CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (SPRING 2025)
PROF. JIGNESH PATEL

Homework #3 (by Ramsey) – Solutions
Due: **Sunday February 23rd, 2025 @ 11:59pm**

IMPORTANT:
- Enter all of your answers into **Gradescope by 11:59pm on Sunday February 23rd, 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)

Revision : 2025/02/12 15:16

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
   $h_1(\mathsf{x}) = \mathsf{x}\ \%\ 4$.

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 (6,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 3.

(b) **[2 points]** After the changes from part **(a)**, delete (2, C), insert key-value (10, E), and lastly insert (14, F). 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:** C is first deleted, which inserts a tombstone into Entry 2. E is then inserted into Entry 2 (since there is nothing there). Then, F is attempted to be inserted into Entry 2, but since it's occupied by E, F is inserted into Entry 0 instead.

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:
   $h_1(x) = (x \text{ >> } 2) \text{ \& } 0b11$.

3. The hashing function of the second table returns the least significant two bits:
   $h_2(x) = x \text{ \& } 0b11$.

4. When inserting, try table 1 first.

5. When replacement is necessary, first select an element in the second table.

6. The original entries in the table are shown below.

Table 1	Entry 0	Entry 1	Entry 2	Entry 3
Keys	-	20	-	-

Table 2	Entry 0	Entry 1	Entry 2	Entry 3
Keys	-	-	-	7

Figure 1: Initial contents of the hash tables.

(a) **[2 points]** Select the sequence of insert operations that results in the initial state.
   ■ **Insert 20, Insert 7**   □ Insert 7, Insert 20   □ None of the above

> **Solution:** 20 is inserted into Table 1 `0b01` based on h_1, and 7 experiences a collision and is hashed to Table 2 `0b11` based on h_2.

(b) Starting from the initial contents, insert key 22 and then insert 38. Select the values in the resulting two tables.

    i. Table 1

		Entry	□ 20	□ 7	□ 22	□ 38	Empty
α)	**[1 point]**	Entry 0 (`0b00`)	□ 20	□ 7	□ 22	□ 38	■ **Empty**
β)	**[1 point]**	Entry 1 (`0b01`)	□ 20	□ 7	■ **22**	□ 38	□ Empty
γ)	**[1 point]**	Entry 2 (`0b10`)	□ 20	□ 7	□ 22	□ 38	■ **Empty**
δ)	**[1 point]**	Entry 3 (`0b11`)	□ 20	□ 7	□ 22	□ 38	■ **Empty**

    ii. Table 2

		Entry	20	7	22	38	Empty
α)	**[1 point]**	Entry 0 (`0b00`)	■ **20**	□ 7	□ 22	□ 38	□ Empty
β)	**[1 point]**	Entry 1 (`0b01`)	□ 20	□ 7	□ 22	□ 38	■ **Empty**
γ)	**[1 point]**	Entry 2 (`0b10`)	□ 20	□ 7	□ 22	■ **38**	□ Empty
δ)	**[1 point]**	Entry 3 (`0b11`)	□ 20	■ **7**	□ 22	□ 38	□ Empty

> **Solution:** 22 tries to insert into Table 1 first but due to conflict, it inserts into Entry 2 of Table 2. 38 tries to insert into both tables but conflicts with both, so 38 inserts into Entry 2 of Table 2, replacing 22. 22 is then rehashed into Table 1 Entry 1, replacing 20. 20 is then rehashed into Table 2 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.

□ `[1, 5, 9, 13]`
■ **`[0, 8, 16, 24]`**
□ `[2, 6, 10, 14]`
□ `[3, 7, 11, 15]`
□ None of the above

> **Solution:** The sequence `[0, 8, 16, 24]` will cause an infinite loop because 4 keys map to a 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 1, 2.

   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 2?
   ■ **0**   □ 1   □ 2   □ 3   □ 4   □ None of the above

> **Solution:** There is only one bucket (created on initialization), and it holds both 1 and 2. Since no split has occurred yet, the bucket has local depth $d = 0$.

(b) Starting from the result in **(a)**, you insert keys 9, 11.

   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,b2 = 2 // at local depth 1
> b1 = 1,9 // at local depth 2
> b3 = 11 // at local depth 2

   ii. **[2 points]** What are the local depths of the buckets for each key?
   □ 1 (Depth 1), 2 (Depth 1), 9 (Depth 1), 11 (Depth 1)
   □ 1 (Depth 3), 2 (Depth 1), 9 (Depth 3), 11 (Depth 3)
   ■ **1 (Depth 2), 2 (Depth 1), 9 (Depth 2), 11 (Depth 2)**
   □ 1 (Depth 3), 2 (Depth 1), 9 (Depth 3), 11 (Depth 2)
   □ 1 (Depth 2), 2 (Depth 2), 9 (Depth 2), 11 (Depth 2)
   □ None of the above

> **Solution:** See the previous solution for an explanation.

