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

CS 222/122C – Fall 2017, Final Exam

Principles of Data Management

Department of Computer Science, UC Irvine

Prof. Chen Li

(Max. Points: 100 + 15)

Instructions:

- **This exam has seven (7) questions and one (1) additional extra credit question.**
- **This exam is closed book. However, you can use one cheat sheet (A4 size).**
- **The total time for the exam is 120 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	15	
2	10	
3	13	
4	18	
5	12	
6	12	
7	20	
8 (extra credit)	15	
TOTAL	100 + 15	

Question 1: Short questions (15 points)

a) (3 pts) What is a main advantage of requiring all operators in a DBMS to implement the same interface functions open(), getNext(), and close()?

Answer: Operators can be easily connected to generate a plan since they have the same interface.

b) (3 pts) In the System-R query optimizer, what is the meaning of saying “a subplan (a subtree) has an interesting order for an attribute A”?

Answer: The subplan generates intermediate results sorted on the attribute A. In addition, the attribute A is used later in the remaining plan, i.e., it’s used in a join, group by, distinct, or order by operator.

c) (3 pts) In our analysis of the number of disk IOs for an operator, we always ignore the IO cost of outputting its results. Why?

Answer: These results could be pipelined to the next operator without going to the disk.

d) (3 pts) Suppose we have four tables R(A,B), S(B, C), T(C,D), and W(D,E), and we have a query that is a natural join of these tables. Consider possible tree plans to answer this query. Write (1) a left-deep tree plan; (2) a linear plan that is not a left-deep tree; and (3) a bushy tree plan that is not linear.

Answer: (1) ((R JOIN S) JOIN T) JOINT W; (2) (R JOIN (S JOIN T)) JOIN W; (3) (R JOIN S) JOIN (T JOIN W).

e) (3 pts) Consider the case where a B+ tree with duplicate entries that cannot fit in a single leaf page. There are two approaches:

(1) Keeping entries as <Key, List of RIDs>

(2) Keeping composite key entries as <Key, RID>

Name one advantage and one disadvantage of approach (1) compared to approach (2).

Approach (1) uses space more efficiently since it stores a duplicate key only once in data entries. Drawback is that it needs to maintain overflow pages when list of rids cannot fit in one page.

Question 2: B+ Tree (10 points)

a) (2 pts) Why is bulk loading a B+ tree faster than building it with repeated inserts?

Fewer I/O's, Fewer splits & rebalancing.

For b) and c), consider a table with the following schema.

Students(sid, name, gpa, deptID).

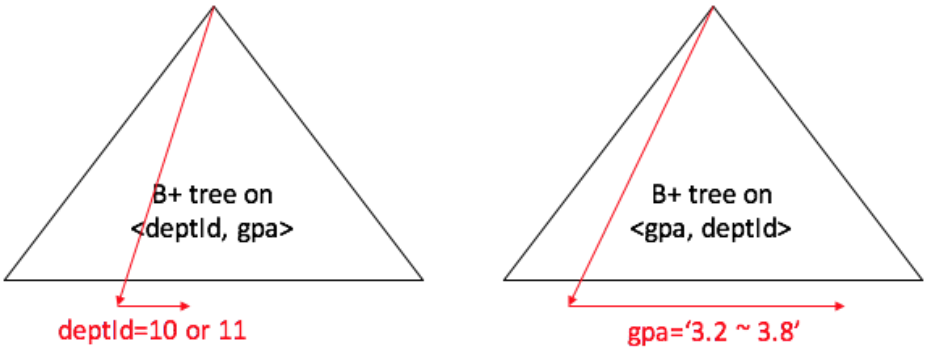
It has an unclustered B+ tree on <gpa, deptID> (called "Index1"), and an unclustered B+ tree on <deptID, gpa> (called "Index2"). We have the following assumptions.

