CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (FALL 2023)
PROF. ANDY PAVLO AND JIGNESH PATEL

# Homework #2 (by Wiam Eddahri) – Solutions
Due: **Sunday September 24 2023 @ 11:59pm**

**IMPORTANT:**
- Enter all of your answers into **Gradescope by 11:59pm on Sunday September 24 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; **3** questions total
- Rough time estimate: ≈4-6 hours (1-1.5 hours for each question)

*Revision* : 2023/10/09 11:08

| Question | Points | Score |
|---|---|---|
| Storage Models | 16 | |
| Extendible Hashing | 19 | |
| B+Tree | 65 | |
| Total: | 100 | |

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

**Graded by:**

Consider a database with a single table `T(andrew_id,full_name,graduation_year, department,gpa)`, where `andrew_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 `andrew_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(gpa) FROM T
    WHERE graduation_year == 2025;
```

i. **[4 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 ☐ $\geq 501$ ☐ Not possible to determine

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

ii. **[4 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`!)
☐ 1-200 ☐ 201-300 ☐ 301-400 [✓ **401-500**] ☐ $\geq 501$ ☐ Not possible to determine

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

(b) Now consider the following query:

```
SELECT andrew_id, gpa FROM T
    WHERE gpa = (SELECT MAX(gpa) FROM T);
```

Question 1 continues...

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

$\alpha$) **[4 points]** What is the *minimum* number of pages that the DBMS will potentially have to read from disk to answer this query?
☐ 1 ☐ 2-5 [✓ **100-200**] ☐ 201-299 ☐ $\geq 300$ ☐ Not possible to determine

**Solution:** 200 pages. 100 pages for the inner select, and 100 pages to get the andrew_id since the the buffer pool will have the gpa pages from the inner select.

$\beta$) **[4 points]** What is the *maximum* number of pages that the DBMS will potentially have to read from disk to answer this query?
☐ 1 ☐ 2-5 [✓ **100-200**] ☐ 201-299 ☐ $\geq 300$ ☐ Not possible to determine

**Solution:** 200 pages. 100 pages for the inner select, and 100 pages to get the andrew_id since the the buffer pool will have the gpa pages from the inner select.

## Question 2: Extendible Hashing................................[19 points]

**Graded by:**

Consider an extendible hashing structure such that:

- Each bucket can hold up to four 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 0, 1, 4, 5.

i. **[2 points]** 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 4 arbitrary values. Thus global depth is same as its initial value of 0.

ii. **[2 points]** What is the local depth of the bucket containing 5?
[✓ **0**] ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ None of the above

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

(b) Starting from the result in **(a)**, you insert keys 10, 11, 12, 13.

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 10, 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.

ii. **[2 points]** What is the local depth of the bucket containing 11?
☐ 0 [✓ **1**] ☐ 2 ☐ 3 ☐ 4 ☐ None of the above

**Solution:** 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 keys 2 and 14.

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

Question 2 continues...

**Solution:** Inserting 2 causes one split to occur, because keys 0, 4, 10, 12, and 2 all hash to the same bucket.
Hence global depth is $g = 2$.

ii. **[3 points]** Starting from the result in **(c)**, you insert keys 15, 3 and 7, what is the global depth?
☐ 0 ☐ 1 [✓ **2**] ☐ 3 ☐ 4 ☐ None of the above

**Solution:** The bucket holding 1,5,13,11 will split, causing its local depth to be 2, the global depth doesn't change.

iii. **[3 points]** Which value, if inserted, will hash to the same bucket as the bucket containing key 1?
☐ 3 ☐ 7 ☐ 11 ☐ All of the above [✓ **None of the above**]

**Solution:** We need a key that hashes to 01

(d) **[3 points]** Starting from the result in **(c)**, which **key(s)**, if inserted next, will cause a split that doubles the table's size?
☐ 33 ☐ 64 ☐ 29 ☐ 61 ☐ All of the above [✓ **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. A key that hashes to 0b11 is the only key that can cause a split that will double the table's size, but none of the keys hash to 0b11.

## Question 3: B+Tree ........................................ [65 points]

**Graded by:**

Consider the following B+tree.

