CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (FALL 2026)
PROF. JIGNESH PATEL

Homework #3 (by Drew Ripberger) – Solutions

Due: **Sunday Oct 4th, 2026 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Sunday Oct 4th, 2026.**
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.
- No AI may be used to complete homework.

For your information:

- Graded out of **100** points; **5** questions total
- Rough time estimate: ≈4-6 hours (1-1.5 hours for each question)

Revision : 2026/10/05 09:58

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 $h_1(\texttt{x}) = \texttt{x \% 4}$.
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 `(2,C)` and `(9,D)`. For `(2,C)`, "2" 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 2, and `D` is inserted into Entry 1.

(b) **[2 points]** After the changes from part **(a)**, delete `(9, D)`, insert key-value `(0, E)`, insert `(4, F)`, and lastly delete `(2, 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 1. `E` is then inserted into Entry 0 (since there is nothing there). Then, `F` is attempted to be inserted into Entry 0, but it's occupied by `E`. We then try to insert `F` into Entry 1, which is successful since it is a tombstone. Finally `C` is deleted which leaves a tombstone in Entry 2.

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	-	-	10	-
Table 2	**Entry 0**	**Entry 1**	**Entry 2**	**Entry 3**
Keys	88	-	-	-

Figure 1: Initial contents of the hash tables.

(a) **[2 points]** Select the sequence of insert operations that results in the initial state.
■ **Insert 10, Insert 88** □ Insert 88, Insert 10 □ None of the above

Solution: 10 is inserted into Table 1 `0b11` based on h_1, and 88 experiences a collision and is hashed to Table 2 `0b10` based on h_2.

(b) Starting from the initial contents, insert key `200` and then insert `976`. Select the values in the resulting two tables.

i. Table 1

α) **[1 point]**	Entry 0 (`0b00`)	☐ 10	☐ 88	☐ 200	■ **976**	☐ Empty
β) **[1 point]**	Entry 1 (`0b01`)	☐ 10	☐ 88	☐ 200	☐ 976	■ **Empty**
γ) **[1 point]**	Entry 2 (`0b10`)	☐ 10	☐ 88	■ **200**	☐ 976	☐ Empty
δ) **[1 point]**	Entry 3 (`0b11`)	☐ 10	☐ 88	☐ 200	☐ 976	■ **Empty**

ii. Table 2

α) **[1 point]**	Entry 0 (`0b00`)	☐ 10	■ **88**	☐ 200	☐ 976	☐ Empty
β) **[1 point]**	Entry 1 (`0b01`)	☐ 10	☐ 88	☐ 200	☐ 976	■ **Empty**
γ) **[1 point]**	Entry 2 (`0b10`)	■ **10**	☐ 88	☐ 200	☐ 976	☐ Empty
δ) **[1 point]**	Entry 3 (`0b11`)	☐ 10	☐ 88	☐ 200	☐ 976	■ **Empty**

Solution: 200 tries to insert into both tables but conflicts with both, so 200 inserts into Entry 2 of Table 1, replacing 10. 10 is then rehashed into Table 2 Entry 2. 976 then inserts into Table 1 Entry 0.

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

☐ `[2, 1, 112, 991]`
☐ `[4, 10, 11, 654]`
☐ `[876, 765, 654, 543]`
■ **`[13, 93, 77, 127]`**
☐ None of the above

Solution: The sequence `[13, 93, 77, 127]` will cause an infinite loop because there is only one entry that the keys can occupy in table 1, and only 2 entries they can occupy in table 2. Therefore, there are only 3 total entries to store the 4 elements.

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 2, 9.

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 9?

■ **0** ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ None of the above

Solution: There is only one bucket (created on initialization), and it holds both 2 and 9. Since no split has occurred yet, the bucket has local depth $d = 0$.

(b) Starting from the result in **(a)**, you insert keys 4, 6.

