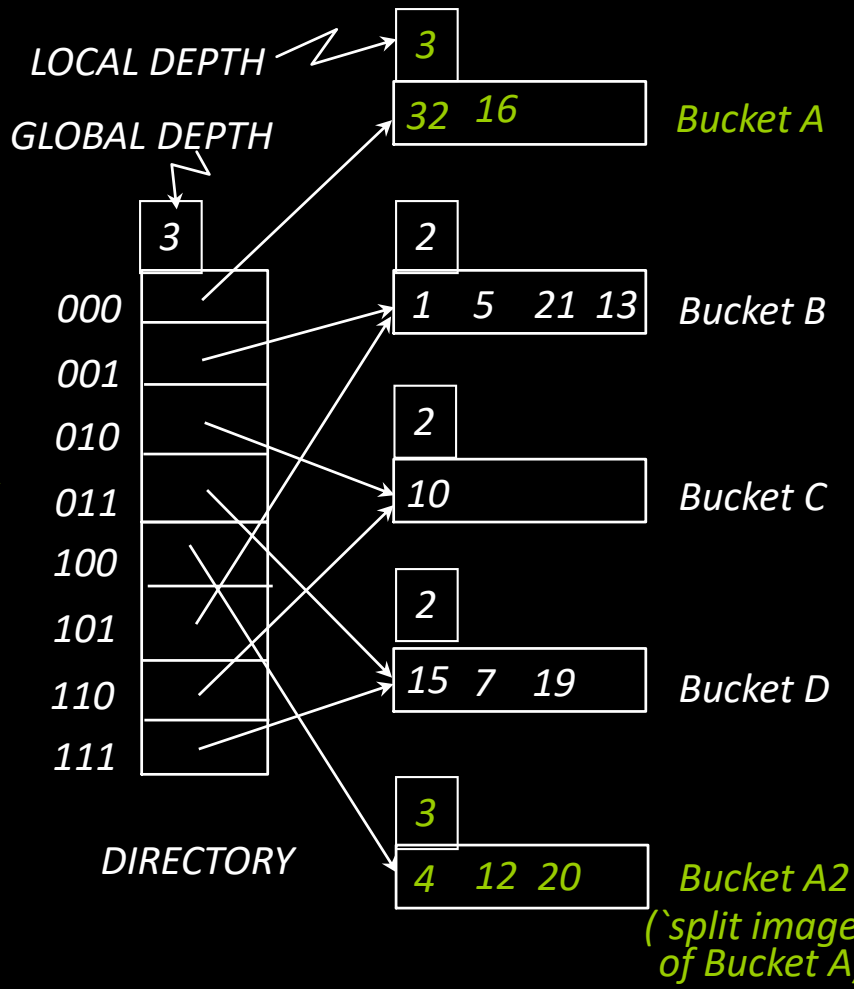
Extensible Hashing Example

- Directory is array of size 4.
- Assume a bucket capacity of 4
- What info this hash data structure maintains?
  - Global Depth: # of bits needed to tell which bucket an entry belongs to
  - Local Depth: # of bits used to determine if an entry belongs to a specific bucket



Insert 20 (10100): Double Directory





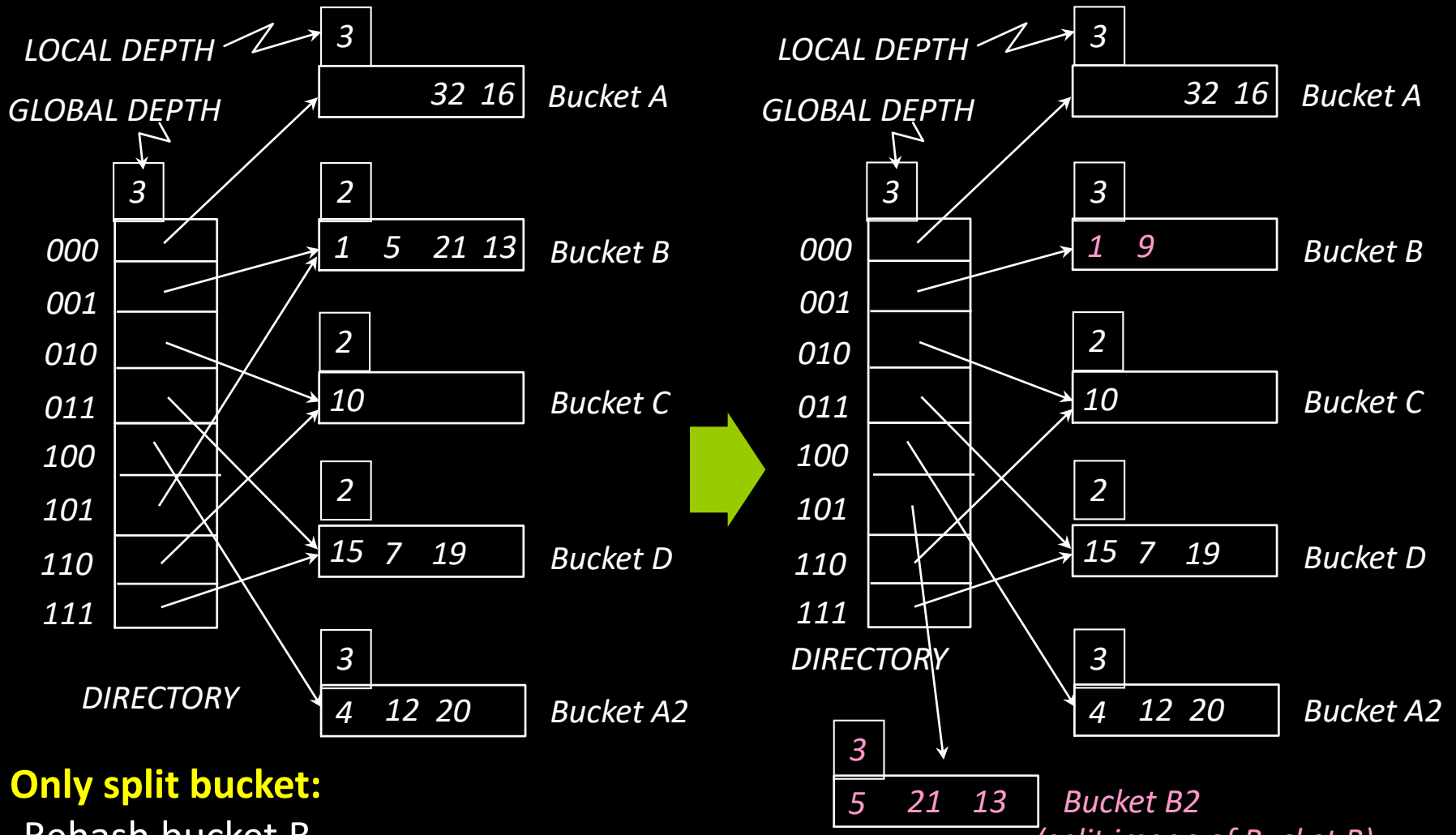
Split the bucket when local depth = global depth

