CS222/CS122C: Principles of Data Management

UCI, Fall 2019
Notes #07

Hash Tables, Comparisons of Indexes

Instructor: Chen Li

- Index
  - Tree
    - Static
      - ISAM
    - Dynamic
      - B^+
  - Hash Table
    - Static
    - Dynamic
      - Extendable
      - Linear

look up for one specific record

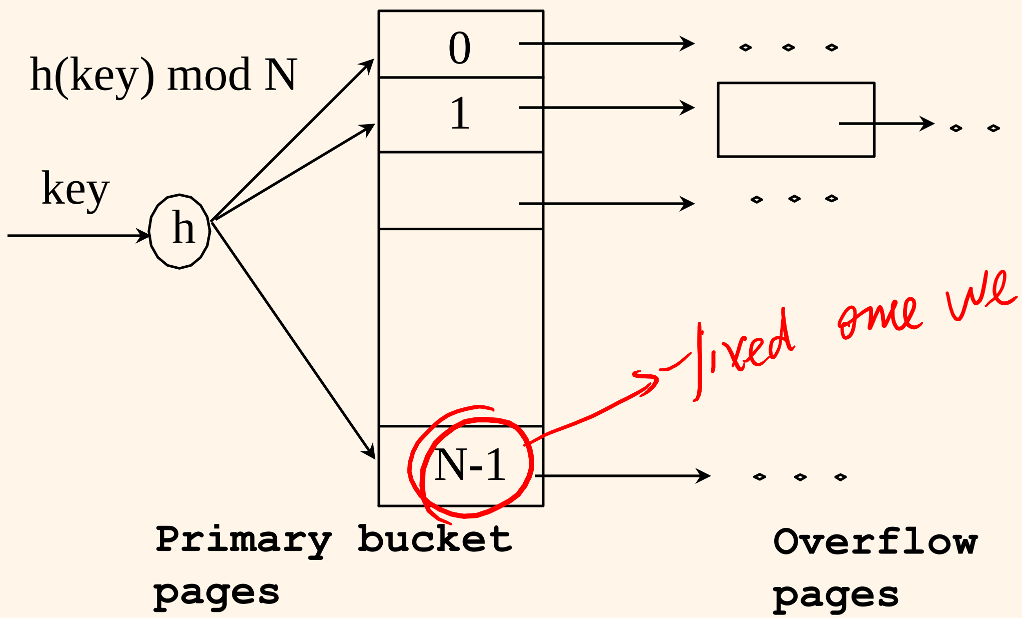
Introduction

- *Hash-based* indexes are best for *equality selections*. ***Cannot*** support range searches.
- Hashing
  - Static hashing
  - Dynamic hashing
    - Extendible hashing
    - Linear hashing
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

Static Hashing → know # of Records

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod M = bucket (page) to which data entry with key *k* belongs. (M = # of buckets)

hash function



desire to uniformly distribute the record across all the buckets.

fixed once we know number of records

Static Hashing (Contd.)

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.

