Database Systems

Index: Hashing

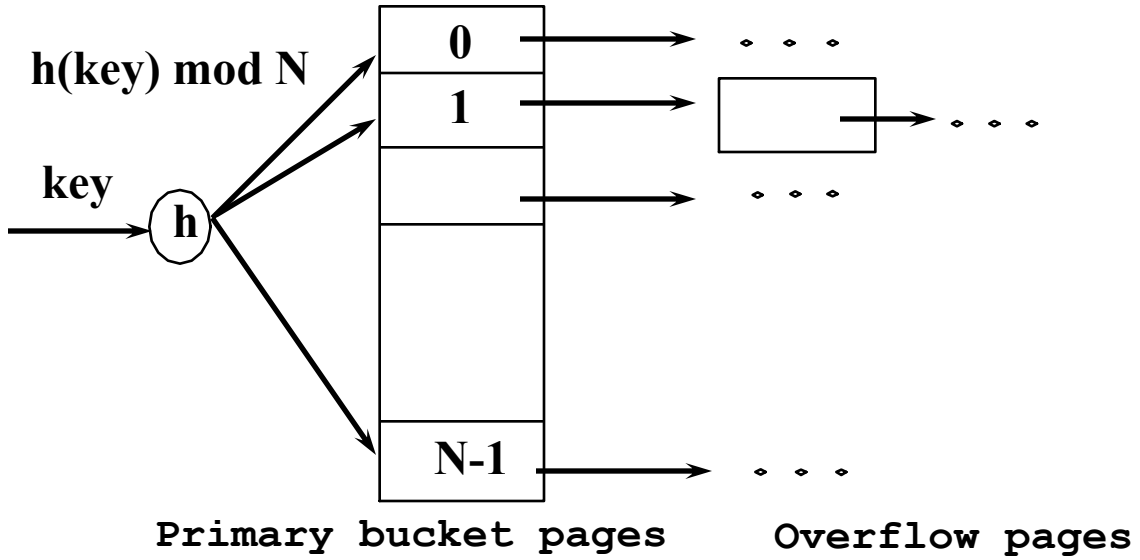
Based on slides by Feifei Li, University of Utah

Hashing

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

Static Hashing

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



Static Hashing (Contd.)

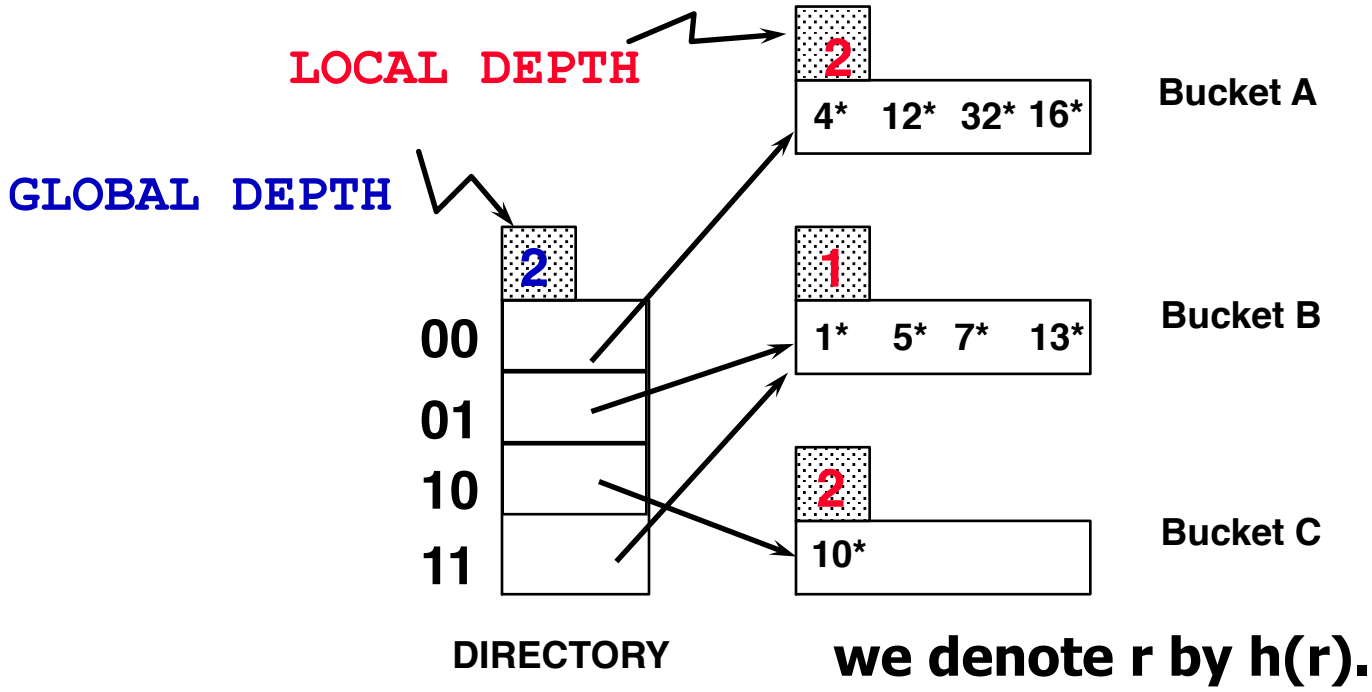
- Buckets contain *data entries*.
- Hash function works on *search key* field of record *r.* Use its value MOD N to distribute values over range 0 ... N-1.
  - **h**(*key*) = (a * *key* + b) mod P (for some prime P and a, b randomly chosen from the field of P) usually works well.
  - a and b are constants; lots known about how to tune **h**.
  - more on this subject later
- Long overflow chains can develop and degrade performance.
  - ***Extendible*** and ***Linear Hashing***: Dynamic techniques to fix this problem.

