STUDENT NAME: _____ **STUDENT ID:** _____

CS 222P – Fall 2017, Midterm Exam

Principles of Data Management
Department of Computer Science, UC Irvine
Prof. Chen Li
(Max. Points: 100)

Instructions:

- **This exam has six (6) questions.**
- **This exam is closed book. However, you can use one cheat sheet (A4 size).**
- **The total time for the exam is 75 minutes, so budget your time accordingly.**
- **Be sure to answer each part of each question after reading the whole question carefully.**
- **If you don't understand something, ask for clarification.**
- **If you still find ambiguities in a question, write down the interpretation you are taking and then answer the question based on that interpretation.**

QUESTION	POINTS	SCORE
1	12	
2	21	
3	16	
4	18	
5	16	
6	17	
TOTAL	100	

Question 1: Short questions (12 points)

a) (3 pts) Explain the main idea of the PAX page structure to store records.

PAX partitions each page into minipages based on fields

b) (3 pts) Why do most DBMS systems use their own buffer manager instead of using the memory manager of the operating system?

DBMS knows more about its operations. It can use optimized policies to improve performance.

c) (3 pts) Explain a main advantage of using tables to store catalog information.

Reuse logic about tables and records.

d) (3 pts) Explain how the dirty bit of a frame is used by a buffer manager.

A writer of a frame needs to set the dirty bit of a page to 1, indicating that the page has been changed. If the dirty bit is 1, the buffer manager needs to flush the page to the disk before replacing it. If the bit is 0, the flush is not needed.

Question 2: Record Manager (21 points)

Suppose we have a table with the following schema:

Product(pid INT, price FLOAT, name VARCHAR(20), category VARCHAR(20))

We store the table as a heap file of variable-length records. Each record is stored as a sequence of bytes with a directory of pointers. Assume that:

(1) The directory uses a 2-byte offset to store an ending position for each field.

(2) The offset starts from 0, which is the beginning of the directory.

(3) The schema is known in all record operations so there is no need to store the number of fields in each record;

(4) Integers and floating point numbers are 4 bytes.

(5) Each NULL value is represented using a special value -1 in the corresponding pointer in the directory.

Consider this record:

(12, 999.0, "iPhone X", NULL)

a) (4 pts) Draw a byte array that explains how this record is stored using the format stated above. Clearly indicate the number of bytes in each segment and all their values.

The ending position of each field can either be the last byte of the field, or the byte immediately after the field. Both are correct, as long as the answer is consistent.

12	16	24	-1	12	999.0	i	P	h	o	n	e		X
2	2	2	2	4	4	8							

or

11	15	23	-1	12	999.0	i	P	h	o	n	e		X
2	2	2	2	4	4	8							

b) (4 pts) Draw a byte array using the API data format from the course projects, i.e., the format for *"*data"* in functions *"insertRecord()"* and *"updateRecord()"* for RBFM. For the null indicator, show the values of the 0/1 bits. Clearly indicate the number of bytes in each segment and all their values.

00010000	12	999.0	8	i	P	h	o	n	e		X
1	4	4	2 or 4	8							

An additional 4 bytes number for number of fields are also correct, even though it's not in our project. For null indicator, both 00010000 and 00001000 are correct.

c) (3 pts) Give one advantage of the data storage format in a) compared to the API format in b). O(1) field access.

d) (10 pts) When inserting a new record, we always first check the last page. If the last page doesn't have enough space, we need to find an existing page that can hold the record, or create a new page if we can't find it.

Suppose we have two ways to implement it:
(1) Scan each page from the beginning of the file until we find a suitable page.
(2) Use a single header page to store the free-space information of each page and look up the header page.

Assume that:
(1) After each insert, all the modified pages should be immediately flushed to disk.
(2) Initially the file doesn't have any pages, even the header page in implementation (2).
(3) We know where the last page is without any disk IO.

