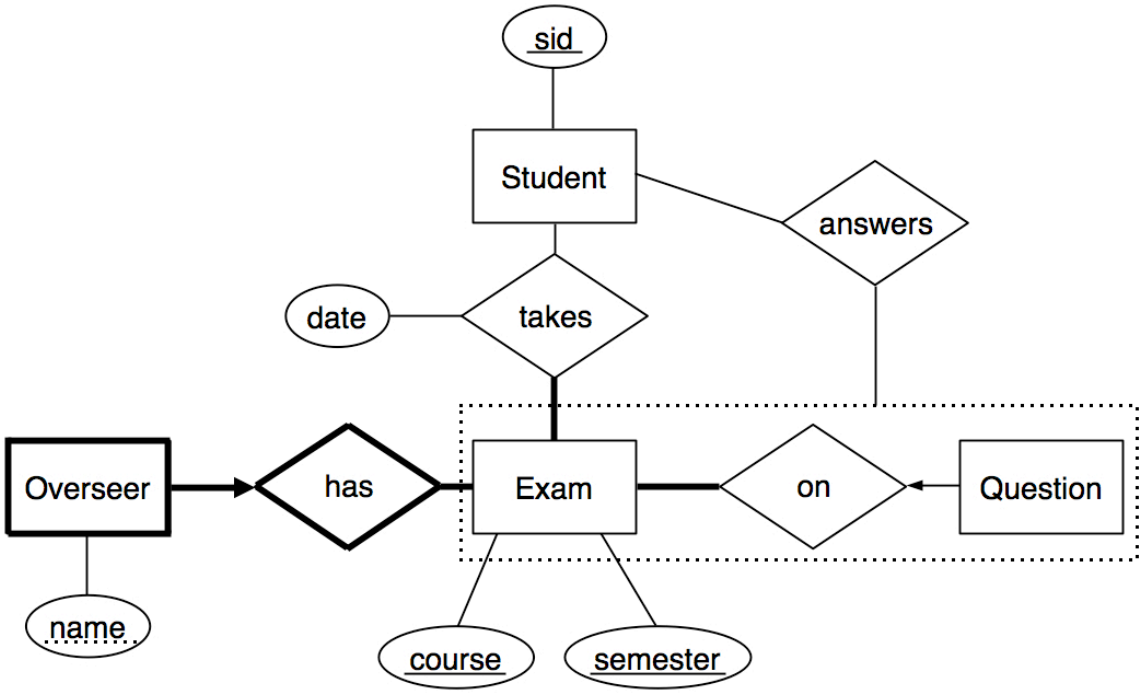
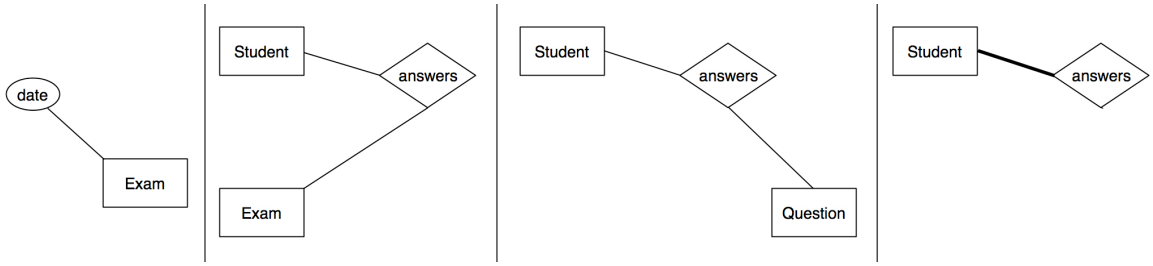
Midterm Exam: Introduction to Database Systems: Solutions

1. **Entity-Relationship Model [16 points]**

Below is the preferred solution:



The following variants were also accepted:



