2. **Index Use.** *Consumer Price Index.* (10 points) [Exercise]

Table **R** has a *clustered* tree index of type alternative #2 on A, B, C, D. (Assume **R** has additional attributes; e.g., E, F, . . .) The index pages contain 133 *index records* (encompassing 134 pointers), on average;[1] data-entry pages contain 50 data entries each, on average; and data-record pages contain 20 data records each, on average.

A's values range over 1..10,000; B's over 1..1,000; C's over 1..100; and D's over 1..10.

For Questions 2c & 2d, assume "smart" processing; that is, that the processor would minimize I/O usage with the selection information.

a. (3 points) An index record contains effectively the information A, B, C, D, and a pointer (an address to another page). A data entry contains effectively A, B, C, D, and an RID (which is an address to another page *and* a slot number). A data entry is *slightly* larger than an index record—by a slot number—but only slightly. So explain how it is possible there are 133 index records per page, on average, but *only* 50 data entries per page, on average.

Key compression.

b. (2 points) Say that table **R** has 1,000,000 records. How deep is the index tree?

Fan-out is 134. We must index 20K pages. $134^2 < 20K$ *but* $134^3 > 20K$. *So we need three levels of index pages (the root plus two more) to have page-ID "pointers" to the 20K data-entry pages.*
These three index-page layers plus the data-entry layer means the tree is four pages deep.

[1] This accounts for the *fill factor*. A page could hold more than 133 index records.

c. (3 points) Estimate the I/O cost of

```
select * from R where A = 1111 and B > 700;
```

using the index as the access path.

$1M/10K = 100$ matching A. 30% (1000 − 700)/1000) of these match B. So, 30 matching records, in all.
Four I/O's to get to the first $A = 1111 and B > 700$ data entry. All 30 matches likely on the same data-entry page. 2 I/O's to fetch the data-record pages containing the 30 records. (Clustered, 20 records per page.) So 6 I/O's predicted.

d. (2 points) Estimate the I/O cost of

```
select * from R where A > 9500 and C > 90;
```

using the index as the access path. (Assume "smart" processing.)

$1M \cdot 500/10K = 50K$ matching records for $A > 9500$. Cannot match for $C > 90$ by the index (since B intervenes in the search key and there is no equality predicate on B in the query). But $10/100 = 10\%$ of the 50K will match also for $C > 90$ and need fetching; so 5K record fetches.
4 I/O's to the data-entry page with the first $A > 9500$ entry. Scan $20K/20 = 1K$ of data-entry pages. When $C > 90$, also fetch the record: 5K fetches costing 2.5K I/O's as we can estimate there are two matching records per fetched data-record page and the index is clustered.
So 3,504 I/O's are estimated in all.

2. (10 points) **Index Logic.** *Take the next index to the left.* [short answer / exercise]

a. (5 points) You are told that the following indexes are available on the table **Employee**:

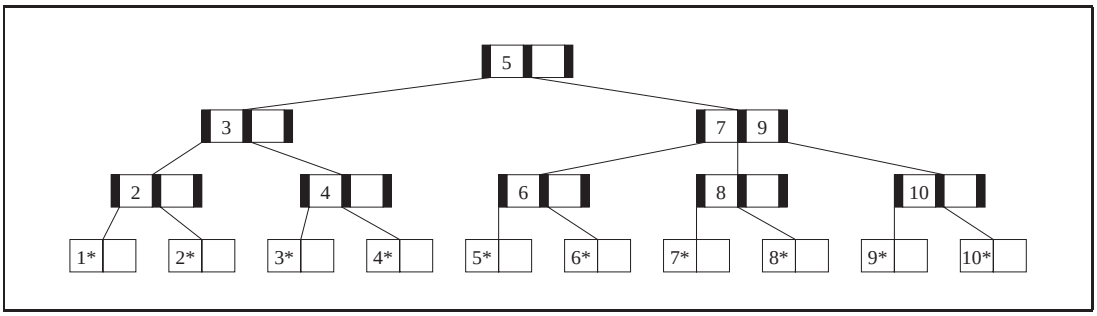
	key	type	clustered?
A.	name, address	tree	yes
B.	age, salary	hash	no
C.	name	tree	no
D.	salary, age	hash	no
E.	name, age	tree	yes

You are suspicious that this information is not correct. Why? Identify three problems with what is reported.