Extendible Hashing

- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
  - Reading and writing all pages is expensive!
- *Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory,* splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page*!
  - Trick lies in how hash function is adjusted!

Example

- Directory is array of size 4.
- Bucket for record *r* has entry with index = `*global depth*' least significant bits of h(*r*);
  - If h(*r*) = 5 = binary 101, it is in bucket pointed to by 01.
  - If h(*r*) = 7 = binary 111, it is in bucket pointed to by 11.

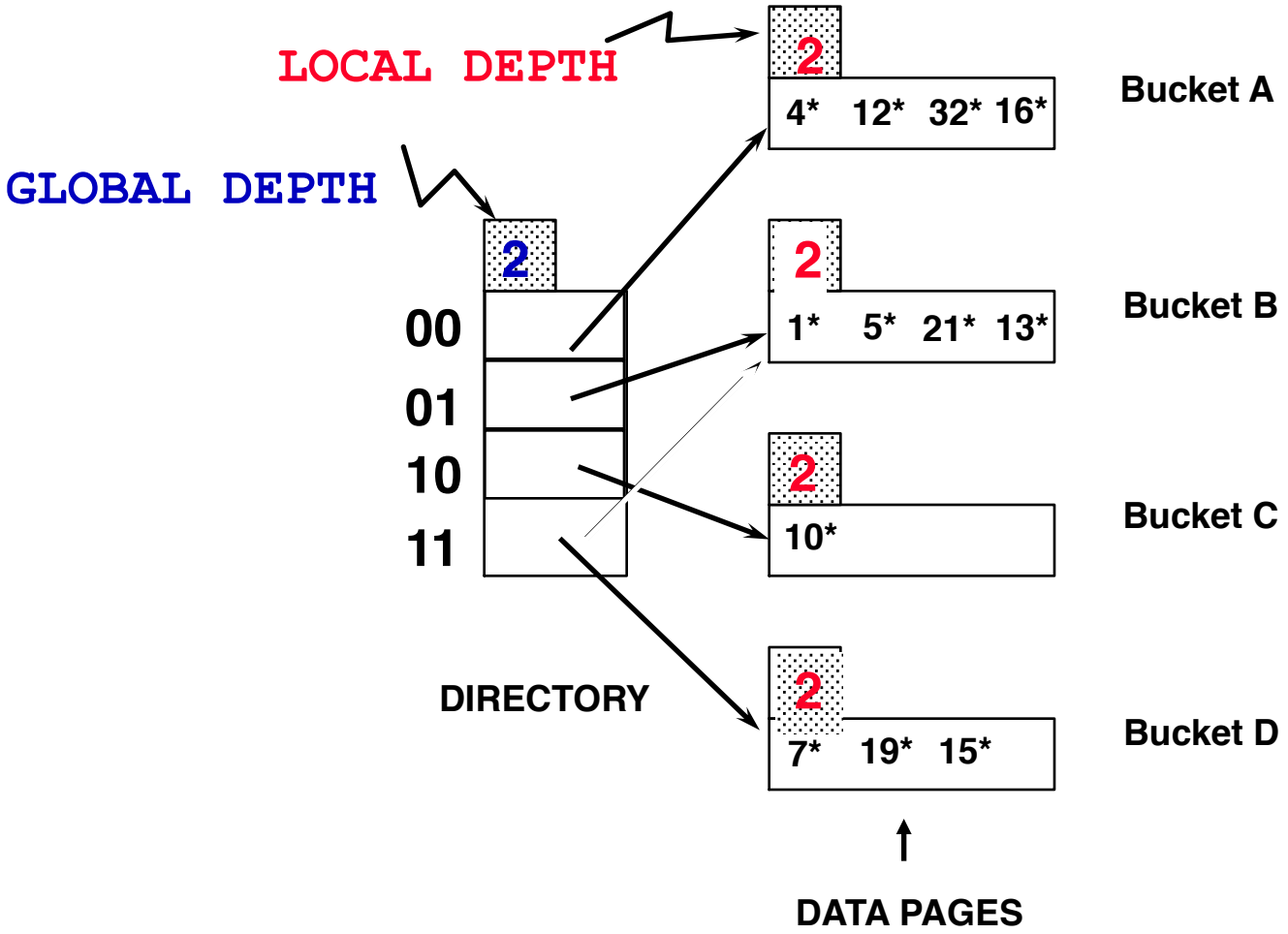


Handling Inserts

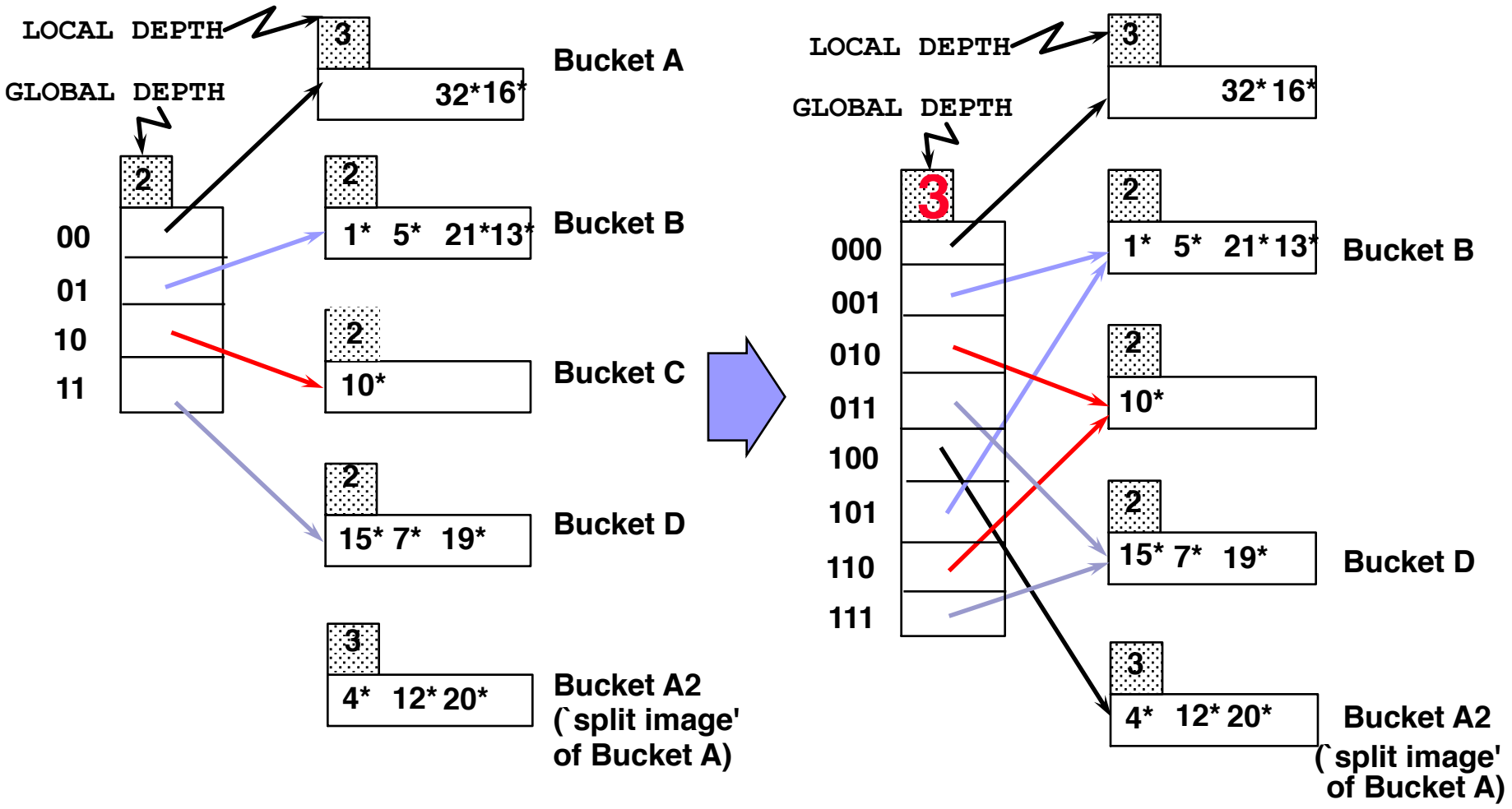
- Find bucket where record belongs.
- If there's room, put it there.
- Else, if bucket is full, *split* it:
  - increment local depth of original page
  - allocate new page with new local depth
  - re-distribute records from original page.
  - add entry for the new page to the directory

Example: Insert 21, then 19, 15

- 21 = 10101
- 19 = 10011
- 15 = 01111



Insert h(r)=20 (Causes Doubling)



Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in either 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.

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - 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.

Linear Hashing

- A dynamic hashing scheme that handles the problem of long overflow chains without using a directory.

- Directory avoided in LH by using *temporary* overflow pages, and choosing the bucket to split in a *round-robin* fashion.

- When *any* bucket overflows split the bucket that is currently pointed to by the "*Next*" pointer and then increment that pointer to the next bucket.

Linear Hashing – The Main Idea

- Use a family of hash functions h_0, h_1, h_2, ...
- $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$
  - N = initial # buckets
  - **h** is some hash function
- h_{i+1} doubles the range of h_i (similar to directory doubling)

Linear Hashing (Contd.)

