Hw3 Solutions & Proj3

Discussion 08

Outline

- Hw3 solutions
- Project 3

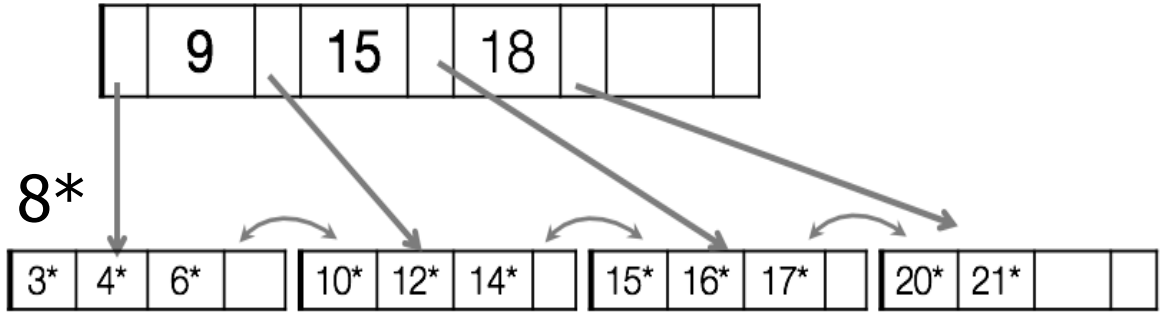
Q1. B+ Tree Operations

- Each node must contain between d <= m <= 2d entries (where m is the number of entries) with the exception of the root node
- The left pointer points to values that are strictly less than the key value.
- During redistribution, favor RIGHT node over LEFT node
- During the splitting of a node, the left node will have one more value than the right node
- For insertions, favor REDISTRIBUTION over SPLITTING
- For deletions, favor MERGING over REDISTRIBUTION
- Only redistribute one entry at a time

Q1

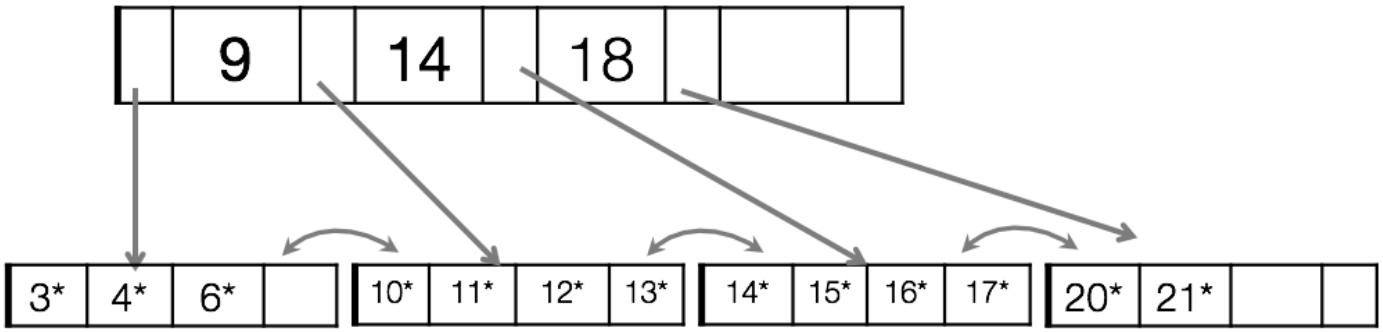
Draw out the resulting B+ tree after EACH of the the following chains of operations is performed on the given B+ tree.