Initially the table is empty. We are going to insert 10 records one by one, and each page can store exactly 3 records.
Fill in the following two tables for the number of reads and writes for each insertion.

When inserting a record requires creating a new page, there should be 0 Read and 1 Write. Because you can just write the page with the record in it.

For the header page implementation. You can choose to 1) always check the last page first, or 2) always check the header page first.
1) will have 2 Reads when inserting record 4, 7, and 10.
2) will have 1 Read when inserting record 4, 7 and 10.
Both are correct.

When creating a new page, there are 0 Read and 1 Write.

Implementation 1 (scan)		
Inserted Record	Number of Reads	Number of Writes
1	0	1
2	1	1
3	1	1

Implementation 2 (header page)		
Inserted Record	Number of Reads	Number of Writes
1	0	2
2	2	2
3	2	2

4	1	1
5	1	1
6	1	1
7	2	1
8	1	1
9	1	1
10	3	1

4	2 (or 1)	2
5	2	2
6	2	2
7	2 (or 1)	2
8	2	2
9	2	2
10	2 (or 1)	2

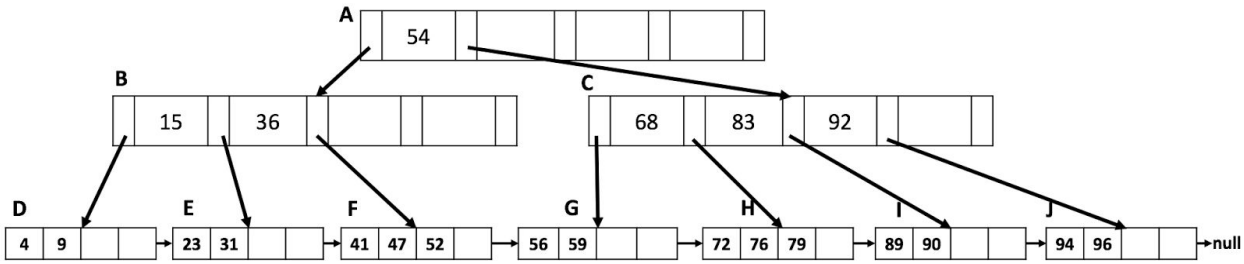
1 Read and 1 Write when creating a new page:

Implementation 1 (scan)		
Inserted Record	Number of Reads	Number of Writes
1	1	1
2	1	1
3	1	1
4	2	1
5	1	1
6	1	1
7	3	1
8	1	1
9	1	1
10	4	1

Implementation 2 (header page)		
Inserted Record	Number of Reads	Number of Writes
1	2	2
2	2	2
3	2	2
4	2	2
5	2	2
6	2	2
7	2	2
8	2	2
9	2	2
10	2	2

Question 3: B+ Tree (16 points)