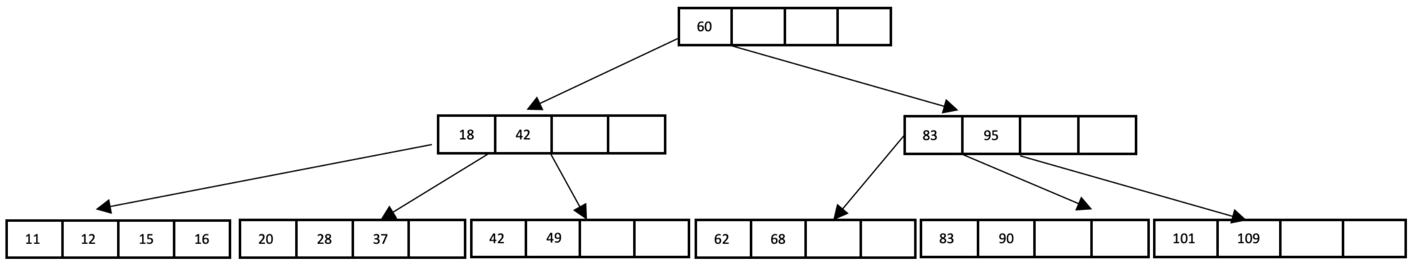


Figure 1: B+ Tree of order $d = 5$ and height $h = 3$.

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

- You should always consider redistribution before trying to merge two nodes.

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.

Question 3 continues…

(a) **[5 points]** How many tree nodes must be fetched to answer the following query: Get all records with search key smaller than 42.

☐ 6 ☐ 7 [✓ **5**]

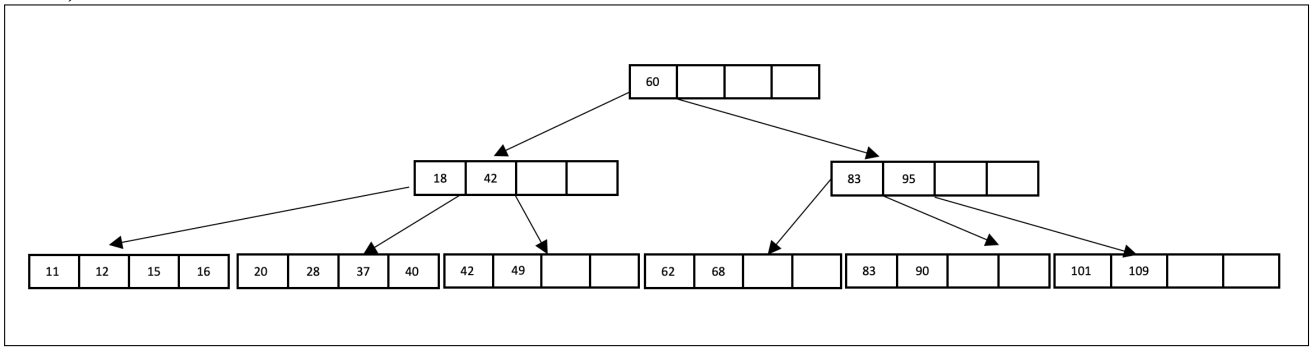
[✓ **4**] ☐ None

**Solution:** We would first read to fetch the root, and the first internal node, then we can either:
Start from the leftmost leaf and continue fetching leaf nodes until you hit a value bigger than 42.
Start from the leftmost leaf and continue fetching leaf nodes until you reach the leaf which page id is equal to the page id of the page before 42.

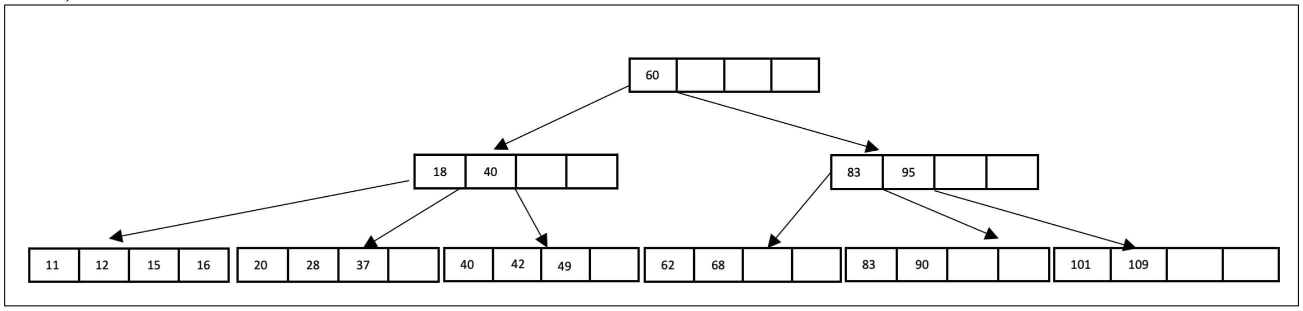
Question 3 continues…

(b) **[5 points]** Insert $40^*$ into the B+tree. Select the resulting tree.