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
`Bucket 00` = {4} // at local depth 2
`Bucket 10` = {2, 6} // at local depth 2
`Bucket 01 / 11` = {9} // at local depth 1

ii. **[2 points]** What are the local depths of the buckets for each key?

☐ 2 (Depth 2), 9 (Depth 2), 4 (Depth 1), 6 (Depth 2)
☐ 2 (Depth 1), 9 (Depth 3), 4 (Depth 3), 6 (Depth 3)
☐ 2 (Depth 3), 9 (Depth 1), 4 (Depth 3), 6 (Depth 2)
■ **2 (Depth 2), 9 (Depth 1), 4 (Depth 2), 6 (Depth 2)**
☐ 2 (Depth 2), 9 (Depth 2), 4 (Depth 2), 6 (Depth 2)
☐ None of the above

Solution: See the previous solution for an explanation.

(c) Starting from the result in **(b)**, you insert keys 7, 14.

i. **[2 points]** What is the global depth of the resulting table?

☐ 0 ☐ 1 ☐ 2 ■ **3** ☐ 4 ☐ None of the above

Solution: 7 inserts into `Bucket 01 / 11` and causes a split. 14 inserts into `Bucket 10` and causes the table to double, then a subsequent split. The updated table looks as follows: Global depth = 3
Bucket 000 / 100 = {4} // at local depth 2 Bucket 010 = {2} // at local depth 3 Bucket 110 = {6, 14} // at local depth 3 Bucket 001 / 011 / 101 / 111 = {9, 7} // at local depth 1

ii. **[2 points]** What are the local depths of the buckets for each new key?
- ☐ 7 (Depth 3), 14 (Depth 3)
- ☐ 7 (Depth 2), 14 (Depth 2)
- ☐ 7 (Depth 1), 14 (Depth 2)
- ■ **7 (Depth 1), 14 (Depth 3)**
- ☐ 7 (Depth 2), 14 (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? Treat each key individually.

■ **28** ■ **8** ☐ 51 ■ **58** ☐ 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 increase the table's global depth? Treat each key individually.

☐ 13 ☐ 41 ☐ 15 ■ **46** ☐ None of the above

Solution: The values must hash to **Bucket 110**, 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 32, 64, 128, 256. What is the global depth of the resulting table?

☐ 4 ☐ 5 ☐ 6 ■ **7** ☐ 8 ☐ ≥ 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 $g = 7$, 32 and 64 will each be mapped to their own bin. 128 and 256 will share the same bin.

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

Graded by:

Consider the following B+tree.

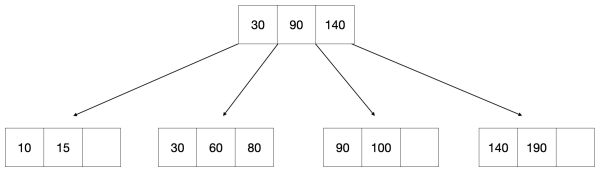


Figure 2: B+ Tree of order $d = 4$ and height $h = 2$.

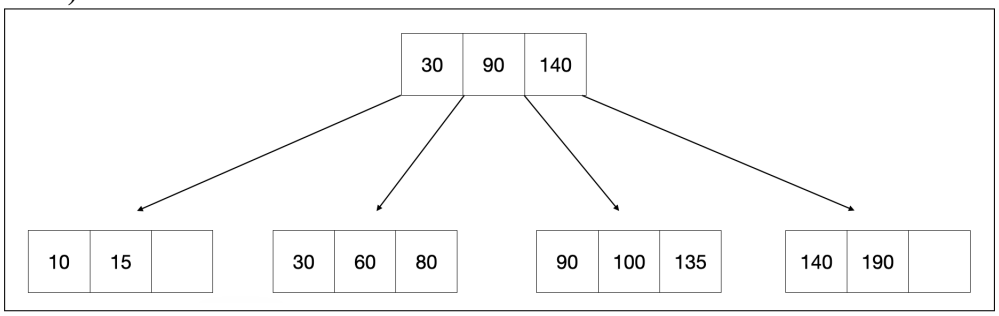
When answering the following questions, be sure to follow the procedures described in class and in your textbook. You can make the following assumptions:

- 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 $\lceil \frac{d}{2} \rceil$.

