Database System Architecture and Implementation

Hash-Based Indexes
Orientation

DBMS

Web Forms  Applications  SQL Interface

SQL Commands

Executor  Parser

Operator Evaluator  Optimizer

Files and Index Structures

Buffer Manager  Disk Space Manager

Index and Data Files  Catalog

Transaction Manager  Lock Manager

Recovery Manager

Database

We are still here!

Module Overview

- Overview of hash-based indexing
- Static hashing
- Extendible hashing
- Linear hashing

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Hash-Based Indexing

In addition to tree-structured indexes (B+ trees), typical DBMS also provide support for **hash-based index structures**

- “unbeatable” when it comes to support **equality selections**
- can answer equality such queries using a **single I/O operation** (more precisely 1.2 operations), if the hash index is carefully maintained while the underlying data file for relation \( R \) grows and shrinks
- other query operations, like (equality joins) internally require a **large number of equality tests**
- presence (or absence) of support for hash indexes can make a real difference in such scenarios

```
SELECT * 
FROM   R 
WHERE  A = k
```
Hash Indexes vs. B+ Tree Indexes

- Locating a record with key $k$
  - B+ tree search *compares* $k$ to other keys $k'$ organized in a (tree-shaped) search data structure
  - hash indexes *use the bits of $k$ itself* (independent of all other stored records and their keys) to find (i.e., *compute the address of*) the record

- Range queries
  - B+ trees handle range queries efficiently by leveraging the sequence set
  - hash indexes provide no support for range queries (hash indexes are also known as *scatter storage*)
Overview of Hash-Based Indexing

• Static hashing
  – used to illustrate *basic concepts* of hashing
  – much like ISAM, static hashing does **not** handle updates well

• Dynamic hashing
  – *extendible hashing* and *linear hashing*
  – refine the hashing principle and adapt well to record insertions and deletions

• Hashing granularity
  – in contrast to in-memory applications where record-oriented hashing prevails, DBMS typically use *bucket-oriented hashing*
  – a bucket can *contain several records* and may have an *overflow chain*
  – a bucket is a *(set of) page(s)* on secondary memory

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Static Hashing

Build a static hash index on attribute \( A \)

1. Allocate a fixed area of \( N \) (successive) disk pages, the so-called **primary buckets**
2. In each bucket, install a pointer to a chain of **overflow pages**
   initially, set this pointer to **null**
3. Define a **hash function** \( h \) with **range** \([0, \ldots, N – 1]\), the **domain** of \( h \) is the type of \( A \), e.g.,
   \[
   h : \text{INTEGER} \rightarrow [0, \ldots, N – 1]
   \]
   if \( A \) has the SQL type **INTEGER**

- Evaluating the hash function \( h \) on a given data value is **cheap**: it only involves a few CPU instructions
Static Hashing

A primary bucket and its chain of overflow pages is referred to as a **bucket**.
Each bucket contains index entries $k^*$, which can be implemented using any of the variants 1, 2, and 3.
Static Hashing

• Operations \( h\text{search}(k) \), \( h\text{insert}(k) \), and \( h\text{delete}(k) \) for a record with key \( A = k \) depend on the hashing scheme

<table>
<thead>
<tr>
<th>Static hashing scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. apply hash function ( h ) to key value, i.e., compute ( h(k) )</td>
</tr>
<tr>
<td>2. access primary bucket page with number ( h(k) )</td>
</tr>
<tr>
<td>3. search, insert, or delete the record with key ( k ) on that page or, if necessary, access the overflow chain of bucket ( h(k) )</td>
</tr>
</tbody>
</table>

• If the hashing scheme works well and overflow chain access can be avoided altogether
  – \( h\text{search}(k) \) requires a **single I/O operation**
  – \( h\text{insert}(k) \) and \( h\text{delete}(k) \) require **two I/O operations**
Collisions and Overflow Chains

• At least for static hashing, **overflow chain management** is important
  – generally, we do **not** want hash function $h$ to avoid collisions, i.e.,
    
    $$ h(k) = h(k') \text{ even if } k \neq k' $$

    (otherwise as many primary buckets as different keys in the data file or even in A’s domain would be required)
  – however, it is important that $h$ **scatters** the domain of A **evenly across** $[1, \ldots, N-1]$ in order to avoid long overflow chains for few buckets
  – otherwise, the I/O behavior of the hash table becomes **non-uniform** and **unpredictable** for a query optimizer
  – unfortunately, such “good” hash functions are hard to discover
The birthday paradox