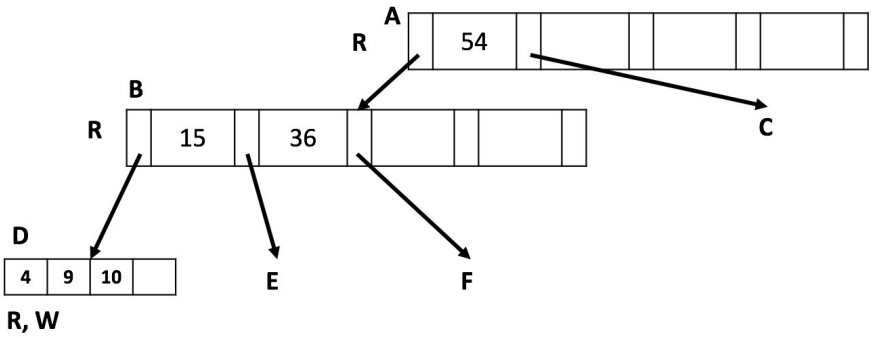
Consider the following B+ tree

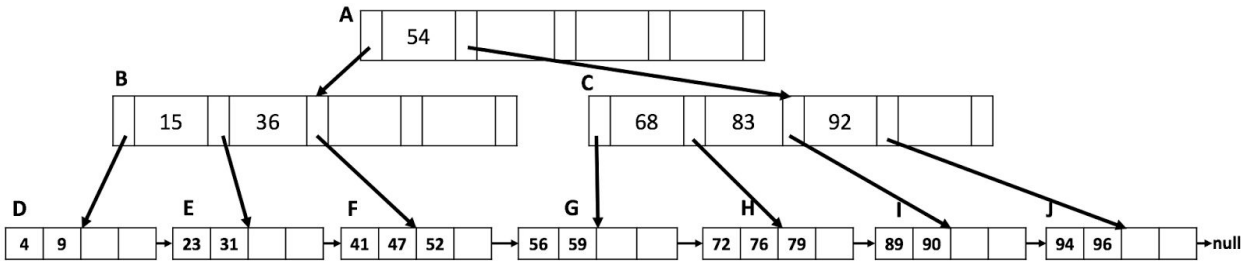


For the following questions, draw the updated B+ tree after each operation. If there is no ambiguity, you can just draw the part that is changed. For each operation, clearly add an “R” (for read) and “W” (for write) on all the affected pages, as well as the original A/B/C/.../J labels.

Assume:
(1) Each node can hold up to 4 entries.
(2) For delete operations, you can borrow from the **right** sibling.

Example: Insert 10 on the **original** tree.

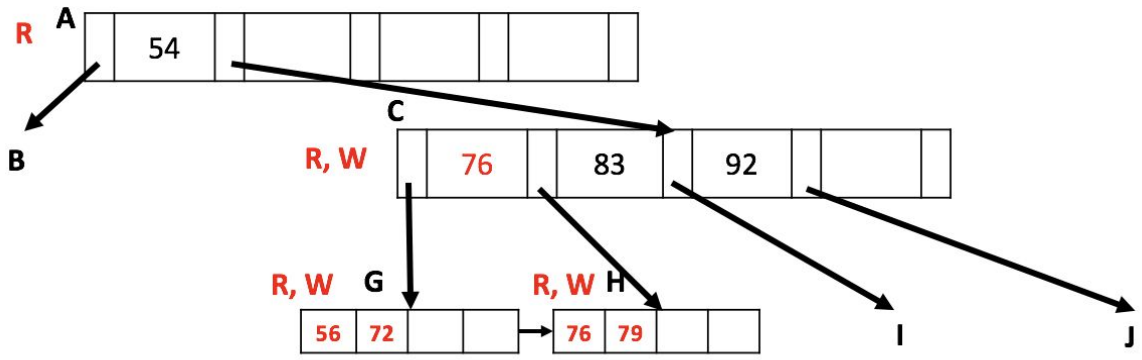




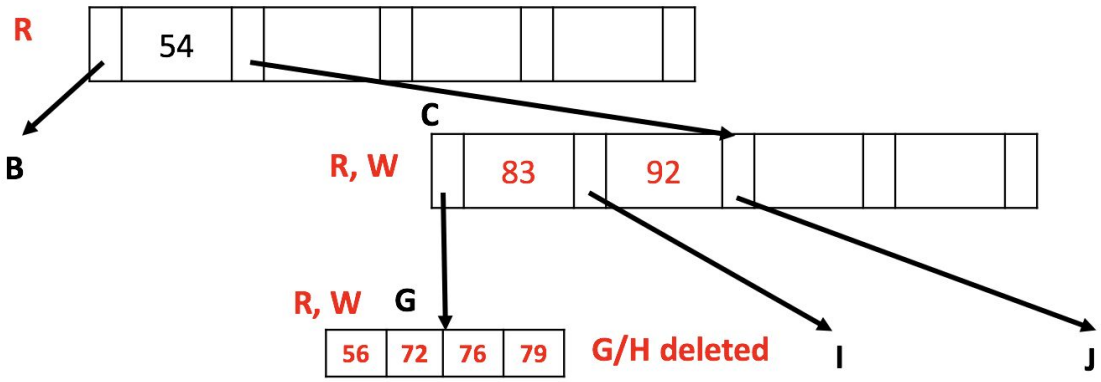
The same b+ tree is copied here for your convenience.

a) (8 pts) Delete 59 on the **original** tree.

