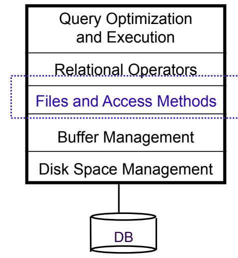
Indexing

"If you don't find it in the index, look very carefully through the entire catalogue."
— Sears, Roebuck, and Co., Consumer's Guide, 1897

Context



Multiple File Organizations

Many alternatives exist, *each good for some situations, and not so good in others:*

- Heap files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best for retrieval in *search key* order, or only a `range' of records is needed.
- Clustered Files (with Indexes): Coming soon…

Cost Model for Analysis

B: The number of data blocks
R: Number of records per block
D: (Average) time to read or write disk block

Average-case analyses for *uniform random* workloads

We will ignore:
- Sequential vs. Random I/O
- Pre-fetching
- Any in-memory costs

☛ ***Good enough to show the overall trends!***

More Assumptions

Single record insert and delete.
Equality selection - exactly one match
For Heap Files:
- Insert always appends to end of file.

For Sorted Files:
- Files compacted after deletions.
- Selections on search key.

Question all these assumptions and rework
- As an exercise to study for tests, generate ideas

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records			
Equality Search			
Range Search			
Insert			
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	
Equality Search			
Range Search			
Insert			
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	
Equality Search	½BD (primary, if found)	$(\log_2 B) * D$	
Range Search			
Insert			
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	
Equality Search	½BD (primary, if found)	$(\log_2 B) * D$	
Range Search	BD	$[(\log_2 B) +$ #match pages$]*D$	
Insert			
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	
Equality Search	½BD (primary, if found)	$(\log_2 B) * D$	
Range Search	BD	$[(\log_2 B) +$ #match pages$]*D$	
Insert	D+D (read/write)	$((\log_2 B)+½B+½B)D$ (read/write half)	
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	
Equality Search	½BD (primary, if found)	$(\log_2 B) * D$	
Range Search	BD	$[(\log_2 B) +$ #match pages$]*D$	
Insert	D+D (read/write)	$((\log_2 B)+½B+½B)D$ (read/write half)	
Delete	½BD + D (primary, if found)	$((\log_2 B)+½B+½B)D$ (read/write half)	

Indexes

Allow record retrieval *by value* in ≥1 field:
- Find all students in the "CS" department
- Find students with a gpa > 3
- Find students with firstname "Bob", lastname "Nob"

Index : disk-based data structure for fast lookup by value
- *Search key:* any subset of columns in the relation.
  - *Search key* need not be a *key* of the relation
  - I.e. There can be multiple items matching a search key

Index contains a collection of *data entries*
- (***k***, {items})
- Items associated with each search key value ***k***
- Data entries come in various forms, as we'll see

Alternatives for Data Entry **k*** in Index

Three alternatives:

1. Actual data record (with key value **k**)
2. <**k**, rid of matching data record>
3. <**k**, list of rids of matching data records>

Choice is orthogonal to the indexing technique.
B+ trees, hash-based structures, R trees, GiSTs, …

Can have multiple (different) indexes per file.
E.g. file sorted by *age*, with a hash index on *salary* and a B+tree index on *name*.

Alternatives for Data Entries (Contd.)

Alternative 1:
Actual data record (with key value ***k***)

- Index as a file organization for records
  - Alongside Heap files or sorted files
- At most one Alt. 1 index per relation
- No "pointer lookups" to get data records

Alternatives for Data Entries (Contd.)

Alternative 2
<**k**, rid of matching data record>
Alternative 3
<**k**, list of rids of matching data records>

- Alts. 2 or 3 *required* to support multiple indexes per relation!
- Alt. 3 more compact than Alt. 2,
  … but *variable sized data entries*
  - even if search keys are of fixed length.
- For large rid lists, data entry spans multiple blocks (!)

Clustered vs. Unclustered Index

In a clustered index:

- index data entries are stored in (approximate) order by value of search key in data records
- A file can be clustered *on at most one* search key.
- Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
- Alternative 1 => clustered
  - but not vice-versa!