Next: Dynamic hashing

- Extendible hashing
- Linear hashing

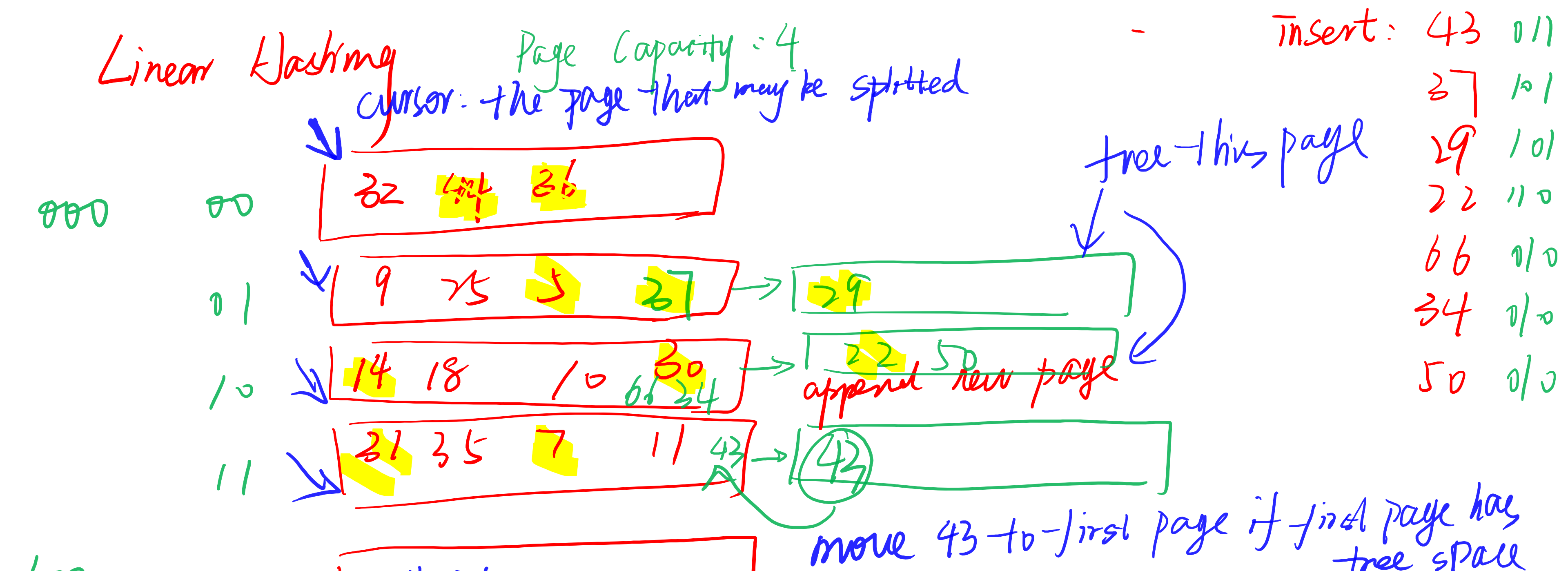
Ex: d0=2
so N=4

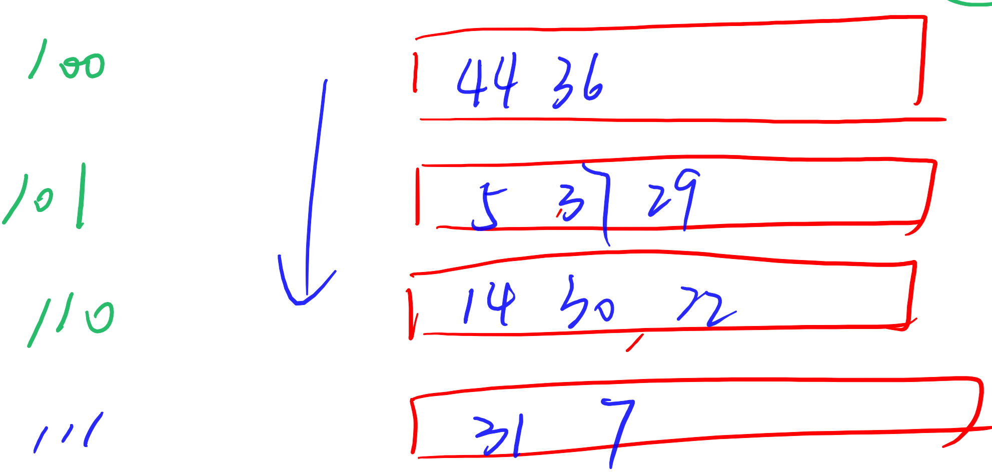
Linear Hashing

- *Idea*: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$; N = initial # buckets
  - **h** is some hash function (range is *not* just 0 to N-1)
  - If $N = 2^{d0}$, for some *d0*, $\mathbf{h}_i$ consists of applying **h** and looking at the last *di* bits, where $di = d0 + i$.
  - $\mathbf{h}_{i+1}$ doubles range of $\mathbf{h}_i$ (≈directory doubling)

