COMP322003 Introduction to Database (Fall 2018)

KNU CSE
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DATA KNOWLEDGE ENGINEERING LABORATORY

HASHING AND INDEXING

Chapters 16.8-17

- Hashing, Tree-Structured Indexes

Prof. Young-Kyoon Suh

Chapter Outline

- Hashing Techniques
  - Static hashing
    - Internal hashing and external hashing
  - Dynamic hashing
    - Extendible hashing, dynamic hashing, linear hashing

- Indexing structures
  - Single-level ordered indexes
    - Primary index, clustering index, and secondary index
  - Multilevel indexes
    - ISAM: static index
  - Dynamic multilevel indexes
    - Search tree
    - Tree-based indexes
      - B+-tree: dynamic index

Introduction

- A ***hash*** **(*direct*)** ***file:*** another type of primary file organization
  - Based on ***hashing***, applying a randomized (**hash**) function, say, h to field value of a record
    - Then the h yields address of the disk block of stored record
    - The disk block is called a *bucket* —a unit of storage for containing one or more records. (In other words, h produces a hash bucket address.)
      - $B = h(K)$, where B: the set of all the bucket addresses, K: all the hash key value.
  - Provides very fast access to records under certain search condition
    - Search condition is "equality condition" on the hash field, or *hash key*.
  - Once the desired block is found by the hash key value, a search for the record within the block can be carried out in the buffer.
- Hashing
  - Also used as an internal search structure within a program whenever a group of records is accessed exclusively by using the value of one field.

static hashing *with a fixed number of buckets, say M, allocated*

Internal Hashing (in Memory)

- For internal files, hashing is typically implemented as a **hash table** via an array of records.

[Index]	Name	Ssn	Job	Salary
0				
1				
2				
3				
	⋮			
$M-2$				
$M-1$				

Array

- Suppose that we have M slots whose addresses correspond to the indexes.
- One possible hash function: $h(K) = K \textbf{ mod } M$, *where K: an integer hash field value*

BTW, easy to encounter the *collision* problem.

Internal Hashing

Collison

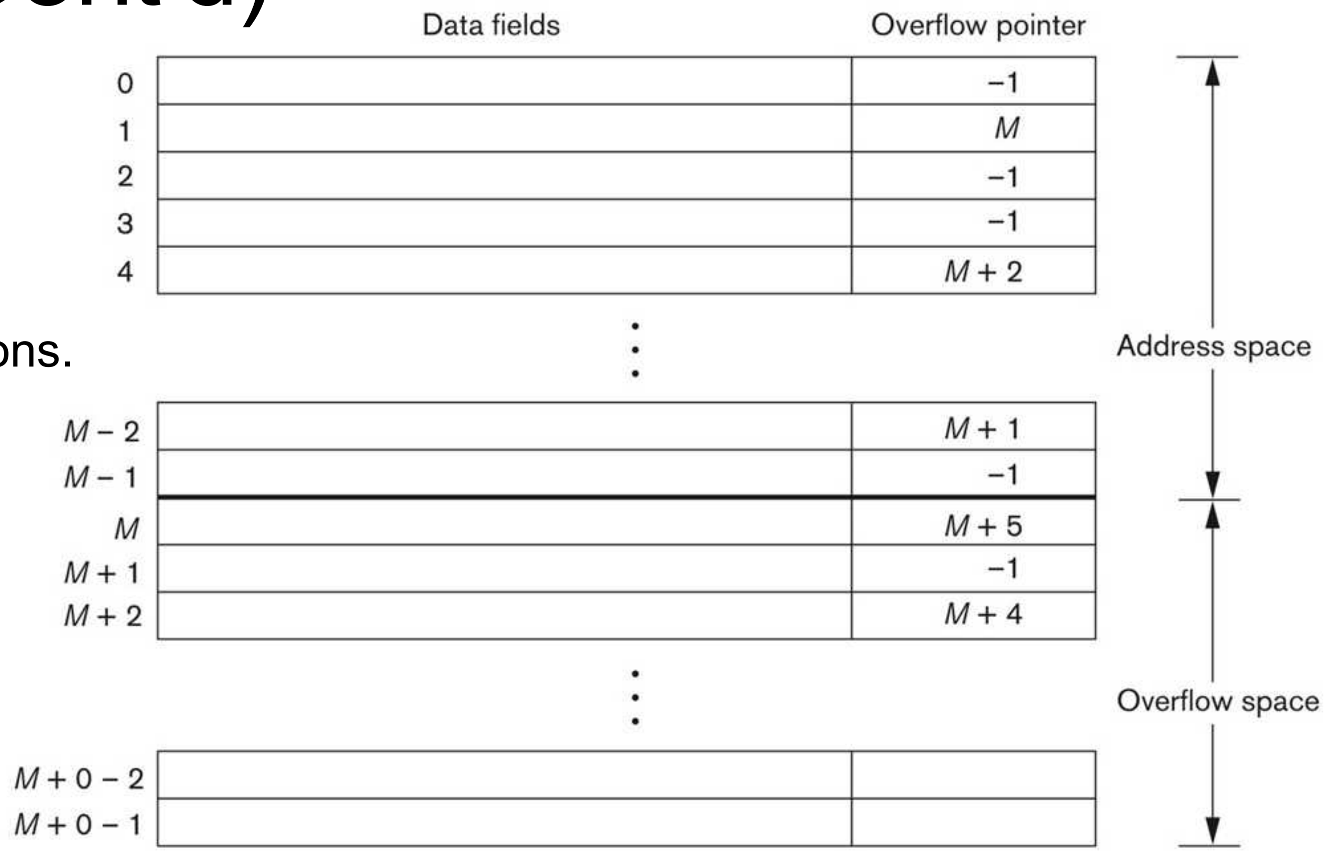
- Occurs when the hash field value of a record that is being inserted *hashes to an address* that "already" contains a different record.
- How to resolve?
  - *Open addressing*
    - Checks the subsequent positions until an empty position is found

```
i <- hash_address(K); a <- i;
if (location i is occupied){
   i <- (i +1) mod M;
   while((i <> a) && (location i occupied)){
         i <- (i + 1) mod M;
         if(i == a) { all positions are full.}
         else  new_hash_address <-  i;
   }
}
```

Internal Hashing

Collison (Cont'd)

- *Chaining*
  - Extending the array with a number of overflow positions.
  - Pointer field added to each record location



• null pointer = −1
• overflow pointer refers to position of next record in linked list

- *Multiple hashing*
  - Uses a second hashing function if the first results in a collision.

Internal Hashing

The Goal of a Good Hashing Function

1) To distribute the records uniformly over the address space in order to minimize collisions

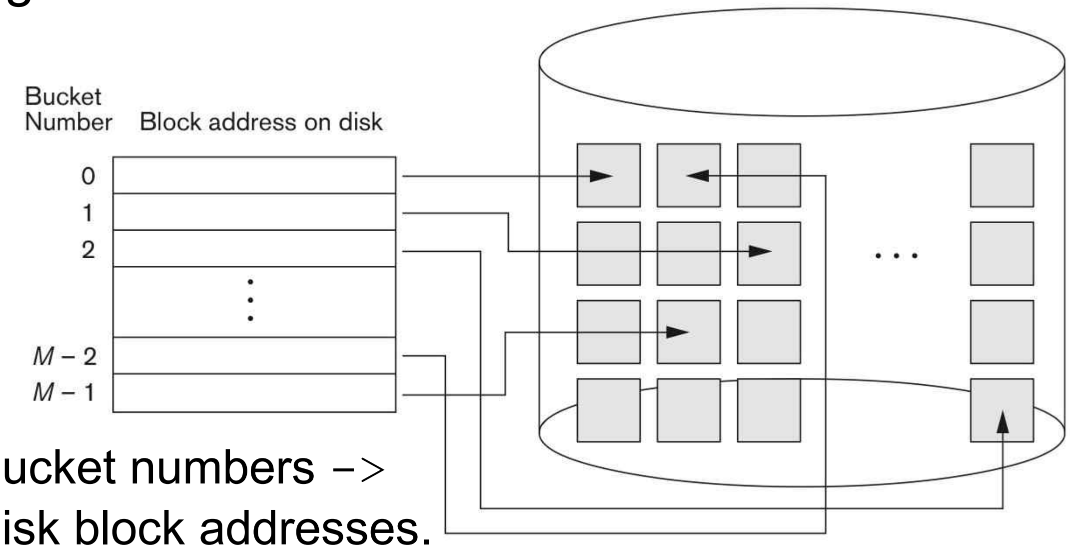
- Making it possible to locate a record with a given key just once

2) To achieve the above yet occupy the buckets fully, thus not leaving many empty (unused) location

- “best” to keep a hash file between 70 and 90% full so that:
  - The number of collisions is low, and we don’t waste too much space

External Hashing for Disk Files

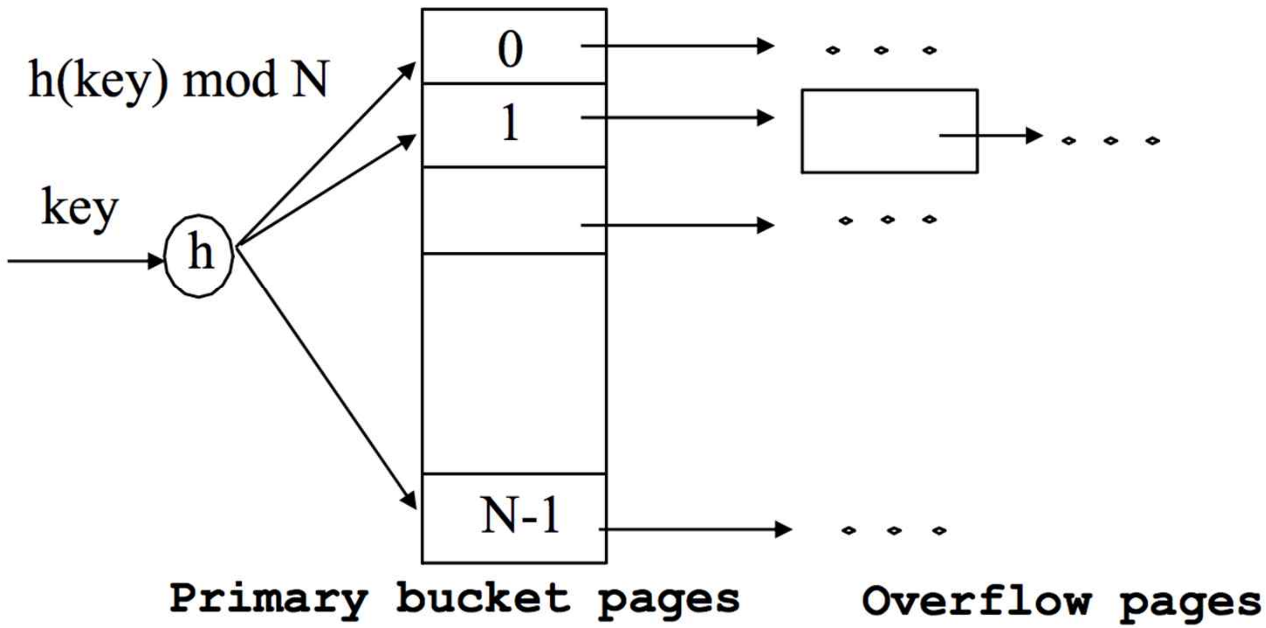
- ***External hashing***: hashing for disk files
  - The target address space: made of **buckets**
  - Each bucket holds multiple records.
    - Bucket: one disk block or a cluster of contiguous disk blocks.
  - The hashing function maps a key into a relative bucket number.
  - A *table* maintained in the file header converts the bucket number into the corresponding disk block address.



bucket numbers -> disk block addresses.

Another Look on External (Static) Hashing

- # primary (bucket) pages: N
  - Fixed, allocated sequentially, never de-allocated
  - Overflow pages if needed.
- $h(k)$ $\mathtt{mod}$ N = bucket to which data entry with key k belongs.

