Hashed-Based Indexing

Linda Wu

(CMPT 354 • 2004-2)

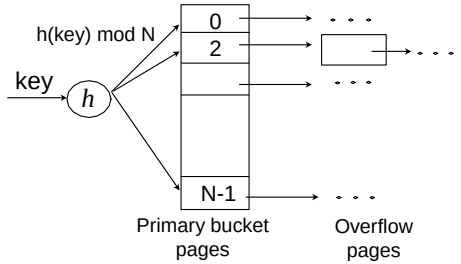
Topics

- Static hashing
- Dynamic hashing
  - Extendible Hashing
  - Linear Hashing

Static Hashing

- An index consists of buckets 0 ~ *N*-1
- A bucket consists of one primary page and, possibly, additional overflow pages
- Buckets contain data entries
- Hash function *h*
  - *h* must distribute values in the domain of search field uniformly over the buckets
  - *h(k) = (a * k + b)* usually works well, a and b are constants for tuning *h*
  - *h(k) mod N*: bucket to which the data entry with search key *k* belongs

Static Hashing (Cont.)



- # of primary pages, N, is fixed
- Primary pages are allocated sequentially, never de-allocated; overflow pages allocated if needed
- Long overflow chains can develop and degrade performance

Static Hashing (Cont.)

- Search
  - Identify the correct bucket using h
  - Search the bucket for the data entry
- Insert
  - Identify the correct bucket using h
  - If no space in the bucket, allocate a new overflow page to the overflow chain of the bucket, put the data entry on this page
- Delete
  - Identify the correct bucket using h
  - Search the bucket for the data entry, remove it
  - If it is the last in an overflow page, remove the page from the chain and de-allocate the page

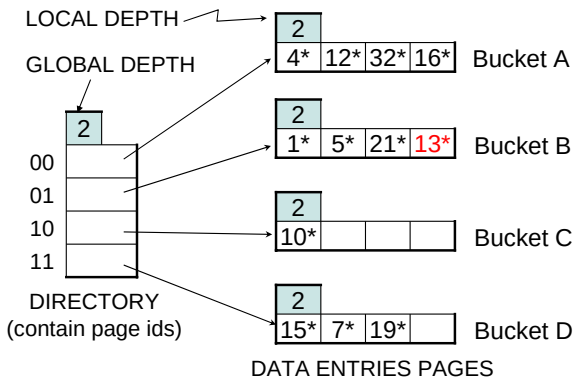
Extendible Hashing

- To avoid overflow page
  - Possible solution: if a bucket is full, reorganize file by doubling # of buckets and redistributing the entries across the new set of buckets
  - * *Reading and writing all pages is expensive!*
- Solution
  - Use a directory of pointers to buckets, double # of buckets by doubling the directory, splitting just the bucket that overflowed!
  - * Directory is 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!

Extendible Hashing (Cont.)

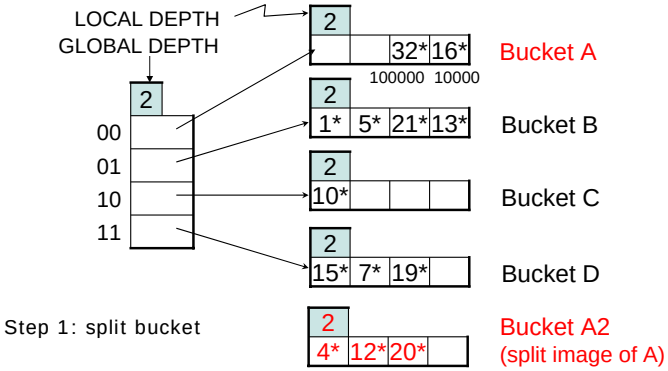
- Example
  - Directory is an array of size 4
  - To find the correct bucket for k
    - Take last global depth (2) bits of $h(k)$
      e.g., if $h(k)$ = 5 = 101 (binary), it is in the bucket pointed to by dictionary element 01
  - To insert
    - If bucket is full, split it (allocate a new page, re-distribute entries)
    - If necessary, double the directory

Extendible Hashing (Cont.)

