CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (SPRING 2023)
PROF. CHARLIE GARROD

Homework #2 (by Arvin Wu) – Solutions

Due: **Friday February 17, 2023 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Friday February 17, 2023.**
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.

For your information:

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

Revision : 2023/02/27 09:54

Question	Points	Score
Storage Models	16	
Cuckoo Hashing	27	
Extendible Hashing	22	
B+Tree	35	
Total:	100	

Question 1: Storage Models . . . . . [16 points]

Graded by:

Consider a database with a single table `T(course_id,course_name,instructor, class_size,hrs_per_week)`, where `course_id` is the *primary key*, and all attributes are the same fixed width. Suppose `T` has 5,000 tuples that fit into 500 pages, Ignore any additional storage overhead for the table (e.g., page headers, tuple headers). Additionally, you should make the following assumptions:

- The DBMS does *not* have any additional meta-data (e.g., sort order, zone maps).
- `T` does *not* have any indexes (including for primary key `course_id`)
- None of `T`'s pages are already in the buffer pool.
- Content-wise, the tuples of `T` will make each query run the longest possible (this assumption is critical for solving part **(a)**)
- The tuples of `T` can be in any order (this assumption is critical for solving part **(b)** when you compute the *minimum* versus *maximum* number of pages that the DBMS will potentially have to read)

(a) Consider the following query:

```
SELECT MAX(hrs_per_week) FROM T
    WHERE class_size > 10;
```

i. **[3 points]** Suppose the DBMS uses the decomposition storage model (DSM) with implicit offsets. How many pages will the DBMS potentially have to read from disk to answer this query? (Keep in mind our assumption about the contents of `T`!)
☐ 1-100 ■ **101-200** ☐ 201-300 ☐ 301-500 ☐ ≥ 501 ☐ Not possible to determine

Solution: 200 pages. There are 100 pages per attribute. 100 pages to find `class_size` for all tuples. In the worst-case scenario for `T`'s content, `hrs_per_week` for all tuples must be accessed as well. Hence, another 100 pages must be read.

ii. **[3 points]** Suppose the DBMS uses the N-ary storage model (NSM). How many pages will the DBMS potentially have to read from disk to answer this query? (Keep in mind our assumption about the contents of `T`!)
☐ 1-100 ☐ 101-200 ☐ 201-300 ■ **301-500** ☐ ≥ 501 ☐ Not possible to determine

Solution: 500 pages. To find `class_size` for all tuples, all pages must be accessed.

(b) Now consider the following query:

```
SELECT course_name, instructor, class_size FROM T
    WHERE course_id = 15445 OR course_id = 15645;
```

i. Suppose the DBMS uses the decomposition storage model (DSM) with implicit offsets.

α) **[3 points]** What is the *minimum* number of pages that the DBMS will potentially have to read from disk to answer this query?

☐ 1 ■ **2-4** ☐ 5-100 ☐ 101-200 ☐ 201-500 ☐ $\geq$ 501
☐ Not possible to determine

Solution: 4 pages. Suppose both primary keys appear on the first page. Since all attributes are of the same fixed width, each attribute of `course_id=15445` and `course_id=15645` will also appear on the same page. We'll thus need to read 1 page to find the two primary keys and read 3 pages to access `course_name`, `instructor`, `class_size` at their corresponding offsets.

β) **[3 points]** What is the *maximum* number of pages that the DBMS will potentially have to read from disk to answer this query?

☐ 1 ☐ 2-4 ☐ 5-100 ■ **101-200** ☐ 201-500 ☐ $\geq$ 501
☐ Not possible to determine

Solution: 106 pages. There are 100 pages per attribute. In the worst case, we scan through all 100 pages to find the two primary keys. In the worst case, the two primary keys will be located on different pages. Since all attributes are of the same fixed width, each attribute of `course_id=15445` and `course_id=15645` will also appear on different pages. Hence we must read 2 pages to access each attribute of `course_id=15445` and `course_id=15645` at their corresponding offsets. Thus, we read 6 pages in total to access `course_name`, `instructor`, `class_size`.

ii. Suppose the DBMS uses the N-ary storage model (NSM).

α) **[2 points]** What is the *minimum* number of pages that the DBMS will potentially have to read from disk to answer this query?

■ **1** ☐ 2-4 ☐ 5-100 ☐ 101-200 ☐ 201-500 ☐ $\geq$ 501
☐ Not possible to determine

Solution: We find the tuples of both primary keys on the first page. No need to look in other pages since all attributes are stored together.

β) **[2 points]** What is the *maximum* number of pages that the DBMS will potentially have to read from disk to answer this query?

☐ 1 ☐ 2-4 ☐ 5-100 ☐ 101-200 ■ **201-500** ☐ $\geq$ 501
☐ Not possible to determine

Solution: 500 pages. At least one tuple with matching primary key is located on the last page. We must thus scan through every page.

Question 2: Cuckoo Hashing . . . . . [27 points]

Graded by:

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.

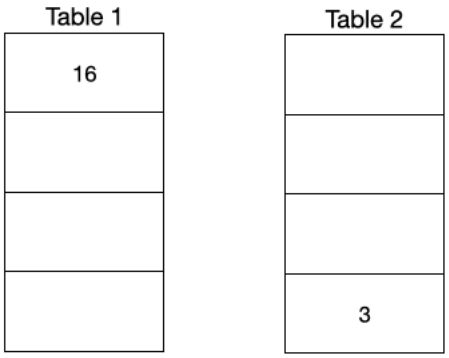


Figure 1: Initial contents of the hash tables.