- A "gpa" value is a float uniformly distributed over the range [3.0, 4.0].
- A "deptID" value is an integer uniformly distributed over the range [0, 99]
- The value distributions of multiple attributes are independent.

Consider the following query:

```
SELECT dept, AVG(gpa)
FROM Students
WHERE deptID = 10 or deptID = 11 AND gpa BETWEEN 3.2 AND 3.8
GROUP BY deptID;
```

b) (4 pts) Consider the access method of using the <gpa, deptID> index or the access method of using the <deptID, gpa> index to answer the query. For each of them, draw a diagram to illustrate its traversal behavior using the constants in the query ("deptID = 10 or 11" and "gpa between 3.2 and 3.8"). Make sure to draw enough details.



c) (4 pts) Based on the analysis above, which one is more IO efficient? Why?

(2), need to visit fewer # of nodes in tree.

Question 3: External Sort (13 points)

Suppose we have N = 9,300 pages of fixed-length records in a heap file. We have B = 31 available pages in memory to sort the file using the external sort algorithm covered in the lectures.

a) (4 pts) For each pass (including pass 0 of generating the initial runs), write down the number of sorted runs, and the size of a single sorted output run. Calculate the number of I/Os required to sort the entire file, excluding the writes in the last pass.

Pass 0: 300 sorted runs of 31 pages each
Pass 1: 10 sorted runs of 930 pages each
Pass 2: 1 sorted file of 9300 pages

5 * 9300 = 46500

b) (4 pts) Consider the final merge phase of external sort as discussed in class. Suppose the number of runs is **A**, the number of pages per run is **B**, and the number of records for each page is **C**. In class we covered an algorithm to merge the runs in memory using a priority queue (heap). Write a formula for the cost of this algorithm (for the final merge phase), and briefly explain the answer.

Answer: 2 * log(A) * (A * B * C). Each record needs to be pushed into and popped out of the heap (priority queue), and each push/pop operation has a cost of log(A). If we combine the pop of the previous top element and the push of the new element into one step, we may get rid of the "2 *" in the formula.

c) (5 pts) In the algorithm of generating the initial runs in pass 0 as covered in our lectures, we implicitly assumed that the records are fixed length, so that we can do **in-place swap** of two records in memory. Now consider the case where the records are **variable-length**, which does not support in-place swap since different records have different lengths. Describe a method to sort them in memory **efficiently**.

You need to allocate some space in memory to store a directory of pointers pointing to each record. Then when you sort the variable-length records, you swap the pointers instead of the actual record.
If the directory is small, then you need at least one buffer page. If there are too many records and one buffer page cannot hold that many pointers, you need more buffer pages as needed.

Question 4: Join (18 points)
Suppose we have the following two tables.

- Customer(**customer_id (primary key)**, name, age...)
    - 3,000 records (total), 200 pages;
    - An unclustered B+ tree on “customer_id”;
- Order(date, **customer_id**, price...)
    - 20,000 records (total), 500 pages;
    - “customer_id” is a foreign key to “Customer.customer_id”;
    - An unclustered B+ tree on “customer_id”;
    - The “customer_id” values are uniformly distributed.

We want to join them on their “customer_id” attributes (Customer.customer_id = Order.customer_id).