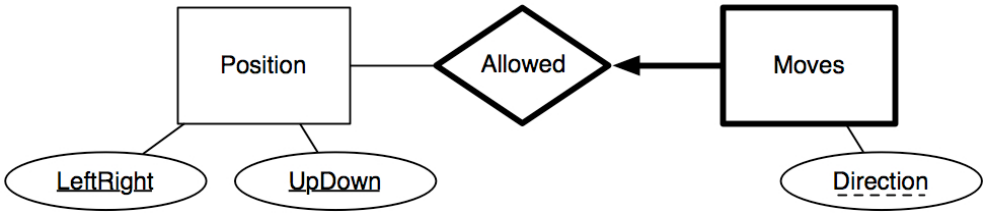
a. [12 points] Complete the diagram above to be a valid E-R diagram reflecting the following constraints. (Be sure to make your bold lines **very bold**!)
   - A student, uniquely identified by her SID, takes an exam (for example, Midterm #1) on exactly one date. A student may take any number of exams (for example, Midterm #1, Midterm #2, and a final exam), and every exam is taken by at least one student. An exam is uniquely identified by the combination of a course and a semester.

- Every exam has at least one overseer (for example, Prof. Hellerstein, Derrick, and Bill). An overseer is uniquely identified by an exam and the overseer's name.
- There is at least one question on every exam, and a question appears on at most one exam. A question on an exam may be answered by any number of students, and a student may answer multiple questions on an exam.

Points for question 1(a) were assigned according to the following rubric:

+1	*"name" is underlined with a dotted line and connected to "Overseer" with a regular line.*
+1	*"Overseer" and "has" are bolded, and there is a bold arrow from "Overseer" to "has"*
+1	*"Exam" and "has" are connected with a bold line*
+1	*"Exam" and "takes" are connected with a bold line*
+1	*"Exam" and "on" are connected with a bold line*
+1	*"course" and "semester" are underlined with solid lines. "Exam" and "course" are connected with a solid line. "Exam" and "course" are connected with a solid line.*
+1	*"Question" is connected to "on" with a regular arrow*
+1	*"date" is connected to "takes" (preferred) or to "Exam" with a regular line, and is not underlined.*
+1	*"takes" is connected to "Student" with a regular line*
+1	*"sid" is underlined and connected to "Student" with a regular line*
+1	*"Student" is connected to "answers" with either a regular or a bold line*
+1	*"answers" is either connected to "Exam", "Question", or to an aggregate surrounding "Exam", "on", and "Question", with a regular line.*
-1	*Extraneous markings were included, such as bolding a relation other than Overseer or underlining a relation.*

- [2 points] Consider the following E-R diagram, which is a fragment of a simple board game schema that captures the legal moves available in each position on a board:

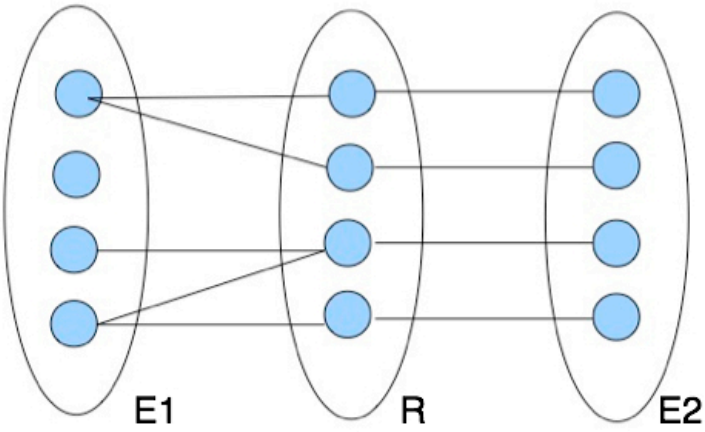


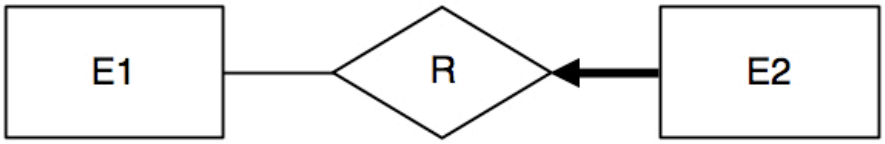
We want to translate "Moves" into an SQL table and maintain the constraints in the ER diagram. Correctly complete the SQL statement below (note that "--" begins a comment in SQL):

```
CREATE TABLE Moves
   (direction CHAR(2), -- one of N/S/E/W/NE/NW/SE/SW
    LeftRight CHAR, -- one of A-P
    UpDown INTEGER, -- one of 1-16
    PRIMARY KEY(Direction, LeftRight, UpDown),          (1 pt)
    FOREIGN KEY (LeftRight, UpDown) REFERENCES Position  (1/2 pt)
      ON DELETE CASCADE                                  (1/2 pt)
);
```