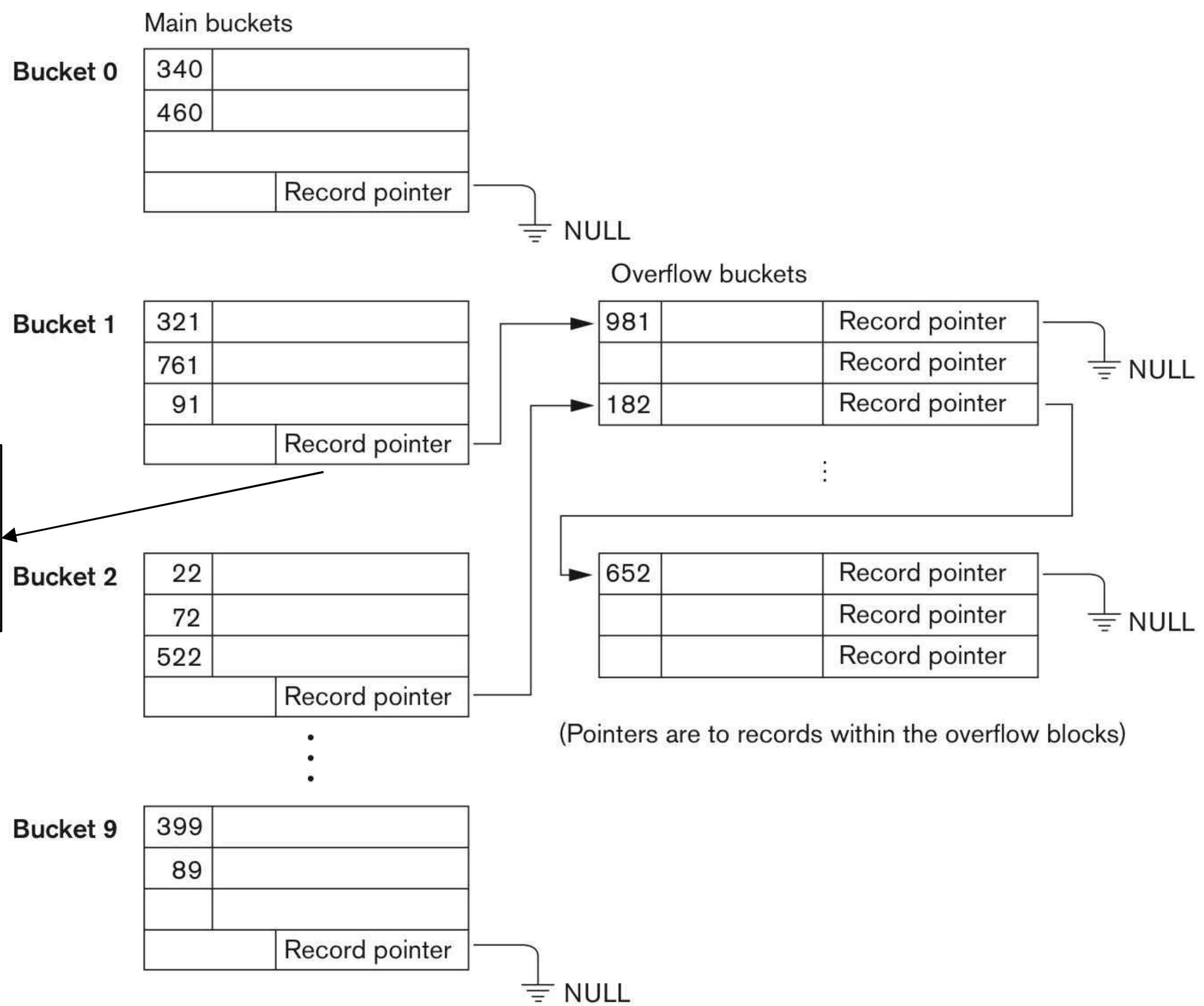


External Hashing for Disk Files

Handling Overflow for Buckets

- By chaining

block (overflow bucket) address + relative record position

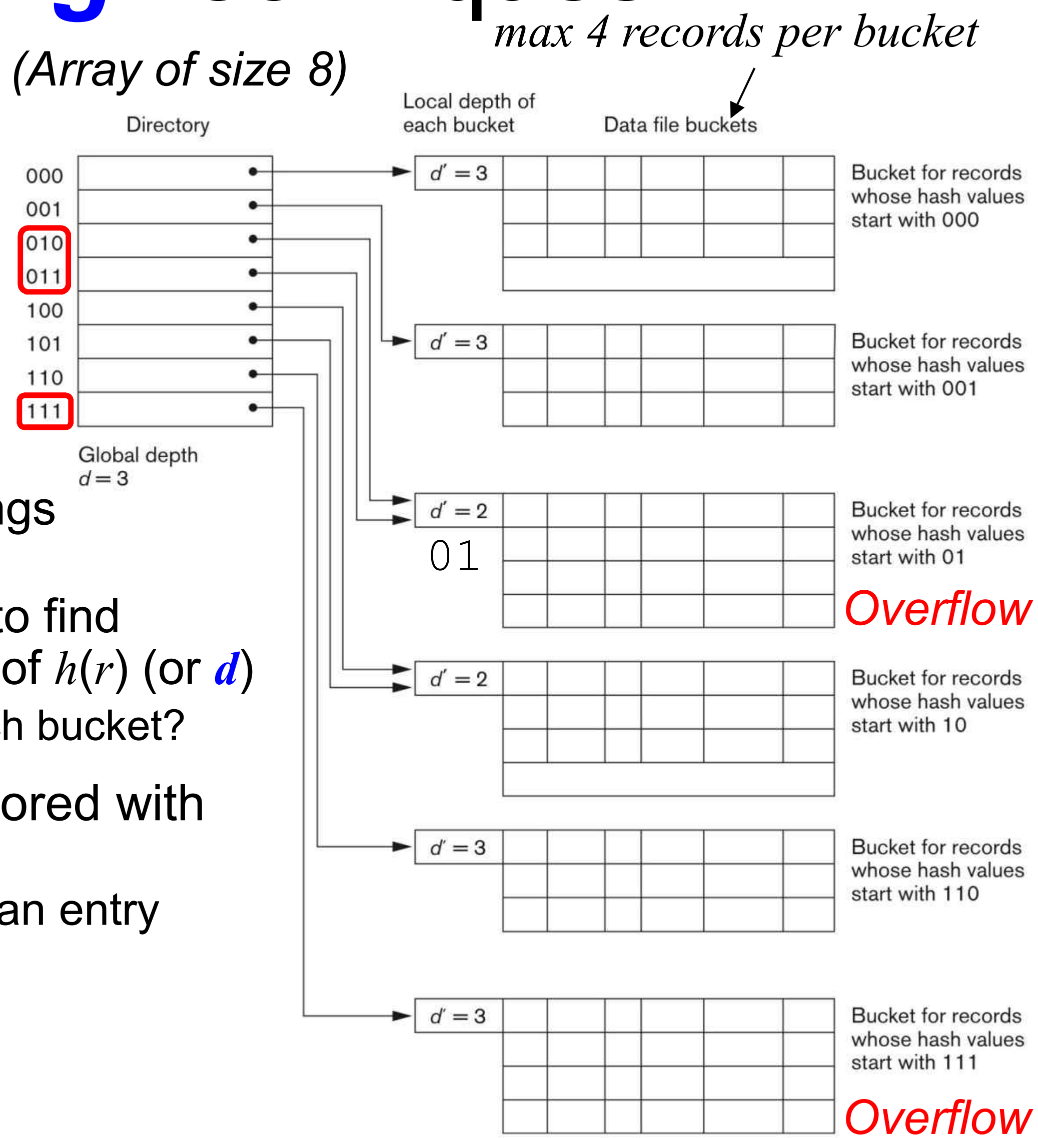


Drawbacks of ***Static Hashing***

- So far, we've discussed static hashing techniques.
- They are superfast, if a search field concerns a hash key field; key-to-address mapping by the hash function.
- But it is *expensive* when searching for a record with a value of some field other than hash field. Why? Linear search …
- A more serious drawback: ***fixed*** hash address space
  - It is difficult to expand or shrink the file "dynamically".
- Any solution?
  - Let's the fixed hash address space to make it *dynamically organized*!

Dynamic Hashing Techniques

- ***Extendible hashing***
  - Uses a directory (an array of 2^d bucket addresses.)
  - d: *global depth of the directory*; used as an *index*
    - Max # of bits needed to tell which bucket an entry belongs to.
    - For a hash key field, say r, to find bucket for r, take *last # bits* of $h(r)$ (or d)
      - If $h(r)$ = 5 = '0...**101**', which bucket?
  - d': *local depth of a bucket*; stored with each bucket
    - # of bits used to determine if an entry belongs to this bucket

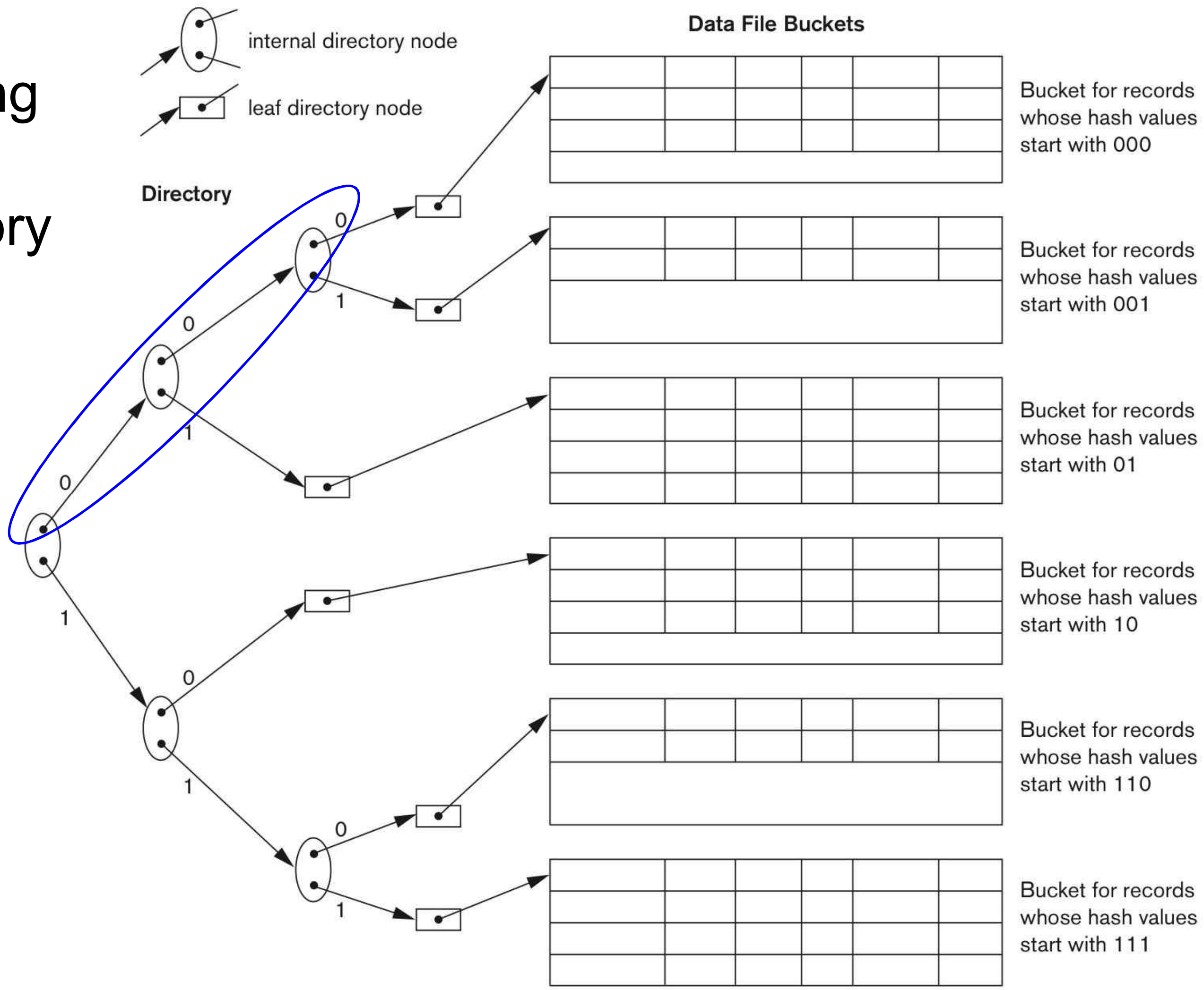


Dynamic Hashing Techniques: Extendible Hashing

- Doubling occurs if a bucket overflows.
  - When the local depth (d') of the bucket is equal to the global depth (d)
- Reducing by half occurs if $d > d'$ for "**all**" the buckets
  - After some deletions occur
- Advantages
  - File performance doesn't degrade as file grows.
    - Static external hashing: collisions increase, and the respective chaining effectively increases the average number of accesses per key.
  - No space is allocated for future growth.
    - But additional buckets can be allocated dynamically as needed.
    - Space for the directory table: $O(2^k)$, where k = # of bits in the hash value.
  - Splitting causes *minor* reorganization in most cases.
    - Since only the records in one bucket are redistributed to 2 new buckets.
- Disadvantage: "2" block accesses

Dynamic Hashing Techniques: *Dynamic hashing*

- Precursor to extendible hashing
- Maintains tree-structured directory with
  - *Internal node*: has two pointers with 0 or 1 bit, each
  - *Leaf node*: holds a pointer to the actual bucket with records



Dynamic Hashing Techniques: *Linear hashing*

- An alternative to extendible hashing
  - Idea: allowing a hash file to expand and shrink buckets *without* the need of a directory
    - Use a family of hash functions h_0, h_1, h_2, ...
  - Starts with h_0 (= K mod M).
  - Later, if collision happens in *any* bucket, then bucket 0 is split into two buckets: 0 (itself) and M (a new bucket) via h_1(= K mod $2M$).
    - If another collision, 1 and M+1 via h_2, etc.
  - By doing so, it handles the problem of long overflow chains without using the directory by splitting buckets in *round-robin*.

PHYSICAL DATABASE DESIGN

Chapter 17

- Indexing Structures for Files

(Assume that a file already exists with some primary organization.)

Introduction

- Why *indexes*(색인) (extra auxiliary access structures)?
  - Used to speed up record retrieval in response to certain search conditions.
    - E.g., See the **Index** Section on page 1245 in the textbook. Why there?
- Index structures are "additional files on disk" that provide *secondary access paths*,
  - Providing alternate ways to access the records *without* affecting the physical placement of records in the primary data file on disk.
  - The index structures enable efficient access to records based on the **indexing fields** used to construct the index.

Introduction (Cont'd)

- Note that *any* field can be used to create an index.
  - *Multiple indexes* on different fields can be constructed.
  - *An (multi-level) index on multiple fields* can be constructed.
    - Both types of indexes on the same file
- To find a record(s) in the data file based on a search condition on an indexing field,
  - The index file is searched, guiding you to **pointers** to one or more disk blocks in the data file where the required records are located.
- To organize the index,
  - Most indexes are based on ordered files (*single-level indexes*) and use tree data structures (*multi-level indexes*, for instance, **B^+-trees**)

TYPES OF SINGLE-LEVEL ORDERED INDEXES

Chapter 17.1

Ordered Index (정렬 색인)

- Idea: similar to the idea behind the index used in a textbook.
  - Lists "important terms" (c.f., *index keys*) in alphabetical order along with a list of page numbers (c.f., *block addresses*) where the term appears on the book.
- *Indexing field* (attribute)
  - A single field of a file, on which an index structure is usually defined.
  - Each value of the index field is stored into the index file with a list of pointers to all disk blocks containing records with that field value.
- Values in index are "ordered," or "sorted"
  - Why? To do a binary search (rather than linear search) on the index.
- The index file is *much smaller* than the data file. Why?

Several Types of Ordered Index

- ***Primary index***
  - Specified on the ***ordering key*** field of an **ordered file** of records.
- ***Clustering index***
  - Specified on a ***non-key ordering*** field (having the same value for the field in numerous records)
    - The data file is called a **clustered file**.
  - Note that a file can have <u>**at most one** physical ordering field</u>, so it can have **at most** one primary index or one clustering index, *but not both*.
- ***Secondary index***
  - Specified on any ***non-ordering*** field of file.
  - A data file can have several secondary indexes.

Primary Indexes (주요 색인)

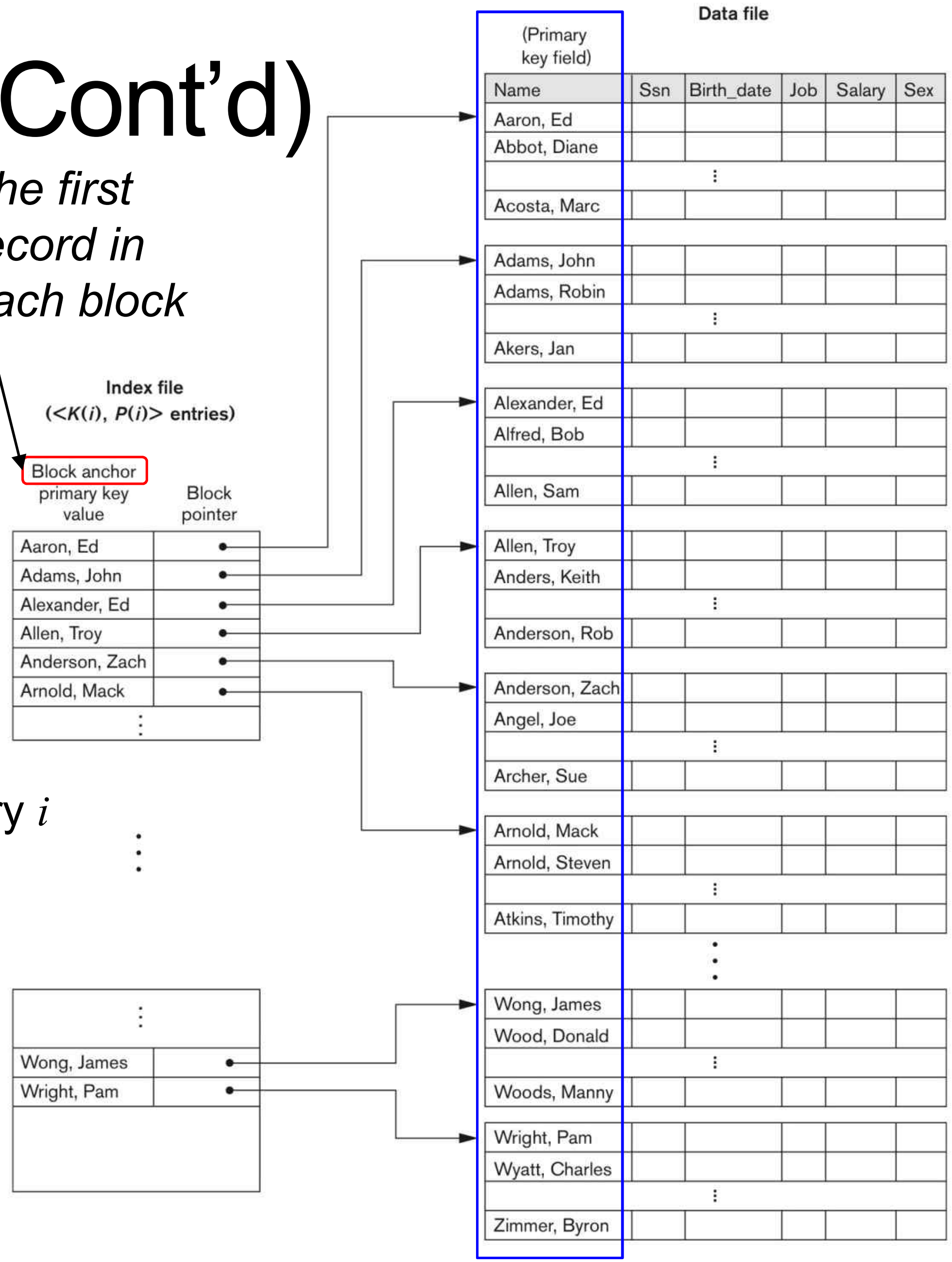
- ***Ordered file*** with index records (entries) consisting of two fields:
  1) *Primary key*: $K(i)$
     - *The value of the primary key field* for the first record in a block
  2) *Pointer to a disk block*: $P(i)$, where i indicates a certain index entry.
     - Here, the disk block is the one having the key value.

Primary Indexes (Cont'd)

- Data file: Figure 16.7
  - Discussed last class
  - Ordering key field: `Name`
    - Primary key: assume that names are *unique*.
- Index file (primary index)
  - Each entry in the index has:

    $<K(i), P(i)>$, where
    - $K(i)$: a `Name` value at index entry i
    - $P(i)$: a pointer at index entry i
      - $<K(1)$ = (`Aaron, Ed`), $P(1)$ = *address of block* $1>$
      - $<K(2)$ = (`Adams, John`), $P(2)$ = *address of block* $2>$
      - $<K(3)$ = (`Alexander, Ed`), $P(3)$ = *address of block* $3>$

The first record in each block



Characteristics of Indexes

- Indexes may be *dense* or *sparse*.
  - ***Dense index*** has an index entry for *every search key value* (so every record) in the data file.
  - ***Sparse index*** has entries for *only some of the search values*.
    - A.k.a. ***nondense index***
    - **Fewer** entries than the number of records in the file.

- Q: Which type of primary index is the one on the previous slide?
  - If so, is a primary index *dense* or ***sparse***?

Characteristics of Primary Indexes (Cont'd)

- The index file for a primary index occupies a much smaller space than does the data file. Why? **Two** reasons.

  1) There are *fewer index entries* than there are records in the data file.

  2) Each index entry is *typically smaller in size* than a data record because it has only two fields, or (Key, $Block\ Address$).

  - Hence, a binary search on the index file requires *fewer block accesses* than a binary search on the data file.

Calculating # of Block Accesses with *No Index* File

- Ex 1) An ordered file with r = 300,000 records stored on a disk with block size B = 4 KBytes.
  - File records are of fixed size and unspanned, with record length R = 100 bytes.
  - bfr: $ceiling(B / R)$ = ? records per block
  - # blocks needed for the file: $ceiling(r / bfr)$ = ?
  - How many binary searches on the data file?