- if yes, double the directory + rehash the entries and distribute into two buckets

- if no, rehash the entries and distribute them into two buckets.

- increment the local depth.

Insert 9 (1001): only Split Bucket



Only split bucket:

-Rehash bucket B

-Increment local depth

Extensible Hashing - Points to Note

- What is the global depth of directory?
  - Max # of bits needed to tell which bucket an entry belongs to.
- What is the local depth of a bucket?
  - # of bits used to determine if an entry belongs to a specific bucket
- When does bucket split cause directory doubling?
  - Bucket is full & local depth = global depth.
- How to efficiently double the directory?
  - Directory is doubled by copying it over and `fixing' pointers to the splited bucket

Linear Hashing (LH)

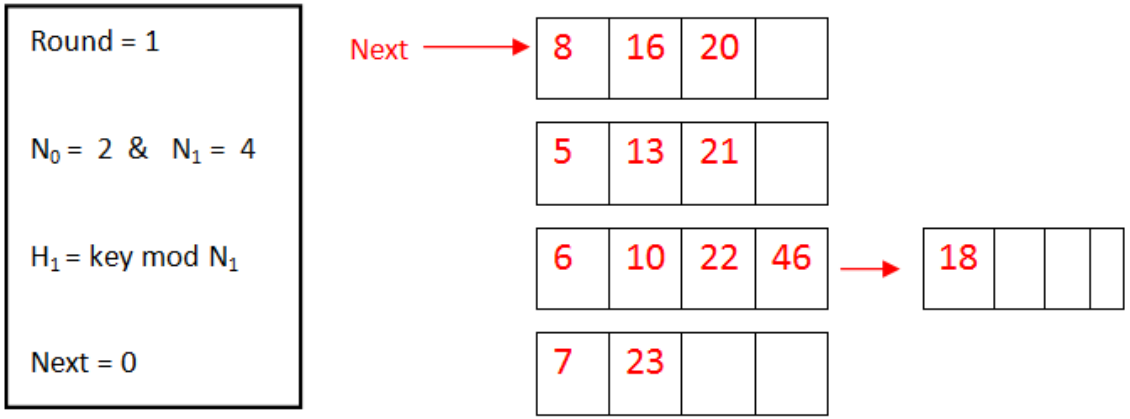
- Another dynamic hashing scheme, as an alternative to Extendible Hashing.
- What are problems in static/extendible hashing?
  - Static hashing can cause long overflow chains
  - Extendible hashing: data skew causing large directory
    - Data skew = Multiple entries with same hash value
    - If bucket already full of same hash value, will keep doubling forever!
- Is it possible to come up with a more balanced solution?
  - Shorter overflow chains than static hashing
  - No need for directory

=> Linear Hashing is the answer

Linear Hashing Algorithm

- Initial Stage.
  - The initial level distributes entries into N_0 buckets.
  - The hash function to perform is noted h_0
- *Idea*: Use a family of hash functions h_0, h_1, h_2, ...
  - $h_i(key) = key \bmod (2^i N_0)$; N_0 = initial # buckets
  - h_{i+1} doubles the range of h_i (similar to directory doubling)

