CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (FALL 2025)
PROF. ANDY PAVLO

Homework #3 (by Will) – Solutions

Due: **Sunday October 3rd, 2025 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Sunday October 3rd, 2025.**
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.

For your information:

- Graded out of **100** points; **5** questions total
- Rough time estimate: ≈4-6 hours (1-1.5 hours for each question)
- Each part is all or nothing. There is no partial credit.

Revision : 2025/10/06 10:58

Question	Points	Score
Linear Hashing and Cuckoo Hashing	18	
Extendible Hashing	20	
B+Tree	27	
Bloom Filter	20	
Alternate Index Structures	15	
Total:	100	

Question 1: Linear Hashing and Cuckoo Hashing . . . . [18 points]

Graded by:

For warmup, consider the following *Linear Probe Hashing* schema:

1. The table has a size of 4 slots, each slot can only contain one key-value pair.
2. The hashing function is
   $h_1(\mathtt{x}) = \mathtt{x \ \% \ 4}$.
3. When there is a conflict, it finds the next free slot to insert key-value pairs.
4. The original table is empty.
5. Uses a tombstone when deleting a key.

(a) **[2 points]** Insert key/value pairs `(3,C)` and `(8,D)`. For `(3,C)`, "3" is the key and "C" is the value. Select the value in each entry of the resulting table.

  i. Entry 0 (key % 4 = 0) ☐ C ■ **D** ☐ Empty
  ii. Entry 1 (key % 4 = 1) ☐ C ☐ D ■ **Empty**
  iii. Entry 2 (key % 4 = 2) ☐ C ☐ D ■ **Empty**
  iv. Entry 3 (key % 4 = 3) ■ **C** ☐ D ☐ Empty

Solution: `C` is inserted into Entry 3, and `D` is inserted into Entry 0.

(b) **[2 points]** After the changes from part **(a)**, delete `(8, D)`, insert key-value `(0, E)`, insert `(4, F)`, and lastly delete `(3, C)`. Select the value in each entry of the resulting table.

  i. Entry 0 (key % 4 = 0) ☐ Tombstone ☐ C ☐ D ■ **E** ☐ F ☐ Empty
  ii. Entry 1 (key % 4 = 1) ☐ Tombstone ☐ C ☐ D ☐ E ■ **F** ☐ Empty
  iii. Entry 2 (key % 4 = 2) ☐ Tombstone ☐ C ☐ D ☐ E ☐ F ■ **Empty**
  iv. Entry 3 (key % 4 = 3) ■ **Tombstone** ☐ C ☐ D ☐ E ☐ F ☐ Empty

Solution: `D` is first deleted, which inserts a tombstone into Entry 0. `E` is then inserted into Entry 0 (since there is nothing there). Then, `F` is attempted to be inserted into Entry 0, but since it's occupied by `E`, `F` is inserted into Entry 1 instead. Finally `C` is deleted which leaves a tombstone in Entry 3.

Consider the following *Cuckoo Hashing* schema:

1. Both tables have a size of 4.
2. The hashing function of the first table returns the fourth and third least significant bits: $h_1(\texttt{x}) = \texttt{(x >> 2) \& 0b11}$.
3. The hashing function of the second table returns the least significant two bits: $h_2(\texttt{x}) = \texttt{x \& 0b11}$.
4. When inserting, try table 1 first.
5. When replacement is necessary, first select an element in the *first* table.
6. The original entries in the table are shown below.

Table 1	**Entry 0**	**Entry 1**	**Entry 2**	**Entry 3**
Keys	-	-	-	445
Table 2	**Entry 0**	**Entry 1**	**Entry 2**	**Entry 3**
Keys	-	-	-	15

Figure 1: Initial contents of the hash tables.

(a) **[2 points]** Select the sequence of insert operations that results in the initial state.
■ **Insert 445, Insert 15** ☐ Insert 15, Insert 445 ☐ None of the above

Solution: 445 is inserted into Table 1 `0b11` based on h_1, and 15 experiences a collision and is hashed to Table 2 `0b11` based on h_2.