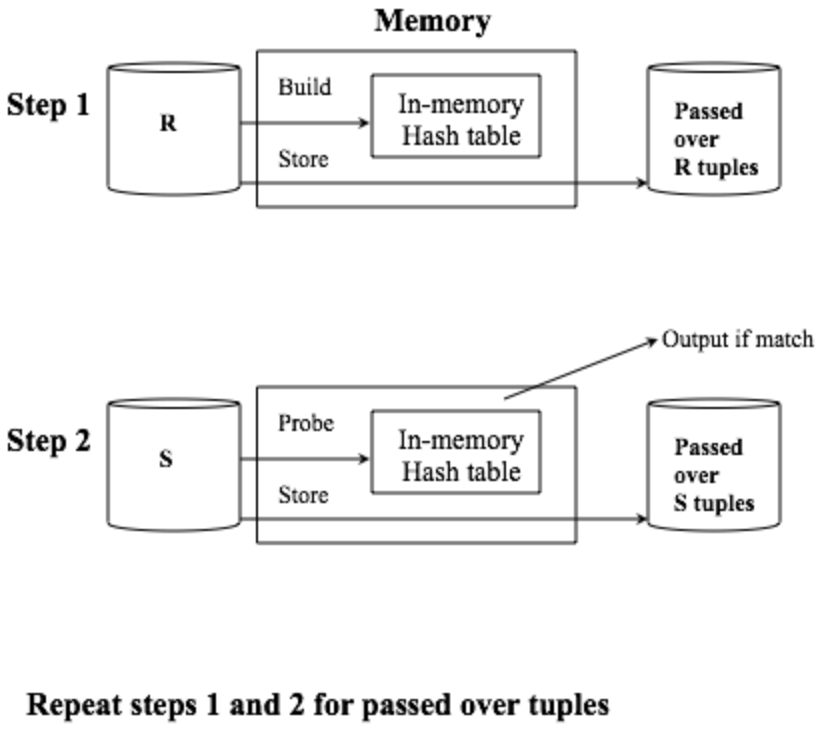
a) (4 pts) Estimate the IO cost of the **Index Nested Loop Join,** by scanning “Customer” heap file as the outer table and using the B+ tree of Order.customer_id. Assume all **non-leaf** pages of the B+ tree are cached in memory.

200 + 3000 * (1 + ceiling(20,000/3,000))
Assumption: we only need to access one leaf node of the Order.cid B+ tree.

b) (4 pts) Consider the case where we want to use **Grace Hash Join**. Assume we have **B = 16** in-memory buffer pages. We treat the smaller table as the outer table to build the in-memory hashtable in the building phrase. Calculate the number of disk I/Os in the **partitioning** phase and the number of disk I/Os in the **building/probing phase**, excluding the final writes.

Need 1 partition passes.
(2 * 2 - 1) * (200 + 500) = 2100

c) (5 pts) Draw a diagram to explain the main idea of Simple hash join.



Repeat steps 1 and 2 for passed over tuples

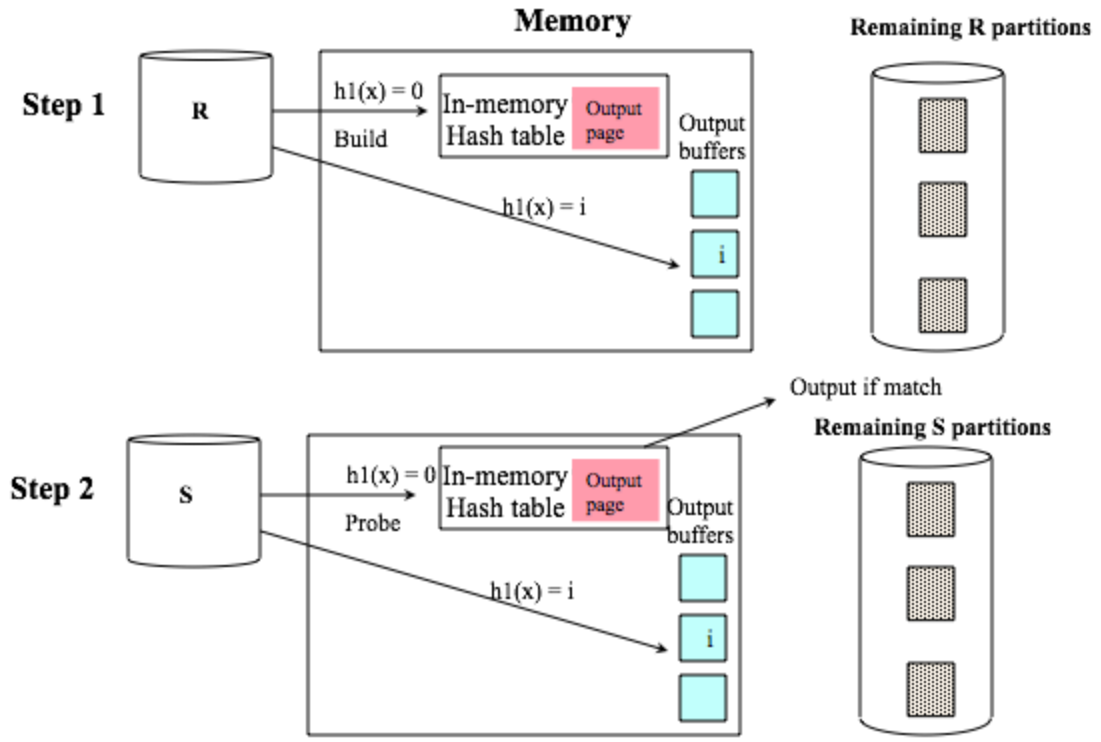
Suppose R is the smaller relation.

