COS 597A:
Principles of
Database and Information Systems

Dynamic indexing structures

1

Last time

- File = a collection of (pages of) records
- File organizations:
  - two issues
    - how records assigned pages
    - how pages put on disk
  - 3 organizations
    - Heap: linked list (or directory) of pages
    - Sorted sequentially stored pages
    - Hashing: records in pages of buckets
- Indexing for more efficient retrieval
  - two types
    - index search key matches file organization
    - index organization independent of file organization

2

Search Tree Recap

- Motivation: get log(# file pages) search cost without needing sequential file for data or index
- Design strategy:
  - high fanout tree => shallow tree
  - each node fits in one file page
- Static versus dynamic

3

Dynamic Trees

- Tree changes to keep balance as file grows/shrinks
- Tree height: longest path root to leaf
- N data entries
  - clustered index: page of data file
  - unclusterd index: page of (value, record pointer) pairs
- Want tree height proportional to logN always

4

B+ Trees

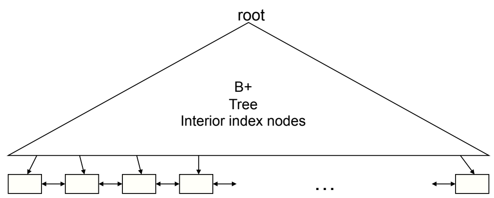
- Most widely used dynamic tree as index
- Most widely used index
- Properties
  - Data entries only in leaves
    - Compare B-trees
  - One page per tree node, including leaves
  - All leaves same distance from root => balanced
  - Leaves doubly linked
    - Gives sorted data entries
  - Call search key of tree "B+ key"

5

B+ trees continued

- To achieve equal distance all leaves to root cannot have fixed fanout
- To keep height low, need fanout high
  - Want interior nodes full
- Parameter d - order of the B+ tree
- Each interior node except root has m keys for $d \le m \le 2d$
  - m+1 children
- The root has m keys for $1 \le m \le 2d$
  - Tree height grows/shrinks by adding/removing root
- d chosen so each interior node fits in one page

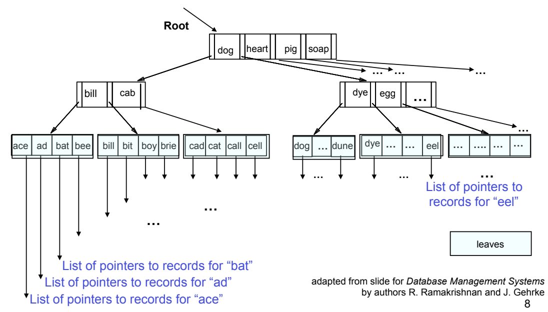
6



Leaves will be 1/2 full to full as well

Example B+ Tree

order = 2: 2 to 4 search keys per interior node

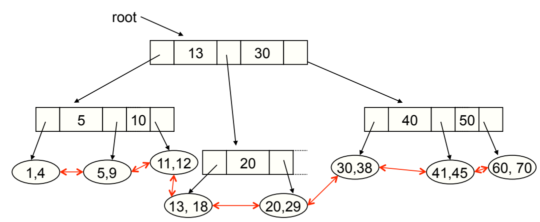


adapted from slide for *Database Management Systems* by authors R. Ramakrishnan and J. Gehrke

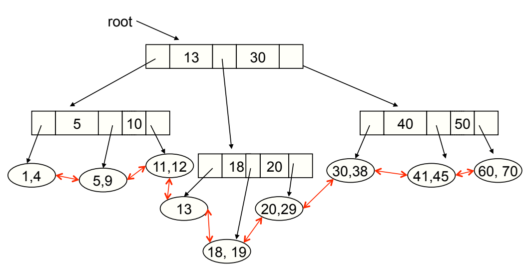
Inserting and Deleting

1. Method ➔ on board
2. Examples

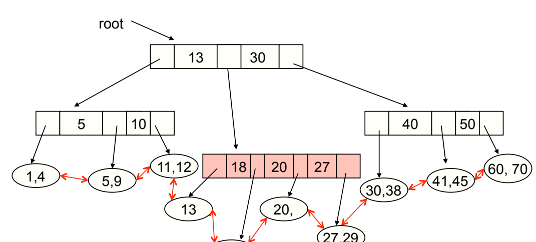
Starting configuration B+ tree of order d=1



