CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (FALL 2024)
PROF. ANDY PAVLO

Homework #3 (by William) – Solutions

Due: **Sunday October 6th, 2024 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Sunday October 6th, 2024.**
- **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 : 2024/10/07 13:13

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 have 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 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 pair `(1,A)` and `(7,B)`. For `(1,A)`, "1" is the key and "A" is the value. Select the value in each entry of the resulting table.

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

Solution: A is inserted into Entry 1, B is inserted into Entry 3.

(b) **[2 points]** After the changes from part **(a)**, delete `(1, A)`, insert key value `(5, D)`, and lastly insert `(9, C)`. Select the value in each entry of the resulting table.

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

Solution: A is first deleted, which inserts a tombstone into entry 1. D is then inserted into entry 1 (since there is nothing there). Then, C is attempted to be inserted into entry 1, but since it's occupied by D, C is inserted into entry 2 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: $h_1(\texttt{x}) = \texttt{(x >> 2) \& 0b11}$.
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 second table.
6. The original entries in the table are shown in the figure below.

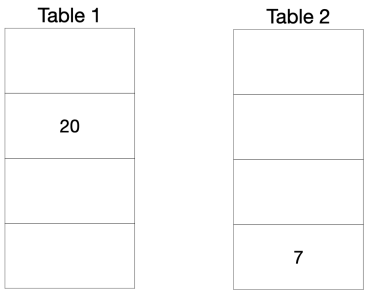


Figure 1: Initial contents of the hash tables.

(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, 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

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

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.

☐ `[0, 4, 17, 20]`

■ **`[0, 4, 16, 20]`**

☐ `[1, 4, 16, 20]`

☐ `[1, 5, 17, 22]`

☐ None of the above

Solution: 0 is inserted into Table 1 Entry 0. 4 is inserted into Table 1 Entry 1. 16 conflicts in Table 1, so is inserted into Table 2 Entry 0. Inserting 20 then starts the infinite loop.

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 ☐ $\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 $g = 7$, 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.

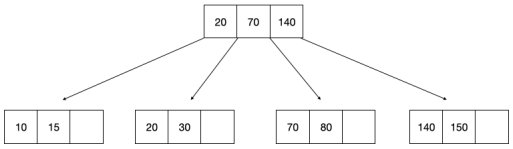


Figure 2: B+ Tree of order $d = 4$ and height $h = 2$.

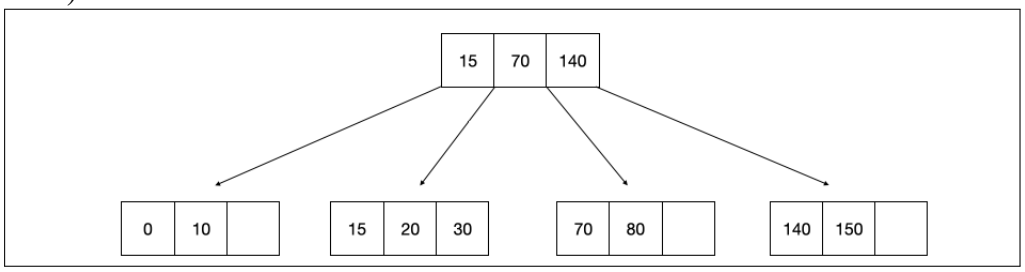
When answering the following questions, be sure to follow the procedures described in class and in your textbook. You can make the following assumptions:

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

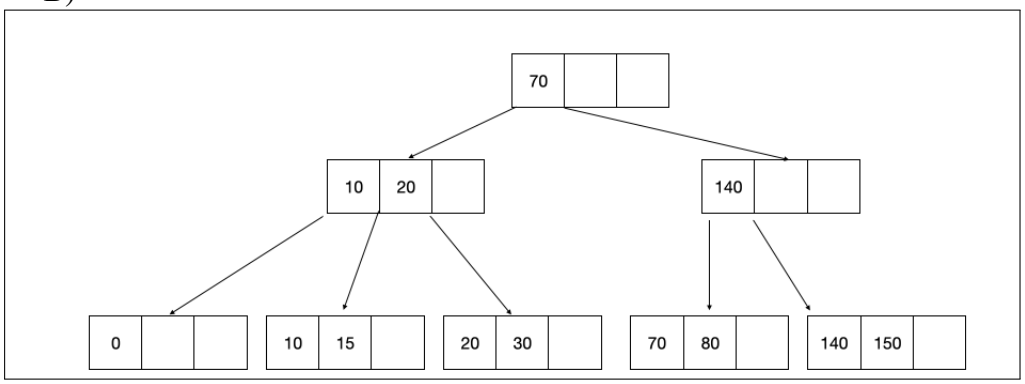
Note that B+ tree diagrams for this problem omit leaf pointers for convenience. The leaves of actual B+ trees are linked together via pointers, forming a singly linked list allowing for quick traversal through all keys.