1. Insert 11*, 13*
2. Insert 9*
3. Delete 20*
4. Insert 5*
5. Insert 1*, 2*, 7*, 8*
6. Delete 14*, 15*



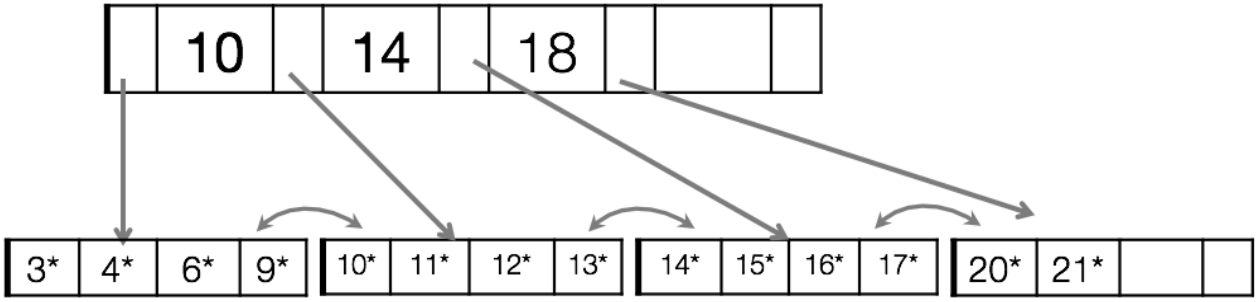
Q1

Insert 11*, 13*



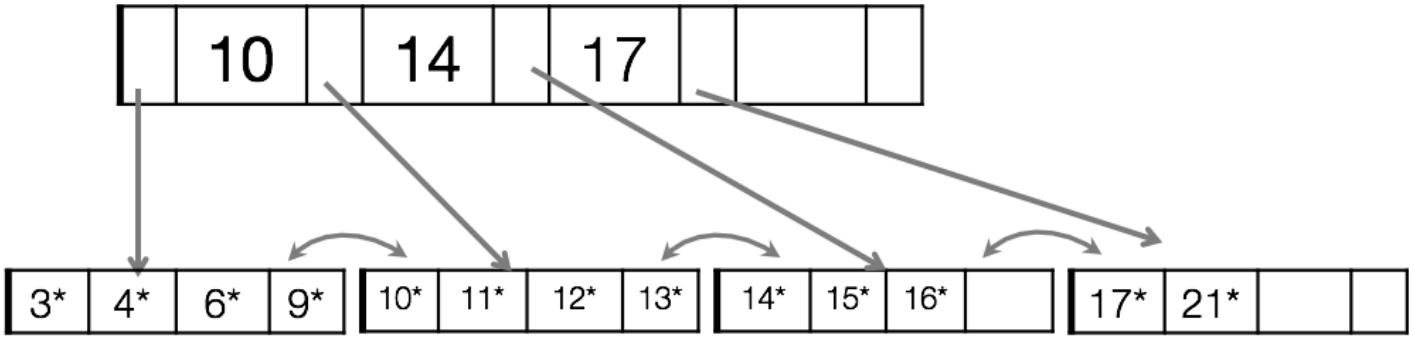
Q1

Insert 9*



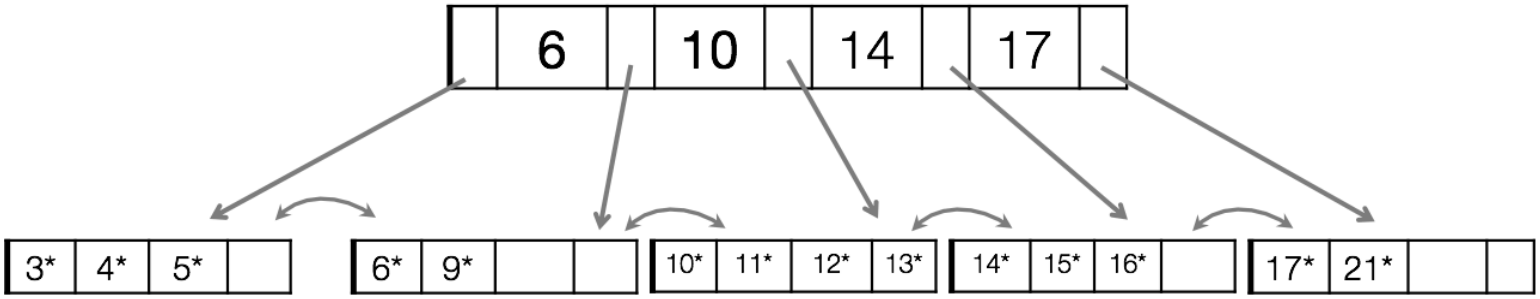
Q1

Delete 20*



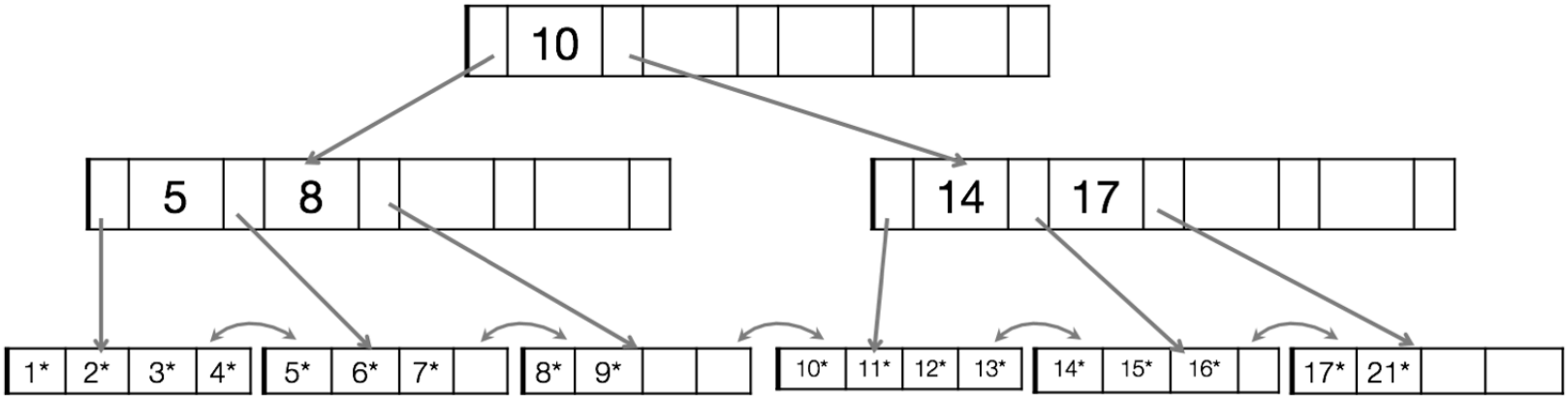
Q1

Insert 5*



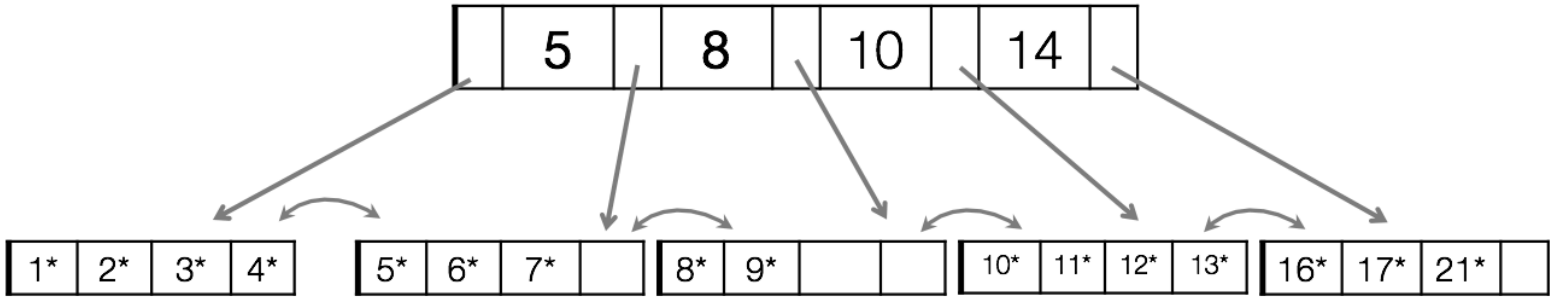
Q1

Insert 1*, 2*, 7*, 8*



Q1

Delete 14*, 15*



Q2: Clustering and IO costs

Consider a table Sailors, where there is a B+ tree index on attribute Age, and the following query:

SELECT * FROM Sailors ORDER BY Age;

Furthermore, assume that the first level (root) of the B+ tree is kept in memory, and that the B+ tree has the following properties:

The properties of this B+ tree are:

- total size of records = 2MB
- size of record = 1KB
- size of page = 4KB
- the tree has an order of 4
- the fill factor is 67%

Q2. Clustering and IO costs

Assume 1MB = 1024KB.

Assume only root node is in memory and nothing else is saved in memory.

We try to fit all records in the smallest number of pages possible and each node is in it's own page.

Assume alternate 2 (as described in lecture) is used.

Q2

1) What is the number of record pages and leaf nodes? What is the height of the B+ tree?

Record pages = 2MB/4KB = 512

Data records = 2MB/1KB = 2048

Leaf nodes =2048/(8*0.67) = 382

Height 1 nodes = 382/(8*0.67) = 71

Height 2 nodes = 71/(8*0.67) = 13

Height 3 nodes = 13/(8*0.67) = 2

Root node = 1

The height is 4.

Q2

2) How many IOs (worst case) will it take to run this query if our index was clustered?

