CS202: Data Structures

Spring 2022

Lecture 12

Mobin Javed

Slides adapted from Ihsan Ayyub Qazi

B+ Trees

Why B+ Trees?

- B+ Trees are heavily used in databases and file systems
  - Databases: Sleepycat/BerkeleyDB, MySQL, SQLite
  - File systems: MacOS HFS/HFS+, ReiserFS, Windows NTFS, Linux ext3, shmfs
- They work very well with large datasets

Traversing Large Datasets

- Suppose we had many pieces of data e.g., $n = 2^{30} \approx 10^9$
- In the worst-case, how many hops would be traversed to find a node?
  - BST
  - AVL

Traversing Large Datasets

- Suppose we have n = $2^{30} \approx 10^9$ data items
- In the worst-case, how many hops would be traversed to find a node?
  - BST: $\approx 10^9$
  - AVL: ≈ 30

Memory Considerations

- What does a binary search tree node contain?
  - Pointers to the left child, right child, and the parent
  - Key and data/value
- Suppose each pointer and the key takes up 4 bytes and the data takes up 1 KB (=1024 bytes) of memory
  - How much space (in bytes) does a tree with 10^9 nodes take?
  - How many nodes of the tree can live in a 1 GB of RAM?

Memory Considerations

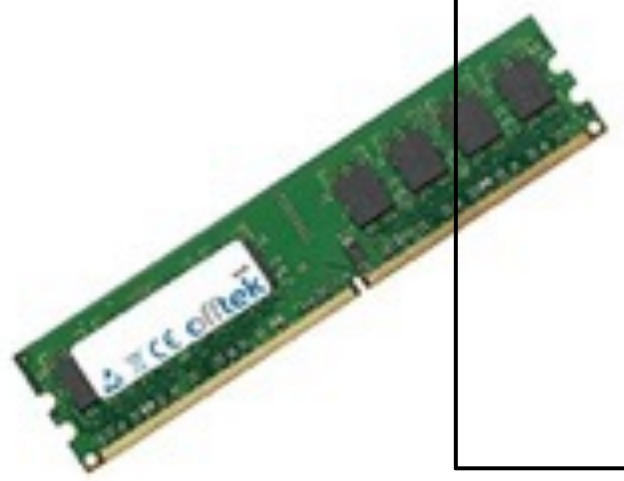
- What does a binary search tree node contain?
  - Pointers to the left child, right child, and the parent
  - Key and data/value
- Suppose each pointer and the key takes up 4 bytes and the data is 1 KB (=1024 bytes)
- How much space (in bytes) does a tree with 10^9 nodes take?
  - Memory taken by each node: (1024 + 4x4) = 1040 bytes
  - Memory used by the tree: $1040 * 10^9 \approx 10^{12}$ bytes
- How many nodes of the tree can live in a 1 GB of RAM?
  - Number of nodes that can fit in a 1 GB RAM: 1 GB/1040 $\approx 10^6$

B+ Tree: A Memory-Conscious Data Structure

- So far, we have taken for granted that memory access in the computer is constant and easily accessible
  - This is not always true!
  - At any given time, some memory might be cheaper and easier to access than others
  - Sometimes the OS provides the program an illusion, and says an object is “in memory” when it’s actually on the disk

Memory Hierarchy

"Every desktop/laptop/server is different but here is a plausible configuration these days"



	Cycles to access	Access latency
CPU Registers	1	0.5ns
L1 Cache: 128KB = 2^{17}	2	1ns
L2 Cache: 2MB = 2^{21}	30	15ns
Main memory: 2GB = 2^{31}	250	125ns
Disk: 1TB = 2^{40}	Random: 30,000,000 Streaming: 5000	15,000,000ns 2500ns

Hardware Constraints

- Back on 32-bit machines, each program had access to 4GB of memory
  - However, this isn't feasible to provide!
  - Sometimes there isn't enough available, and so memory that hasn't been used in a while gets pushed to the disk
- Memory that is frequently accessed goes to the cache, which we know is faster than RAM

Locality (1/2)

- Operating systems (OSes) use temporal and spatial locality to speed up access
- Temporal locality
  - Memory recently accessed is likely to be accessed again
  - Bring recently used data into faster memory
- Spatial locality
  - Nearby memory is likely to be accessed
  - Bring a block of nearby data into faster memory (e.g., running a loop over an array)

Locality (2/2)