Calculating # of Block Accesses via a Primary Index File

- Given b blocks in the **primary index** file,
  - # block accesses via an index file to locate a record = $\log_2 b + 1$
  - Why $\log_2 b$? Why 1? Try a binary search on a record with a `Name` value of "`Akers, Jan`" on Slide 23.

Calculating # of Block Accesses via a *Primary Index* File (Cont'd)

- Ex 1') An ordered file with r = 300,000 records stored on a disk with block size B = 4 KBytes.
  - The ordering key field of the file (V): 9 bytes long.
  - A block pointer (P): 6 bytes long.
  - Size of each index entry (R_i): (9 + 6) = 15 bytes.
  - bfr for the index: $ceiling(B / R_i)$: ? entries (records) per block
  - # *index entries* (= # of data records) = ? entries
    - How big is the index file then?
  - # *index blocks* = *? entries* / bfr = ? blocks
  - # *binary searches* on the index file: ? block accesses
  - # block accesses to locate a data record via the index file = ?
  - How much improvement over the *non-index* approach?

Problems with a Primary Index?

- Major problem: *insertion* and *deletion* of records
  - If we attempt to insert a record in its “correct” position in the data file,
    - Not only *move records to make space for the new record*
    - But also *change some index entries*
    - Moving records will change the block anchors of some blocks.
- How to solve the insertion?
  - Use *unordered overflow file*
  - Use a *linked list of overflow records for each block* in the data file
    - Similar to dealing with overflow records in the hashing technique
    - To improve retrieval time, sort the records within each block and its overflow linked list.
- How to solve the deletion?
  - Use *deletion markers*

Clustering Indexes (군집 색인)

- If file records are *physically ordered* on a “non-key” field without a distinct value for each record,
  - ***Clustering field***: such a non-key field.
  - ***Clustered file***: such data file having the clustering field
- ***Clustering index***: another type of *nondense* index; Why?
  - Created and used to speed up retrieval of all the records having the same value for the <u>clustering field</u>.
    - What’s the difference from a primary index?
  - *Ordered file with two fields*, consisting of:

    1) *The field of the same type* as the clustering field of the clustered file

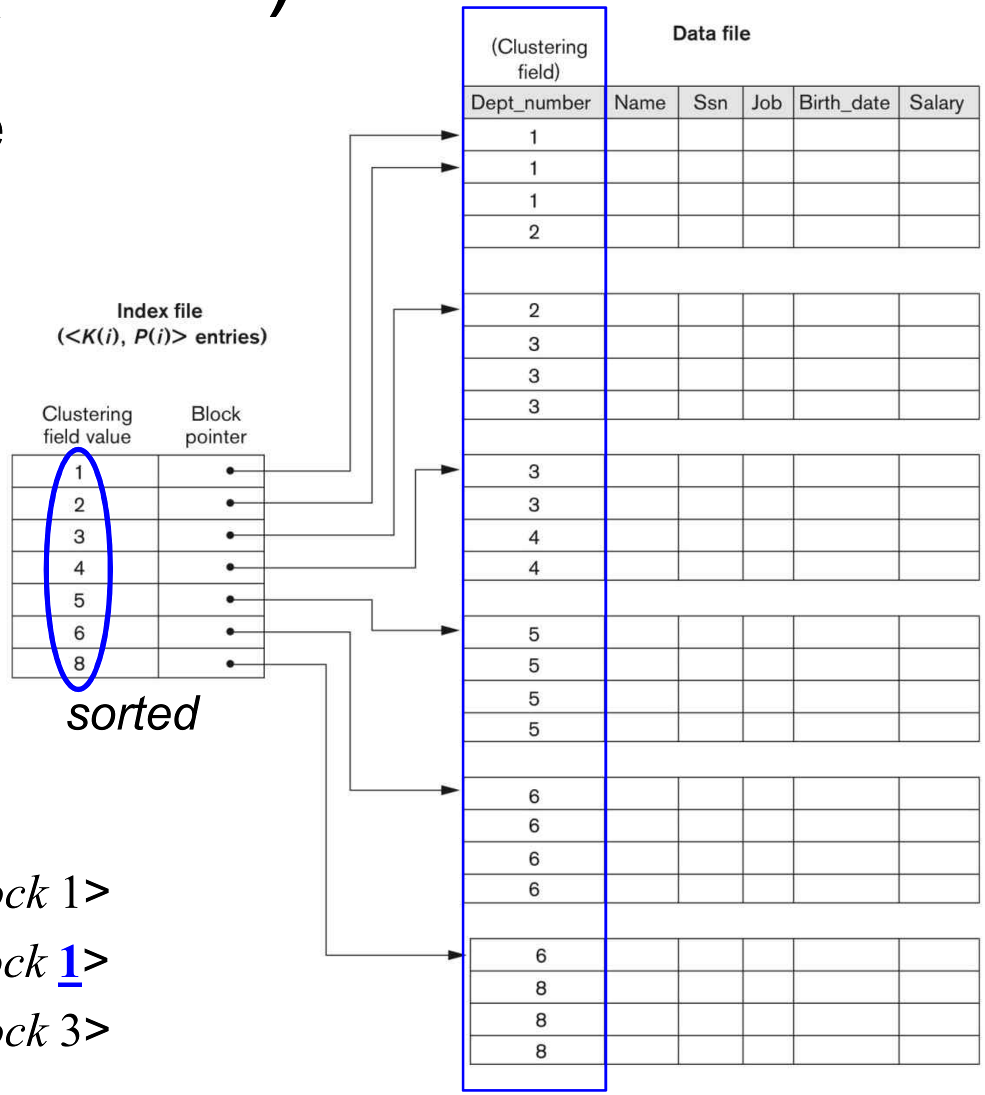
    2) *A disk block pointer*
    - An index entry: <(each distinct) *value* ***v***,

      a pointer to the ***first block*** in the data file having ***v***>

Clustering Indexes (Cont'd)

- Data file: an `EMPLOYEE` file
  - `Dept_number`: Ordering nonkey field (clustering field)
- Index file (clustering index)
  - Each entry in the index has:

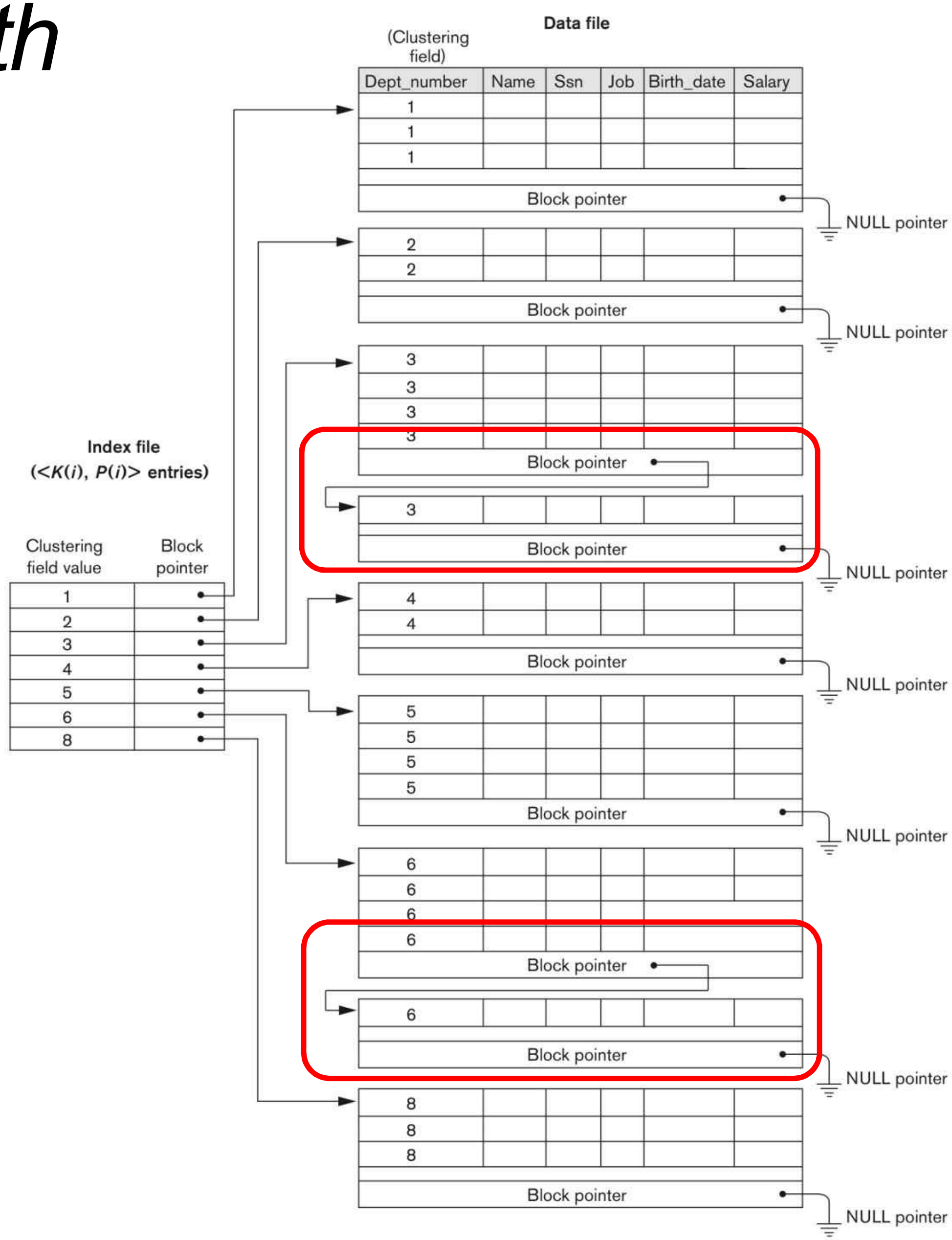
    <$K(i)$, $P(i)$>, where
    - $K(i)$: a clustering field value at index entry i
    - $P(i)$: a pointer at index entry i
  - Example:
    - <$K(1)$ = `1`, $P(1)$ = *address of block* 1>
    - <$K(2)$ = `2`, $P(2)$ = *address of block* **1**>
    - <$K(3)$ = `3`, $P(3)$ = *address of block* 3>



Problems with a Clustering Index?

- Still major problem: *insertion* and *deletion* of records
  - Why? *Physically ordering*….
- How to solve the insertion/deletion?
  - To *reserve a whole block* (a cluster of continuous blocks) for each (distinct) value of the clustering field
    - See the next slide.
  - By doing so, all records with that distinct value are placed in the (same) block (or block cluster).
  - This makes insertion/deletion relatively straightforward.

Clustering Index *with a Separate Block Cluster*



Calculating # of Block Accesses via a *Clustering Index* File

- Ex 1”) An ordered file with r = 300,000 records stored on a disk with block size B = 4 KBytes.
  - Imagine that it is ordered by `Zipcode` (as clustering field).
    - There are 1,000 distinct zip codes in the file: or, an average of 300 records per zip code, assuming the codes are evenly distributed.
    - # index entries (r_i): ?
  - The size of an i-th index record: 5-byte Zipcode + 6-byte block ptr
    - Record length (R_i): 11 bytes
  - bfr_i : $ceiling(B / R_i)$ = ? records per block
  - *# index blocks* = ceiling(r_i / bfr_i) = ? blocks
  - *# binary searches* on the clustering index file: ? block accesses
  - # block accesses to locate a data record via the clustering index file = ? block accesses

Secondary Indexes (보조 색인)

- Provide secondary means of accessing a data file
  - Some primary access exists.
  - Ordering data records doesn't matter: could be hashed.
- May be created on:
  - A key field with a unique value in every record, and
  - A nonkey field with duplicate values.
- *Ordered file with two fields*:
  - *Indexing field*, $K(i)$: the same type as some non-ordering field of the data file
  - *Record* (why?) or *block* (why?) *pointer*, $P(i)$: the pointer to the record or block containing the indexing field value
- *Many* sec. indexes can be created for the same data file.
  - Each represents an additional means of accessing that file based on some specific field.

Secondary Index on a Key Field

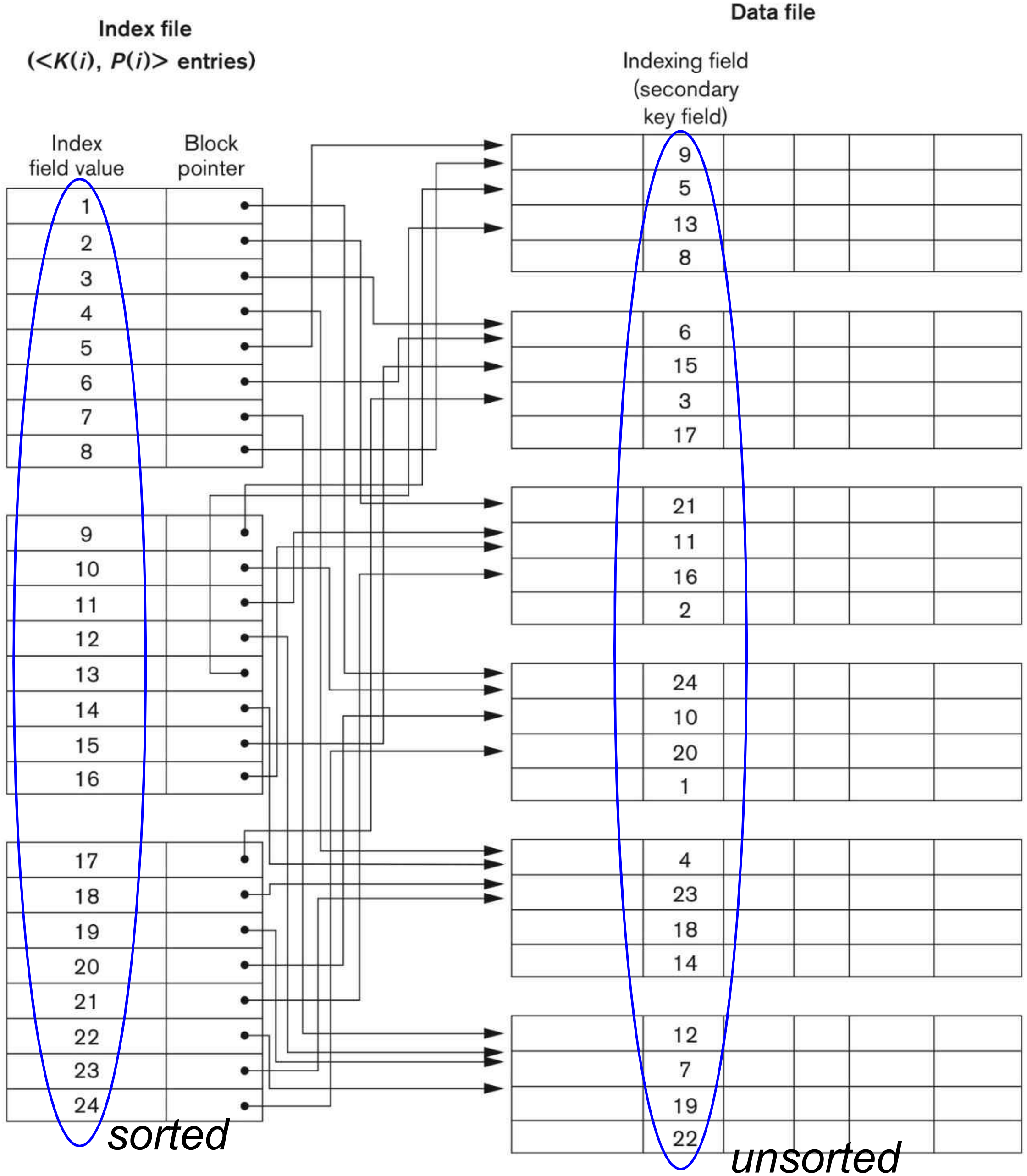
- The key field: has a *unique* value for *every* record.
  - So called a **secondary** key.
    - Corresponding to any `UNIQUE` key field (or perhaps primary key attribute)
  - In this case, there's one index entry for *each record* in the data file.
    - Each index entry has two components:

      <*value of the field for the record*,

      a pointer to the block containing the record or to the record itself>
    - Then, is this type of secondary index ***dense*** or *nondense*?

Secondary Indexes (Cont'd)

- A ***dense secondary index*** on a nonordering key field of a data file
  - With *block* pointers



Calculating # of Block Accesses via a *Secondary Index* File

- Ex 1”) An ordered file with r = 300,000 records with fixed length records of size R = 100 bytes, stored on a disk with block size B = 4 KBytes.
    - The file has b = 7,500 blocks (as calculated before).
  - Say, we want to find a record with a specific value for the secondary key, or a nonordering *key* field of the file
    - The secondary key field: 9 bytes long, a block pointer: 6 bytes.

(Note that if it were not for the secondary index, it would be a linear search, on average requiring b / 2 = 7,500 / 2 = 3,750 block accesses.)