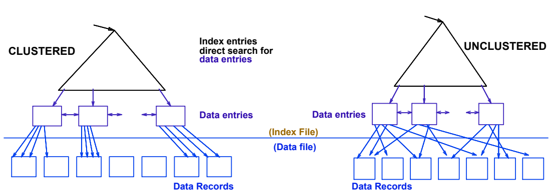
Note: there is another definition of "clustering"

- Data Mining/AI: grouping similar items in n-space

Clustered vs. Unclustered Index

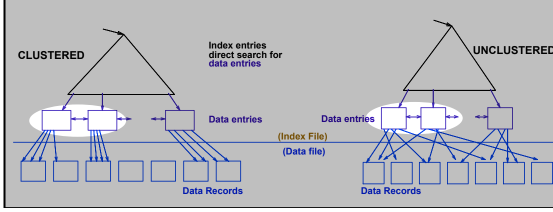
Alternative (2) data entries, data records in a Heap file.

- To build clustered index, first sort the Heap file
  - with some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
  - Thus, order of data recs is `close to', but not identical to, the sort order.



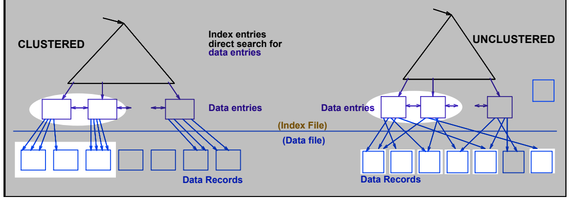
Clustered vs. Unclustered Index

- Alternative (2) data entries, data records in a Heap file.
  - To build clustered index, first sort the Heap file
    - with some free space on each block for future inserts
  - Overflow blocks may be needed for inserts.
    - Thus, order of data recs is `close to', but not identical to, the sort order.



Clustered vs. Unclustered Index

- Alternative (2) data entries, data records in a Heap file.
  - To build clustered index, first sort the Heap file
    - with some free space on each block for future inserts
  - Overflow blocks may be needed for inserts.
    - Thus, order of data recs is `close to', but not identical to, the sort order.



Unclustered vs. Clustered Indexes

Clustered Pros
- Efficient for range searches
- Potential locality benefits
  - Disk scheduling, prefetching, etc.
- Support certain types of compression
  - More soon on this topic

Clustered Cons
- More expensive to maintain
  - on the fly or "sloppily" via reorgs
  - Heap file usually only packed to 2/3 to accommodate inserts

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	1.5 BD
Equality Search	½BD (primary, if found)	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pages]*D	$[(\log_F 1.5B) +$ *#match pages*]*D
Insert	D+D (read/write)	$((\log_2 B)+½B+½B)D$ (read/write half)	$((\log_F 1.5B)+2) * D$
Delete	½BD + D (primary, if found)	$((\log_2 B)+½B+½B)D$ (read/write half)	$((\log_F 1.5B)+2) * D$

Tree-Structured Indexes

Selections of form field <op> constant
- Equality selections (op is =)
  - Either "tree" or "hash" indexes help here.
- Range selections (op is one of <, >, <=, >=, BETWEEN)
  - "Hash" indexes don't work for these.

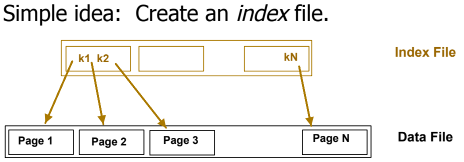
Or more complex selections (e.g. spatial containment)
There are fancier indexes that can do this…

Tree-structured indexing =>
both *range selections* and *equality selections*.
- *ISAM*: static structure; early technology (not that useful).
  - ISAM =Indexed Sequential Access Method
- *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

Range Searches (ISAM)

"*Find all students with gpa > 3.0*"
- Sorted file?
  Binary search to find first, scan to find others.
- Cost of binary search in a database can be quite high. Q: Why?

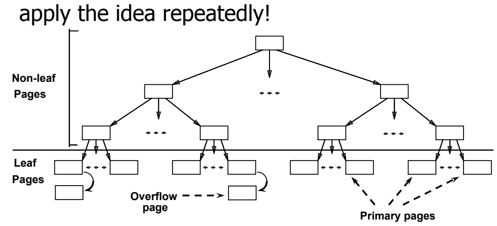
Simple idea: Create an *index* file.



☛ *Can do binary search on (smaller) index file!*

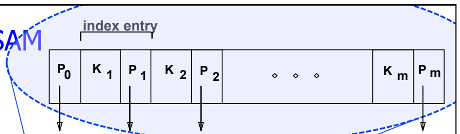
ISAM

Index file may still be quite large. But we can apply the idea repeatedly!