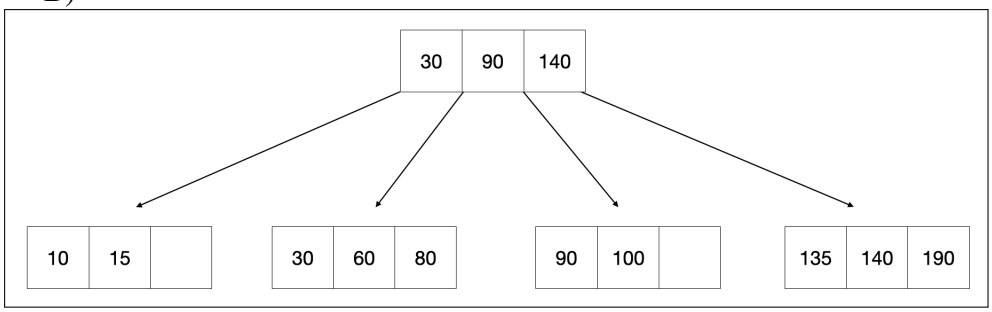
Note that B+ tree diagrams for this problem omit leaf pointers for convenience. The leaves of actual B+ trees are linked together via pointers, forming a singly linked list allowing for quick traversal through all keys.

(a) **[4 points]** Insert 135^* into the B+tree. Select the resulting tree.

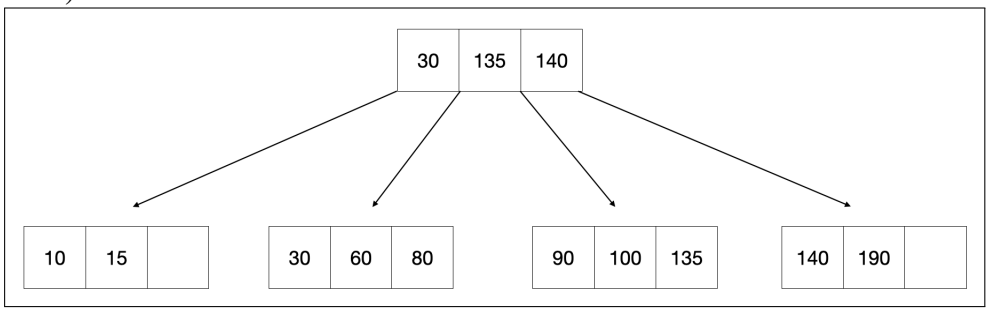
■ **A)**



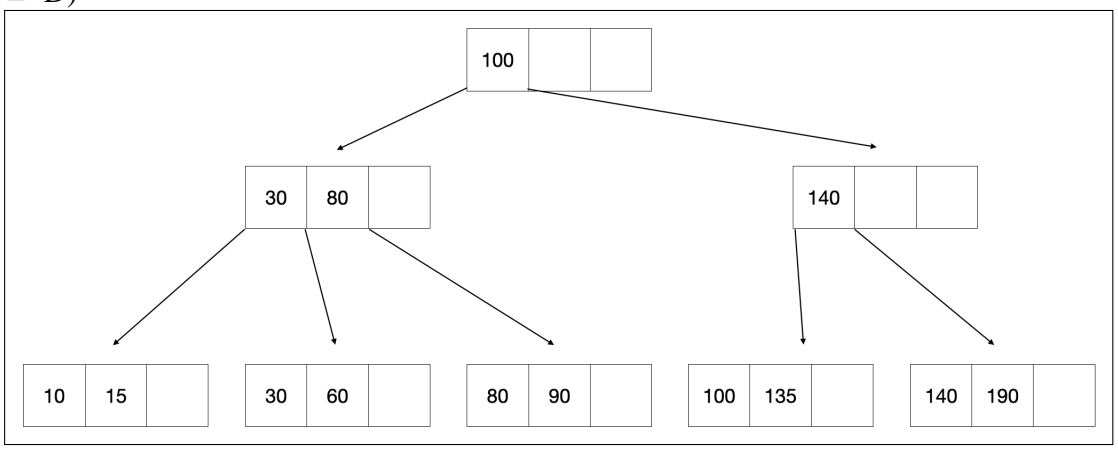
□ B)