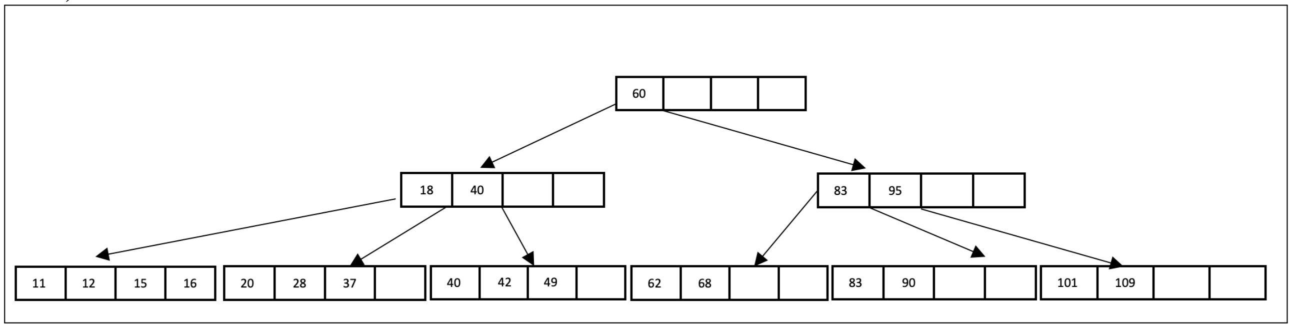
■ **A)**



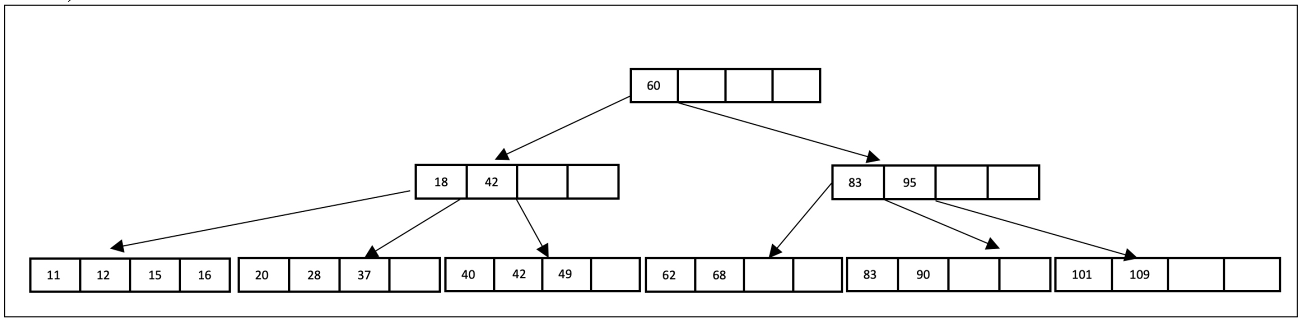
☐ B)



☐ C)



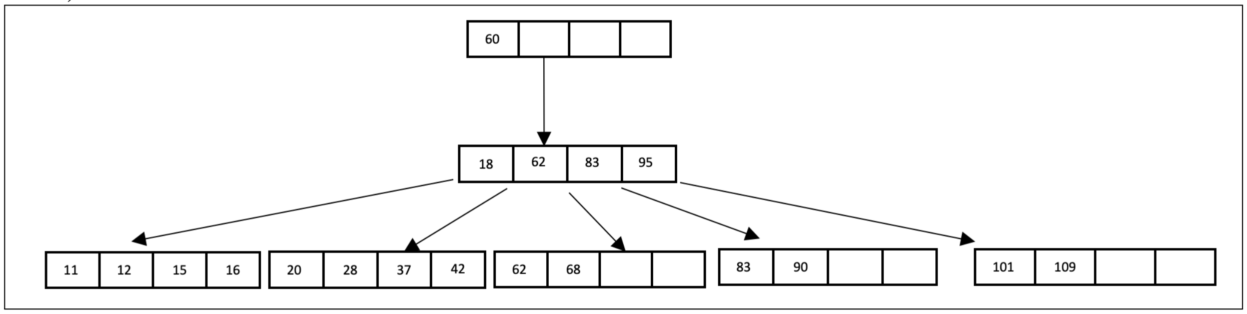
☐ D)



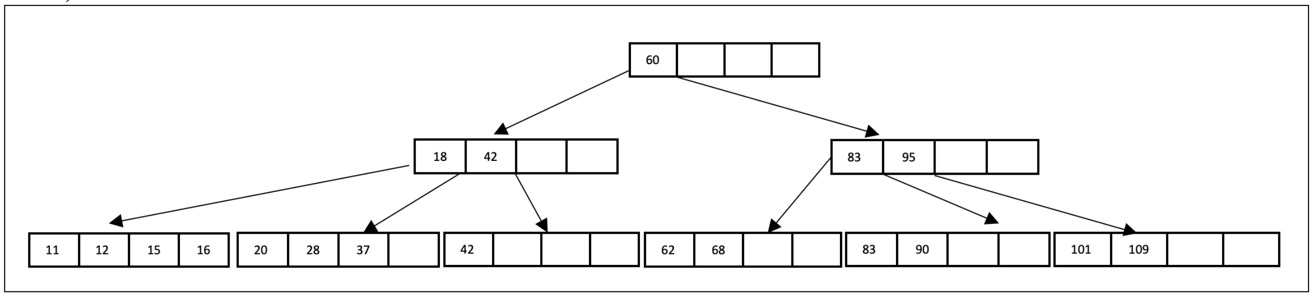
**Solution:** Since the leaf still has space, 40 can be inserted without further action.