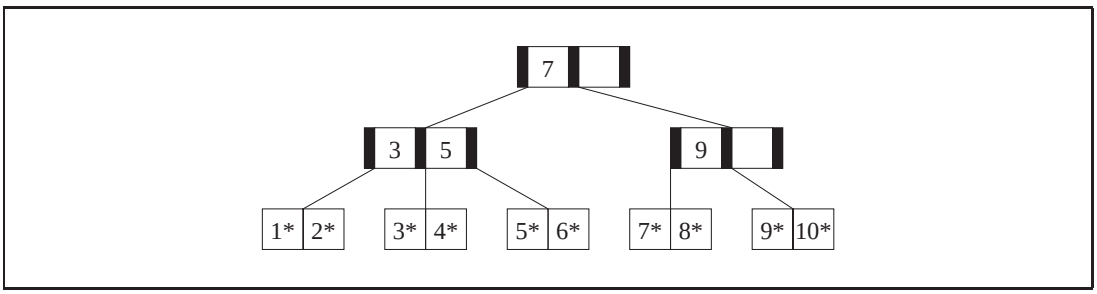
- *It is impossible that **C.** is unclustered if **A.** or **E.** are clustered.*
- *It is not possible to have two clustered indexes on the same table with different keys: **A.** & **E.***
- *It makes no sense to have **B.** and **D.**. They are identical functionally.*
- ...

4. (10 points) **B+ Trees.** *How low can you go?* [short answer / exercise]

a. (3 points) Consider a B+ tree of order one. Construct an example of B+ tree with the *worst* (largest) possible depth that indexes 10 data records.

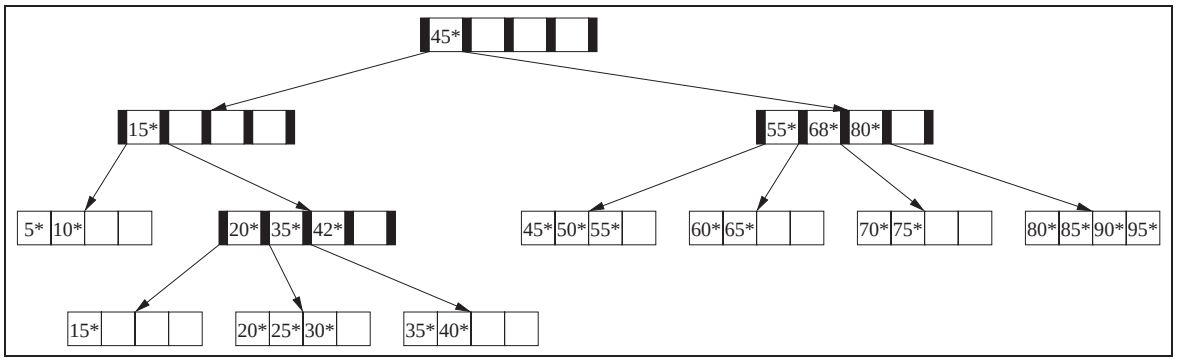


b. (3 points) Consider a B+ tree of order one. Construct an example of B+ tree with the *best* (smallest) possible depth that indexes 10 data records.



Some people have argued that for 3a, ***C*** *is also a good answer. It is true that blocked I/O is used for prefetching. However, prefetching and blocked I/O are different. Prefetching still can be advantageous, even if not done with blocked I/O. And block I/O is used for many things in databases than prefetching.* ***A*** *is a correct answer, and a better answer than* ***C****.*

c. (4 points) Identify four distinct problems with the following "B+ tree".



i. *Not all nodes are at least half full.*

ii. *Record 55 is out of place. By the apparent convention in the tree, it should be to the right of key 53 in the index node. (Otherwise, many other records are misplaced.)*

iii. *In a B+ tree, all leaves should be at the* same *depth. This is not the case here.*

iv. *There is no leaf page to the right of key 42. Either the key should not be there, or there should be a leaf page.*

v. *Keys in index pages are marked with ∗'s, indicating these are data records themselves. Data records are not stored in index pages in a B+ tree.*

5. (10 points) **Linear Hash Indexes.** *Coming up empty.* [analysis]

a. (5 points) Dr. Dogfurry, infamous database researcher, has noted that sometimes when a new bucket creation is triggered—whenever an overflow page has been made—*no* keys redistribute into the new bucket. He reasons that it does not make sense to make the new bucket in such cases.

So the algorithm should check whether redistribution would put anything into the new bucket; if not, a new bucket is not made, and `next` is not advanced.

What is wrong with Dr. Dogfurry's change? What bad consequences would this have?