□ C)



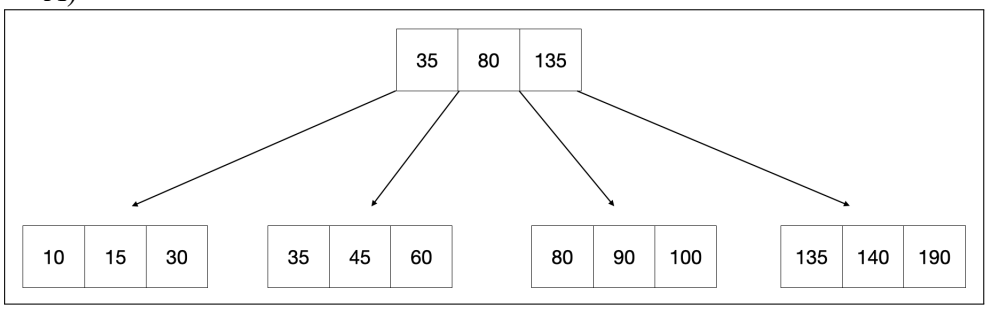
□ D)



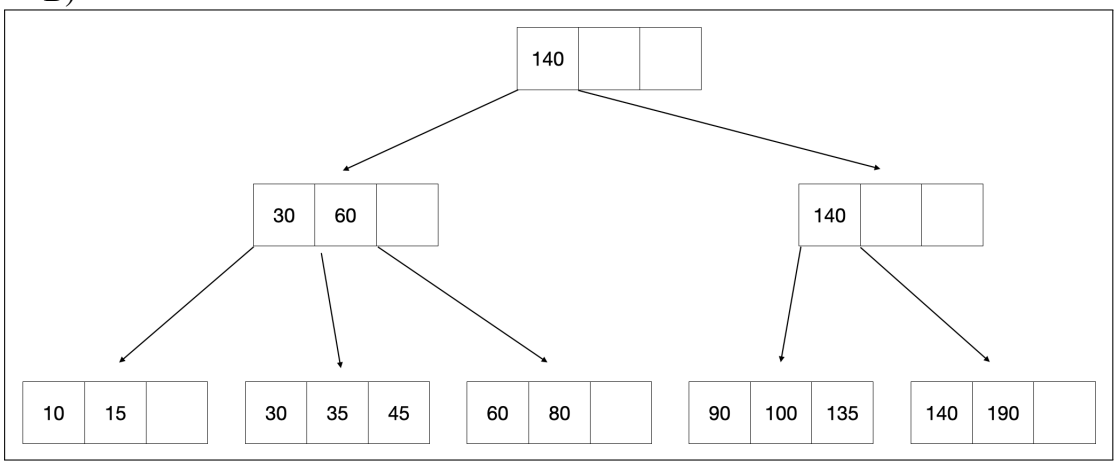
Solution: Inserting 135^* adds one element in the third-from-left leaf. It should not cause any splits or merges.