In step 1, read the records of R page by page. For each record, apply a hash function h. If the hash value h(x) is 0, keep it in memory to build a hash table. Otherwise, pass it to the disk through a buffer page.

In step 2, read the records of S page by page. For each record, apply the same hash function h. If the hash value h(x) is 0, do a lookup in the in-memory hash table and find matching results, which are output as results though one page buffer. Otherwise, pass it to the disk through one page buffer.

Repeat step 1 and step 2 for those passed-over records on the disk using a sequence of different hash functions, until the remaining disk records of R can fit into memory. Then load these pages into memory to build a hash table, and scan the disk pages of S to do the in-memory join.

d) (5 pts) Draw a diagram to explain the main idea of **Hybrid hash join**. In addition, explain how this algorithm combines the ideas from the grace hash join and simple hash join.



Step 3: process remaining R/S partitions as in the Grace Join

Suppose R is the smaller relation.

In step 1, read the records of R page by page. For each record, apply a hash function h. If the hash value h(x) is 0, keep it in memory to build a hash table for partition R0. Otherwise, pass it to the buffer page i = h(x), which will eventually flushed to the disk to generate partition Ri.

In step 2, read the records of S page by page. For each record, apply the same hash function h. If the hash value h(x) is 0, do a lookup in the in-memory hash table of R0 and find matching results, which are output as results though one page buffer. Otherwise, pass it to the buffer page i = h(x), which will eventually flushed to the disk to generate partition Si.

Use the second step of grace join to join each (Ri, Si) partition pairs.

This join algorithm combines the idea of “keeping one partition in memory to reduce their disk IOs” from Simple hash join and the idea of “partitioning those remaining records using multiple output buffers” from Grace hash join.

Question 5: Distinct (12 points)

In SQL, "`DISTINCT`" is a useful operator to remove duplicates in query results. Suppose UCI has an applicant table with the following schema:

`Applicant(name, email, phone, major)`: 10,000 records, 10 records per page

We want to know the number of distinct value on the "name" attribute.

a) (3 pts) Write down the SQL query to calculate the number of distinct names.
select count(distinct name)
from Applicant

For the following questions, suppose each page can hold 25 names, and we have 15 memory pages. Write down the intermediate steps as well.

b) (3 pts) What is the I/O cost of the `DISTINCT` operation if a sort-based approach is used (i.e., sort the values first and scan sorted values to remove duplicates)? Exclude the cost of the final writes.
We need to two merge passes.
Thus, the total is 1000 + 400 * 4 = 1000 + 1600 = 2600

c) (3 pts) What is the I/O cost of the `DISTINCT` operation if a hash-based approach is used (i.e., perform hash partitioning first, and then remove duplicates within each partition)? Again, exclude the cost of the final writes.
Same to b), 2600

