Midterm 1 Review

Fall 2017

SQL

Single Table Queries

```
SELECT [DISTINCT] <columns>
FROM <single table>
[WHERE <predicate>]
[GROUP BY <columns>
    [HAVING <predicate>] ]
[ORDER BY <column list>];
```

SQL Joins (Multi-Table Queries)

```
SELECT [DISTINCT] <columns>

FROM <table>

    [INNER | {LEFT | RIGHT| FULL} {OUTER}] JOIN <table>

    ON <predicate>

[WHERE <predicate>]
```

Other SQL Keywords To Know

- UNION
- INTERSECT
- IN
- EXISTS
- COUNT
- ALL
- ANY
- NOT

Example SQL MT Question 1: Sp15 MT1 V 4

```
Homes(home_id int, city text, bedrooms int, bathrooms int,
area int)

Transactions(home_id int, buyer_id int, seller_id int,
transaction_date date, sale_price int)

Buyers(buyer_id int, name text) Sellers(seller_id int, name
text)
```

Fill in the blanks in the SQL query that finds, for each home in Berkeley, the id of the home and the price for which it was sold. If the home has not been sold yet, the price should be NULL.

Answer Sheet

SELECT ______________________________

FROM ______________________________

______________ JOIN ______________________

ON____

______________________________;

Answer Sheet

SELECT H.home_id, T.sale_price

FROM Homes H

LEFT OUTER JOIN Transactions T

ON H.home_id = T.home_id

WHERE H.city = “Berkeley”;

Example SQL MT Question 2: Sp15 Final VI 3a

Passenger(pid int, first_name text NOT NULL, last_name text NOT NULL)

Driver(did int, first_name text NOT NULL,last_name text NOT NULL)

Trip(tid int, pid int references Passenger(pid) NOT NULL, did int references Driver(did) NOT NULL, start_time timestamp NOT NULL, end_time timestamp NOT NULL, distance decimal NOT NULL, passenger_rating decimal, driver_rating decimal)

Which drivers have a perfect 5.0 average rating from all their trips that received driver ratings? (Return just the unique did)?

Answer Sheet

```
SELECT __________
FROM Trip AS T1
WHERE 5.0 = ALL (
    SELECT _________________
    FROM Trip AS T2
    WHERE ________________________________________________ );
```

Answer Sheet

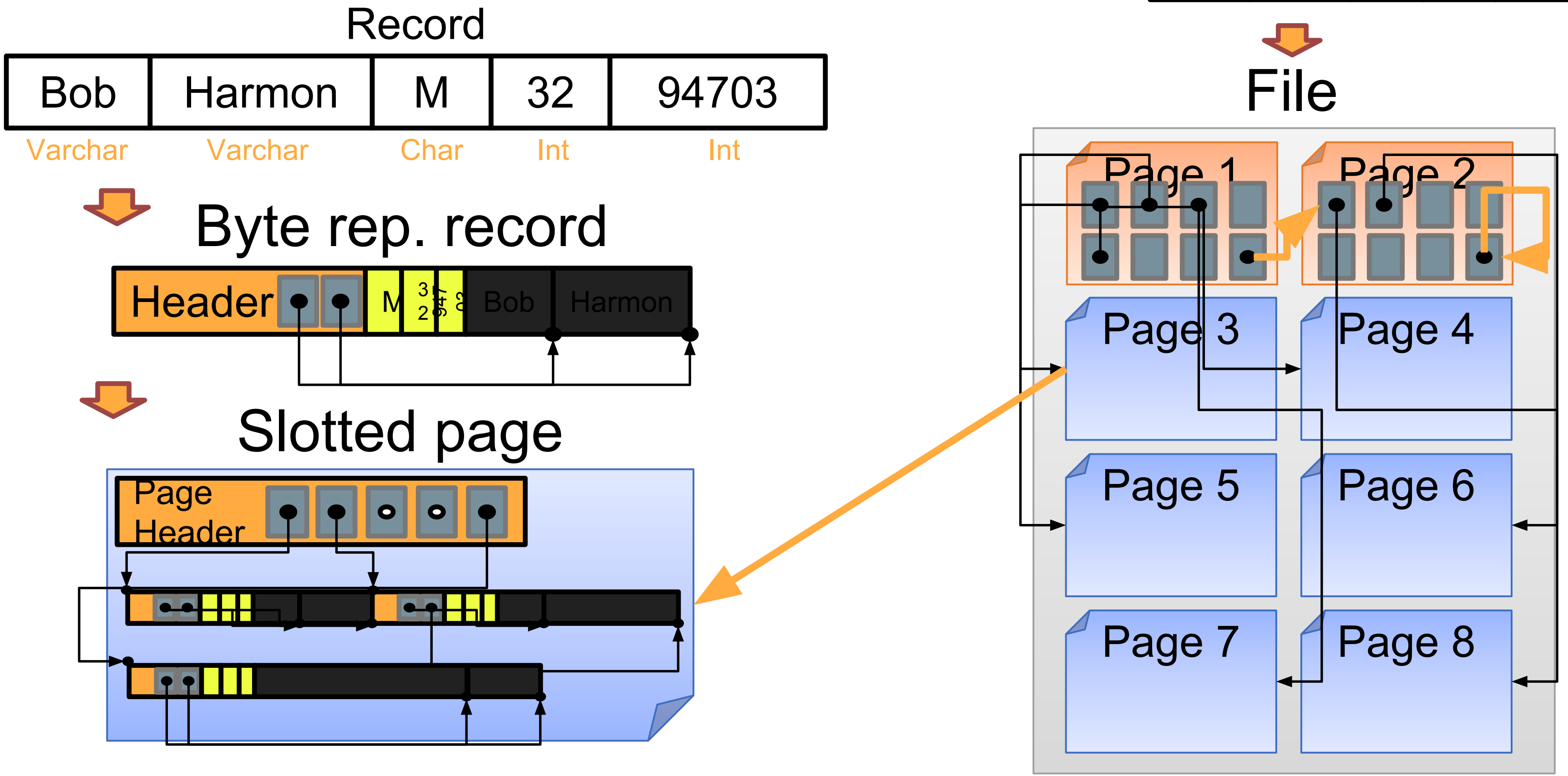
```
SELECT T1.did
FROM Trip AS T1
WHERE 5.0 = ALL (
    SELECT T2.driver_rating
    FROM Trip AS T2
    WHERE T1.did = T2.did AND driver_rating <> NULL );
```

Files and Buffer Management

File of pages, page of records

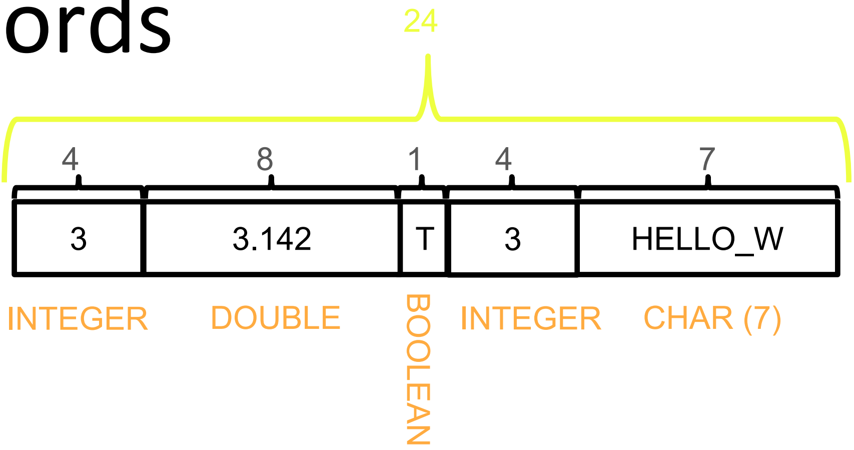
Table

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alice	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110

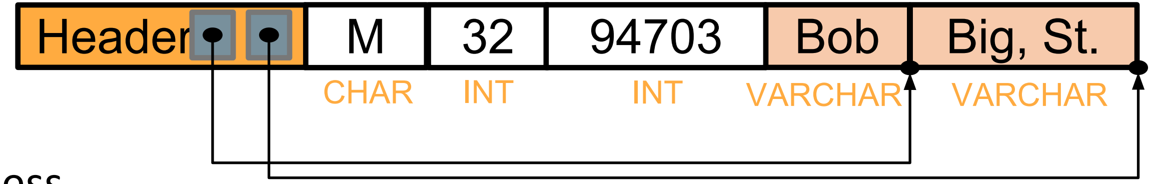


File of pages, page of records

- Fixed length
  - Fast access to any field via arithmetic
  - Wasteful of space

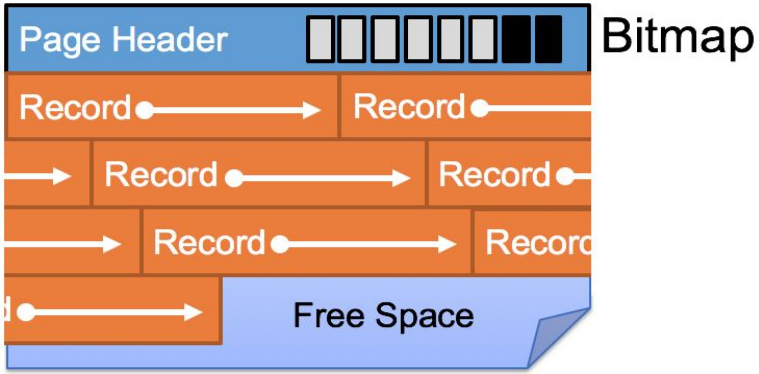


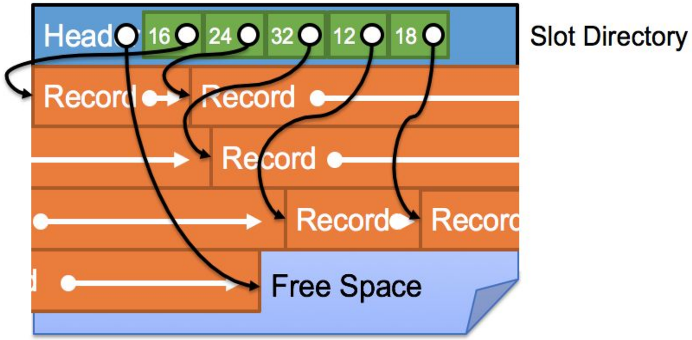
- Variable length (few options)
  - Delimiters
  - Field lengths
  - **Header record:** direct access to any field



Records on a page

- Fixed length (unpacked)
  - Use a bitmap to indicate which slots are free
- Variable length
  - Use a slot directory to store length and pointer to beginning of record