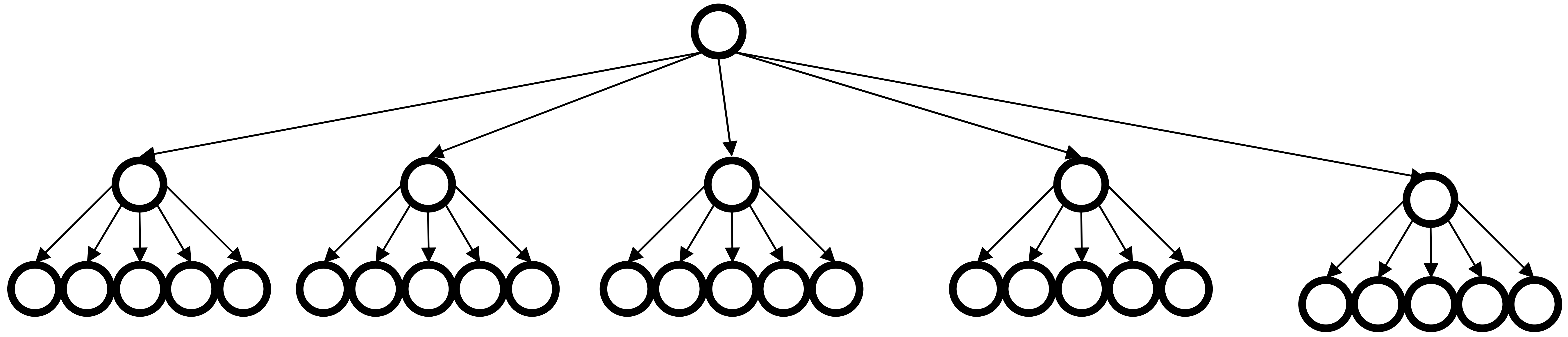
- The OS is always processing this information and deciding which is the best
  - This is why arrays are faster in practice, they are always next to each other in memory
  - Each new node in a BST may not even be in the same page in memory!!

Minimizing Random Disk Accesses

- In the previous example, we considered a good chunk of our data structure lives on the disk i.e., ≈(10^9-10^6) nodes
- Traversing through the tree means lots of random (slow) disk accesses!
- How can we address this problem?

M-ary Search Trees

- Suppose we use a search tree with M children/node
  - We use an array to store children in sorted order
  - Choose M so that it fits into a disk block (1 access for an array)



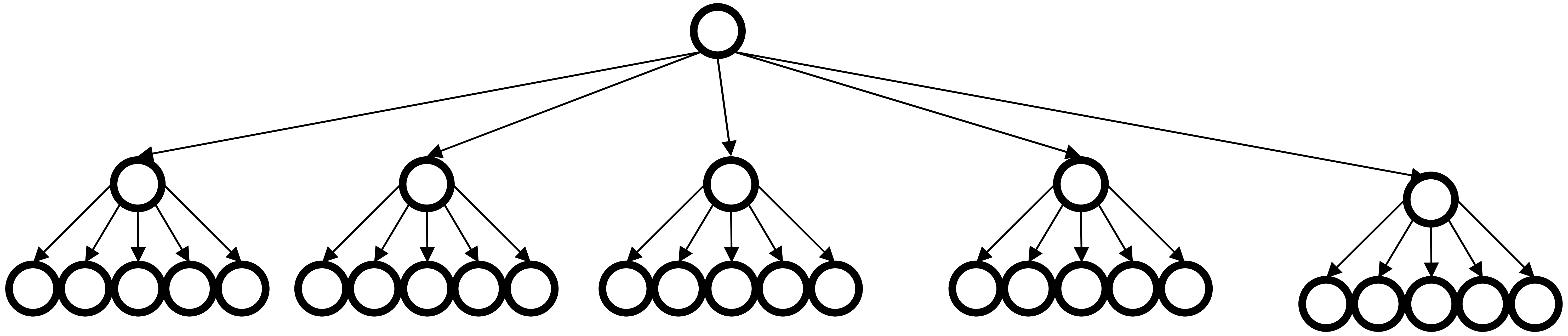
Perfect tree of height **h** has $n=(M^{h+1}-1)/(M-1)$ nodes

Q. What is the height of this tree?

Q. What is the worst-case running time of find?

M-ary Search Trees

- Suppose we use a search tree with M children/node



Q. What is the height of this tree? O(**$\log_M$** n)

Example: M = 256 (=2^8) and n = 2^{40} that's 5 hops instead of 40 hops!

