CSE-4411A Test #1

Sur / Last Name:
Given / First Name:
Student ID:

- **Instructor:** Parke Godfrey
- **Exam Duration:** 75 minutes
- **Term:** Fall 2010

Answer the following questions to the best of your knowledge. Your answers may be brief, but be precise and be careful. The exam is closed-book and closed-notes. Calculators, etc., are fine to use. Write any assumptions you need to make along with your answers, whenever necessary.

There are four major questions, each with parts. Points for each question and sub-question are as indicated. In total, the test is out of 50 points.

If you need additional space for an answer, just indicate clearly where you are continuing.

Marking Box	
1.	/10
2.	/15
3.	/10
4.	/15
Total	/50

1. (10 points) **Buffer Pool & Operations.** *We're on the same page.* [SHORT ANSWER]

a. (6 points) Consider that you have a buffer pool that can hold three pages. There are 26 pages in the database on disk, labelled as *A*... *Z*. Consider each of the sequences of page requests (*access patterns*) below and say the number of I/Os that would be used for the **LRU**, **MRU**, and **Clock** replacement policies. (Spaces in the access patterns are only for readability.) Assume that

- you start with an empty buffer pool each time;
- for **Clock**, the frame pointer starts with frame 0 each time;
- no page is written on (made *dirty*);
- the page is pinned and immediately unpinned each time; and
- the timestamp is updated each time the page is pinned (for **LRU** and **MRU**).

access pattern	**LRU**	**MRU**	**Clock**
ABC DAB CD	8	5	8
JKL MLJ NK	7	6	7

b. (4 points) Consider that an operation that uses a given B+ tree index repeatedly for equality searches (*probes*); e.g., first for *aardvark*, then for *badger*, then for *capibara*, and so forth. Assume that the values for which the operation probes are in sorted order.

What replacement policy would be most beneficial for this, and why?

MRU. Each probe starts with the root page of the tree. Then fetches the appropriate second-layer index page, and so on. Then fetches the data entry page.
For each next probe, the data-entry page fetched is likely different. But the root page fetched is always the same. LRU would be replacing the root and likely common first few index pages in the branch, leaving the data-entry page which is not likely to be reused!
So LRU would work poorly. MRU would work well here.
Some people thought the question implied we would go through this list multiple times, like repeated scans. (That is not what I intended.) They said sequential flooding then could be a problem, thus MRU would be better.

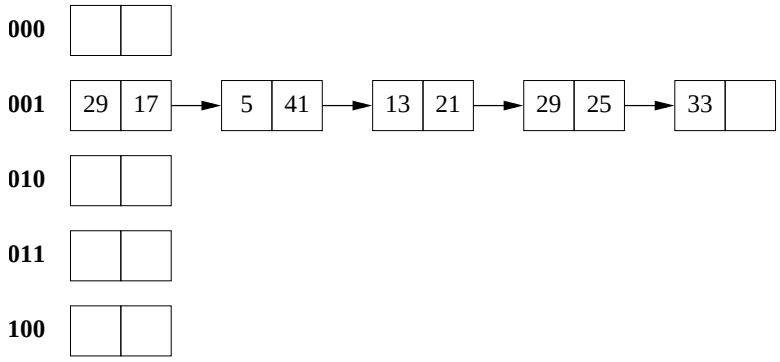
2. (15 points) **Indexes & Index Mechanics.** *Index Zoology.* [Exercise]

Consider the data values *giraffe* (29), *capibara* (17), *badger* (5), *impala* (41), *elephant* (13), *aardvark* (21), *hedgehog* (29), *donkey* (25), and *fox* (33). The number after each represents its hash value using the hash function **h**.

---

a. (4 points) Consider a linear hash index that has hash buckets that have room for two entries each, and that starts with a single bucket. (So initially, *zero* bits are used to find an item.)

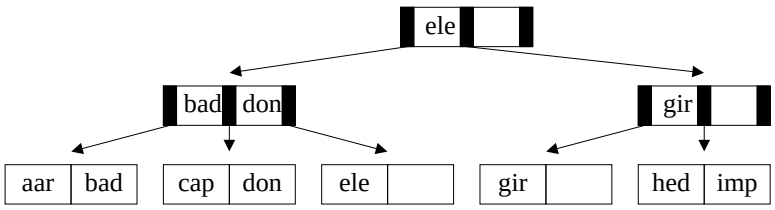
Show how the linear hash index would look after adding the nine entries from above (*giraffe*, *capibara*, ...) in that order using **h** as the hash function. Assume a split rule that splits once per overflow occurrence.