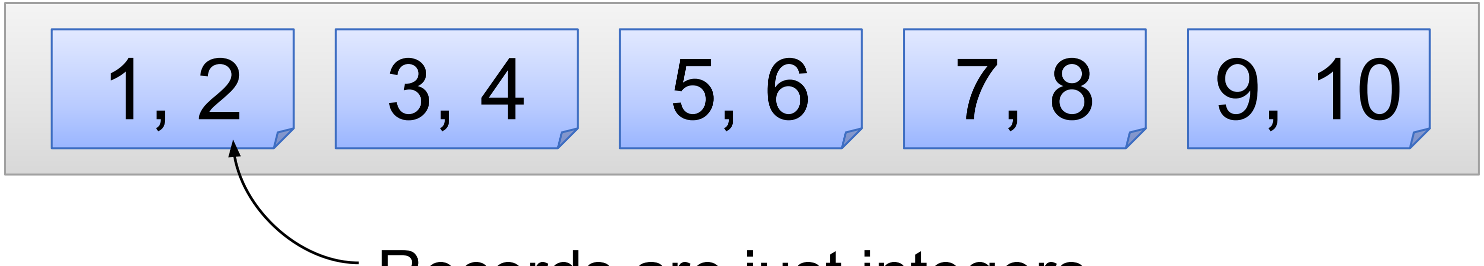




Heap & sorted files

Heap File

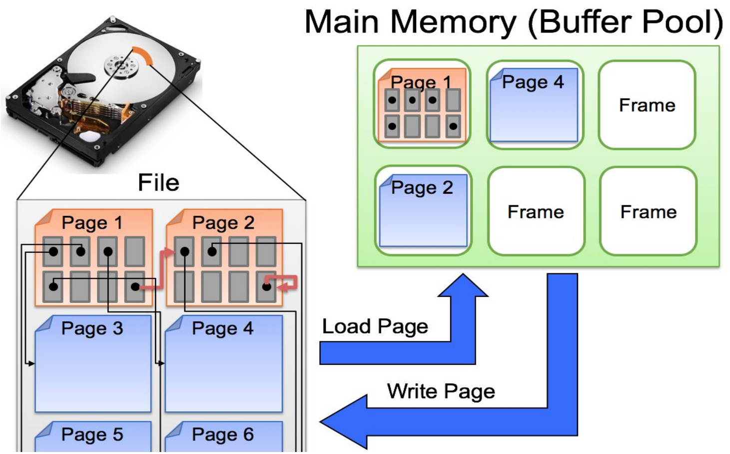
Sorted file



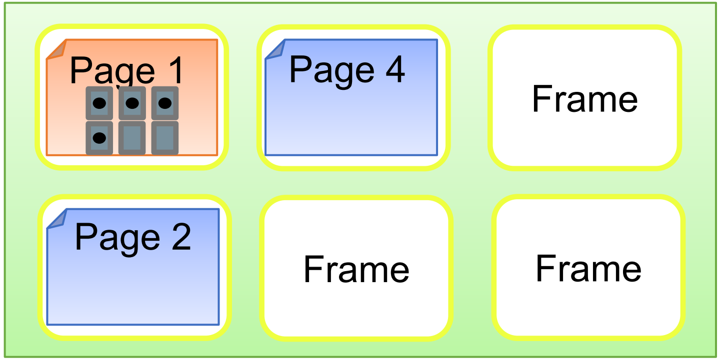
Records are just integers

Mapping pages into memory

- DB operates on in-memory pages
- Buffer manager hides the fact not all data in RAM
- Challenges:
  - No free frames → evict which page?
  - What if page is being used?



When a page is requested...



- Buffer pool information "table" contains: <frame#, pageid, pin_count, dirty>

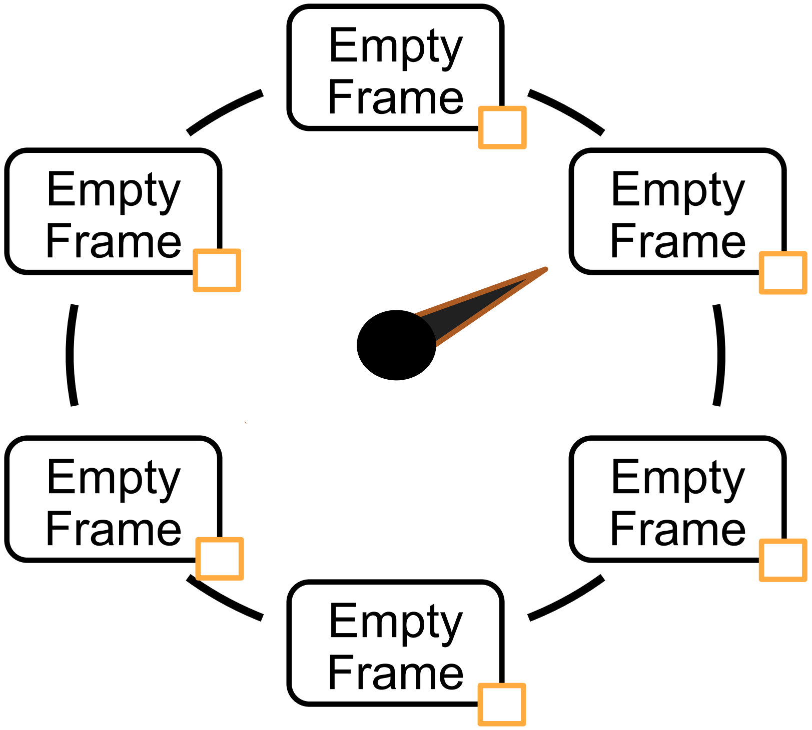
1. If requested page is not in pool:
   a. Choose a frame for *replacement*
   b. If frame "dirty", write current page to disk
   c. Read requested page into frame
2. *Pin* the page (pin_count++) and return its address

Buffer replacement policy

- We choose replacement policy based on *access pattern*, to *minimize I/Os*
- Least-recently-used (LRU)
  - Good temporal locality: heap data-structure can be costly
- Most-recently-used (MRU)
  - Handles sequential flooding well
- Clock
  - Approximation to LRU, simpler data structure

Clock replacement policy

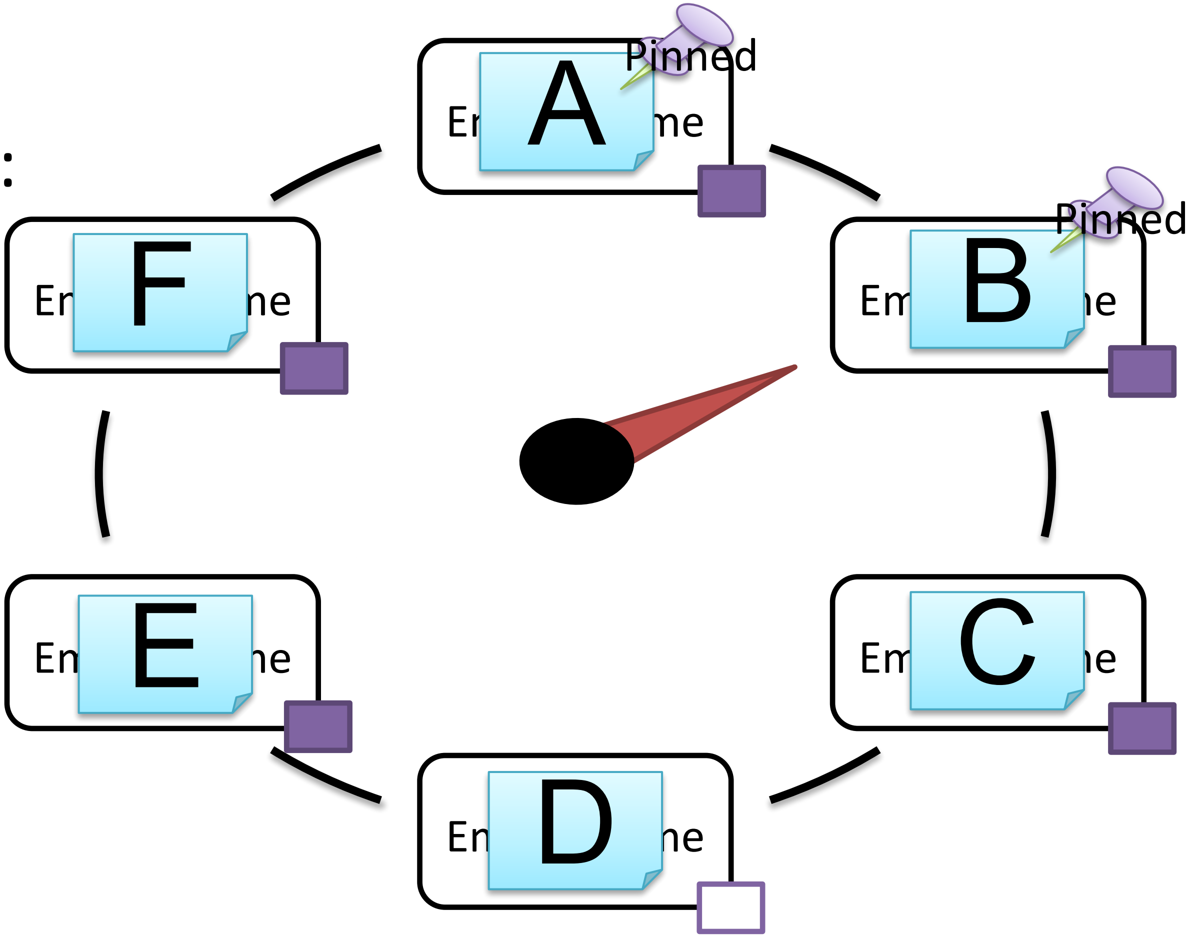
- Arrange frames in logical cycle
- Introduce “reference bit”
  - aka second chance bit



Want to insert page:

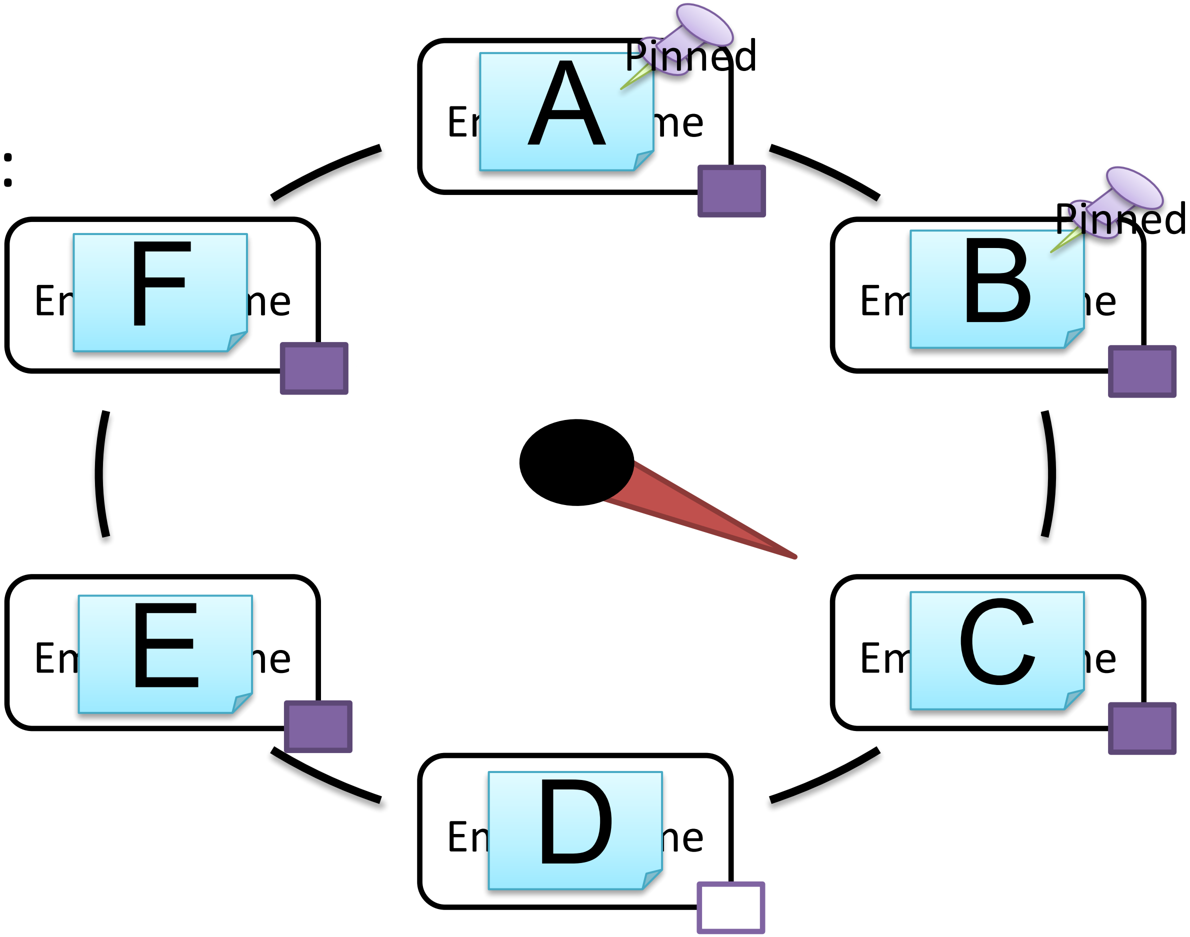
G

Current frame has pin-count > 0
→ Skip



Want to insert page:

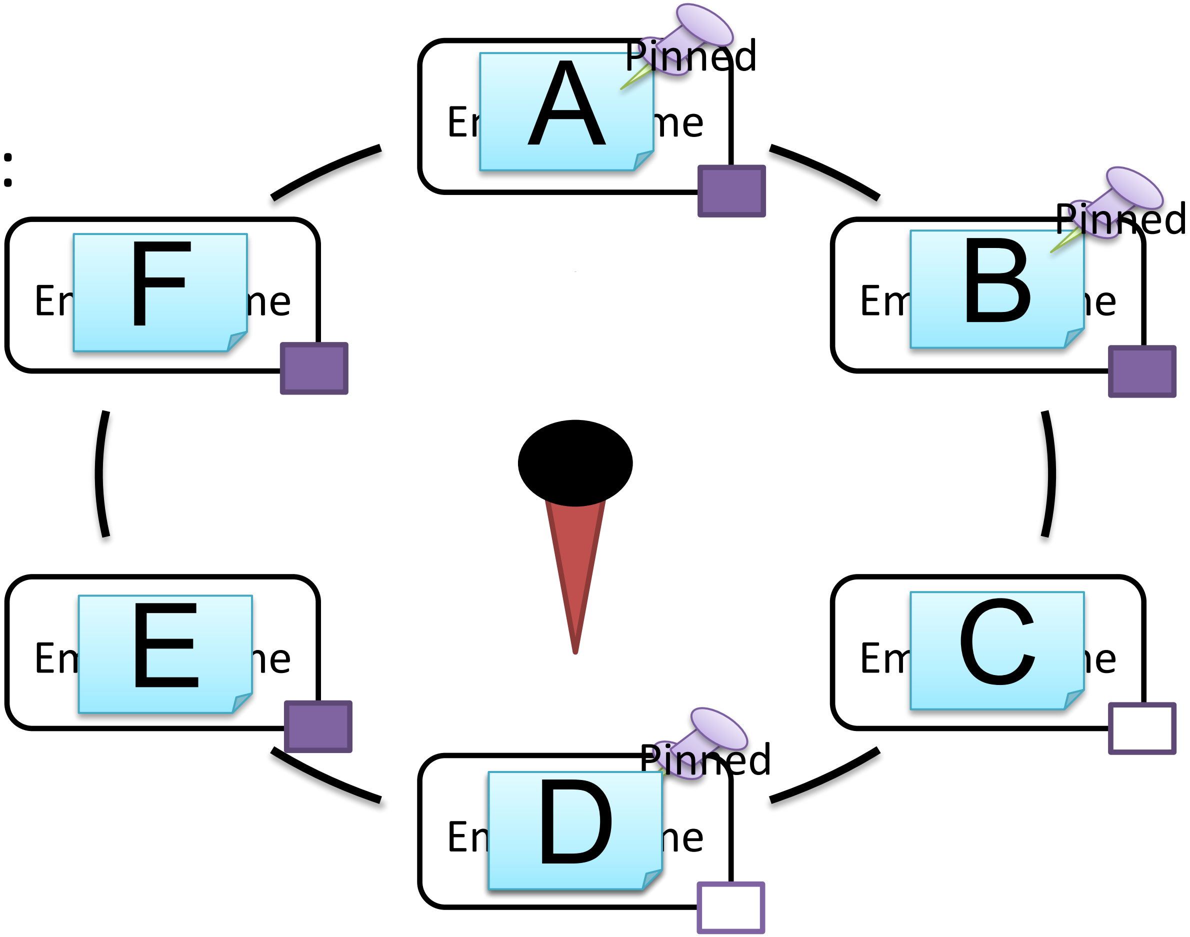
G



Not Pinned and Ref. bit is set →

1. Clear Ref. Bit
2. Skip

Want to insert page:



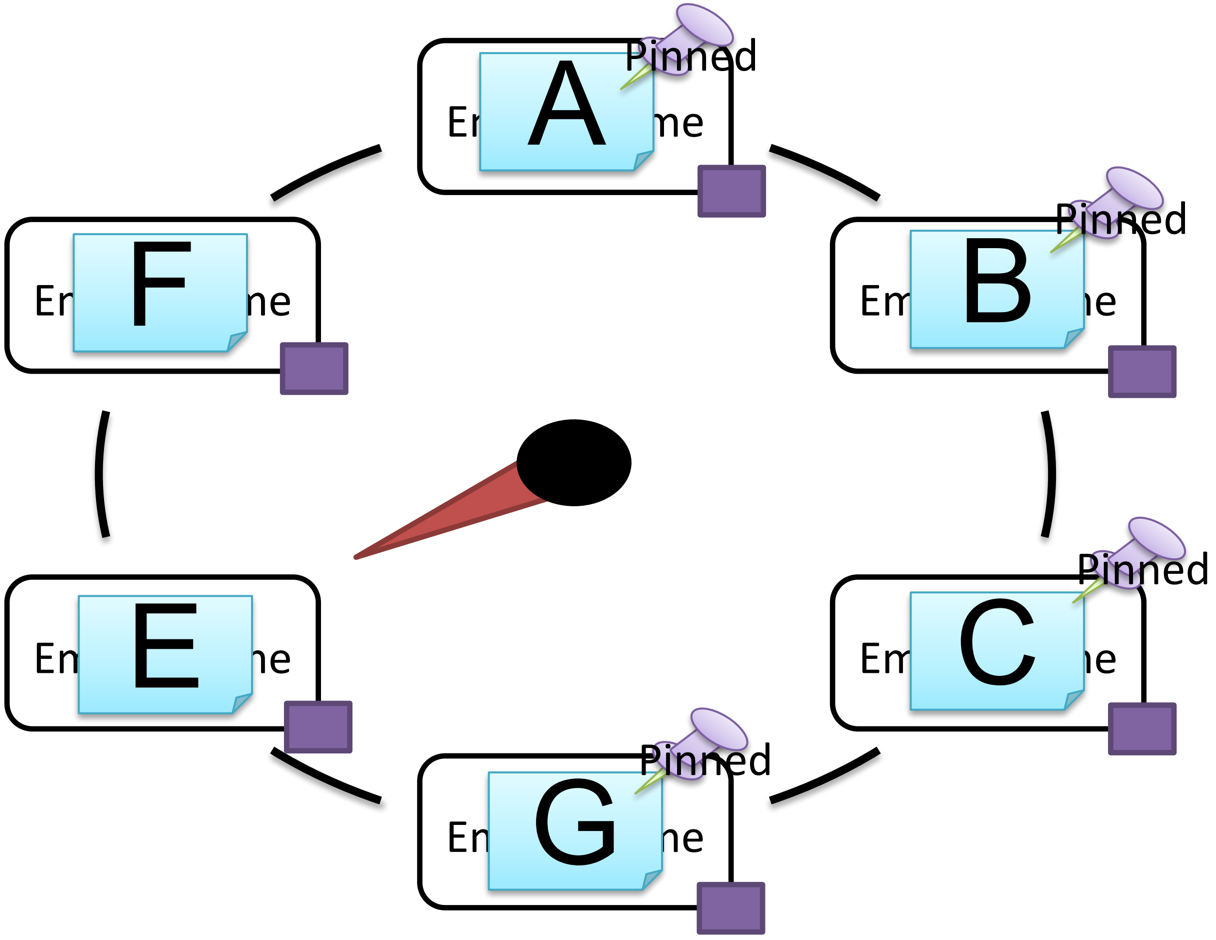
Not pinned and
Ref. bit unset:

1. Evict
2. Copy Page
3. Set pinned

Request for Page C:

Cache Hit!:

1. Pin (inc.)
2. Set ref bit.



Example Buffer Management: Sp14 Mt1 III

1. **Given 4 buffer pages and an access pattern of pages:**
   a. **I, L, O, V, E, D, B, Y, I , P, P, E**
   b. **Which pages are in the buffer pool at the end if we used an MRU cache policy?**

2. **Given 4 buffer pages and an access pattern of pages:**
   a. **A, B, T, P, H, A, C, N, M, O, A, A, D, E, A, B, C, B, E, A, F, G, H, A, C, N, M, O, A, T, P, H**
   b. **Which pages are in the buffer pool at the end if we used a LRU cache policy?**