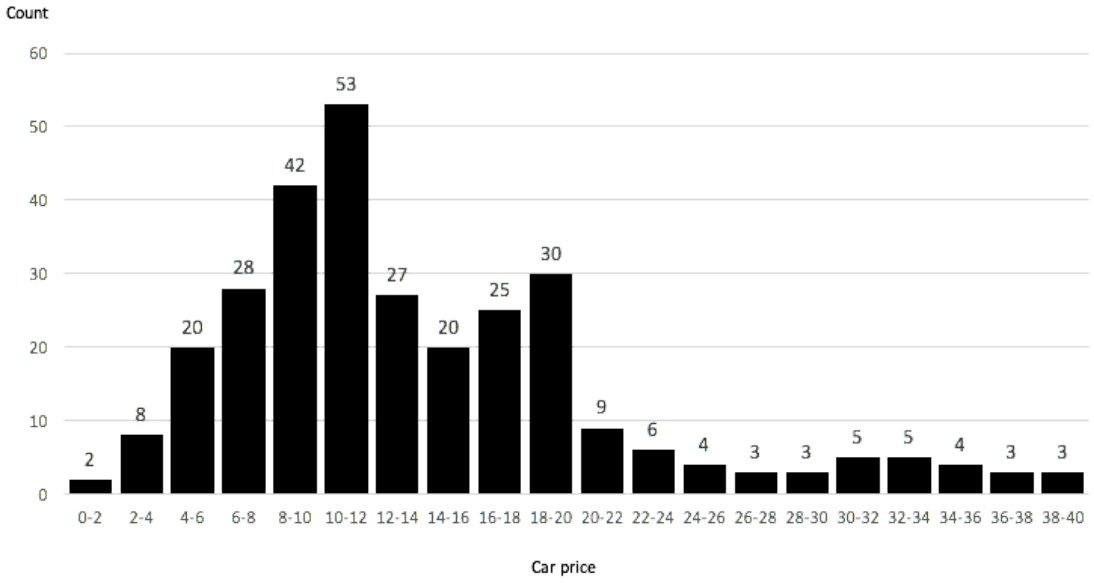
d) (3 pts) For each method in b) and c), propose an optimization to reduce the size of the intermediate results.
For b, eliminate duplicate results during sort/merge phase
For c, eliminate duplicate results during partitioning/building phase

Question 6: Cost Estimation (12 points)

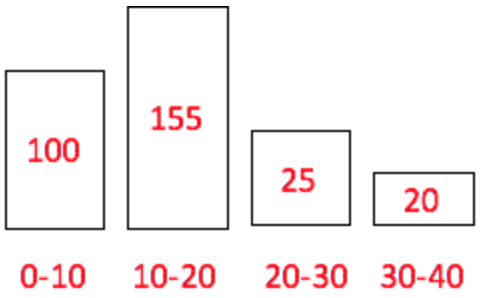
Consider a table with the following schema:

`CAR_SALES(name, maker, price).`

It has 300 records with the following distribution of "price" values of `double` type.



a) (3 pts) Draw a **4**-bucketed equi-width histogram.