Calculating # of Block Accesses via a *Secondary Index* File (Cont'd)

- Suppose that we construct a secondary index on that nonordering key field of the file.
  - Each index entry size = (9 + 6) = 15 bytes
  - *bfr* = ? entries per block
  - # *dense index* entries = # records in the data file = ? entries
  - # index blocks needed = ? blocks
  - A binary search on this index needs: ? block accesses
  - To search a record using the index: ? block accesses
- Why "more" block accesses than that of the primary index?
  - As the primary index was nondense, so it was shorter: how many blocks for the primary index?
- A secondary index usually need *more storage space* and *longer search time* than primary index.
  - Instead improved search time for arbitrary record

Summary of Index Types

Table 17.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	**Index Field Not Used for Physical Ordering of the File**
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Table 17.2 Properties of Index Types

Type of Index	**Number of (First-Level) Index Entries**	**Dense or Nondense (Sparse)**	**Block Anchoring on the Data File**
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

* Of course, we can also create a secondary index on a nonkey, nonordering field of a data file.

DYNAMIC MULTILEVEL INDEXES

Chapter 17.2

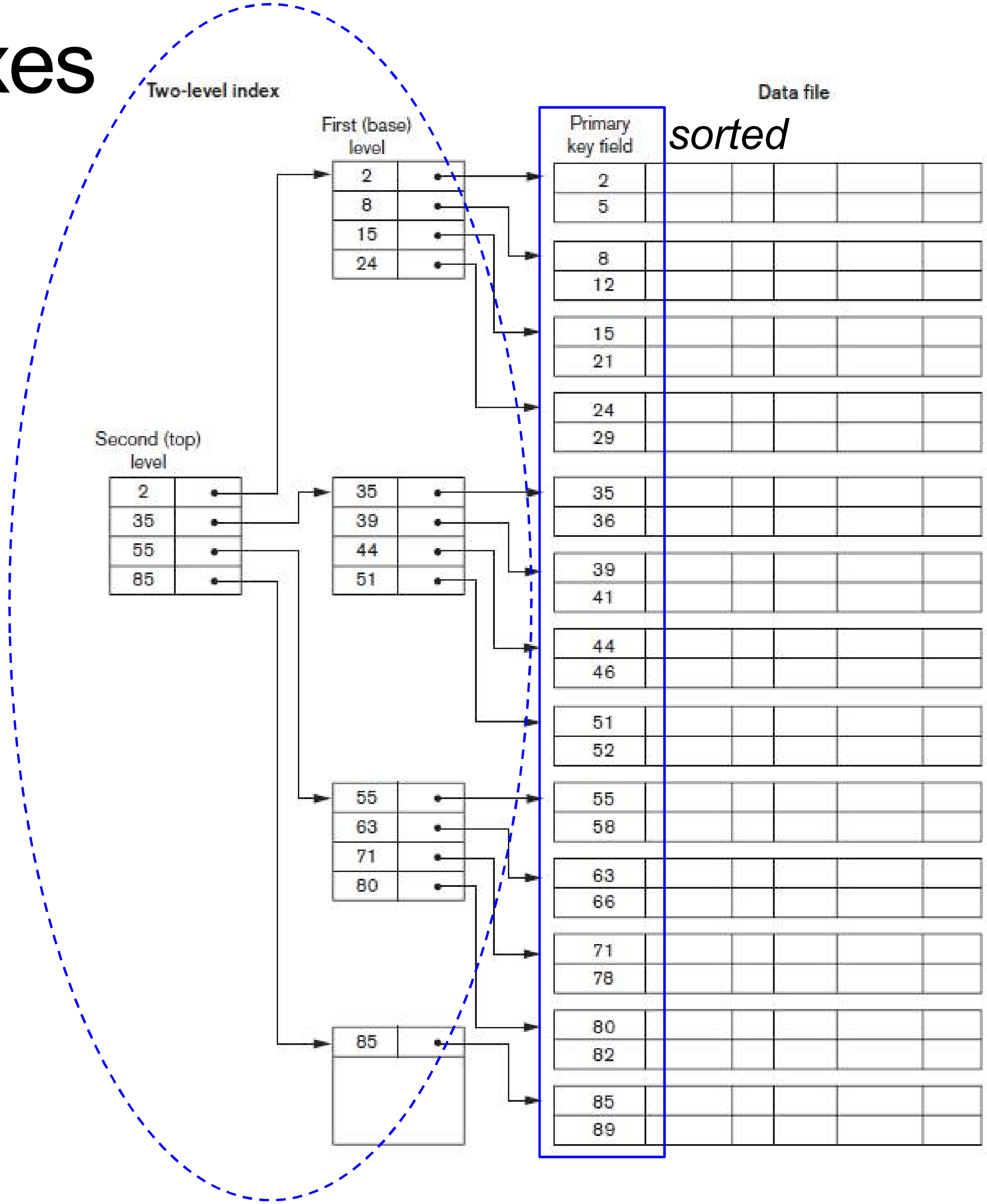
Multilevel Indexes

- Designed to greatly reduce remaining search space as search is conducted
  - By increasing a blocking factor (called *fanout*) in an index block
- The first (base) level of a multilevel index
  - The index file itself
- Second level (intermediate)
  - Primary index to the first level
- Third level (highest at the moment)
  - Primary index to the second level

Multilevel Indexes (Cont'd)

- A two-level primary index
  - Like **ISAM** (Indexed Sequential* Access Method) with ***static structure***
    - For more details, see Appendix.

**Indexed sequential file*: an <u>ordered file</u> with a multilevel primary index on its ordering key field



DYNAMIC MULTILEVEL INDEXES

Chapter 17.3

- Using B-Trees and B^+-Trees

(Also, refer to Chapter 10: Tree-Structured Indexes, Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke)

Why Dynamic?

- ISAM: still faced with the problems dealing with index insertions and deletions

  1) All index levels are a *physically ordered file*, which does not change subsequently the structure under inserts/deletes.

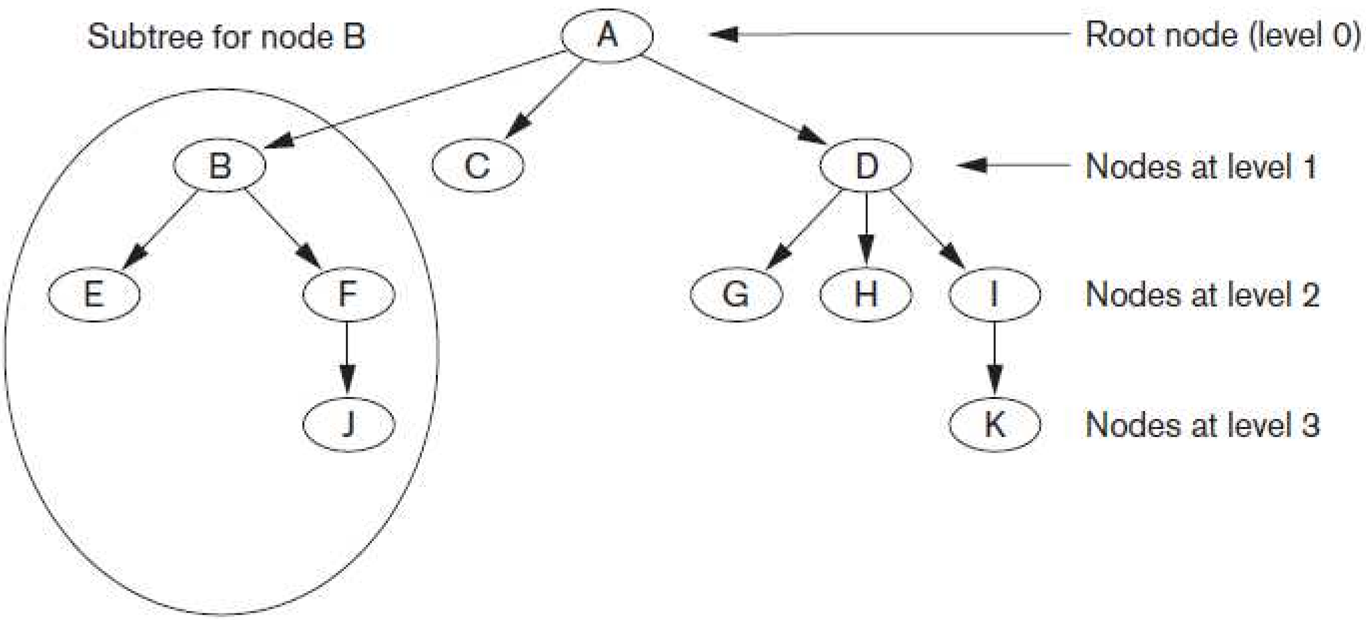
  2) *Poor performance* by overflow pages.

- To solve such inflexibility,
  - Leave some space in each of index blocks for inserting new entries.
  - Uses appropriate insertion/deletion algorithms for creating/deleting new index blocks when the data file grows and shrinks.

- Hence, people invented *dynamic multilevel index*, based on tree data structure.

Review of Tree Data Structure

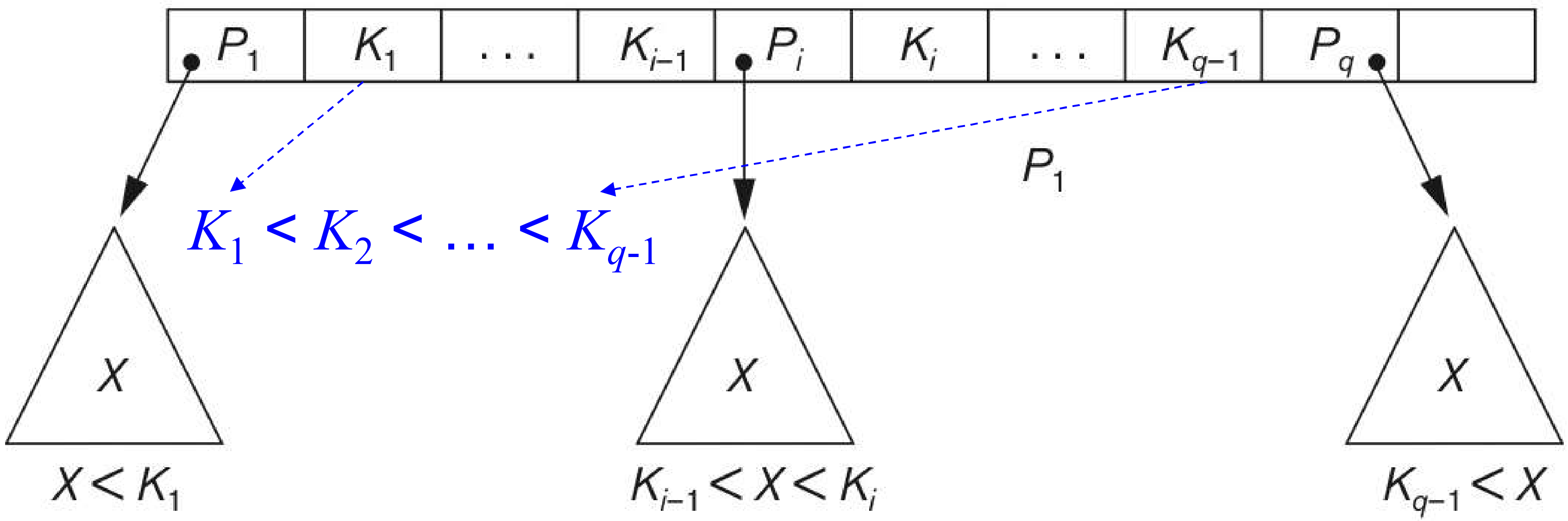
- A tree data structure showing an "unbalanced" tree



(Nodes E, J, C, G, H, and K are leaf nodes of the tree)

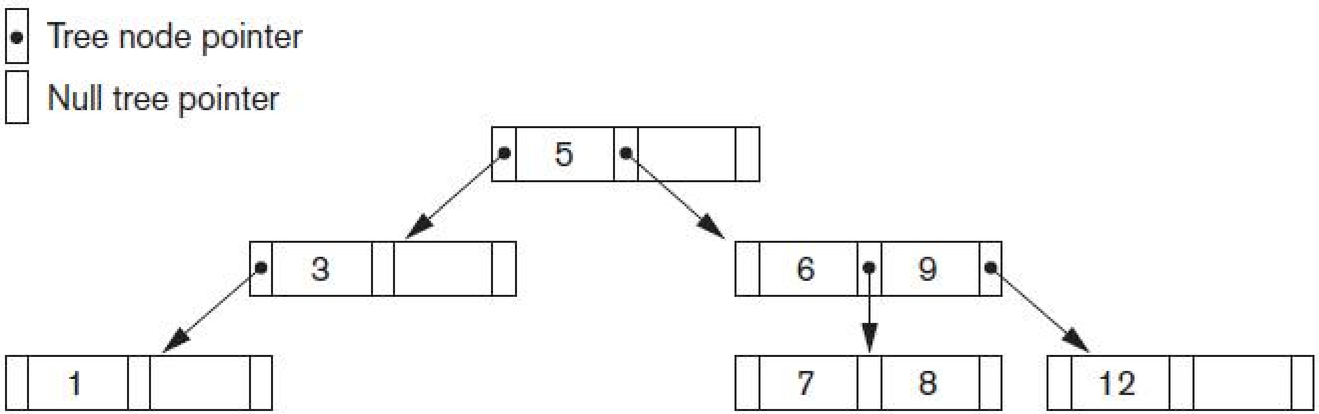
Search Trees (in Textbook)

- A special type of tree
  - Used to guide the search for a record (stored in a disk file), given the value, say X, of one of record's fields
  - The values in the tree: the values of one of the *search fields* of the file.
- A *search tree of fanout* p (in the below example, 3):
  - Each node contains **at most** i) p-1 search values, and ii) p (node) pointers.



Search Trees (Cont'd)

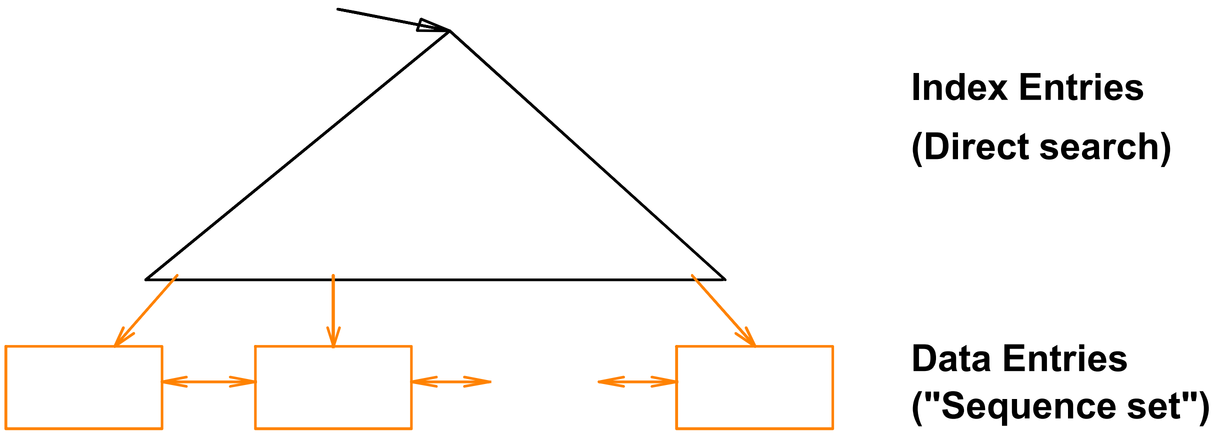
- Example of a search tree of fanout = 3 and integer search values.



- Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the two constraints:
  1) Each tree node is assigned to *a disk block*.
  2) When a new (data) record is inserted in the (data) file, the search tree *must be updated* by inserting a tree entry containing (i) the *search field value* of the new record and (ii) a *pointer* to the new record.

B+-Tree Structures: *Most Widely Used Index*

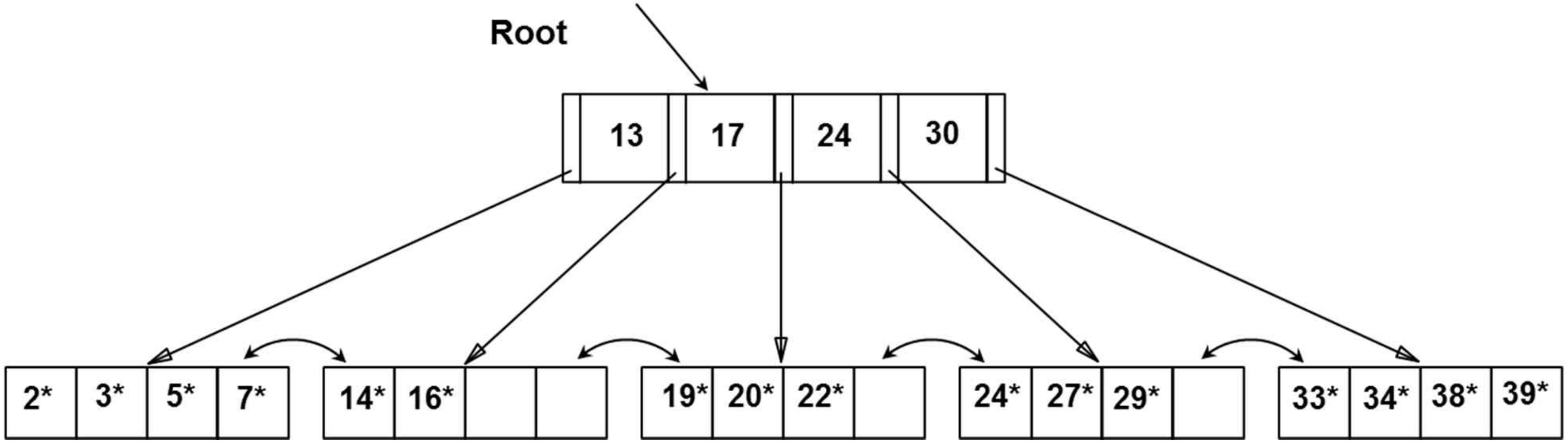
- Insert/delete can be performed at $\log_p N$ cost
  - (p = fanout, N = # leaf pages)
- Keep tree height-balanced.
- Minimum 50% occupancy (except for root).
- Each node contains d <= m <= 2d entries.
  - The parameter d is called the *order* of the tree.
- Supports equality and range-searches efficiently.