# of IOs = 4(get the first leaf node)
+ 381(get the rest leaf nodes)
+ 512(read data records in sequence)
= 897

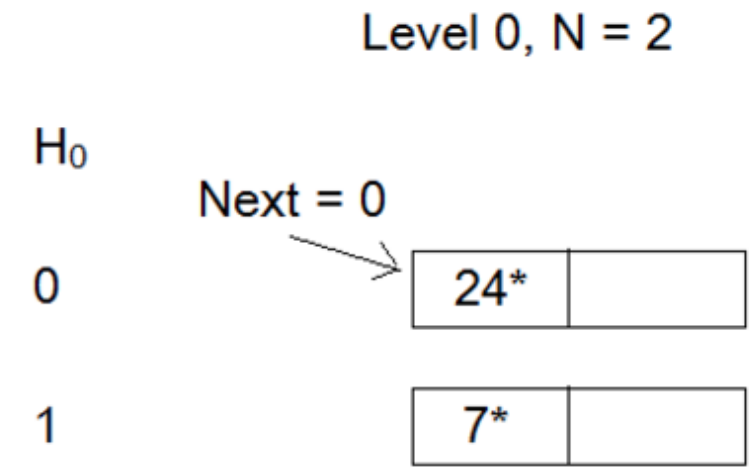
Q2

3) How many IOs (worst case) will it take to run this query if our index was unclustered?

# of IOs = 4(get the first leaf node)
+ 381(get the rest leaf nodes)
+ 2048(worst case)
= 2433

Q3. Linear Hashing

Assume we initialize the Linear Hashing index as follows (the hash function used is given below the Linear Hashing index):



h_i (value) = h (value) mod (2^iN)

Q3

1') What is the minimum number of data entries that can be inserted, until we have 4 buckets and level is equal to 1?

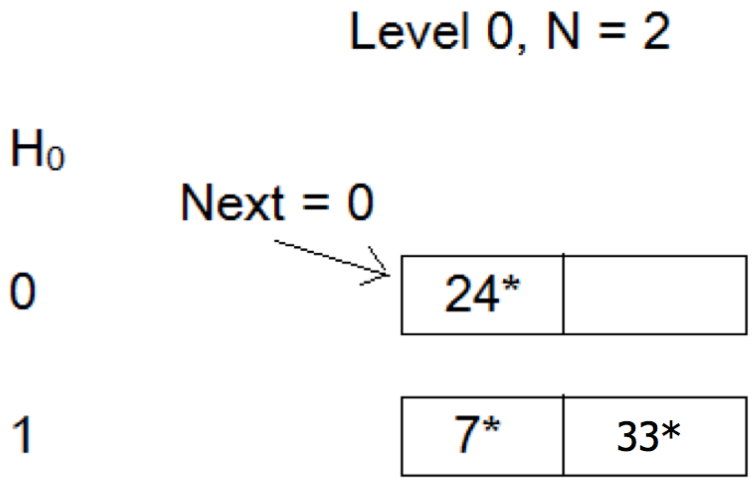
3 (e.g. insert 9*, 11*, 13*)

Q3. Linear Hashing

2) Perform the following insertions on the linear hashing index: 33, 54, 11, 26, 42. Assume that we use the same algorithm discussed in the lecture. Draw the index structure after each insertion, and show the next pointer and level at each step.