- Algorithm proceeds in `rounds'. Current round number is "*Level*".
- There are N_{Level} (= N * 2^{Level}) buckets at the beginning of a round
- Buckets 0 to *Next-1* have been split; *Next* to N_{Level} have not been split yet this round.
- Round ends when all initial buckets have been split (i.e. *Next* = N_{Level}).
- To start next round:

  Level++;

  Next = 0;

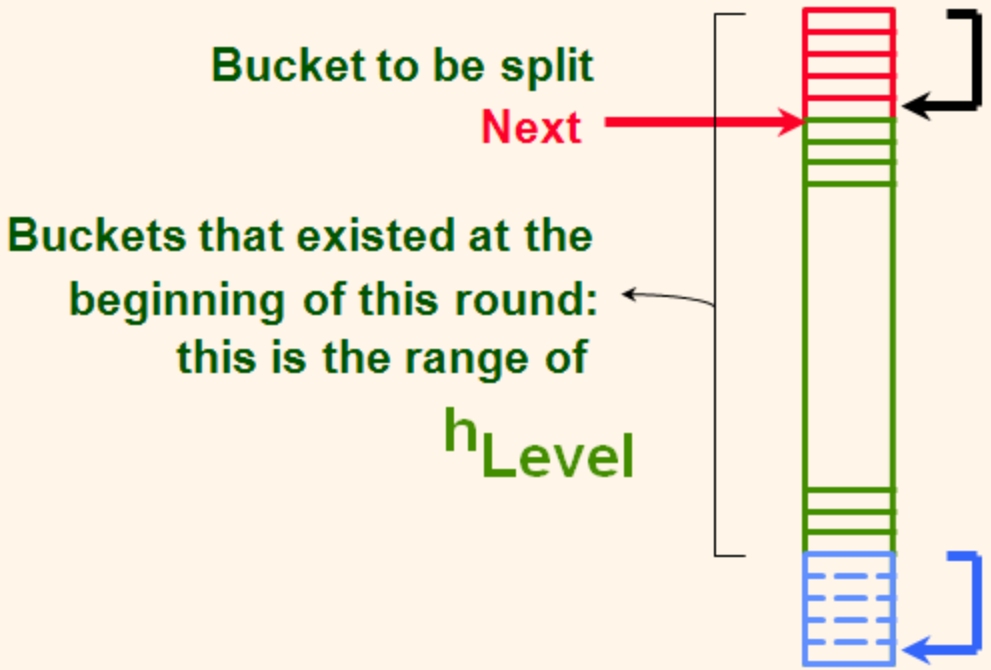
Linear Hashing - Insert

- Find appropriate bucket
- If bucket to insert into is full:
  - Add overflow page and insert data entry.
  - Split *Next* bucket and increment *Next*.
    - Note: This is likely NOT the bucket being inserted to!!!
    - to split a bucket, create a new bucket and use $\mathbf{h}_{Level+1}$ to re-distribute entries.
- Since buckets are split round-robin, long overflow chains don't develop!

Overview of Linear Hashing - Insert

Overview of LH File

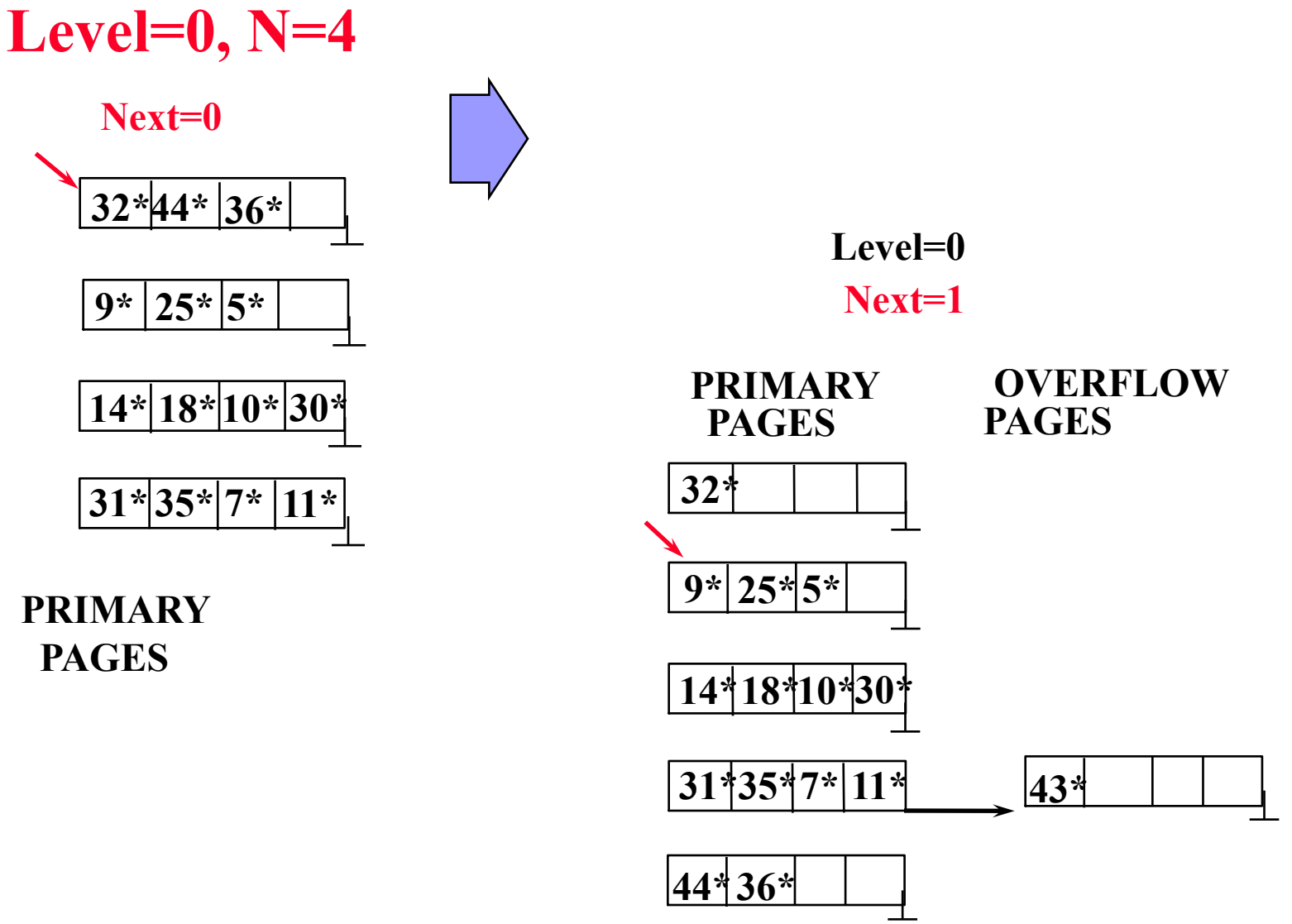
- In the middle of a round.



