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

Homework #3 (by Ramsey)
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 15:15

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]

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 `(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

(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

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: $h_1(\texttt{x}) =$ `(x >> 2) & 0b11`.
3. The hashing function of the second table returns the least significant two bits: $h_2(\texttt{x}) =$ `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 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

(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

(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

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

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

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

(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

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

(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

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

(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

(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

(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

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

Consider the following B+tree.

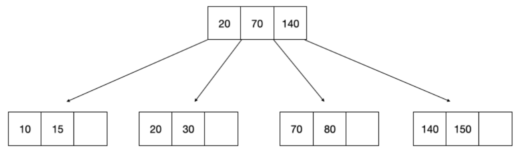


Figure 2: B+ Tree of order $d = 4$ and height $h = 2$.

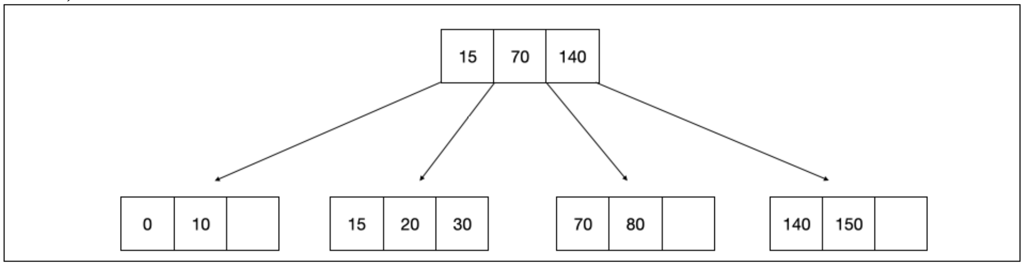
When answering the following questions, be sure to follow the procedures described in class and in your textbook. You can make the following assumptions:

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

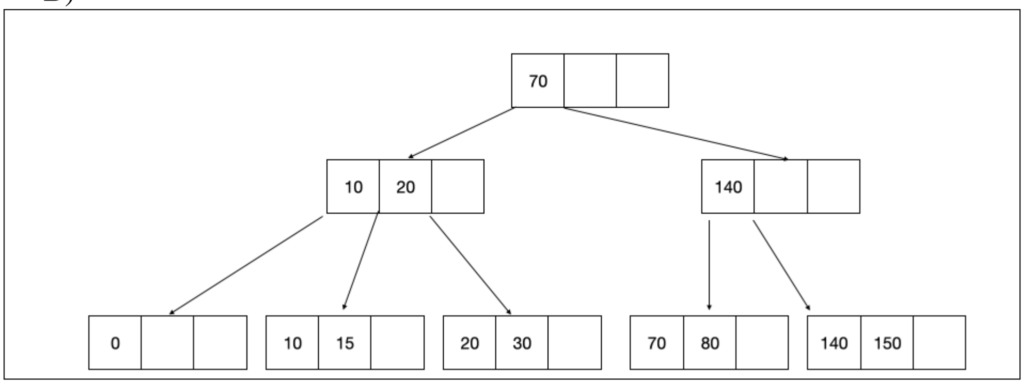
Note that B+ tree diagrams for this problem omit leaf pointers for convenience. The leaves of actual B+ trees are linked together via pointers, forming a singly linked list allowing for quick traversal through all keys.

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

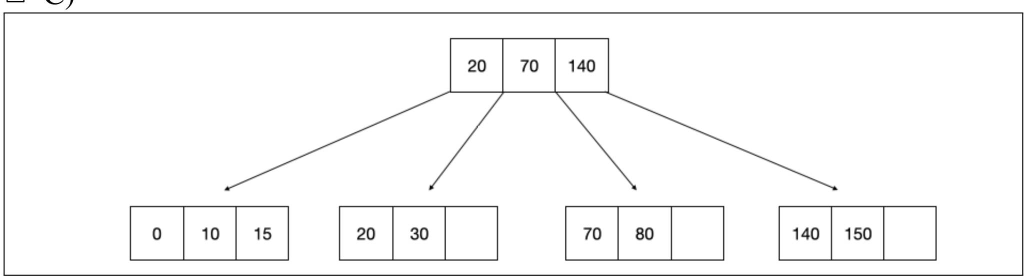
☐ A)



