Indexing

Cost Model for Execution

- How should we estimate the costs for executing a statement?
  - ☆ Number of I/Os
  - ☆ CPU Execution Cost
  - ☆ Network Cost in distributed system (ignore for now)
- ☆ Assumption in this course
  - ☆ I/O cost >>> CPU cost
  - ☆ Real systems also consider CPU
- Simplifications
  - ☆ only consider disk reads (ignore writes -- assume read-only workload)
  - ☆ only consider number of I/Os and not the individual time for each read (ignores page pre-fetch)
  - ☆ Average-case analysis; based on several simplistic assumptions.
    - ☛ Good enough to show the overall trends!

Typical Operations

- Scan over all records
  - `SELECT * FROM Students`
- Point Query
  - `SELECT * FROM Students WHERE sid = 100`
- Equality Query
  - `SELECT * FROM Students WHERE starty = 2015`
- Range Search
  - ```
    SELECT * FROM Students
        WHERE starty > 2012 and starty <= 2014
    ```
    ```

Typical Operations

- Insert
  - `INSERT INTO Students VALUES (23, ‘Bertino”,2016,… )`
- Delete
  - `DELETE FROM Students WHERE sid = 100`
  - `DELETE FROM Students WHERE endyear < 1950`
- Update
  - Delete+insert

Indexes

- Even a sorted file only supports queries on sorted attributes.
- Solution: Build an index for any attribute (collection of attributes) that is frequently used in queries
  - ☆ **Additional information / extra data structure** that helps finding specific tuples faster
  - ☆ We call the collection of attributes over which the index is built the **search key attributes** for the index.
  - ☆ Any subset of the attributes of a relation can be the search key for an index on the relation.