(a) **[1 point]** Select the sequence of insert operations that results in the initial state.

■ **Insert 16, insert 3** □ Insert 3, insert 16 □ None of the above

Solution: 16 is inserted into table 1 `0b00` based on h_1, 3 experiences a collision and is hashed to table 2 `0b11` based on h_2.

(b) Insert key 8 and then delete 16. Select the value in each entry of the resulting two tables.

i. Table 1

α) **[1 point]** Entry 0 (`0b00`) ☐ 3 ☐ 8 ■ **Empty**
β) **[1 point]** Entry 1 (`0b01`) ☐ 3 ☐ 8 ■ **Empty**
γ) **[1 point]** Entry 2 (`0b10`) ☐ 3 ■ **8** ☐ Empty
δ) **[1 point]** Entry 3 (`0b11`) ☐ 3 ☐ 8 ■ **Empty**

ii. Table 2

α) **[1 point]** Entry 0 (`0b00`) ☐ 3 ☐ 8 ■ **Empty**
β) **[1 point]** Entry 1 (`0b01`) ☐ 3 ☐ 8 ■ **Empty**
γ) **[1 point]** Entry 2 (`0b10`) ☐ 3 ☐ 8 ■ **Empty**
δ) **[1 point]** Entry 3 (`0b11`) ■ **3** ☐ 8 ☐ Empty

Solution: 8 is inserted into table 1 `0b10` based on h_1.

(c) After the changes from part **(b)**, insert key 27 and then insert 4. Select the value in each entry of the resulting two tables.

i. Table 1

α) **[1 point]** Entry 0 (`0b00`) ■ **3** ☐ 8 ☐ 27 ☐ 4 ☐ Empty
β) **[1 point]** Entry 1 (`0b01`) ☐ 3 ☐ 8 ☐ 27 ■ **4** ☐ Empty
γ) **[1 point]** Entry 2 (`0b10`) ☐ 3 ■ **8** ☐ 27 ☐ 4 ☐ Empty
δ) **[1 point]** Entry 3 (`0b11`) ☐ 3 ☐ 8 ☐ 27 ☐ 4 ■ **Empty**

ii. Table 2

α) **[1 point]** Entry 0 (`0b00`) ☐ 3 ☐ 8 ☐ 27 ☐ 4 ■ **Empty**
β) **[1 point]** Entry 1 (`0b01`) ☐ 3 ☐ 8 ☐ 27 ☐ 4 ■ **Empty**
γ) **[1 point]** Entry 2 (`0b10`) ☐ 3 ☐ 8 ☐ 27 ☐ 4 ■ **Empty**
δ) **[1 point]** Entry 3 (`0b11`) ☐ 3 ☐ 8 ■ **27** ☐ 4 ☐ Empty

Solution: 27 is inserted into table 2 `0b11` based on h_2, replacing 3, which is rehashed into table 1 `0b00`. 4 is inserted into table 1 `0b01` based on h_1.

(d) After the changes from parts **(b)** and **(c)**, insert key 19 and then delete 27. Select the value in each entry of the resulting two tables.

i. Table 1

α) **[1 point]** Entry 0 (`0b00`) ■ **3** □ 8 □ 4 □ 19 □ Empty

β) **[1 point]** Entry 1 (`0b01`) □ 3 □ 8 ■ **4** □ 19 □ Empty

γ) **[1 point]** Entry 2 (`0b10`) □ 3 □ 8 □ 4 □ 19 ■ **Empty**

δ) **[1 point]** Entry 3 (`0b11`) □ 3 □ 8 □ 4 □ 19 ■ **Empty**

ii. Table 2

α) **[1 point]** Entry 0 (`0b00`) □ 3 ■ **8** □ 4 □ 19 □ Empty

β) **[1 point]** Entry 1 (`0b01`) □ 3 □ 8 □ 4 □ 19 ■ **Empty**

γ) **[1 point]** Entry 2 (`0b10`) □ 3 □ 8 □ 4 □ 19 ■ **Empty**

δ) **[1 point]** Entry 3 (`0b11`) □ 3 □ 8 □ 4 ■ **19** □ Empty

Solution: Inserting 19 into table 2 `0b11` replaces 27. 27 is rehashed into table 1 `0b10`, replacing 8. 8 is rehashed into table 2 `0b00`. 27 is then deleted.

(e) **[2 points]** What is the smallest key that potentially causes an infinite loop given the table that results from part **(d)**?

□ 33 □ 34 ■ **35** □ 36 □ 37 □ 51 □ None of the above

Solution: 3, 19, 35 all have $h_1(x) = 0$, $h_2(x) = 3$, which would continue to replace each other.

Question 3: Extendible Hashing . . . . . [22 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

(a) Starting from an empty table, insert keys 6, 1.

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 6?

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

Solution: There is only one bucket (created on initialization), and it holds both 6 and 1. Since no split has occurred yet, the bucket has local depth $d = 0$.

(b) Starting from the result in **(a)**, you insert keys 17, 10.