Example

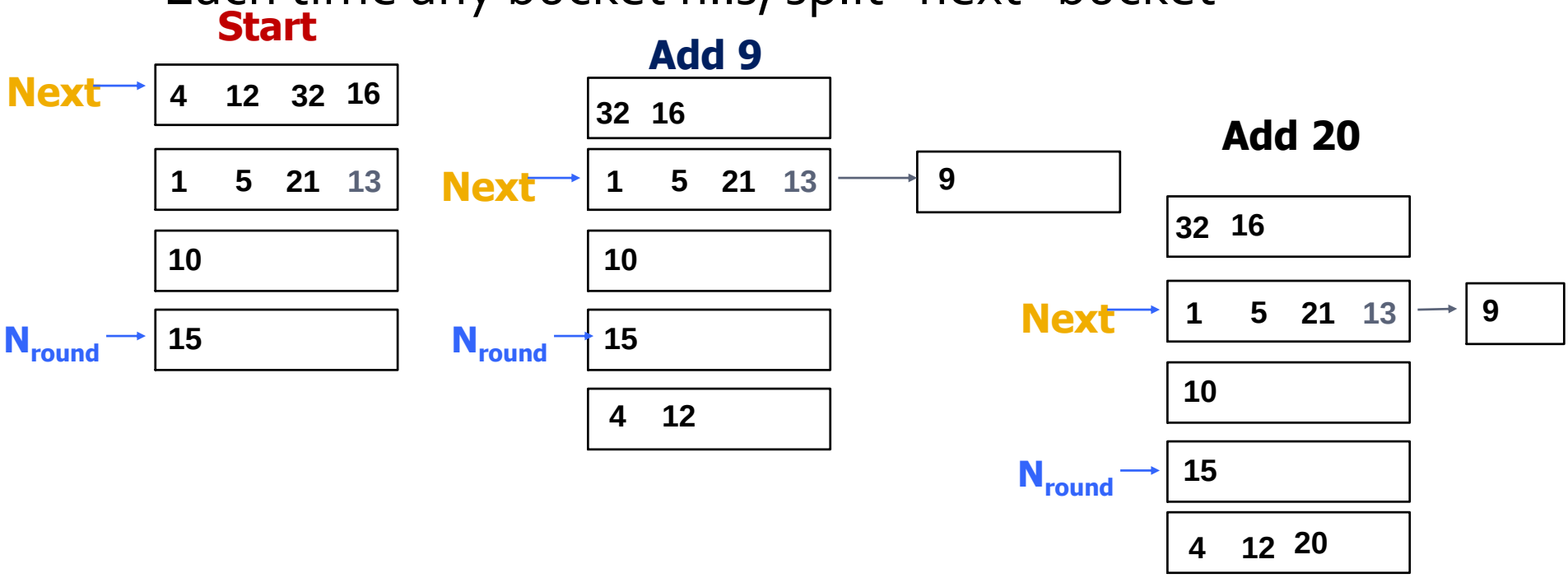


Linear Hashing Verbal Algorithm

- Splitting buckets
  - If a bucket overflows its primary page is chained to an overflow page + some bucket is split
    - The first bucket to be split is the first bucket (*not* necessarily the bucket that overflows)
    - The **next** bucket to be split is the second bucket in the file … and so on until the N^{th} has been split
    - When buckets are split their entries (including those in overflow pages) are distributed using $\boldsymbol{h_1}$
  - To access split buckets the next level hash function (h_1) is applied
  - H_1 maps entries to $2N_0$ buckets
- Alternatively, splitting can be triggered when a fill factor (average occupancy in buckets e.g. 80%) is exceeded