(c) Starting from the result in **(b)**, you insert keys 13, 27.

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

> **Solution:** 13 inserts into b1 and causes a split. 27 inserts into b3 without a split. The updated table looks as follows: Global depth = 3
> b0,b2,b4,b6 = 2 // at local depth 1
> b1 = 1,9 // at local depth 3
> b5 = 13 // at local depth 3
> b3,b7 = 11,27 // at local depth 2

  ii. **[2 points]** What are the local depths of the buckets for each new key?
    □ 13 (Depth 1), 27 (Depth 1)
    □ 13 (Depth 1), 27 (Depth 2)
    □ 13 (Depth 2), 27 (Depth 2)
    ■ **13 (Depth 3), 27 (Depth 2)**
    □ 13 (Depth 3), 27 (Depth 3)
    □ 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?
  ■ **5**   □ 17   □ 43   ■ **8**   □ None of the above

> **Solution:** To avoid a split in the current table the value must map to one of the following: b0,b2,b4,b6,b5. Out of the options provided only 5 and 8 hash to one of those.

(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?
  □ 0   ■ **3**   □ 5   ■ **17**   □ None of the above

> **Solution:** There are two options. The first is to insert a key that hashes to b1, since it is the only full bucket whose local depth is equal to the global depth. **17** maps to this bucket.
>
> The other option is **3**. 3 maps to bucket (b3,b7). Inserting **3** causes a local depth increase to (depth = 3). However, 3, 11, and 27 all re-hash to the same bucket. This results in another split that increases the global depth.

(f) **[4 points]** Starting from an empty table, insert keys 32, 64, 128, 512. 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 512 will share the same bin.

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

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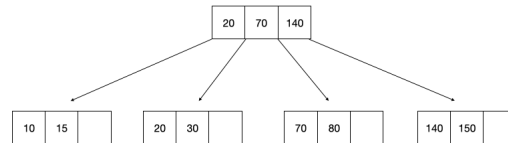
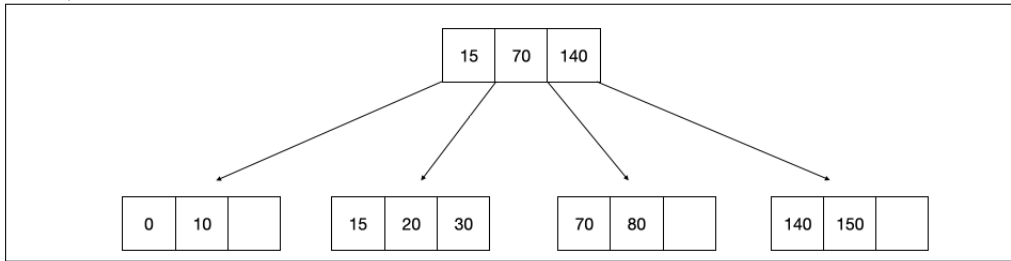Figure 2: B+ Tree of order $d = 4$ and height $h = 2$.

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.

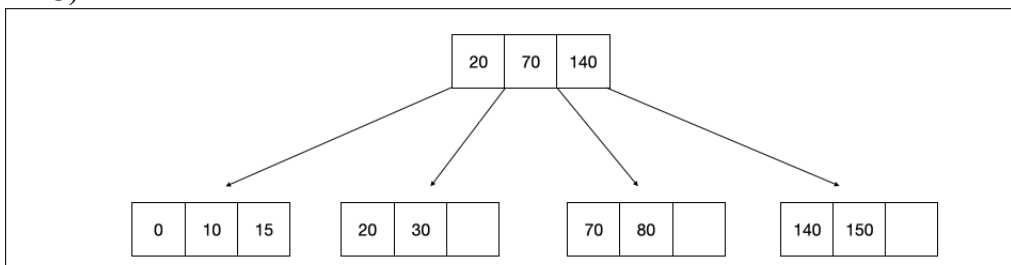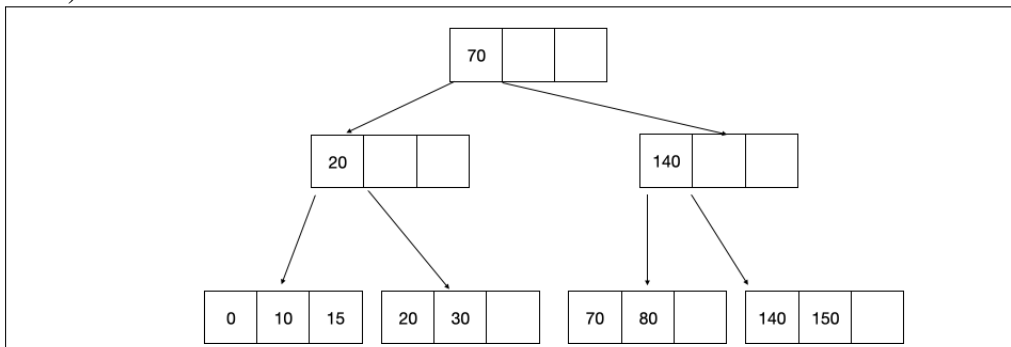(a) **[4 points]** Insert 0^* into the B+tree. Select the resulting tree.