Q. What is the worst-case running time of find?

$O(\log_2 M * \log_M n)$

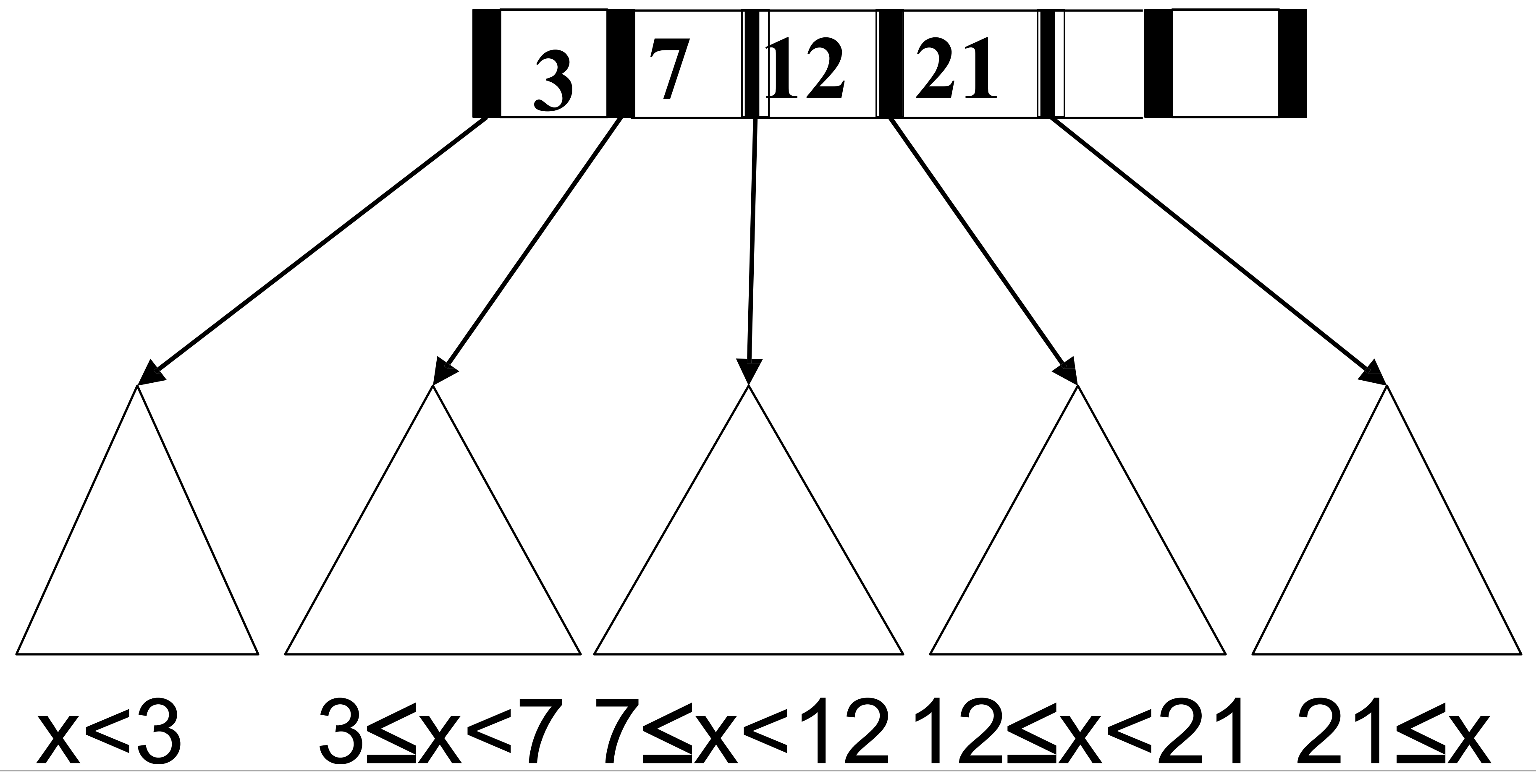
What are the benefits of M-ary Search Trees?

Benefits of M-ary Search Trees

- Smaller height
  - Path length reduces as we increase M
- Potential improvements in the running time of operations
  - Smaller height means potentially smaller number of nodes to traverse
  - Storing of children nodes in an array exploits memory locality (good for caches)
  - Caveats:
    - Time required for performing binary search
- … how can we further improve upon M-ary search trees?
  - Thought: Can we avoid loading data of nodes we may not need?