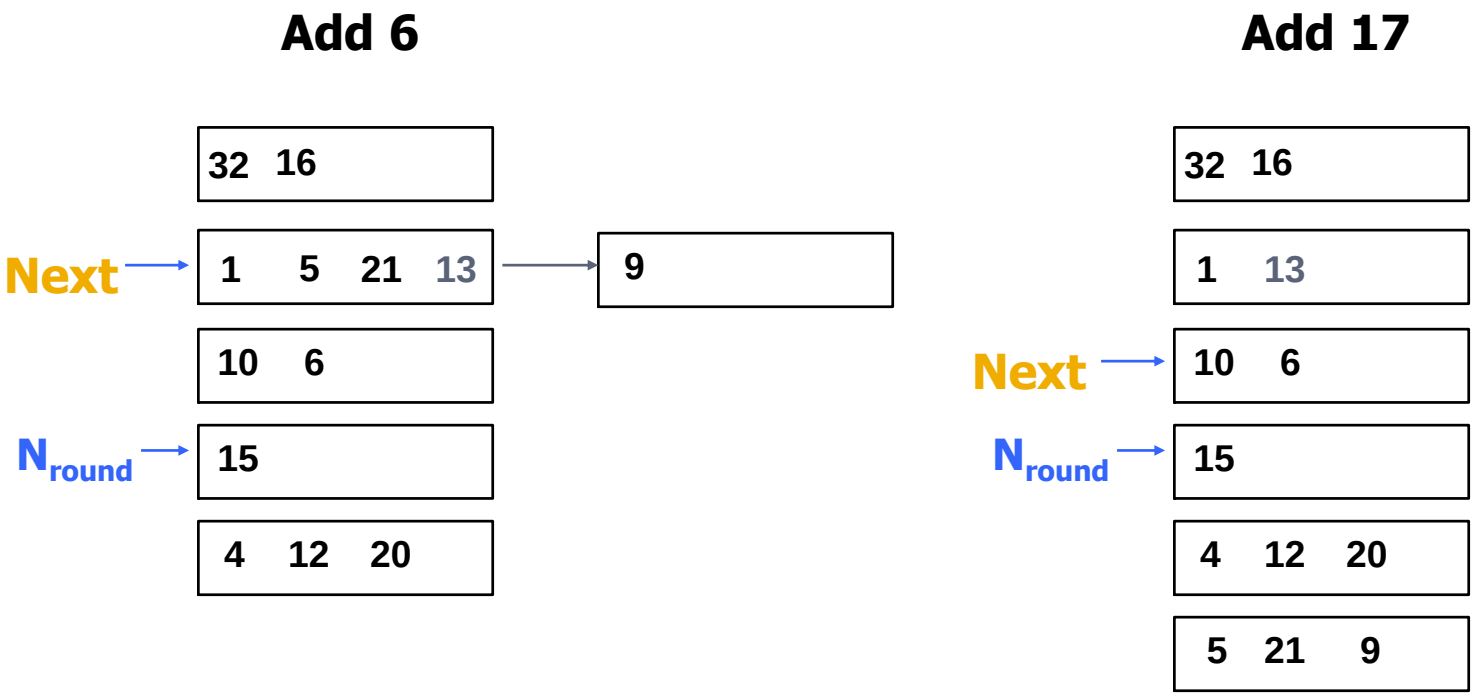
Linear Hashing Example

- Let's start with N = 4 Buckets
  - Start at "round" 0, "next" 0
  - Each time any bucket fills, split "next" bucket



Linear Hashing Example (cont)

- Overflow chains do exist, but eventually get split
- Instead of doubling, new buckets added one-at-a-time

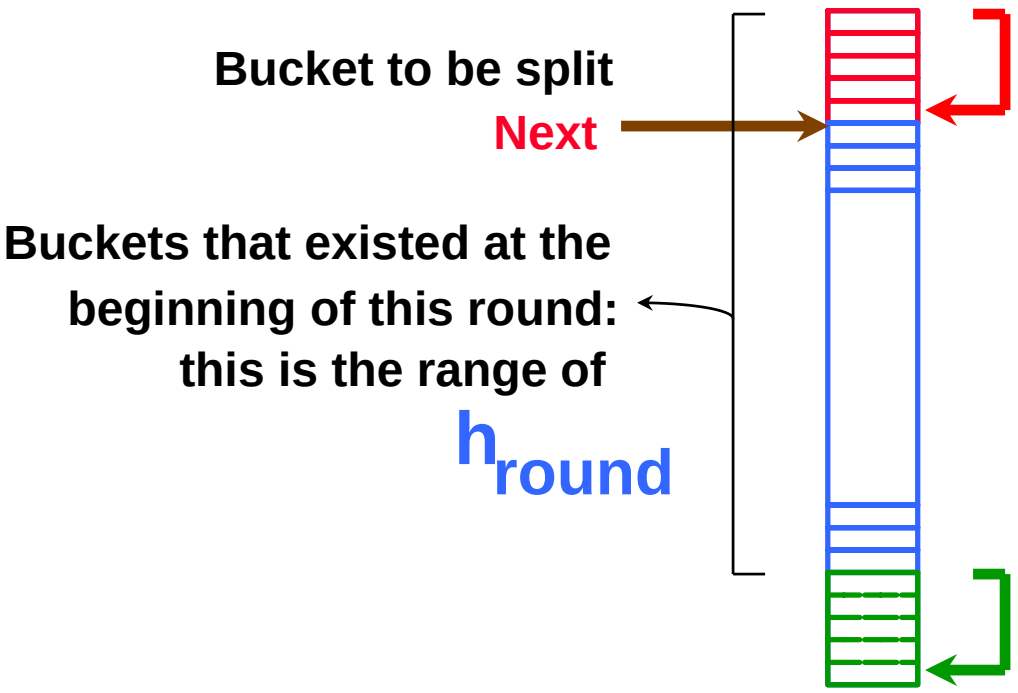


Linear Hashing (Contd.)