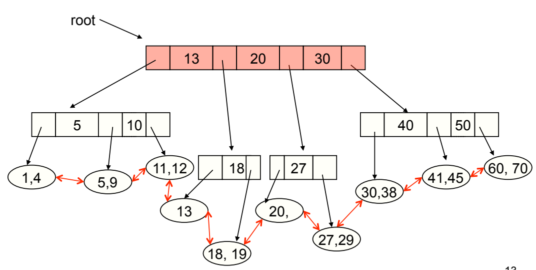
Insert 19: split leaf; expand parent with key 18



Insert 27 split leaf; expand parent with key 27 => too full

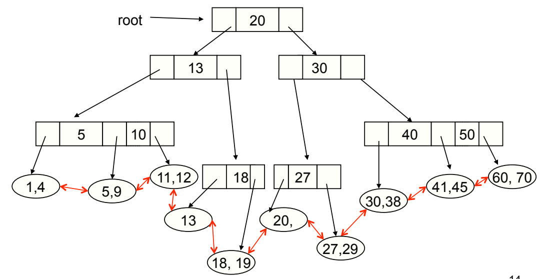


Insert 27
split leaf; split parent;
expand grandparent with key 20 => too full



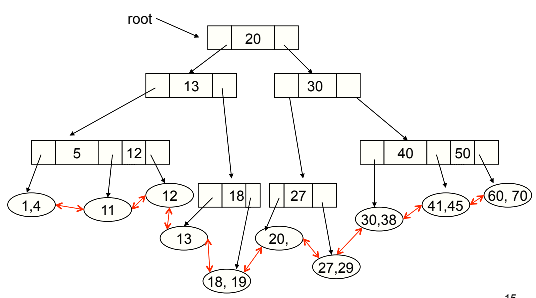
13

Insert 27
split leaf; split parent; split grandparent
new root with key 20



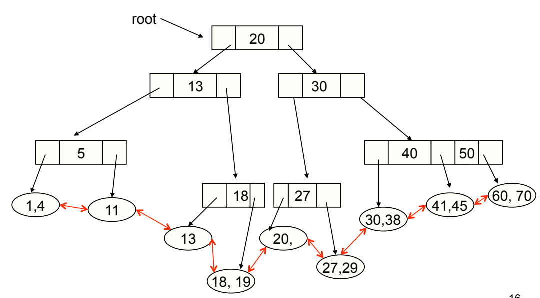
14

Delete 5, then 9
redistribute from right sibling



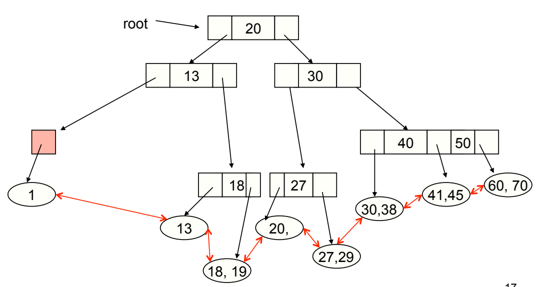
15

Delete 12
merge leaves, delete key from parent



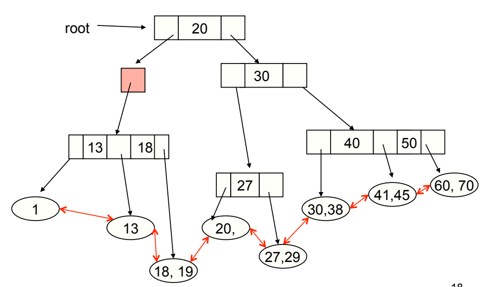
16

Delete 4, then 11
merge leaves, delete key from parent
=>parent not full enough



17

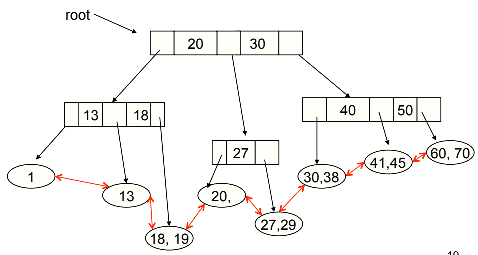
Delete 4, then 11
merge leaves, merge parent, bringing down key 13
=>grandparent not full enough



18

Delete 4, then 11

merge leaves; merge parent, bringing down key 13
merge grandparent, bring down key 20,
remove root



Dynamic hashing

- Have talked about static hash
  - Pick a hash function and bucket organization and keep it
  - Assume (hope) inserts/deletes balance out
  - Use overflow pages as necessary
- What if database growing?
  - Overflow pages may get too plentiful
  - Reorganize hash buckets to eliminate overflow buckets
    - Can't completely eliminate

Family of hash functions

- Static hashing:

  choose one good hash function *h*
  - What is "good"?

