Database Systems

File Organization and Indexes

This Lecture…

- File Organization
- Indexes

Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

File Organization

- Three types
  - Heap File Organization
  - Sequential File Organization
  - Hashing File Organization

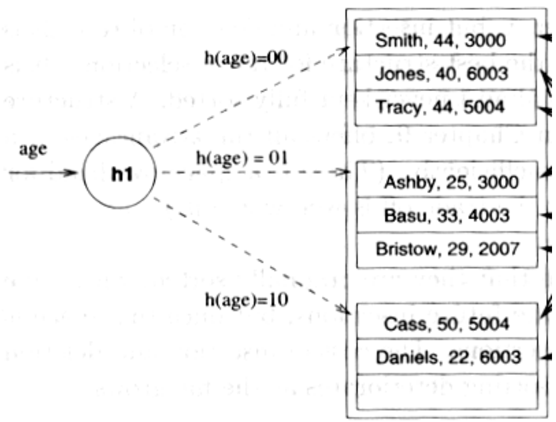
Alternative File Organizations

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

- Heap files: Suitable when typical access is a file scan retrieving all records.
  - Search (Equality/Range) needs to scan the file
  - Insert: At the end of file
  - Delete: Search for record and delete record
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
  - Search (Equality/Range): Efficient
  - Insert: Finding the position, inserting & move records
  - Delete Search for record, delete & move records

Alternative File Organizations… (contd.)

- Hashed Files: Good for equality selections.
  - File is a collection of *buckets.* Bucket = *primary* page plus zero or more *overflow* pages.
  - *Hashing function* **h**: **h**(*r*) = bucket in which record *r* belongs. **h** looks at only some of the fields of *r*, called the *search fields*.



File hashed on age

Alternative File Organizations… (contd.)

- Hashed Files:
  - Search (Equality): good for equality (if based on search key). Otherwise scan table
  - Search (Range): needs to scan the file
  - Insert: search for primary bucket (hash) and insert
  - Delete: search for primary bucket (hash) if available, else scan file & delete record

Structure of a file

- All data is stored in logical storage called **files**
- **Files** are considered to be a set of **pages**
- Pages are collection of **slots**
- Each slot contains a **record**
- Each record contains a **record id**
- *Record id = <page id, slot #>*

Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).

Characteristics

- Indexes provide fast access
- Indexes takes space
  - Need to be careful in creating only useful indexes
- May slow-down certain inserts/updates/deletes (maintain indexes)

Explain on board

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

- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.

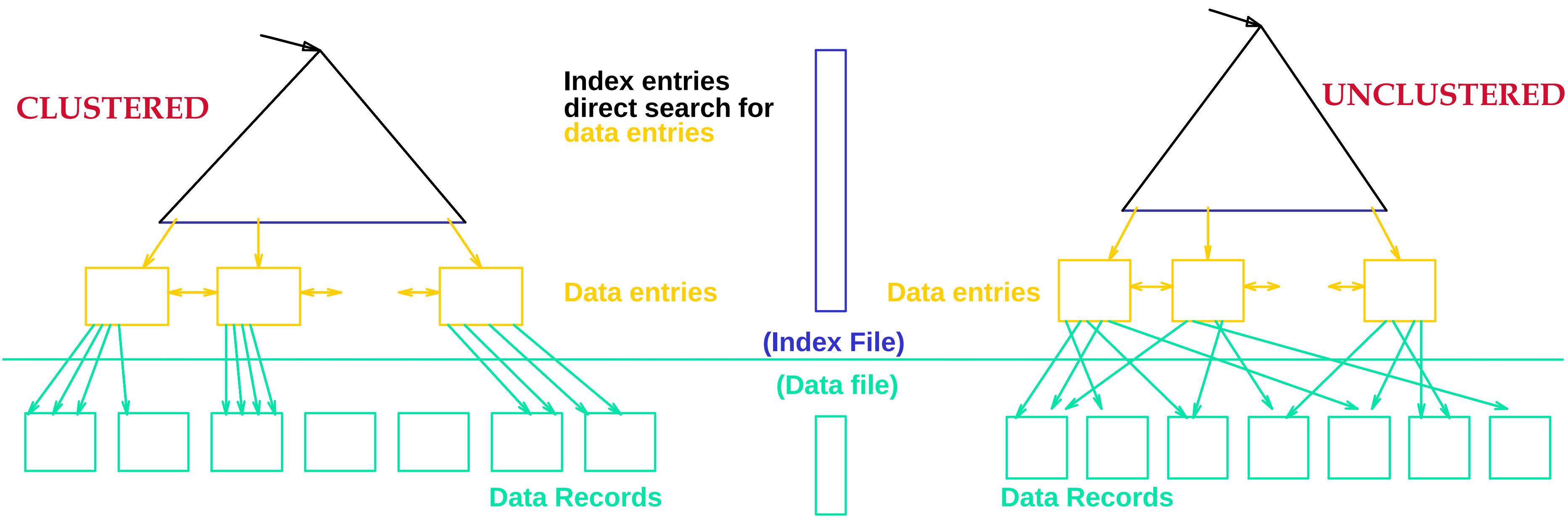
- Three alternatives:
  1. Data record with key value **k** (Alt. 1)
  2. <**k**, rid of data record with search key value **k**> (Alt. 2)
  3. <**k**, list of rids of data records with search key **k**> (Alt. 3)

Terminology

- File of records containing index entries = **index file**
- There are several organization techniques for building index files = **access methods**

Properties of Indexes...