B+-Tree Structures (Cont'd)

- Internal nodes: direct the search.
  - Some search field values from the leaf nodes repeated to guide search.
    - No data records are stored in the internal nodes.
- (Only) Leaf nodes: store data pointers
    - C.f., In B-tree, data pointers can appear at any node.
  - Leaf nodes have an entry for every value of the search field.
    - "Linked" to each other (as opposed to those of the B-tree)
  - A data pointer to the record if search field is a key field.
  - A data pointer to a block containing pointers to the data file records if search field is nonkey field.

Example of B+ Tree with Order = 2



- Search begins at root.
  - Key comparisons direct the search to a leaf node (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...
  - Based on the search for 15*, we know it is **NOT** in the tree!!!

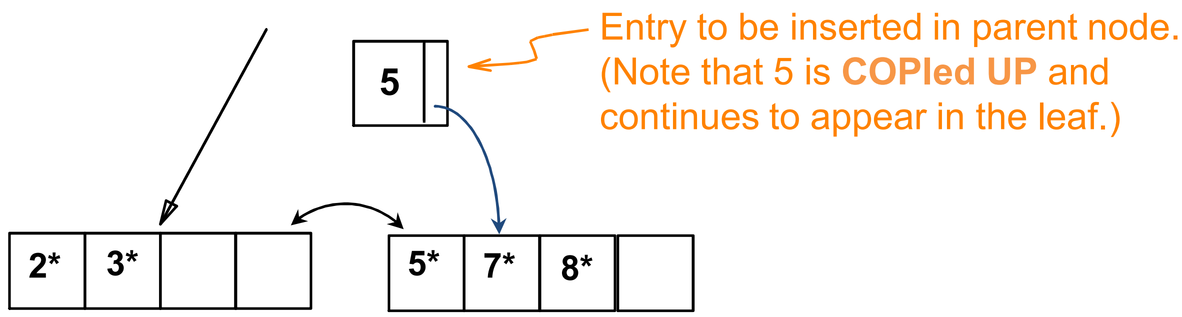
*: key value in leaf

Inserting a Data Entry into a B+ Tree

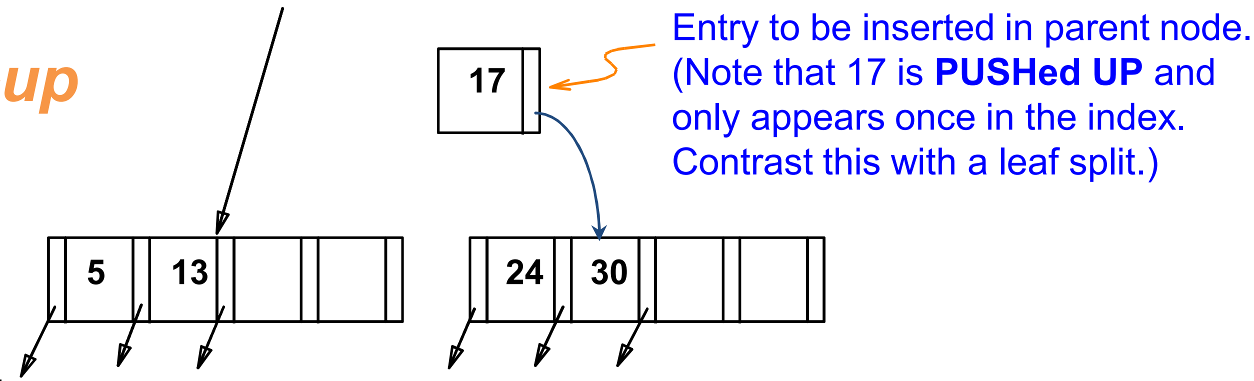
- Find correct **leaf** *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *DONE*!
  - Else, must ***split*** *L (into L and a new node L2)*
    - Redistribute entries evenly, **COPY UP** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively.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

Inserting 8* into Example B+ Tree

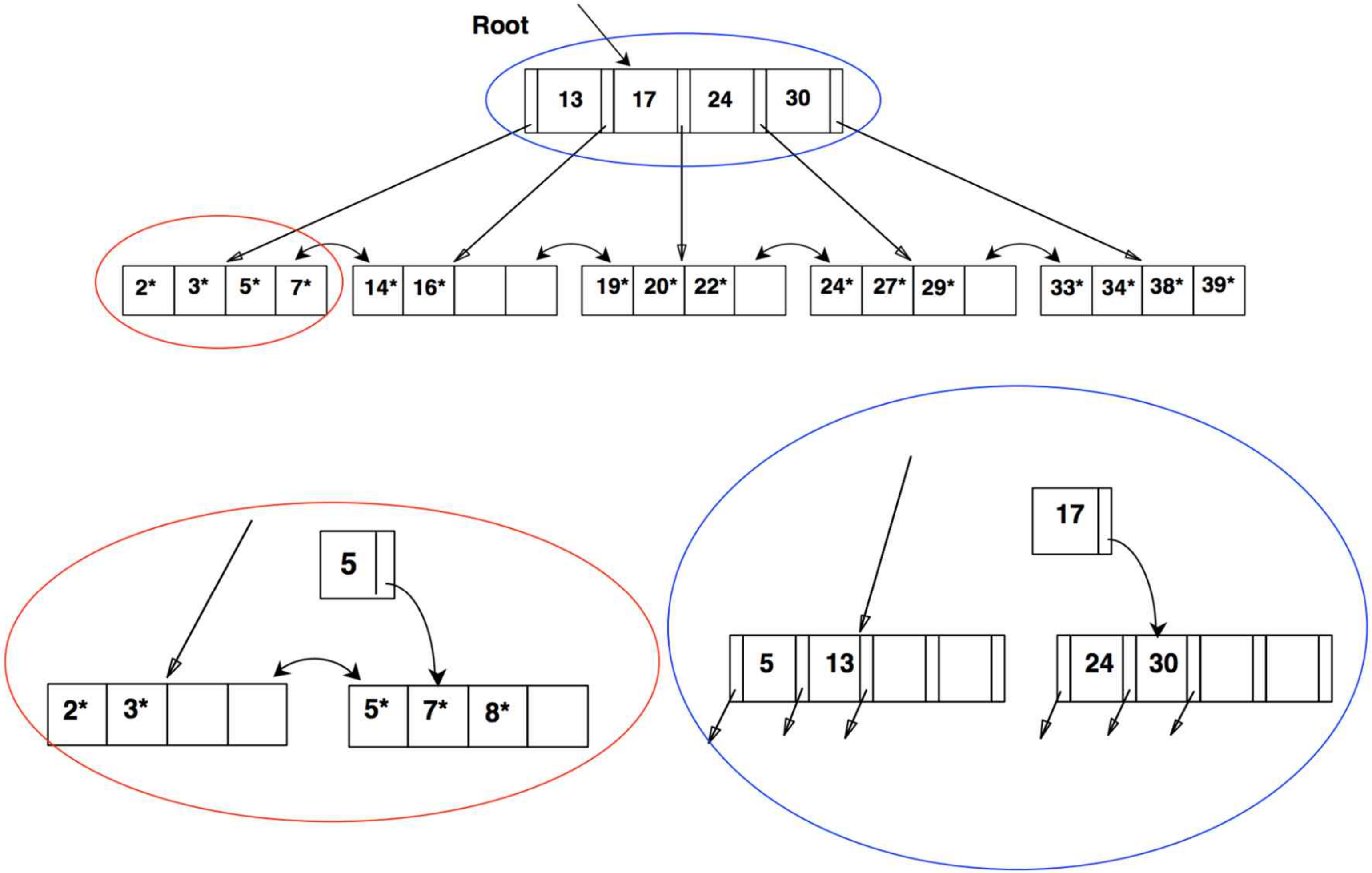
- Observe *how minimum occupancy* is guaranteed in both leaf and index page splits.



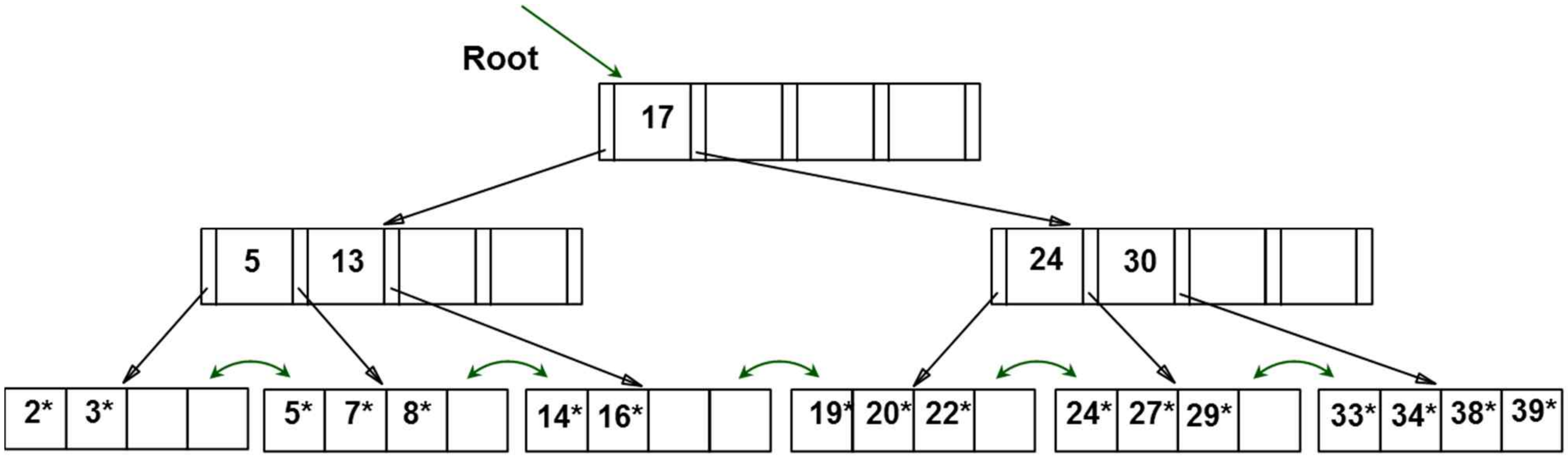
- Note difference between ***copy-up*** and ***push-up***.
  - Be sure you understand the reasons for this.



Inserting 8* into Example B+-Tree



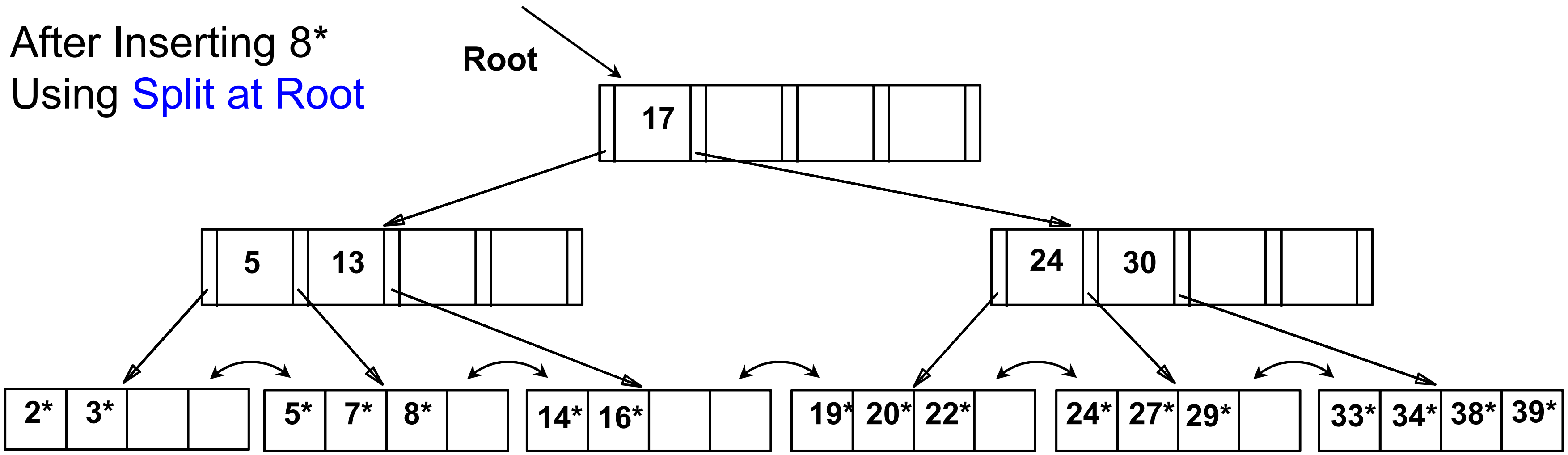
Example B+ Tree After Inserting 8*



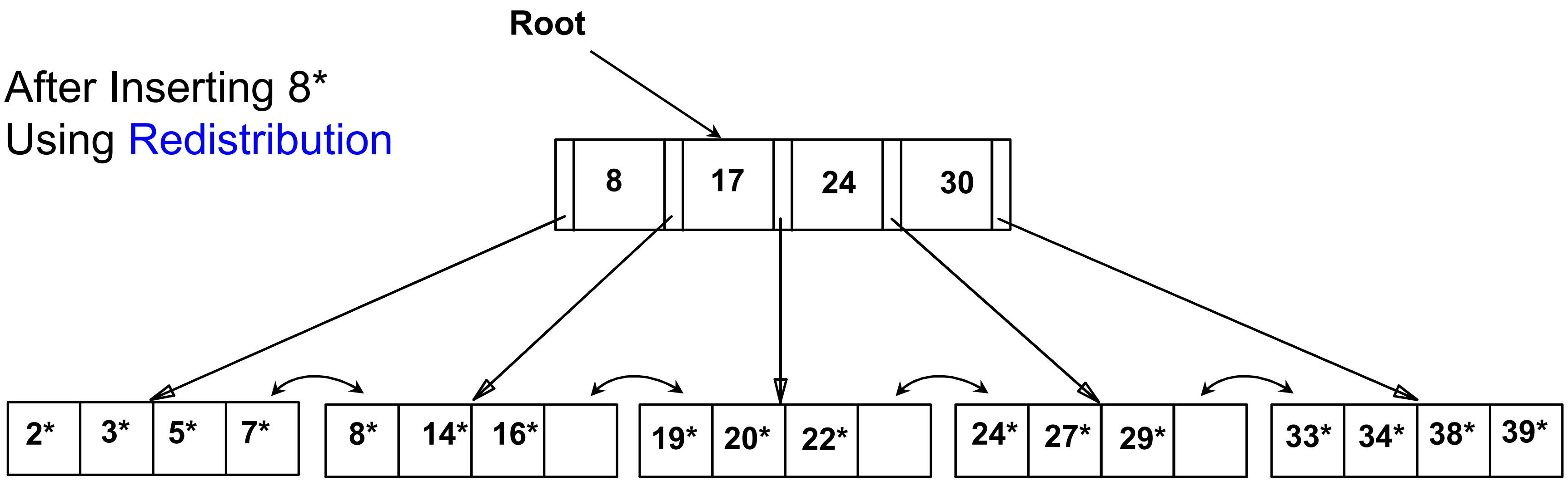
- Notice that root was split, leading to increase in height.
  - Level 2 -> Level 3 by the split.
- In this example, we CAN avoid split by re-distributing entries.
  - As the redistribution improves average occupancy.
  - However, this is usually NOT done in practice. Why? Increasing I/O!