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

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

Solution: One split occurred when inserting 17, because the first bucket (on initialization) is full. After this split:

- The global depth is incremented by 1 to $g = 1$
- The table doubles in size (the hashing function now reads one more bit than before)
- A new bucket is created and the keys in the bucket that caused the split are rehashed. Bucket `b0` holds 6 while bucket `b1` holds 1 and 17

Key 10 is inserted into bucket `b0` (no splitting occurs).
Global depth is now $g = 1$.

ii. **[1 point]** What is the local depth of the bucket containing 10?

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

Solution: Inserting 17 causes a split in the first bucket (created on initialization), so a new bucket is created. The table now has two buckets in total (`b0` and `b1`), and keys from the *split bucket* are rehashed into buckets `b0` and `b1`. The local depth of *destination buckets* equals local depth of *split bucket* plus 1. Thus, buckets `b0` and `b1` have local depth $d = 1$.

(c) Starting from the result in **(b)**, you insert key 25.

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

Solution: Inserting 25 causes 3 consecutive splits to occur, because 25, 1, and 17 all have `001` as their lowest 3 bits. After the first two splits, $g = 3$. Keys 25, 1, and 17 will all still be hashed to the same bucket `b001`. The third and final split will increment global depth to $g = 4$, after which 1 and 17 will be hashed to bucket `b0001` while 25 will be hashed to bucket `b1001`.
Hence global depth is $g = 4$.

ii. **[3 points]** What is the local depth of the bucket containing 25?
☐ 0 ☐ 1 ☐ 2 ☐ 3 ■ **4** ☐ None of the above

Solution: Key 25 is hashed to bucket `b1001`, which splitted from `b001`, which splitted from `b01`, which splitted from `b1`. These three splits causes bucket `b1001` to have depth $d = 4$.

iii. **[2 points]** What is the local depth of the bucket containing 17?
☐ 0 ☐ 1 ☐ 2 ☐ 3 ■ **4** ☐ None of the above

Solution: Key 17 is hashed to bucket `b0001`, which splitted from `b001`, which splitted from `b01`, which splitted from `b1`. These three splits causes bucket `b1001` to have depth $d = 4$.

iv. **[1 point]** What is the local depth of the bucket containing 6?
☐ 0 ■ **1** ☐ 2 ☐ 3 ☐ 4 ☐ None of the above

Solution: Key 6 is hashed to bucket `b0110`. Bucket `b0110` is the same bucket as buckets `b0000`, `b0010`, `b0100`, `b1000`, `b1010`, `b1100`, and `b1110`. This is because the second, third, and fourth splits (during each the table doubled in size) did not occur on bucket `b0` (bucket `b0110` point to the same bucket as `b0` earlier). Thus bucket `b0110` has the same local depth as `b0`, which is $d = 1$.

v. **[1 point]** Which value, if inserted, will hash to the same bucket as the bucket containing key 10?
■ **4** ☐ 7 ☐ 9 ☐ 13 ☐ None of the above

Solution: Key 4 will hash to `0100`. Key 10 hashes to `1010`. Both hash results point to the same bucket, as explained above.

(d) **[2 points]** Starting from the result in **(c)**, which **key(s)**, if inserted next, will cause a split that doubles the table's size?
☐ 5 ☐ 12 ☐ 24 ☐ 29 ■ **33** ☐ None of the above

Solution: To cause a split that doubles the table size, one must insert a key that hashes to a full bucket whose local depth equals global depth. Key 33 is the only one that matches such requirement. Key 33 will hash to `0001`, so it will be placed in bucket `b0001`. Since, bucket `b0001` is full and has local depth $d = 4 = g$, inserting 33 will double the table's size.

(e) **[2 points]** Starting from the result in **(c)**, which **key(s)**, if inserted next, will cause a split **without** doubling the table's size?

☐ 5 ■ **12** ■ **24** ☐ 29 ☐ 33 ☐ None of the above

Solution: To cause a split without doubling the table size, one must insert a key that hashes to a full bucket whose local depth is less than global depth. Key 12 and 24 are the only ones that match such requirement. Key 12 will hash to `1100`, so it will be placed in bucket `b1100`, which is same as all other buckets ending in `0`. Since, bucket `b1100` is full (containing 6 and 10) and has local depth $d = 1 < g$, inserting 12 will cause a split without doubling the table's size. Key 24 will hash to `1000`, so it will be placed in bucket `b1000`, which is same as all other buckets ending in `0`. For the same reason as 12, this insertion will cause a split without doubling the table's size.

(f) **[3 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 ☐ $\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. Keys 64, 128, and 256 share the same lower 6 bits. Hence, the table will split until global depth reaches $g = 7$. When $g = 7$, 64 will no longer be hashed to the same bucket as 128 and 256.

Question 4: B+Tree . . . . . [35 points]

Graded by:

Consider the following B+tree.

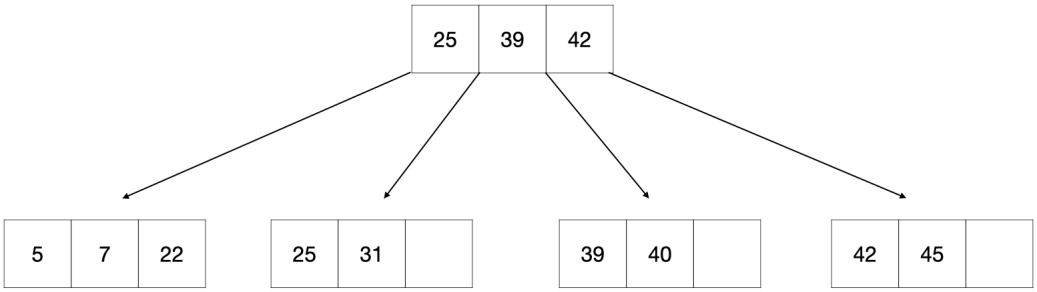


Figure 2: B+ Tree of order $d = 4$ and height $h = 2$.