This is quite bad. The #buckets will not grow until *a record is added to that very bucket in question that would result itself redistribute to the new bucket if this bucket were split. In the meantime, long overflows can develop on the other buckets, greatly degrading performance.*
Namely, Dr. Dogfurry is not understanding how linear hashing works. The `next` *bucket needs to be spit when an overflowoccurs* in order to *ensure that, on average, the #overflows is less than one.*

4. (15 points) **Query Evaluation.** *What* do *polysci students take?!* [exercise / analysis]

   Consider tables

   - **student**(id, name, major) with 100,000 records on 2,000 pages
   - **enrol**(id, course#, section, term, grade) with 4,000,000 records on 40,000 pages

   There is a foreign key from **enrol** onto **student**.

   Available indexes:

   - hash index on **student** on id (linear hash)
   - clustered tree index on **enrol** on id, course#, section, term (index pages are 3 deep)
   - unclustered tree index on **enrol** on course#, id (index pages are 3 deep)

   Statistics:

   - the number of values of **student**.major: 100
   - values of **enrol**.course#: 1000, ..., 4999 (so 4000 values)

   Consider the query

```
select name, S.id, course#, section, term, grade
    from student S, enrol E
    where S.id = E.id
      and course# is between 4000 and 4999
      and major = 'political science';
```

   You have an allocation of 25 buffer frames.

---

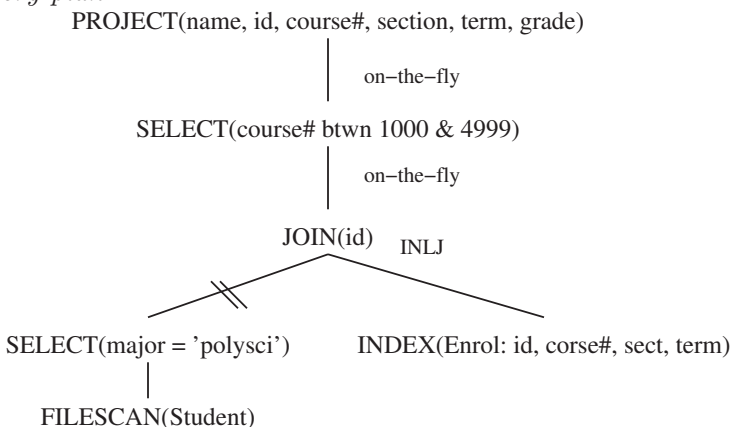
a. (3 points) Estimate the cardinality of the query.

> **E** *has a foreign key on* **S**, *so we start with 4,000,000 "***ES***" records. The reduction factor (RF) of* $\sigma_{course\#}$ *is* $\frac{1000}{4000}$, *or* $\frac{1}{4}$. *RF of* σ_{major} *is* $\frac{1}{100}$. *Therefore, the cardinality estimation of the query is* $4,000,000 \cdot \frac{1}{4} \cdot \frac{1}{100} = 10,000$.

b. (7 points) Devise a good query plan for the query. Show the query tree, *fully* annotated with the chosen algorithms and access paths.

Estimate the cost of your plan. For full credit, you should have a plan that costs less than 10,000 I/O's.

Here is one query plan.



σ: 2,000 I/O's to filescan **S**. *1,000 records* ($\frac{1}{100}$) *survive.*

⋈: *1,000 probes. Each costs*

- *3 I/O's Index pages*
- *1 I/O data-entry (DE) page*
- *1 I/O data-record (DR) page*

Each probe matches 40 records. Since the index is clustered, all are likely on the same DR page.

Total: 7,000 I/O's
If we assume that the root of the probe index stays in the buffer pool (BP) between probes (a quite reasonable assumption), then each probe is just 4 I/O's, and the total is 6,001 I/O's.

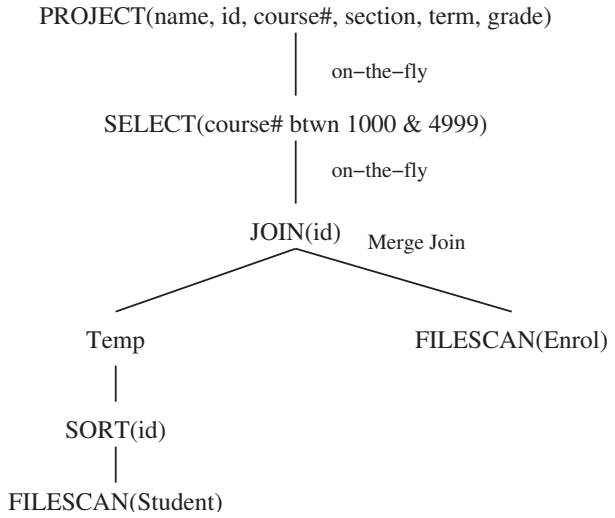
c. (5 points) Consider the query

```
select name, S.id, course#, section, term, grade
    from student S, enrol E
    where S.id = E.id
      and course# is between 4000 and 4999;
```

This is the same as before, but with the "`major = 'political science'`" condition dropped.