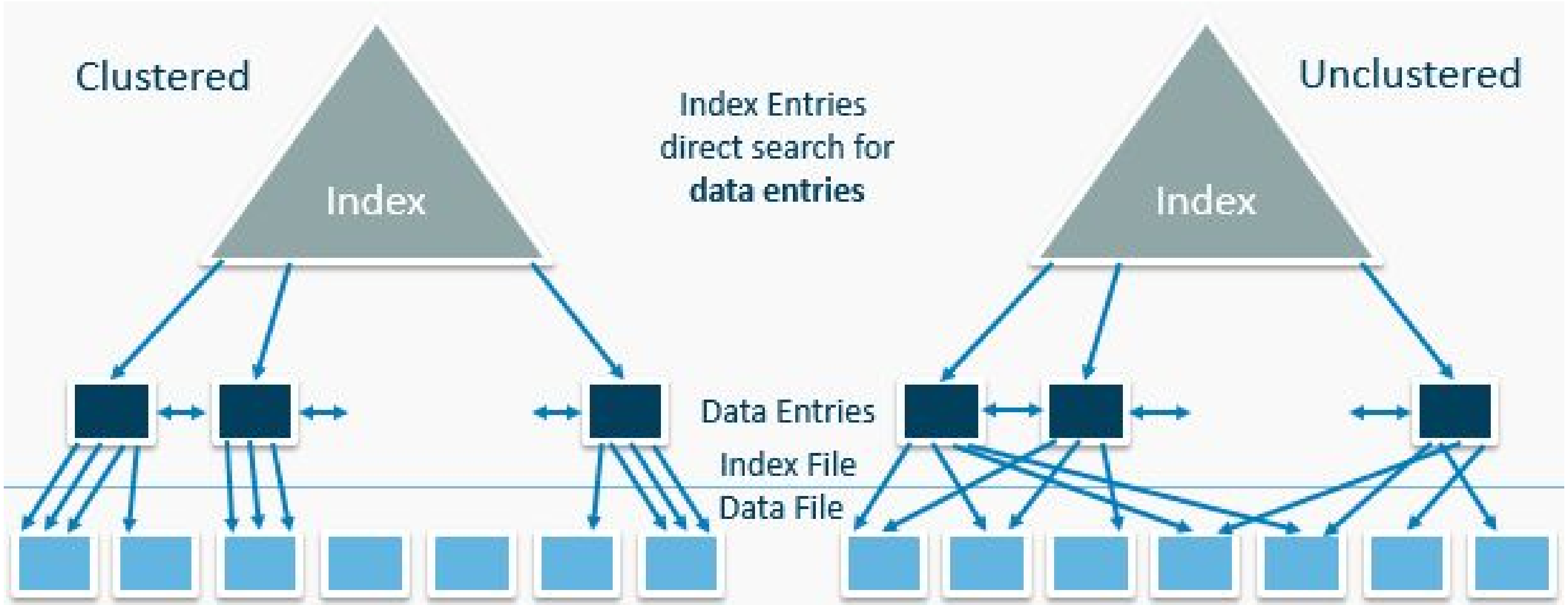
Example Buffer Management: Sp14 Mt1 III

1. **Given 4 buffer pages and an access pattern of pages:**
   a. **I, L, O, V, E, D, B, Y, I , P, P, E**
   b. **Which pages are in the buffer pool at the end if we used an MRU cache policy?**
   c. **ELOY**
2. **Given 4 buffer pages and an access pattern of pages:**
   a. **A, B, T, P, H, A, C, N, M, O, A, A, D, E, A, B, C, B, E, A, F, G, H, A, C, N, M, O, A, T, P, H**
   b. **Which pages are in the buffer pool at the end if we used a LRU cache policy?**
   c. **ATPH - LRU must hold the most recently used pages**

Indexes/B+ Trees

Indexes Overview

- Index: a data structure that enables fast lookup of data entries by search key



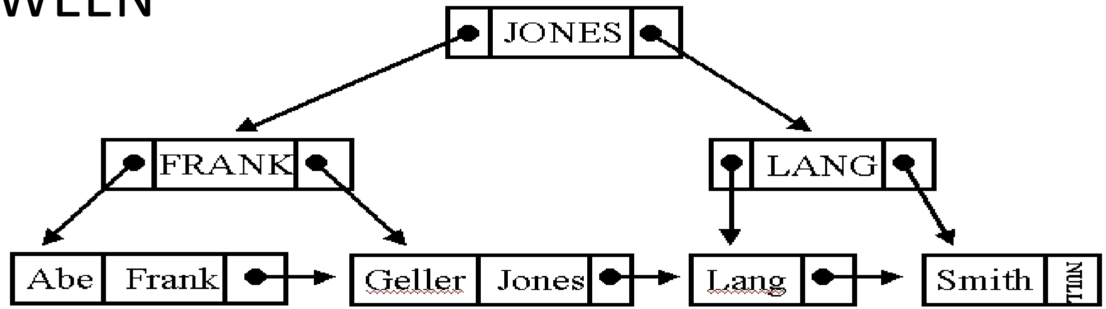
How data is stored in the index

- **k –** the search key, (a subset of the table's columns)
  - e.g. date of birth, (lastname, firstname)

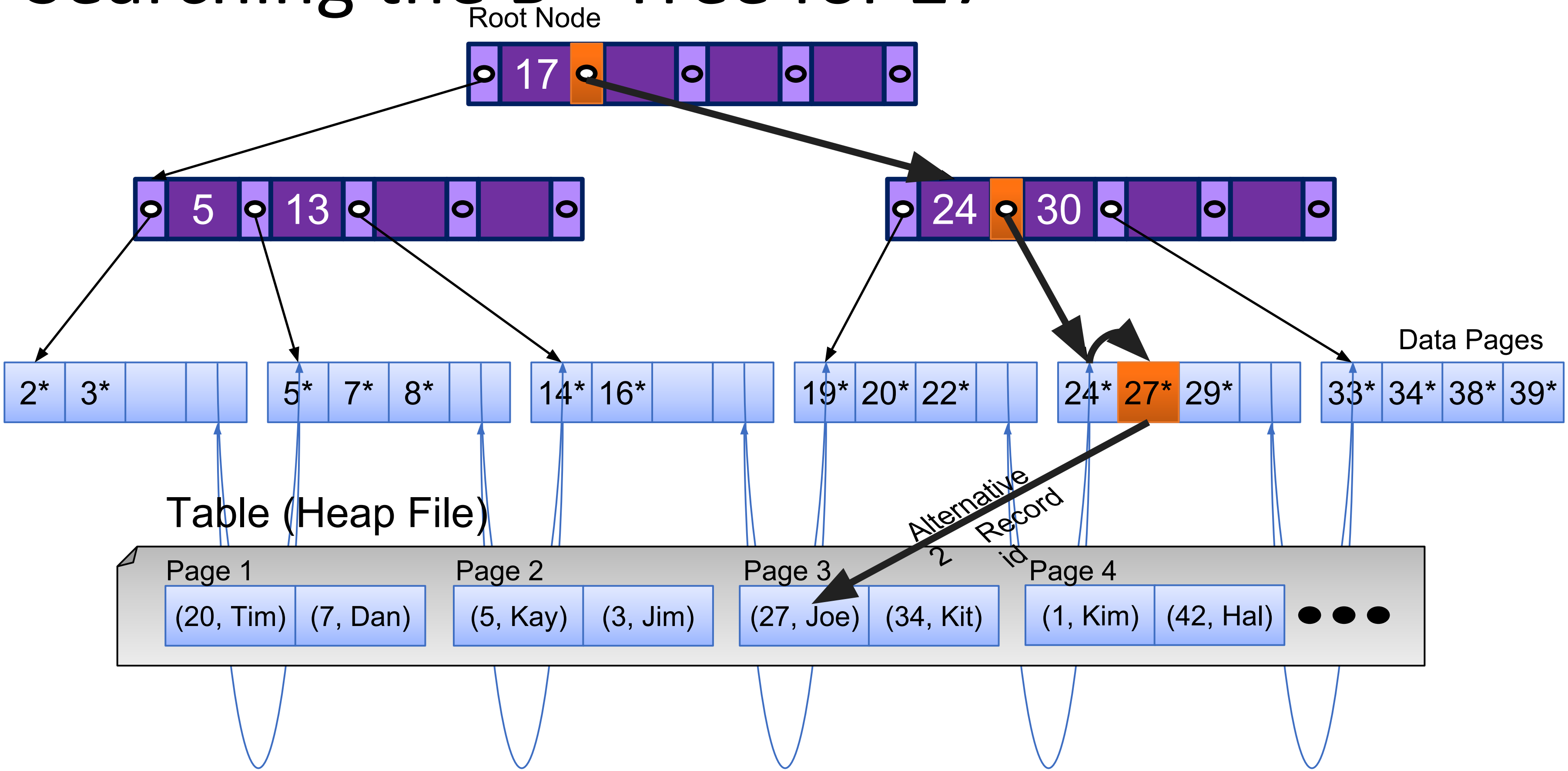
- Three alternatives:
  - **By value:** actual data record (with key value **k**)
  - **By reference:** <**k**, rid of matching data record>
  - **By list of refs.:** <**k**, list of rids of *all* matching data records>

B+ tree indices

- B+ Tree efficiently (logarithmic) supports both:
  - Equality selections =
  - Range selections
    - <, >, <=, >=, BETWEEN

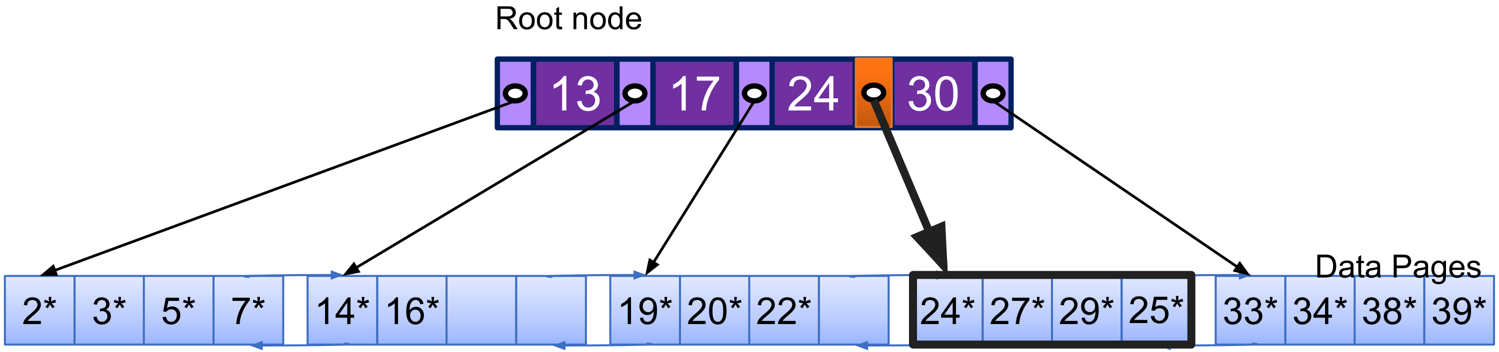


Searching the B+-Tree for 27



Inserting 25* into a B+-Tree

- Find the correct leaf
- If there is room in the leaf just add the entry
  - Sort the page leaf page by key

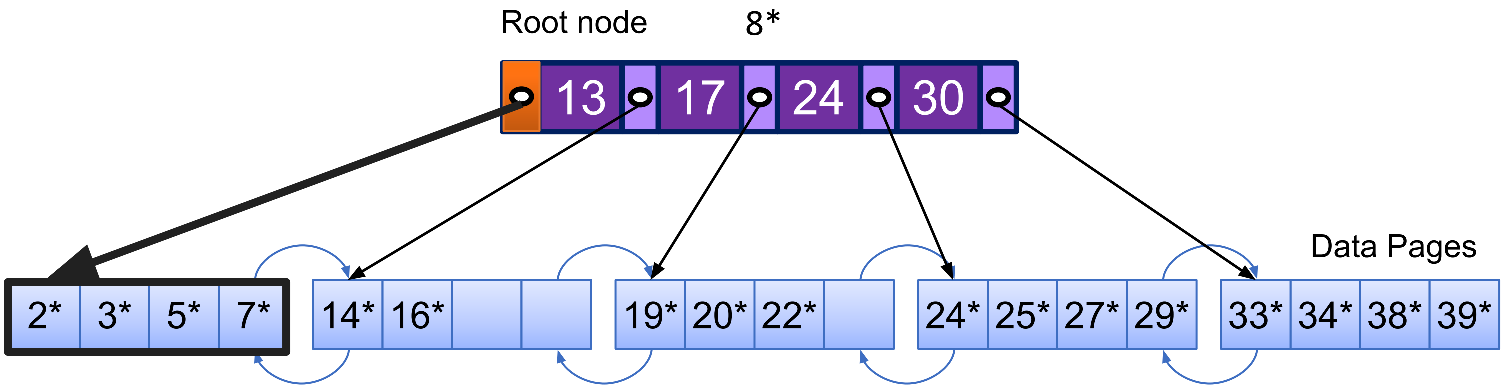


That was easy!