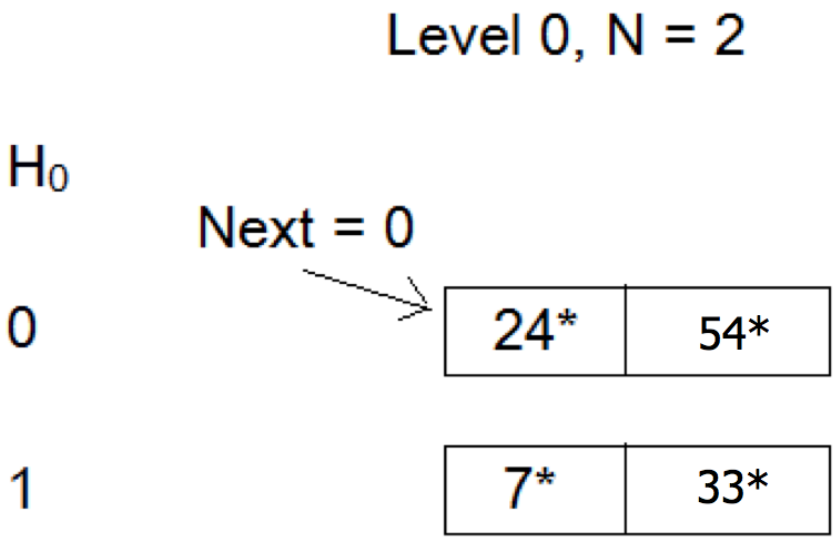
Q3. Linear Hashing

Insert 33*



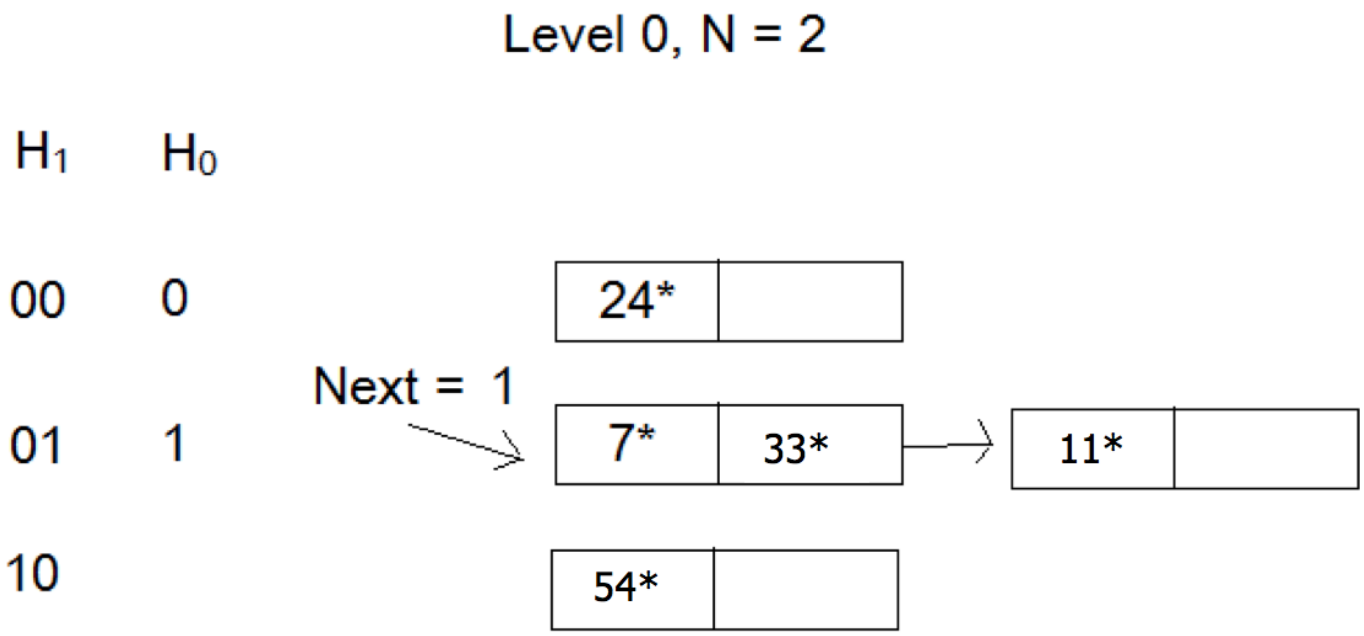
Q3. Linear Hashing

Insert 54*



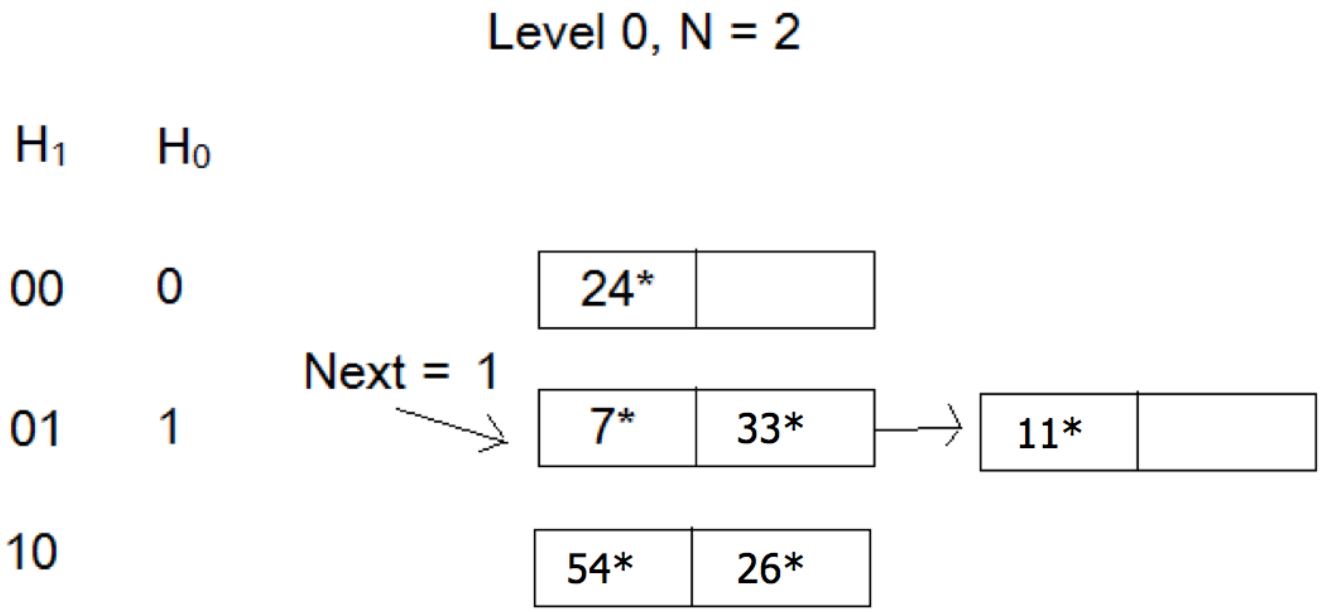
Q3. Linear Hashing

Insert 11*



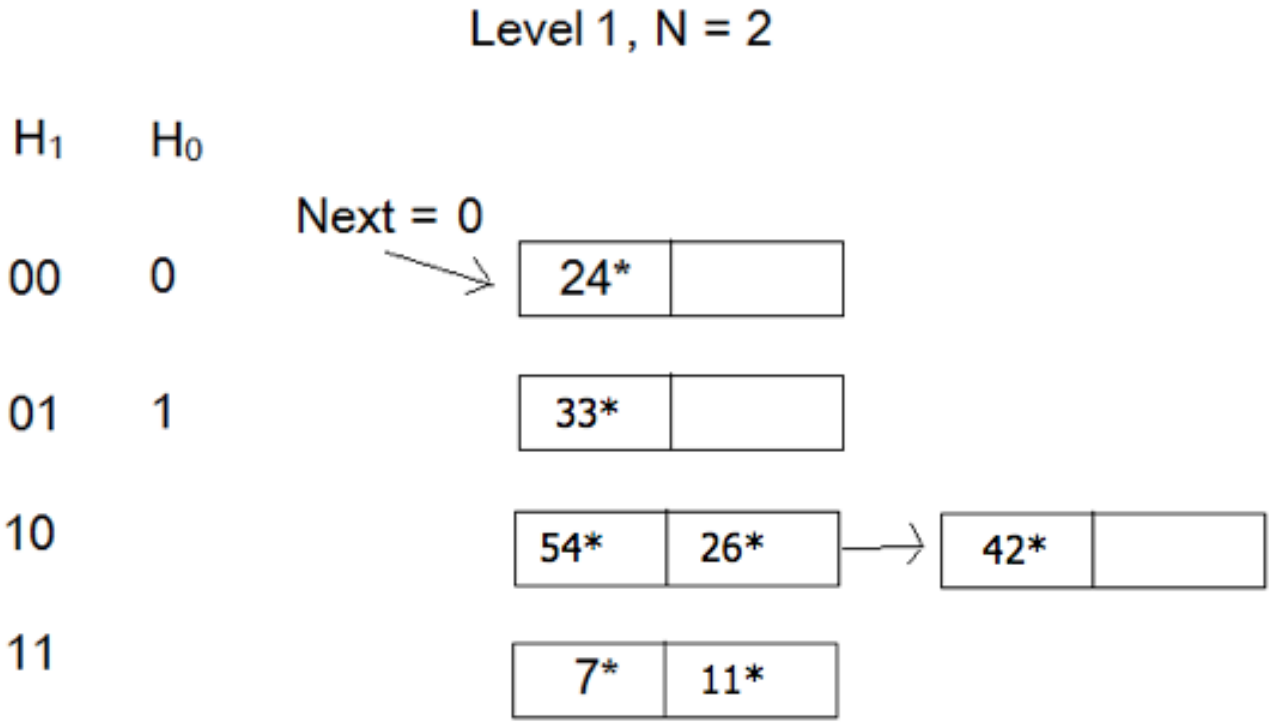
Q3. Linear Hashing

Insert 26*



Q3. Linear Hashing

Insert 42*



Q4. Extendible Hashing

Suppose we have an extendible hash index that uses the hash function:

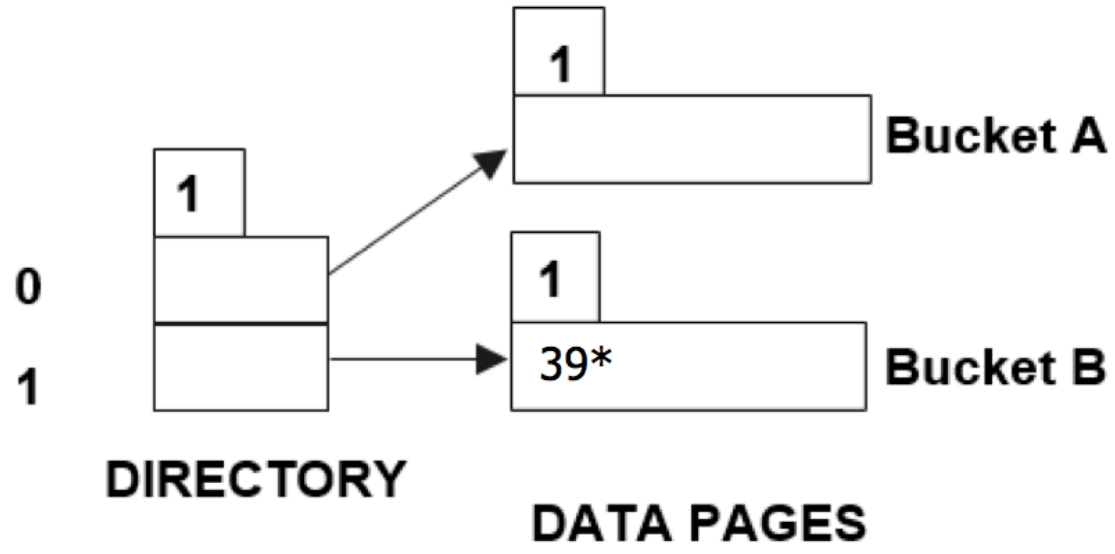
hashvalue = last **d** bits of the binary version of the given number

where d is the global depth. We will be using a bucket capacity of 2 for this problem. The initial global depth and local depth are 1. Assume that we use the same hashing function as above (except i is now the global depth)

Our index is initially empty. Then the following insertions occur: 39, 16, 26, 49, 18, 13, 41. Draw the index structure after each insertion. Be sure to clearly show the global and local depths and all bucket pointers at every step.