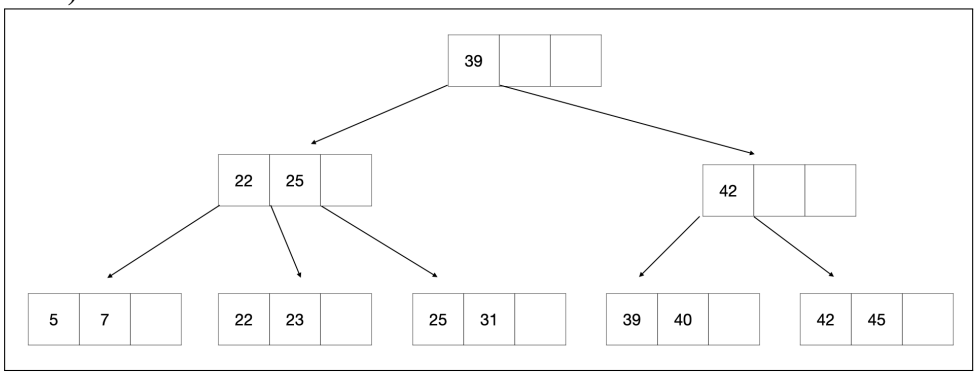
When answering the following questions, be sure to follow the procedures described in class and in your textbook. You can make the following assumptions:

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

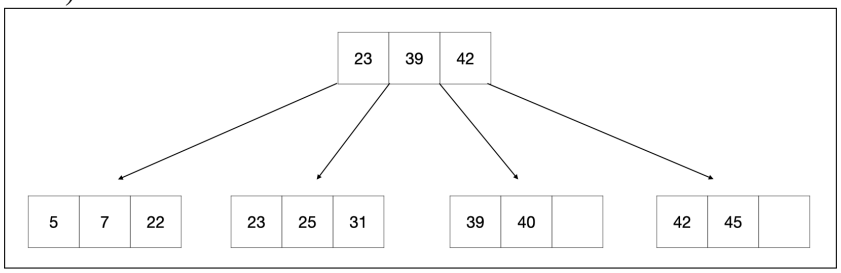
Note that B+ tree diagrams for this problem omit leaf pointers for convenience. The leaves of actual B+ trees are linked together via pointers, forming a singly linked list allowing for quick traversal through all keys.

(a) **[5 points]** Insert 23^* into the B+tree. Select the resulting tree.

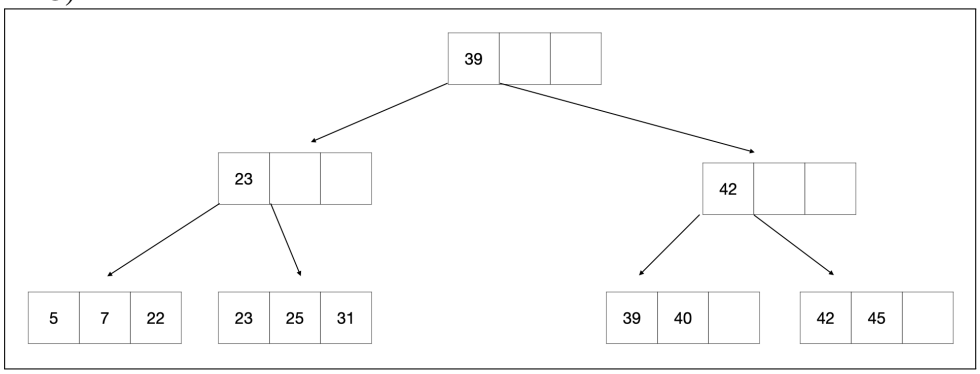
■ **A)**



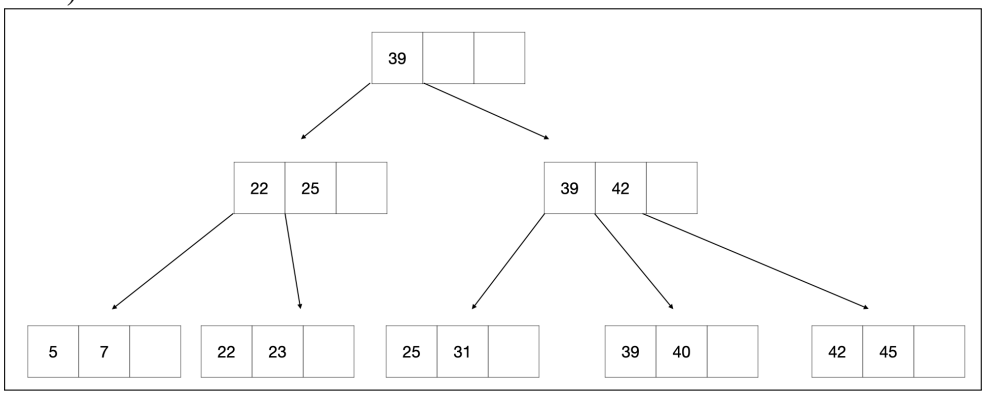
□ B)