  - ☆ *Search key* is not the same as *primary key / key candidate*

Creating an index in DB2

search key attribute

- **Simple**
  - ☆`CREATE INDEX ind1 ON Students(sid);`
  - ☆`DROP INDEX ind1;`

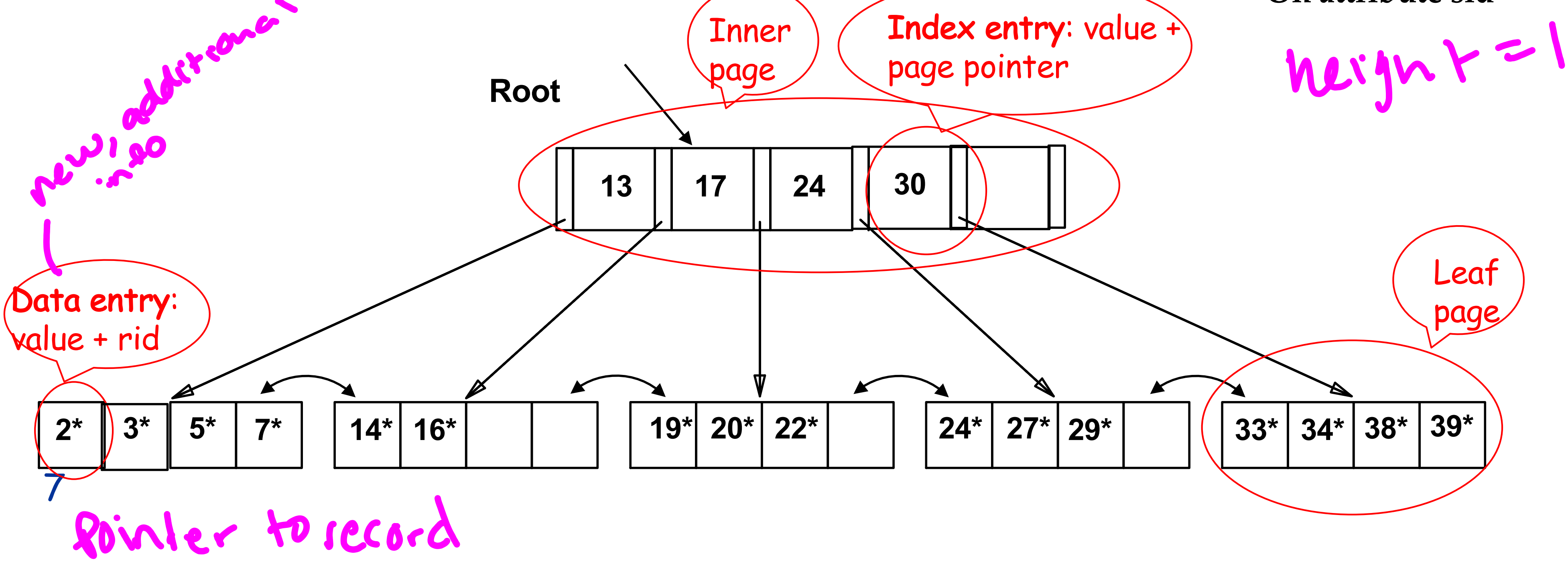
helpful for equality or range queries on sid

not free - extra data, takes up space. makes updates/inserts/deletes more expensive

B+ Tree: The Most Widely Used Index

- Each node/leaf represents one page
  - ☆ Since the page is the transfer unit to disk
- Leafs contain *data entries* (denoted as k*)
  - ☆ For now, assume each data entry *represents* one tuple. The data entry consists of two parts
    - Value of the search key (k)
    - Record identifier (rid = (page-id, slot))
  - ☆ That is: data entry is NOT a tuple but a pointer to a tuple
- Root and inner nodes have auxiliary *index entries*

Index for skaters
On attribute sid



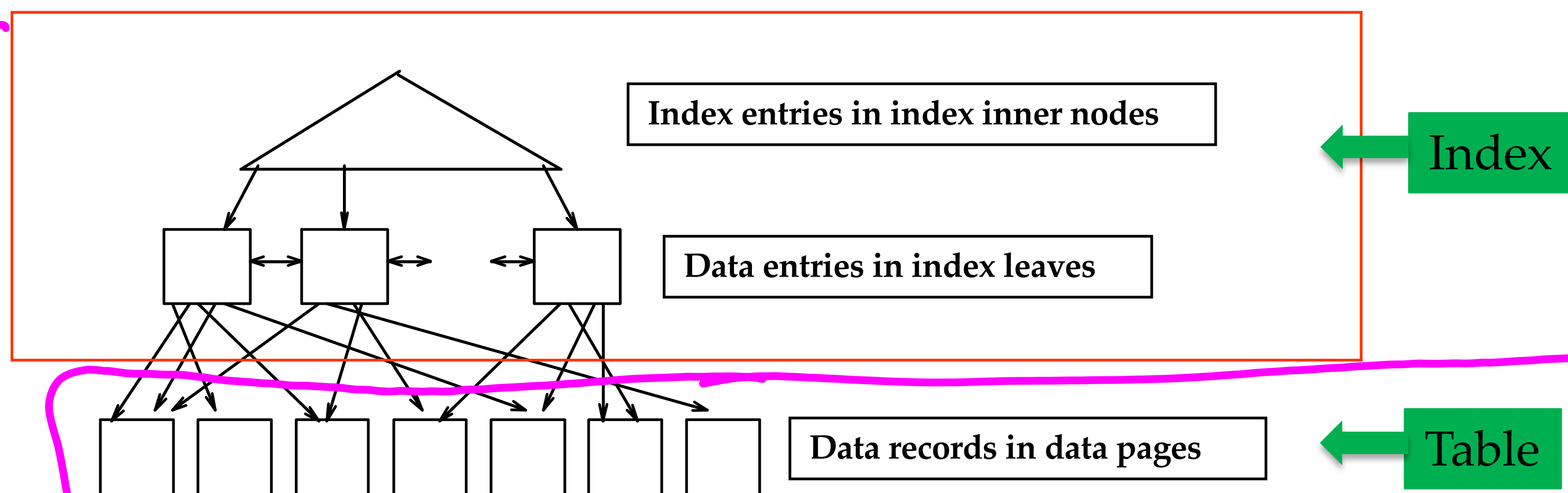
Disclaimer

- Size:
  – Example relation has only 16 tuples
    - You don't build an index for such a relation
  – Think of thousands, and million and more tuples
- Better Example:
  – McGill's student relations
  – Sid = student id = 15-digit number
- Index pages:
  – Inner pages contain hundreds of index entries
  – Leave pages contain hundreds of data entries

B+ Tree (contd.)

- *height-balanced.*
  - ☆Each path from root to tree has the same height
- F = fanout = number of children for each node (~ number of index entries stored in node)
- N = # leaf pages
- Insert/delete at log $_F$ N cost;
- Minimum 50% occupancy (except for root).