(b) Starting from the initial contents, insert key `463` and then insert `789`. Select the values in the resulting two tables.

i. Table 1

α) **[1 point]** Entry 0 (`0b00`) ☐ 445 ☐ 15 ☐ 463 ☐ 789 ■ **Empty**

β) **[1 point]** Entry 1 (`0b01`) ☐ 445 ☐ 15 ☐ 463 ■ **789** ☐ Empty

γ) **[1 point]** Entry 2 (`0b10`) ☐ 445 ☐ 15 ☐ 463 ☐ 789 ■ **Empty**

δ) **[1 point]** Entry 3 (`0b11`) ☐ 445 ☐ 15 ■ **463** ☐ 789 ☐ Empty

ii. Table 2

α) **[1 point]** Entry 0 (`0b00`) ☐ 445 ☐ 15 ☐ 463 ☐ 789 ■ **Empty**

β) **[1 point]** Entry 1 (`0b01`) ■ **445** ☐ 15 ☐ 463 ☐ 789 ☐ Empty

γ) **[1 point]** Entry 2 (`0b10`) ☐ 445 ☐ 15 ☐ 463 ☐ 789 ■ **Empty**

δ) **[1 point]** Entry 3 (`0b11`) ☐ 445 ■ **15** ☐ 463 ☐ 789 ☐ Empty

Solution: 463 tries to insert into both tables but conflicts with both, so 463 inserts into Entry 3 of Table 1, replacing 445. 445 is then rehashed into Table 2 Entry 1. 789 then inserts into Table 1 Entry 1.

(c) **[4 points]** Consider completely empty tables using the same two hash functions. Select which sequence of insertions below will cause an infinite loop.

☐ `[1, 5, 9, 13]`
☐ `[2, 7, 10, 14]`
☐ `[4, 10, 11, 15]`
■ **`[1, 9, 17, 25]`**
☐ None of the above

Solution: The sequence `[1, 9, 17, 25]` will cause an infinite loop because 4 keys map to 3 entries.

Question 2: Extendible Hashing .... [20 points]

Graded by:

Consider an extendible hashing structure such that:

- Each bucket can hold up to two records.
- The hashing function uses the lowest g bits, where g is the global depth.
- A new extendible hashing structure is initialized with $g = 0$ and one empty bucket
- If multiple keys are provided in a question, assume they are inserted one after the other from left to right.

(a) Starting from an empty table, insert keys 10, 20.

i. **[1 point]** What is the global depth of the resulting table?
■ **0** ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ None of the above

Solution: No split has occurred yet because the first bucket (on initialization) can hold 2 arbitrary values. Thus global depth is same as its initial value of 0.

ii. **[1 point]** What is the local depth of the bucket containing 10?
■ **0** ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ None of the above

Solution: There is only one bucket (created on initialization), and it holds both 10 and 20. Since no split has occurred yet, the bucket has local depth $d = 0$.

(b) Starting from the result in **(a)**, you insert keys 3, 4.

i. **[2 points]** What is the global depth of the resulting table?
☐ 0 ☐ 1 ■ **2** ☐ 3 ☐ 4 ☐ None of the above

Solution: After the inserts and splits, the table looks like the following:
Global depth = 2
`b0` = 4,20 // at local depth 2
`b2` = 10 // at local depth 2
`b1,3` = 3 // at local depth 1

ii. **[2 points]** What are the local depths of the buckets for each key?
☐ 3 (Depth 1), 4 (Depth 1), 10 (Depth 1), 20 (Depth 1)
☐ 3 (Depth 1), 4 (Depth 3), 10 (Depth 3), 20 (Depth 3)
☐ 3 (Depth 3), 4 (Depth 1), 10 (Depth 3), 20 (Depth 2)
■ **3 (Depth 1), 4 (Depth 2), 10 (Depth 2), 20 (Depth 2)**
☐ 3 (Depth 2), 4 (Depth 2), 10 (Depth 2), 20 (Depth 2)
☐ None of the above

Solution: See the previous solution for an explanation.