B+ Trees

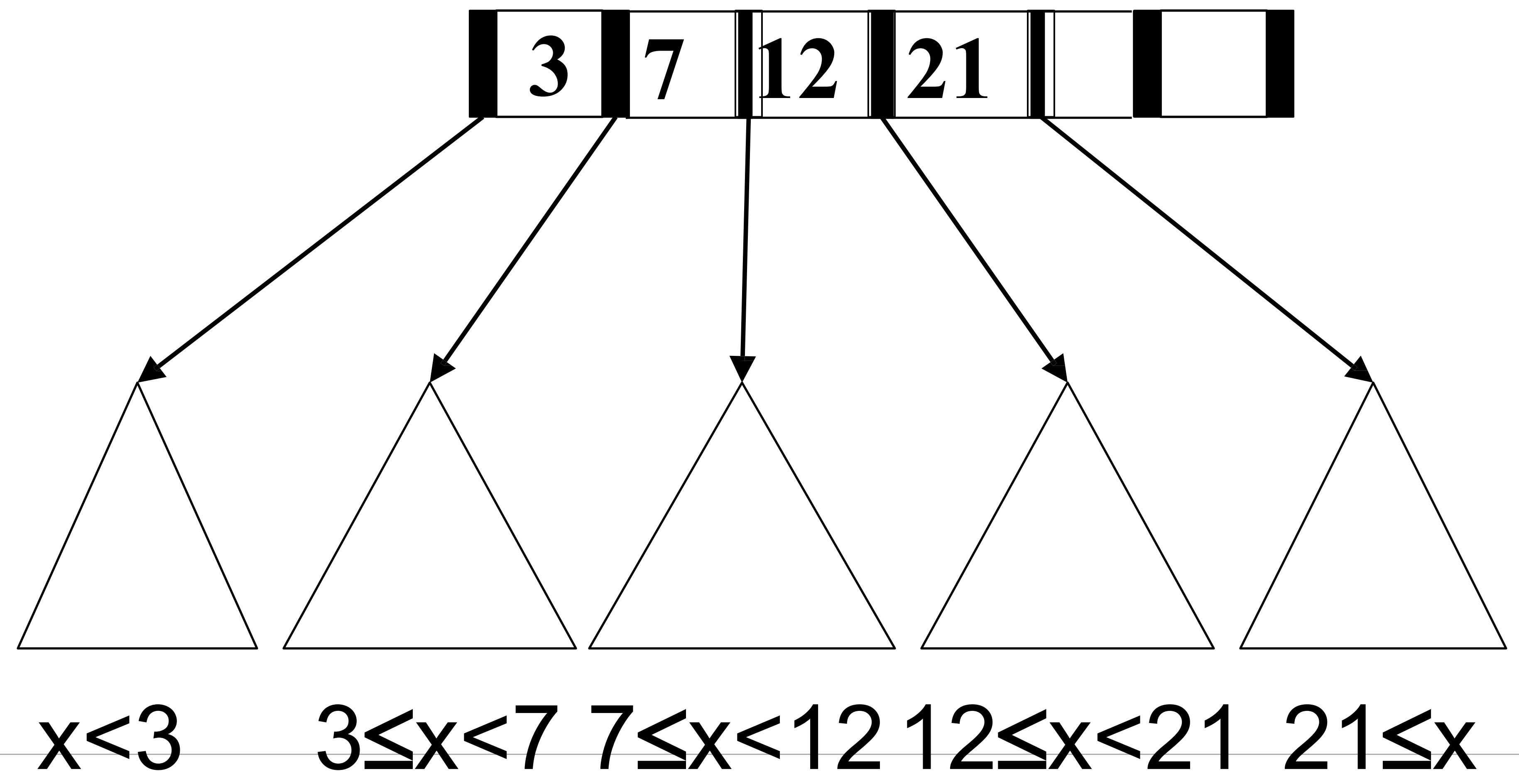
- B+ trees have two types of nodes: internal nodes (signposts) & leaf nodes (i.e., data nodes)
  - Each internal node has room for up to M-1 (sorted) keys & M pointers
  - Each leaf node has room for L key-value pairs, sorted by key
- Note: Creator of the B+ tree must pick M and L!



Remember:
- Leaves store data
- Internal nodes are 'signposts'

Search over B+ Trees

- Different from BST in that we <u>don’t store data in internal nodes</u>
- But find is still an easy root-to-leaf recursive algorithm
  - At each internal node do binary search on (up to) M-1 keys to find the branch to take
  - At the leaf do binary search on (up to) L data items
- But to get logarithmic running time, we need a balance condition…



B+ Tree Structure Properties

- Internal nodes
  - store up to M-1 keys
  - have between $\lceil M/2 \rceil$ and M children
- Leaf nodes
  - store data and all leaf nodes are at the same depth
  - contain between $\lceil L/2 \rceil$ and L data items, stored in sorted order
- Root (special case)
  - has between 2 and M children (or root could be a leaf)

Why half full? It ensures that the tree stays balanced

Why can M be equal to 2 in case of the root?
If n is relatively small compared to M and L,
it may not be possible for the root to be half-full