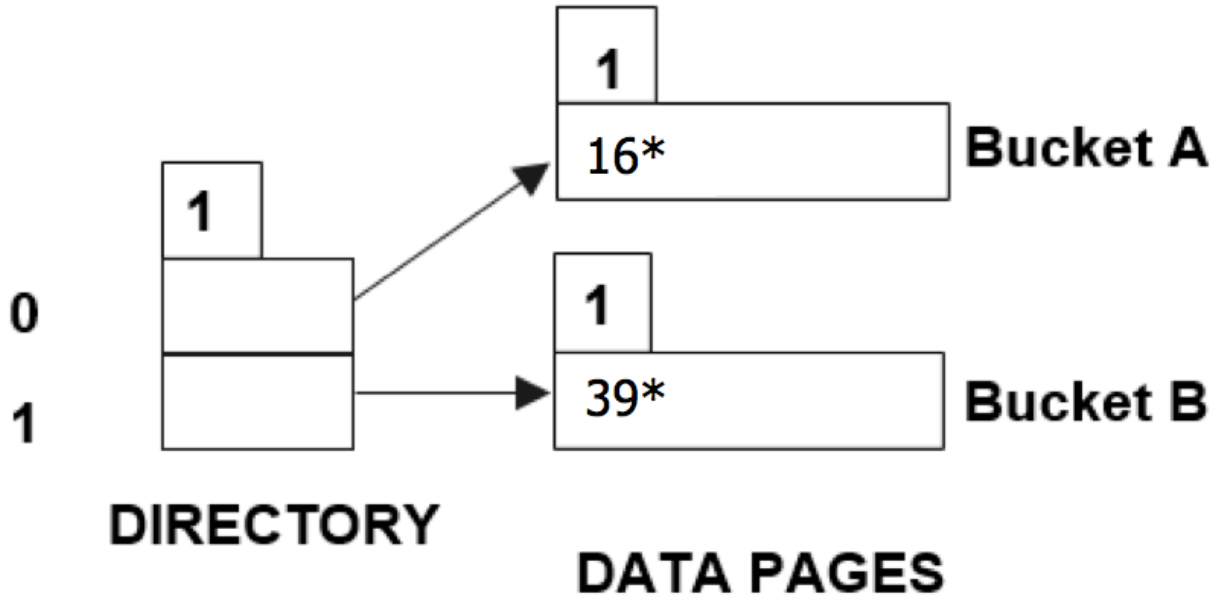
Q4

Insert 39



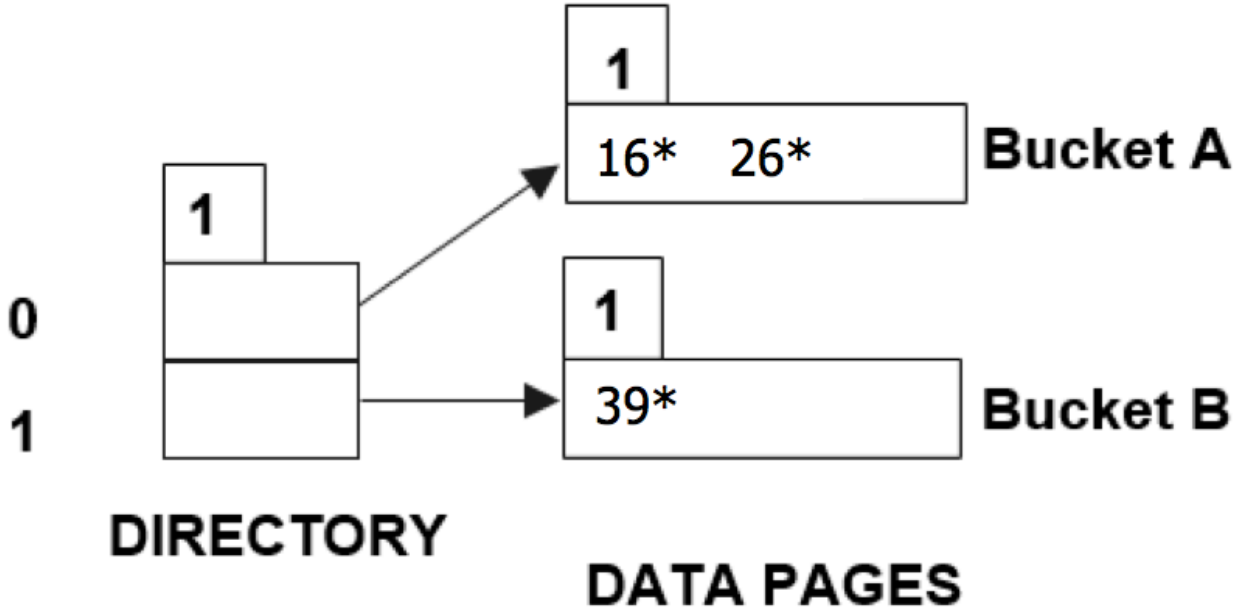
Q4

Insert 16



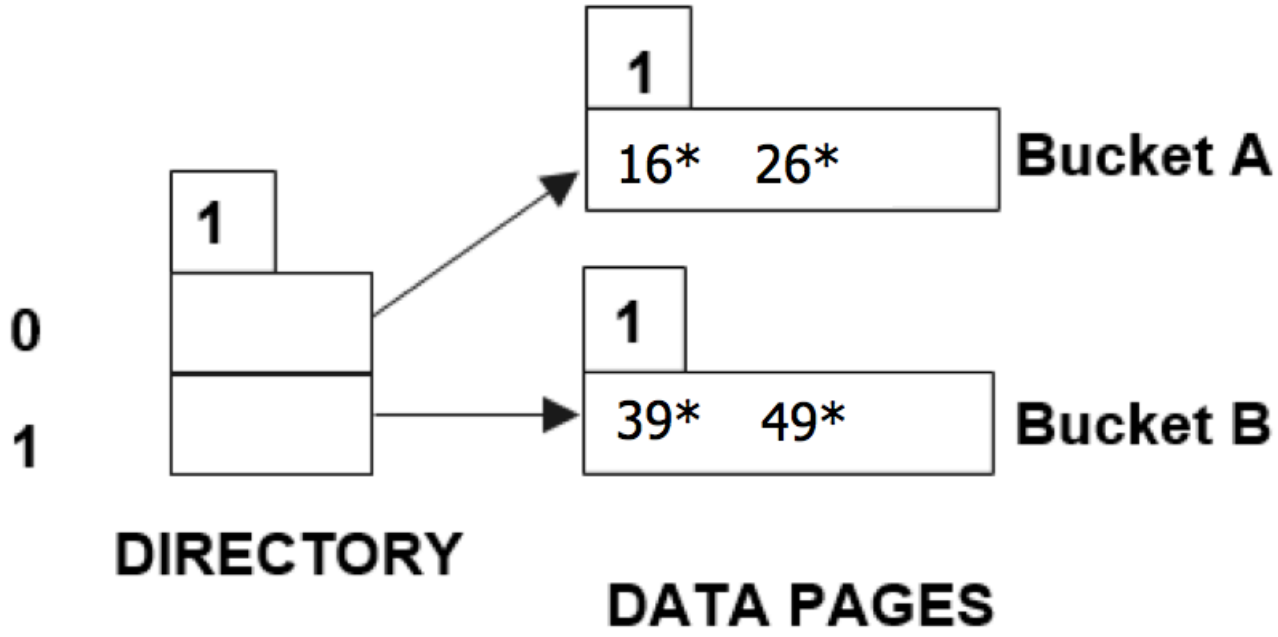
Q4

Insert 26



