Introduction

- **As for any index, 3 alternatives for data entries** $k^*$:
  - A data record with key value $k$
  - $<k$, rid of data record with search key value $k$
  - $<k$, list of rids of data records with search key $k$
  - Choice orthogonal to the indexing technique
- **Hash-based indexes are best for equality selections. Cannot support range searches.**
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.
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)

Static Hashing (Contd.)

- Buckets contain data entries.
- Hash fn works on search key field of record $r$. Use its value MOD $M$ to distribute values over range $0 \ldots M-1$.
  - $h(key) = (a \times key + b)$ 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

- **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 = \textit{`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 01.

\[ \text{we denote } r \text{ by } h(r). \]
## 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**

We denote \( r \) by \( h(r) \).
Points to Note

- \(20 = \text{binary } 10100\). 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.
Directory Doubling

Why use least significant bits in directory?
- Allows for doubling via copying!

6 = 110

Least Significant vs. Most Significant

Comments on 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.
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 $h_0, h_1, h_2, \ldots$
- $h_i(key) = h(key) \mod (2^iN)$
  - $N =$ initial # buckets
  - $h =$ 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 \( N_{\text{Level}} = N \times 2^{\text{Level}} \) buckets at the beginning of a round.
- Buckets 0 to \( \text{Next-1} \) have been split; \( \text{Next} \) to \( N_{\text{Level}} \) have not been split yet this round.
- Round ends when all initial buckets have been split (i.e. \( \text{Next} = N_{\text{Level}} \)).
- To start next round:
  \[ \text{Level}++; \]
  \[ \text{Next} = 0; \]

LH Search Algorithm

- To find bucket for data entry \( r \), find \( h_{\text{Level}}(r) \):
  - If \( h_{\text{Level}}(r) \geq \text{Next} \) (i.e., \( h_{\text{Level}}(r) \) 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 \( h_{\text{Level}}(r) \) or bucket \( h_{\text{Level}}(r) + N_{\text{Level}} \); must apply \( h_{\text{Level}+1}(r) \) to find out.
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 $h_{\text{Level}+1}$ to re-distribute entries.

- Since buckets are split round-robin, long overflow chains don’t develop!

---

**Example: Insert 43 (101011)**

<table>
<thead>
<tr>
<th>h</th>
<th>h0</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>00</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>(This info is for illustration only!)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level=0</th>
<th>Next=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>32h</td>
<td></td>
</tr>
<tr>
<td>9h</td>
<td>25h</td>
</tr>
<tr>
<td>14h/18h/10h/36h</td>
<td>43h</td>
</tr>
<tr>
<td>31h/35h/7h/11h</td>
<td></td>
</tr>
</tbody>
</table>

Level=0, N=4
**Example: End of a Round**

**Insert 50 (110010)**

**Level=0, Next = 3**

```
<table>
<thead>
<tr>
<th>h1</th>
<th>h0</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>9* 25*</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>66* 18* 10* 34*</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>31* 35* 7* 11* 43*</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>44* 36*</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>5* 37* 29*</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>14* 30* 22*</td>
<td></td>
</tr>
</tbody>
</table>
```

**Level=1, Next = 0**

```
<table>
<thead>
<tr>
<th>h1</th>
<th>h0</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>9* 25*</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>66* 18* 10* 34*</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>43* 35* 11*</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>44* 36*</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>5* 37* 29*</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>14* 30* 22*</td>
<td></td>
</tr>
</tbody>
</table>
```

---

**Summary**

- **Hash-based indexes:** best for equality searches, cannot support range searches.
- **Static Hashing** can lead to long overflow chains.
- **Extendible Hashing** avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. *(Duplicates may require overflow pages.)*
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.
Summary (Contd.)

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on ‘dense’ data areas.
  - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash-based indexes, a skewed data distribution is one in which the hash values of data entries are not uniformly distributed!