Consider the people in a group as the domain and use their birthday as hash function $h$ (i.e., $h : \text{Person} \rightarrow [0, \ldots, 364]$)

*If the group has 23 or more members, chances are 50% that two people share the same birthday (collision)*

**Check** for yourself

1. Compute the probability that $n$ people all have different birthdays

   $$P(n) = \begin{cases} 
   1 & \text{if } n = 1 \\
   P(n - 1) \times \frac{365 - (n - 1)}{365} & \text{if } n > 1 
   \end{cases}$$

2. Try to find “birthday mates” at the next larger party
Hash Functions

• If key values were purely random, a “good” hash function could simply extract a few bits and use them as a hash value
  – key value distributions found in databases are not random
  – it is impossible to generate truly random hash values from non-random key values

• But is it possible to define hash functions that scatter even better than a random function?

• Fairly good hash functions can be found rather easily by
  – division of the key value
  – multiplication of the key value

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Design of a hash function

1. **By division:** simply define
   \[ h(k) = k \mod N \]
   - this guarantees that range of \( h(k) \) to be \([0, ..., N - 1]\)
   - **prime numbers** work best for \( N \)
   - choosing \( N = 2^d \) for some \( d \) effectively considers the least \( d \) bits of \( k \) only

2. **By multiplication:** extract the fractional part of \( Z \cdot k \) (for a specific \( Z \)) and multiply by hash table size \( N \)
   \[ h(k) = \lfloor N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \rfloor \]
   - the (inverse) **golden ratio** \( Z = \frac{\sqrt{5} + 1}{2} \approx 0.6180339887 \) is a good choice (according to D. E. Knuth, “Sorting and Searching”)
   - for \( Z = \frac{\hat{Z}}{2^w} \) and \( N = 2^d \) (\( w \) is the number of bits in a CPU word), we simply have \( h(k) = msb_d(\hat{Z} \cdot k) \), where \( msb_d(x) \) denotes the \( d \) **most significant bits** of \( x \) (e.g., \( msb_3(42) = 5 \))

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Static Hashing and Dynamic Files

• Effects of dynamic files on static hashing
  – if the underlying **data file grows**, developing overflow chains spoil the otherwise predictable I/O behavior (1-2 I/O operations)
  – if the underlying **data file shrinks**, a significant fraction of primary hash buckets may be (almost) empty and waste space
  – in the worst case, a hash table can **degrade into a linear list** (one long chain of overflow buckets)

• As in the case of ISAM case, static hashing has **advantages** when it comes to concurrent access
  – allocating a hash table of size 125% of the expected data capacity, i.e., only 80% full, will typically give good results
  – data file could be rehashed periodically to restore this ideal situation (**expensive** operation and the index **cannot be used** during rehashing)
Dynamic Hashing

- Dynamic hashing schemes have been devised to overcome these limitations of static hashing by:
  - combining the use of hash functions with directories that guide the way to the data records (e.g., extendible hashing)
  - adapting the hash function (e.g., linear hashing)

Curb your enthusiasm!

Stand-alone hash indexes are very rare!
- **Microsoft SQL Server, Oracle**, and **DB2**: support for B+ tree indexes only
- **PostgreSQL**: support for both B+ tree and hash indexes (linear hashing)
- **MySQL**: depending on storage engine, both B+ tree and hash indexes are supported
- **Berkeley DB**: support for both B+ tree and hash indexes (linear hashing)

However, almost all of these systems implement the **Hybrid Hash Join** (physical) operator that uses hashing to compute the equijoin of two relations (see L. D. Shapiro: “Join Processing in Database Systems with Large Main Memories”, 1986)
Extendible Hashing

- **Extendible hashing** adapts to growing (or shrinking) data files.

- To keep track of the actual primary buckets that are part of the current hash table, an in-memory bucket directory is used.

---

Example: Extendible hash table setup (ignore the fields for now)

![Diagram of Extendible Hashing](diagram.png)

- **Note:** This figure depicts the entries as $h(k)^*$, not $k^*$. 