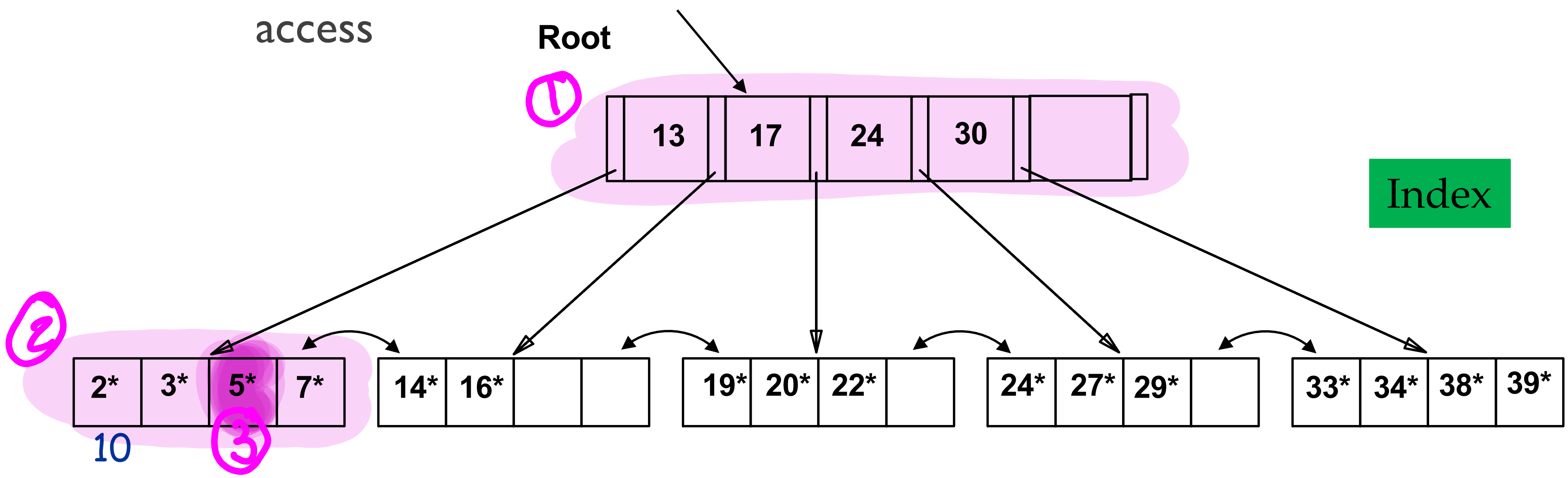
inner pages



pages that contain data

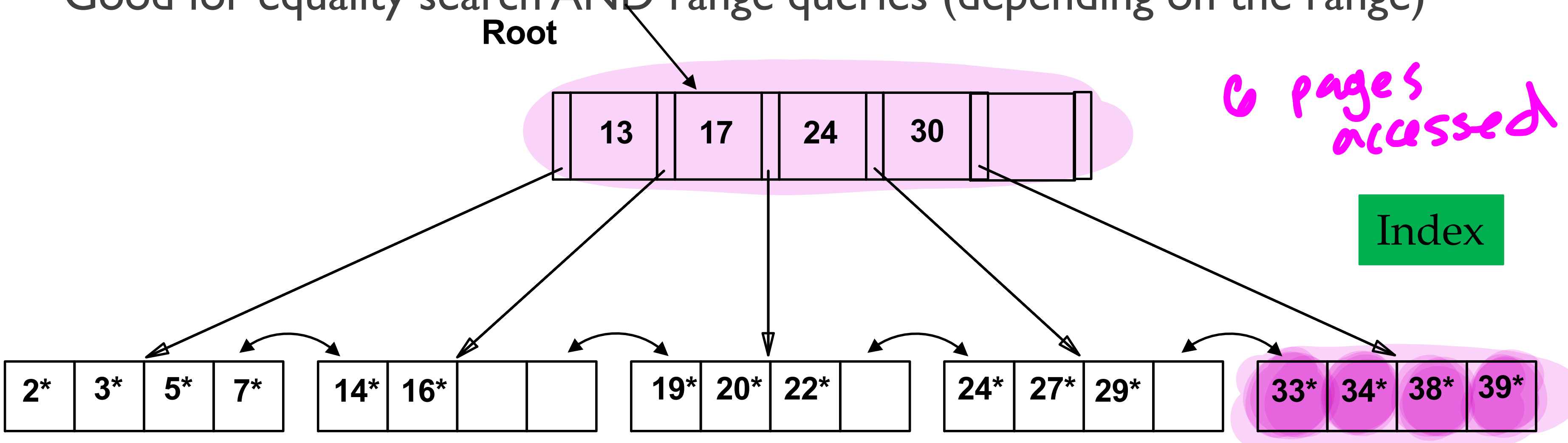
Example B+ Tree

- Example tree has height of 1
- "`Select * from Skaters where sid = 5`"
  - Search begins at root, and key comparisons direct it to a leaf then data page w/ record
  - Number of pages accessed:
    - three: root, leaf, data page with the corresponding record
  - Number of I/O: likely 2
    - depends of how much of tree is already in the buffer in main memory
    - rough assumption: root and intermediate nodes are always in main memory; index leaves and data pages not in main memory upon first access



Example B+ Tree

- “`Select * from Skaters where sid = 5`”
  - I/O costs:
    - one for leaf page with data entry, one for data page with data record
- “`Select * from Skaters where sid >= 33`” range query
  - I/O costs: 5
    - one for leaf page
    - four for data pages with records
- Good for equality search AND range queries (depending on the range)



6 pages accessed

Index

Inserting a Data Entry

1 • Find correct leaf *L.*

2 • Put data entry into *L.*

2.1 – If *L* has enough space, *done*!