- [2 points] The figure with ovals and circles below illustrates a relationship set between two entity sets. Each circle is an entity or relationship, and the edges connect them. The ER diagram right below that figure needs to be completed with constraints that are consistent with the edges in the diagram. For each of the following, circle the correct answer: *(1/2 pt each)*

  - A bold edge between E1 and R is : **A)** needed **B)** optional **C)** disallowed
  - An arrowhead between E1 and R is: **A)** needed **B)** optional **C)** disallowed
  - A bold edge between E2 and R is: **A)** needed **B)** optional **C)** disallowed
  - An arrowhead between E2 and R is: **A)** needed **B)** optional **C)** disallowed





2. **Query Processing [15 points]**
Consider the following relations:

```
CREATE TABLE Employee (SSN integer, DepartmentID integer,
  PRIMARY KEY (SSN),
  FOREIGN KEY (DepartmentID) REFERENCES Department);
```
(100,000 tuples; 1100 pages)

```
CREATE TABLE Department (DepartmentID integer, Name char(40),
  PRIMARY KEY (DepartmentID));
```
(1000 tuples; 50 pages)

And consider the following join query:

```
SELECT SSN, DepartmentID, Name
FROM Employee, Department
WHERE Employee.DepartmentID = Department.DepartmentID
```

Assume there are no indexes available, and both relations are in arbitrary order on disk. Assume that we use the refinement for sort-merge join that joins during the final merge phase. However, assume that our implementation of hash join is simple: it cannot perform recursive partitioning, and does not perform *hybrid* hash join. The optimizer will not choose hash join if it would require recursive partitioning.

For each of these questions, 2 points was given for the correct algorithm and 3 points for the correct cost. If the algorithm was incorrect but the cost was correct for the given algorithm, 1 point was given. One point was deducted if the name of an algorithm wasn't quite right. If the cost calculation contained a few minor errors, or was incomplete, 1 or 2 points were deducted.

- [5 points] Assume you have B=3 memory buffers, enough to hold 3 disk pages in memory at once. (Remember that one buffer must be used to buffer the output of a join). What is the best join algorithm to compute the result of this query? What is its cost, as measured in the number of I/Os (i.e., pages requested. **Do *not* include the cost of writing the final output.** You may ignore buffer pool effects in this question.)

    - Sort-Merge Join: Supposing number of passes to sort Employee is P_E and number of passes to sort Department is P_D, cost is $(2P_E - 1)|\text{Employee}| + (2P_D - 1)|\text{Department}|$ with the refinement and $(2P_E + 1)|\text{Employee}| + (2P_D + 1)|\text{Department}|$ without the refinement. Either answer was accepted, since the problem asked you to use the refinement but it's not technically possible in this case (not enough buffers).
      In this case $P_E = 10$ (at the beginning of each pass there are 1100, 367, 184, 92, 46, 23, 12, 6, 3, 2 sorted runs). Similarly $P_D = 6$ (runs are 50, 17, 9, 5, 3, 2). Credit was given if you were off by one in either case. Consequently the following are all valid answers for cost in this problem:
      19150, 19250, 19350, 21350, 21450, 21550, 21650, 23550, 23650, 23750, 23850, 25850, 25950, 26050
      (The actual answer is 23750)
    - Hash Join: Not applicable, since $B^2 = 9 < 50 = \min(|\text{Employee}|, |\text{Department}|)$. Could only be done with recursive partitioning, which was excluded.
    - Doubly-nested loop join: cost is NumTuples(Department)*|Employee| + |Department| = (1000)(1100) + 50 = 1100050 > 23750
    - Page-oriented doubly-nested loop join: cost is |Department|*|Employee| + |Department| = (50)(1100) + 50 = 55050 > 23750
    - Block nested loops join: B – 2 = 1, so same as page-oriented doubly-nested loop join.