(but there's more)

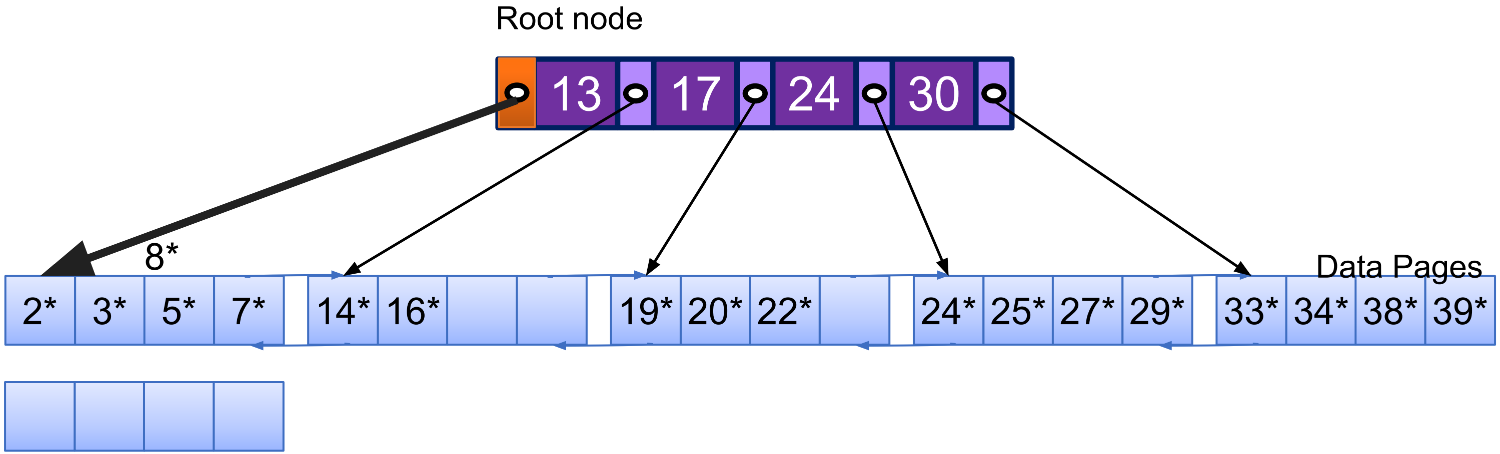
Inserting 8* into a B+-Tree

- Find the correct leaf



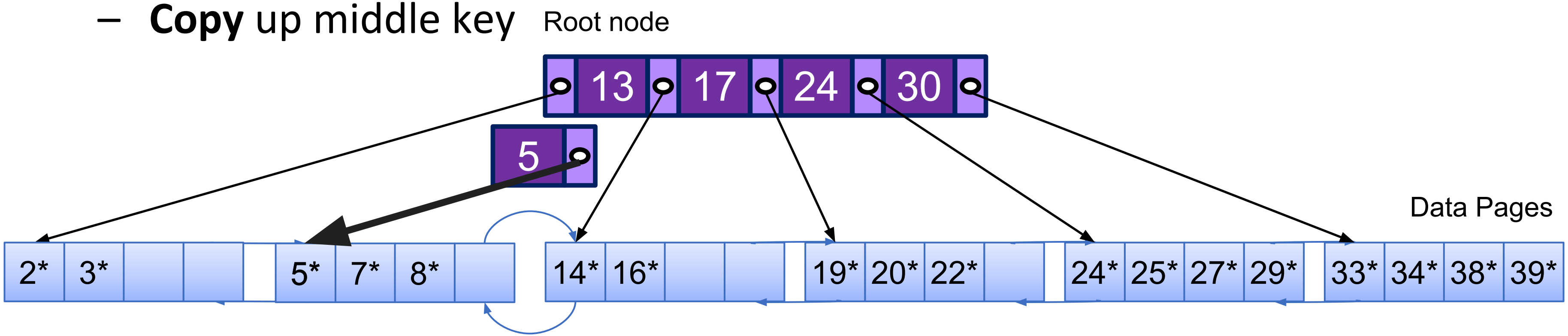
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly



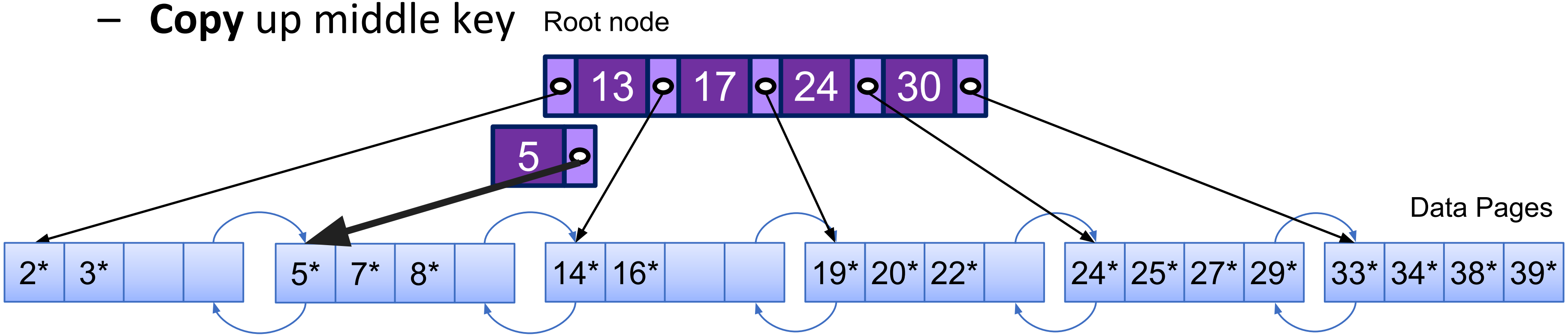
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key



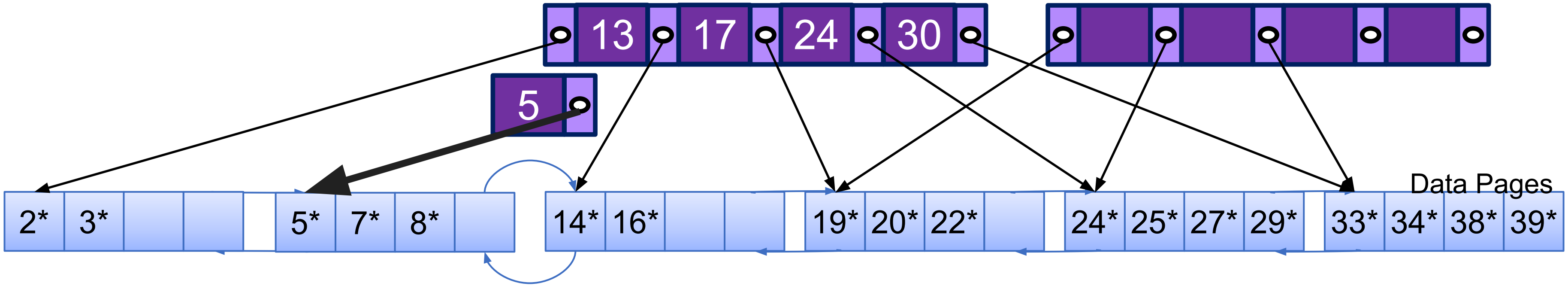
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key



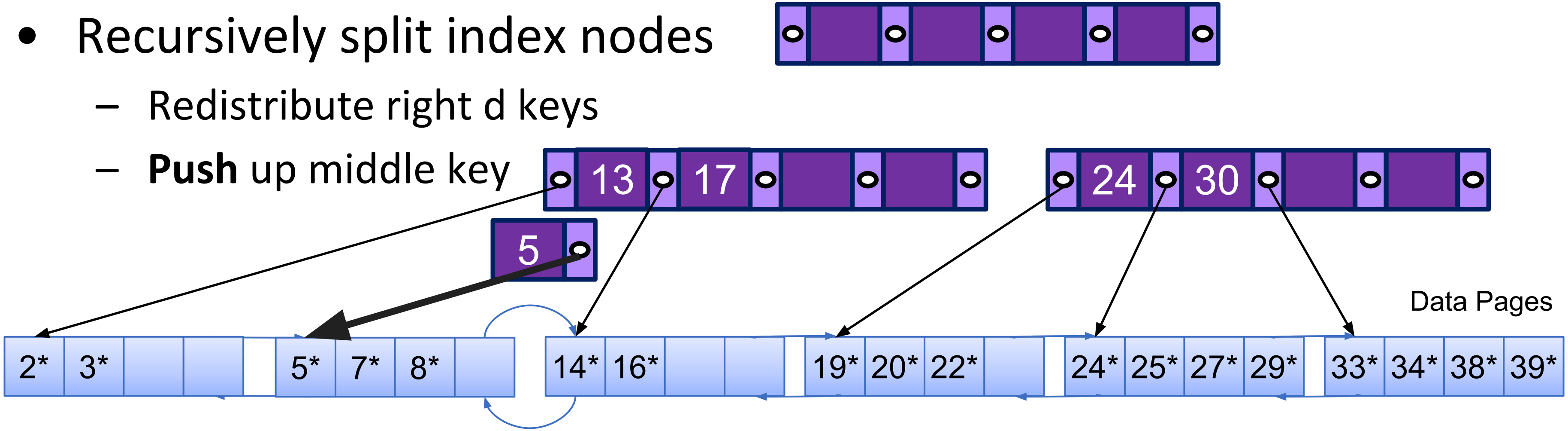
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys