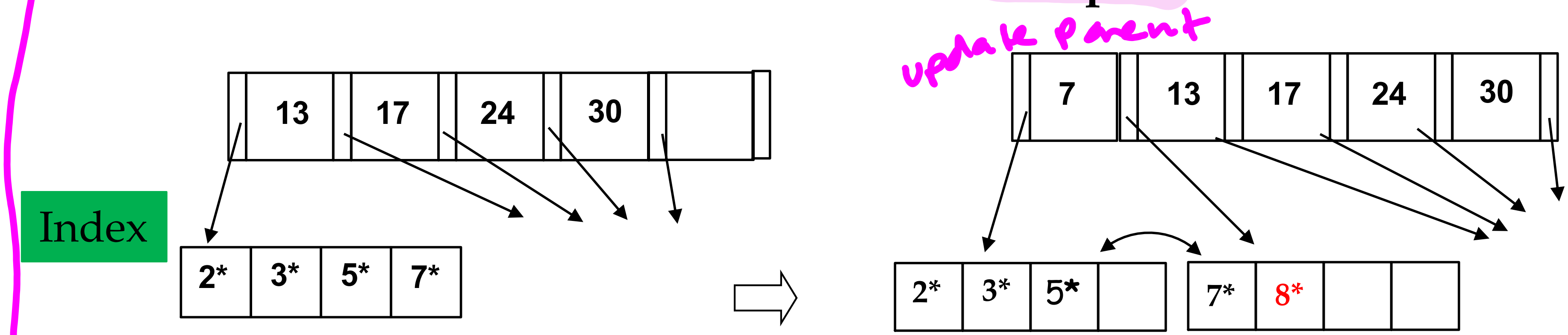
2.2 – Else, must *split* *L* *(into L and a new node L2)*

  - 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.*

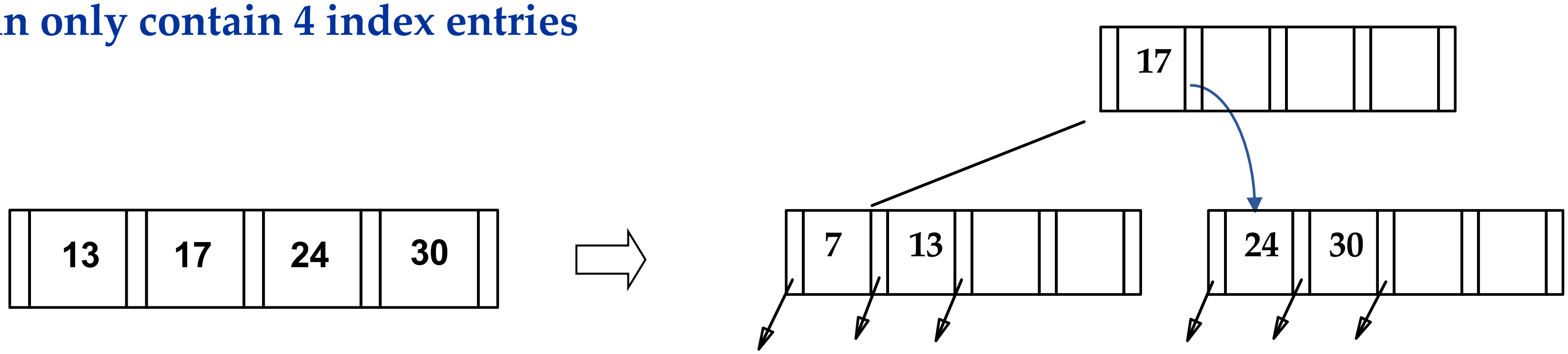
Inserting 8* into Example B+ Tree

Insert into Leaf with leaf split



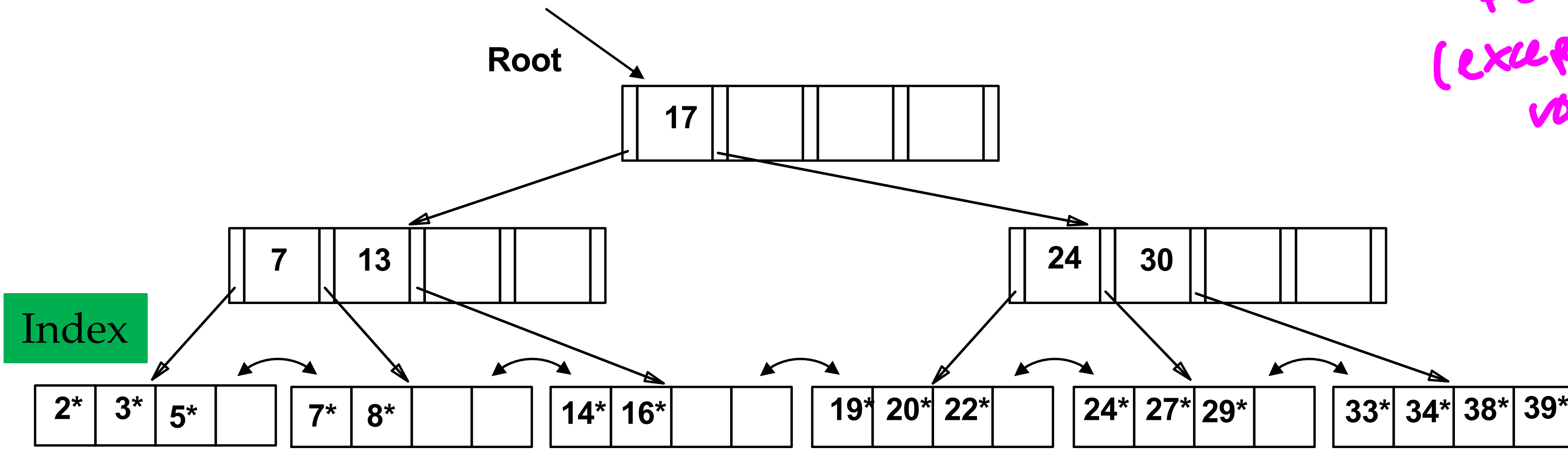
Insert into internal node with node split

Assume that inner pages
can only contain 4 index entries



Example: After Inserting 8*

all half full (except root)



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

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

height only increases when root splits

pointer to record

Data Entry Alternatives: indirect Indexing

- Indirect Indexing I
  - so far: k* = <**k**, rid of data record with search key value **k**> (indirect indexing)
  - on non-primary key search key: (2015, rid1), (2015, rid2), (2015, rid3), …
    - several entries with the same search key side by side
- Indirect indexing II several tuples with the value
  - <**k**, list of rids of data records with search key **k**> (indirect indexing)
  - on non-primary key search key: (2015, (rid1, rid2, rid3,…)), (2016, (rid…

don't repeat values

- Comparison:
  - first requires more space (search key repeated)
  - second has variable length data entries
  - second can have large data entries that span a page

list vs individual