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

Overview of LH File

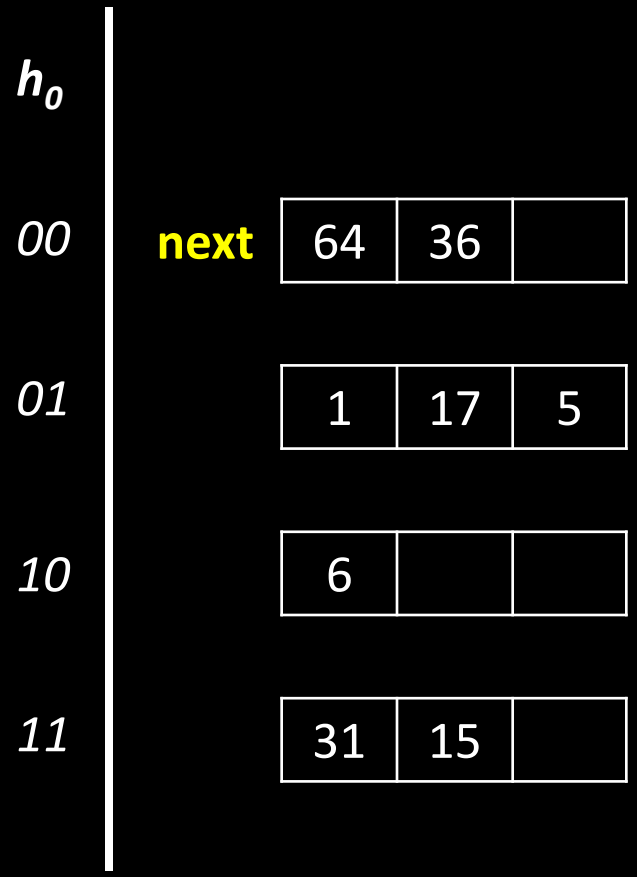
- **In the middle of a round.**



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

buckets created (through splitting of other buckets) in this round

Linear Hashing Example

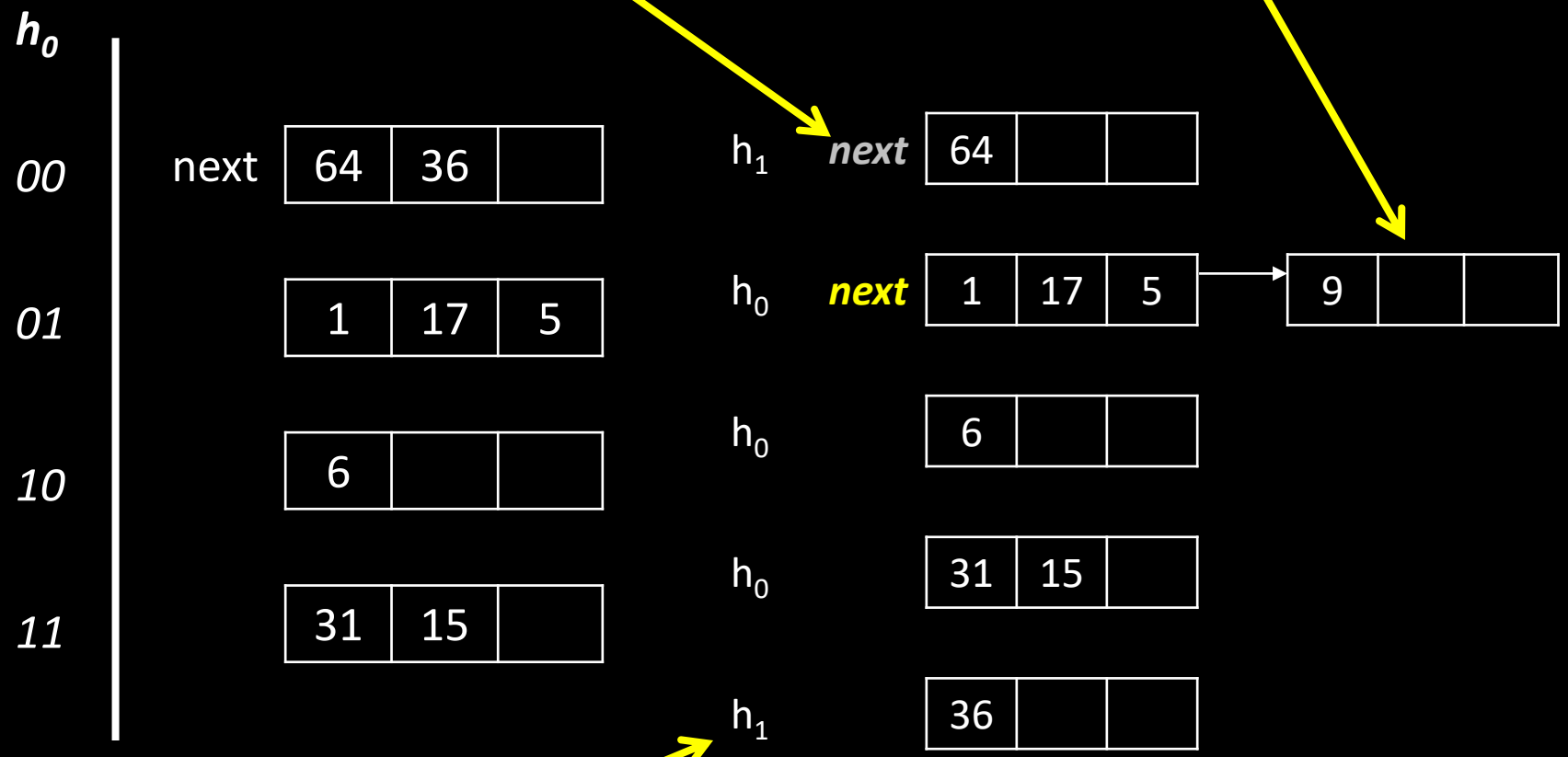


- Initially, we have $N_0 = 4$ buckets
- Assume three entries fit on a bucket
- h_0 range = 4 buckets
- Note that ***next*** indicates which bucket is to split next (Round Robin)
- Now consider what happens when 9 (1001) is inserted (which will not fit in the second bucket)

Insert 9 (1001)