(c) **[5 points]** Starting with the intial tree, delete $49^*$. Select the resulting tree.

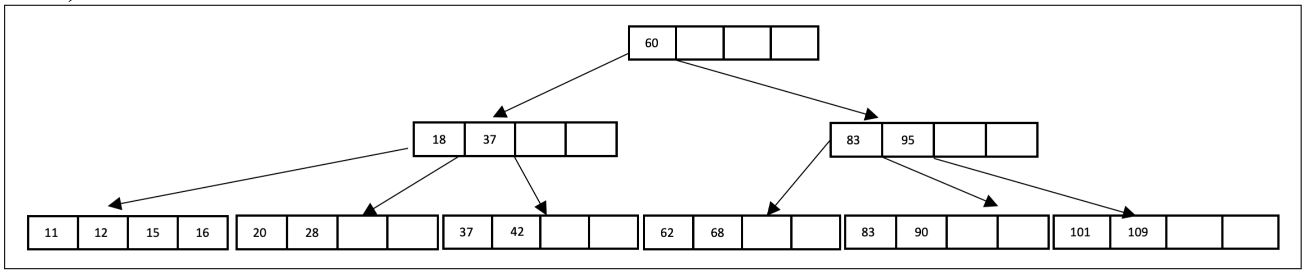
☐ A)



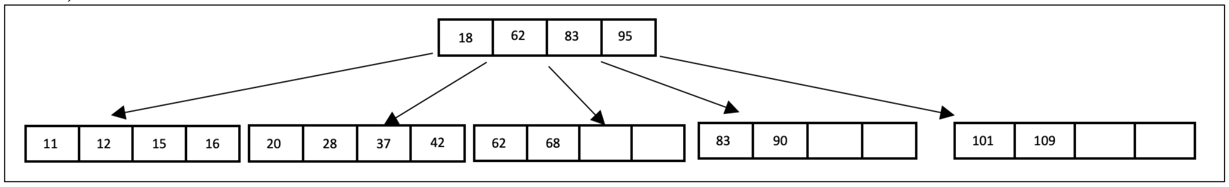
☐ B)



[✓ C)]



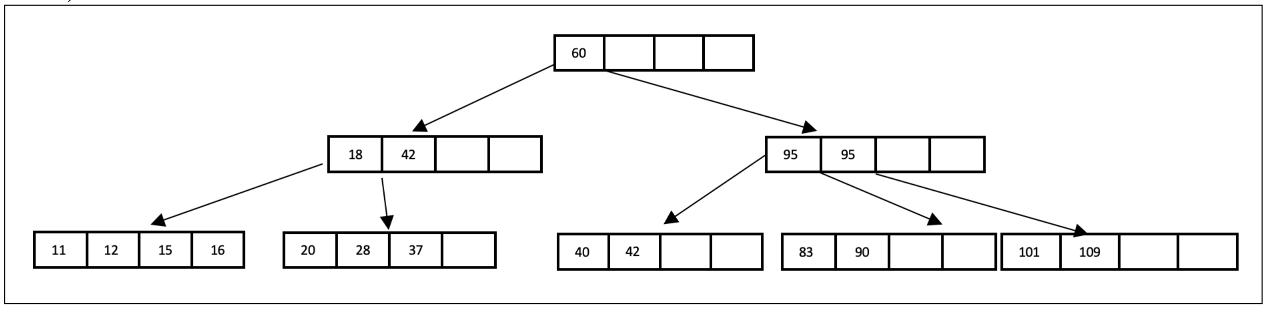
☐ D)



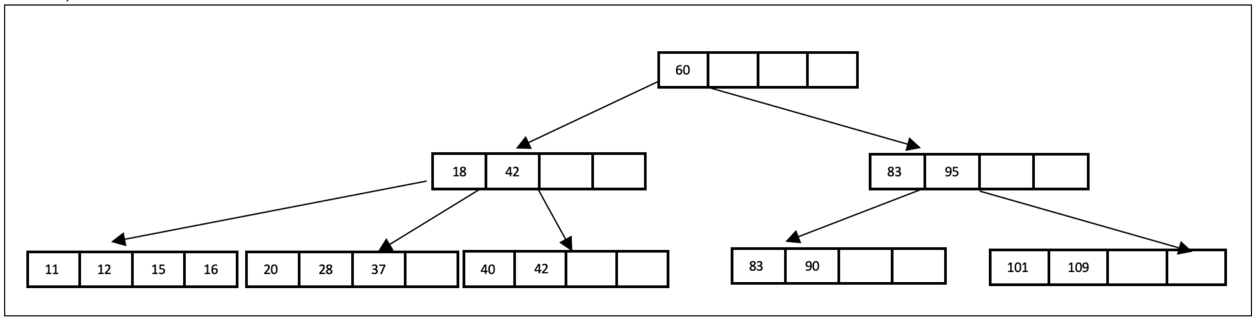
**Solution:** Deleting `49*` causes the leaf node