This is with a split every time a new overflow page is made, not every time something is added to an overflow page! Also, note that buckets are created in hash-index sequence.

b. (4 points) Consider a B+ tree index of order one that initially consists of an empty root node.

Show a possible resulting B+ tree adding the nine entries from above (*giraffe*, *capibara*, ...) in that order. (Of course, the hash values do not apply here.) Use usual split rules and redistribution.



You can get something that looks different, depending on redistribution. This also uses the choice that '<=' goes to the left, '>' to the right.
I abbreviated the names to three letters just for breviety. That might actually be done in the index pages for key compression, but would not be done in the leaf pages.
*Note that the values (*elephant*, etc.) are used in the index key. The hash values from 2a have nothing to do with a tree index!*

c. (2 points) Are there any anomalies in the indexes created in 2a and 2b?
If so, what are they, and what might be done to fix them?

There is a problem with the hash index in 2a. All the values are hashing to the same bucket! There is bad skew for our values with this hash function. We should choose a new hash function.
The tree index in 2b seems to be behaving fine.

(Scratch space.)

As the database administrator, you observe that there are many queries on the table **Student** that have an equality predicate on major (e.g., major = 'Computer Science') and a range predicate on gpa (e.g., gpa $\geq$ 6.5).

d. (3 points) What type of index—tree versus hash, clustered versus unclustered—with what search key would you suggest to create, and why?

A tree index, since a range (gpa) is involved. The search key can be on major, gpa, since the two predicates (equality, then range) will prefix match into that.
Since the GPA range (within a given major) could match many records, it would be beneficial that the index be clustered, to reduce the I/O's for retrieving the matched records. (However, we would have to weigh this against the benefits of having another index on **Student** *clustered instead. Also, if we know the queries are highly selective—match few records—it would not matter.)*

e. (2 points) If you also knew that the queries often need to return st# (**Student**'s primary key), name, and gpa—but just those columns—and that your index had to be unclustered, what might you propose?

We can add those columns to the index search key. That is, major, gpa, st#, name. Then the queries could be answered as index-only.

3. (10 points) **General.** *Catch a tiger...* [Multiple Choice]

Choose *one* best answer for each of the following. Each is worth one point. There is no negative penalty for a wrong answer.

---

a. Physical database independence has the consequence that

A. applications cannot access records directly, but only via queries.
B. applications need not know the schema of the database to compose queries.
C. records from the same table have to be stored within the same file.
D. indexes must be used to access the data.
E. pointers cannot be used internally in the database system.

---

b. Relational database management systems (RDBMSs) typically implement their own buffer pool managers rather than using the operating system's (OS's) facilities because

A. OS's do not handle paging between disk and main memory.
B. they need control over when a page is written back to disk.
C. an RDBMS's buffer pool manager can page faster than the OS's facilities can.
D. paging by the OS has no replacement policy.
E. it is easy enough to implement, so why not?

---

c. Which of the following changes would *not* make researchers re-evaluate standard database system design?

A. CPU's become significantly faster.
B. CPU's with hundreds of cores become common.
C. Inexpensive, fast, non-volatile main memory becomes commonly available.
D. Computers are standardly configured with a petabyte (10^15 bytes) of RAM.
E. Relational schemas with thousands of columns per table become commonplace.

---

d. Which of the following is *false*?

A. Many records fit on a page, on average.
B. Sequential reads and writes are important to a database system's performance.
C. I/O time usually dominates CPU time in database operations.
D. Page size is determined by the query.
E. The technology trend is that the ratio of CPU speeds to disk I/O speeds is growing over time.

---

e. Indexes are important for all but which of the following reasons?

A. They help check integrity constraints.
B. They can speed up selections.
C. They can be useful for joins.
D. They provide useful statistics to the query optimizer.
E. They reduce the storage overhead of the database.

f. How do index *search keys* differ from logical *primary keys*?

A. The order of the columns in a search key makes a difference.
B. Search keys do not imply uniqueness.
C. One table can have many indexes, thus many search keys.
D. All of the above.
E. None of the above.

---

g. Consider

I. unclustered tree indexes
II. clustered tree indexes
III. unclustered hash indexes
IV. clustered hash indexes

Equality match queries can benefit from

A. Just **I**.
B. Just **I** & **II**.
C. Just **I**, **II**, & **IV**.
D. Just **II** & **IV**.
E. Potentially any of **I**, **II**, **III**, & **IV**.