(c) Starting from the result in **(b)**, you insert keys 0, 12.

i. **[2 points]** What is the global depth of the resulting table?
☐ 0 ☐ 1 ☐ 2 ☐ 3 ■ **4** ☐ None of the above

Solution: 0 inserts into `b0` and causes a split. 12 inserts into `b4` and causes a split. The updated table looks as follows: Global depth = 4
`b0,8` = 0 // at local depth 3
`b2,6,10,14` = 10 // at local depth 2
`b1,3,5,7,9,11,13,15` = 3 // at local depth 1
`b4` = 4,20 // at local depth 4
`b12` = 12 // at local depth 4

ii. **[2 points]** What are the local depths of the buckets for each new key?
- ☐ 0 (Depth (3), 12 (Depth 3)
- ■ **0 (Depth (3), 12 (Depth 4)**
- ☐ 0 (Depth (4), 12 (Depth 3)
- ☐ 0 (Depth (4), 12 (Depth 4)
- ☐ 0 (Depth (2), 12 (Depth 4)
- ☐ None of the above

Solution: See the previous solution for an explanation.

(d) **[3 points]** Starting from **(c)**'s result, which **key(s)**, if inserted next, will **not** cause a split?
■ **95** ☐ 100 ■ **38** ☐ 36 ☐ None of the above

Solution: Only the selected values hash to buckets from **c** that are not full.

(e) **[3 points]** Starting from the result in **(c)**, which **key(s)**, if inserted next, will cause a split and increase the table's global depth?
☐ 1 ☐ 5 ☐ 34 ■ **68** ☐ None of the above

Solution: The values must hash to **b4**, since it is the only full bucket whose local depth is equal to the global depth.

(f) **[4 points]** Starting from an empty table, insert keys 64, 128, 256, 512. What is the global depth of the resulting table?
☐ 4 ☐ 5 ☐ 6 ☐ 7 ■ **8** ☐ $\geq$9

Solution: Since each bucket can hold at most two keys, three or more keys cannot hash to the same bucket without causing splits. When $g = 8$, 64 and 128 will each be mapped to their own bin. 256 and 512 will share the same bin.

Question 3: B+Tree .... [27 points]

Graded by:

Consider the following B+tree.

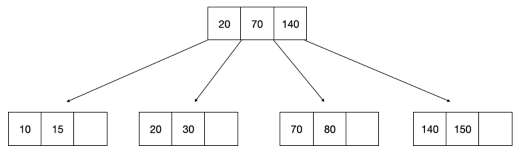


Figure 2: B+ Tree of order $d = 4$ and height $h = 2$.

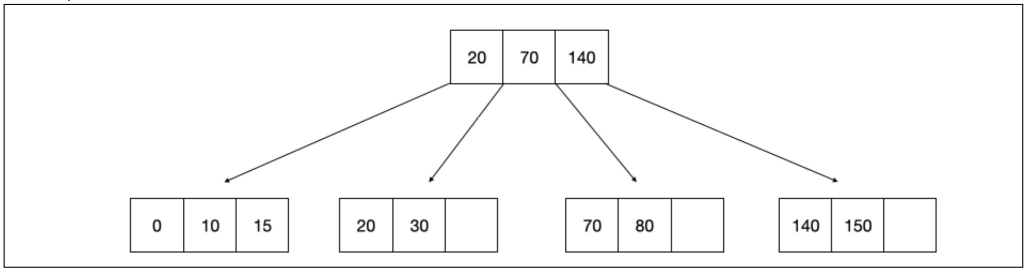
When answering the following questions, be sure to follow the procedures described in class and in your textbook. You can make the following assumptions:

- A left pointer in an internal node guides towards keys $<$ than its corresponding key, while a right pointer guides towards keys $\geq$.
- A leaf node underflows when the number of **keys** goes below $\lceil \frac{d-1}{2} \rceil$.
- An internal node underflows when the number of **pointers** goes below $\lceil \frac{d}{2} \rceil$.