---

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Extendible Hashing Search

Search for a record with key \( k \)

1. Apply \( h \), i.e., compute \( h(k) \)
2. Consider the last \( 2 \) bits of \( h(k) \) and follow the corresponding directory pointer to find the bucket

- The meaning of the fields might become clear now

Global and local depth annotations

- **Global depth** \( n \) (at hash directory)
  
  *Use the last \( n \) bits of \( h(k) \) to lookup a bucket pointer in the directory* (the directory size is \( 2^n \))

- **Local depth** \( d \) (at individual buckets)
  
  *The hash values \( h(k) \) of all entries in this bucket agree on their last \( d \) bits*
**Example: Find a record with key $k$ such that $h(k) = 5$**

**Example:** To find a record with key $k$ such that $h(k) = 5 = 101_2$, follow the second directory pointer ($101_2 \land 11_2 = 01_2$) to bucket B, then use entry $5^*$ to access the record.
Extendible Hashing Search

### Searching in extendible hashing

```plaintext
function hsearch(k):
    n ← ⌈n⌉;
    (global depth of hash directory)
    b ← h(k) & (2^n - 1);
    (mask all but the low n bits)
    ↑bucket ← bucket[b];
end
```

- **Remarks**
  - `bucket[0, …, 2^n − 1]` is an **in-memory array** whose entries point to the hash buckets
  - search returns a pointer to hash bucket containing potential hit(s)
  - `&` and `|` denote **bit-wise and** and **bit-wise or** (like in C, C++, Java, etc.)
Extendible Hashing

- Insert a record with key $k$

1. Apply $h$, i.e., compute $h(k)$
2. Use the last 2 bits of $h(k)$ to lookup the bucket pointer in the directory
3. If the primary bucket still has capacity, store $h(k)^*$ in it
   Otherwise…?

- We cannot start an overflow chain hanging off the primary bucket as that would compromise uniform I/O behavior
- We cannot place $h(k)^*$ in another primary bucket since that would invalidate the hashing principle
Example: Insert a record with $h(k) = 13$

To insert a record with key $k$ such that $h(k) = 13 = 1101_2$, follow the second directory pointer (entry 01) to bucket B (which still has empty slots) and place 13* there.
**Example: Insert a record with** \( h(k) = 20 \)

Inserting a record with key \( k \) such that \( h(k) = 20 = 10100_2 \) causes an **overflow** in primary bucket \( A \) and therefore a **bucket split** for \( A \).

1. Split bucket \( A \) by creating a new bucket \( A_2 \) and use bit position + 1 to redistribute the entries:

\[
\begin{align*}
4 & = 100_2 \\
12 & = 1100_2 \\
32 & = 100000_2 \\
16 & = 10000_2 \\
20 & = 10100_2
\end{align*}
\]

Note that now 3 bits are used to discriminate between the old bucket \( A \) and the new split bucket \( A_2 \).
Extendible Hashing Insert

Example: Insert a record with \( h(k) = 20 \)

2. To address the new bucket, the directory needs to be **doubled** by simply copying its original pages (bucket pointer lookups now use + 1 = bits)

3. Let bucket pointer for \( 100_2 \) point to A2, whereas the directory pointer for \( 000_2 \) still points to A

![Diagram of Extendible Hashing](image)

Slides Credit: Michael Grossniklaus – Uni-Konstanz
### Doubling the directory

In the previous example, the directory had to be double to address the new split bucket. Is doubling the directory always necessary when a bucket is split? Or, how could you tell whether directory doubling is required or not?
Extendible Hashing Insert

### Doubling the directory

In the previous example, the directory had to be double to address the new split bucket. Is doubling the directory always necessary when a bucket is split? Or, how could you tell whether directory doubling is required or not?

- If the local depth of the split bucket is smaller than the global depth, i.e., \( d < n \), directory doubling is **not** necessary.

---

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Extendible Hashing Insert

- If the local depth of the split bucket is smaller than the global depth, i.e., \( d < n \), directory doubling is not necessary.

**Example: Insert a record with** \( h(k) = 9 \)

- Insert record with key \( k \) such that \( h(k) = 9 = 1001_2 \)
- The associated bucket \( B \) is split by creating a new bucket \( B_2 \) and redistributing the entries.

\[
\begin{array}{c}
\text{bucket } B \\
1^* 5^* 21^* 13^*
\end{array}
\begin{array}{c}
\text{bucket } B_2 \\
3 \\
1^* 9^*
\end{array}
\begin{array}{c}
\text{bucket } B \\
3 \\
5^* 21^* 13^*
\end{array}
\begin{array}{c}
\text{bucket } B_2 \\
3
\end{array}
\]

- The new local depth of \( B \) and \( B_2 \) is \( 3 \) and thus does not exceed the global depth.