Comparison of Example of B+ Tree

After Inserting 8*
Using Split at Root



After Inserting 8*
Using Redistribution

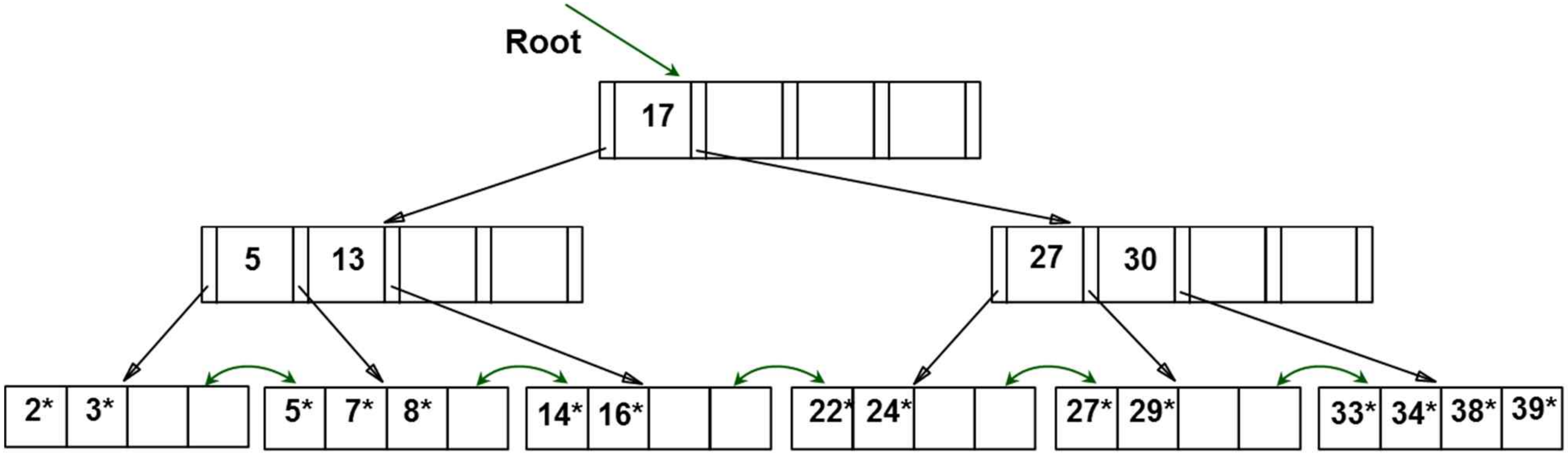


Deleting a Data Entry from a B+ Tree

- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, DONE!
  - If L has only $d\text{-}1$ 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.

  (As the merged node becomes the new root)

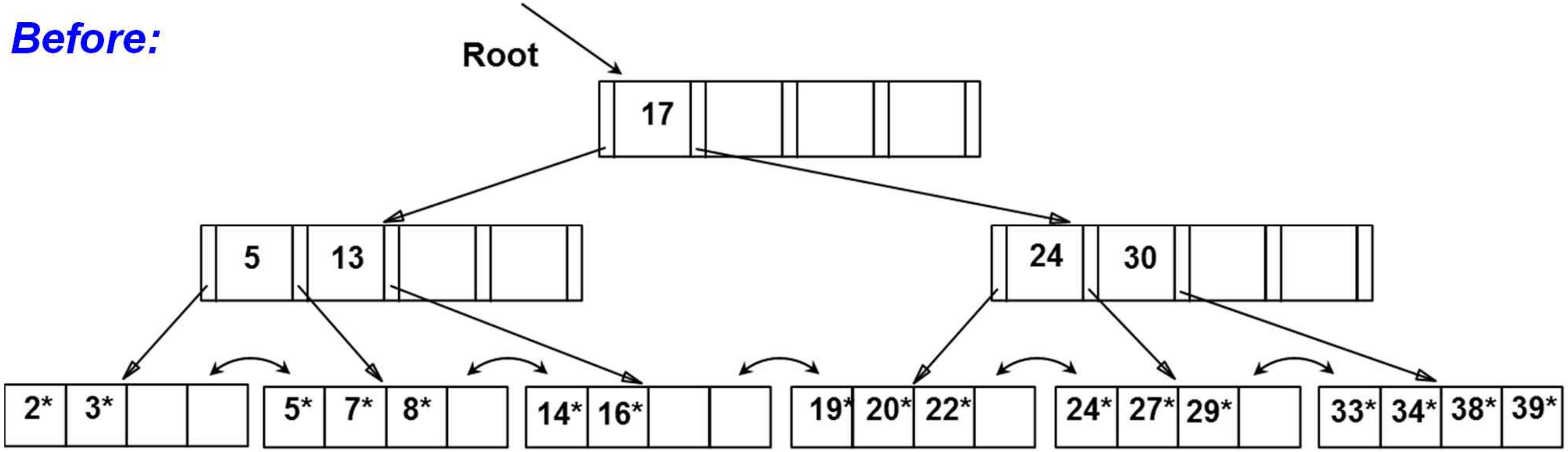
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...



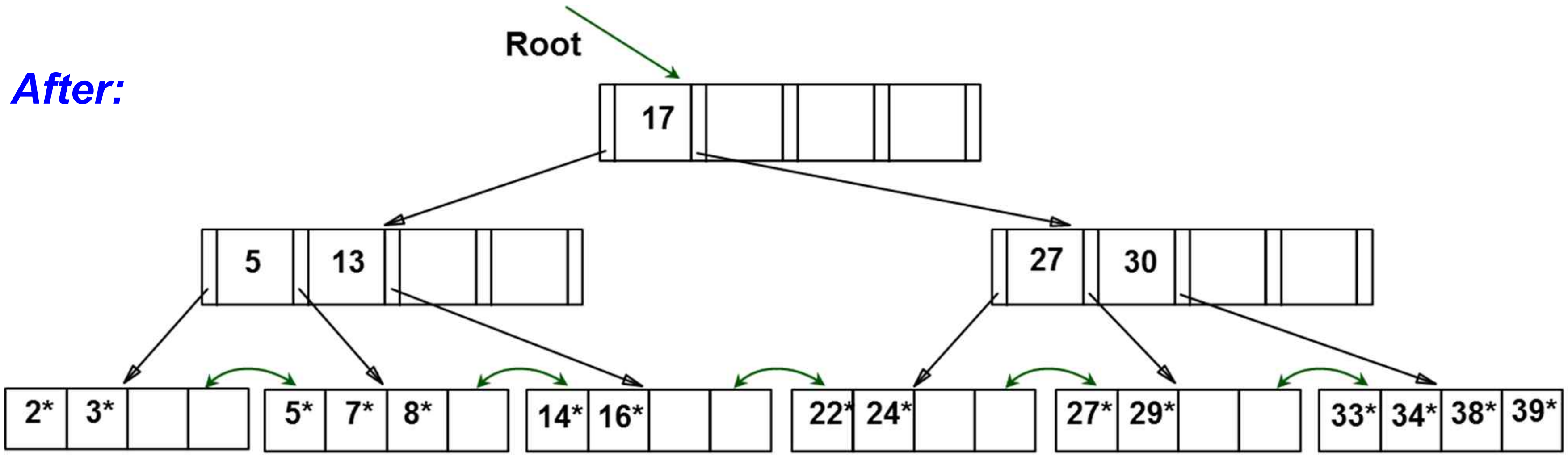
- Deleting 19* is easy.
- Deleting 20* is done with re-distribution as the sibling has three entries.
  - 24 in the sibling is moved to the node that contained 20*.
- Notice how middle key is ***copied up***.
  - (The old leftmost search key, 24, is replaced by 27.)

Example B+-Tree Before/After Deleting 19*, 20*

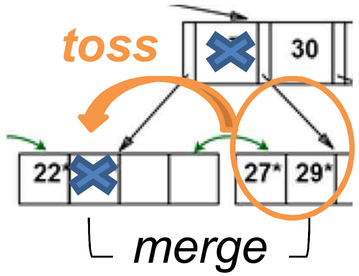
Before:



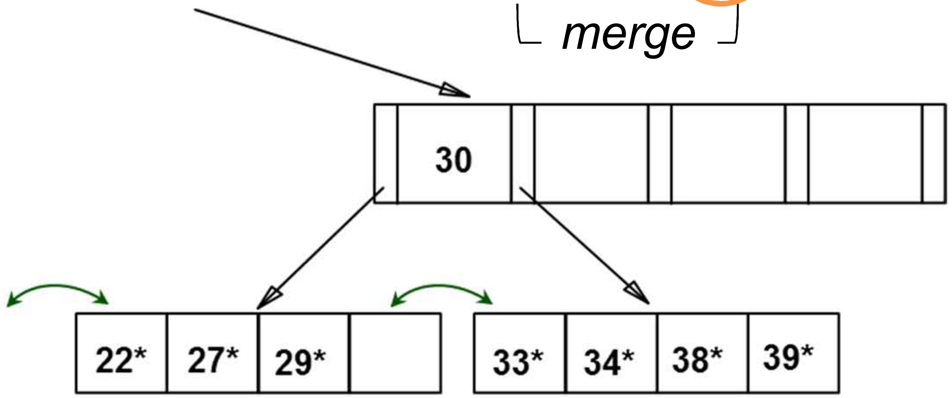
After:



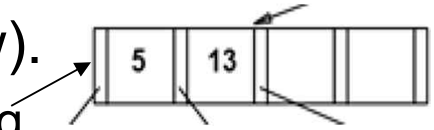
... And Then Deleting 24*



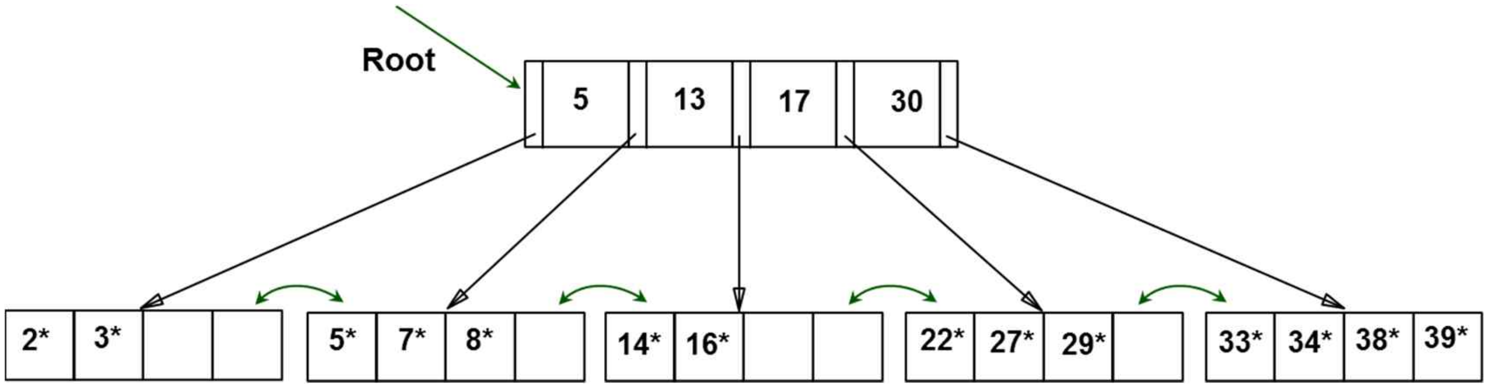
- Cannot redistribute, as the sibling contains only two entries. So must **merge**.
- Observe '**toss**' of index entry (right).



- Observe '**pull down**' of index entry (below).



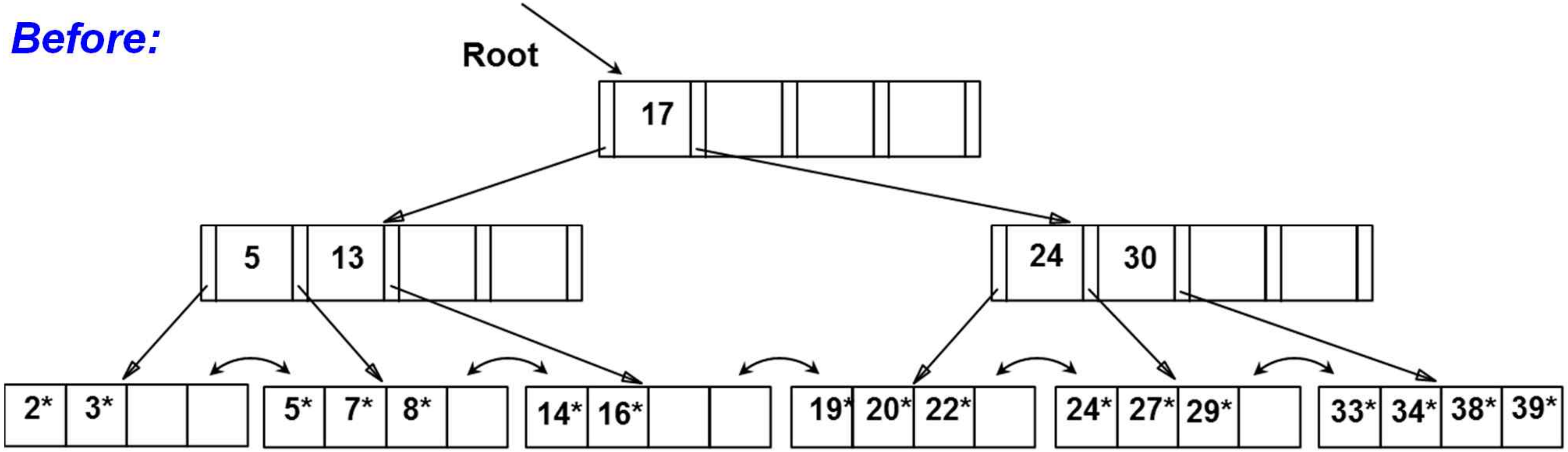
  - We must merge with the node of the left sibling.
  - The splitting key, 17, is pulled down to its child, becoming the new root.
    - The key must be deleted from the old root containing it due to the merge.



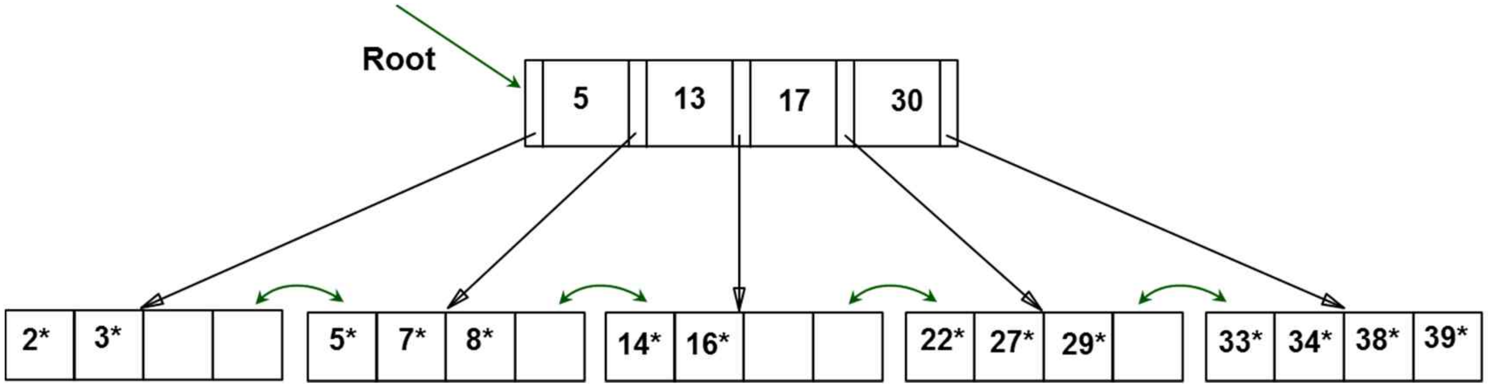
Before/After Deleting 24*

For non-leaf redistribution, see Slides 77-78 in Appendix.

Before:



After:



Summary

- Hashed files (and indexes) are ideal for (exact) equality searches.
- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- 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.

(ISAM: a static structure
  - Only leaf pages modified; overflow pages are needed.
  - Overflow chains can degrade performance.)