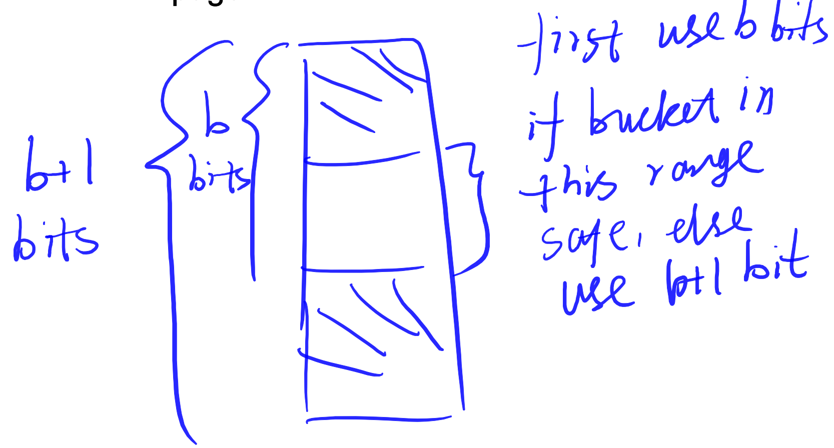
Ex: d0=2
so N=4

Linear Hashing





Each time we append a new page, we split the page where the cursor points to, and redistribute the data of the splitted page.\ And move the cursor to the next bucket. when there is an empty page, we free the page.



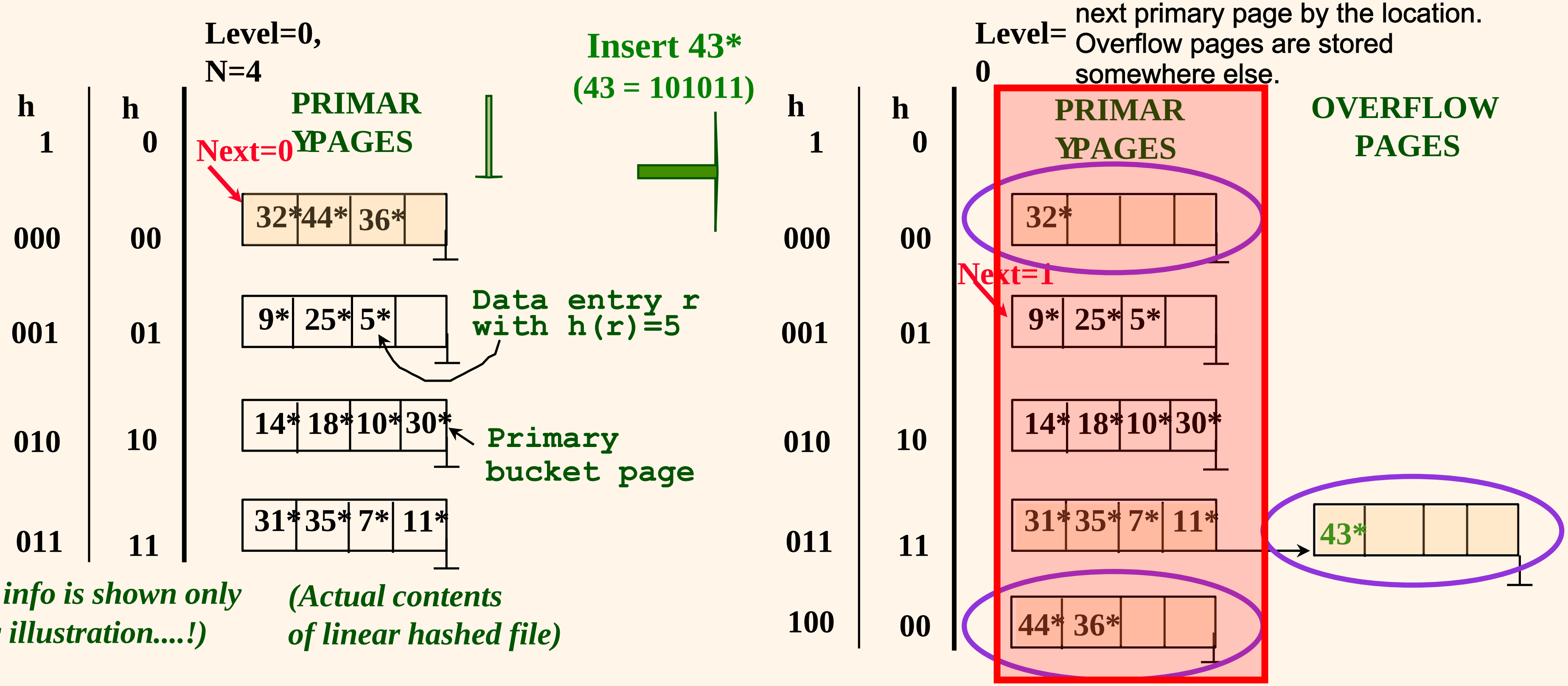
Linear Hashing (Contd.)