- Clustered vs. Unclustered Index

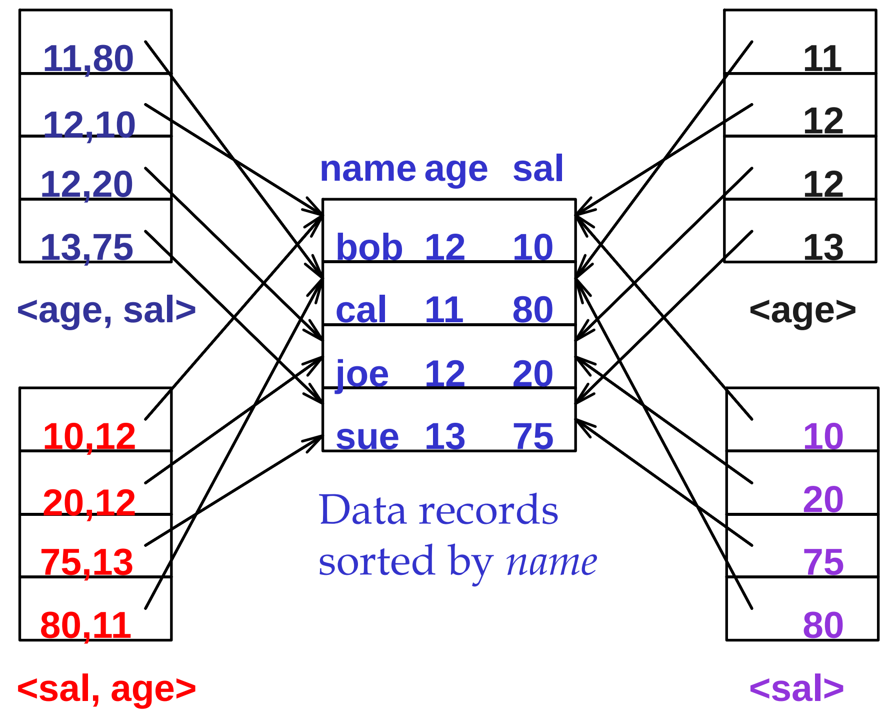


- Can have at most one clustered index per table
- Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Properties... (contd.)

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.

Examples of composite key indexes using lexicographic order.



Data records sorted by *name*

Data entries in index sorted by <*sal,age*>

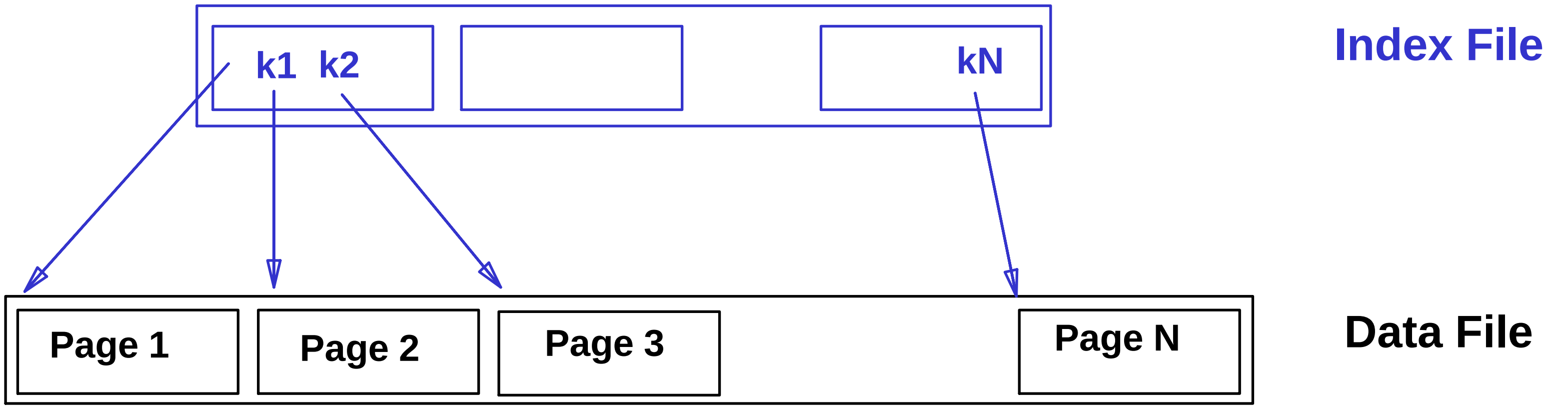
Data entries sorted by <*sal*>

Indexes in SQL…

- Index is not a part of SQL-92
- However, all major DBMSs provide facilities for index creation
  - CREATE INDEX…
  - DROP INDEX…
- Oracle and SQL Server support indexes (clustered and non-clustered indexes)