**Modifying the directory’s bucket pointer for** \( 1001_2 \) **is sufficient (see next slide)**.
Extendible Hashing Insert

Example: Insert a record with \( h(k) = 9 \) (cont’d)

```
3
32* 16* 3
1* 9*
10*
5* 21* 13*

bucket A
bucket B
bucket C
bucket D
bucket A2
bucket B2
```

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Insert in extendible hashing

function hinsert(k*)
    \[ n \leftarrow n \] ;
    \text{(global depth of hash directory)}
    \[ b \leftarrow \text{hsearch}(k) ; \]
    \text{if } b \text{ has capacity then}
    \quad \text{place } k^* \text{ in bucket } b ;
    \text{else } \ldots
    \text{end}
Extendible Hashing Insert

Insert in extendible hashing (cont’d)

function hinsert(k*)
    \[ n \leftarrow n \; ; \]
    \begin{align*}
        n & \quad \text{(global depth of hash directory)} \\
        b & \leftarrow \text{hsearch}(k) ; \\
        \text{if } b \; \text{has capacity then} \ldots \\
        \text{else} \\d & \leftarrow d ; \\
        d & \leftarrow d + 1 ; \\
        \text{(local depth of bucket } b) \\
        \text{create a new empty bucket } b_2 ; \\
        d & \text{reach } k^* \text{ in bucket } b \; \text{do} \quad \text{(redistribute entries of bucket } b \text{ including } k^*) \\
        \text{if } h(k^*) \& 2^d \neq 0 \text{ then move } k^* \text{ to bucket } b_2 ; \\
        d & \leftarrow d + 1 ; \\
        \text{(new local depths for buckets } b \text{ and } b_2) \\
        \text{if } n < d + 1 \text{ then} \quad \text{(directory has to be doubled)} \\
        \text{allocate } 2^n \text{ directory entries } bucket[2^n, \ldots, 2^{n+1} - 1] ; \\
        \text{copy } bucket[0, \ldots, 2^n - 1] \text{ into } bucket[2^n, \ldots, 2^{n+1} - 1] ; \\
        n & \leftarrow n + 1 ; \\
        bucket[(h(k) \& (2^n - 1)) | 2^n] & \leftarrow @(b_2) ; \\
    \end{align*}
end

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Overflow Chains in Extendible Hashing

<table>
<thead>
<tr>
<th>Overflow chains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extendible hashing uses overflow chains hanging off a bucket only as a last resort. Under which circumstances will extendible hashing create an overflow chain?</td>
</tr>
</tbody>
</table>
Overflow Chains in Extendible Hashing

Overflow chains

Extendible hashing uses overflow chains hanging off a bucket only as a last resort. Under which circumstances will extendible hashing create an overflow chain?

יים If considering $d + 1$ bits does not lead to a satisfying record distribution in procedure $hinsert(k^*)$ (skewed data, hash collisions)

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Extendible Hashing Delete

- Routine $h\text{delete}(k^*)$ locates and removes entry $k^*$
  - deleting an entry $k^*$ from a bucket may leave this bucket empty
  - an empty buckets can be merged with its split bucket
  - however, this step is often omitted in practice

<table>
<thead>
<tr>
<th>Delete in extendible hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>When is the local depth decreased?</td>
</tr>
<tr>
<td>When is the global depth decreased?</td>
</tr>
</tbody>
</table>
## Extendible Hashing Delete

- Routine `hdelete(k*)` locates and removes entry `k*`
  - deleting an entry `k*` from a bucket may leave this bucket **empty**
  - an empty buckets can be **merged** with its split bucket
  - however, this step is often **omitted** in practice

### Delete in extendible hashing

<table>
<thead>
<tr>
<th>When is the local depth decreased?</th>
</tr>
</thead>
<tbody>
<tr>
<td>✇ when a bucket is merged with its split bucket, the local depth of the merged bucket is decreased</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>When is the global depth decreased?</th>
</tr>
</thead>
<tbody>
<tr>
<td>✇ if every directory entry points to the same bucket as its split directory entry (i.e., 0 and $2^n$ point to bucket A, 1 and $2^n + 1$ point to bucket B, etc.), the directory can be halved and the global depth decreased</td>
</tr>
</tbody>
</table>