□ C)



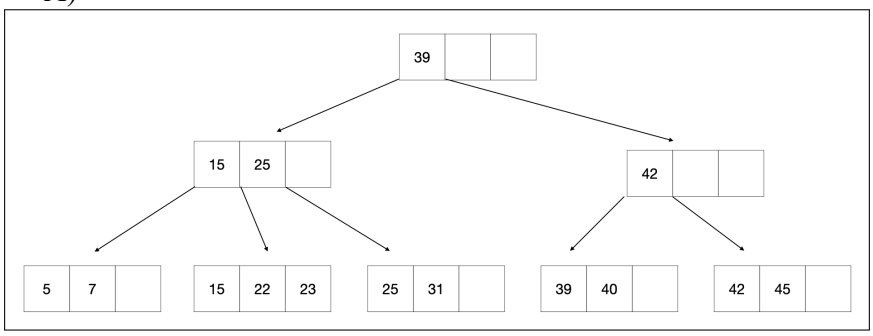
□ D)



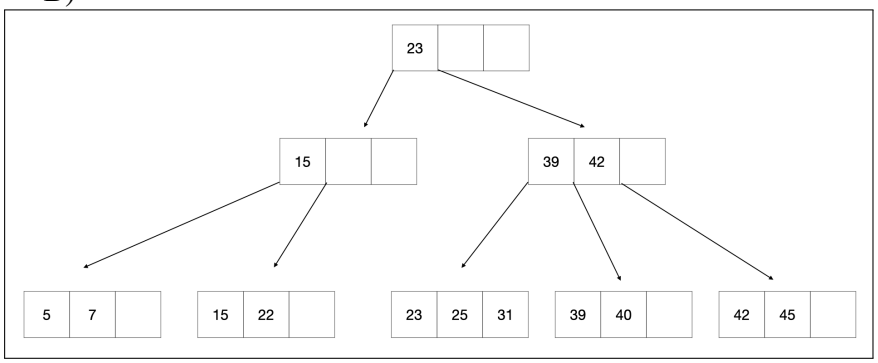
Solution: Inserting 23^* causes the left-most leaf to be split. 22^* is copied up as a result, which causes a split in the root. The split root pushes up 39^* to be the new root.

(b) **[5 points]** Starting with the tree that results from **(a)**, insert 15^*. Select the resulting tree.

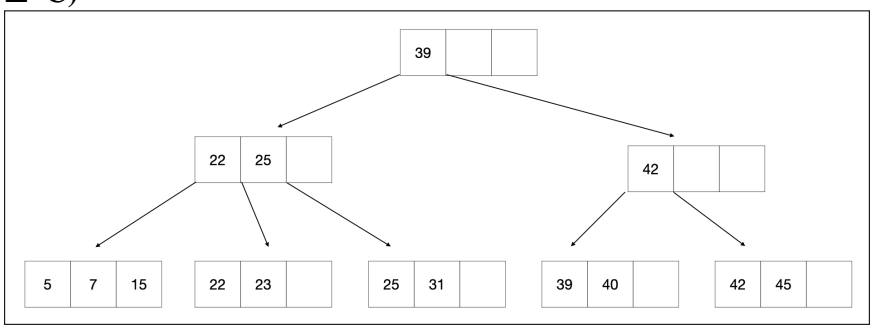
☐ A)



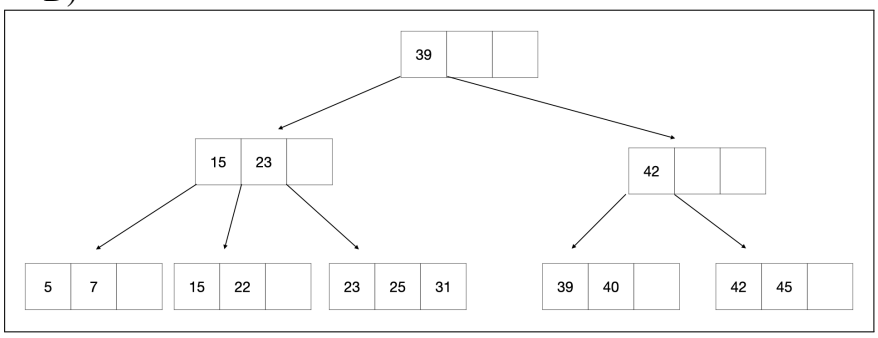
☐ B)



■ **C)**



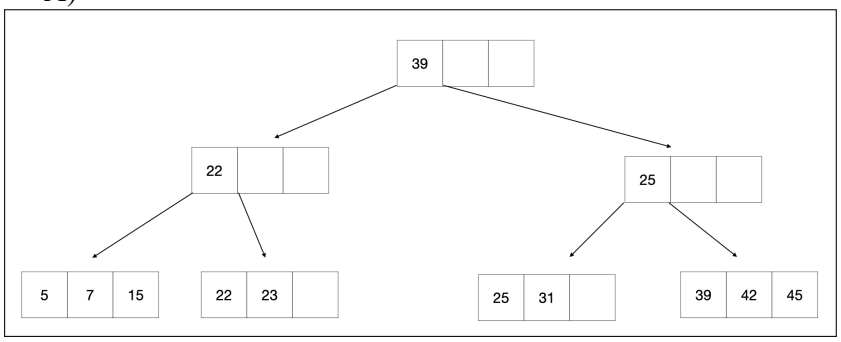
☐ D)