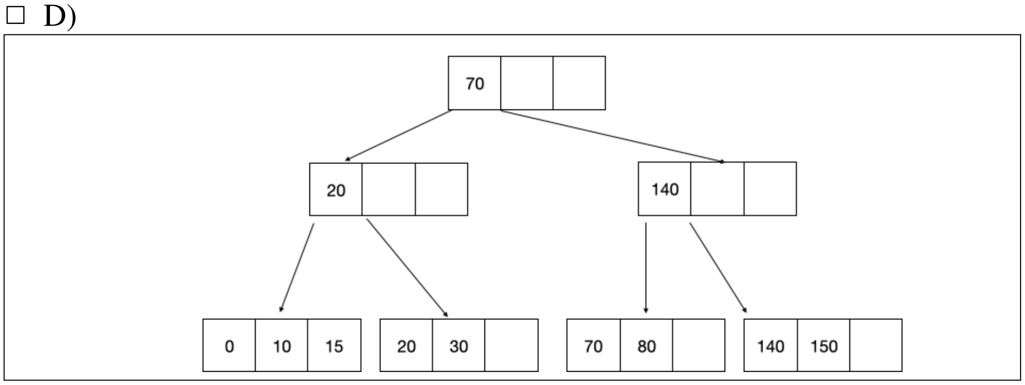
☐ B)



☐ C)

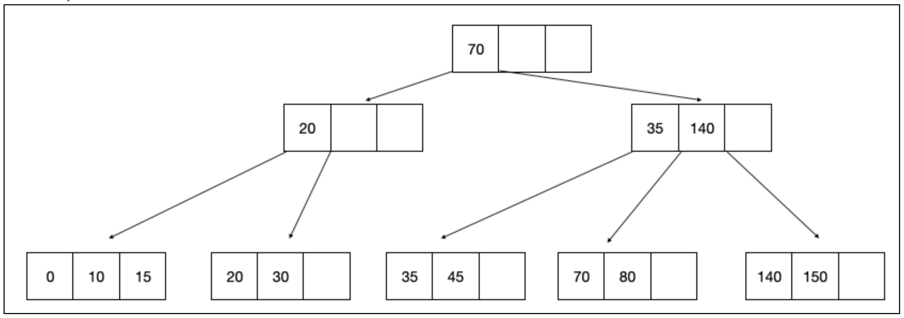


☐ D)

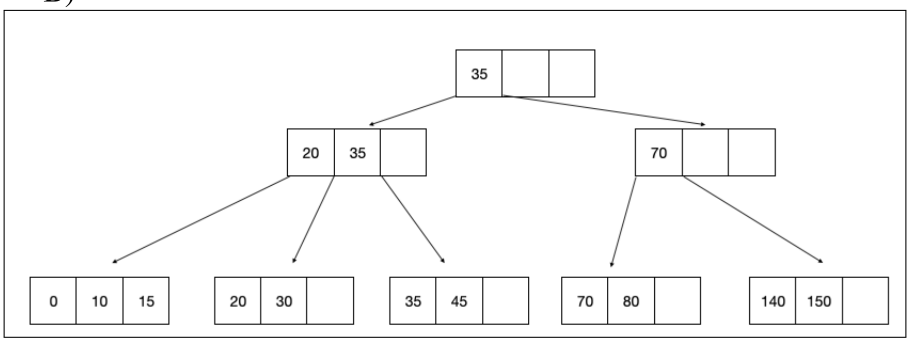


(b) **[5 points]** Starting with the tree that results from **(a)**, insert 35^* and then 45^*. Select the resulting tree.

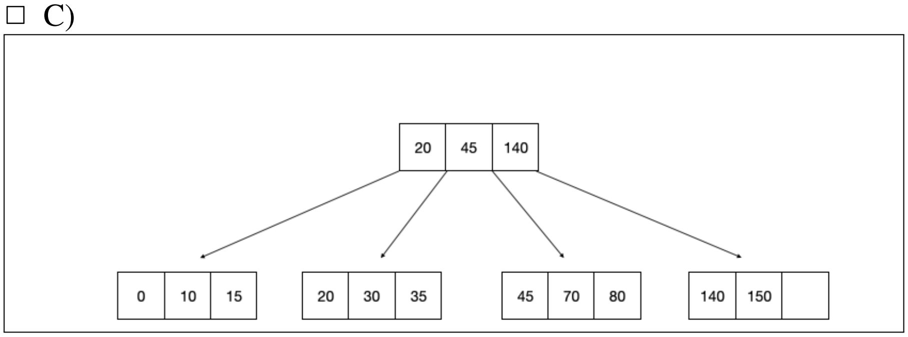
☐ A)