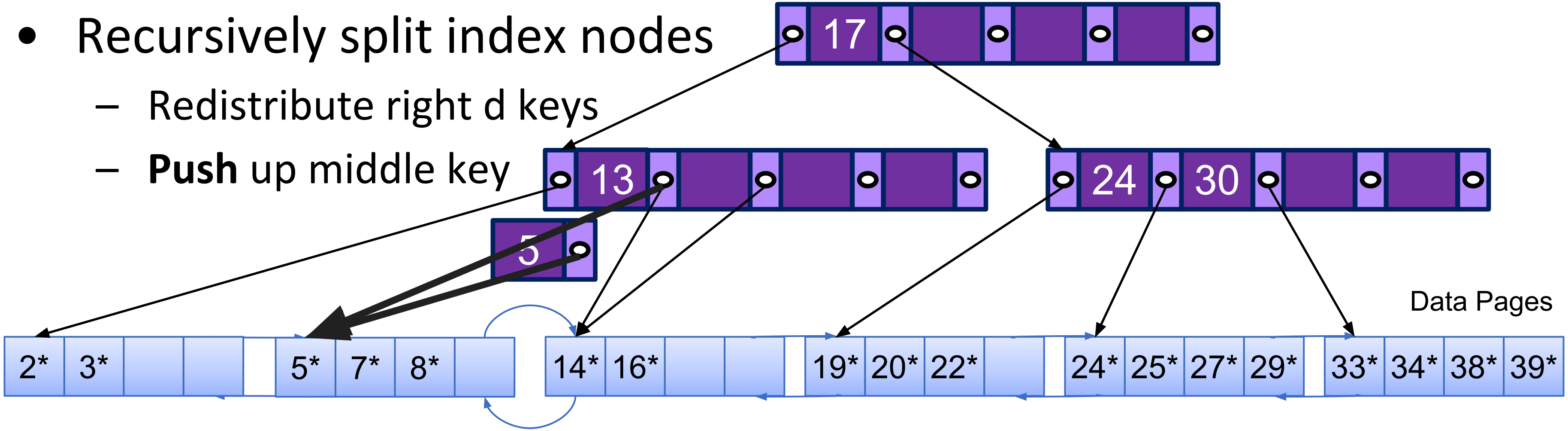
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key



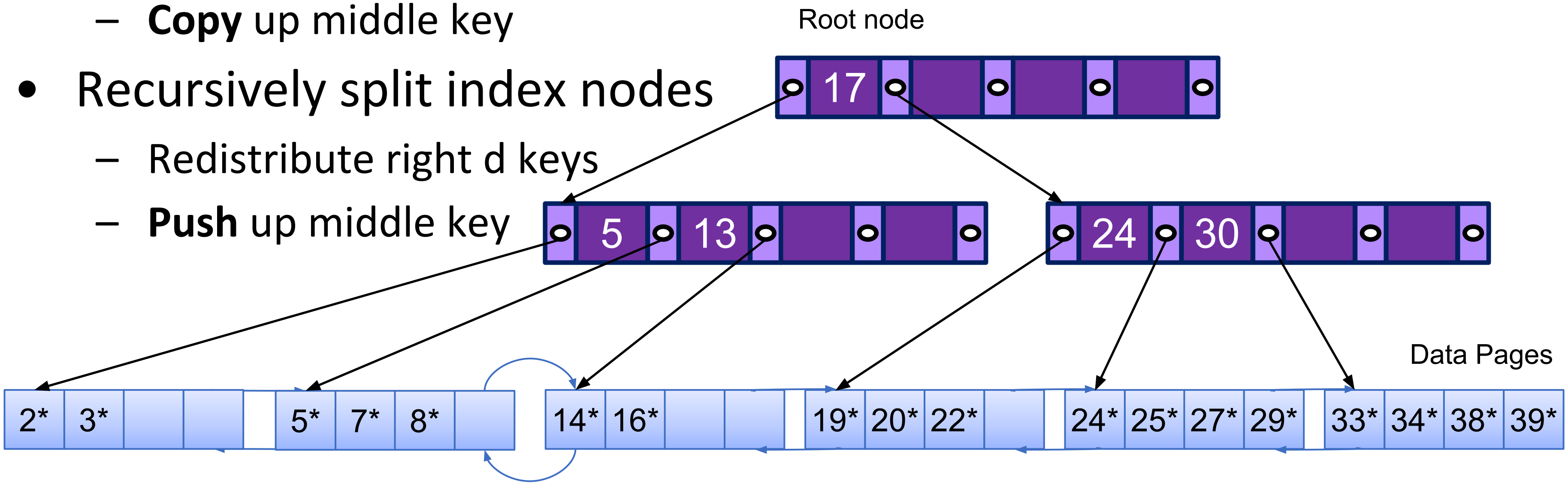
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key

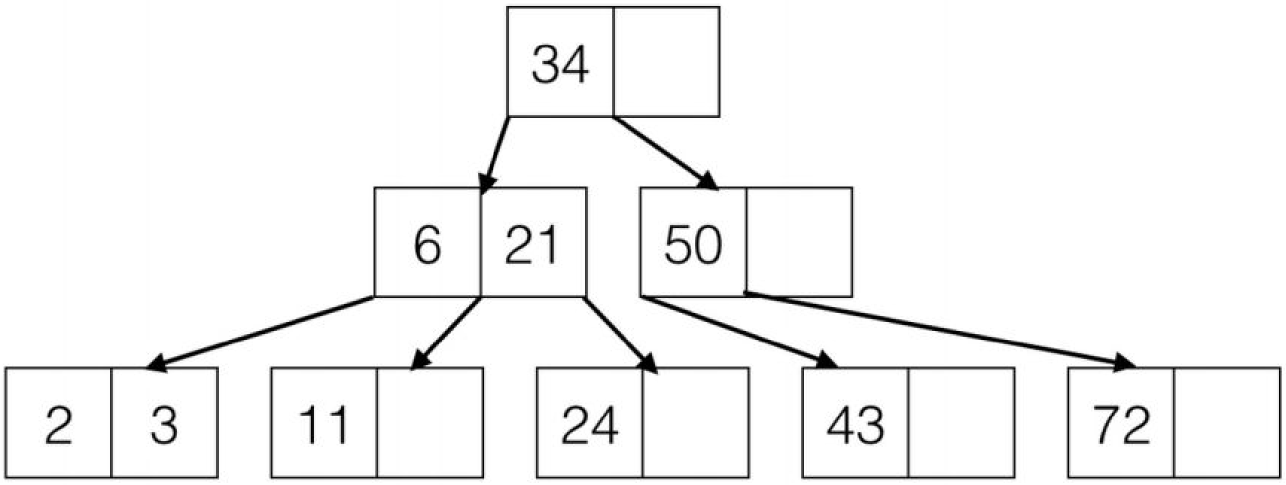


Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key

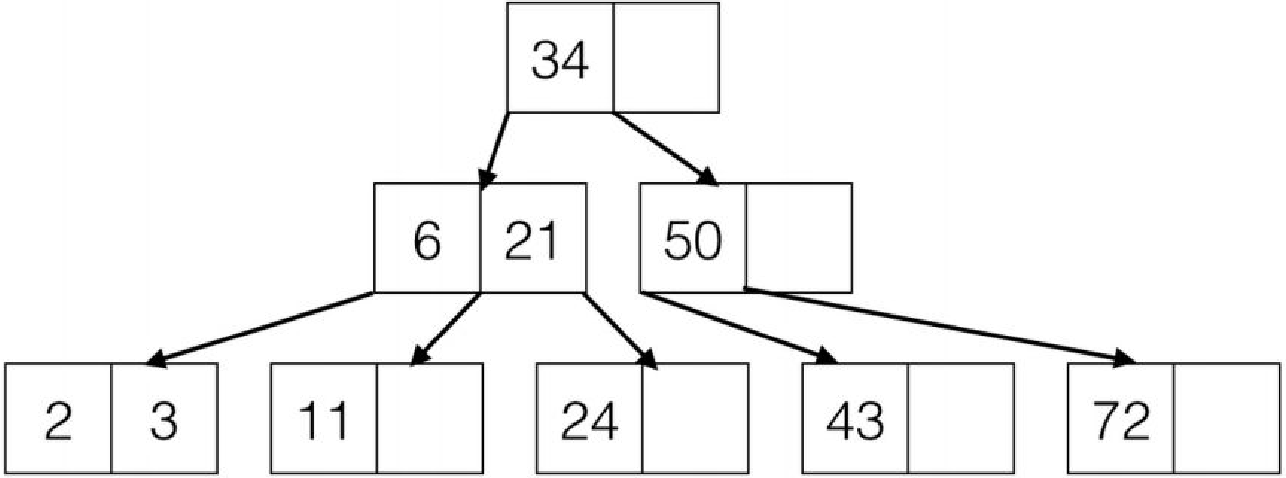


Example B+-tree: Sp15 Mt1 IV Pt 2



What's the minimum number of keys you could insert to change the height of the above tree?

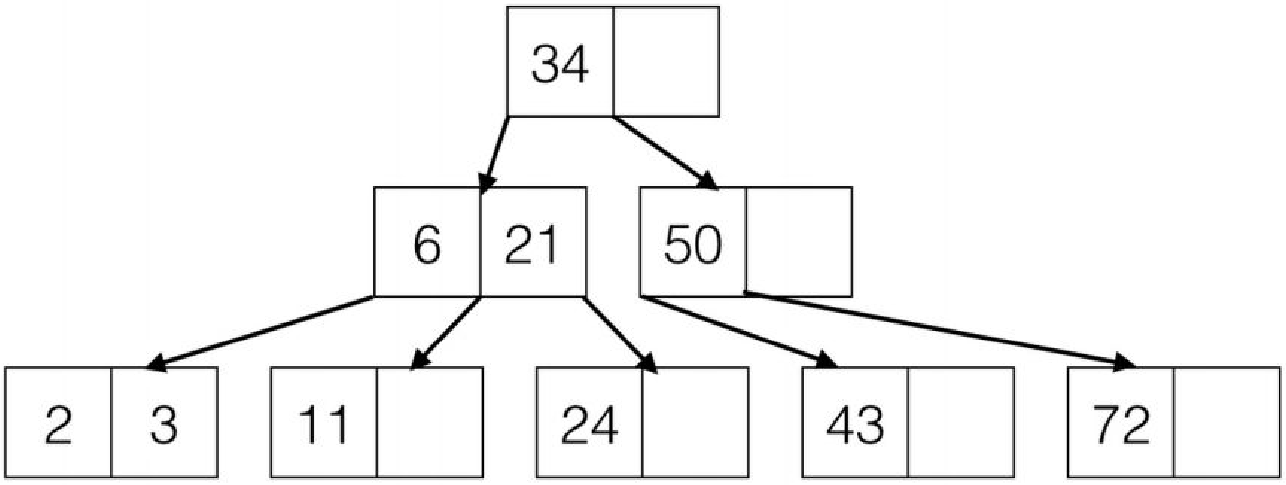
Example B+-tree: Sp15 Mt1 IV Pt 2



3, possible insertions: 1, 4, 5

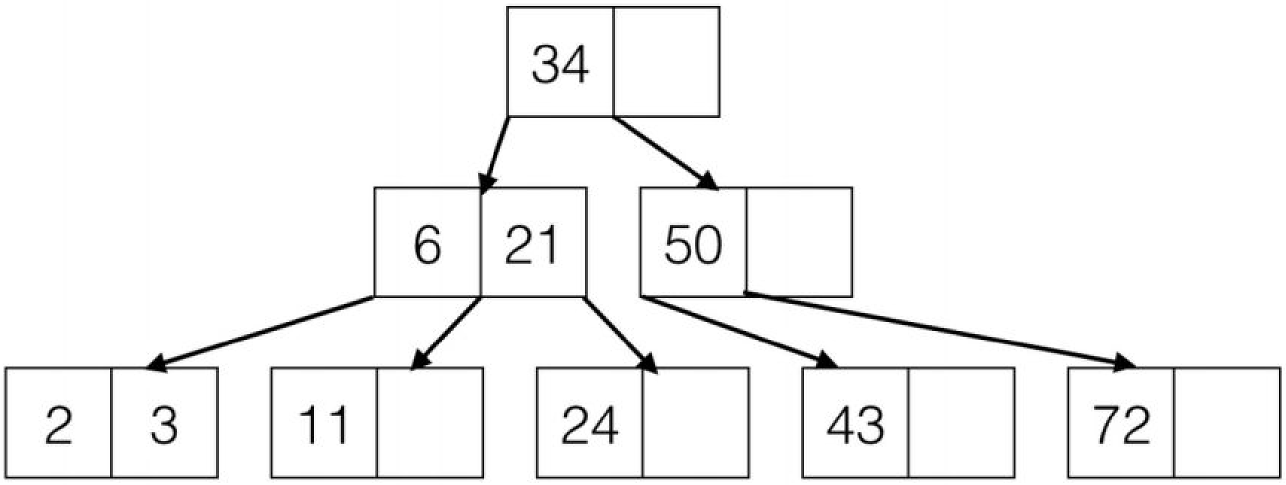
What's the minimum number of keys you could insert to change the height of the above tree?