Range Searches

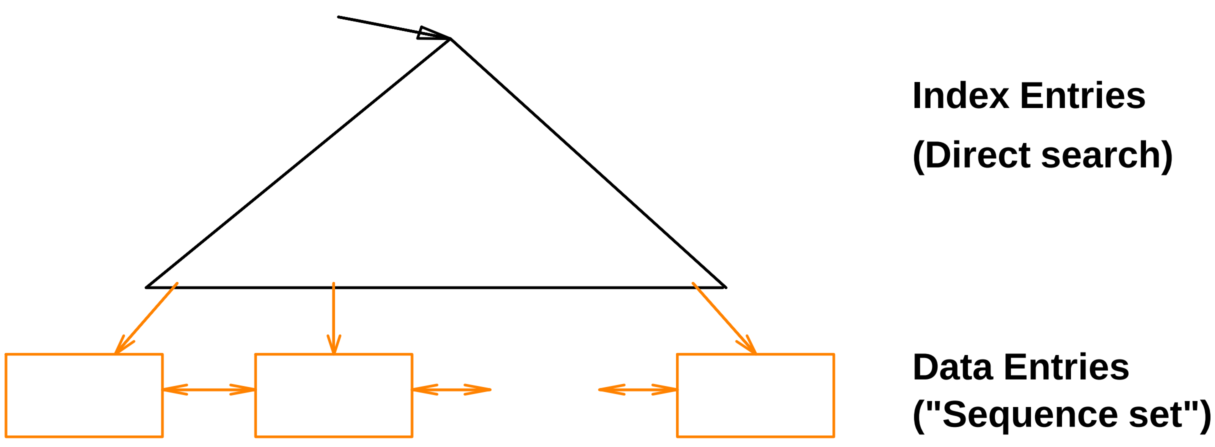
- *``Find all students with gpa > 3.0''*
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.



Can do binary search on (smaller) index file!

B+ Tree: The Most Widely Used Index

- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node (*except root*) contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.

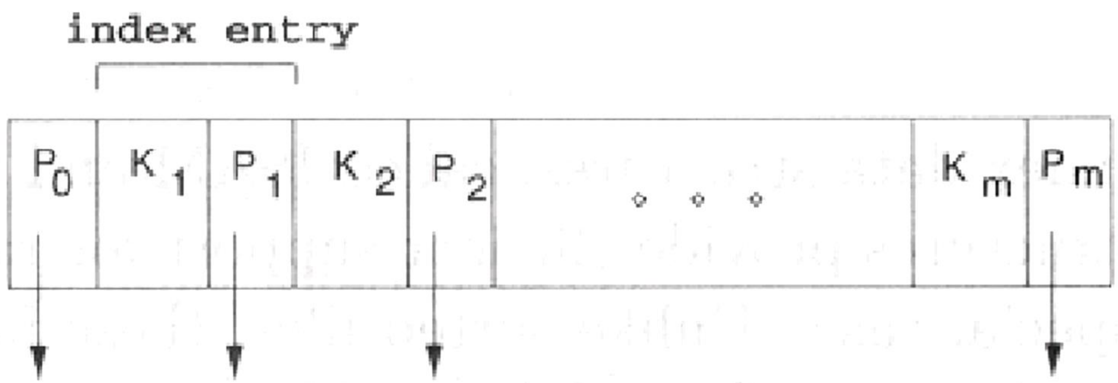


B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

B+ Tree…

- Search begins at root, and key comparisons direct it to a leaf
- Each Node has search keys (K_i) and pointers (P_i).
- P_i points to a sub-tree in which all key values K are such that $K_i \leq K < K_{i+1}$



Search

func *tree_search* (nodepointer, search key value *K)* **returns** nodepointer

// Searches tree for entry

if *nodepointer is a leaf, return nodepointer;

else,

if $K < K1$ then return tree_search(Po, *K);*

else,

if $K \geq Km$ then return tree_search(Pm, *K)* // m = # entries

else,

find *i* such that $Ki \leq K < Ki+1$;