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round R) buckets are split. Buckets in 0 to *Next-1* have been split; *Next* to N_R have yet to be split.
  - Current round number is called *Level*.
- **Search:** To find bucket for data entry r, find $\mathbf{h}_{Level}(r)$:
  - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R', then r belongs here.
  - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or to bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out which.

Example of Linear Hashing

- On split, $\mathbf{h}_{Level+1}$ is used to re-distribute entries.

Primary page is stored compactly, and we don't use pointer to get the next page. Instead, we could get the next primary page by the location. Overflow pages are stored somewhere else.



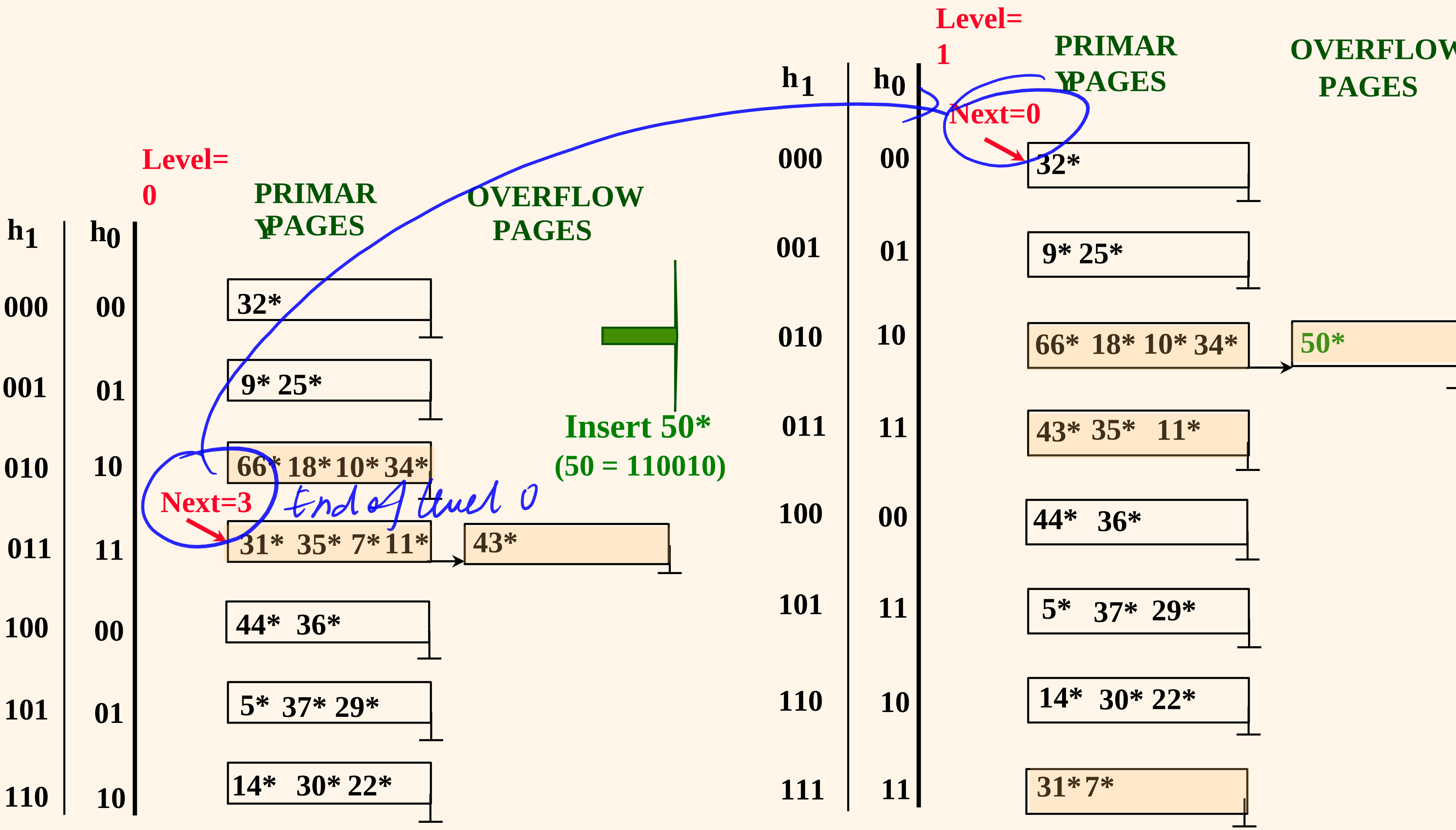
(h_i info is shown only for illustration....!)