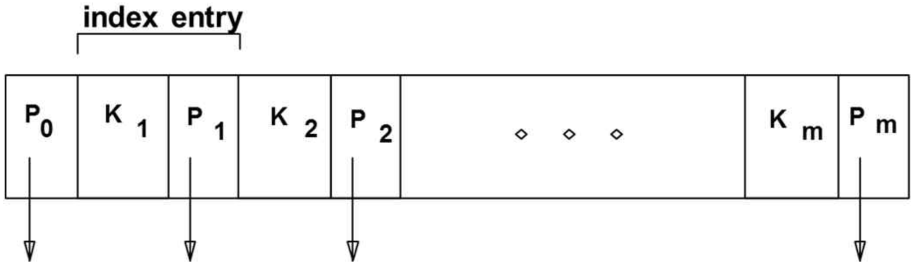
APPENDIX

Indexed Sequential Access Method (ISAM)

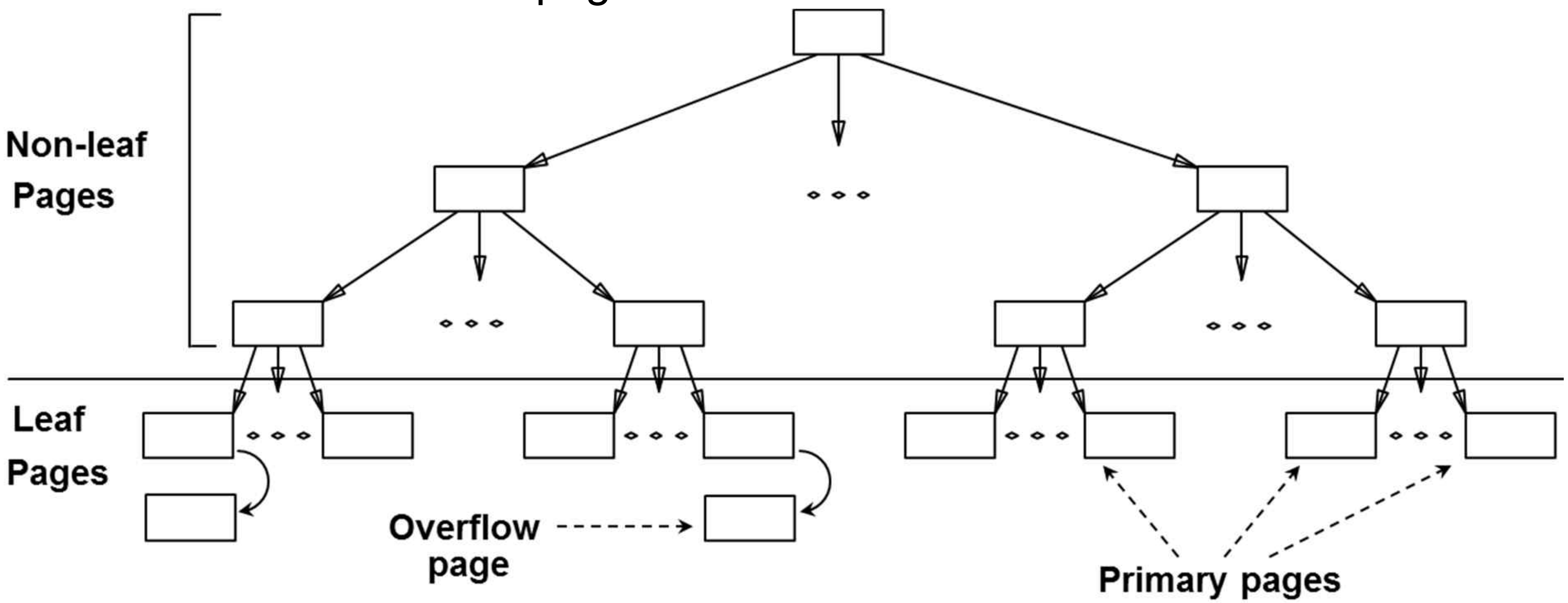
- ISAM: A two-level index
    - Invented by IBM
  - Closely related to the organization of the disk in terms of *cylinders* and *tracks*.
    - The first level: a *cylinder* index
      - <Key value of an anchor record for each cylinder, a pointer to the track index for the cylinder>
    - The second level: a *track* index
      - <Key value of an anchor record for each track in the cylinder, a pointer to the track>

Indexed Sequential Access Method (ISAM) (Cont'd)

- **Static** structure; an index with a *tree* data structure
  - Each tree node: a disk page
  - All the data resides in the leaf pages
    - Additional *overflow* pages chained to some leaf pages.



<ISAM Tree node>



Comments on ISAM

Data Pages
Index Pages
Overflow pages

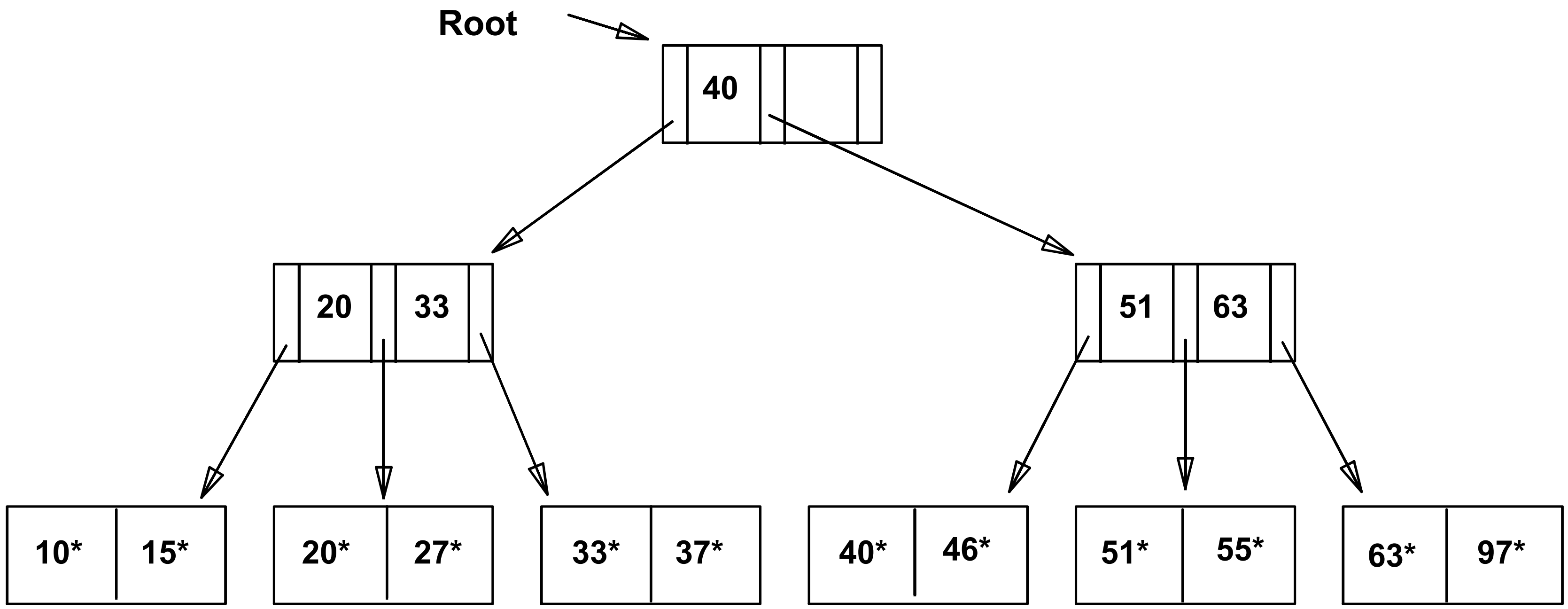
<Page Allocation in ISAM>

- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\log_F N$; F = # entries/index pg, N = # leaf pgs
- *Insert*: Find leaf data entry belongs to, and put it there.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

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

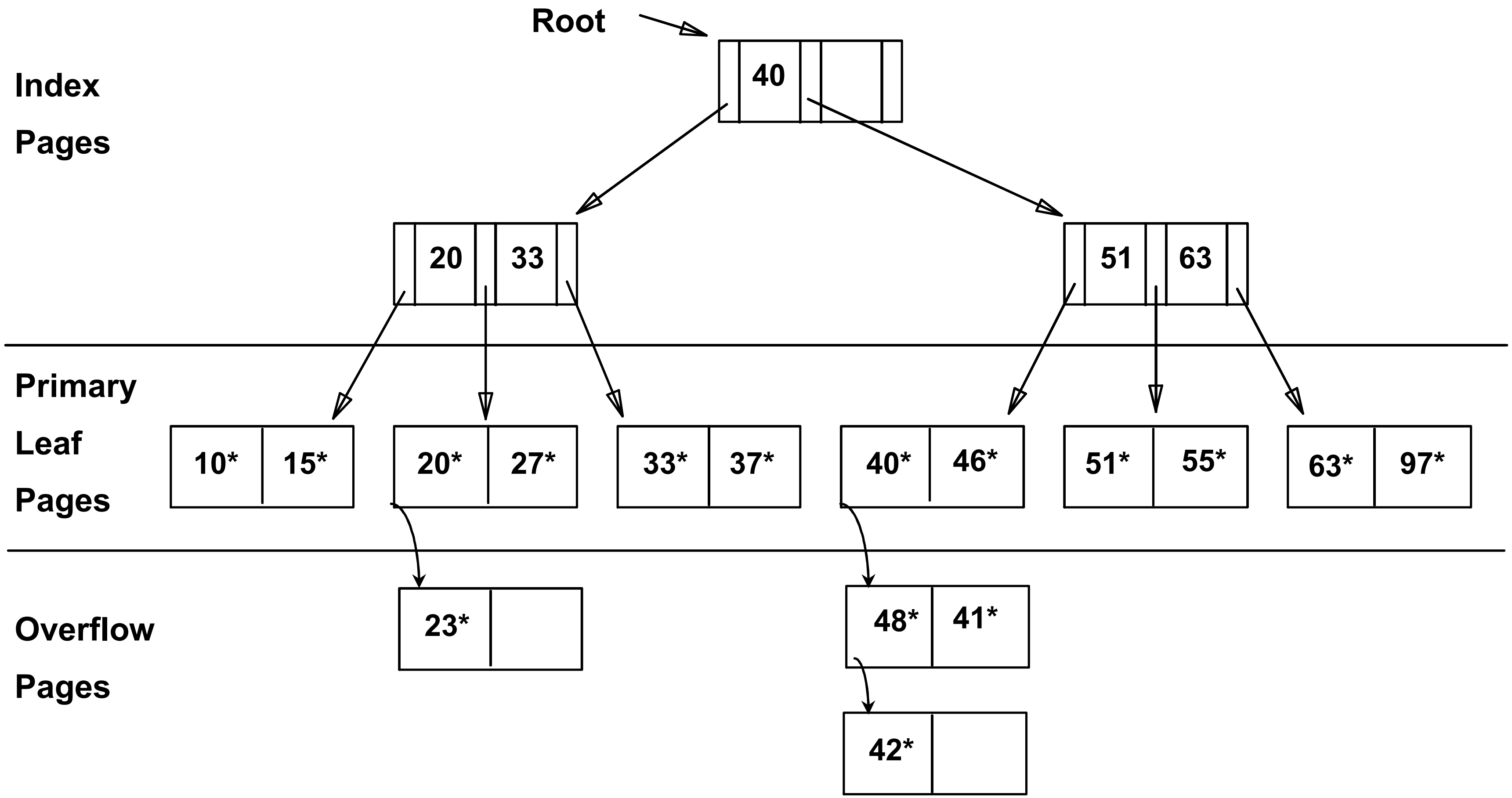
Example of ISAM Tree

- Each node can hold 2 entries.

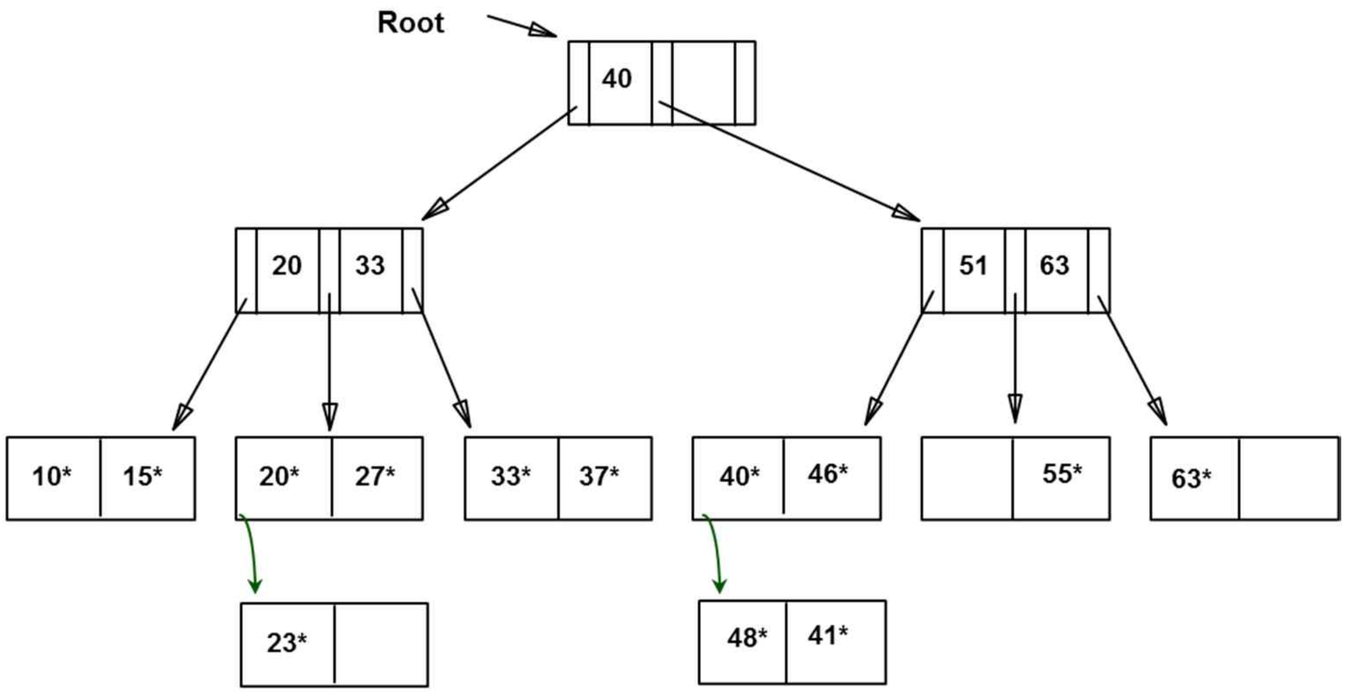


- “No need for ‘next-leaf-page’ pointers. Why?

After Inserting 23*, 48*, 41*, 42* ...



... Then Deleting 42*, 51*, 97*



** Note that 51* appears in index levels, but not in leaf!*

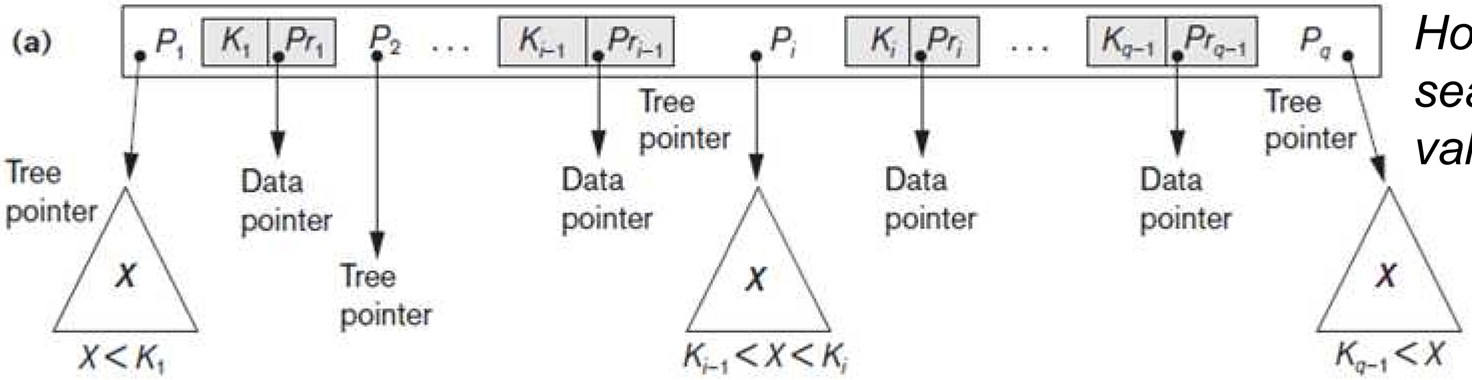
B-Trees

- Provides multi-level access structure
- Used as **primary file organizations**
  - Whole records (rather than <key, record(block) pointer>) are stored within the B-tree nodes
- Tree is always “balanced.”
  - “B” indicates “balanced”.
- Space wasted by deletion, if any, NEVER becomes excessive.
  - Each node is at least “half-full”.
  - Empirically, about 67% full for every node
- Each node in a B-tree of order p can have at most p-1 search values.

B-Tree Structures in Textbook

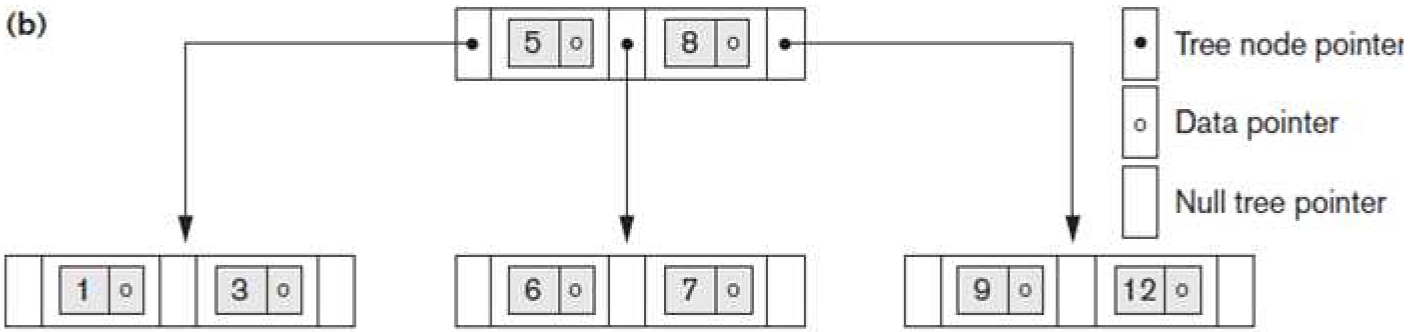
- Tree pointer (P_i): a pointer to another node in the B-tree
- Data pointer (Pr_i): a pointer
  - To the record whose search key field value = K_i (for key field)
  - To the data file block containing that record (for nonkey field)

(a)



How many search values?

