Selection Queries

B+-tree is perfect, but....

to answer a selection query (ssn=10) needs to traverse a full path.

In practice, 3-4 block accesses (depending on the height of the tree, buffering)

Any better approach?

Yes! Hashing

- static hashing
- dynamic hashing
Hashing

- **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 bucket pages is fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- \( h(k) \mod N \) = bucket to which data entry with key \( k \) belongs.

\[ h(key) \mod N \]

(N = # of buckets)
Static Hashing (Cont.)

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Use its value MOD N to distribute values over range 0 ... N-1.
  - \( h(key) = (a \ast key + b) \) usually works well.
  - \( a \) and \( b \) are constants; lots known about how to tune \( h \).
- Long overflow chains can develop and thus degrade performance.
  - *Extendible Hashing* and *Linear Hashing*
    - Dynamic techniques to fix this problem.
Extendible Hashing

- Suppose a 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 2 = `global depth` of 2 = 2 least significant bits of \( h(r) \);

If \( h(r) = 5 = \text{binary} 
 101 \), it is in bucket pointed to by 01.
If \( h(r) = 7 = \text{binary} 
 111 \), it is in bucket pointed to by 11.

\[
\text{Local Depth} \rightarrow \text{Global Depth} \\
\text{Bucket A: 4* 12* 32* 16*} \\
\text{Bucket B: 1* 5* 7* 13*} \\
\text{Bucket C: 2* 10*}
\]

* we denote \( r \) by \( h(r) \).
Handling Inserts

- Find bucket where record belongs.
- If there’s room, put it there.
  - else, if bucket is full, *split* it:
    1. increment local depth of original page
    2. allocate new page with new local depth
    3. re-distribute records from original page.
    4. add entry for the new page to the directory
Example: Insert 21, then 19, 15

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

**LOCAL DEPTH**

**GLOBAL DEPTH**

DIRECTORY

Bucket A

Bucket B

Bucket C

Bucket D

DATA PAGES
Insert $h(r) = 20$ (Causes Doubling)

$20_{10} = 10100_2$
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, \( \text{local depth of bucket} = \text{global depth} \).
  - Insert causes \( \text{local depth} \) to become \( > \text{global depth} \); 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 \( h_0, h_1, h_2, \ldots \)

- \( h_i(key) = h(key) \mod (2^i N) \)
  - \( N = \) 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 called “Level”.
- There are $N_{Level} = N \times 2^{Level}$ buckets at the beginning of a round.
- $Next$ points to the current bucket.
- 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;
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.

i.e., look in two places
Example: Search 44 (11100), 9 (01001)

Level=0, Next=0, N=4

(This info is for illustration only!)
Linear Hashing - Insert

- Find appropriate bucket using \( h_{\text{LEVEL}} \) (or \( h_{\text{LEVEL+1}} \))
- 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 into!!!
    - 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)

Level=0, N=4

Next=0

000 00
001 01
010 10
011 11

(This info is for illustration only!)

Next=1

000 00
001 01
010 10
011 11

PRIMARIES

OVERFLOW

000 00
001 01
010 10
011 11

(This info is for illustration only!)
Example: End of a Round

Insert 50 (110010)

Level=0, Next = 3

Level=1, Next = 0

Insert 50 (110010)
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.
Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.

- Overflow page chains are **not** likely to be long.
- Space utilization could be lower than Extendible Hashing, since splits are 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!