(Actual contents of linear hashed file)

Insert records (see textbook)

- Insert h(r) = 43, 37, 29, 22, 66, 34, 50

Example: End of a Round

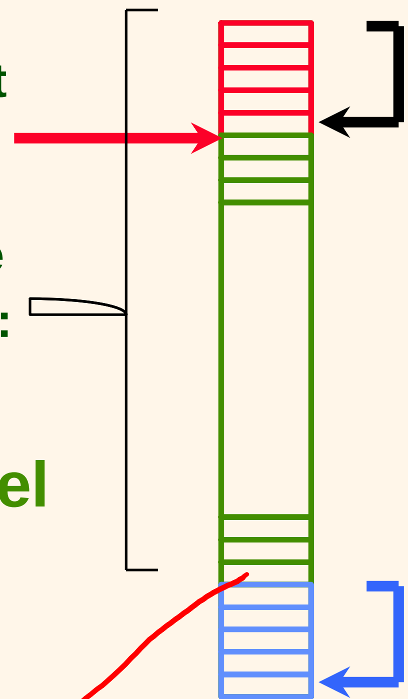


Overview of Linear Hashing

- In the middle of a round.

Bucket to be split
Next

Buckets that existed at the beginning of this round: this is the range of h_{Level}



Buckets split in this round: If h_{Level} (search key value) is in this range, must use $h_{Level+1}$ (search key value) to decide if entry is in `split image' bucket.

`Split image' buckets: created (through splitting of other buckets) in this round

once next reach NR, go back or go on?

Linear Hashing (Contd.)

- **Insert**: Find bucket by applying $\mathbf{h}_{Level}$ / $\mathbf{h}_{Level+1}$:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (*Maybe*) Split *Next* bucket and increment *Next*.
- Can choose any criterion to 'trigger' a split.
- Since buckets are split round-robin, long overflow chains don't develop! (See why?)

Comparisons of Indexes

B: total number of pages that are used to store records, each page is full.
R: total number of records per page.

Assumptions:

- B+ tree: ⅔ space utilization ratio per page
  - So B * 1.5 pages of records
- Size of an index entry = 1/10 of record entry
  - So an unclustered index B+ page: 1.5 * 1/10 = 0.15 pages
- Hash table: 80% space occupancy
  - So each of pages for an unclustered hash table: B * 1/0.8/10 = 0.125 B

$\frac{1}{10}$ of hash pages store index

cluster B+ tree: store records in the B+ tree.

unclustered tree: store key in B+ tree, key points to a record in a heap file.

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD average is 0.5 B	BD	2D	Search +D
(2) Sorted	BD	$D\log_2 B$	$D(\log_2 B$ + # pgs with match recs)	Search + BD	Search +BD influence all forward records
(3) Clustered	1.5BD	$D\log_F 1.5B$	$D(\log_F 1.5B$ + # pgs w. match recs)	Search + D write back to disk	Search +D
(4) Unclust. Tree index	BD(R+0.15)	$D(1 + \log_F 0.15B)$	$D(\log_F 0.15B$ + # pgs w. +# of records match recs)	Search + 2D write back to tree and heap file	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D → read index + read record.	BD (scan heap file) Because Hash has no order.	Search + 2D	Search + 2D

- *Several assumptions underlie these (rough) estimates!*