Show a good query plan (annotated tree) for this query. (By good, I mean better than picking just any naïve approach.) What is its cost?

Here is one query plan.



Enrol *is sorted on* **id** *because of its clustered index.*

- ***Sort****: Pass 0 is 80 runs plus two merge passes, so 24,000 I/O's.*
- ⋈*: Read* **Temp** *at 2,000 I/O's and read* **Enrol** *at 40,000 I/O's. Total: 42,000 I/O's*

Total: 76,000 I/O's

2. [10pt] **Buffer Pool.** *Learning to swim.* EXERCISE

a. [2pt] LRU (*least recently used*) is known to be generally a good buffer-pool replacement strategy in support of most SQL operations.
Why?

It is the principle of data locality*. If a piece of data is needed for an operation, it is more likely to be needed again soon by the operation. This is generally true for SQL queries.*

2pt data locality
1pt partial explanation in right direction

b. [3pt] Spell out the *steps* that the buffer-pool manager needs to make to handle a *pin* call; e.g., pin(1729). Assume the replacement policy is LRU.

```
1  IF 1729 is in the buffer pool THEN
2      remove from replacement queue (or the like)
3      increment the frame descriptor's pincount
4      return frame address to caller
5  ELSE
6      find a frame to replace by replacement policy
7      IF none, throw exception
8      IF page in chosen frame is 'dirty' THEN write it to disk
9      READ 1729 into selected frame
10     set frame descriptor's pincount to 1
11     return frame address to caller
```

+1pt replacement
+1pt pins, writes old page, if needed
+1pt returns frame address

4. [10pt] **Index Mechanics.** *It's the carburetor, of course.* EXERCISE

   Assume the search-key rule to go *left* if '<' and to go *right* if '≥'.

---

a. [2pt] You conduct some disk forensics and you discover a B+ tree structure with the data as in Figure 1.

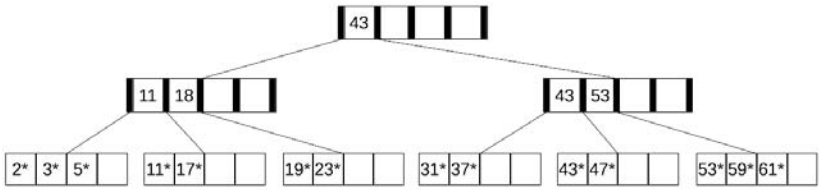


Figure 1: Recovered forensics B+ tree.

You suspect there is something wrong with this B+ tree. What is wrong?

Search keys in index pages should be moved up*, not* copied up*. So that* 43 *appears twice has to be a mistake. Indeed, it is; Note that records* 31∗ *and* 37∗ *should be to the* left *of the root, but they are not. They could not be located.*

+2pt That 43 appears twice. Or that records 31∗ and 37∗ are less than 43 but are misplaced.

---

b. [3pt] Add a record with search-key value F to the B+ tree in Figure 2.

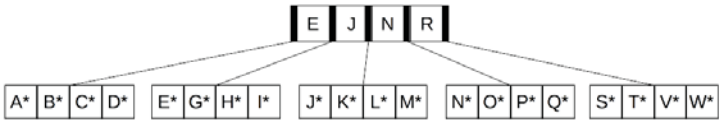
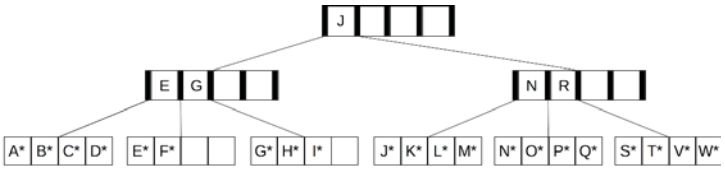


Figure 2: B+ tree for addition.



There can be other legal configurations that work. But now the tree is

- two *deep in index levels,*
- *E, F, G, H, and I were distributed, and*
- *we had a proper* copy up *of G and* push up *of J.*

+1pt each for the points above.