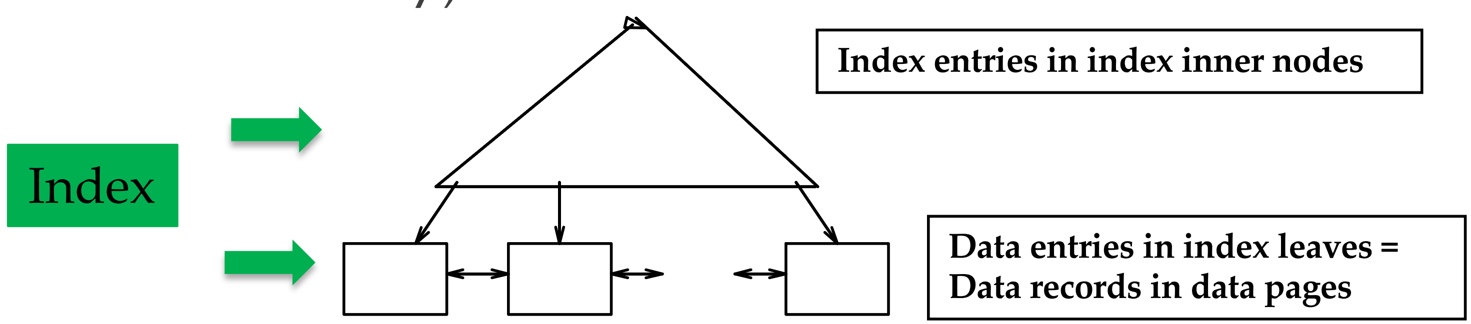
Direct Indexing

- Instead of data-entries in index leaves containing rids, they could contain the entire tuple
  - data-entry = tuple
  - no extra data pages ←
- This is kind of a sorted file with an index on top

leaf pages = data pages

Data Entry Alternatives: Direct Index

- Structure
  - <**k**, full record>
  - Leaf pages contain entire records
    - Data entries = records (not only pointers to them)
    - e.g., index on sid of Skaters
    - (1,lilly,10,16), (2,debby,8,10)…
  - Leafs represent sorted file for data records.
  - Inner nodes above leafs provide faster search
  - Typically at most one direct index per relation
    - (Otherwise, data records duplicated, leading to redundant storage and potential inconsistency.)



Index Classification

secondary - search key is not unique

- ❑ Primary vs. secondary: If search key contains primary key, then called primary index.
  - ☆ *Unique* index: Search key is primary key or unique attribute.
- ❑ Clustered vs. unclustered:
  - ☆ clustered:
    - ● Relation in file sorted by the search key attributes of the index
  - ☆ unclustered: (not sorted or sorted by a diff. attribute)
    - ● Relation in heap file or sorted by an attribute different to the search key attribute of the index.
  - ☆ 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!