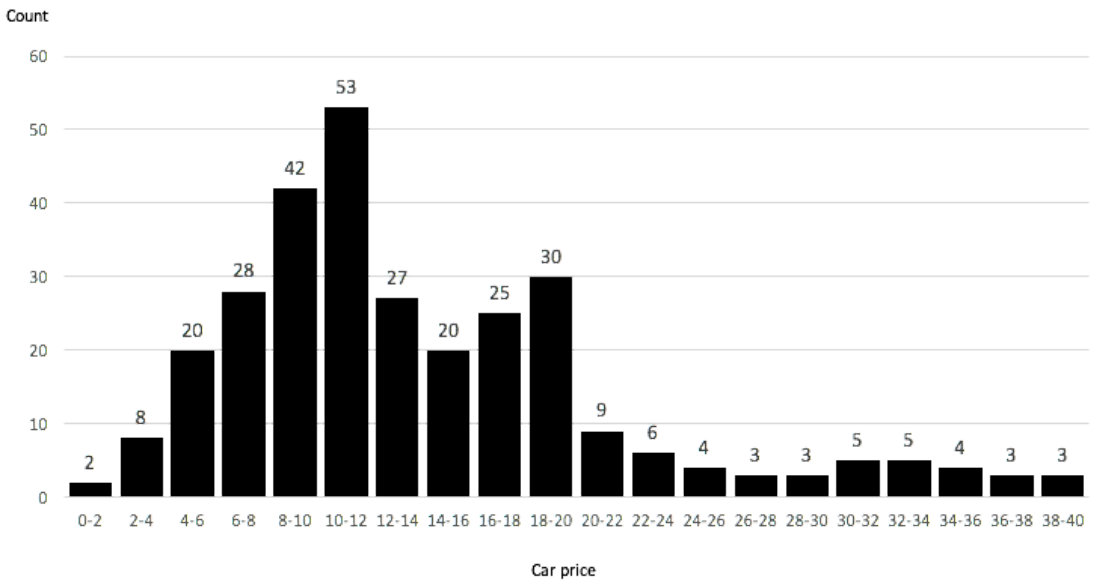


b) (3 pts) Consider the following query:

```
SELECT * FROM CAR_SALES WHERE price <= X;
```

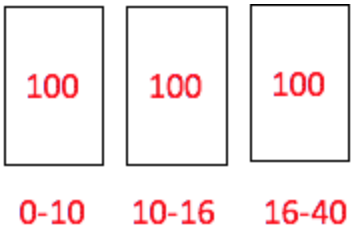
What X value can make 290 as its estimated number of records based on the equi-width histogram in a)?

35



The original graph is copied here for you convenience.

c) (3 pts) Draw a **3**-bucketed equi-height histogram.



d) (3 pts) Consider the following query:

```
SELECT *
FROM CAR_SALES
WHERE price BETWEEN 36 AND 40;
```

What is the estimated number of results from this query based on the equi-height histogram in c)? You can leave your answer as a formula.

4/24 * 100

Question 7: System-R Optimizer (20 points)

Consider the following relations about products, customers, suppliers, and their relationships, together with available B+ tree indexes:

- Products(pid, pname, price)
  - Clustered index on "**pid"** (meaning "product id");
  - Unclustered index on "**price";**
- Customers(cid, cname, email, state)
  - Clustered index on "**cid"** (meaning "customer id");
  - Unclustered index on "**cname";**
- Buys(cid, pid, date)
  - With information about what customers bought what products and when;
  - Clustered index on "**pid"**;
  - Unclustered index on "**cid";**
- Suppliers(sid, sname, telephone, state)
  - Clustered index on "**sid"** (meaning "supplier id");
  - Unclustered index on "s**name";**
- ProductBySuppliers(pid, sid, price)
  - Clustered index on "**pid"**;
  - Unclustered index on "**sid".**

The meanings of the tables and attributes are self explanatory.

Consider the following query:

```
SELECT sid, SUM(P.Price)
FROM Products P, Buys CP, Customers C,ProductBySuppliers PS,Suppliers S
WHERE P.pid=CP.pid AND CP.cid=C.cid AND P.pid=PS.pid AND PS.sid=S.sid
      AND P.price > 200 AND C.state = 'california'
GROUP BY S.sid
```

We want to use the techniques in the System-R optimizer to generate an efficient physical plan.

a) (2 pts) Write the meaning of the query in plain English.

For each supplier, for all its products with a price more than $200 and sold to a California customer, compute their total price.

b) (6 pts) For each of the following tables, write down all the access methods considered by the optimizer. Specify which of them will be considered for the next phase and explain why. Write down all the available interesting orders for each access method.