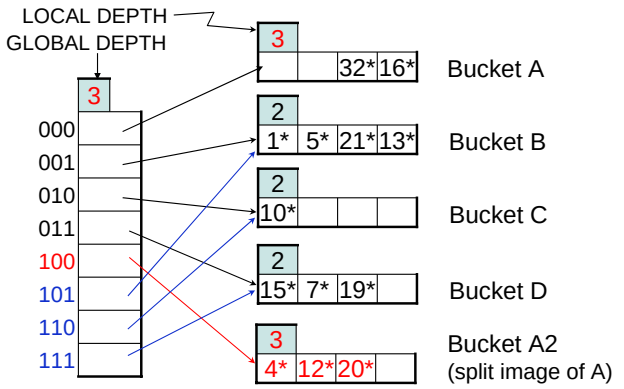


Extendible Hashing (Cont.)

- Insert entry r with h(r) = 20



Extendible Hashing (Cont.)



Step 2: double directory and increase depth

Extendible Hashing (Cont.)

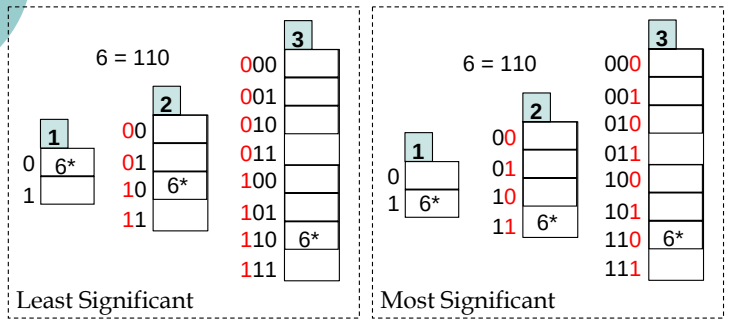
- Points to note
  - Allocate a new bucket page; write both this page and the old bucket page; double the dictionary array
  - *h(r)* = 20 = binary 10100: last 2 bits (00) tell *r* belongs in A or A2; last 3 bits (000 / 1000) are needed to tell which bucket
- Global depth of directory
  - Max # of bits 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

Extendible Hashing (Cont.)

- When does bucket split cause directory doubling?
  - Before inserting, local depth of bucket = global depth
  - Inserting causes local depth to become > global depth
- Directory is doubled by copying it over and adjusting pointer to split image page (use of least significant bits enables efficient doubling via copying of directory!)

Extendible Hashing (Cont.)

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



Extendible Hashing (Cont.)

- Delete
  - Locate the data entry by computing its hash value, taking the last bits, and looking in the bucket pointed to by this directory element
  - Remove the data entry
  - If the removal makes the bucket empty, it can be merged with its split image; local depth is decreased
  - If each directory element points to same bucket as its split image, the directory can be halved; global depth is decreased

* The last 2 steps can be omitted

Extendible Hashing (Cont.)

- I/O cost of equality search
  - If the directory fits in memory, equality search can be answered with one disk access
  - otherwise, two
- Collision (duplicate) handling
  - Collision: multiple data entries with the same hash value
  - Use overflow page when more data entries than will fit on a page have the same hash value

Linear Hashing

- An alternative to Extendible Hashing
- Not require directory; handle duplicates
- Idea: use a family of hash functions h_0, h_1, h_2, ...
  - $h_i(key) = h(key) \bmod (2^i N)$

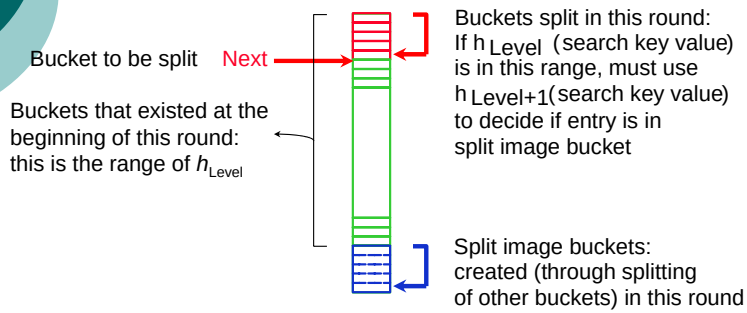
    N = initial # buckets; h is some hash function (range is not 0 to N-1)
  - If N = 2^{d0}, h_i consists of applying h and looking at the last d_i bits, where $d_i = d_0 + i$
  - h_{i+1} doubles the range of h_i (similar to directory doubling)

Linear Hashing (Cont.)