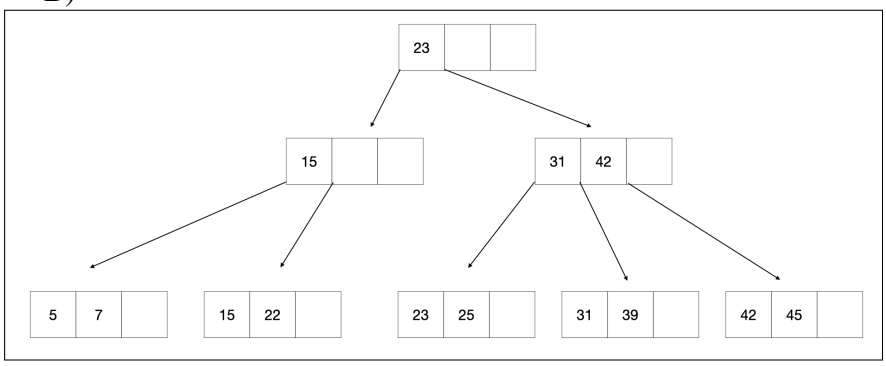
Solution: Since the left-most leaf still has space, 15^* can be inserted without further action.

(c) **[5 points]** Starting with the tree that results from **(b)**, delete 40^*. Select the resulting tree.

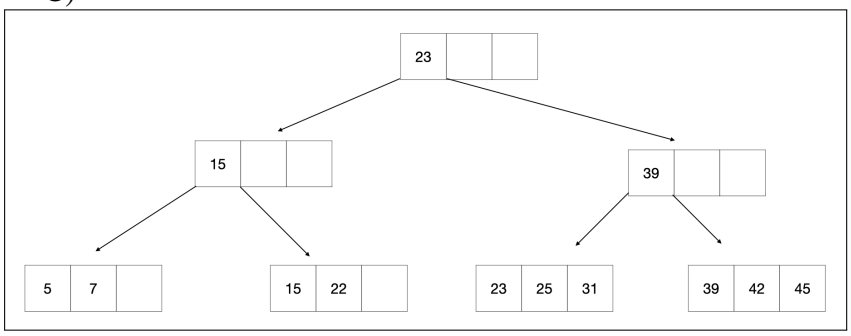
☐ A)



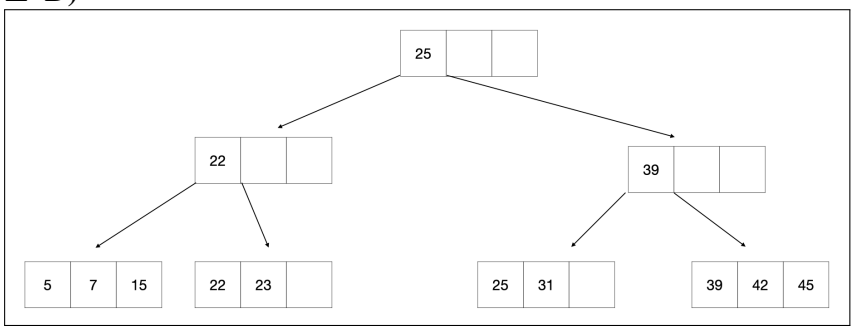
☐ B)



☐ C)



■ **D)**

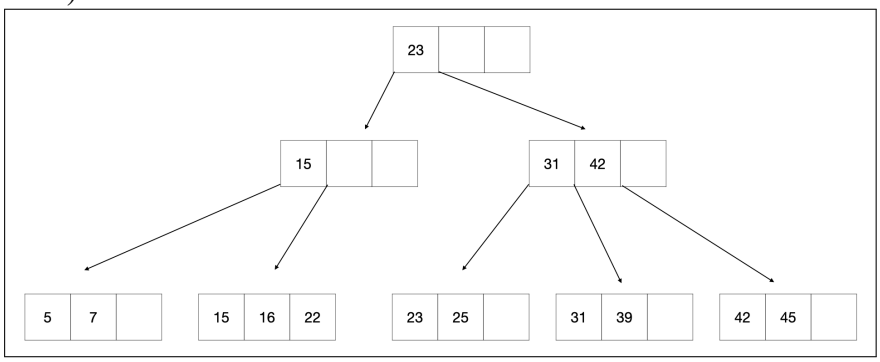


Solution: Deleting 40^* causes the leaf node holding it to underflow. As borrowing from sibling fails, the leaf node (originally holding 40^*) merges with the leaf node to the right.

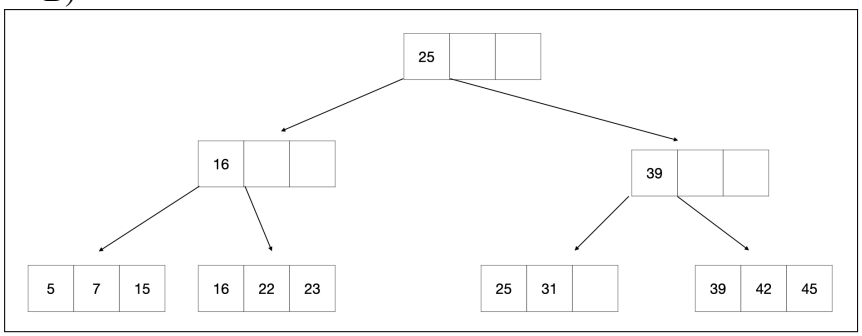
Merging causes 42^* to be deleted from their parent. Their parent succesfully borrows from the internal node to the left, modifying the root to reflect the change in key.