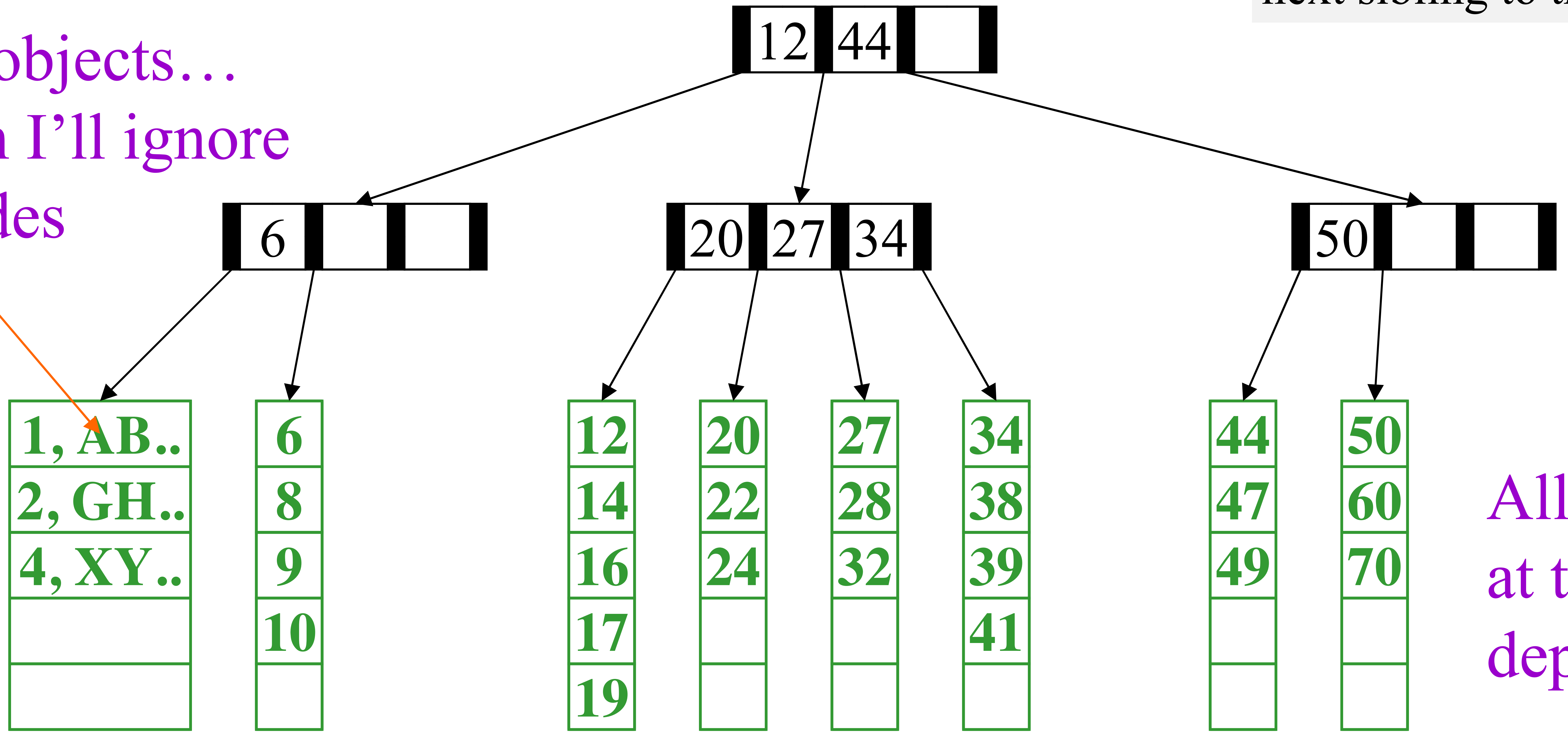
B+ Tree Example

B+ Tree with $M = 4$ (# pointers in internal node)

and $L = 5$ (# data items in leaf)

Definition for later: "neighbor" is the next sibling to the left or right.

Data objects… which I'll ignore in slides



All leaves at the same depth

What is the Worst-Case Complexity for Search?

- Find the correct subnode at every signpost
  - $O(\log_2 M)$
- Go through the depth of the tree
  - $O(\log_M N)$
- Find the object in the leaf
  - $O(\log_2 L)$
- Total find = $O(\log_2 L + \log_2 M * \log_M N)$

Disk Friendliness

- What makes B+ trees disk-friendly?

- Many keys stored in a node
  - All brought to memory/cache in one disk access

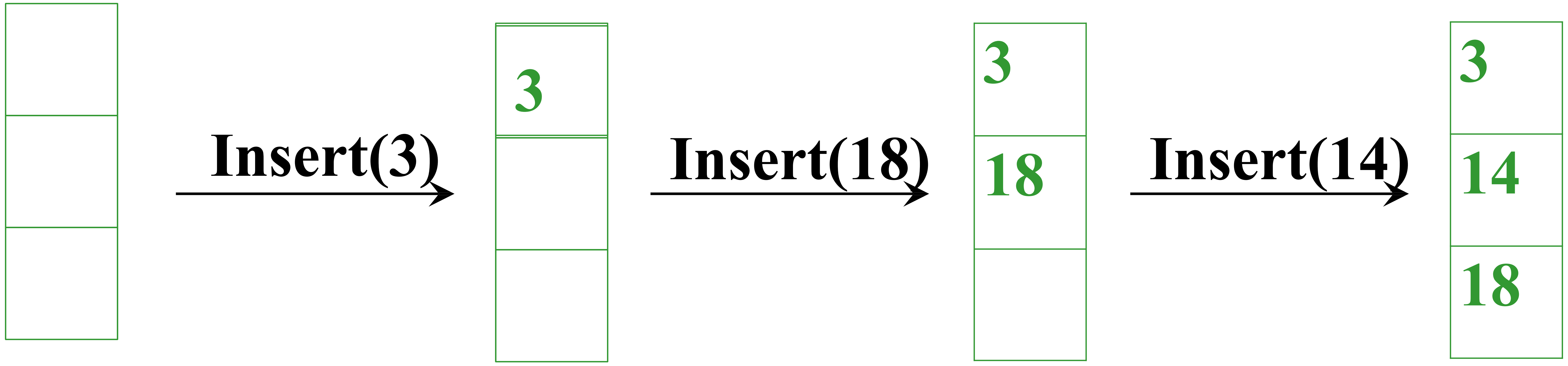
- Internal nodes contain only keys
  - Much of tree structure can be loaded into memory irrespective of data object size
  - Data resides in disk!