- Linear Hashing avoids directory by using overflow pages, and choosing bucket to split round-robin
  - Splitting proceeds in "rounds"; current round number is *Level* (initial value is 0)
  - Bucket to split is denoted by *Next*
    - *Next* is initialized to 0 when a new round begins, and increased by 1 after a splitting
    - Buckets (0 ~ *Next-1)* have been split; buckets (*Next* ~ N_{Level}) yet to be split
  - Splitting is triggered when an overflow page is added, and $h_{Level+1}$ redistributes entries between this bucket and its split image
  - The round *Level* ends when all N_{Level} initial buckets are split

Linear Hashing (Cont.)

- In the middle of a round



Linear Hashing (Cont.)

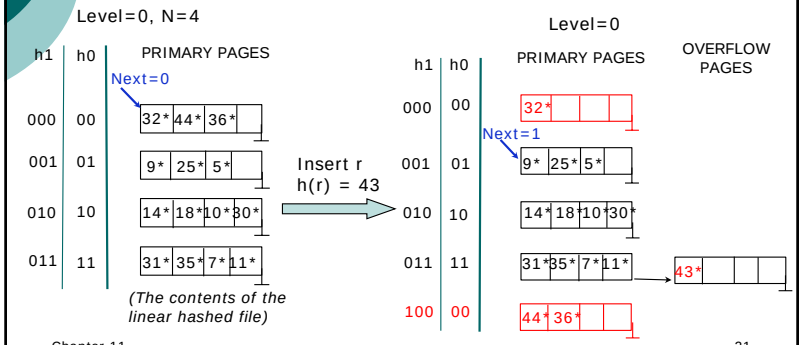
- Search for data entry r
  - To find bucket for data entry *r,* calculate $h_{Level}(r)$
  - If $h_{Level}(r)$ is in the range (*Next* ~ N_R), *r* belongs here
  - Else, *r* could belong to bucket $h_{Level}(r)$ or bucket $h_{Level}(r)$ + N_{Level}; must apply $h_{Level+1}(r)$ to find out
- Insert
  - Find bucket by applying h_{Level} / $h_{Level+1}$
  - If bucket to insert into is full
    - Add overflow page and insert data entry
    - Split *Next* bucket (its entries are redistributed by $h_{Level+1}$), and increment *Next*

Linear Hashing (Cont.)

- Actually any criterion can be chosen to trigger splitting
- Since buckets are split round-robin, long overflow chains don't develop!
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is implicit in how the # of bits examined is increased

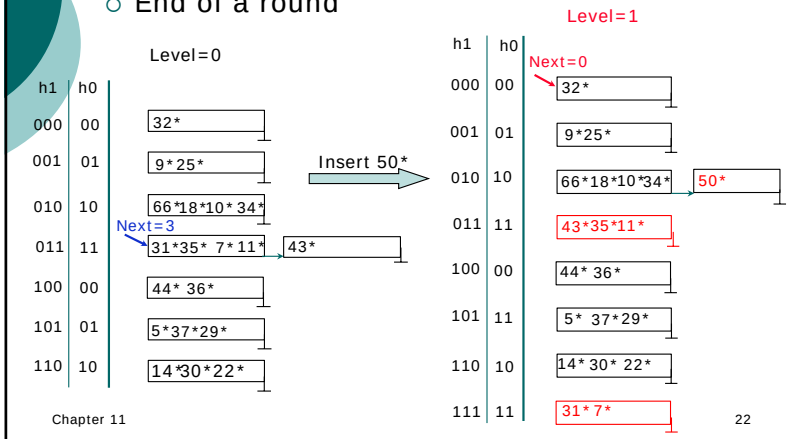
Linear Hashing (Cont.)

- On split, $h_{Level+1}$ is used to redistribute entries



Linear Hashing (Cont.)

- End of a round



Extendible vs. Linear Hashing

- The two schemes are actually quite similar
  - Begin with an imaginary directory in LH with N elements
  - First split is at bucket 0 (the imaginary directory is doubled at this point). Since elements <1,N+1>, <2,N+2>, ... are the same, we need only create directory element N, which differs from 0, now. When bucket 1 splits, create directory element N+1, etc.
  - The directory is doubled gradually. Also, primary bucket pages are created in order. If they are allocated in sequence too, we don't need a directory!

Extendible vs. Linear Hashing (Cont.)

Moving from h_i to h_{i+1} in LH corresponds to doubling the directory in EH

- Extendible Hashing
  - Directory is doubled in a single step
  - always splitting the appropriate bucket (dense data area): reduced # of splits and a higher bucket occupancy
- Linear Hashing
  - Directory is doubled gradually over the course of a round
  - A directory can be avoided by a clever choice of the buckets to split