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

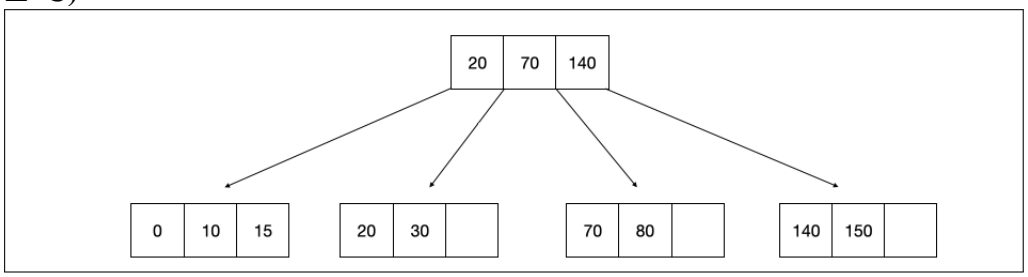
☐ A)



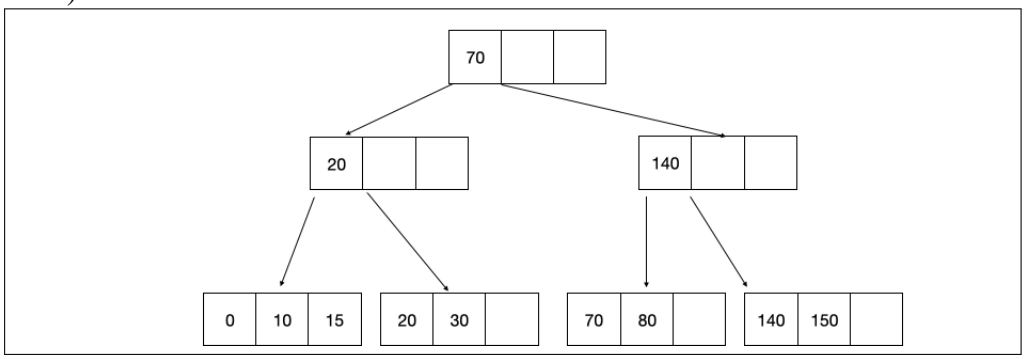
☐ B)



■ **C)**



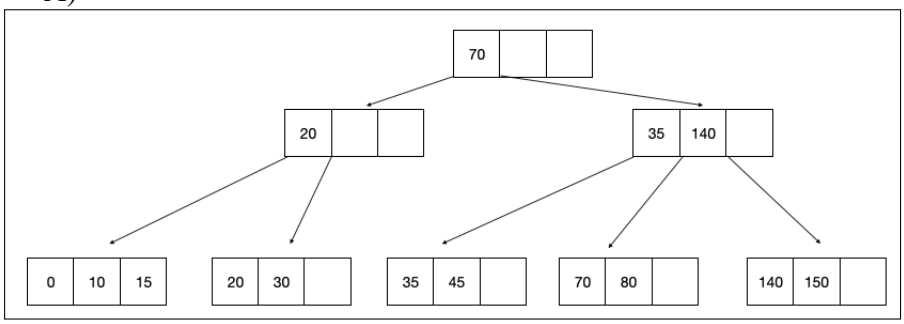
☐ D)



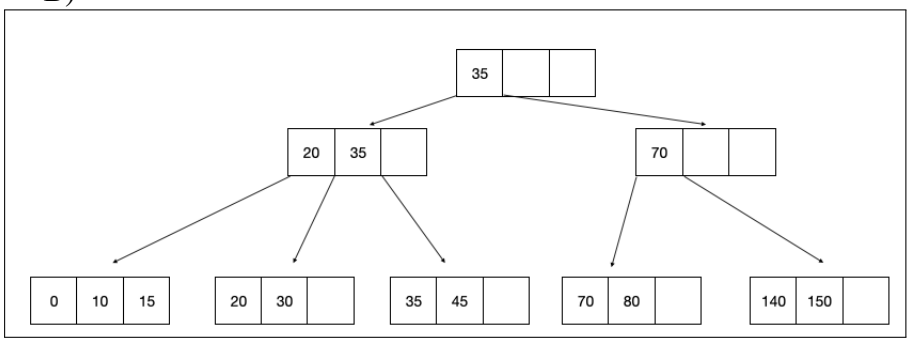
Solution: Inserting 0^* adds one element in the left-most leaf. It should not cause any splits or merges.