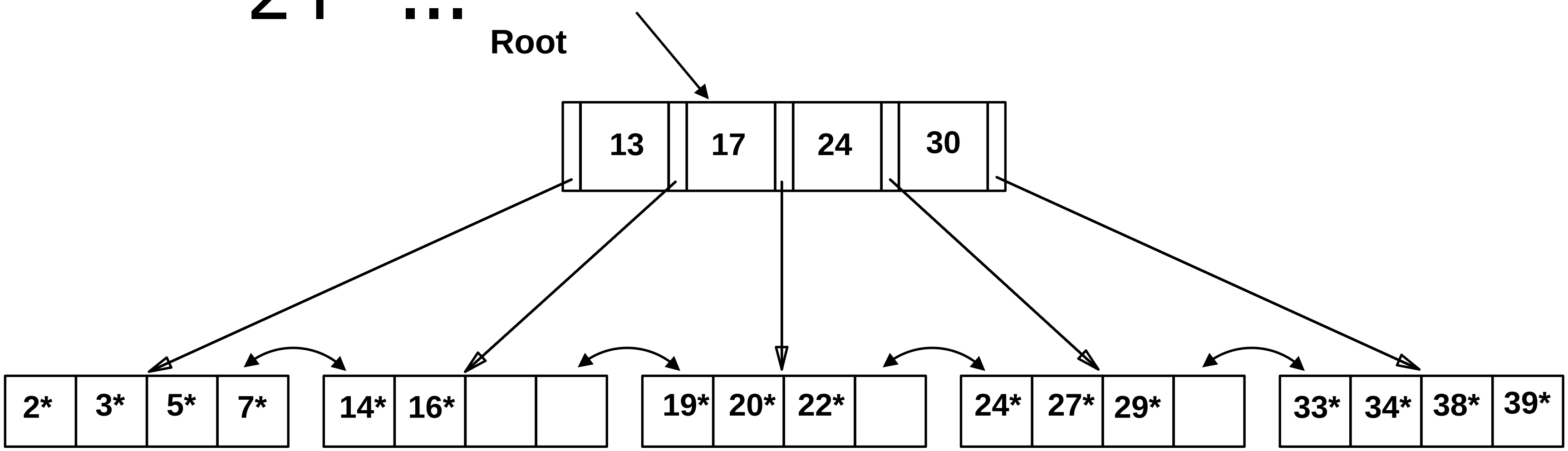
return tree_search(Pi, *K)*

end if

end if

Example B+ Tree…

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



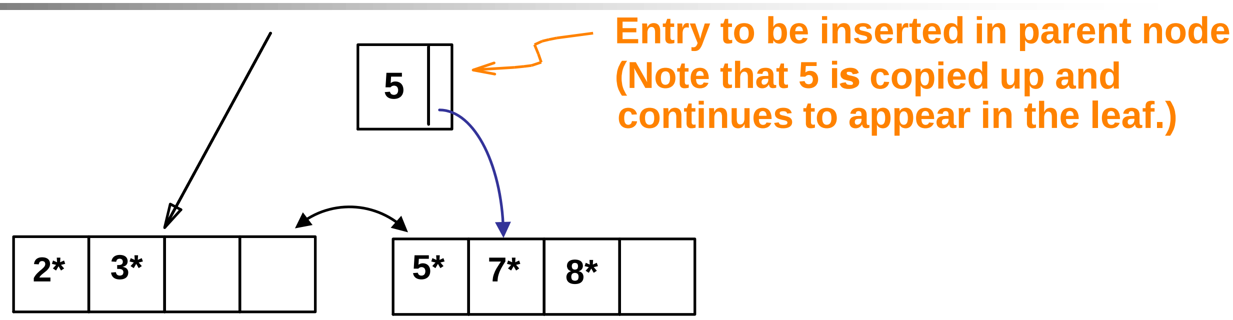
- *Based on the search for 15*, we <u>know</u> it is not in the tree!*