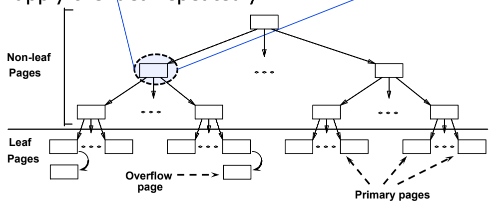


☛ *Leaf pages contain data entries.*

ISAM



Index file may still be quite large. But we can apply the idea repeatedly!

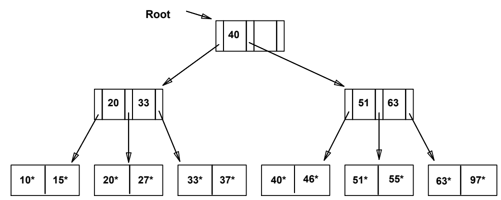


☛ *Leaf pages contain data entries.*

Example ISAM Tree

Index entries: **<search key value, page id>**

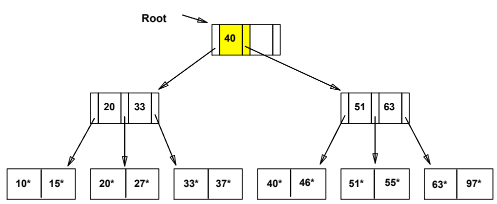
- Guide search for data entries *in leaves.*
- Example where each node can hold 2 entries:



Example ISAM Tree

Index entries: **<search key value, page id>**

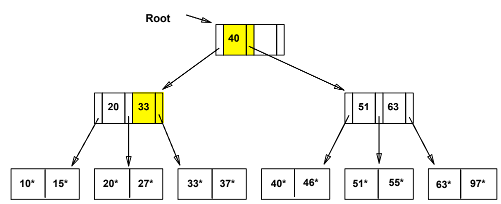
- Guide search for data entries *in leaves.*
- Example where each node can hold 2 entries:



Example ISAM Tree

Index entries: **<search key value, page id>**

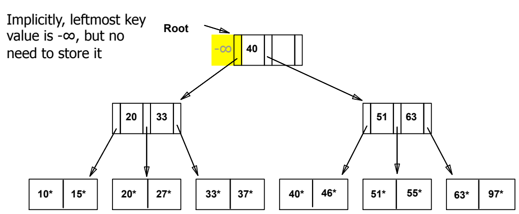
- Guide search for data entries *in leaves.*
- Example where each node can hold 2 entries;



Example ISAM Tree

Index entries: **<search key value, page id>**

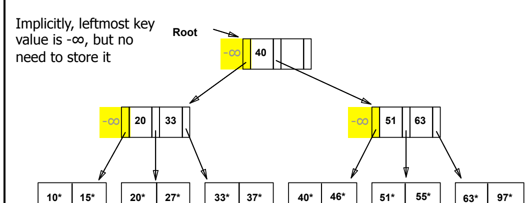
- Guide search for data entries *in leaves.*
- Example where each node can hold 2 entries:



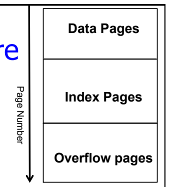
Example ISAM Tree

Index entries: **<search key value, page id>**

- Guide search for data entries *in leaves.*
- Example where each node can hold 2 entries:



ISAM is a STATIC Structure



- *File creation*:
  - Leaf (data) pages stored in order by search key
  - then index pages
  - then overflow pages
- *Search*: Start at root, follow to leaf
- Cost = $\log_F N$
  - F = # entries/page (i.e., fanout)
  - N = # leaf pages
  - no need for "next-leaf-page" pointers (Why?)
- *Insert*: Find leaf that data entry belongs to, and put it there. **Overflow page** if necessary.
- *Delete*: Seek and destroy! If deleting a tuple empties an overflow page, de-allocate it and remove from linked-list.

Static tree structure: *inserts/deletes affect only leaf pages.*

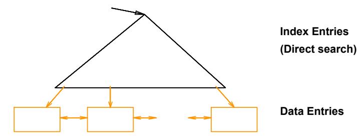
B+ Tree: The Most Widely Used Index

Insert/delete at $\log_F N$ cost; but keep tree *height-balanced*.
F = fanout, N = # leaf pages

Tree is not width-balanced: varying fanouts per node
- BUT: minimum 50% node occupancy (except for root).
- Each node contains *m entries where* d <= *m* <= 2d entries. "d" is called the *order* of the tree.

Supports equality and range-searches efficiently

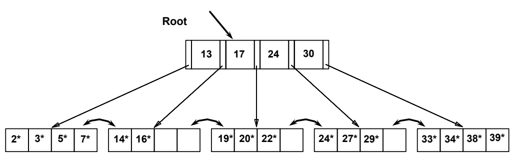
Like ISAM, but structure is dynamic.