- [5 points] Suppose that instead of having 3 memory buffers we have B=11 memory buffers. What is the best join algorithm now, and what is its cost (again, without writing final output or considering buffer hits)?
  - Hash Join: Since $B^2 = 121 > 50 = \min(|\text{Employee}|, |\text{Department}|)$, we can use hash join in this problem. We partition Employee into 10 partitions, then partition Department into 10 partitions (average size 5), then load each Department partition into memory while streaming through the corresponding larger Employee partition. Since there is no recursive partitioning, total cost is $3(|\text{Department}| + |\text{Employee}|) = 3(1100 + 50) = 3450$.
  - Sort-Merge Join: Supposing number of passes to sort Employee is P_E and number of passes to sort Department is P_D, cost is $(2P_E - 1)|\text{Employee}| + (2P_D - 1)|\text{Department}|$ with the refinement and $(2P_E + 1)|\text{Employee}| + (2P_D + 1)|\text{Department}|$ without the refinement. Either answer was accepted, since the problem asked you to use the refinement but it's not technically possible in this case (not enough buffers). Another option was to use the refinement on the Department relation but not the Employee relation, for a cost of $(2P_E + 1)|\text{Employee}| + (2P_D - 1)|\text{Department}|$.
    In this case $P_E = 3$ (at the beginning of each pass there are 1100, 100, 10 sorted runs) and $P_D = 2$ (runs are 50, 5). The following are valid answers for cost: 5650, 7850, 7950. These are more than the cost of hash join, but were still accepted because that solution was not known at the time of grading.
  - Block nested loops join: Cost is $(\text{ceiling}(|\text{Department}|/(B-2)) * |\text{Employee}|) + |\text{Department}| = (6)(1100) + 50 = 6650$. This is faster than Sort-Merge Join (since the refinement giving 5650 cost can't actually be used), but slower than Hash Join.
  - Doubly-nested loop join, page-oriented doubly-nested loop join: costs same as in first part above, both far more than the above options.

- [5 points] Suppose we raise the number of memory buffers to B=52, and increase the size of the Departments relation to 500 pages. What is the best join algorithm now, and what is its cost (no writing final output, ignoring buffer hits)?

  - Hash Join: Since $B^2 = 2704 > 500 = \min(|\text{Employee}|, |\text{Department}|)$, we can use hash join in this problem. Since there is no recursive partitioning, total cost is $3(|\text{Department}| + |\text{Employee}|) = 3(1100 + 500) = 4800$.
  - Sort-Merge Join: Number of buffers is now large enough to sort each relation in two passes, with enough room to do the refinement for both relations ($1100/52 + 500/52 = 32 < 52$). So cost is $3(1100 + 500) = 4800$, same as Hash Join.
  - Doubly-nested loop join: cost is $\text{NumTuples(Department)}*|\text{Employee}| + |\text{Department}|$ or about $(10000)(1100) + 500 = 11000500 > 4800$
  - Page-oriented doubly-nested loop join: cost is $|\text{Department}|*|\text{Employee}| + |\text{Department}| = (500)(1100) + 500 = 550500 > 4800$
  - Block nested loops join: Cost is $(\text{ceiling}(|\text{Department}|/(B-2)) * |\text{Employee}|) + |\text{Department}| = (500/(52-2))(1100) + 500 = 11500 > 4800$.

3. **Files and Buffer Management [10 points]**

- [3 points] Consider a buffer pool of 3 frames, and a heap file of 100 sequential pages with pageIDs from 1 to 100. Assume we scan the heap file **twice** from start to finish. Starting with an empty buffer pool, using an MRU replacement strategy, which pages will be in the buffer pool after the second scan?

  1, 99, 100. Initially the pool fills with 1, 2, 3. Then the 3 position is overwritten until the end of the first pass, at which point it is 1, 2, 100. During the second pass, 1 and 2 are hits, then the 2 position is overwritten until 99 is reached. Finally 100 is a hit.

- [1 point] What is the hit rate (#hits/#requests) in the scenario of part (a)?
  3/200 or 1.5%. Only 1, 2, and 100 are hit, once each.

- [3 points] To save a random I/O, your friend suggests that we scan the file once from pageID 1 to pageID 100, and then switch into reverse and scan from pageID 100 back down to 1. Again starting with an empty buffer pool and using MRU, what pages will be in memory at the end of this scan?

  1, 2, 3. Initially the pool fills with 1, 2, 3. Then the 3 position is overwritten until the end of the first pass, at which point it is 1, 2, 100. During the second pass, 100 is a hit, then the 100 position is overwritten until 3 is reached. Finally 1 and 2 are hits.