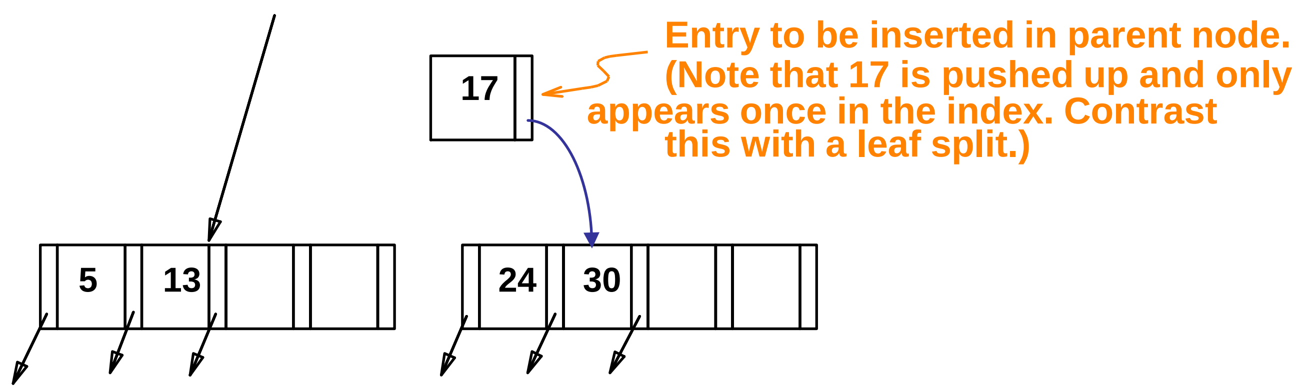
Inserting a Data Entry into a B+ Tree

- 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.*

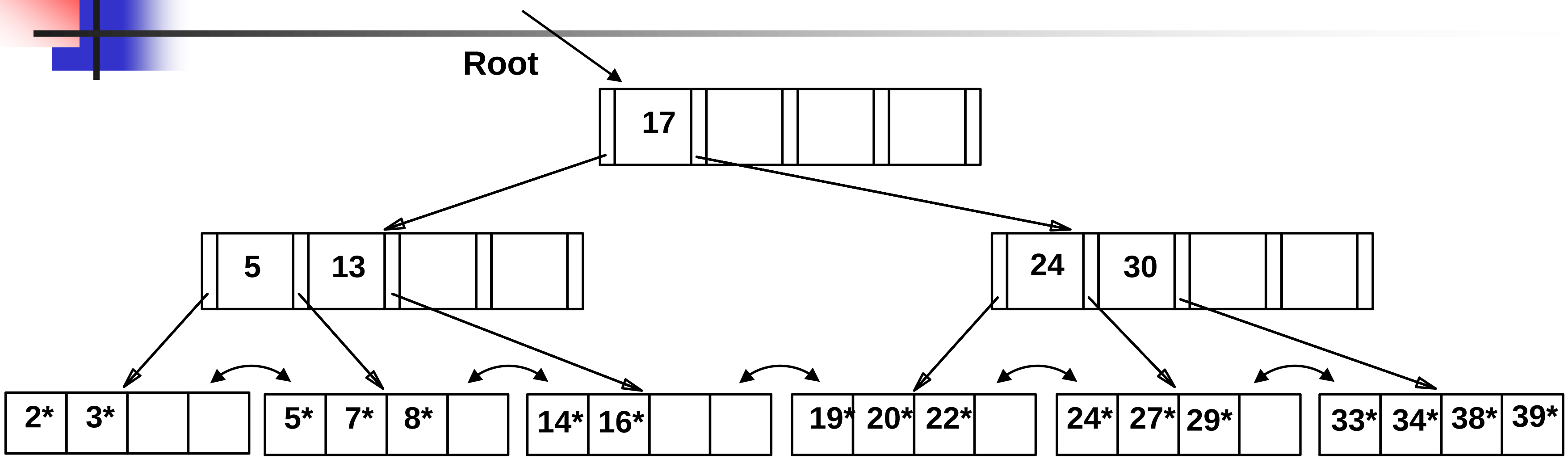
Inserting 8* into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.





Example B+ Tree After Inserting 8*

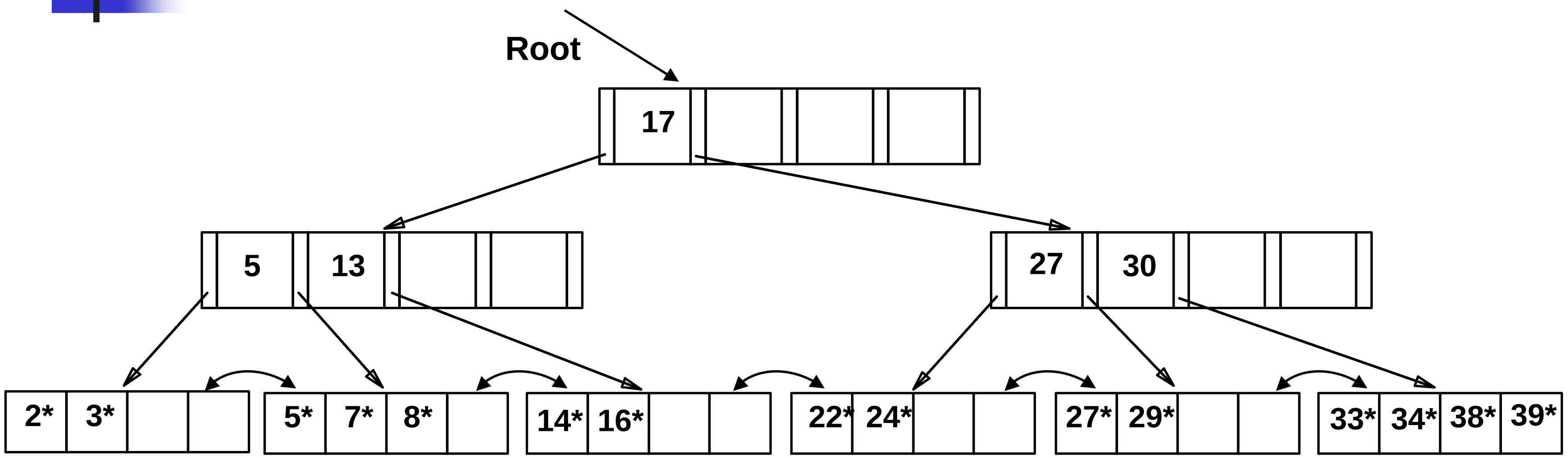


- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

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.