- Dynamic hashing:

  chose a family of good hash functions
  - $h_0, h_1, h_2, h_3, \dots h_k$
  - h_{i+1} refines h_i :

    if $h_{i+1}(x)= h_{i+1}(y)$ then $h_i(x)=h_i(y)$

A particular hash function family

- Commonly used: integers mod 2^i
  - Easy: low order i bits
- Base hash function: any *h* mapping hash field values to positive integers
- $h_0(x)= h(x) \bmod 2^b$ for a chosen *b*
  - 2^b buckets initially
- $h_i(x)= h(x) \bmod 2^{b+i}$
  - *Double buckets each refinement*
- *If x integer, $h(x)= x$ sometimes used*
  - What does this assume for h_0 to be *good?*

Specifics of dynamic hashing

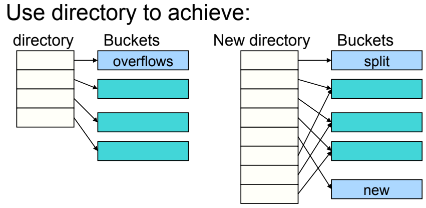
- Conceptually double # buckets when reorganize
- Implementation: don't want to allocate space may not need
  - One bucket overflows, double all buckets? **NO!**

Solution?
One choice: extendible hashing
- Reorganize when and where need

(Second choice in text book: linear hashing)

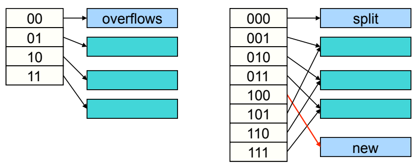
Extendible hashing

- When a bucket overflows,
  - actually split that bucket in two
  - Conceptually split all buckets in two
- Use directory to achieve:



Extendible hashing details

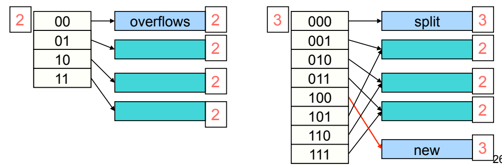
- Indexing directory with $h_i(x) = h(x) \bmod 2^{b+i}$
- On overflow, index directory with
  $h_{i+1}(x) = h(x) \bmod 2^{b+i+1}$
- Directory size doubles
- Add one bucket



- What did we do?
  - Split overflowing bucket m
    - Allocate new bucket
  - Copy directory
  - Change pointer of directory entry $m+2^{b+i}$

Keep track of how many bits actually using

- depth of directory: global depth
- depth of each bucket: local depth (WHY KEEP?)



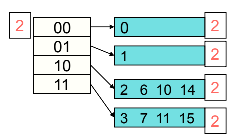
Rule of bucket splitting

- On bucket m overflow:
  - If depth(directory) > depth(bucket m)
    - Split bucket m into bucket m and bucket $m+2^{depth(m)}$
    - Update depth buckets m and $m+2^{depth(m)}$
    - Update pointers for all directory entries pointing to m
  - If depth(directory) = depth(bucket m)
    - Split bucket m into bucket m and bucket $m+2^{depth(m)}$
    - Update depth buckets m and $m+2^{depth(m)}$
    - Copy directory and update depth(directory)
    - Change pointer of directory entry $m+2^{depth(m)}$

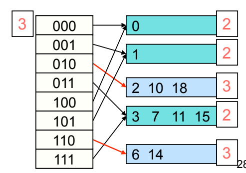
Example

Buckets: max 4 keys and data per bucket
Start with 4 buckets: depth(directory)=2

Insert records with hash values h(r) = 0, 1, 2, 3, 6, 10, 14, 7, 11, 15:



Then insert h(r) = 18
bucket '10' overflows
=> split

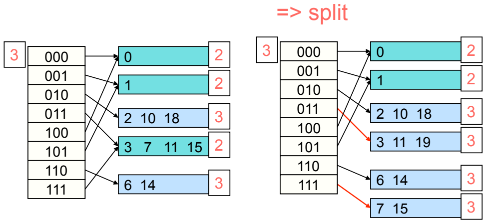


Example continued

Buckets: max 4 keys and data per bucket

After inserted h(r)=18:

Then insert h(r) = 19
bucket '11' overflows
=> split

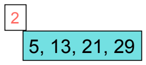


Extendible hashing observations

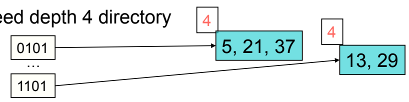
- Splitting bucket does not always evenly distribute contents
  - $h_i(x)$ may equal $h_{i+1}(x)$, $h_{i+2}(x)$, …
- May need to split bucket several times
  - NOT: global depth – min(local depth) = 1
- Can accept some overflow pages or split aggressively
- Almost no overflow pages with good hash function and aggressive splitting.
- If h(x) = h(y) always same bucket
  - cannot avoid overflow if too many of these!

Example bad bucket overflow

Bucket:



h(key) mod 2^2 = 1
h(key) mod 2^3 = 5
If add new entry with h(key)= 37 then h(key) mod 2^3 = 5
=>splitting once not enough
Need depth 4 directory



Index Operation Costs

Extendible Hashing Costs

Assume: One page per bucket; no overflow pages

- Look up: # pages read = 1 + 1
  - Assumes directory on disk
- Insert without overflow
  = look-up cost + 1 to write page of bucket
- Insert with overflow - splitting once:
  = look-up cost + 1 to write page of original bucket
  + 1 to write page of new bucket
  + 2 * (# disk pages of directory) to copy
- Splitting once may not be enough

Extendible Hashing Costs

One page per bucket; use some overflow pages

- Look up: add (# overflow pages) worst case
- Insert without splitting: add 1 **if** add new overflow page
- Insert with splitting once:
  add (# overflow pages) **always** to look-up cost
  add (# overflow pages) to write cost worst case
  - must read overflow pages to split
  - adding 1 new bucket (page), so end up with # overflow pages within 1 of number had before

B+ tree costs: preliminaries

- height of B+ tree = length of path: root → leaf
  $\leq \lceil \log_{d+1} (N) \rceil + 1$
  - N is number of leaves of tree
  - d+1 is min fanout of interior nodes except root
  - + 1 is for root
- typically root kept in memory
  - keep as many levels of tree as can in memory
  - buffer replacement algorithm may do, or pin

B+ tree costs: What is N?

- B+ tree file organization:
  - each leaf holds records
    $N \geq \lceil$(# records in file / # records fit in a page)$\rceil$
    $N \leq$ 2* $\lceil$(# records in file / # records fit in a page)$\rceil$
    assuming no duplicate search key values
- B+ tree primary index on sorted sequential file:
  - each leaf holds pointers to file pages
    - can be sparse index
      - one key value (smallest) for each file page
    - (key value, pointer) pairs in leaves
      - assume fit between d and 2d in leaf
    $\lceil$(# pages in file) / 2d)$\rceil \leq N \leq \lceil$(# pages in file) / d)$\rceil$
    assuming no key value spans multiple pages

B+ tree costs: What is N?

- B+ tree secondary index:
  - each leaf holds pointers to page of pointers
    - indirection: pointers in point to records
    - must be dense
    - (key value, pointer) pairs in leaves
      - assume fit between d and 2d in leaf

$N \le \lceil(\text{\# key values in file}) / d)\rceil$

$N \ge \lceil(\text{\# key values in file}) / 2d)\rceil$

37

B+ tree costs: retrieval

- retrieving single record

  \# of pages accessed =

  height of B+-tree

  \+ 1 for root if on disk

  $+ \begin{cases} 1 & \text{if leaves pt to records} \\ 2 & \text{if leaves pt to page of pointers to records} \end{cases}$

  $\le \lceil \log_{d+1} (N) \rceil + 3$

- typical height?

38

Indexing summary

- dynamic search tree: B+ trees
- dynamic hash table: extendible hashing
- size of index depends on parameters
  - dense or sparse?
  - storing records? pointers to records?
    pointers to pages of pointers to records?
- disk I/O cost same order as "in core" running time.
  - hash constant time
  - search tree as log(N)

39