Example B+-tree: Sp15 Mt1 IV Pt 1



What's the maximum number of keys you could insert without changing the height of the above tree?

Example B+-tree: Sp15 Mt1 IV Pt 1



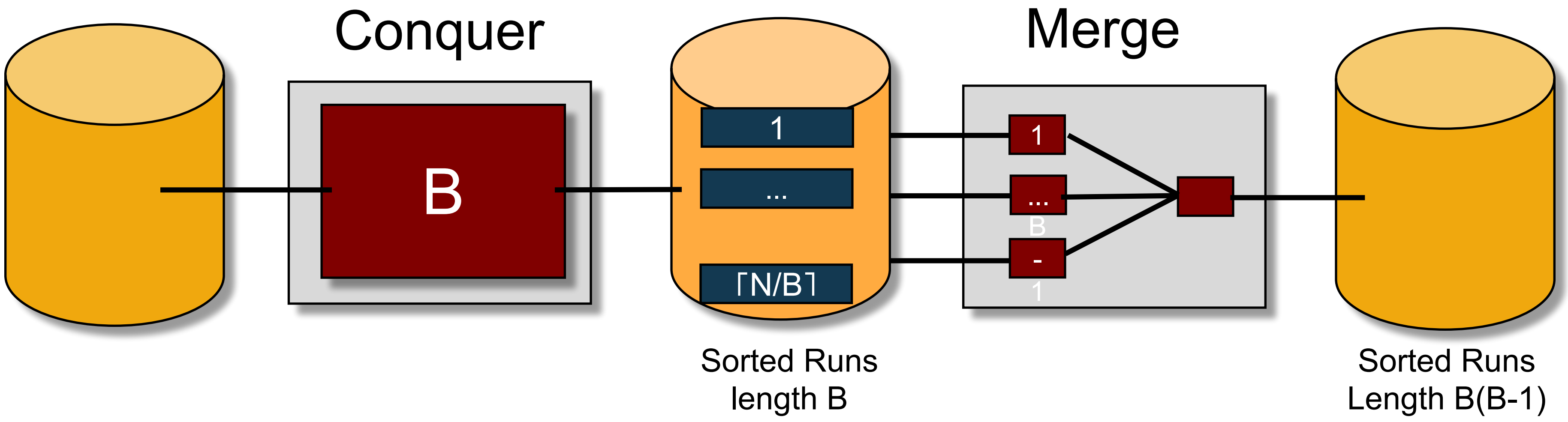
12, possible insertions: 12, 25, 42, 73, 41, 40,74, 71, 39, 38, 70, 69

What's the maximum number of keys you could insert without changing the height of the above tree?

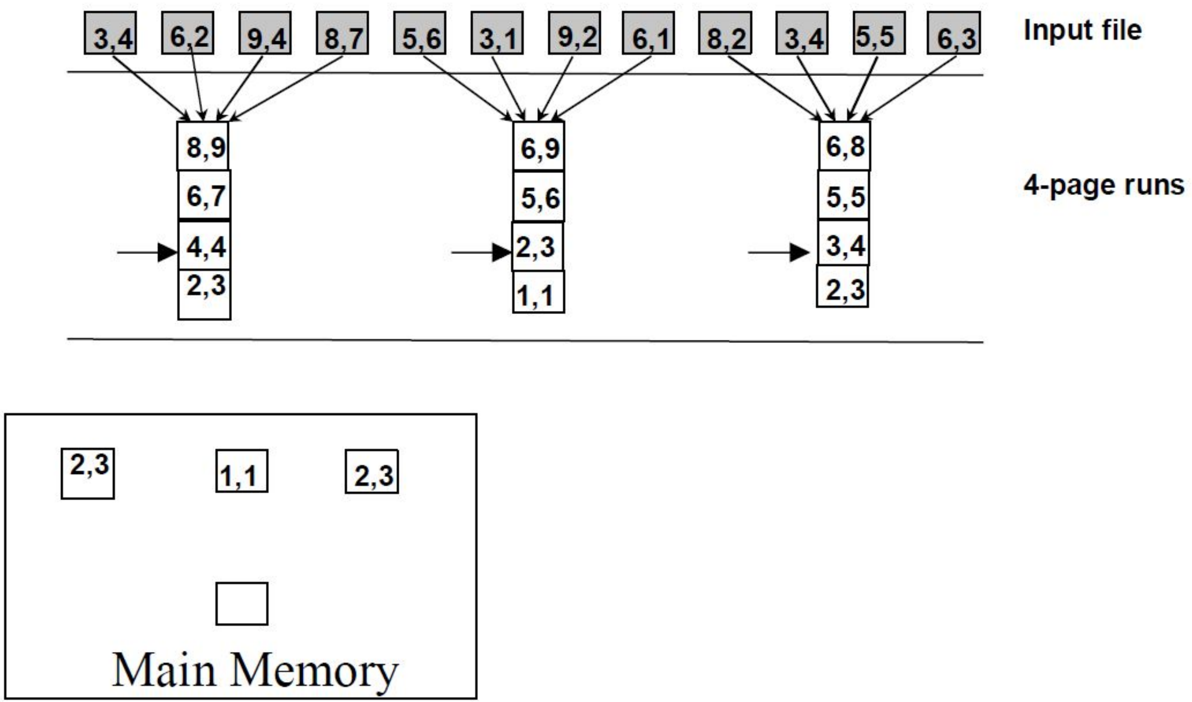
Sorting and Hashing

General External Merge Sort

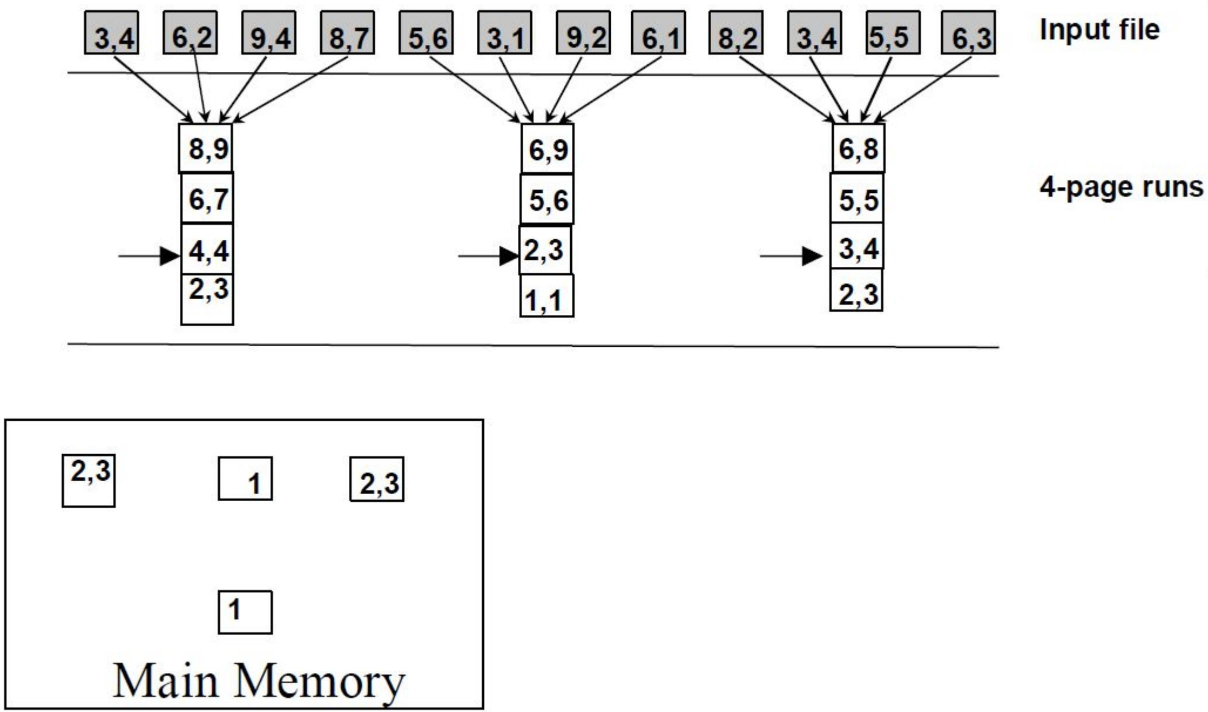
- To sort a file with N pages using B buffer pages:
  - Pass 0: use B buffer pages. Produce ⌈N/B⌉ sorted *runs* (groups of pages) of B pages each.
  - Pass 1, 2, …, etc.: merge B-1 runs at a time.