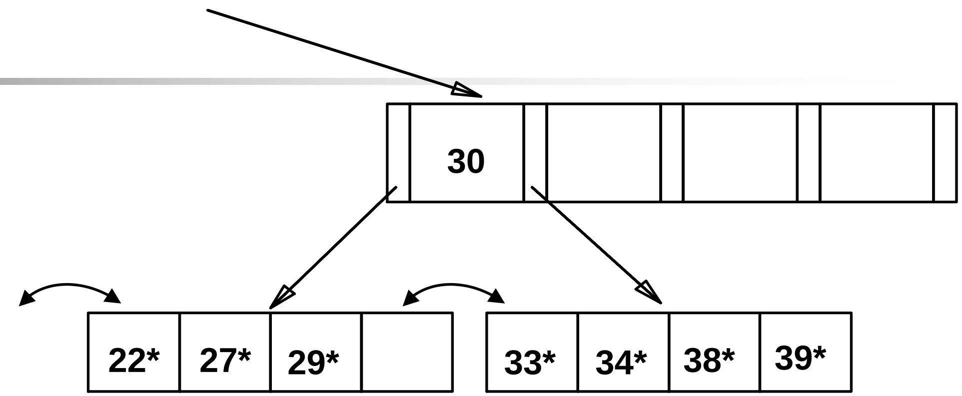
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