□ A)

| 15 | 70 | 140 |

| 0 | 10 | | | 15 | 20 | 30 | | 70 | 80 | | | 140 | 150 | |

□ B)

| 70 | | |

| 10 | 20 | | | 140 | | |

| 0 | | | | 10 | 15 | | | 20 | 30 | | | 70 | 80 | | | 140 | 150 | |

■ C)

| 20 | 70 | 140 |

| 0 | 10 | 15 | | 20 | 30 | | | 70 | 80 | | | 140 | 150 | |

□ D)

| 70 | | |

| 20 | | | | 140 | | |

| 0 | 10 | 15 | | 20 | 30 | | | 70 | 80 | | | 140 | 150 | |
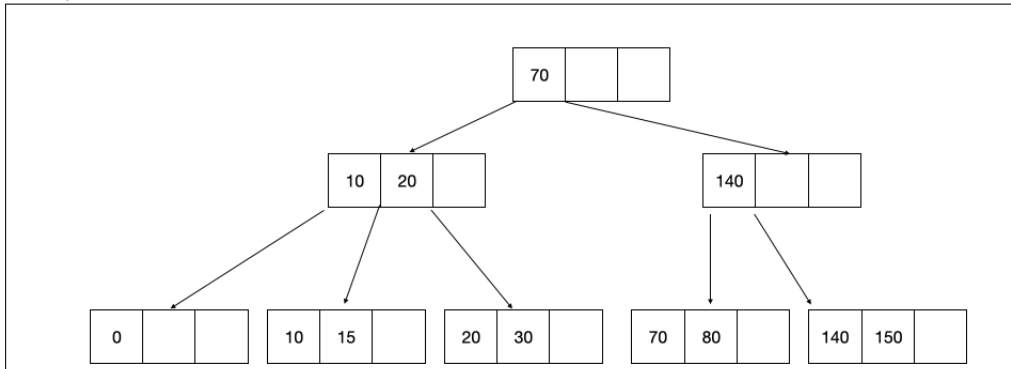
Solution: Inserting 0^* adds one element in the left-most leaf. It should not cause any splits or merges.