- [1 point] What is the hit rate (#hits/#requests) in the scenario of part (c)?

  3/200 or 1.5%. Only 100, 2, and 1 are hit, one each.

- [1 point] Consider a sorted file organization as we studied in class, with *N* tightly packed pages of records. Write an expression for expected (average case) number of I/O requests for an equality lookup in the file.

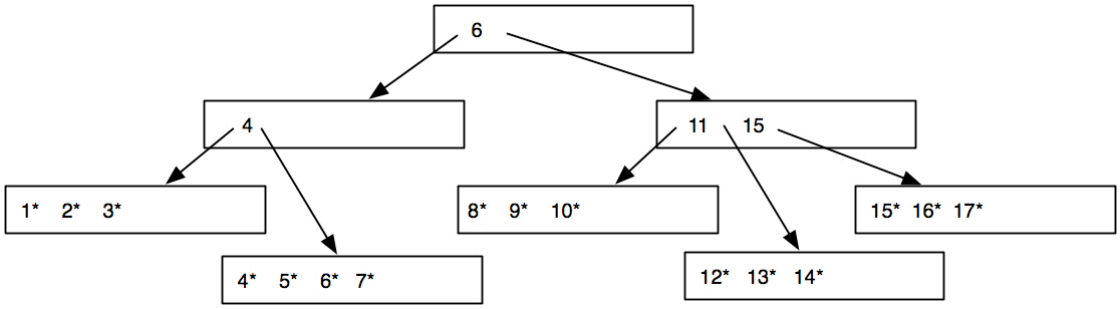
  $\log_2 N$. This can be achieved with binary search on the pages, followed by searching the page containing the record in memory.

- [1 point] Again using MRU and starting with an empty buffer pool, what is the expected hit rate in the buffer pool for the scenario of part (e)?

  0%. No page is read more than once.

4. **B+-Trees [15 points]**

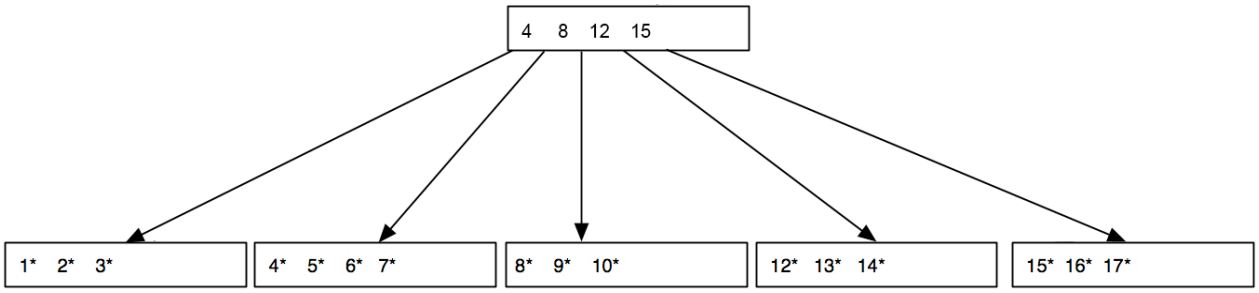
- [5 points] The B+-tree drawn below has order **3** (i.e. max 6 pointers per internal node), and contains a number of errors. Circle the errors and draw a new correct B+-tree over the same data entries (the entries in the leaves).



There are two errors above and one non-error:

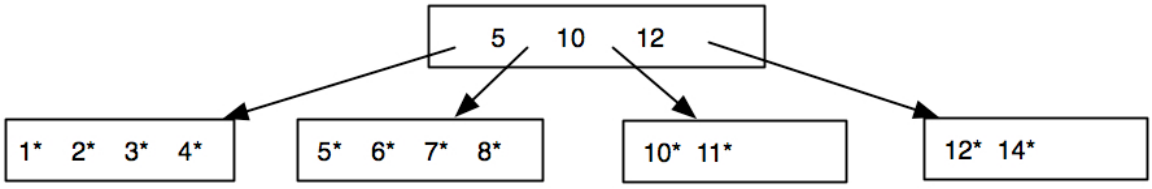
- *(1 pt) The internal node containing only "4" is underfull. It has 2 pointers, less than the minimum of 3 (which is half the maximum of 6).*
- *(1 pt) The value "7" is in the left subtree of the root, but should be in the right subtree, since the key in the root node is 6 and 7 > 6.*
- *(1 pt) The internal node containing "11 15" is not underfull, because it contains 3 pointers to child nodes, which is exactly half the maximum (6 pointers). Additionally, the presence of the key "11" which is not present in any leaf is not an error (internal nodes may contain values which were once present in leaves but have since been deleted). This point was deducted if this internal node was marked erroneous.*

