Indexes

• Additional data structure used to reduce the pages accesses necessary to find a row or rows

• Search Key

• Search Key is not necessarily unique

• Location Mechanism
  – Algorithm+Data Structure
Index Terminology

• Clustered or unclustered
• Integrated or unintegrated
• Sparse or dense
B+Tree

- Good for equality and range searches
- Cost of the search depends on height of the tree
- B+Trees are very short and wide
- Keys in the leaves of an order $M$, height $n$ B+tree?
  - Maximum number of keys in leaves: $(M)^n(M-1)$
  - Minimum number of keys in leaves: $2 * (\text{ceiling}(M/2))^{(n-1)} * ((\text{ceiling}(M/2)-1)$

- For example a height 3, order 201 B+tree has between 2,040,200 and 1,624,120,200 keys in the leaves
B+Tree Index

• create index ind1 on T1(A3)
  – Select A1, A2 From T1 where A3 = 100
  – Select A1, A2 From T1 where A3 >= 50 and A3 <= 100

• create index ind2 on T1(A1, A2)
  – Select A1, A2 From T1 where A1 = 100 and A2 >= 50
  – Select A1, A2 From T1 where A1 = 100
  – Select A1, A2 From T1 where A2 = 50

• create index ind3 on T1(A1, A2, A3)
  – Select A1, A2, A3 From T1 where A1 >= 100
What does the tree look like after 45 is inserted?
B+Tree Example after 45 is inserted

What does the tree look like after 70 is inserted?
B+Tree Example after inserting 70

What does the tree look like after 130 is deleted?
What does the tree look like after 260 is deleted?
What does the tree look like after 20 is deleted?
B+Tree Example after 20 is deleted
Extendable Hashing

• Type of hashing that eliminates chains of pages caused by collisions
• Range of hash function has to be extended to accommodate additional buckets
• **Example:** family of hash functions based on $h$:
  – $h_k(v) = h(v) \mod 2^k$ (use the last $k$ bits of $h(v)$)
  – At any given time a unique hash, $h_k$, is used depending on the number of times buckets have been split
Extendable Hashing – Example

Extendable hashing uses a directory (level of indirection) to accommodate family of hash functions
Suppose next action is to insert sol, where \( h(sol) = 10001 \).
**Problem:** This causes overflow in \( B_1 \)
Example (cont’ d)

**Solution:**
1. Switch to $h_3$
2. Concatenate copy of old directory to new directory
3. Split overflowed bucket, $B$, into $B$ and $B’$, dividing entries in $B$ between the two using $h_3$
4. Pointer to $B$ in directory copy replaced by pointer to $B’$

Note: Except for $B’$, pointers in directory copy refer to original buckets.

$current\_hash$ identifies current hash function.
Example (cont’d)

Problem: When $B_i$ overflows, we need a mechanism for deciding whether the directory has to be doubled

Solution: $bucket\_level[i]$ records the number of times $B_i$ has been split. If $current\_hash > bucket\_level[i]$, do not enlarge directory

Next action: Insert judy, where $h(judy) = 00110$ $B_2$ overflows, but directory need not be extended
Example (cont’d)
Extendible Hashing Problem

What does an extendable hash table with a bucket size of 2 look like after the following values are inserted? Assume the starting table has 2 buckets and used $h_1$

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100010</td>
</tr>
<tr>
<td>323</td>
<td>101001</td>
</tr>
<tr>
<td>90</td>
<td>111011</td>
</tr>
<tr>
<td>80</td>
<td>001101</td>
</tr>
<tr>
<td>37</td>
<td>110111</td>
</tr>
<tr>
<td>205</td>
<td>010100</td>
</tr>
<tr>
<td>100</td>
<td>000110</td>
</tr>
<tr>
<td>120</td>
<td>110110</td>
</tr>
</tbody>
</table>