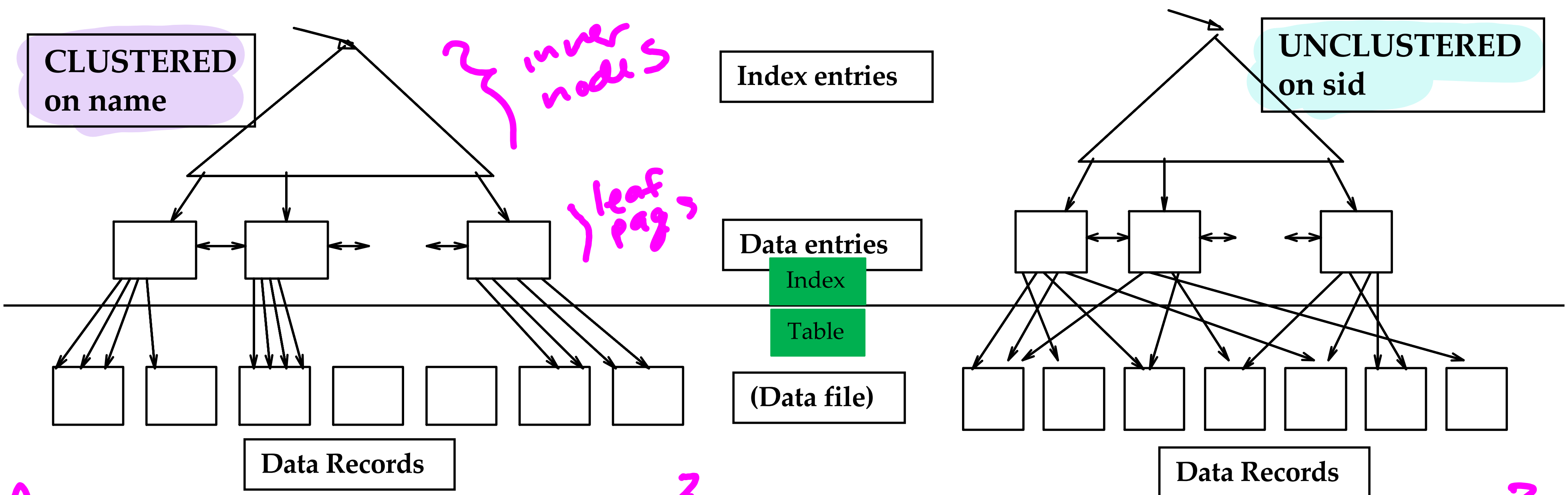
direct index is a clustered index

indirect index is either clustered or non-clustered index

Clustered vs. Unclustered Index

leaf pages contain rid which points to data page

- Example for Students:
  - ☆Indirect clustered index on name
  - ☆Indirect unclustered on sid



A Z A Z

data sorted by name

data sorted by name

① how big is my index, how many data pages

B+-tree cost example: $\boxed{\text{Given}}$

- About the relation
  - Relation R(A,B,C,D,E,F)
  - A and B are int (each 4 Bytes), C-F is char[40] (160 Bytes)
  - Values of B are within [0;19999] uniform distribution
  - 200,000 tuples
- About the heap file (unsorted)
  - Each data page has 4 K and is around 80% full (assume)
    - I know, I know: with fixed sized records we don't need to leave space, but in real life records don't have fixed length and we should leave space….
  - The size of an rid = 10 Bytes
- About the index to be built (on B) indirect index alternative II - one data entry per different value of the search key
  - An index page has 4K
  - Index pages are filled between 50% - 100% (this always holds!!) - assume they are on average 75% full
  - The size of a pointer in intermediate pages: 8 Bytes

Search key is a value of B

Size of B+tree: To calculate

- ☆ **Number of data pages in the heap file**
  - ☆ size of the tuple = 168 fill 9/3
  - ☆ 4000 * 0.75 / 168 ≈ 18 tuples in a page on an average. or 200000 * 168 / 3000 = 11200
  - ☆ 200000 / 18 ≈ 11111 pages in total.

index page size

tuples

75% of 4k = 3000

- ☆ **The number of leaf pages**
  - ☆ distinct values of B = 20000
  - ☆ 200000 / 20000 = 10 records per val of B total # tuples / # values of B
  - ☆ sizeof(B) + 10 rids * sizeof(rids)
  - ☆ 4 + 10 * 10 = 104 bytes/data entry size of data entry
  - ☆ 4000 * 0.75 / 104 ≈ 28.846 data entries/page on an average.
  - ☆ 20000/28.846 ≈ 694 leaf pages.

L_{min}: 4000 · 0.50/104

L_{max}: 4000 · 1.0/104

- ☆ **The height of the tree**
  - ☆ 4 + 8 = 12 bytes intermediate node entry size of key + pointer size
  - ☆ Can fit 4000 * 0.75 / 12 = 250 entries per intermediate node
  - ☆ This requires 3 intermediate nodes right above the leaf pages (to manage 694 leaves).
  - ☆ 1 node above then to hold pointers to the 3 intermediate pages - root.
  - ☆ height = 2 (the number of edges from root to a leaf node)

B+ Trees in Practice

- Typical order d of inner nodes: 200 (I.e., an inner node has between 100 and 200 index entries)
  - Typical fill-factor: 67%.