B+ Tree Insertions

Insertion: Basic Idea

- Insert into the correct leaf (in sorted order)
- If the leaf overflows
  - split into two
  - attach new child to parent
  - add new key to parent
- Recursively overflow as necessary
- If the root overflows, make a new root

Building a B+ Tree with Insertions

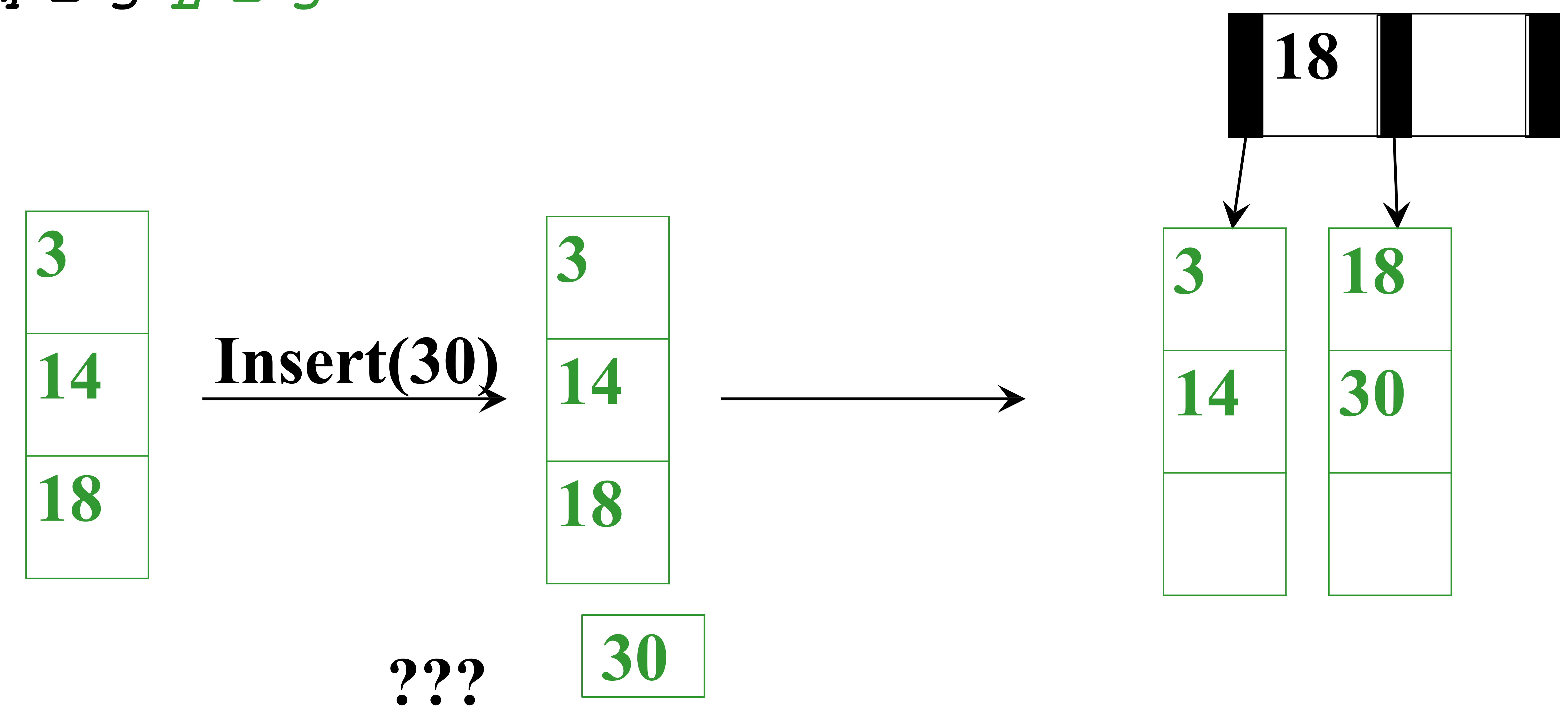


The empty B+ tree (the root will be a leaf at the beginning)