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Linear Hashing

• Similar to extendible hashing, linear hashing can adapt its underlying data structure to record insertions and deletions
  – linear hashing does not need a hash directory in addition to the actual hash table buckets
  – linear hashing can define flexible criteria that determine when a bucket is to be split
  – linear hashing may perform bad if the key distribution is skewed
Linear Hashing

- Linear hashing uses an ordered family of hash functions
  - sequence of hash functions $h_0, h_1, h_2, \ldots$ (subscript is often called level)

Hash Function Family

Given an initial hash function $h$ and an initial hash table size $N$, one approach to define such a family of hash functions $h_0, h_1, h_2, \ldots$ would be

$$h_{\text{level}}(k) = h(k) \mod (2^{\text{level}} \cdot N)$$

where $\text{level} = 0, 1, 2, \ldots$
Example: $h_{\text{level}}$ with range $[0, \ldots, N - 1]$

$h_{\text{level}+1}$

$h_{\text{level}+2}$

$0$

$N - 1$

$N$

$2 \cdot N - 1$

$0$

$2 \cdot N - 1$

$1$

$4 \cdot N - 1$

$2 \cdot N$

$1$

$4 \cdot N - 1$

$1$

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Linear Hashing

 FN Basic linear hashing scheme

1. Initialize level ← 0 and next ← 0
2. The current hash function in use for searches (insertions/deletions) is
   \( h_{level} \)
   active hash buckets are those in the range of \( h_{level} \), i.e., [0, …, \( 2^{level} \cdot N – 1 \)]
3. Whenever the current hash table overflows
   • insertions filled a primary bucket beyond c% occupancy
   • overflow chain of a bucket grew longer than \( p \) pages
   • or (insert your criterion here)
   the bucket at hash table position next is split

Note: In general the bucket that is split is not the bucket that triggered the split!

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Bucket split

1. Allocate a new bucket and append it to the hash table at position $2^{level} \cdot N = \text{next}$
2. Redistribute the entries in bucket next by rehashing them via $h_{level + 1}$ (some entries will remain in bucket $\text{next}$, some will move to bucket $2^{level} \cdot N + \text{next}$)

3. Increment $\text{next}$ by 1

- All buckets with positions $< \text{next}$ have been rehashed
Linear Hashing

**Rehashing**

With every bucket split, next walks down the hash table. Therefore, hashing via $h_{level}$ (search, insert, and delete) needs to take **current next position** into account.

$$h_{level}(k) \begin{cases} < \text{next: bucket already split, rehash: find record in bucket} \\
\quad h_{level+1}(k) \\
\geq \text{next: bucket not yet split, i.e., bucket found} \end{cases}$$

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Split rounds: what happens if next is incremented beyond the hash table size?

A bucket split increments next by 1 to mark the next bucket to be split. How would you propose to handle the situation when next is incremented beyond the currently last hash table position, i.e.,

\[ \text{next} > 2^{\text{level}} \cdot N - 1? \]
Linear Hashing

Split rounds: what happens if $next$ is incremented beyond the hash table size?

A bucket split increments $next$ by 1 to mark the next bucket to be split. How would you propose to handle the situation when $next$ is incremented beyond the currently last hash table position, i.e.,

$$next > 2^{level} \cdot N - 1?$$

- If $next > 2^{level} \cdot N - 1$, all buckets in the current hash table are hashed via function $h_{level+1}$
- Linear hashing proceeds in a round-robin fashion: if $next > 2^{level} \cdot N - 1$
  1. increment $level \leftarrow level + 1$
  2. reset $next \leftarrow 0$ (start splitting from top of hash table again)

In general, an overflowing bucket is not split immediately, but—due to round-robin splitting—no later than in the following round.

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Linear Hashing

- Setup of linear hash table used in running example
  - bucket capacity of 4, initial hash table size $N = 4$, $level = 0$, $next = 0$
  - split criterion: allocation of a page in an overflow chain

Example: linear hash table ($h_{level(k)}$* shown)

<table>
<thead>
<tr>
<th>$level = 0$</th>
<th>$h_1$</th>
<th>$h_0$</th>
<th>hash buckets</th>
<th>overflow pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>32<em>44</em>36*</td>
<td>next</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>9<em>25</em>5*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>14<em>18</em>10<em>30</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>31<em>35</em>7<em>11</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Example: insert record with key $k$ such that $h_0(k) = 43$
Linear Hashing