(d) **[5 points]** Starting with the tree that results from **(c)**, insert 16^*. Select the resulting tree.

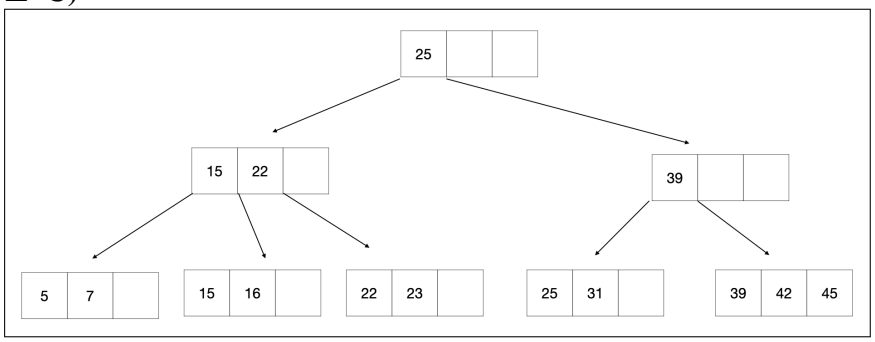
☐ A)



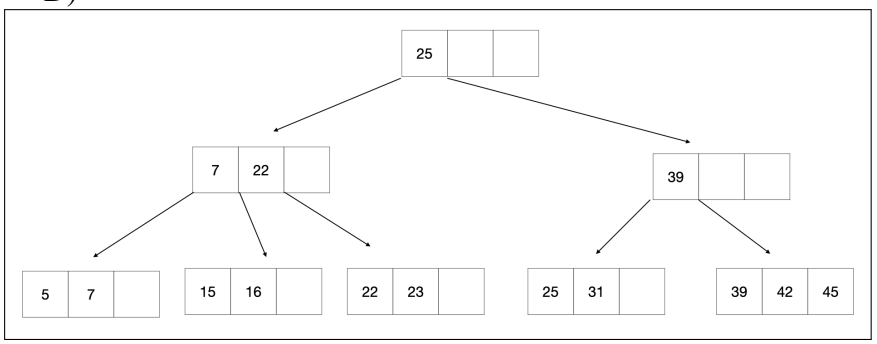
☐ B)



■ C)



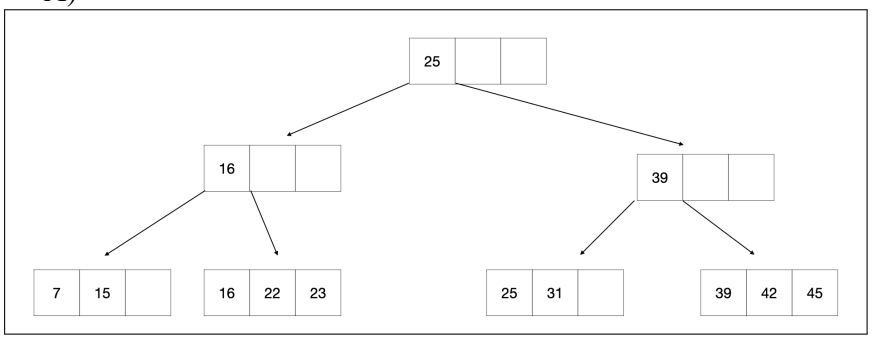
☐ D)



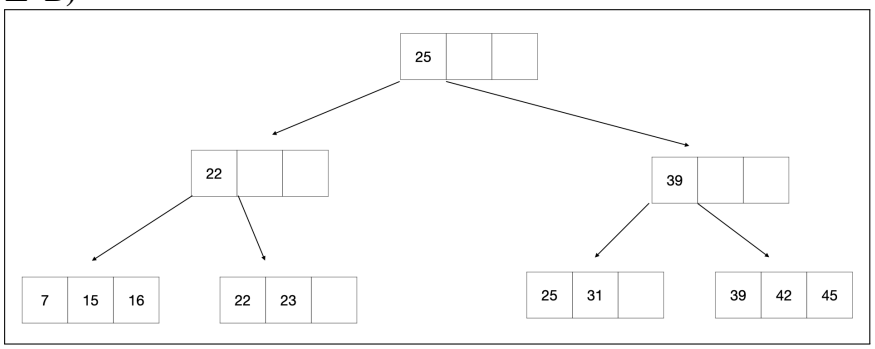
Solution: Inserting 16^* causes the left-most leaf node to be split, copying up 15^* to the parent. Since the parent has enough space to hold 15^*, no further action is necessary.

(e) **[5 points]** Finally, starting with the tree that results from **(d)**, delete 5^*. Select the resulting tree.