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

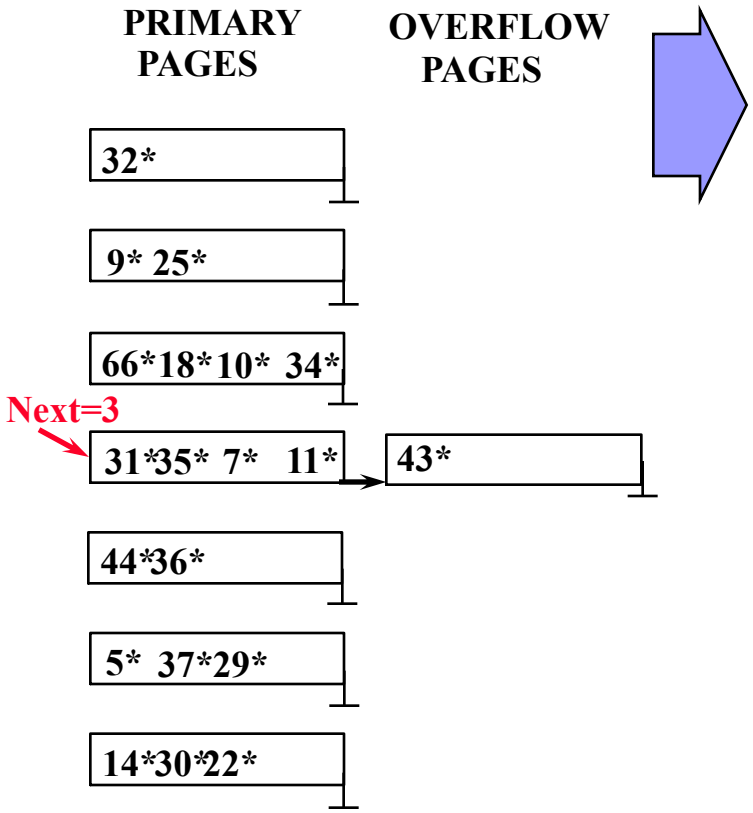
Example: Insert 43 (101011)