  - average fanout = 133
- Leaf nodes have often less entries since data entries larger (rids)
- Typical capacities (roughly)
  - Height 3: 133^4 = 312,900,721 records
  - Height 2: 133^3 = 2,352,637 records
  - Height 1: 133^2 = 17,689 records

$133^3 \cdot 100 = 235263700$

$133^2 \cdot 100 = 1768900$

$133 \cdot 100 = 13300$

- Can often hold top levels in buffer pool:
  - Level 1 (root) = 1 page = 4 Kbytes
  - Level 2 = 133 pages = 0.5 Mbyte
  - Level 3 = 17,689 pages = 70 MBytes

Multi-attribute index

- Index on `Skaters(age,rating);`
- Order is important:
  - Here data entries are first ordered by age
  - Skaters with the same age are then ordered by rating within groups
  - assume youngest skater is 3, oldest is 25 (list of rids: *):
    - the leaf pages then would look like:

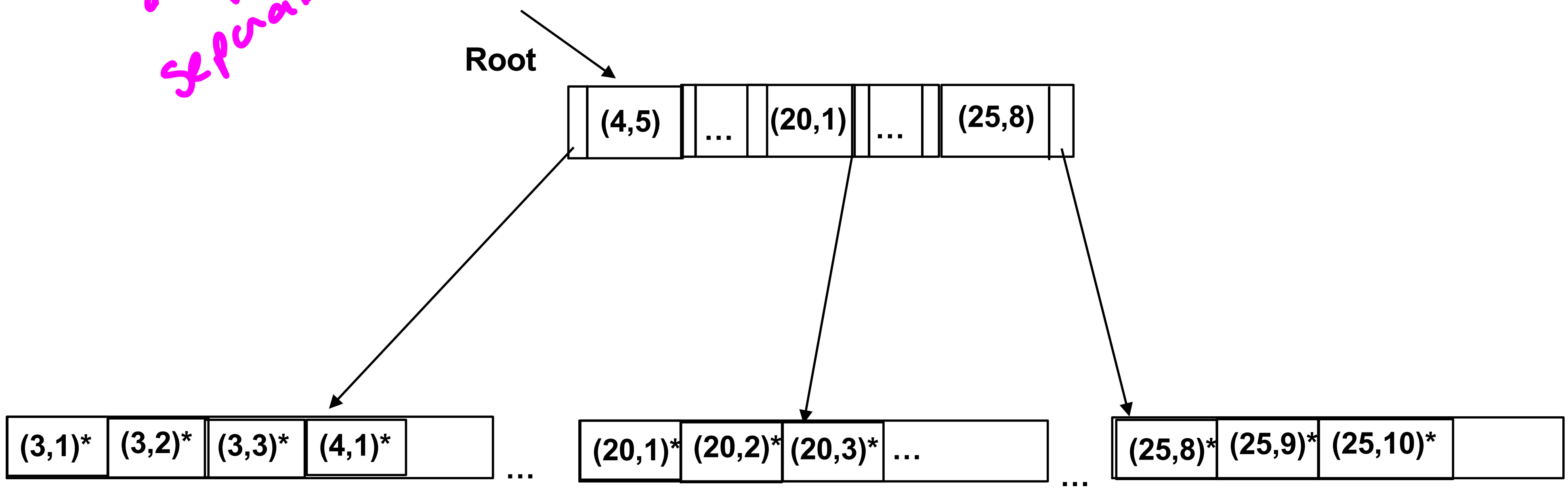
(3,1)*	(3,2)*	(3,3)*	(4,1)*		…	(20,1)*	(20,2)*	(20,3)*	…		…	(25,8)*	(25,9)*	(25,10)*		

youngest skater w/ lowest rating

oldest skater w/ highest rating

why?
↳ not many ages or ratings separately

What does it support



- What does it support?
  - SELECT * FROM Skaters WHERE age = 20;
    - yes
  - SELECT * FROM Skaters WHERE age = 20 AND rating < 5;
    - Yes
  - SELECT * FROM Skaters WHERE rating < 5; – query only on second attribute
    - no – would have to go through all entries
      ↳ index is not useful

Index in DB2

- Simple
  - ☆ `CREATE INDEX ind1 ON Students(startyear);`
  - ☆ `DROP INDEX ind1;`
- Index also good for referential integrity (uniqueness)
  - ☆ `CREATE UNIQUE INDEX indemail ON Students(email)`
- Additional attributes
  - ☆ `CREATE UNIQUE INDEX ind1 ON Students(sid) INCLUDE(name)`
  - ☆ Index only on sid
  - ☆ Data entry contains key value (sid) + name + rid
  - ☆ `SELECT name FROM Students WHERE sid = 100`
    - Can be answered without accessing the real data pages of Students relation!