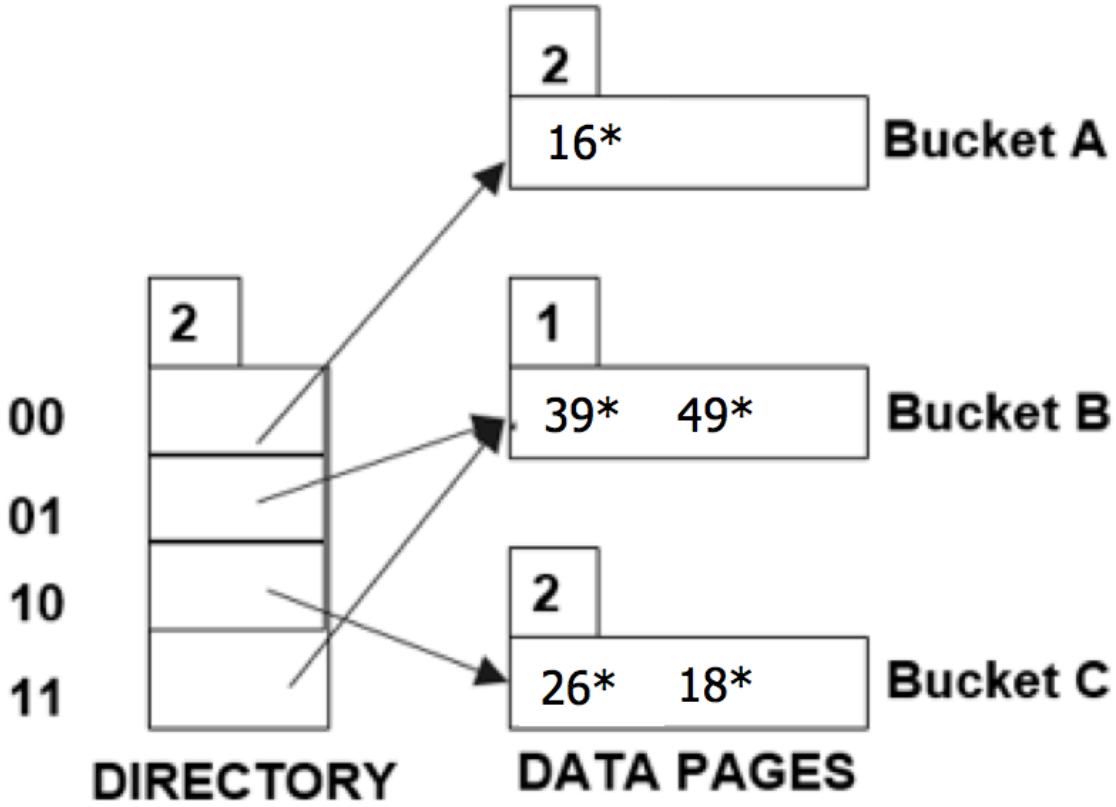
Q4

Insert 49



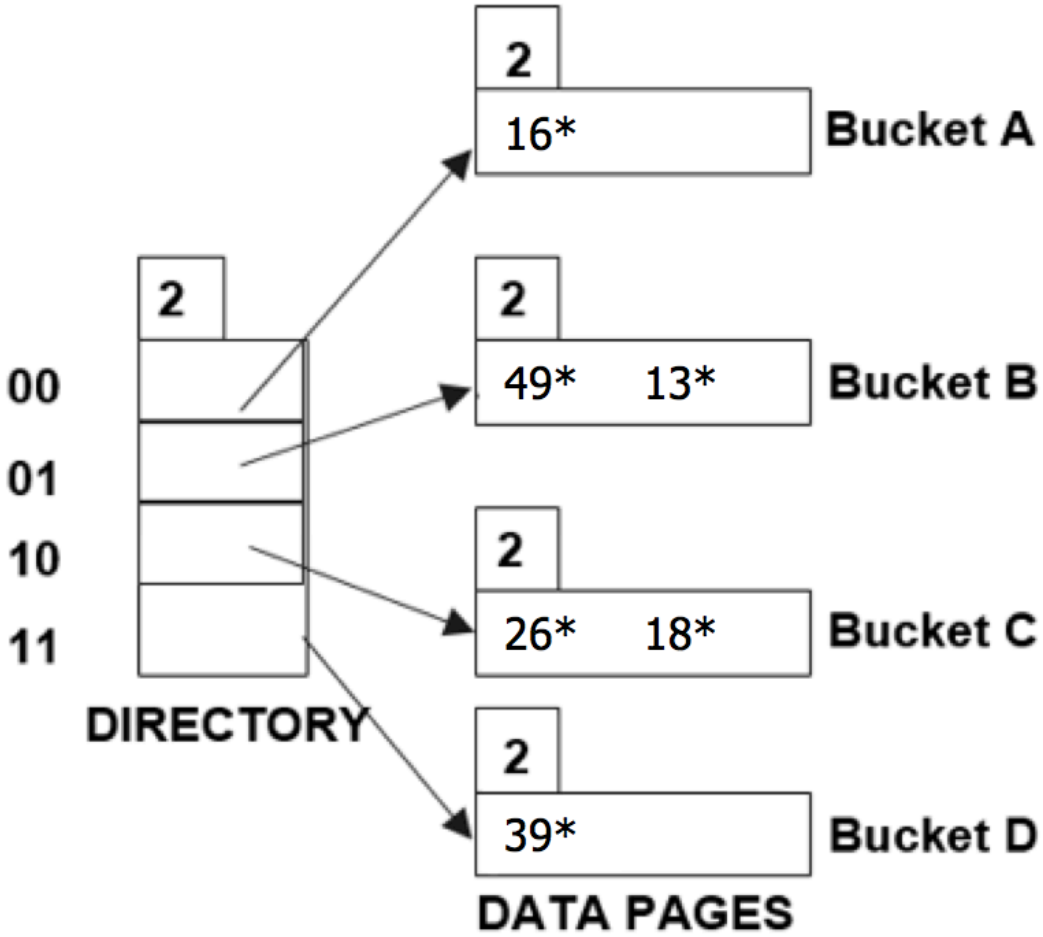
Q4

Insert 18


2
16*
Bucket A
2
1
39*
49*
Bucket B
00
01
10
11
2
26*
18*
Bucket C
DIRECTORY
DATA PAGES