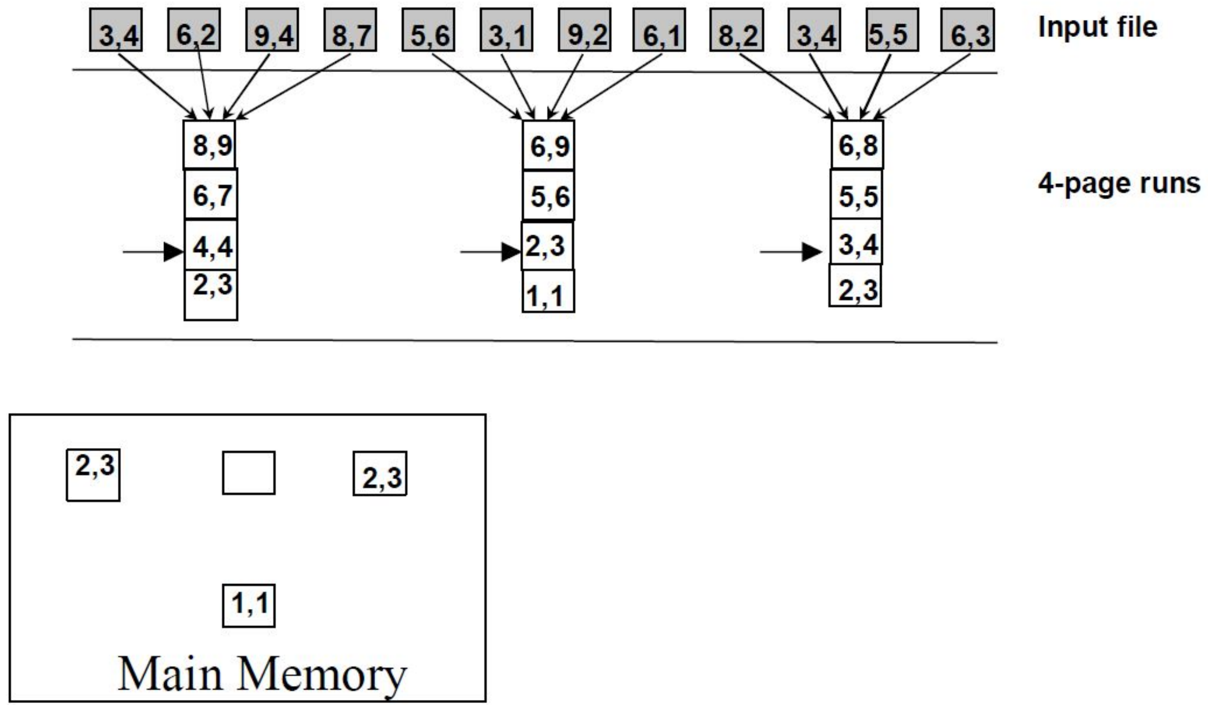
Example: B = 4, N = 12



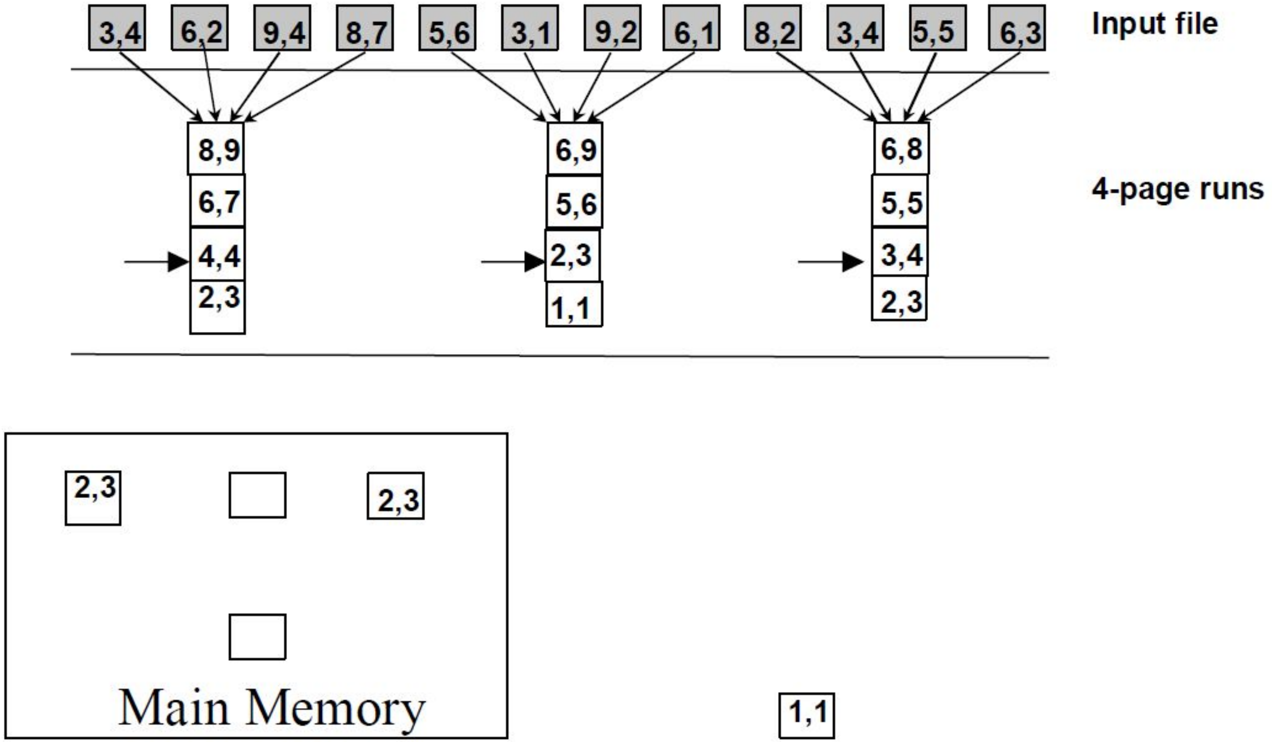
Using the Output Buffer



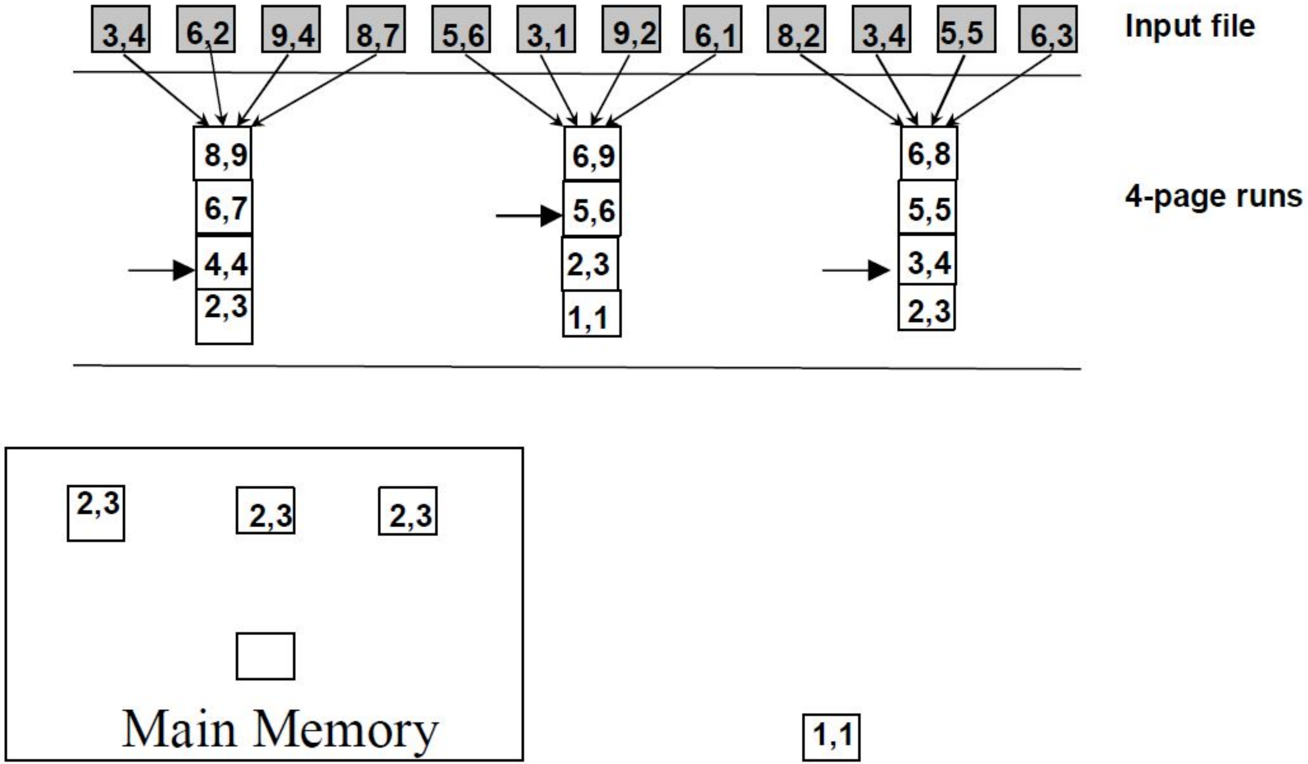
Using the Output Buffer



Clearing the Output Buffer: Writing to Disk



Bringing in the next page



Cost of External Merge Sort

- Breaking down the number of passes formula
  - **1 + ⌈log$_{B-1}$ ⌈N/B⌉ ⌉**
  - 1: Pass 0
  - ⌈N/B⌉: number of sorted runs after Pass 0
  - log$_{B-1}$: In each of Pass 1, 2, 3,..., can merge B-1 *sorted runs of any length*
- Cost: in each pass we read & write all N pages
  - 2N * (# of passes)

Which one is better?

- In common:
  - Same memory requirement for 2 passes
  - Same I/O cost
- Sorting pros:
  - Great if input already sorted (or almost sorted) w/heapsort
  - Great if need output to be sorted anyway
  - Not sensitive to “data skew” or “bad” hash functions
- Hashing pros:
  - For duplicate elimination, scales with # of values
  - Load balances in parallel case easily

Hashing Practice

You are a farmer. You have a table, Animals, that stores information about each animal you have. You can store **4 tuples per page**.

You want to group the animals by their type. Since you don't care about any form of ordering, you decide to **hash** the animals into groups. You have **101 buffer pages** available and there are **32,000 animals**.

Sp2012 MT1, QI

Hashing Practice

4 tuples per page; **101 buffer pages; 32,000 animals**.

1) How many times to we have to run the Partitioning stage of hashing to hash the animals, assuming all partitions end up being the same length? How long will each partition be?

Hashing Practice

4 tuples per page; **101 buffer pages; 32,000 animals**.

1) How many times to we have to run the Partitioning stage of hashing to hash the animals, assuming all partitions end up being the same length? How long will each partition be?

   32,000 animals / 4 animals per page = 8000 pages.

   After one pass of partitioning, each partition will be 8000 / (101-1) = **80 pages**. Since 80 < 101, each partition will be able to fit during the rehash stage.

   Therefore the answer is **1 time**.

Hashing Practice

4 tuples per page; 101 buffer pages; 32,000 animals.

3) It's the end of the world and we can only keep 1 of each animal. Fortunately we only have *X* unique animal types. We want to use hashing to get **exactly one** of each of the *X* types of animals. What is the maximum value of *X* that we can handle using only the partition phase and **guarantee we don't see any duplicate animals**?

Hashing Practice

4 tuples per page; 101 buffer pages; 32,000 animals.

3) It's the end of the world and we can only keep 1 of each animal. Fortunately we only have *X* unique animal types. We want to use hashing to get **exactly one** of each of the *X* types of animals. What is the maximum value of *X* that we can handle using only the partition phase and **guarantee we don't get any duplicate animals in the partitions**?

output buffers * tuples per page = (101-1) * 4 = **400**

Explanation: we stream the records in a page at a time, and every time we see a duplicate we throw it out. (We can tell if an animal type is a duplicate by hashing and comparing it to the records already in the output buffer.) If we haven't seen the animal type then we place it in the appropriate output buffer.

Now, the confusing part: The reason we can't guarantee any more than 400 unique animals is that once we fill an output buffer, we write it to disk and clear the page in memory. At this point we cannot be sure that any new records that hash to that output buffer are not duplicates.

For example, let's say we represent an animal's type with an integer `animal_type`, and also our hash function assigns animals to partitions based on `animal_type` % 100.

Let's say that in the course of streaming records and hashing them to output buffers, we see `animal_types` 1, 101, 201, 301. These four records will fill up the output buffer for (`animal_type` % 100 = 1). We will then write this output buffer to disk and clear the records from memory.

Now, what happens if we stream in a record with `animal_type` 101? We would hash it to the same output buffer, which is now empty. How can we be sure we haven't seen `animal_type` 101 before? The answer is we can't. (we actually saw it before, but it's on disk, so we don't know that.)

Therefore, the best we can do is fill each output buffer with four unique animal types.