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

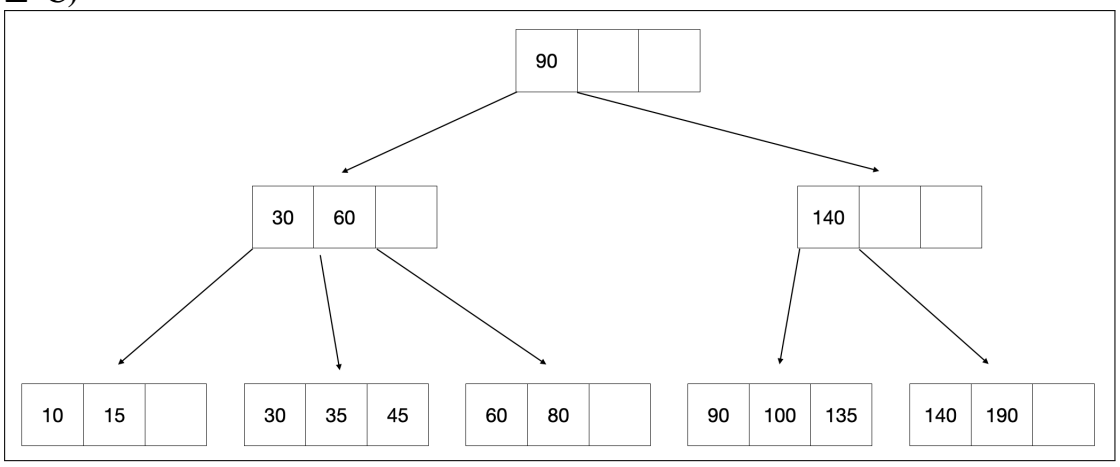
☐ A)



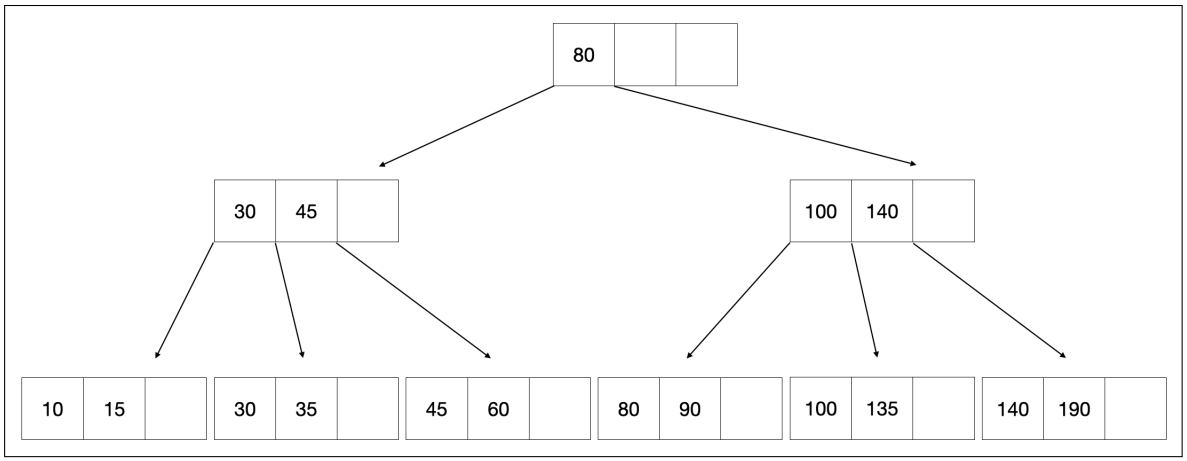
☐ B)



■ **C)**



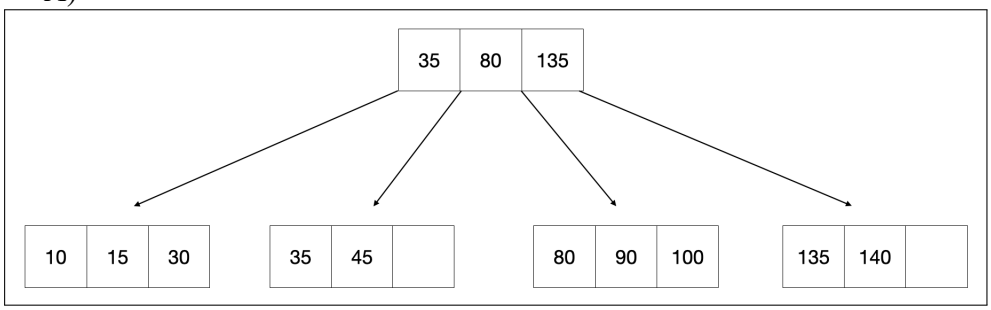
☐ D)