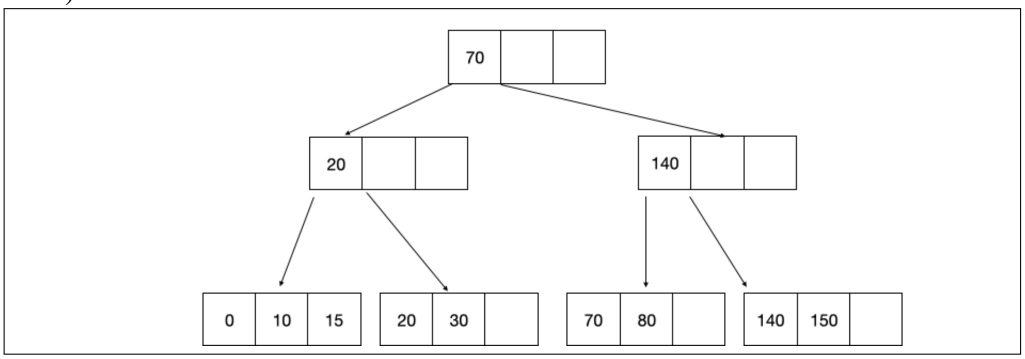
Note that B+ tree diagrams for this problem omit leaf pointers for convenience. The leaves of actual B+ trees are linked together via pointers, forming a singly linked list allowing for quick traversal through all keys.

(a) **[4 points]** Insert 0^* into the B+tree. Select the resulting tree.

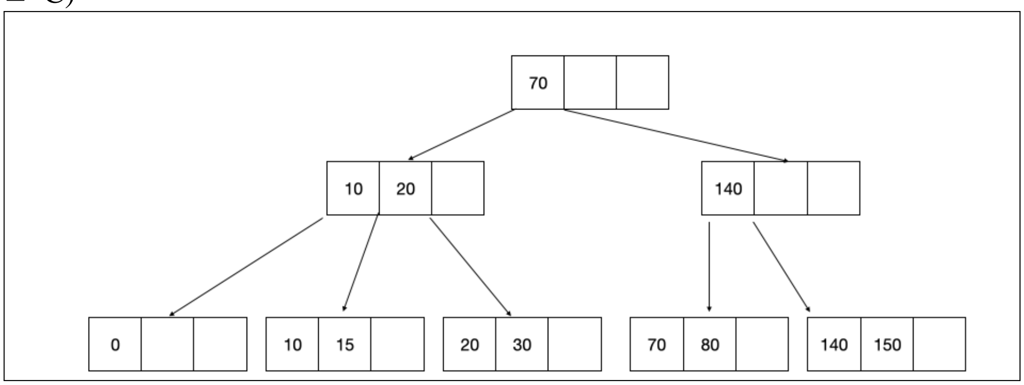
■ **A)**



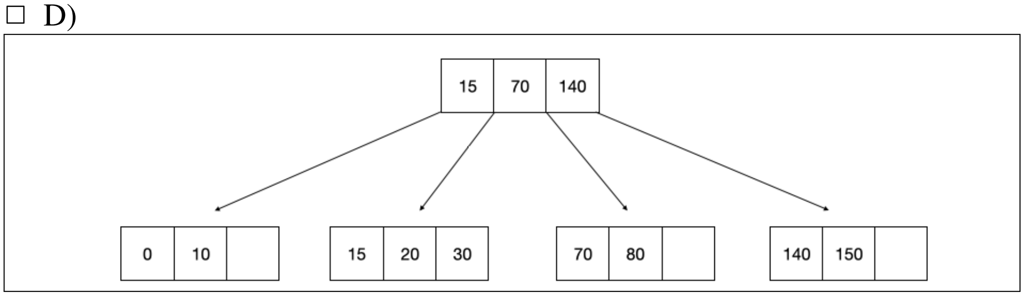
☐ B)