Example: insert record with key $k$ such that $h_0(k) = 37$

<table>
<thead>
<tr>
<th>level = 0</th>
<th>$h_1$</th>
<th>$h_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>32*</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>9* 25* 5* 37*</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>14* 18* 10* 30*</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>31* 35* 7* 11*</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>44* 36*</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>43*</td>
</tr>
</tbody>
</table>

$37 = 100101_2$

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Linear Hashing

Example: insert record with key $k$ such that $h_0(k) = 29$

<table>
<thead>
<tr>
<th>$level = 0$</th>
<th>$h_1$</th>
<th>$h_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>32*</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>9* 25*</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>14<em>18</em>10<em>30</em></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>31<em>35</em>7<em>11</em></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>44<em>36</em></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>5<em>37</em>29*</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

$29 = 11101_2$

Slides Credit: Michael Grossniklaus – Uni-Konstanz
**Linear Hashing**

Example: insert record with key $k$ such that $h_0(k) = 22, 66, \text{ and } 34$

<table>
<thead>
<tr>
<th>$level = 0$</th>
<th>$h_1$</th>
<th>$h_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>32</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>9* 25*</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>66* 18* 10* 34*</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>31* 35* 7* 11* 43*</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>44* 36*</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>5* 37* 29*</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>14* 30* 22*</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

22 = 10110$_2$
66 = 1000010$_2$
34 = 100010$_2$

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Example: insert record with key $k$ such that $h_0(k) = 50$

\[ 50 = 110010_2 \]

Note: Rehashing a bucket means to rehash its overflow chain as well.
Linear Hashing Search

Search in linear hashing

```python
function hsearch(k)
    b ← h_{level}(k);
    if b < next then  # (b has already been split, record for key may be
        b ← h_{level+1}(k);
        in bucket b or bucket \(2^{level} \cdot N + b \rightarrow \text{rehash})
    return bucket[b];
end
```

• Remarks
  - `bucket[0, …, 2^{level} \cdot N – 1]` is an **in-memory array** containing hash table bucket (page) addresses
  - Variables `level` and `next` are **global variables** of the linear hash table,
    \( N \) is constant
### Insert in linear hashing

```plaintext
function hinsert(k*)
    b ← h_{level}(k);
    if b < next then
        b ← h_{level+1}(k);
        place k* in bucket[b];
    if overflow(bucket[b]) then
        (last insertion triggered a split of bucket)
        allocate a new bucket b';
        bucket[2^{level} \cdot N + next] ← @b';
        foreach entry k* in bucket[next] do
            place entry k* in bucket[h_{level+1}(k')];
        next ← next + 1;
        if next > 2^{level} \cdot N - 1 then
            (every bucket of the hash table been split)
            level ← level + 1;
            next ← 0;
end
```

**Note:** Predicate `overflow(·)` is a tunable parameter to control triggering of splits.
Linear Hashing Delete (Sketch)

Remarks

- linear hashing deletion is essentially the “inverse” of $hinsert(\cdot)$
- possible to replace $empty(\cdot)$ with a suitable $underflow(\cdot)$ predicate

Insert in linear hashing

```plaintext
function hdelete(k)
    if empty(bucket[2^{level} \cdot N + next]) then
        (deletion left last bucket empty)
        remove page pointed to by bucket[2^{level} \cdot N + next] from hash table;
        next ← next – 1;
        if next < 0 then
            (round-robin scheme for deletion)
            level ← level – 1;
            next ← 2^{level} \cdot N + next;
    end
```

Slides Credit: Michael Grossniklaus – Uni-Konstanz
Extendible vs. Linear Hashing

• Directory vs. no directory
  – suppose linear hashing also used a directory with elements \([0, \ldots, N - 1]\]
  – since first split is at bucket 0, element \(N\) is added to the directory
  – imagine the directory is actually doubled at this point
  – since element 1 is the same as element \(N + 1\), element 2 is the same as element \(N + 2\), and so on, copying these elements can be avoided
  – at end of the round, all \(N\) buckets are split and directory doubled in size

• Directory vs. hash function family
  – choice of hashing functions is very similar to effect of directories
  – moving from \(h_i\) to \(h_{i+1}\) corresponds to doubling the directory: both operations double effective range into which key values are hashed
  – doubling range in a single step vs. doubling range gradually

• New idea behind linear hashing is that directory can be avoided by a clever choice of the bucket to split