holding it to underflow. The node can borrow from its left neighbor, and 37 is copied up.

(d) **[5 points]** Starting with the intial tree, delete $62^*$ and delete $68^*$. Select the resulting tree.

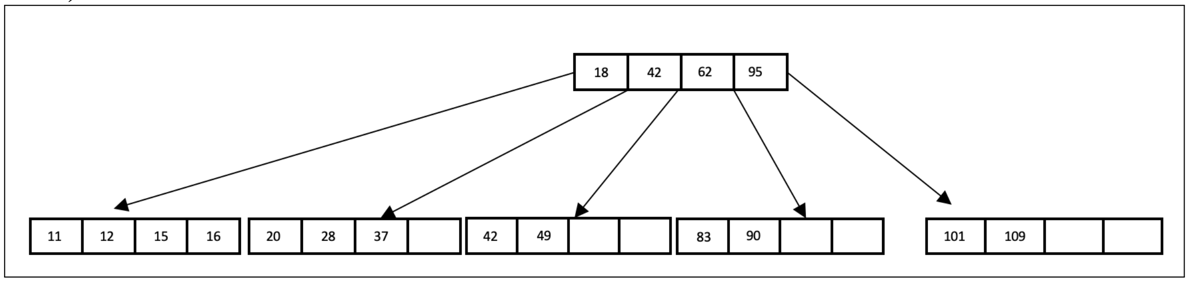
□ A)



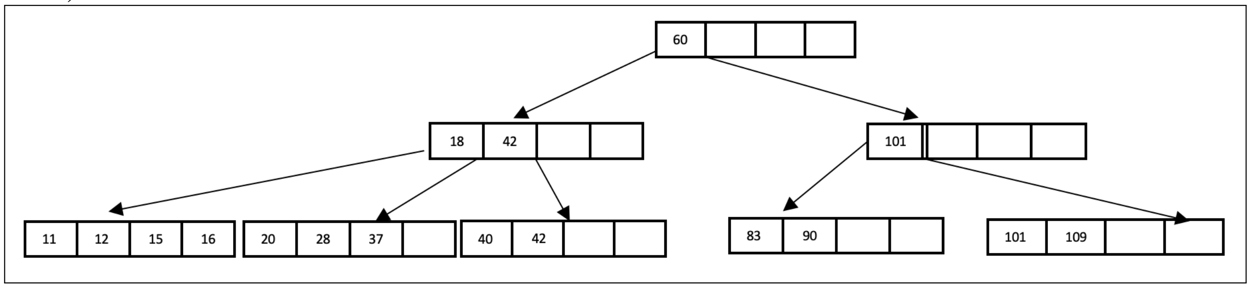
□ B)



[✓ **C)**]



□ D)