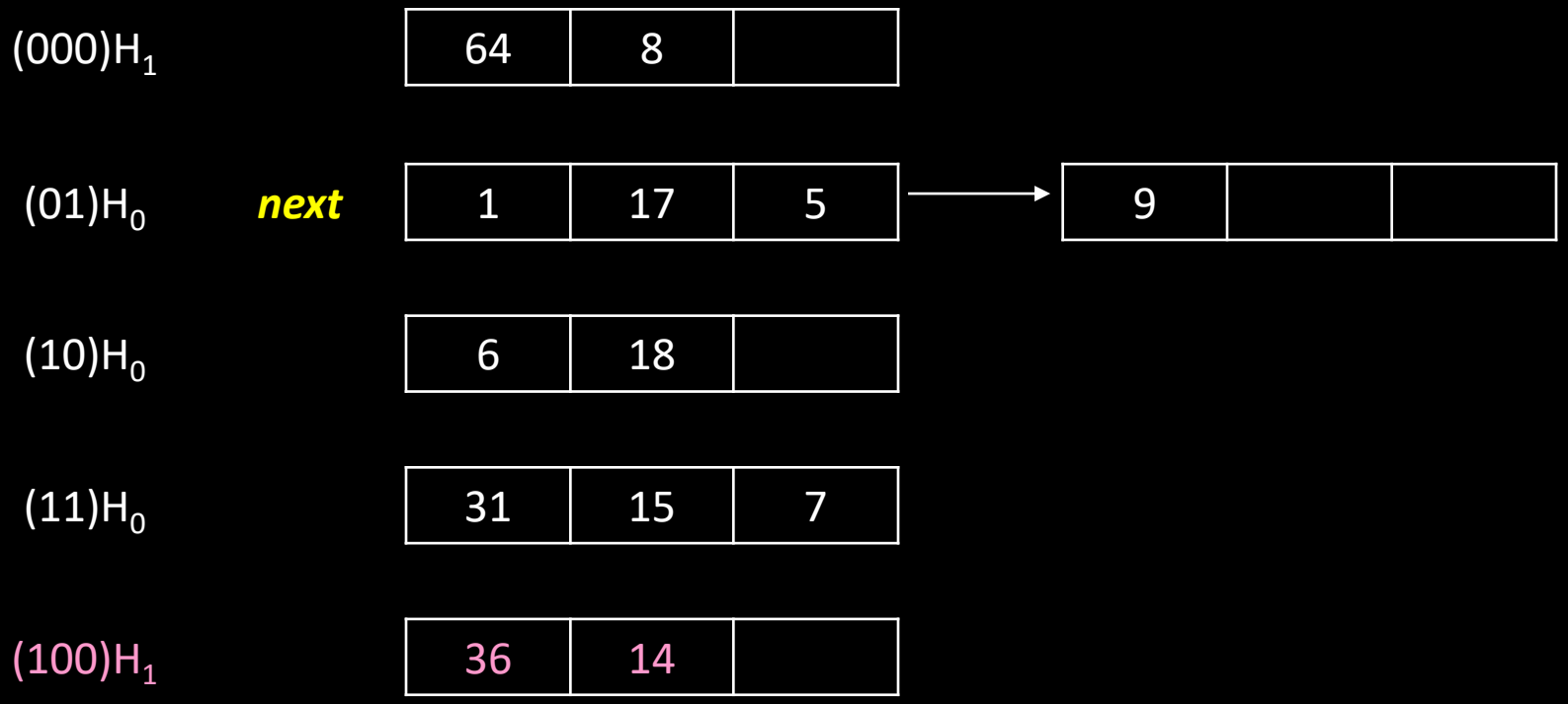
- The bucket indicated by **next** (the first one) is split
- **Next** is incremented.

An overflow bucket is chained to the primary bucket to contain the inserted value.
This causes a split to occur.



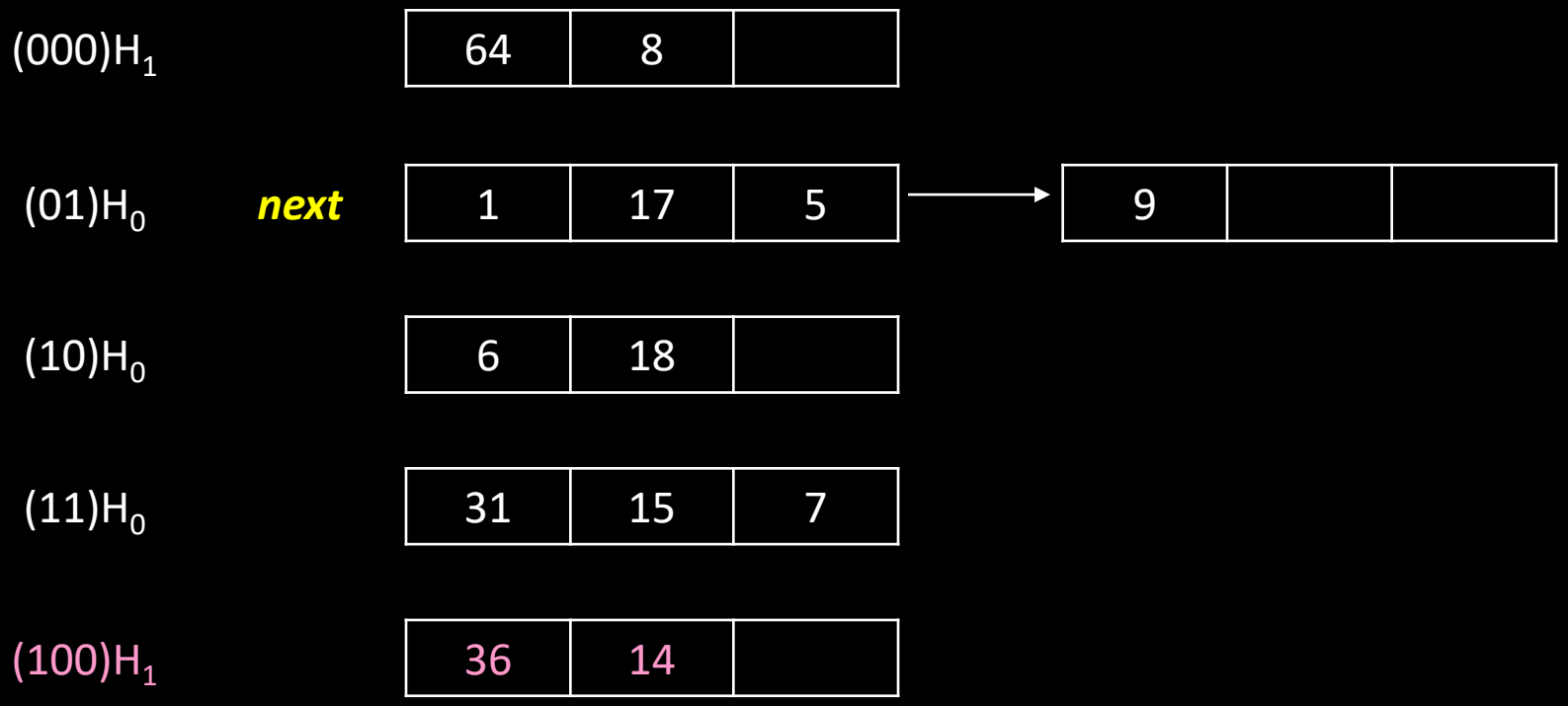
For subsequent inserts, apply hash function h_0 first, if the key is hashed to bucket before **next** pointer. Then h_1 must be used to insert the new entry.