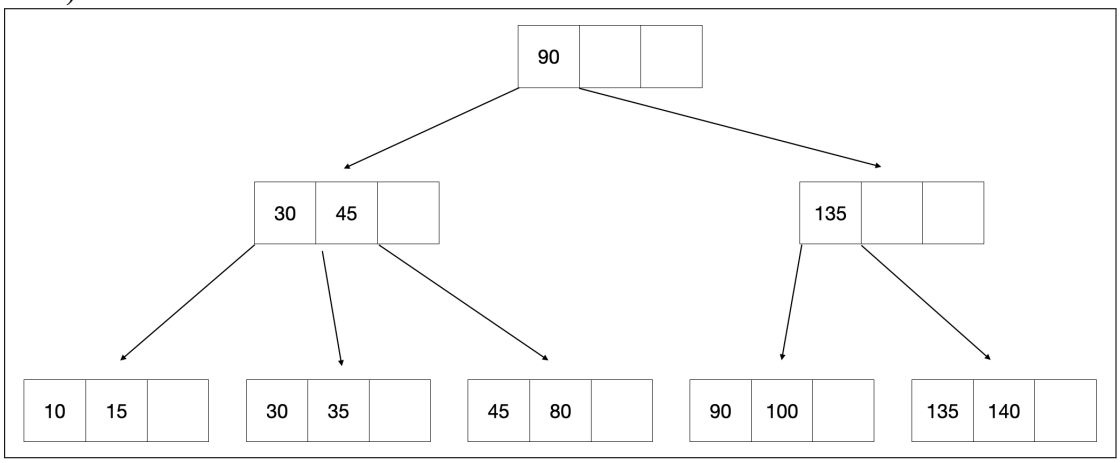
Solution: Inserting 35^* causes the second-from-left leaf to split. As the root-level node is full, the root-node also splits. The 45^* is inserted without any splits or merges.

(c) **[8 points]** Starting with the tree that results from **(b)**, delete 190^* and then 60^*. Select the resulting tree.

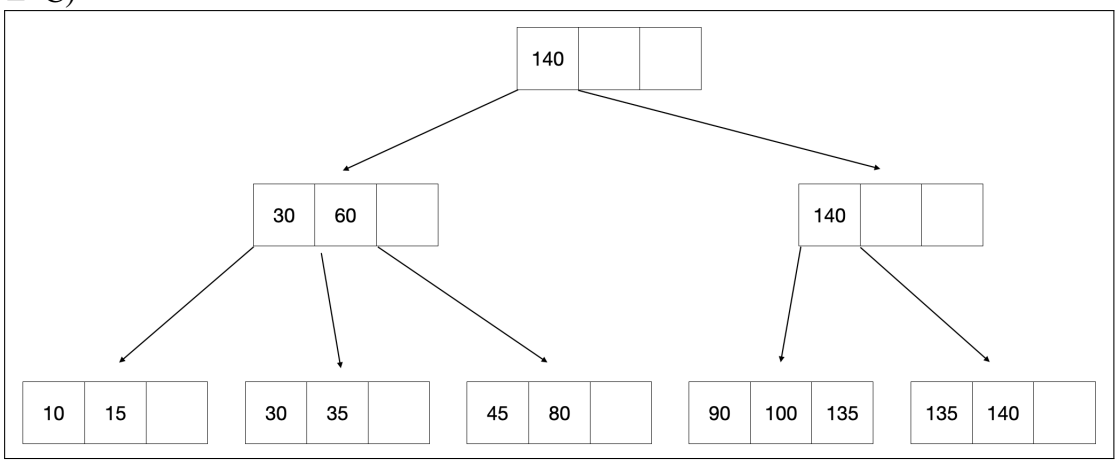
☐ A)