**Solution:** Deleting `62*` and `68*` would delete the leaf node holding their parent to be also deleted.

Question 3 continues…

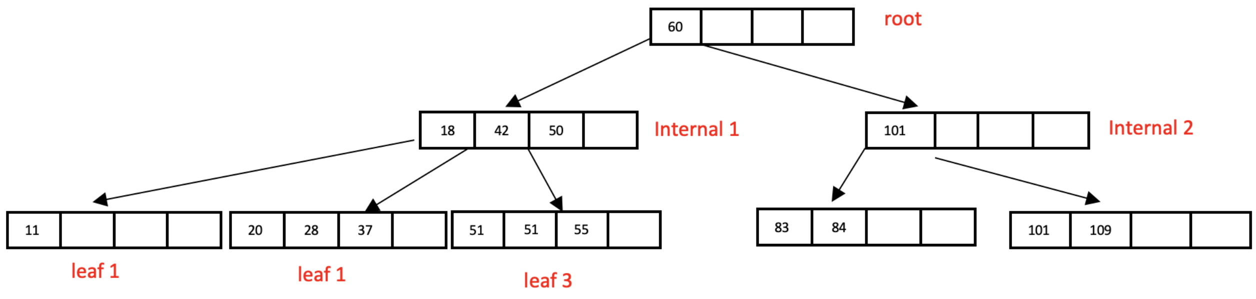


Figure 2: B+tree with violations

(e) The B+Tree shown in Figure 2 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 3**?
☐ Key order property ☐ Half-full property ☐ Balance property
☐ Separator keys [✓ **None**]

ii. **[2 points]** Which properties are violated by **Leaf 1**?
☐ 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 has only 3 children and 42 is smaller than 51.

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

**Solution:** There must be at least 2 keys.

v. **[2 points]** Which properties are violated by **Root**?
☐ Key order property ☐ Half-full property ☐ Balance property
☐ Separator keys [✓ **None**]

Question 3 continues...

(f) i. **[5 points]** A DBMS may potentially use separate buffer pools for a B+Tree's inner node pages and for its leaf node 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

ii. **[5 points]** A read-only thread needs to hold at most two latches at the same time.
[✓ **True**] ☐ False

**Solution:** A reader needs a latch on the parent and child (or two siblings in a leaf node scan) at most.

iii. **[5 points]** A write thread needs to hold at most three write latches at the same time
☐ True [✓ **False**]

**Solution:** A thread may need to hold an arbitrarily large number of write latches, for every node from the root to a leaf (plus potentially siblings of nodes along the path), in circumstances where the write might cause a rebalancing of the root

iv. **[5 points]** A write thread might hold only one write latch (Consider a tree that has more than one leaf node)
[✓ **True**] ☐ False

**Solution:** Using the optimistic latch coupling/crabbing scheme that we discussed in class, the thread can grab read latches, and only grab one write latch at the leaf in circumstances where the write doesn't cause any changes to the structure of the tree.

v. **[5 points]** A read thread must release its latches in the order they were acquired (i.e., FIFO) to prevent concurrency errors.
☐ True [✓ **False**]

**Solution:** Threads can release latches in any order, although FIFO is best for performance.

vi. **[5 points]** A write thread must release its latches in the order they were acquired (i.e., FIFO) to prevent concurrency errors.
☐ True [✓ **False**]

**Solution:** Threads can release latches in any order, although FIFO is best for performance.

vii. **[5 points]** The minimum space utilization for a B+ tree index is 50 percent.
☐ True [✓ **False**]

Question 3 continues. . .

**Solution:** By the definition of a B+ tree, each index page, except for the root, has at least d and at most 2d key entries. Therefore—with the exception of the root—the minimum space utilization guaranteed by a B+ tree index is 50 percent, but since the root is part of the tree, the answer is False.