Insert 8 (1000), 7(111), 18(10010), 14(1100)

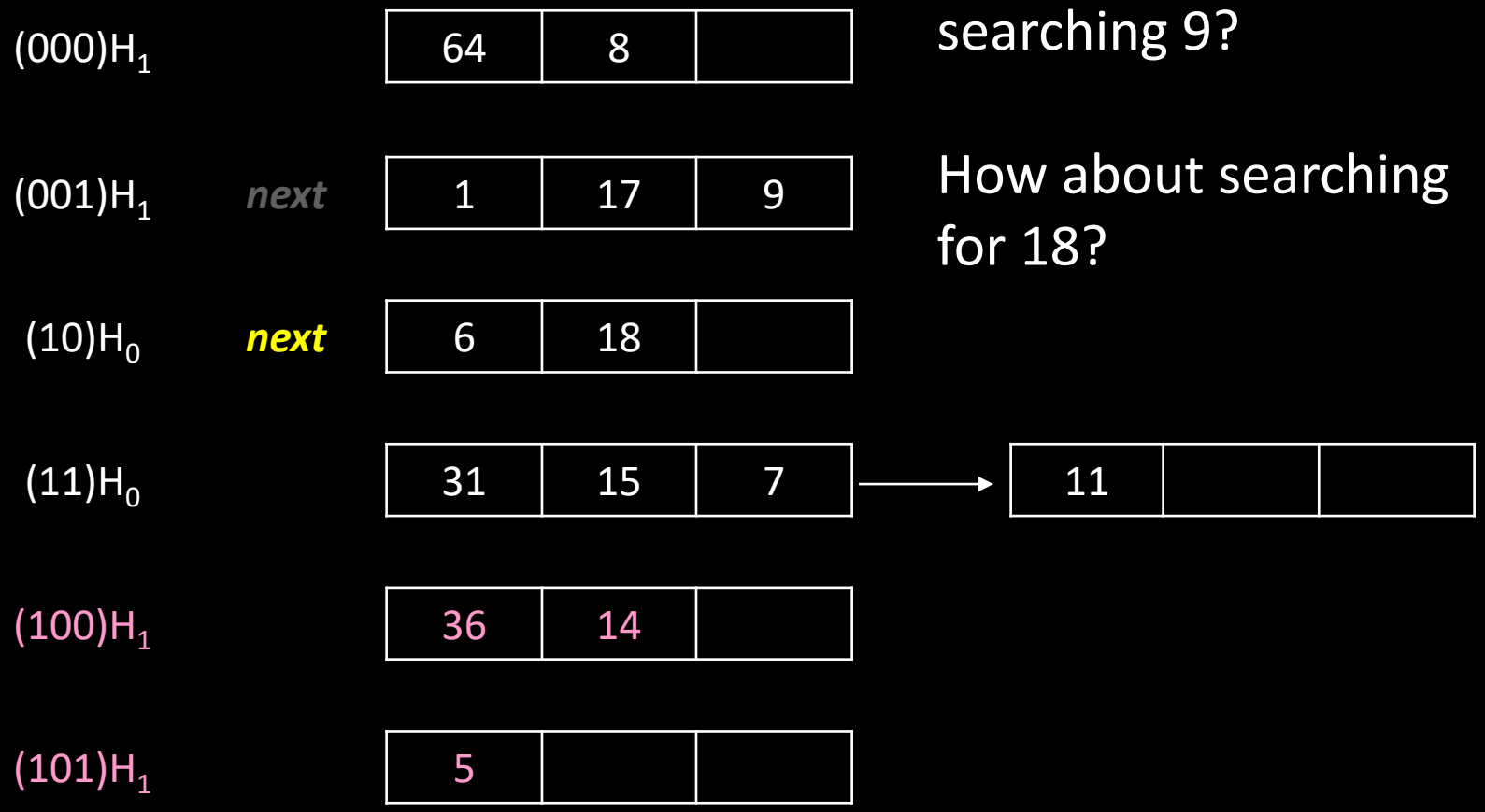


- Note that the split bucket is not necessary the overflow bucket. The split bucket is chosen based on round robin.
- Need to use both h_0 and maybe h_1. **Apply h_0 first, if it is hashed to bucket before next pointer. Then use h_1.**