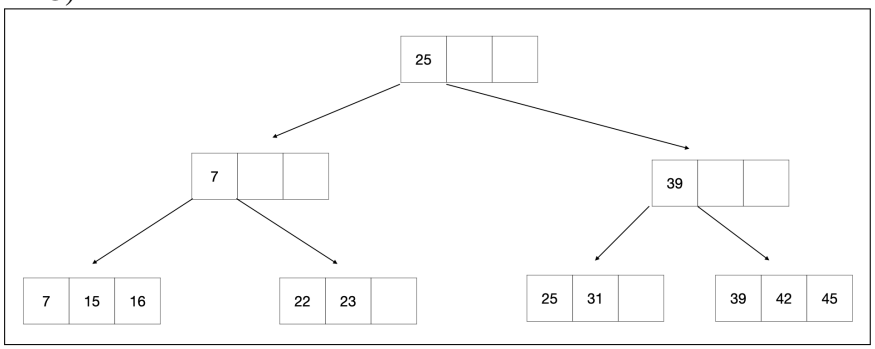
☐ A)



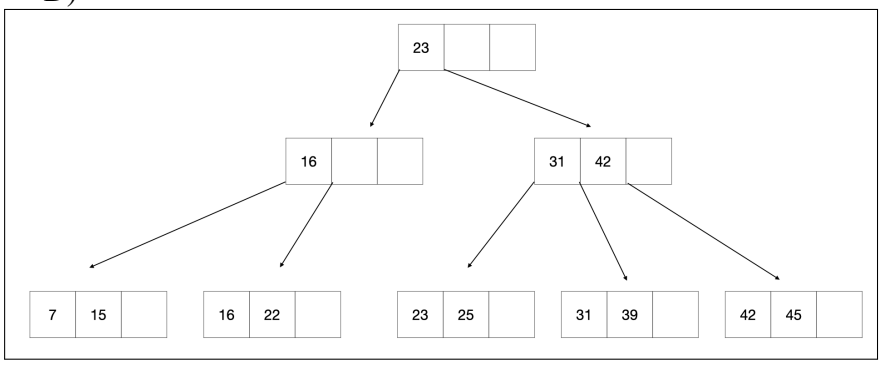
■ **B)**



☐ C)



☐ D)



Solution: Deleting 5^* causes the left-most leaf to underflow. Failing to borrow from its sibling, the leaf node (originally holding 5^*) merges with the leaf to the right and removes 15^* from the parent. The parent does not underflow after such removal, so no further action is necessary.

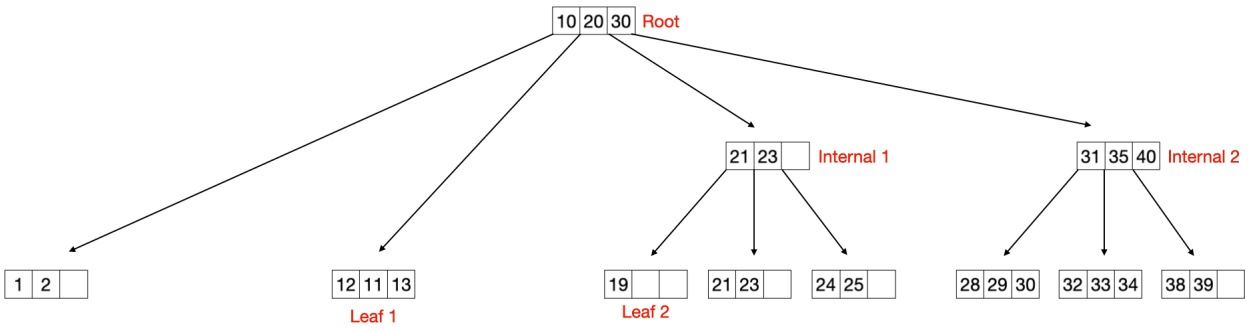


Figure 3: B+tree with violations

(f) The B+Tree shown in Figure 3 is invalid. That is, its nodes violate the correctness properties of B+Trees that we discussed in class. If the tree is invalid, select all the properties that are violated for each node. If the node is valid, then select 'None'. There will be **no** partial credit for missing violations.

Note:

- *If a node's subtrees are not the same height, the balance property is violated at that node only.*
- *If a node's subtrees contain values not in the range specified by the node's separator keys, the separator keys property is violated at that node.*

i. **[2 points]** Which properties are violated by **Leaf 1**?
■ **Key order property** ☐ Half-full property ☐ Balance property
☐ Separator keys ☐ None

Solution: Keys should be in the order of 11, 12 ,13.

ii. **[2 points]** Which properties are violated by **Leaf 2**?
☐ Key order property ■ **Half-full property** ☐ Balance property
☐ Separator keys ☐ None

Solution: There must be at least 2 keys.

iii. **[2 points]** Which properties are violated by **Internal Node 1**?
☐ Key order property ☐ Half-full property ☐ Balance property
■ **Separator keys** ☐ None

Solution: The node's middle child contains 23. According to the node's separator keys, the middle child can only contain values between 21 (inclusive) and 23 (exclusive).

iv. **[2 points]** Which properties are violated by **Internal Node 2**?
☐ Key order property ☐ Half-full property ☐ Balance property
■ **Separator keys** ☐ None

Solution: There are 3 separator keys, but only 3 children pointers (there should be 4).

v. **[2 points]** Which properties are violated by **Root**?

☐ Key order property ☐ Half-full property ■ **Balance property**
■ **Separator keys** ☐ None

Solution: The root is imbalanced, as it has both Leaf 1 and Internal Node 1 as its children. The root's subtree containing Leaf 2 contains the value 19, violating the root's separator keys.