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

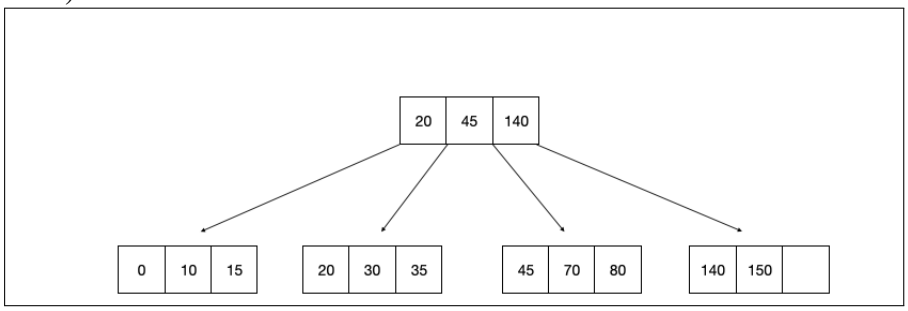
☐ A)



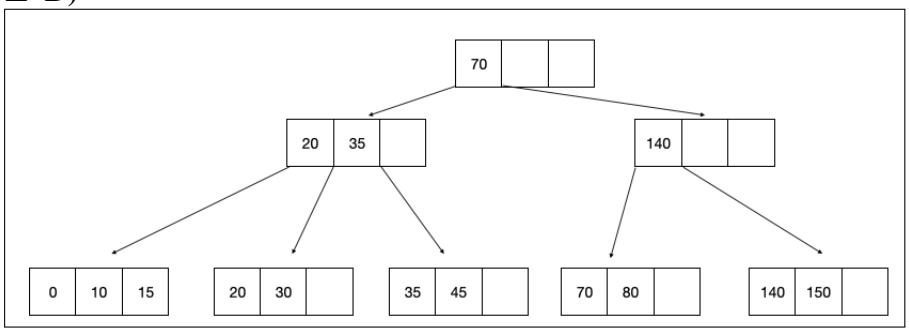
☐ B)



☐ C)



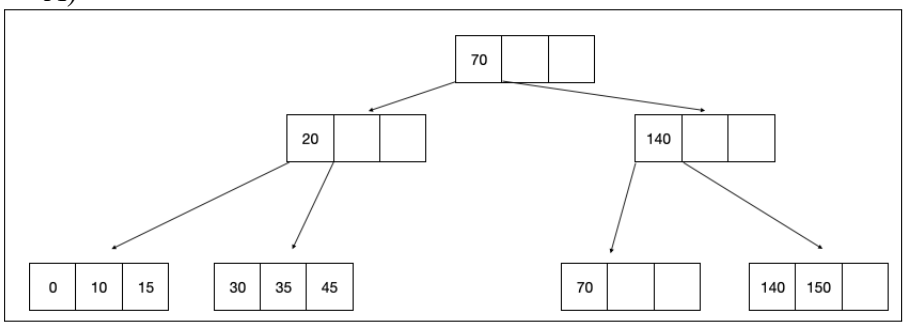
■ **D)**



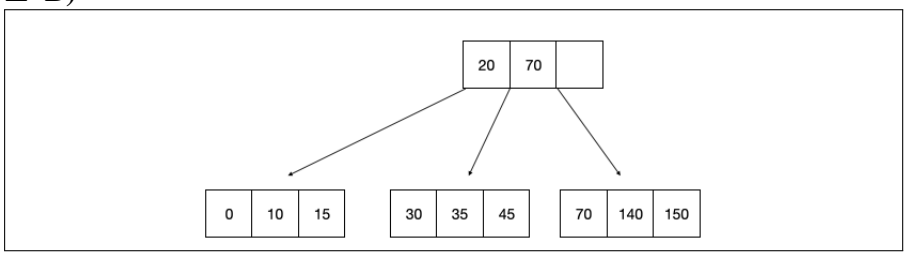
Solution: Inserting 35^* fills in the remaining space of the second leaf node (from the left). After inserting 45^*, the second leaf node splits. As the root-level node is full, the root-level also splits.