Example B+ Tree

Search begins at root,
key comparisons direct it to a leaf

Search for 5*, 15*, all data entries >= 24* ...



B+ Trees in Practice

Typical order: 100. Typical fill-factor: 67%.
- So, average fanout = 133

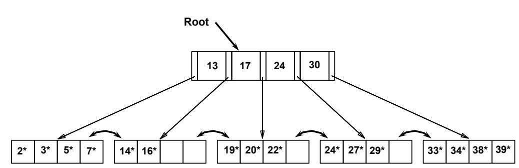
Typical capacities:
- Height 2: 133^3 = 2,352,637 entries
- Height 3: 133^4 = 312,900,700 entries

Can often hold top levels in buffer pool:
- `Level 1 =      1 page  =   4 Kbytes`
- `Level 2 =    133 pages =   1 Mbyte`
- `Level 3 = 17,689 pages = 133 MBytes`

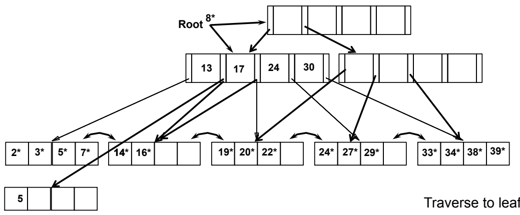
Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- Put data entry onto *L.*
  - If *L* has enough space, *done*
  - Else, must *split L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Example B+ Tree – Inserting 8*

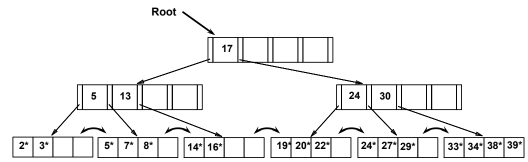


Animation: Insert 8*



Traverse to leaf
Split Leaf
Copy up middle key
Split Parent
New Root
Push up middle key

Final B+ Tree - Inserting 8*



❖ Notice that root was split, leading to increase in height.

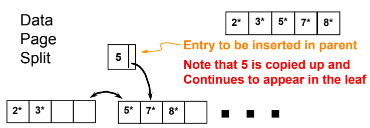
❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Data vs. Index Page Split
(from previous example of inserting "8*")

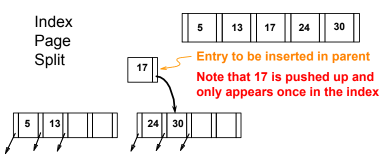
Observe how minimum occupancy is guaranteed in both leaf and index page splits.

Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.

Data Page Split



Index Page Split



Deleting a Data Entry from a B+ Tree

Start at root, find leaf *L* where entry belongs.

Remove the entry.

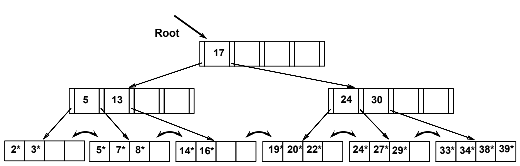
- If L is at least half-full, *done!*
- If L has only **d-1** entries,
  - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L).*
  - If re-distribution fails, *merge L* and sibling.

If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.

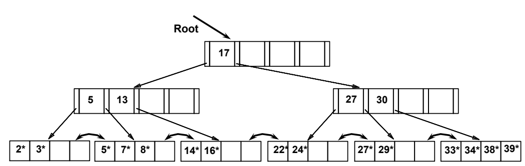
Merge could propagate to root, decreasing height.

In practice, many systems do not worry about ensuring half-full pages.
Just let page slowly go empty …
if it's truly empty, just delete from tree and leave unbalanced.

Example Tree (including 8*) Delete 19* and 20* ...



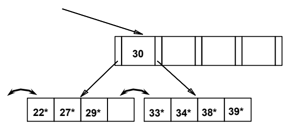
Example Tree (including 8*) Delete 19* and 20* ...

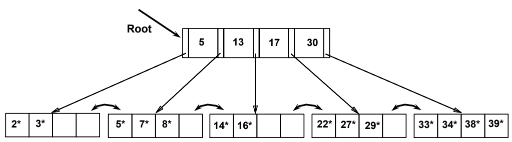


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

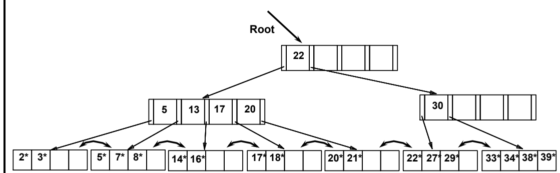
- Must merge.
- Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).