- Clustered index
  - ☆ `CREATE INDEX ind1 on Students(name) CLUSTER`

include some of the tuple in the page

multi-attribute: create index ind2 on students(email, sid)

Summary for B+-trees

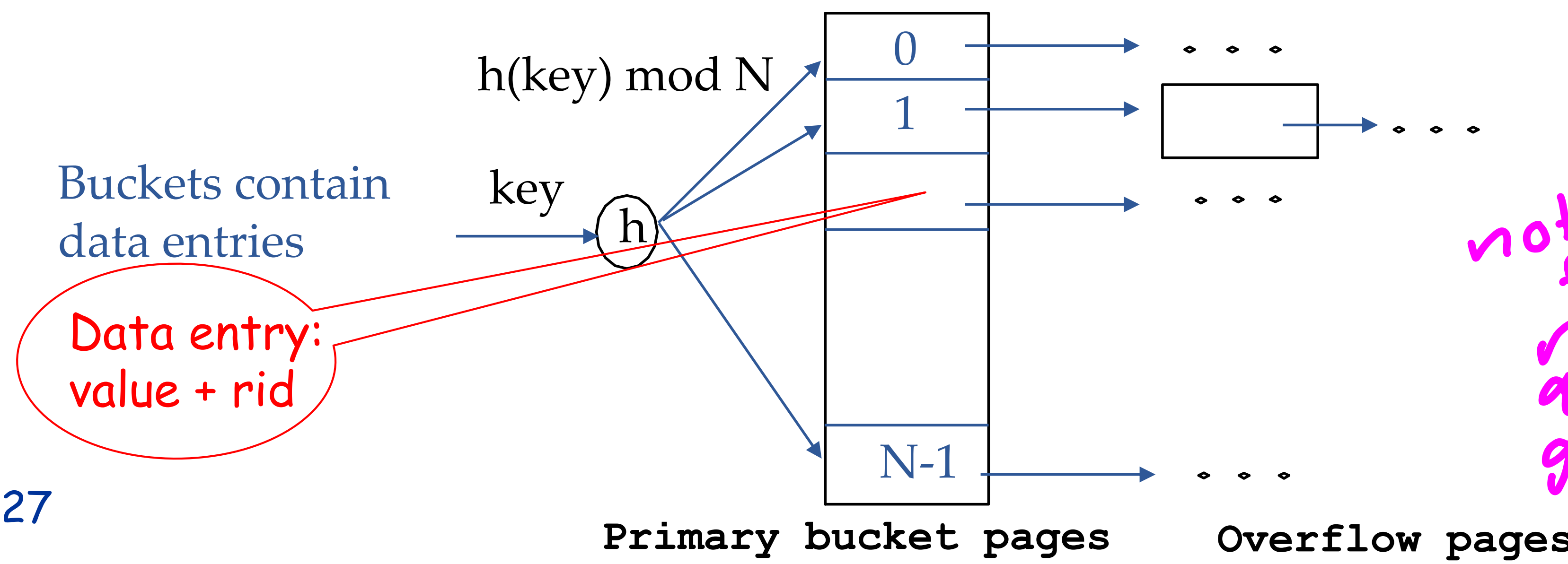
- Tree-structured indexes are ideal for range-searches, also good for equality searches.
  - High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.
- Can have several indices on same tables (over different attributes)
- Most widely used index in database management systems because. One of the most optimized components of a DBMS.

Other index structures besides B tree

Static Hashing

- Similar to standard hashing but with pages as the unit of storage (compare with array-based main memory implementation)
- Decide on a number N of buckets at index creation
- Allocate one primary page per bucket
- Overflow pages for individual buckets are created later as needed
- Buckets contain data entries (same as leaf pages in tree)
- let k be the search key of the index, h a hash function
  - **h**(*k*) mod N = bucket to which data entry with key *k* belongs.

good for point queries



not good for range queries

good for equality search

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**.
- Long overflow chains can develop and degrade performance.
  - Several optimizations developed such to handle scale dynamically (e.g., extensible and linear hashing)

File Organizations

- <u>Hashed Files:</u>.
  - ☆File is a collection of <u>*buckets*</u>. 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*.
  - ☆Best for equality search (only one page access and maybe access to overflow page)
  - ☆No advantage for range queries
  - ☆Fast insert
  - ☆Cost on delete depends on cost for WHERE clause

File Organization

assume each relation is a file:

- Heap files:
  - Linked, unordered list of all pages of the file
  - How does it do?
    - scan retrieving all records (SELECT *)?
      - ok, you have to retrieve all pages anyways
    - Point query on unique attributes
      - not great: have to read on avg. half the pages to return one record
    - range search or equality search on non-primary key
      - not great: have to read all pages to return subset of records.
    - insert
      - yes: Cost for insert low (insert anywhere)
    - delete/update
      - same as for equality/range search -- depends on WHERE clause

File Organizations II

- <u>Sorted Files:</u>
  - ☆ Records are ordered according to one or more attributes of the relation
  - ☆ Is it good for:
    - scan retrieving all records (SELECT *)?
      - ▲ yes, you have to retrieve all pages anyways
    - equality search on sort attribute
      - ▲ good: find first qualifying page with binary search in log2(number-of-pages)
    - range search on sort attribute
      - ▲ good: find first qualifying page with binary search in log2(number-of-pages); adjacent pages might have additional matching records
    - insert
      - ▲ not good: have to find proper page; overflow possible
    - delete/update
      - ▲ finding tuple same as equality/range search depending on WHERE clause
      - ▲ update itself might lead to restructuring of pages
    - Sorted output: (ORDER BY)
      - ▲ good if on sorted attribute