☐ C)



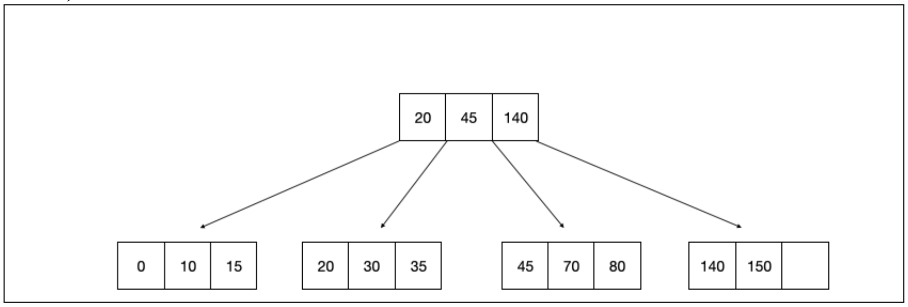
☐ D)



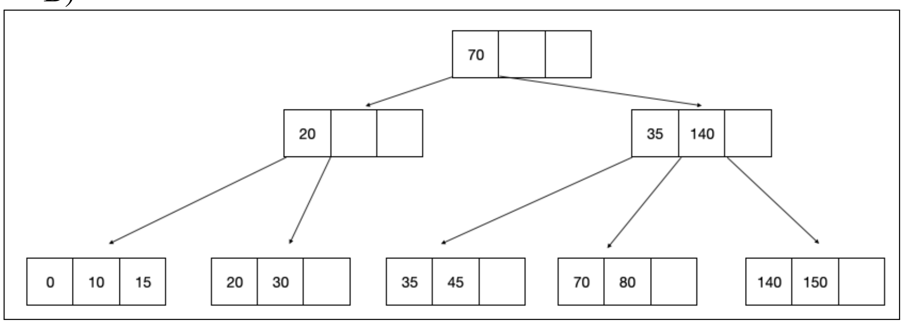
Solution: Inserting 0^* adds one element in the left-most leaf. It should not cause any splits or merges.

(b) **[5 points]** Starting with the tree that results from **(a)**, insert 35^* and then 45^*. Select the resulting tree.

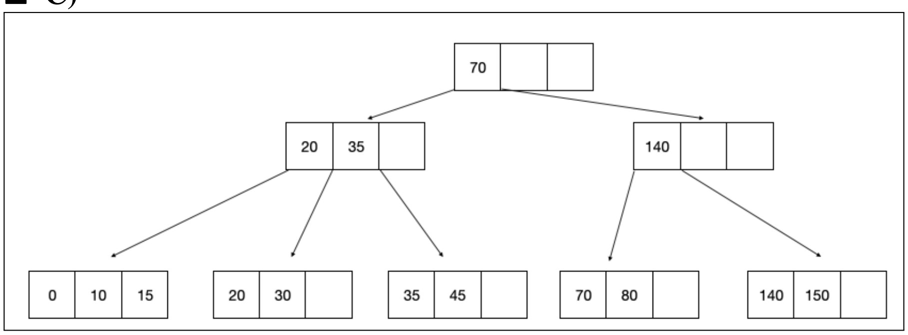
☐ A)