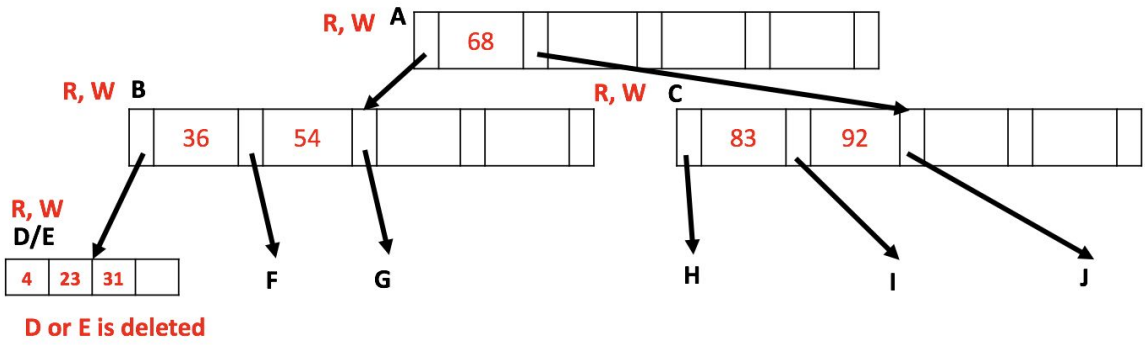
Borrow 72 from H



Or merge G and H.



b) (8 pts) Delete 9 on the **original** tree.



Question 4: Dynamic Hashing (18 points)

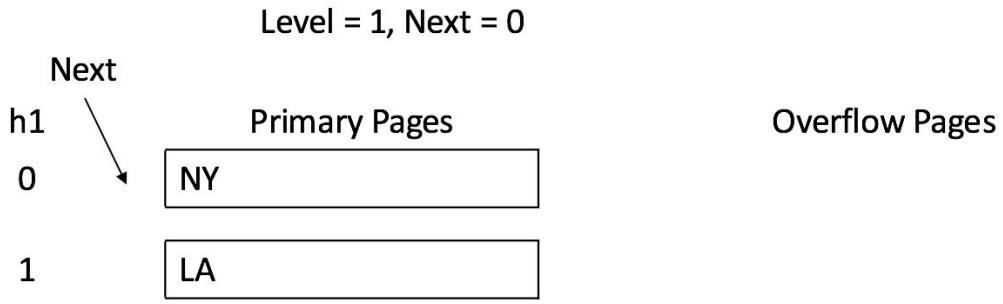
a) (4 pts) Briefly describe a main drawback of static hashing in terms of its performance.
Long overflow chains.

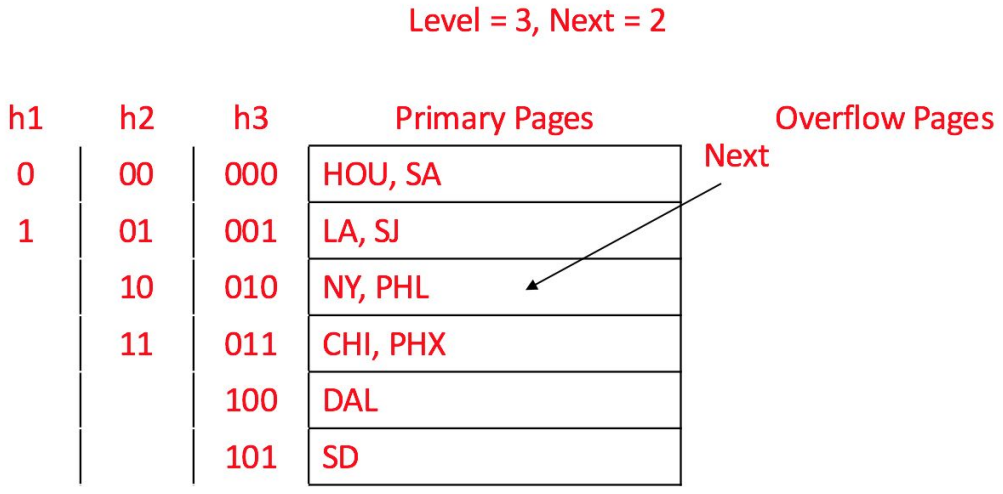
b) (4 pts) Briefly describe how global depth and local depth in extendible hashing are used.
Global depth to figure out which bucket an entry belongs to
Local depth for splitting