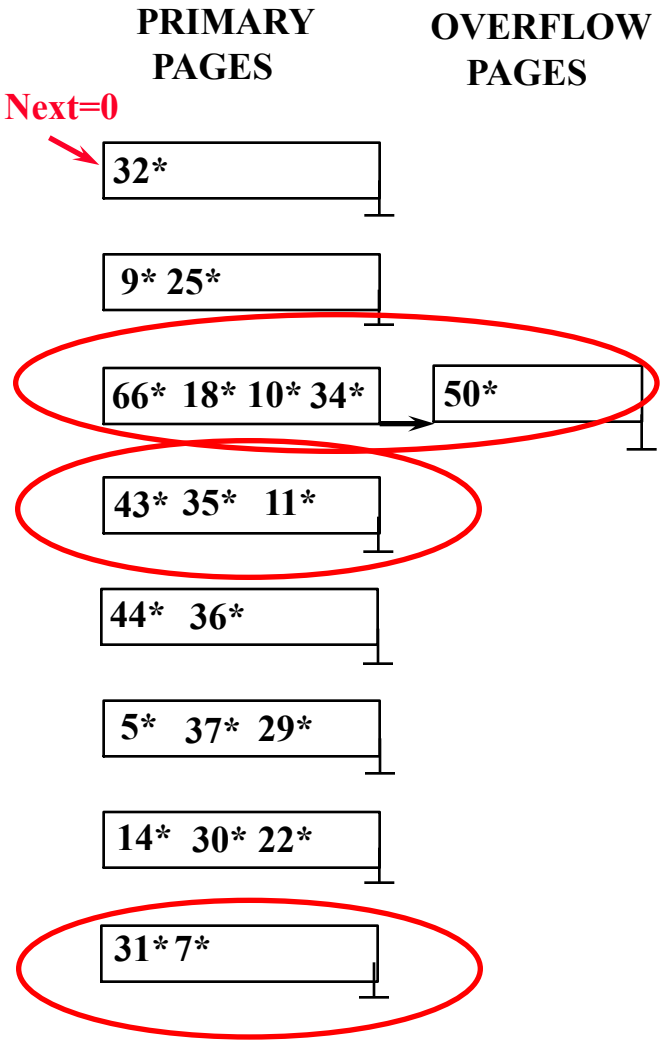
Example: End of a Round

Insert 50 (110010)

Level=0, Next = 3



Level=1, Next = 0

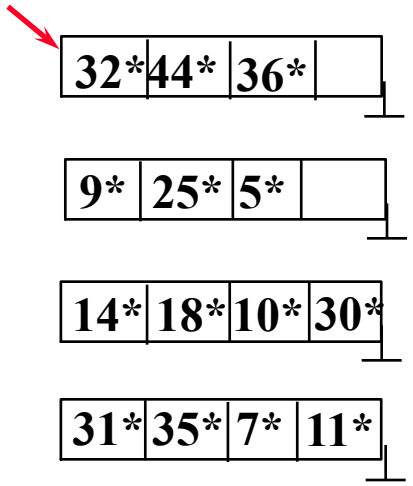


LH Search Algorithm

- To find bucket for data entry *r,* find $\mathbf{h}_{Level}(r)$*:*
  - If $\mathbf{h}_{Level}(r)$ >= Next (i.e., $\mathbf{h}_{Level}(r)$ is a bucket that hasn't been involved in a split this round) then *r* belongs in that bucket for sure.
  - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ ***or*** bucket $\mathbf{h}_{Level}(r) + N_{Level}$ must apply $\mathbf{h}_{Level+1}(r)$ to find out.

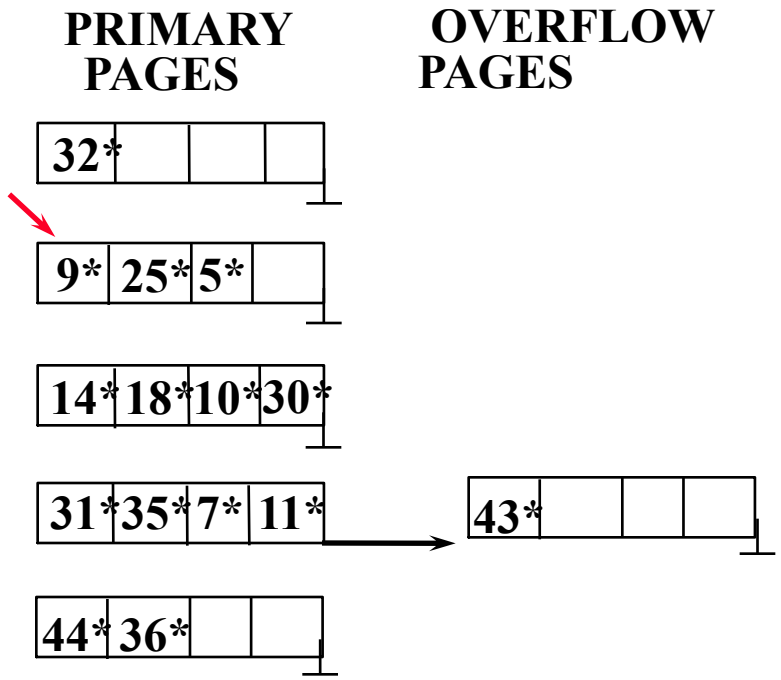
Example: Search 44 (11100), 9 (01001)

Level=0, Next=0, N=4



Example: Search 44 (11100), 9 (01001)

Level=0, Next = 1, N=4



Comments on Linear Hashing

- If insertions are skewed by the hash function, leading to long overflow buckets
  - Worst case: one split will not fix the overflow bucket
- **Delete**: The reverse of the insertion algorithm
  - Exercise: work out the details of the deletion algorithm for LH.

Designing Good Hash Functions

- Formal set up: let [N] denote the numbers {0, 1, 2, . . . , N − 1}. For any set S ⊆ U such that |S|=n, we want to support:
  - add(x): add the key x to S
  - query(x): is the key q ∈ S?
  - delete(x): remove the key x from S

  efficiently!

  We consider the static case here (fixed set S). Note that even though S is fixed, we don't know S ahead of time. Imagine it's chosen by an adversary from $\binom{N}{n}$ ***possible choices.***

  Our hash function needs to work well for any such (fixed) set S.