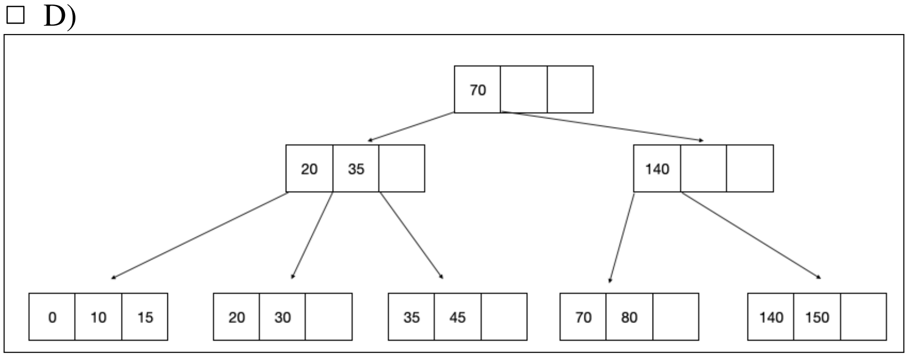
☐ B)



☐ C)

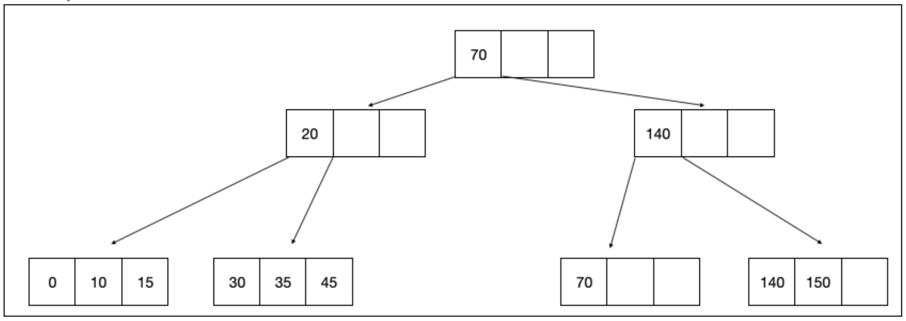


☐ D)

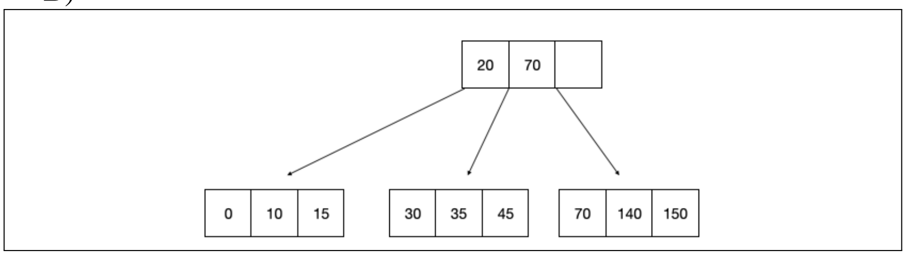


(c) **[8 points]** Starting with the tree that results from **(b)**, deletes 80^* and then 20^*. Select the resulting tree.

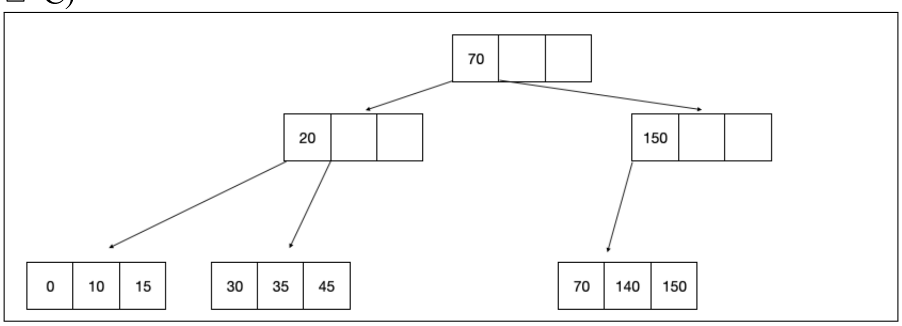
☐ A)



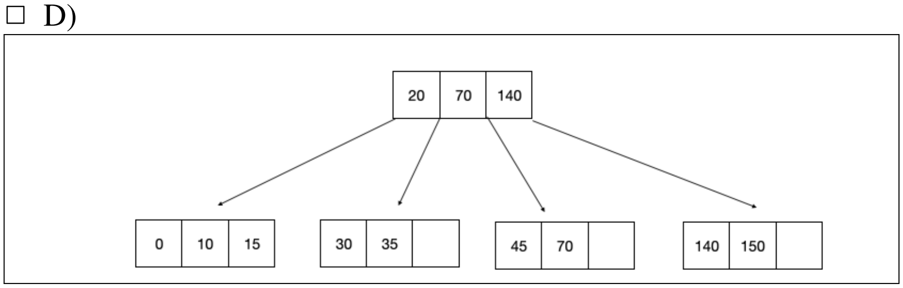
☐ B)



☐ C)



☐ D)



(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

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

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

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

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

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

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

(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

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

(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.

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

(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.

(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.

(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.