Now, consider storing a set of city code records in a table, which has a dynamic hash table. Suppose we want to insert the following 10 entries. We are using a hash function called ***h,*** and the binary hashed value for each entry is given below. Only 2 records can fit into a bucket/page. Assume that the bits of h(k) are used in least-to-most order of significance. For example, for a binary value "1101", we use the two rightmost bits "01" to find the bucket.

City code	h(city)
NY	1010
LA	0001
CHI	0111
HOU	0000
PHX	1011
PHL	1010
SA	1000
SD	1101
DAL	1100
SJ	1001

c) (10 pts) Suppose the index is based on **linear hashing** and our policy is to split the page indicated by the "Next" pointer whenever an overflow page is created due to an insertion. Below is the state of dynamic hashed index after inserting the first two data entries. Draw a diagram representing the final state of the dynamic hash index after inserting all the 10 given data entries.





Or:
When inserting SD, don't split because the overflow page is not full

000 HOU SA
001 LA SD -> SJ (next)
010 NY PHL
011 CHI PHX
100 DAL

Question 5 Index Performance (16 pts)
Suppose you are in charge of managing the database for a university. Consider a table

Student (ID, name, age, GPA, units)

that contains 10,000,000 records (10M) organized as a sorted file ordered by ID, and each page contains 100 records. We assume:

- "ID" is an integer value uniformly distributed over the range [0, 9,999,999].
- "name" is a variable-length string of a maximum length of 20 and with a not-null constraint. We further assume names are unique in this table.
- "age" is an integer value uniformly distributed over the range [10, 49].
- "GPA" is a double value uniformly distributed over the range [0, 4.0].
- "units" is a double value uniformly distributed over the range [0, 999].
- The value distributions of multiple attributes are independent.

a) (3 pts) What is the I/O cost of the full scan over the "Student" table?
10M/100 = 100K

For questions b) - d), describe which of the following access methods would work the best for the given workload. **Briefly explain the reason** and **indicate whether an index-only plan is applicable or not**.

1) Sorted-file scan
2) Unclustered B+ tree index on "age";
3) Unclustered B+ tree index on "GPA";
4) Unclustered B+ tree index on "units";
5) Unclustered B+ tree index on composite key "<age, units>";
6) Unclustered B+ tree index on composite key "<units, age>";

b) (3 pts) SELECT ID, name FROM Student WHERE units<=4;

4) , not applicable.

The predicate on units is very selective, i.e., 1/1000.

c) (3 pts) SELECT * FROM Student WHERE age<=19;

1), not applicable

The predicate on age is not selective, and it's better to use full scan.

d) (7 pts) SELECT COUNT(*), AVG(units)
FROM Student
WHERE GPA BETWEEN 3.0 AND 3.2
AND units<=199;

For this question, also **report the I/O cost** of using the method that you choose. We assume each index page contains 1,000 (composite) key entries and non-leaf pages of a B+ tree are always cached by the buffer manager. Furthermore, RIDs are sorted before accessing the sorted file.

3)+4). Intersect the RIDs fetched from these two indexes.