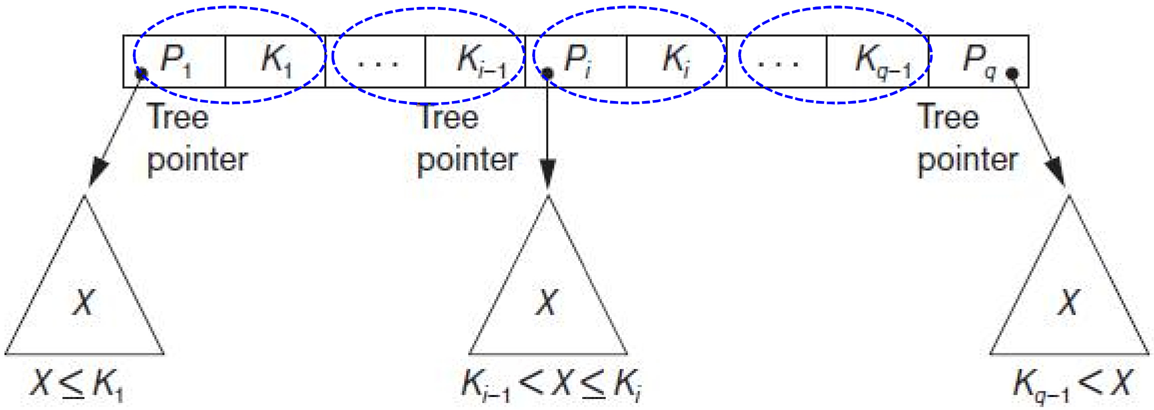
(b)



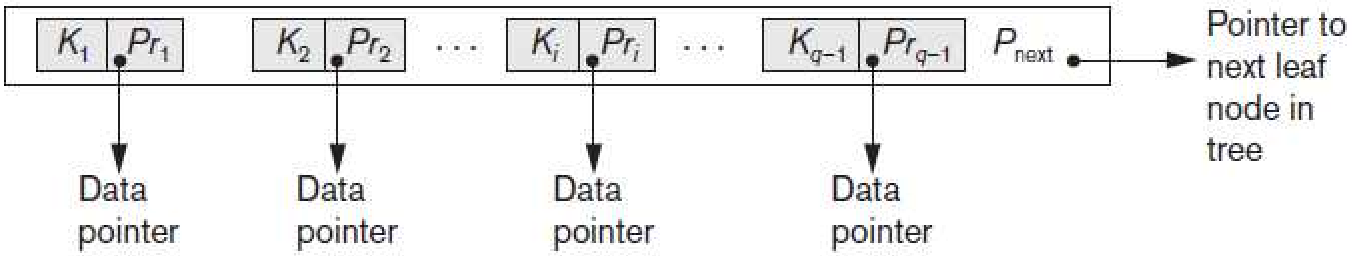
Insertion order: <8, 5, 1, 7, 3, 12, 9, 6>

B+-Tree Structures in Textbook

- Internal node with q-1 search values

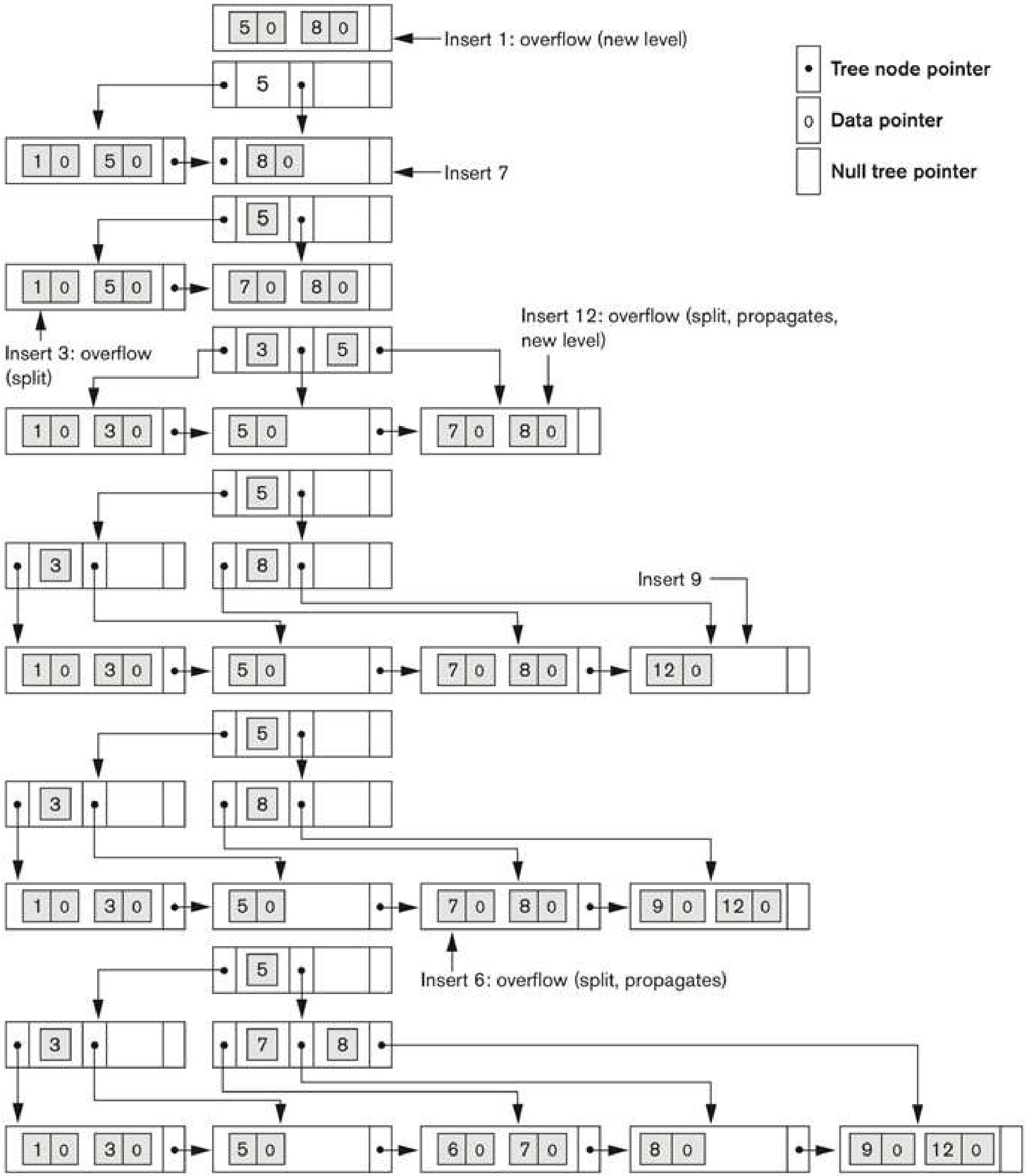


- Leaf node with q-1 search values and q-1 data pointers.



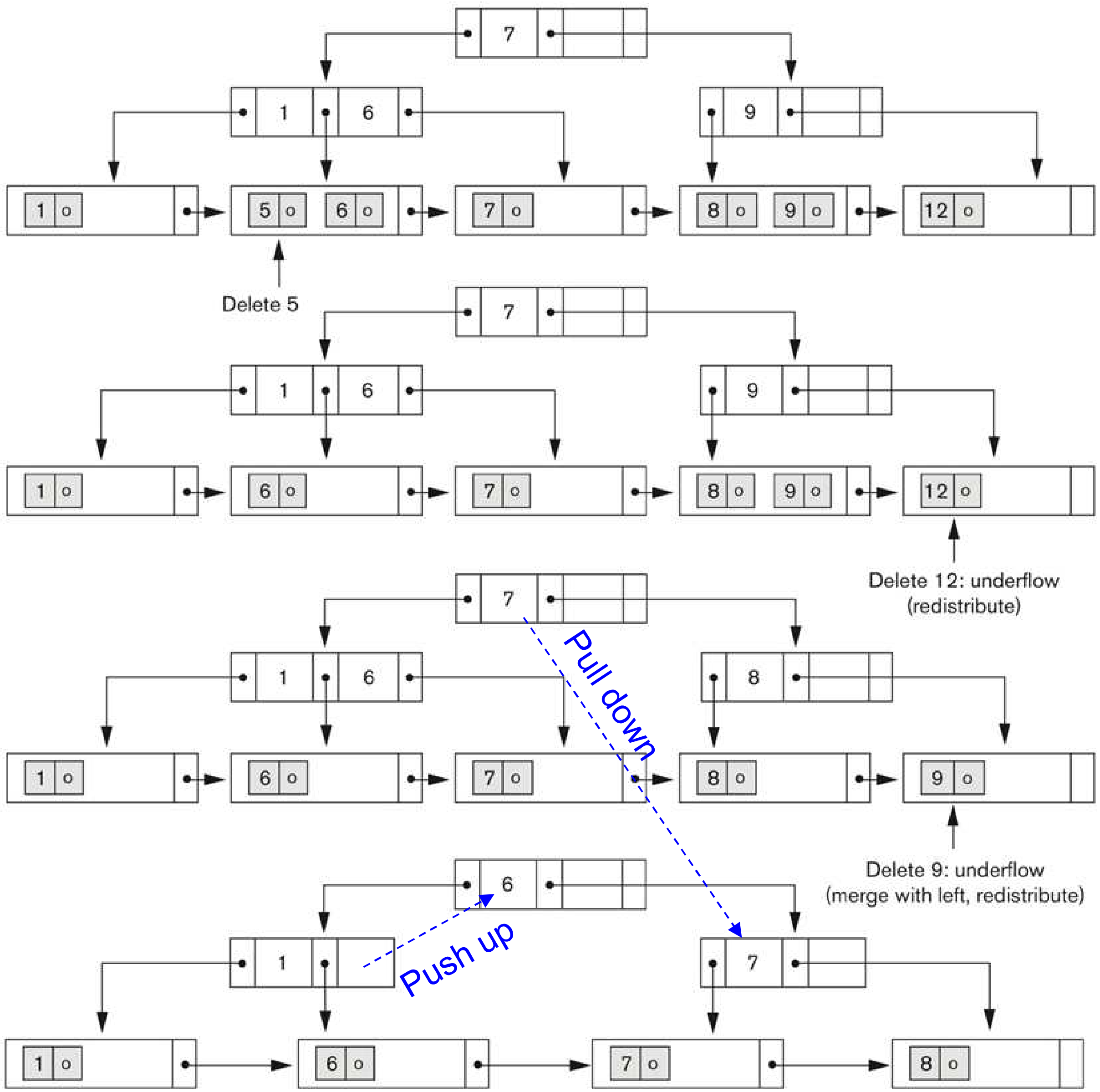
Example of B+-Tree in Textbook

- Fanout= 3
- Order = 1
- Insertion sequence: 8, 5, 1, 7, 3, 12, 9, 6



An Example of Deletion of B+-Tree in Textbook

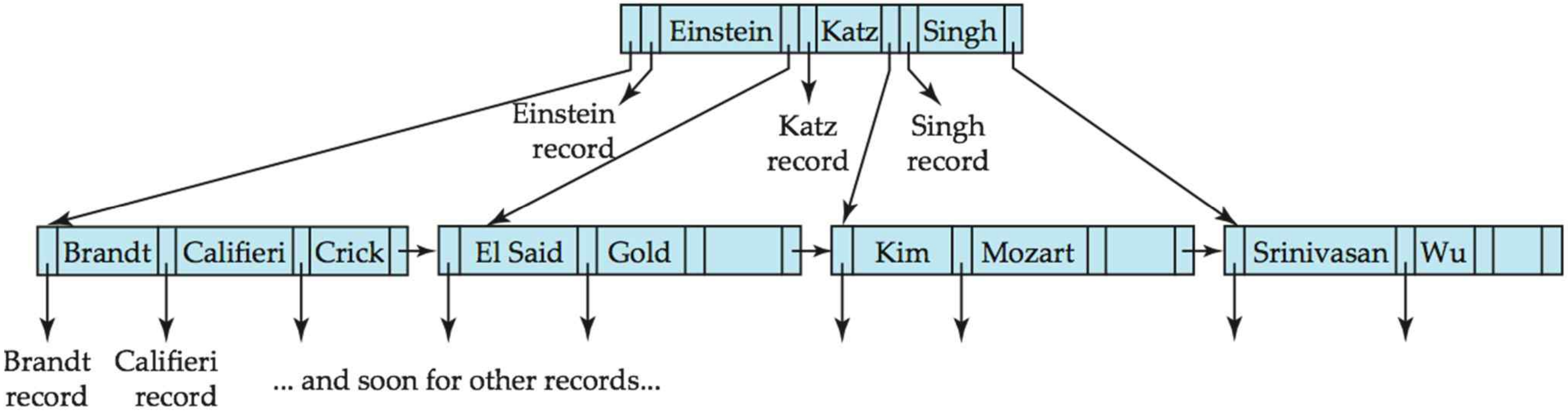
- Fanout= 3
- Order = 1
- Deletion sequence: 5, 12, 9

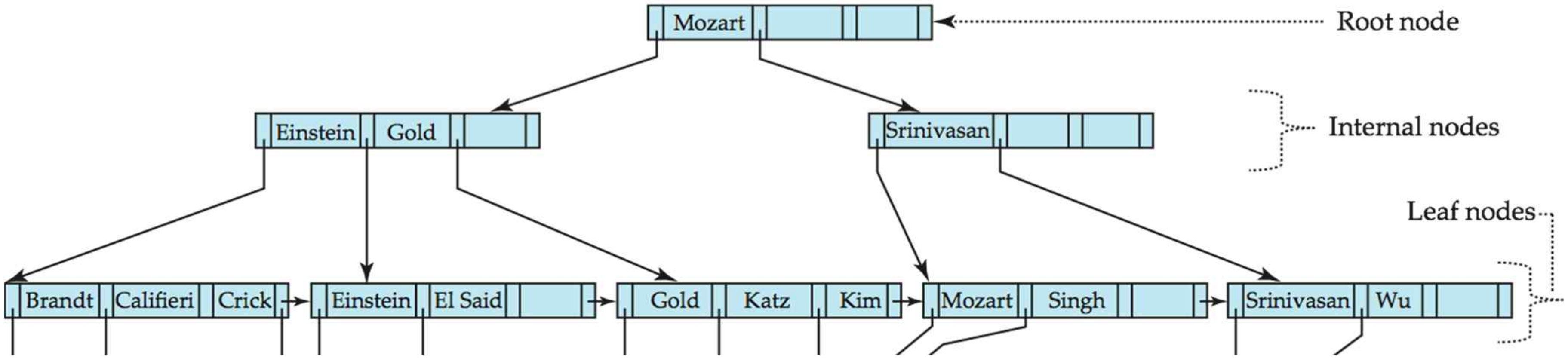


B^+-Trees in Practice

- Typical order: 100. (Typical fill-factor: 67%)
  - Order = minimum search values (or, q-1)
  - 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

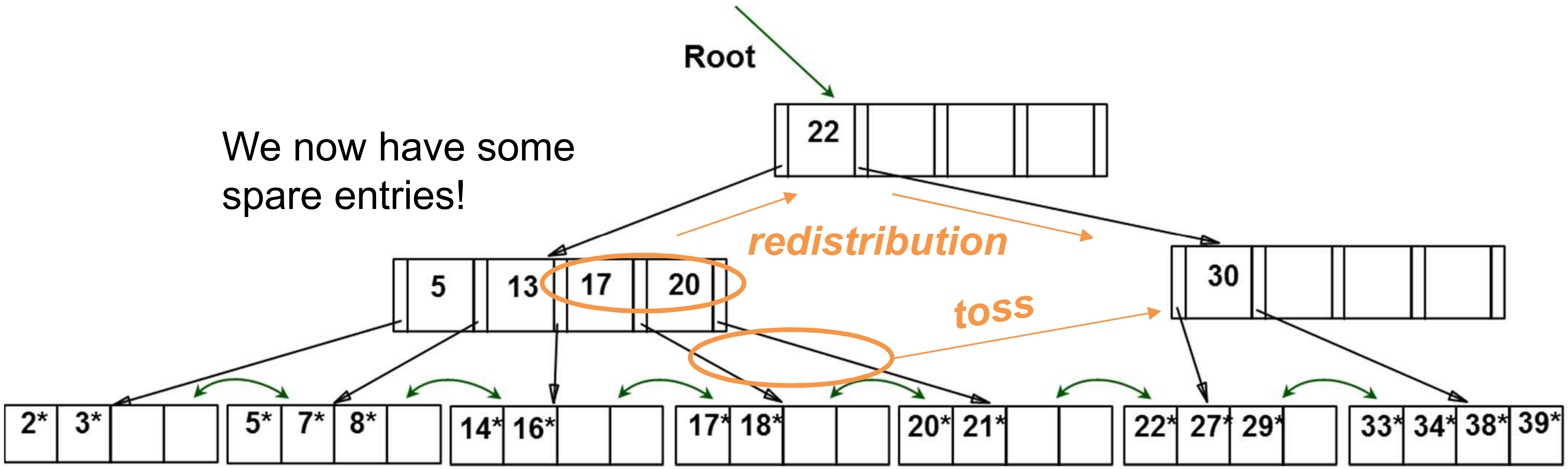
B-Tree (above) vs. B^+-Tree (below)





Example of Non-Leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?!)
  - Similar to the tree on Slide 55 but the left subtree and the root key
- In contrast to previous example, can re-distribute entry from left child of root to right child (through the root).



After Re-distribution

- Intuitively, entries are re-distributed by "*pushing through*" the splitting entry in the parent node.
  - (Indeed, note that it suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for the purpose of illustration.)