Additionally, the root node is not underfull because the root node is specially permitted to be less than half full. Following is a correct new B+-tree over the same data entries:



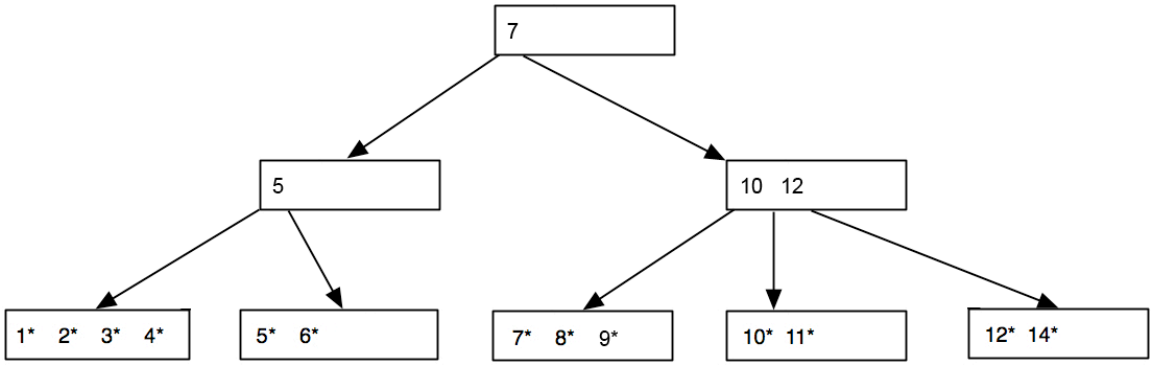
This is the unique correct B+-tree with the exact same leaf nodes. It was also valid to compact the data values into a smaller number of leaf nodes, if the values in the root are adjusted accordingly. It is not possible to use more than one internal node.

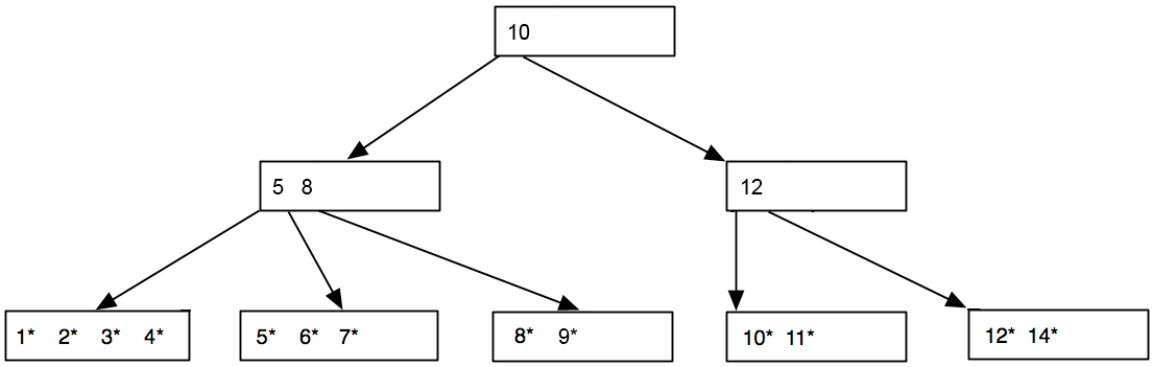
- [5 points] Consider the B+-tree with order **2 (**max **4** pointers per node**)** drawn below. Draw what the tree would look like after inserting **9***



When 9 is inserted, it goes into the node containing "5 6 7 8". However, this node is now overfull and must be split into two nodes "5 6 7" and "8 9". The value 7 is added to the parent internal node, which is now in turn overfull, so it is split into "5 7" and "10 12". Finally a new root node is created and "7" is moved to the root node.