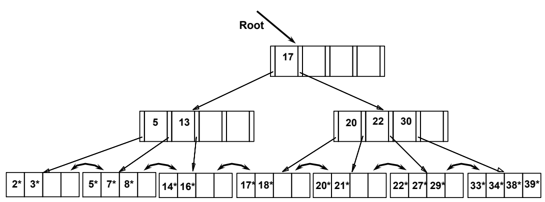
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.



After Re-distribution

- Intuitively, entries are re-distributed by `*pushing through*' the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.



Nice Animation On-Line

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

One small difference to note:

- Upon deletion of leftmost value in a node, it updates the parent index entry
- Incurs unnecessary extra writes

Prefix Key Compression

Important to increase fan-out. (Why?)

Key values in index entries just "direct traffic" => can often compress them.

Dannon Yogurt	Davey Jones	David Smith	Devarakonda Murthy

Prefix Key Compression

Important to increase fan-out. (Why?)

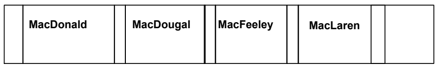
Key values in index entries just "direct traffic" => can often compress them.

Dan	Dave	Davi	De	

- Is this correct?
- Ensure that each index entry is greater than every key value (in any descendant leaf) to its left.
- In practice we compress upon "copy up" from leaf.

Suffix Key Compression

If many index entries share a common prefix

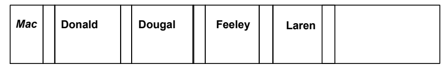


Particularly useful for composite keys

– Why?

Suffix Key Compression

If many index entries share a common prefix



Particularly useful for composite keys

– Why?

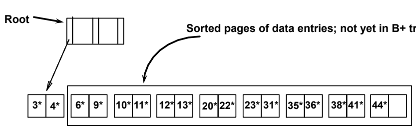
Bulk Loading of a B+ Tree

Given: large collection of records
- Desire: B+ tree on some field

Bad idea: repeatedly insert records
- Slow. Also poor leaf space utilization. Why?
- Play with the web animation: try to make up values that keep leaves full!

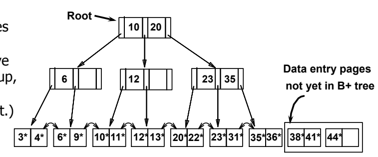
Bulk Loading can be done much more efficiently.

Initialization: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



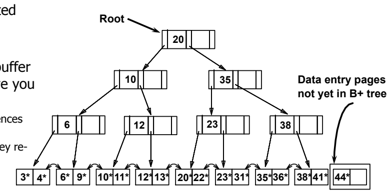
Bulk Loading (Contd.)

Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)



Much faster than repeated inserts.

Exercise: what kind of buffer pool hit rate will this give you for different policies?
- Q1: how many references per page?
- Q1: how often are they re-referenced?



Summary of Bulk Loading

Option 1: multiple inserts.
- Slow.
- Does not give sequential storage of leaves.

Option 2: *Bulk Loading*
- Fewer I/Os during build.
- Leaves will be stored sequentially (and linked, of course).
- Can control "fill factor" on pages.

A Note on `Order'

Order (**d**) makes little sense with variable-length entries

Use a physical criterion in practice: "at least half-full"
- Index pages often hold many more entries than leaf pages
- Variable sized records and search keys =>
  different nodes have different numbers of entries.
- Even with fixed length fields,
  Alternative (3) gives variable length

Many real systems are even sloppier than this --- only reclaim space when a page is *completely* empty.

Summary

- Data entries: 1 of 3 alternatives:
  1. actual data records
  2. <key, rid> pairs, or
  3. <key, rid-list> pairs.
  - Choice orthogonal to *indexing structure (i.e., tree, hash, etc.)*.
- Often multiple indexes per file of data records
  - each with a different search key.
- Indexes can be classified as *clustered* vs. *unclustered*
  - Difs have important consequences for utility/performance.

Summary (cont'd)

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure.
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
  - High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.
  - Typically, 67% occupancy on average.
  - Usually preferable to ISAM; adjusts to growth gracefully.
  - If data entries are data records, splits can change rids!

Summary (cont'd)

- Key compression increases fanout, reduces height.
- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- B+ tree widely used because of its versatility.
  - One of the most optimized components of a DBMS.