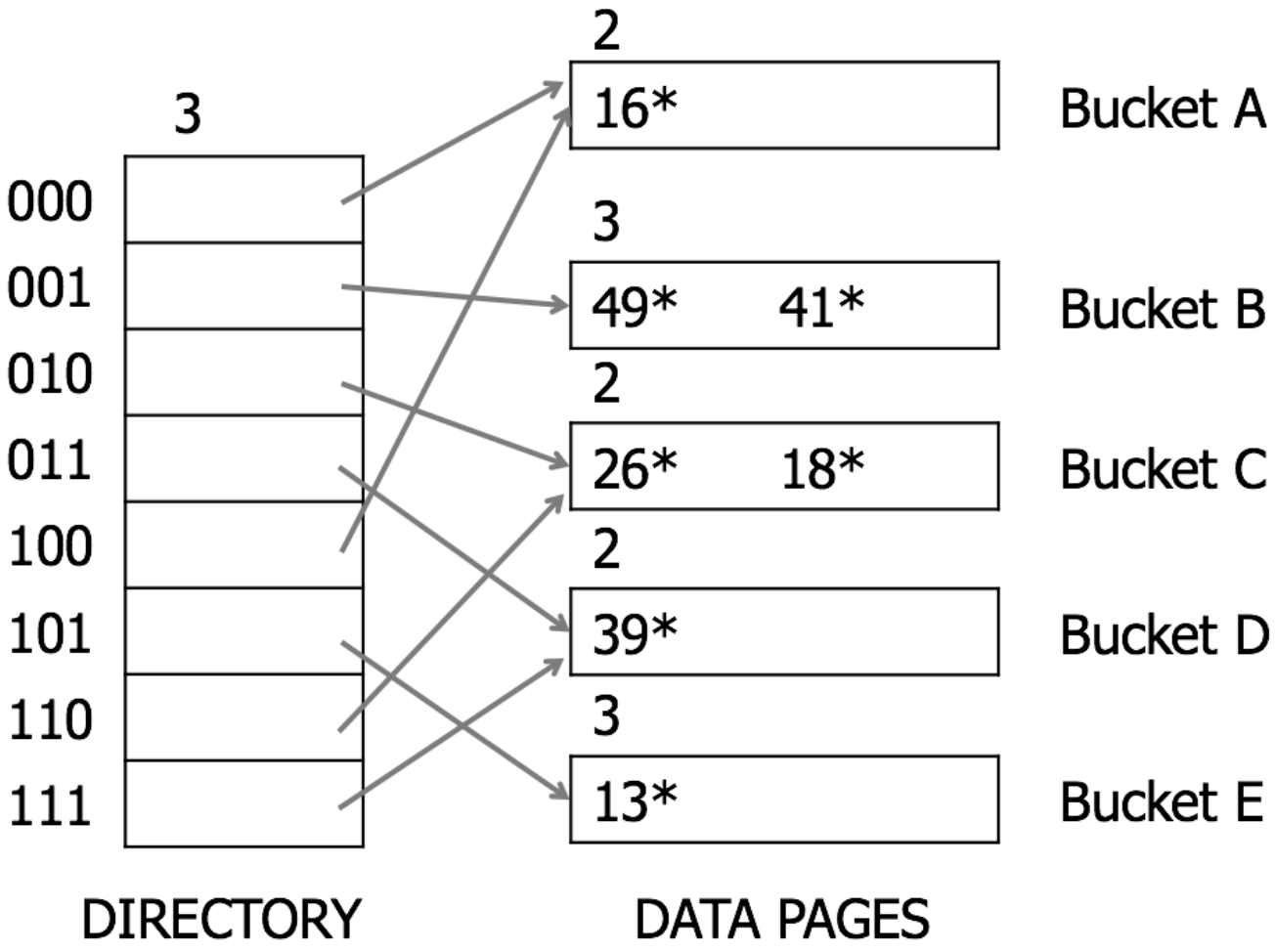
Q4

Insert 13


2
16*
Bucket A
2
2
49*
13*
Bucket B
00
01
10
11
2
26*
18*
Bucket C
DIRECTORY
2
39*
Bucket D
DATA PAGES

Q4

Insert 41


2
16*
Bucket A
3
3
49*
41*
Bucket B
2
26*
18*
Bucket C
2
39*
Bucket D
3
13*
Bucket E
3
000
001
010
011
100
101
110
111
DIRECTORY
DATA PAGES

Q5. Blocked IOs

Suppose we have a data set consisting of a 1TB (1T=1024G) file that we need to sort. We have 8GB of memory (RAM) to assist in sorting. The page size of the system is 32KB and the buffer block size is 32KB. Assume that replacement sort is used for the initial pass.

Q5

1) How many passes do we need on average to sort the dataset, if we only use replacement sort?

B: Buffer pages = 8 GB / 32 KB = 2^{18} = 262144
N: Pages in file = 1TB / 32KB = 2^{25} = 33554432
# of passes = $\log_{B-1}(N/(2*(B-2))) + 1 = 2$

2) How many I/Os are needed if we take 15 passes to sort the dataset?

# of IOs = 2 * N * # of passes = 15 * 226 = 1006632960

Q5

3) How many passes do we need if we use blocked I/Os?

Buffer block page = 32KB / 32KB = 1
F = 218/1 - 1 = 262143
# of sorted runs = N/2(B-2) = 64
# of passes = $\log_{262143}64 + 1 = 2$

Q6. Bulk Loading

In this question we will compare the IO cost between bulk loading and repeated insertions. We make the following assumptions:

- 10,000 records
- A page size of 1024 bytes
- A tree of order 12
- A pointer/value/search key takes 8 bytes
- Leaf nodes hold as many data entries as possible using alternative 2 (as described in lecture).
- Each node takes up exactly one page of memory.
- Our buffer pool can hold exactly as many pages as the height+1 of the fully created B+tree.
- After bulk loading, all inner nodes are half full.

Q6

1) Before bulk loading, we need to sort our record pointers. If we use general external merge sort, how many IOs will it take to do the sort?

# of pointers each leaf node = (1024-8-8) / (8+8) = 63
# of leaf nodes = 10000 / 63 = 159
Height 1 nodes = 159 / 13 = 12
Root node = 1

So height = 2, buffer can hold 3 pages
# of passes = $\log_{B-1}(N/B) + 1 = \log_2(159/3) + 1 = 7$
# of IOs = 2 * 159 * 7 = 2226

Q6

2) Calculate the IO cost to build the B+ tree. Include the cost of "committing" the changes/saving to disk after completion. For simplicity, assume the buffer pool is flushed before this step. Also assume you can control which pages stay in memory.

When bulk loading, we always want to insert in the right node. Therefore every time we split a node, we can control that the nodes in memory are only the nodes that will be updated for future insertions. Since the buffer pool can hold height+1 pages, each node will only need to be created, and never read from again.

Q6

2) Calculate the IO cost to build the B+ tree. Include the cost of "committing" the changes/saving to disk after completion. For simplicity, assume the buffer pool is flushed before this step. Also assume you can control which pages stay in memory.

So each inner node need a write, and each leaf node need a write and a read.

# of IOs = 1(root) + 12(height 1 nodes)
+ 159 * 2(leaf nodes) = 331

Q6

3) Assume that the average number of IOs taken for each repeated insert is half the height of the final tree. Which method (bulk loading or repeated inserts) requires fewer IOs? By how much?

Using repeated inserts: # of IOs = 1 * 10000 = 10000

Using bulk loading: # of IOs = 2226 + 331 = 2557

So bulking loading requires 10000 - 2557 = 7443 fewer IOs than repeated insert.

Postgres

New Tools in Postgres

- **pg_class and pg_stats tables:** these tables store the statistics that the DBMS uses for estimating the cost of query plans
- **EXPLAIN command:** shows the execution plan of a statement

Examining the System Catalog

The following three tables contain statistics information:

- pg_class
  - Total number of entries in each table and index
  - Number of disk blocks occupied by each table and index
- pg_statistics
  - Used by the planner to estimate the selectivity of the WHERE clause (i.e., the expected number of rows that match each condition in the WHERE clause)
- pg_stats
  - Shows information from pg_statistics restricted to a specific user. Restricted to show only information about tables that the current user can access

Further documentation: http://www.postgresql.org/docs/8.4/static/planner-stats.html

Examples of System Catalog Queries

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

```
SELECT attname, n_distinct, most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | n_distinct |
---------+------------+-----------------------------------------------------
 name    |  -0.467008 | {"I- 580                          Ramp","I- 880
 thepath |         20 | {"[(-122.089,37.71),(-122.0886,37.711)]"}
(2 rows)
```

EXPLAIN Command

- Shows the execution plan of a statement
- Can be used for any kind of statement (e.g., SELECT, INSERT, DELETE, VALUES, EXECUTE, DECLARE)
- Syntax:
  EXPLAIN [ANALYZE] [VERBOSE] statement
- Example: a simple query on a table with a single integer column and 10,000 rows

```
EXPLAIN SELECT * FROM foo;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)

EXPLAIN SELECT * FROM foo;
```

EXPLAIN Command (Cont.)

For more details on how to interpret the output of the EXPLAIN command…

http://www.postgresql.org/docs/8.4/static/using-explain.html

Additional Postgres Resources

Documentation: http://www.postgresql.org/docs/8.4/static/

PostgreSQL Book: http://momjian. us/main/writings/pgsql/aw_pgsql_book/