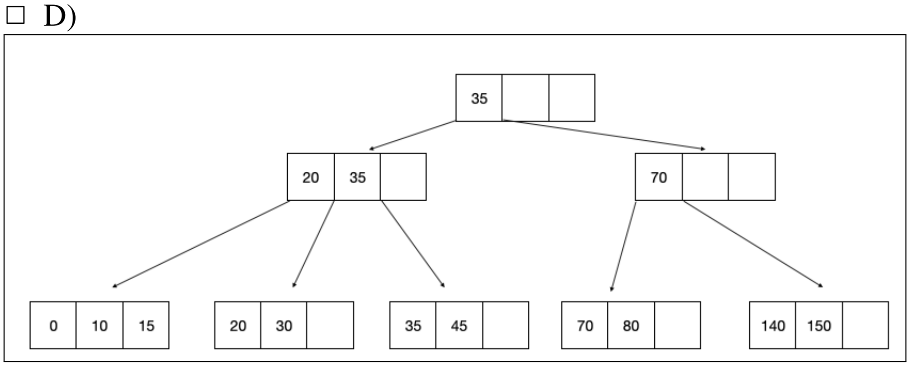
☐ B)



■ **C)**



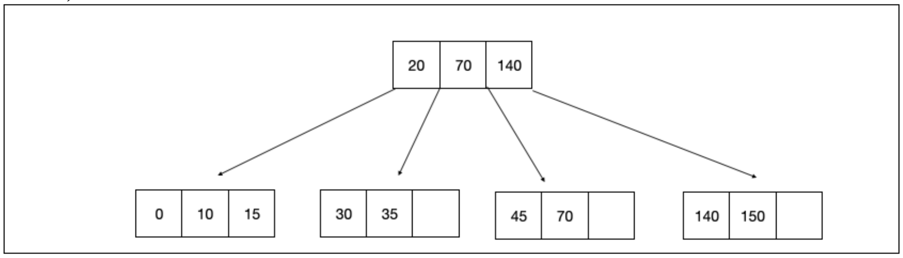
☐ D)



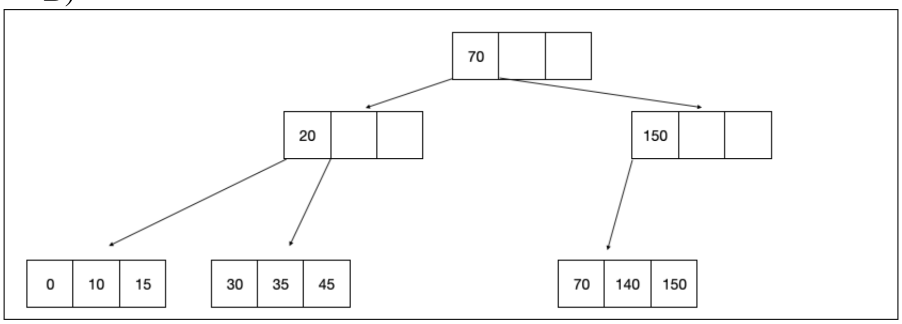
Solution: Inserting 35^* fills in the remaining space of the second leaf node (from the left). After inserting 45^*, the second leaf node splits. As the root-level node is full, the root-level also splits.

(c) **[8 points]** Starting with the tree that results from **(b)**, deletes 80^* and then 20^*. Select the resulting tree.

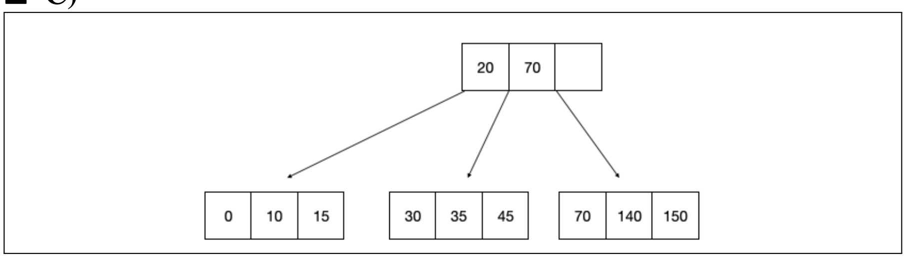
☐ A)