---

h. Alternative #1 for index organization is usually not implemented in practise for tree indexes because

A. when records are redistributed on leaf-page splits, the rid's can change.
B. if the data records are part of the B+ tree, no second index on the table is possible.
C. composite search keys cannot be accommodated with alternative #1.
D. data records are bigger than data entries, so fan-out is reduced.
E. it is impossible to do key compression under alternative #1.

---

i. Variable length fields mean records are variable length. This has the consequence that

A. the buffer pool manager must support variable length frames.
B. different records from the same table can have different numbers of fields.
C. the different fields of the same record must be kept on different pages.
D. slot#'s cannot be determined as fixed addresses on the page, so a slot directory on each page is necessary.
E. B+ tree indexes are not possible for these records because the order of the B+ tree cannot be determined.

---

j. Using replacement sort instead of quicksort for pass zero of the external sort algorithm has the advantage that

A. it is faster than quicksort.
B. it allows for sequential reads, whereas quicksort does not.
C. it produces runs twice as long, on average, as the use of quicksort does.
D. it may reduce the number of I/O's for pass zero, compared with using quicksort.
E. it may reduce the number of I/O's for subsequent passes, compared with using quicksort.

4. (15 points) **External Sorting.** *Do run, do run, do run.* [Analysis]

The standard method works in passes. *Pass 0* sorts blocks of the file into runs. Subsequent passes are *merge passes*. Assume we have B buffer frames allocated for the job. Within a merge pass, each merge step merges $B-1$ runs from the previous pass to produce a new run. The procedure ends in the pass that merges the final $B-1$, or fewer, runs into a single run.

We need not think in terms of separate merge passes, however. Rather, we can think of it just as a sequence of merge steps. The standard external sort routine can be implemented by having each merge step take the $B-1$ *oldest* runs produced—or fewer, if fewer than $B-1$ runs remain—to merge together into a new run. We can call this choice of runs to merge *least recently made*. We repeat the merge step until just one run remains.

Dr. Datta Bas has a suggestion that he believes is an improvement over the standard external sort routine. Pass 0 is the same as before (say by quicksort). But his choice of runs to merge each time in a merge step is *most recently made*. That is, each merge step will take the most recently made $B-1$ runs—or fewer, if fewer remain—to merge.

---

a. (5 points) Consider the buffer pool allocation B to be 5, and a file of 100 pages. Calculate the I/O cost to sort them by the external sort algorithm using the merge selection of *least recently made*, which is *almost* the standard method as described in the textbook.

Pass 0: Makes 20 runs of 5 pages each.

"Pass 1" of merges: 5 times:

Merges 4 oldest runs (5 pages long each) to make a run of 20 pages.

"Pass 2" of merges: 1 time:

Merges 4 oldest runs (20 pages long each) to make a run of 80 pages.

"Pass 3" of merges: 1 time:

Merges oldest remaining run (20 pages long) with run just made of 80 pages to make the finished run of 100 pages.

This is quite similar to the regular algorithm, which merges runs from the previous pass, effectively the least recently made.
Our cost here is 760 I/O's. We save a bit in "Pass 2" as there is simply a single run left after merging four of the five runs. So nothing to merge for it!
For the regular algorithm, we would note it takes four passes. Each pass reads all and writes all, so $4 \times 2 \times 100 = 800$ *I/O's. (In truth, the regular algorithm could save the 40 I/O's too in this example as we do, implemented sensibly.)*
Note we can dispense with the notion of merge passes*, and just look at how many merges are done, and how long the runs are.*

b. (5 points) Again, consider the buffer pool allocation B to be 5, and a file of 100 pages. Calculate the I/O cost to sort them by the external sort algorithm using the merge selection of *most recently made*, Dr. Bas's suggested method.

```
Pass 0: Makes 20 runs of 5 pages each.      200 I/O's
Merge 4 5-page runs.                         40 I/O's
Merge 3 5-page runs & the 20-page run.       70 I/O's
Merge 3 5-page runs & the 35-page run.      100 I/O's
Merge 3 5-page runs & the 50-page run.      130 I/O's
Merge 3 5-page runs & the 65-page run.      160 I/O's
Merge 3 5-page runs & the 80-page run.      190 I/O's
Merge final 5-page run & the 95-page run.   200 I/O's
                                          1,090 I/O's
```

It is more expensive!

---

c. (5 points) Is Dr. Bas's version more efficient or less efficient than the standard external sort routine, in general?

Explain convincingly.

It is much less efficient! It is effectively $\mathcal{O}(N^2)$, not $\mathcal{O}(N \ln N)$.

(Scratch space.)

Relax. Turn in your exam. Go home.