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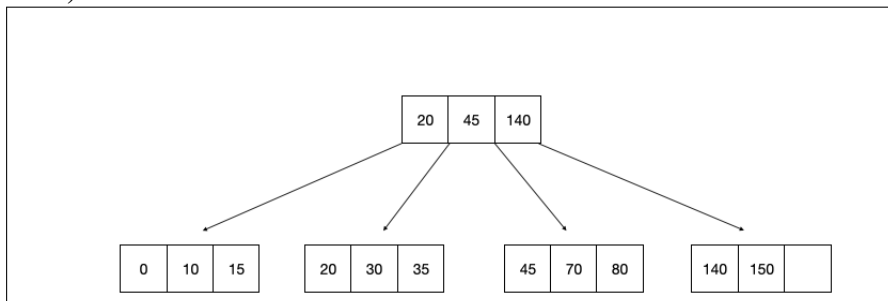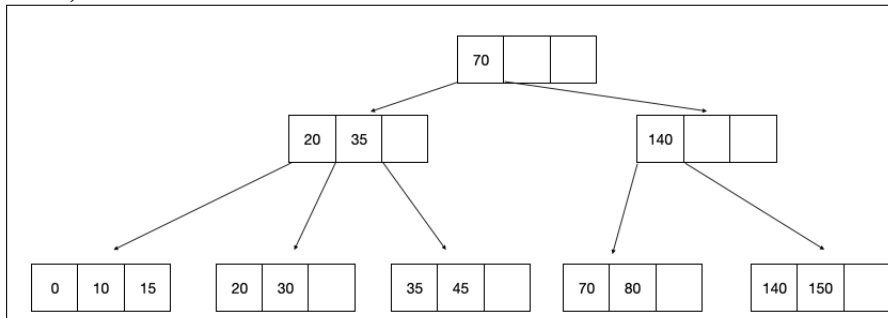
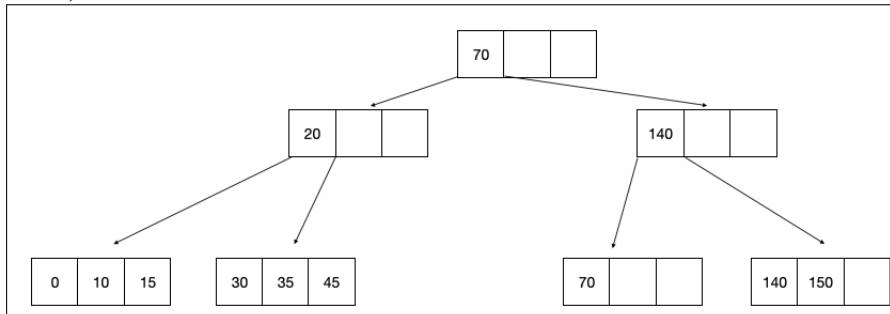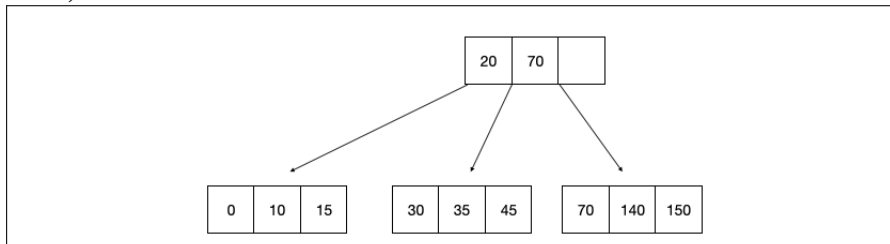☐ A)
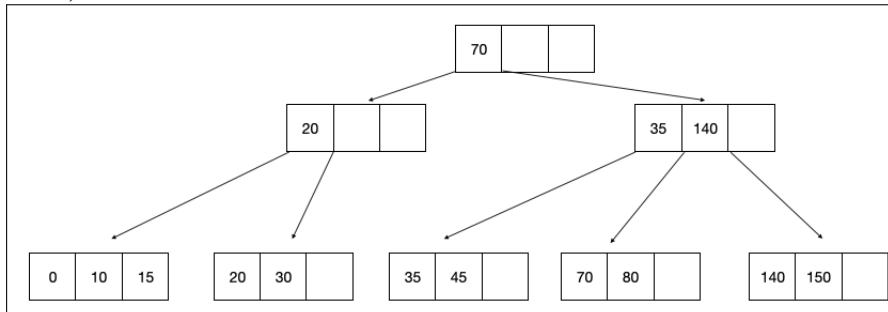
☐ B)
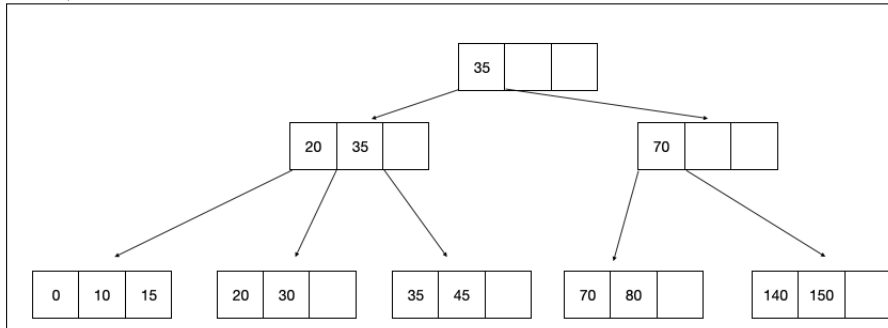
☐ C)