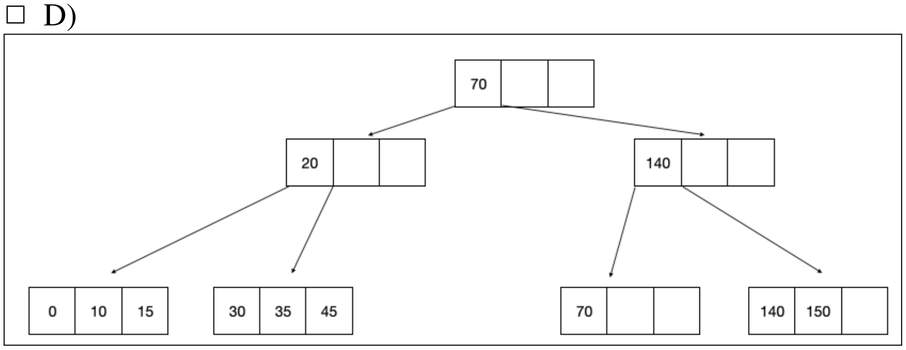
☐ B)



■ **C)**



☐ D)



Solution: Deleting 80^* causes the third leaf node to underflow and causes the right two leaf nodes to merge. After merging, the right internal node underflows, which triggers recursive merging.

Then deleting 20^* causes the second leaf node to underflow. This leads the second and third leaf node to then merge into (30, 35, 45).

(d) i. **[2 points]** Threads must release their latches in the order they were acquired (i.e., FIFO).

☐ True ■ **False**

Solution: Threads can release latches in any order.

ii. **[2 points]** Under optimistic latch coupling, write threads only take the write latch on the root when they restart.

■ **True** ☐ False

Solution: Using the optimistic latch coupling/crabbing scheme that we discussed in class, the thread will take a write latch on the root if it needs to restart.

iii. **[2 points]** A DBMS primarily executes OLTP queries but periodically will execute OLAP style queries (e.g., analytics, book-keeping). The DBMS will benefit from using one buffer pool for inner node pages and a different buffer pool for leaf pages / table pages.

■ **True** ☐ False

Solution: Because the DBMS is aware of whether a page is for a leaf or an inner node, and because B+Tree transformations never change a leaf into an inner node or vice- versa, it's straightforward for a DMBS to use a different buffer pool for inner node pages than for leaf node pages. Such a configuration would help prevent index leaf scans from sequentially flooding the buffer pool and harming the performance of OLTP-style queries on the same index.

iv. **[2 points]** Under optimistic latch crabbing, read-only thread can drop its latch on the current page before acquiring the latch on the next page (e.g., child, sibling).

☐ True ■ **False**

Solution: During traversal, a reader temporarily needs to hold a latch on both the parent and child (or two siblings in a leaf node scan) before releasing the latch on the parent page.

v. **[2 points]** "No-Wait" mode for acquiring sibling latches prevents deadlock by allowing the acquirer to fail if another reader already owns the lock.

■ **True** ☐ False

Solution: A "no-wait" mode prevents threads from getting stuck.

Question 4: Bloom Filter ....[20 points]

Graded by:

Assume that we have a bloom filter that is used to register words relating to Pokemon. The filter uses two hash functions h_1 and h_2 which hash the following words to the following values:

input	h_1	h_2
"Pikachu"	4721	8395
"Bulbasaur"	1568	4207
"Charmander"	9034	2876
"Squirtle"	6412	7559

(a) **[6 points]** Suppose the filter has 8 bits initially set to 0:

bit 0	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7
0	0	0	0	0	0	0	0

Which bits will be set to 1 after "Pikachu" and "Bulbasaur" have been inserted?

■ **0** ■ **1** □ 2 ■ **3** □ 4 □ 5 □ 6 ■ **7**

Solution: Because the filter has 8 bits, we take the modulo of the hashed output and 8.

(b) Suppose the filter has 8 bits set to the following values:

bit 0	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7
0	0	1	1	1	1	0	0

i. **[4 points]** What will we learn using the above filter if we lookup "Charmander"?