Index-only plan is applicable.

#records fetched by GPA=10M*(0.2/4)=0.5M

#index pages accessed by GPA = 0.5M/1K = 500

#records fetched by units = 10M*(200/1000)=2M

#index pages accessed by units = 2M/1K = 2000

#total pages = 2500

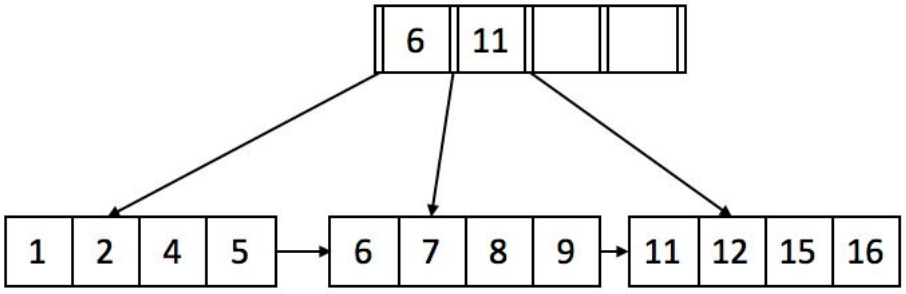
Question 6: Bulk Loading (17 pts)

a) (3 pts) Name at least 3 advantages of bulk loading a B+ tree compared to inserting multiple records into a B+ tree one by one.

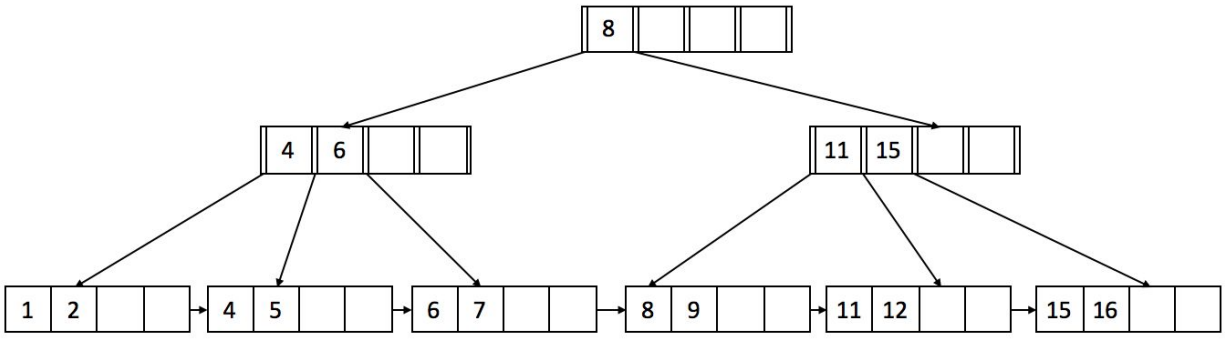
Advantages for concurrency control
Fewer I/Os during build
Leaves would be stored sequentially
Can control "fill factor" on pages
Can optimize non-leaf splits more than shown

b) (5 pts) Consider bulk loading a B+ tree with an order of 2. Suppose the utilization ratio of leaf pages is 100%. Draw the final bulk loaded B+ tree with the following keys:

12, 2, 5, 9, 8, 7, 6, 11, 15, 1, 4, 16



c) (5 pts) Draw the final bulk loaded B+ tree with the same keys in (b) but the utilization ratio of leaf pages being 50%.



d) (4 pts) In general, a bulk loaded B+ tree has a much better shape than a B+ tree built using multiple insertions in terms of its space utilization of each leaf page. Would bulk

loading still be helpful for a hash index? That is, is it possible to use the bulk loading technique to build a hash index with a better space utilization and ideally no overflow pages? Please justify your answer.

No. Overflow pages in hash index are caused by hash collisions, where multiple keys (maybe same keys) are hashed to the same bucket. Bulk loading cannot help to solve this problem, since we cannot change the hash value of these collision keys.