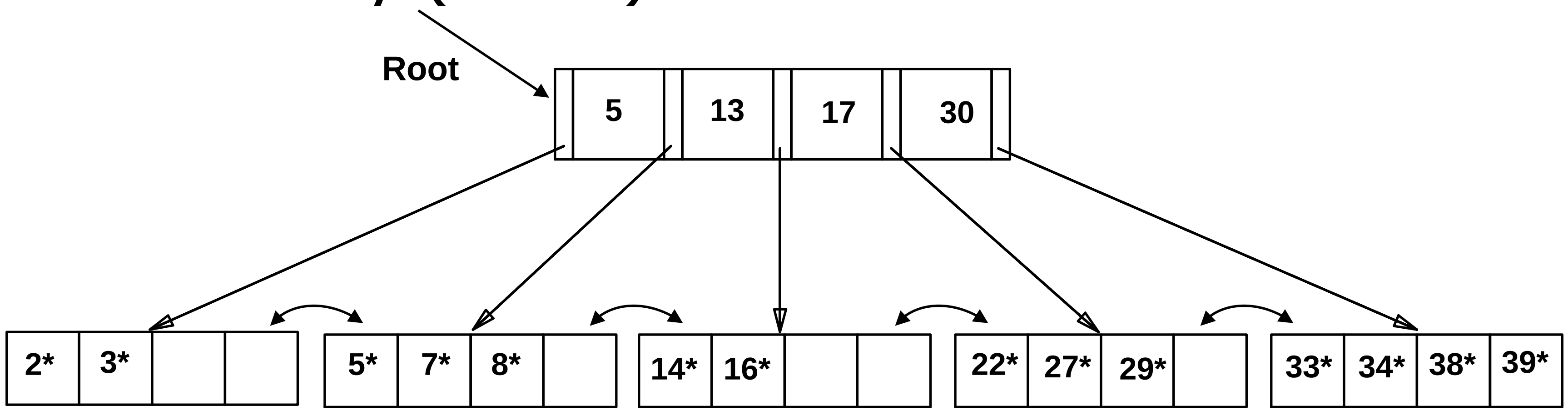


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

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





Duplicates in B+ Trees…

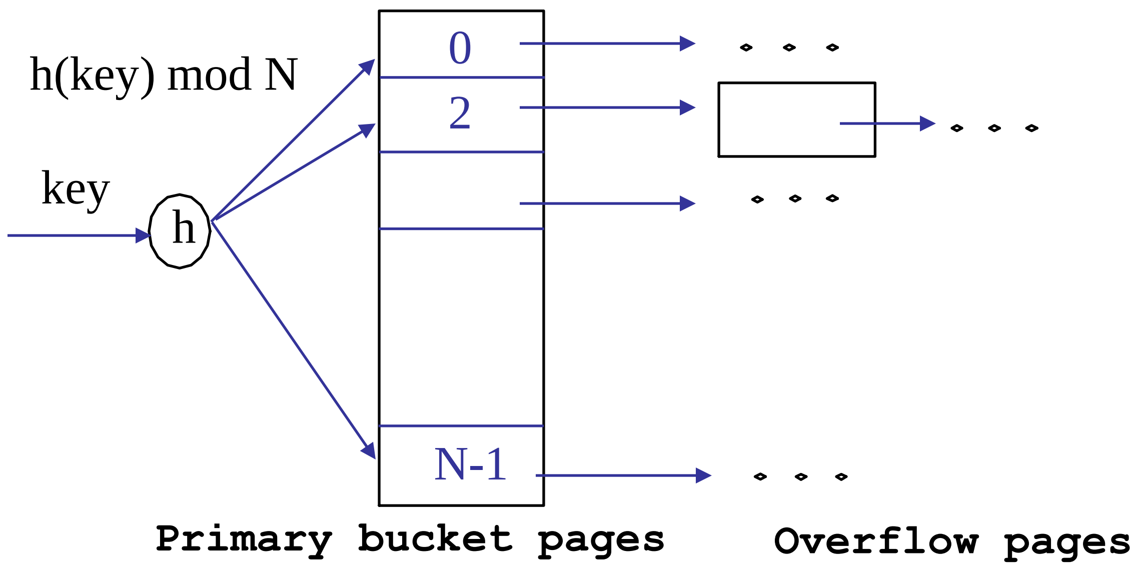
- We have ignored duplicates so far…
- Alternatives…
  - Overflow leaf pages
  - Duplicate values in the leaf pages
  - Make unique key values (by adding rowid's)
    - Preferred approach by many DBMSs

Hashing

- *Hash-based* indexes are best for *equality selections*.
- ***Cannot*** support range searches.
- Static and dynamic hashing techniques exists

Static Hashing

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



Static Hashing… (contd.)

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

Static Hashing… (contd.)

Problems…

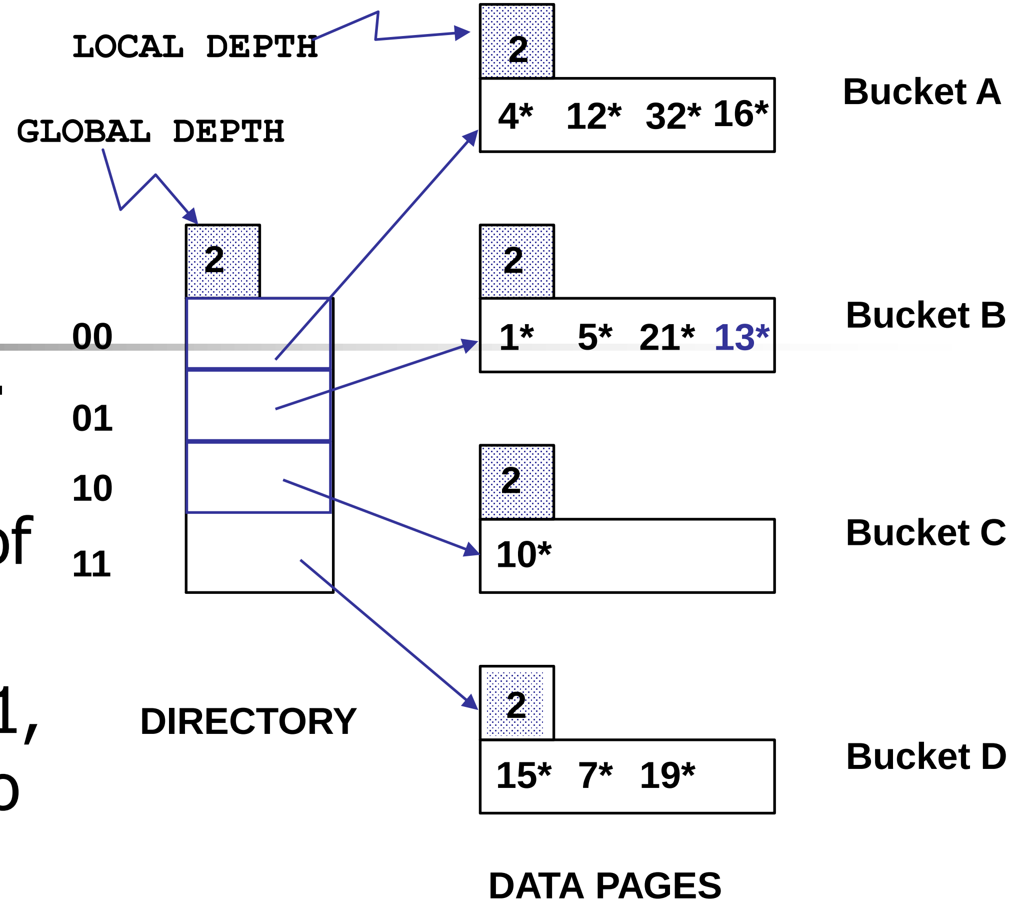
- Insertion can create long overflow chains can develop and degrade performance.
- Deletion may waste space
- *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

Extendible Hashing

- Situation: 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!