□ Charmander has been inserted
□ Charmander has not been inserted
■ **Charmander may have been inserted**
□ Not possible to know

Solution: $9034 \mod 8 = 2$; $2876 \mod 8 = 4$
Because both bits are 1, the filter will return true, meaning Charmander may have been inserted.

ii. **[4 points]** What will we learn if we lookup "Squirtle"?

□ Squirtle has been inserted
■ **Squirtle has not been inserted**
□ Squirtle may have been inserted
□ Not possible to know

Solution: $6412 \mod 8 = 4$; $7559 \mod 8 = 7$
Because bit 7 is 0, the filter will return false, so Squirtle has not been inserted.

(c) **[6 points]** A colleague is interviewing a candidate and would like to first test your knowledge of bloom filters. The colleague has a list of prepared statements and would like you to identify which of them are true. Select all true statements.

- ☐ Bloom filters are more effective than hash indexes for exact-match (or lookup) queries.
- ☐ Using more hash functions will *always* lower a bloom filter's false positive rate.
- ■ **Bloom filters *can* reduce disk I/Os.**
- ■ **Add and lookup operations on bloom filters are parallelizable.**
- ☐ All of the above.

Solution: Using more hash functions can increase the likelihood of false positives due to overlapping bits.

For exact-match queries, query execution is better off using a hash index.

Question 5: Alternate Index Structures . . . . [15 points]

Graded by:

(a) **[5 points]** Your team is considering using a **Radix Tree** for indexing in a new database system. They consulted a large language model for some factual statements about Radix Trees but are unsure about the accuracy of the model's responses. They have asked you to identify all factually correct statements.

■ **Radix Trees are efficient for prefix queries.**
□ Radix Trees support efficient substring searches (e.g., LIKE "%?%").
□ Radix Trees require re-balancing after every insertion or deletion.
■ **The levels of a radix tree have no node requirements.**
■ **For datasets with lots of common prefixes, radix trees can use less space than B+Trees.**
□ None of the above.

Solution: Radix Trees are indeed efficient for prefix queries.

Radix Trees do not support efficient substring searches; they are optimized for prefix matching.

Radix Trees do not require re-balancing like balanced trees (e.g., AVL or Red-Black Trees).

There is no requirement about the size of each level in a Radix Tree.

They can be more space-efficient than B+Trees for certain datasets, especially when keys share common prefixes.

(b) **[5 points]** You are discussing index structures with a colleague. They want to compare **B+Trees**, **Skip Lists**, **Radix Trees**, and **Inverted Indexes**. Select all the true statements.

■ **It is *on average* cheaper to update skip lists than B+Trees.**
□ B+Trees and Skip Lists both guarantee logarithmic complexity for lookups.
□ B+Trees and Inverted Indexes are both efficient at handling substring (e.g., LIKE "%?%") queries.
□ Skip Lists perform better than B+Trees for range queries.
□ None of the above.

Solution: B+Trees are generally more expensive to update than Skip Lists due to the need for node splits and merges.

Skip Lists have **approximate** logarithmic complexity for lookups, but it is not guaranteed.

B+Trees are not efficient for substring queries.

B+Trees generally perform better than Skip Lists for range queries due to their sequential access patterns.

(c) **[5 points]** Suppose you are trying to run the following query:

```
SELECT * FROM PRODUCTS WHERE description LIKE '%laptop%';
```

Assume that there is a non-clustering B+Tree index on `description`. Your query is running slowly. Which of the following choices (if any) would make this query go faster?

- ☐ Drop the index and build a **bloom filter** on `description`.
- ☐ Replace the index with a **hash index** on `description`.
- ☐ Replace the non-clustering B+Tree with a **clustering** B+Tree index on `description`.
- ☐ Replace the index with a **radix tree** on `description`.
- ■ **None of the above.**

Solution: All of these options would not substantially speed up the query. If such queries dominate the workload, the best approach would be to invest in an **inverted index**, which is specifically designed for text-based searches like the one in the query.