Static vs Dynamic

- Static: Given a set S of items, we want to store them so that we can do lookups quickly. E.g., a fixed dictionary.

- Dynamic: here we have a sequence of insert, lookup, and perhaps delete requests. We want to do these all efficiently.

Hash Function Basics

- We will perform inserts and lookups by having an array A of some size M, and a hash function $h : U \rightarrow \{0,\ldots,M-1\}$ (i.e., $h : U \rightarrow [M]$). Given an element x, the idea of hashing is we want to store it in A[h(x)].
  - If $N=|U|$ is small, this problem is trivial. But in practice, N is often big.

- Collision happens when h(x)=h(y)
  - handle collisions by having each entry in A be a linked list.

Desirable Properties

- Small probability of distinct keys colliding: if $x \neq y \in S$ then $\Pr_{h \leftarrow H}[h(x) = h(y)]$ is "small".
  - $h \leftarrow H$ means the random choice over a family H of hash functions.
- Small range: we want M to be small. At odds with first desired property; ideally M=O(N).
- Small number of bits to store a hash function h. This is at least $O(\log_2|H|)$.
- h is easy to compute
- Given this, the time to lookup an item x is O(length of list A[h(x)])

Bad News

- One way to spread elements out nicely is to spread them randomly. Unfortunately, we can't just use a random number generator to decide where the next element goes because then we would never be able to find it again. So, we want h to be something "pseudorandom" in some formal sense.

- (Bad news) For any deterministic hash function h (i.e., $|H|=1$), if $|U| \geq (N-1)M + 1$, there exists a set S of N elements that all hash to the same location.
  - simple pigeon hole argument.

Randomness to the Rescue

- Introduce a family of hash functions, H with $|H|>1$, that h will be randomly chosen from for each key (but use the same choice for the same key).

- Universal Hashing: if $x \neq y \in S$ then $\Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/M$.

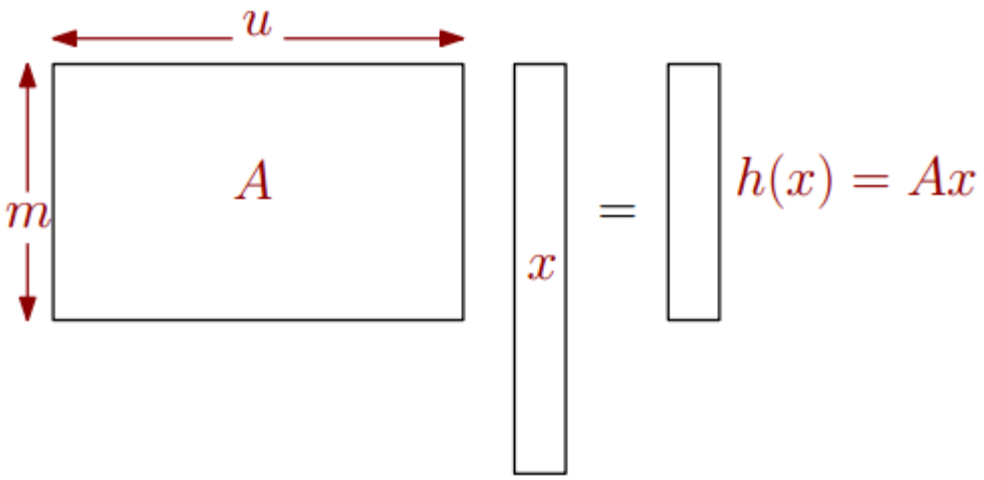
- If H is universal, then for any set $S \subseteq U$ of size N, for any $x \in U$ (e.g., that we might want to lookup, x may not come from S), if we construct h at random according to a universal hash family H, the expected number of collisions between x and other elements in S is at most N/M.

Property of Universal Hashing

- Proof:
  - Each $y \in S$ ($y \neq x$) has at most a 1/M chance of colliding with x by the definition of "universal". So
  - Let Cxy = 1 if x and y collide and 0 otherwise.
  - Let Cx denote the total number of collisions for x. So, $Cx = \sum_{y \in S, y \neq x} Cxy$.
  - We know E[Cxy] = Pr(x and y collide) ≤ 1/M.
  - So, by linearity of expectation, $E[Cx] = \sum y\ E[Cxy] < N/M$.

How to Construct Universal Hashing?

- Consider the case where $|U| = 2^u$ and $M = 2^m$
- Take an u × m matrix A and fill it with random bits. For x ∈ U, view x as a u-bit vector in $\{0, 1\}^u$, and define h(x) := Ax where the calculations are done modulo 2.
- There are 2^{um} hash functions in this family H



h x h(x)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

 Note that $h(\vec{0}) = \vec{0}$, so picking a random function from H does not map each key to a random place

Why it is a universal hash family?