Alternatively, we can make all our splits with the smaller side on the left. In this case, the overfull leaf is split as "5 6" and "7 8 9", 8 is added to the parent node, and "10" is moved to the new root node instead of the new value. Both trees are shown below.

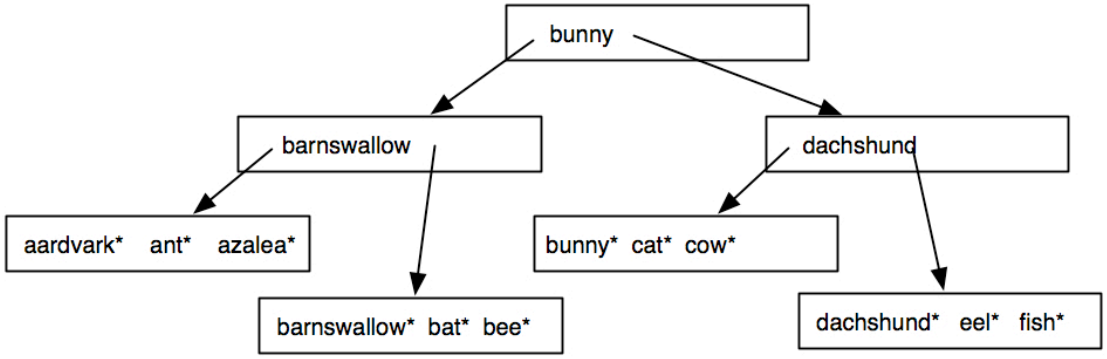


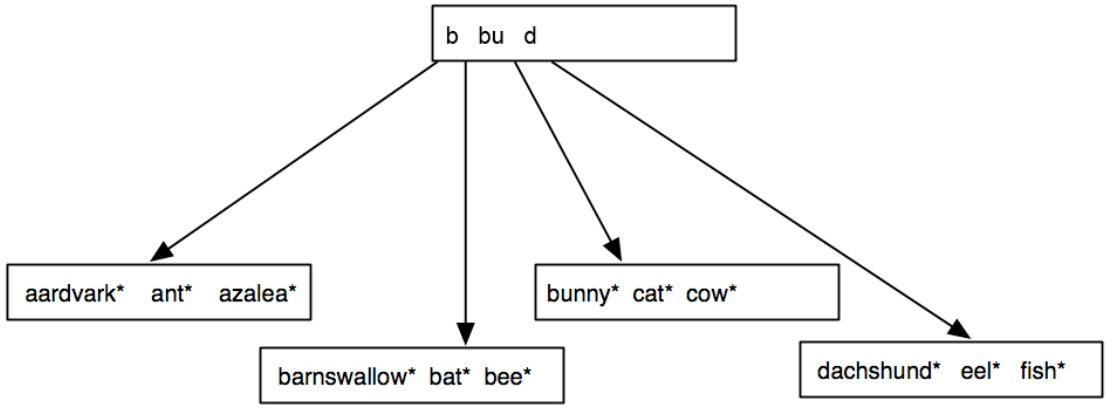


Points were given as follows:

+1	*Result is valid B+-tree. No node is underfull and no keys in internal nodes are incorrect.*
+1	*Leaf split was performed correctly.*
+1	*Internal node split and creation of new root node was performed correctly.*
+1	*Always split with either the smaller side on the left or the smaller side on the right; do not mix both strategies.*

[5 points] Assume we have a B+tree with room for 30 bytes of data on each internal page: so half-full at 15 bytes. Assume that a pageID (a "pointer" in the tree) takes up 4 bytes, and a character takes 1 byte. Consider the 1st B+-tree below. Draw a correct B+-tree for this data that uses the leaf level provided in the 2nd picture below, but employs prefix key compression above.





New root node uses a total of 4(4) + 2 + 3 + 2 = 23 < 30 bytes (assuming string prefixes are null-terminated or length-prefixed with a single byte), so it all fits in one node.

Points were given as follows:

+1	*Result is valid B+-tree. No node is underfull and no keys in internal nodes are incorrect.*
+1	*Some keys were compressed (shortened) by at least one character*
+1	*All keys in internal nodes were compressed (shortened) as short as possible*
+2	*Final tree has only two levels, root and leaves*