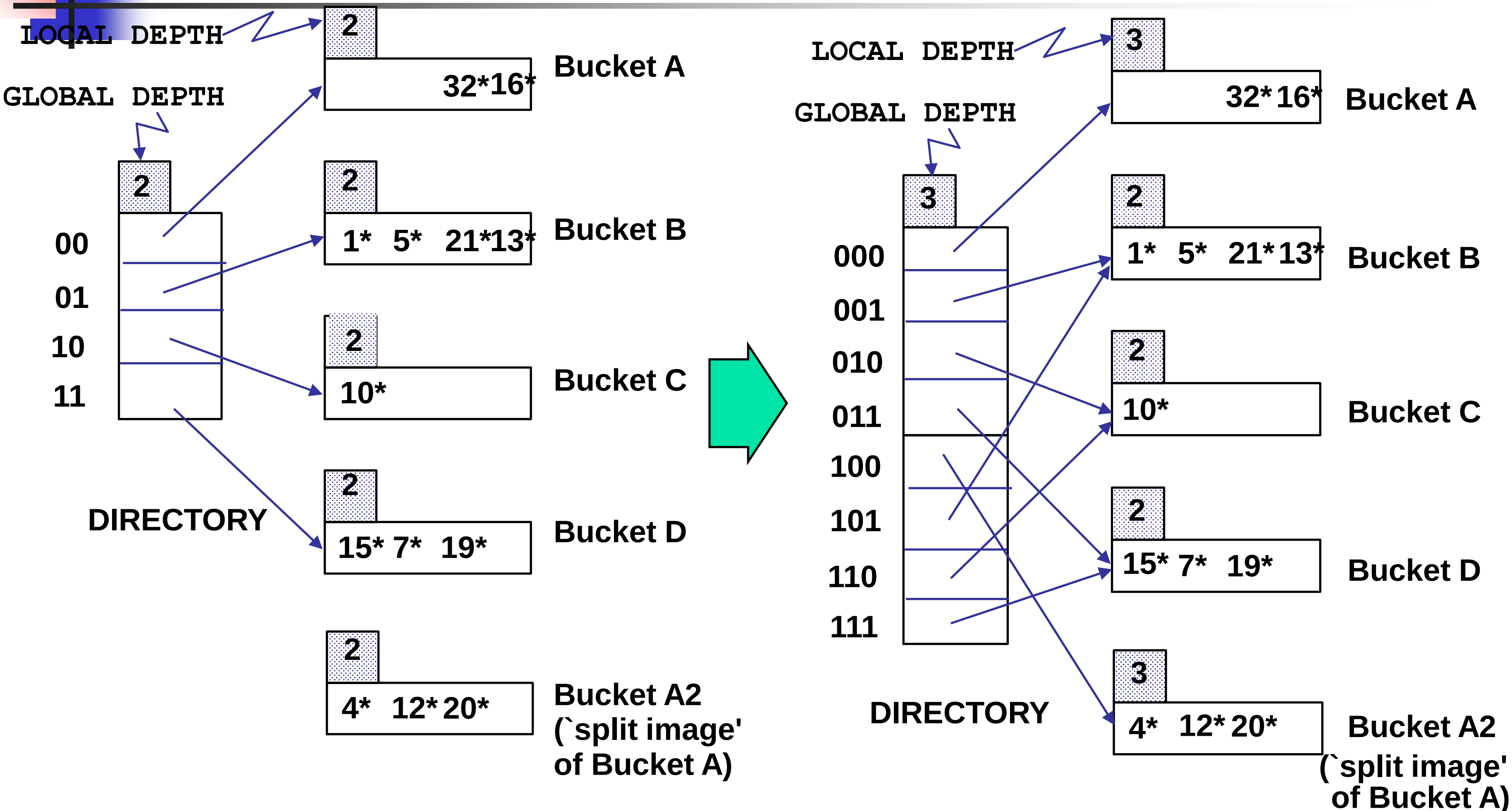
Example

- Directory is array of size 4.
- To find bucket for *r*, take last \`*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.



- **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).
- *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Insert **h**(r)=20 (Causes Doubling)



Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **<u>3</u>** 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.

- Not all splits double the directory size
  - Example: Insert 9*

Points to Note (contd.)

- When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

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!

Comments on Extendible Hashing (contd.)

- **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.

Summary

- File Organizations
- Indexes
  - B+ Tree
  - Hashing (Extendible)