- Proof:
  - We can think of it as adding some of the columns of h (doing vector addition mod 2) where the 1 bits in x indicate which ones to add
  - take an arbitrary pair of keys x, y such that $x \neq y$. They must differ someplace, so say they differ in the ith coordinate and for concreteness say $x_i = 0$ and $y_i = 1$
  - Imagine we first choose all of h but the ith column. Over the remaining choices of ith column, h(x) is fixed.
  - However, each of the 2^m different settings of the ith column gives a different value of h(y) (every time we flip a bit in that column, we flip the corresponding bit in h(y) as we are doing addition modulo 2!).
  - So there is exactly a $1/2^m$ chance that h(x) = h(y)!

Perfect Hashing (for static case)

- We say a hash function is perfect for S if all lookups involve O(1) work.
- Naïve method: an $O(N^2)$-space solution
- Let H be universal and $M = N^2$. Then just pick a random h from H and try it out!

- Claim: If H is universal and $M = N^2$, then $\Pr_{h \sim H}(\text{no collisions in S}) \geq 1/2$

Naïve method: $O(n^2)$ space

- Proof:
  - How many pairs (x,y) in S are there? Answer: $\binom{N}{2}$
  - For each pair, the chance they collide is ≤ 1/M by definition of “universal”
  - So, Pr(exists a collision) ≤ $N(N-1)/2M = N(N-1)/2N^2 < 1/2$.

A O(n) space solution (for static S)

- first hash into a table of size N using universal hashing. This will produce some collisions (unless we are extraordinarily lucky)
- then rehash each bin using Method 1, squaring the size of the bin to get zero collisions

Formally:

- a first-level hash function h and first-level table A,
- N second-level hash functions h1,... ,hN and N second-level tables A1,... ,AN
- To lookup an element x, we first compute i = h(x) and then find the element in A_i [$h_i(x)$].
- We omit the analysis of this method.

Dynamic S?

- **Cuckoo hashing:**
  - **Linear space**
  - **Constant look up time**

  - Pagh, Rasmus; Rodler, Flemming Friche (2001). "Cuckoo Hashing". *Algorithms — ESA 2001*

K-universal hashing and k-wise independent hashing

- A family H of hash functions mapping U to [M] is called k-universal if for any k distinct keys $x_1, x_2, \ldots, x_k \in U$, and any k values $\alpha_1, \alpha_2, \ldots, \alpha_k \in [M]$ (not necessarily distinct), we have

$$\Pr_{h \leftarrow H}[h(x_1) = \alpha_1 \wedge h(x_2) = \alpha_2 \wedge \cdots \wedge h(x_k) = \alpha_k] = 1/M^k .$$

- Such a hash family is also called k-wise independent. The case of k = 2 is called pairwise independent.

- Pairwise indepence: $\Pr[h(x)=a \wedge h(y)=b] = \Pr[h(x)=a] \wedge \Pr[h(y)=b]$

Simple facts about k-universal hash families

- Suppose H is a k-universal family. Then
- a) H is also (k − 1)-universal.
- b) For any $x \in U$ and $\alpha \in [M]$, $\Pr[h(x) = \alpha] = 1/M$.
- c) H is universal.

- Exercise: prove these claims?
- 2-universal is indeed stronger than universal
- The previous construction for universal hashing DOES NOT give 2-universal (since Pr[$h(\vec{0}) = \vec{0}$] = 1 and not 1/M as required above)

How to construct k-wise universal hashing?

- pick a prime p, and let U = [p] and M = p as well.
- p being a prime means that [p] has good algebraic properties: it forms the field Zp (also denoted as GF(p))
- Pick two random numbers a, b ∈ Zp. For any x ∈ U, define:

$$h(x) := (bx + a) \bmod p$$

- Claim: h(x) is 2-universal (note that there are $O(p^2)$ hash functions, i.e., $|H|=O(p^2)$)

Proof for 2-universal

- note that for x1 ≠ x2 ∈ U

$$\begin{pmatrix} h(x_1) \\ h(x_2) \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

To calculate $\Pr[h(x_1) = \alpha_1 \ \wedge \ h(x_2) = \alpha_2]$, we get

$$\Pr\left[\begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}\right] = \Pr\left[\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix}^{-1} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}\right]$$

- Since a, b are chosen randomly, the chance that each of them equals some specified values is at most 1/p x 1/p = $1/p^2$, which is $1/M^2$ as desired for 2-universality.

Apply it in practice and k-universal

- the same idea works for any field. So we could use the field GF(2^u) which has a correspondence with u-bit strings, and hence hash [2^u] → [2^u]. Now we could truncate the last u − m bits of the hash value to get a hash family mapping [2^u] to [2^m] for m ≤ u
- i.e., construct h(x) as in last slide and then mod m.

- Pick k random numbers $a_0, a_1, \ldots, a_{k-1} \in Z_p$. For any x ∈ U, define

$$h(x) := a_0 + a_1 x + a_2 x^2 + \ldots + a_{k-1} x^{k-1} \mod p \text{ (then mod m)}$$

Claim: the above construction forms a k-universal hash family.

Summary

- Many alternative hashing scheme exists, each appropriate in some situation.
- k-wise universal hashing is very useful, as it gives k-wise independence, but large k value means that it’s more expensive to describe the hash functions.