Before Insert 11 (1011)



After Insert 11 (1011)

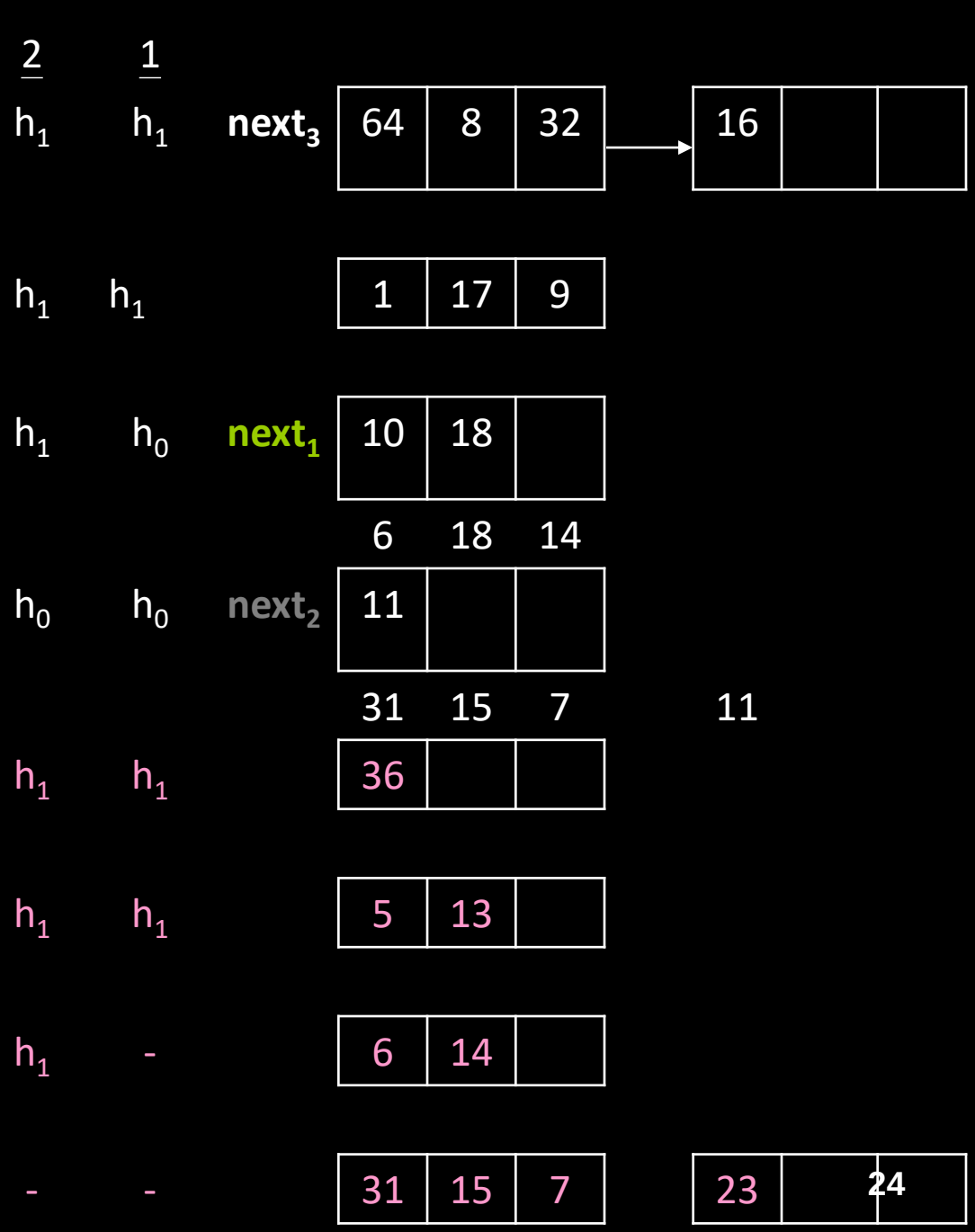


Which hash function (h0 or h1) for searching 9?

How about searching for 18?

Linear Hashing

- Insert 32, $\underline{16}_2$
- Insert 10, 13, $\underline{23}_3$
- After the 2nd split the base level is 1 ($N_1 = 8$), use h_1.
- Subsequent splits will use h_2 for inserts between the first bucket and *next*-1.



Extensible and Linear Hashing

- Linear hashing does not require a dictionary
- Linear hashing may result in less space efficiency because buckets are split before they overflow
- Multiple collisions in one bucket in extensible hashing will result in a large directory
  - Such a directory may not fit on one disk block
- Collisions in linear hashing lead to long overflow chains for the bucket with the collisions
  - Requiring multiple disk reads for that bucket
  - But no increase in the cost of accessing other buckets

Summary

- Use a hash index for point queries only. Use a B-tree if multipoint queries or range queries are used
- Use clustering:
  - if your queries need all or most of the fields of each records returned
  - if multipoint, range queries or order-by queries are often requested