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

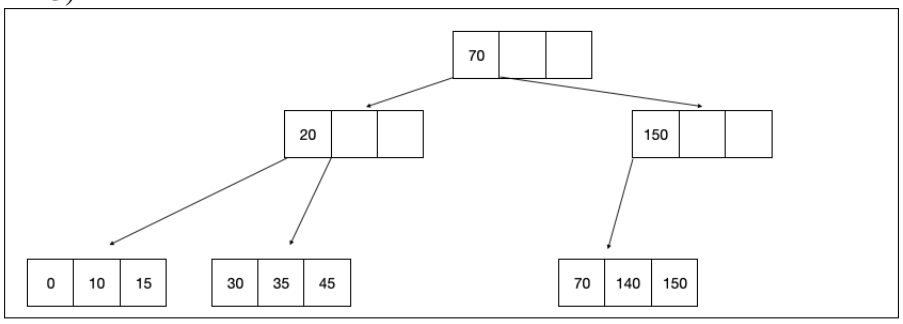
☐ A)



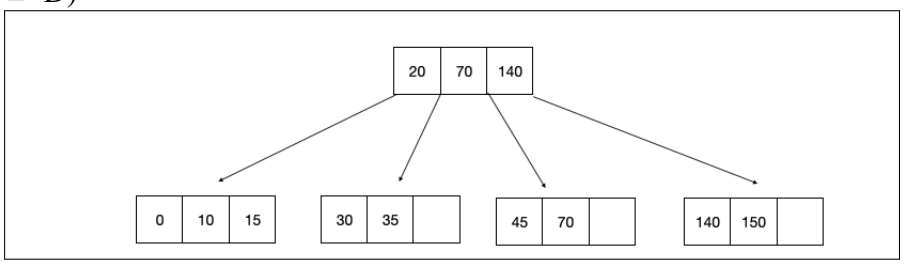
■ **B)**



☐ 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]** 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 names. The filter uses two hash functions h_1 and h_2 which hash the following strings to the following values:

input	h_1	h_2
"DataBootX"	1749	8327
"QueryOptimizeR"	4123	9681
"FilterStream"	5076	2310
"ProtoBloom"	6598	9842

(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 "DataBootX" and "ProtoBloom" 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.
$1749 \mod 8 = 5; 8327 \mod 8 = 7; 6598 \mod 8 = 6; 9842 \mod 8 = 2$

(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	1	0	1	1	1	0	0

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

Solution: $5076 \mod 8 = 4; 2310 \mod 8 = 6$
Because bit 6 is 0, the filter will just return false, so it has not been inserted.

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

Solution: $4123 \mod 8 = 3$; $9681 \mod 8 = 1$
Because both bits are 1, filter will return True, meaning we might have inserted it.

(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 manager is thinking of utilizing a skip list index. They asked a large language model for some factual statements about skip lists but are uncertain about the model's response. They would like you to identify all factually correct statements.

- ■ **Multiple threads can scan, insert, and delete from skip-lists without latches.**
- □ Skip Lists require re-balancing.
- ■ **Single-Linked Skip Lists support finding (keys $\leq$ X) and (keys $\geq$ X).**
- □ When inserting a key into a skip list, the number of towers is a function of the key.
- □ Each level (i) of a skip list *must* have half the nodes as the level below (i+1).
- □ None of the above

Solution: With careful design and using atomic primitives, multiple threads can interact with skip lists without latches. If curious, search for "lock-free skip list".

Single-linked Skip Lists can support both predicates.

When inserting a key into a skip list, the number of towers is randomized. By extension, there is no formal guarantee that a level (i) *must* have half the nodes as the next level below (i+1).

(b) **[5 points]** You are interviewing for a company. The team lead is asking you to compare B+Trees, Skip Lists, Radix Trees, and Inverted Indexes. Select all the true statements.

- □ Both Skip Lists and B+Tree guarantee logarithmic complexity for lookups.
- □ Radix Trees and Inverted Indexes are both efficient at substring predicates.
- □ B+Tree performs better than Radix Trees for prefix 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 not guaranteed.

Radix Trees do not support efficient substring predicates (i.e., LIKE "%?%").

Radix Trees generally perform better than B+Trees for prefix queries.

Inverted index are expensive to update. B+Trees are generally more expensive than skip lists due to the need to potentially split/merge nodes.

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

```
SELECT * FROM PEOPLE WHERE name NOT LIKE '%WuTang%';
```

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

- □ Replace non-clustering B+Tree with a *clustering* B+Tree index on `name`.
- □ Replace the index with a *hash index* on `name`.
- □ Drop the index and build a *bloom filter* on `name`.

□ Replace the index with a *trie or radix tree* on `name`.
■ **None of the above.**

Solution: All of these are ineffective. If these queries dominant the workload, the best thing to do would be to invest in an inverted index. All the above options would not substantially speed up the query.