■ 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.
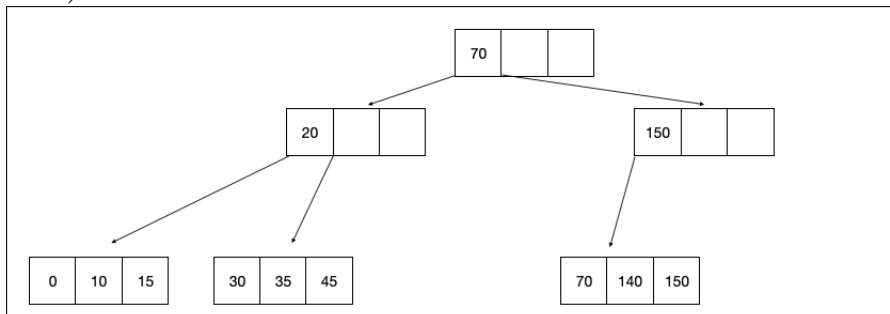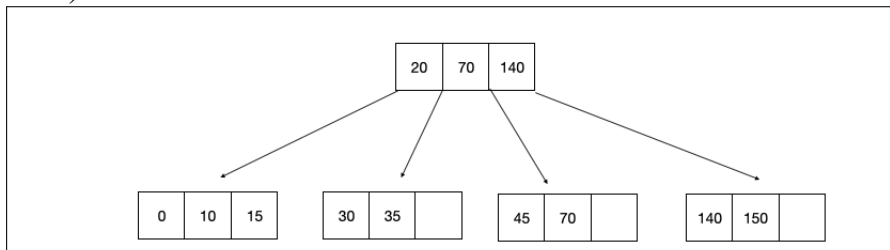
☐ A)

■ 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]** 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.

ii. **[2 points]** Under optimistic latch coupling, write threads never take the write latch on the root to avoid contention.
☐ True ■ **False**

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

iii. **[2 points]** Threads can release their latches in any order.
■ **True** ☐ False

> **Solution:** Threads can release latches in any order.

iv. **[2 points]** "No-Wait" mode for acquiring sibling latches prevents deadlock by allowing a read thread to inspect what another thread is doing.
☐ True ■ **False**

> **Solution:** The "No-Wait" mode *does not* enable inspecting another thread. Rather, a "no-wait" mode prevents threads from getting stuck.

v. **[2 points]** For OLTP-style queries, a DBMS will not benefit from using two separate buffer pools for inner node and leaf 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.

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

Assume that we have a bloom filter that is used to register words relating to common DBMS data structures. 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
"Bloom"	2847	7391
"Hash"	5123	8642
"B+tree"	6078	3210
"Cuckoo"	7598	9845

(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 "Bloom" and "Cuckoo" 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.
$2847 \mod 8 = 7$; $7391 \mod 8 = 7$; $7598 \mod 8 = 6$; $9845 \mod 8 = 5$

(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 "B+tree"?
   □ B+tree has been inserted
   ■ **B+tree has not been inserted**
   □ B+tree may have been inserted
   □ Not possible to know

Solution: $6078 \mod 8 = 6$; $3210 \mod 8 = 2$
Because bit 6 is 0, the filter will return false, so B+tree has not been inserted.

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

> **Solution:** $5123 \mod 8 = 3$; $8642 \mod 8 = 2$
> Because both bits are 1, the filter will return true, meaning Hash may have 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 can eliminate unnecessary disk I/Os.**
☐ We can lower a bloom filter's false positive rate by using more hash functions.
☐ Bloom filters are effective for exact-match (or lookup) queries.
■ **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 require re-balancing after every insertion or deletion.
■ **Radix Trees can be more space-efficient than B+Trees for certain datasets.**
☐ Radix Trees support efficient substring searches (e.g., LIKE "%?%").
☐ Each level in a Radix Tree must have exactly twice the number of nodes as the level above it.
☐ None of the above.

> **Solution:** Radix Trees are indeed efficient for prefix queries and can be more space-efficient than B+Trees for certain datasets, especially when keys share common prefixes.
>
> Radix Trees do not require re-balancing like balanced trees (e.g., AVL or Red-Black Trees).
>
> Radix Trees do not support efficient substring searches; they are optimized for prefix matching.
>
> There is no requirement that each level in a Radix Tree must have exactly twice the number of nodes as the level above it.

(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 Skip Lists both guarantee logarithmic complexity for lookups.
☐ Radix Trees and Inverted Indexes are both efficient at handling range queries.
☐ Skip Lists perform better than B+Trees for range queries.
☐ Update overhead is generally Inverted Index > Skip Lists > B+Tree.
■ **None of the above.**

> **Solution:** All of the above statements are false.
>
> Skip Lists have **approximate** logarithmic complexity for lookups, but it is not guaranteed.
>
> Radix Trees are efficient for prefix queries, but they are not designed for range queries. Inverted Indexes are not efficient for range queries either.
>
> B+Trees generally perform better than Skip Lists for range queries due to their sequential access patterns.
>
> Inverted Indexes are expensive to update, but 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 NOT LIKE
'%organic%';
```

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?

☐ Replace the non-clustering B+Tree with a **clustering** B+Tree index on `description`.
☐ Replace the index with a **hash index** on `description`.
☐ Drop the index and build a **bloom filter** on `description`.
☐ Replace the index with a **trie or 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.