- **Products(pid, pname, price)**
  1) B+ tree scan on pid, interesting order on pid; kept;
  2) B+ tree search on price for the condition “price > 200”; kept if its cost is less than the access method 1).
  3) B+ tree search on pid for a constant, kept for a later index-based join

- **Customers(cid, cname, email, state)**

  1) B+ tree scan on cid, interesting order on cid; kept;
  2) B+ tree search on cname for the condition “state=CA”; kept if its cost is less than the access method 1);
  3) B+ tree search on cid for a constant, kept for a later index-based join

c) (4 pts) Explain how the optimizer generates efficient access methods for joining the tables **Products** and **Buys**. No need to show all the enumerations.

1) Products join Buys: For each access method on Products kept from the previous step, for each access method on Buys kept from the previous step, consider all possible valid join methods: block nested loop join, index-based join, sort merge join, hash join, etc. Estimate the cost of each subplan. Use the interesting order from the previous method, if any, when estimating the cost of a sort-merge join. For each interesting order, select the join method with the smallest cost, and remove those subplans that are dominated by another subplan in terms of both cost and interesting order(s).
2) Repeat the same step for Buys join Products;

d) (4 pts) Write down all possible 4-relation subsets considered by the optimizer by combining the subplans from previous passes. The join order of each subset doesn't matter.

1) (C, CP, P, PS)
2) (CP, P, PS, S)
3) (C, CP, PS, S)

Notice that CP and PS can still join their "pid" attributes. If your answer assumes CP and PS do not join when P is not included, we can also accept an answer that does not include "3)".

System-R optimizer does not consider cross products. So a subset that produces a cross product is wrong.

e) (4 pts) Use this example to briefly explain at least four main ideas in the System-R optimizer as covered in our lectures.

- Interesting orders
- Left-deep trees only
- Avoid Cartesian products
- Dynamic programming to do plan enumeration
- Selection push down
- Deal with group by at the end
- Deal with nested subqueries as blocks, and optimize them separately (not shown in the example)

Question 8: Extra-Credit Question (15 points)

Suppose you're a DBA for an e-commerce company, which has two separate database systems, one for online transaction processing (OLTP), and one for online analytical processing (OLAP). You've set up an incremental migration utility that moves new data from the OLTP system to the OLAP system on a daily basis. Suppose the OLAP system already has a B+ tree with 1,000,000 leaf-node pages, each of which can hold 1,000 records.

a) (5 pts) Suppose we want to insert a large number (5 million) records into the B+ tree. A simple method ("method 1") is to insert them to the index one by one, but this method is not efficient. We want to use another method ("method 2") based on bulk loading covered in class. A main problem is that bulk loading assumes an **empty** B+ tree. Come up with this "method 2" by extending the bulk loading idea, and briefly explain why it's more efficient than "method 1".
Sort the new records. Perform a full scan over the original B+ tree. Merge the results of sorted new records with the results from scanning B+ tree to bulk load a new B+ tree.

Method 2 still does not solve all problems. After the system is running for a few months, it starts taking longer and longer to migrate daily data. ~~Y~~ou come up with another idea of storing new records of each day into a separate B+ tree, which only takes minutes to finish. This method can generate more and more B+ trees over time.

b) (5 pts) Discuss how to deal with "`deleteRecord()`" in this design without modifying those existing B+ trees.
Use a tombstone record to indicate that a key has been deleted.

c) (5 pts) One drawback of this approach is that over time we can have a lot of B+ trees, which need to be accessed to do a search. To solve this problem, we want to periodically merge multiple B+ trees into one. Develop an algorithm to do the merge efficiently. (Hint: be careful about those deleted and updated records in the B+ trees.)
Perform a full scan over multiple B+ trees to get sorted records. During the full scan, use a priority queue to merge results so that a key can be returned at most once. If a key has been deleted (as a tomebstone), we ignore that key.
For the sorted records, we bulkload a new B+tree.