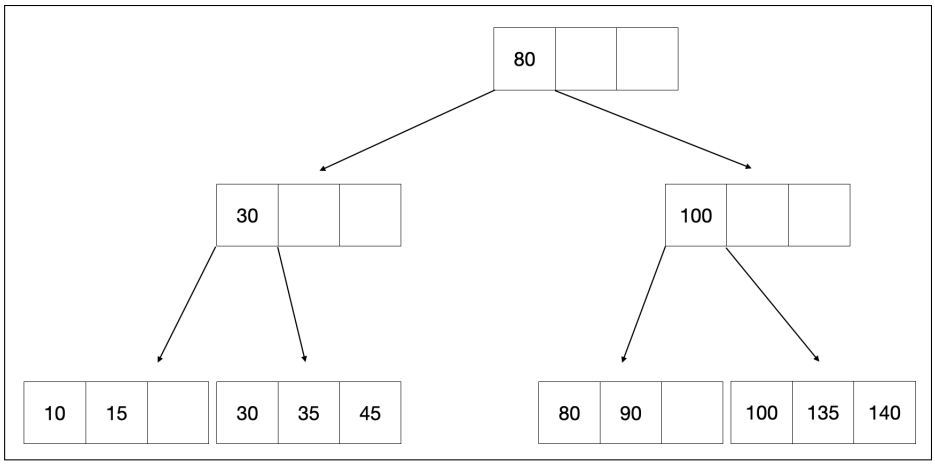
■ **B)**



☐ C)



☐ D)



Solution: For both deletions, the node underflows and needs to borrow from its neighbor.

(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, a write thread will acquire a write latch on the root node only if it is forced to 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 DBMS 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 a writer 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 Middle Earth. 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
"Elves"	9399	1231
"Hobbits"	29	3218
"Wizards"	1032	3899
"Orcs"	3912	4543

(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 "Hobbits" and "Elves" 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
1	0	0	1	0	0	0	1

i. **[4 points]** What will we learn using the above filter if we lookup "Orcs"?
☐ Orcs has been inserted
☐ Orcs has not been inserted
■ **Orcs may have been inserted**
☐ Not possible to know

Solution: $3912 \mod 8 = 0$; $4543 \mod 8 = 7$
Because both bits are 1, the filter will return true, meaning Orcs may have been inserted. However, inserting Elves and Wizards also yields this layout, without inserting Orcs.

ii. **[4 points]** What will we learn if we lookup "Hobbits"?
☐ Hobbits has been inserted
■ **Hobbits has not been inserted**
☐ Hobbits may have been inserted
☐ Not possible to know

Solution: $29 \mod 8 = 5; 3218 \mod 8 = 2$
Because bit 5 is 0, the filter will return false, so Hobbits 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.

■ **Add and lookup operations on bloom filters are parallelizable.**
□ Increasing the size of the bit array has no effect on the false positive rate.
■ **A Bloom filter may return a false positive, but never a false negative.**
□ Using more hash functions will *always* lower a bloom filter's false positive rate.
□ All of the above.

Solution: It is possible to parallelize these operations.

Increasing the size of the bit array can reduce false positives as elements hash to more distinct values.

False negatives are impossible with bloom filters since the bits must be set if an element is added to the filter.

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

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 support efficient substring searches (e.g., LIKE "%?%").
- ☐ Search time in a radix tree does not depend on the length of the key being searched.
- ■ **Two keys with a long common prefix will share a path in a radix tree.**
- ■ **For datasets with lots of common prefixes, radix trees can use less space than B+Trees.**
- ■ **Radix Trees are efficient for prefix queries.**
- ☐ None of the above.

Solution:

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

Search time does depend on the length of the key being searched, since we may need to traverse more nodes for a longer key.

Keys with long common prefixes will share a path until their first character of divergence.

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

Radix Trees are indeed efficient for prefix queries.

(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.

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

Solution: B+Trees are not efficient for substring queries.

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

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

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

(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.