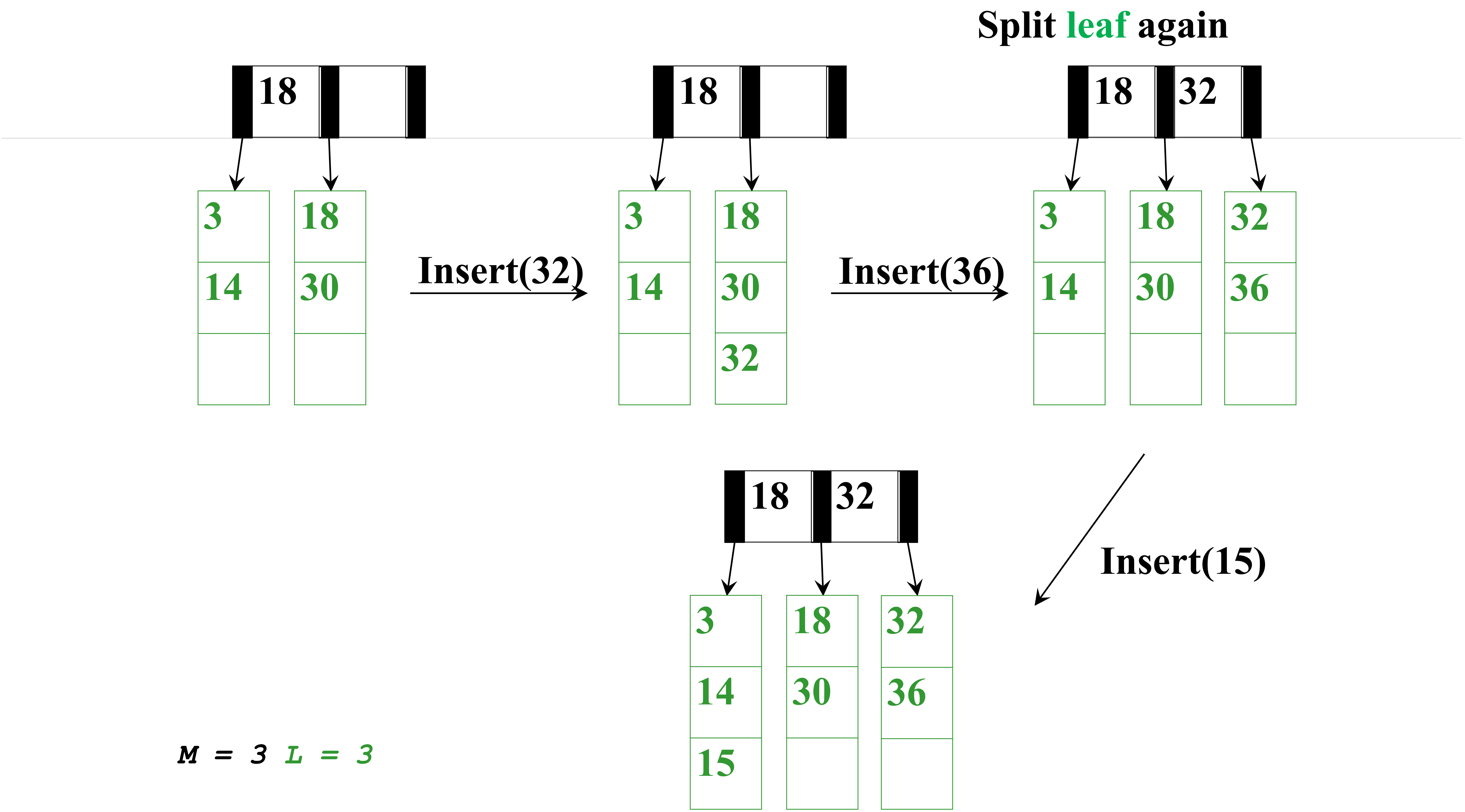
Just need to keep data in order

$M = 3$ $L = 3$

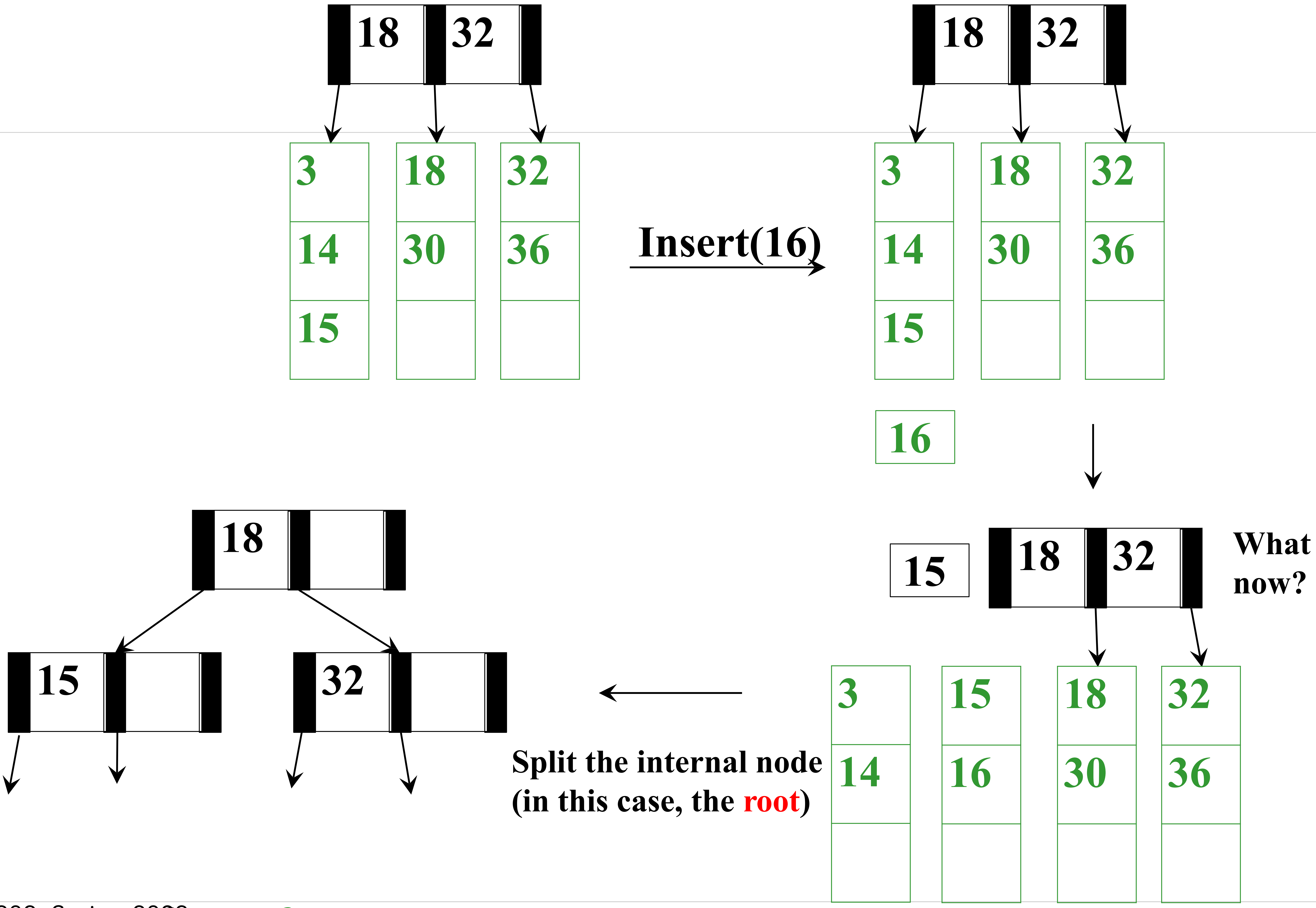
$M = 3$ $L = 3$



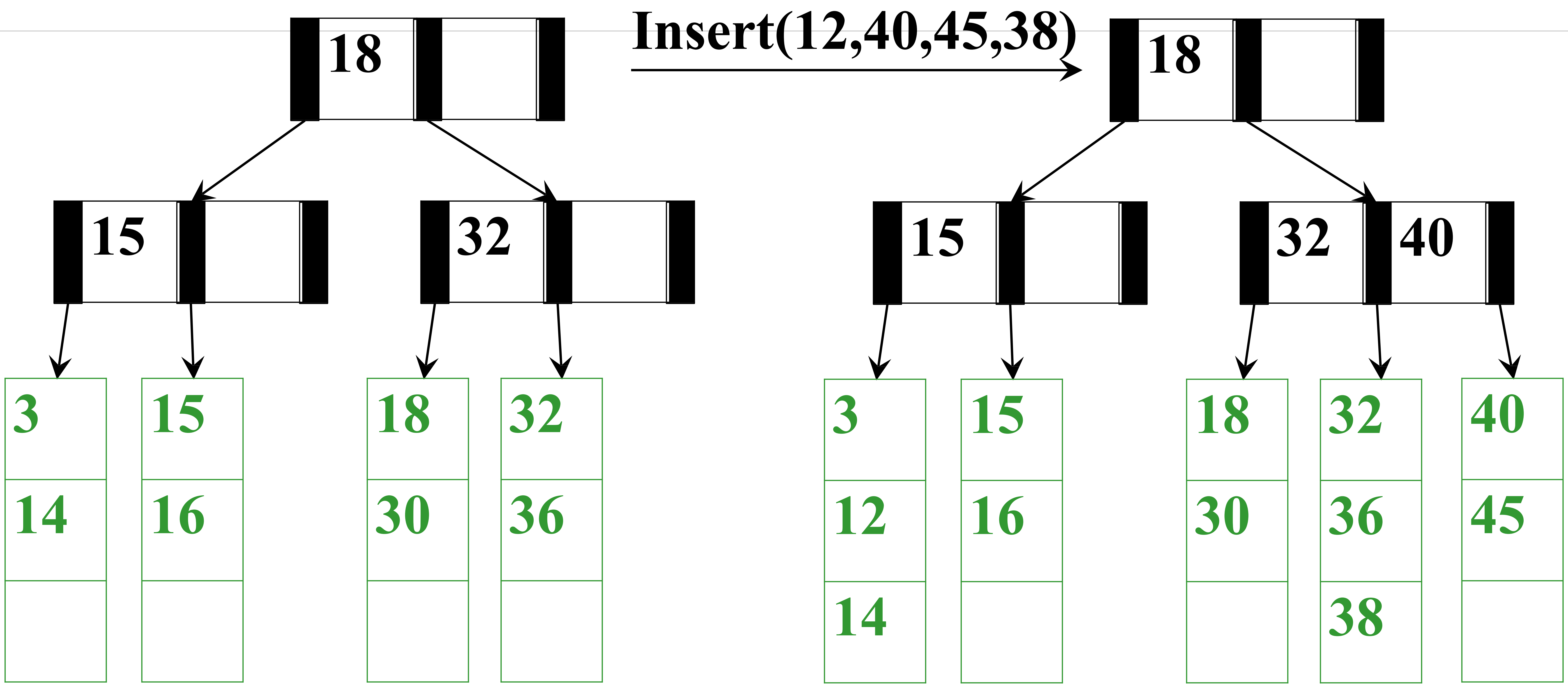
- When we 'overflow' a leaf, we split it into 2 leaves
- Parent gains another child but if there is no parent (like here), we create one; how do we pick the parent key?
  - Smallest key in the right tree (note that all keys < 18 are in the left subtree)



Split leaf again
18
3
14
18
30
Insert(32)
18
3
14
18
30
32
Insert(36)
18
32
3
14
18
30
32
36
Insert(15)
18
32
3
14
15
18
30
32
36
M = 3 L = 3


18
32
3
14
15
18
30
32
36
Insert(16)
18
32
3
14
15
18
30
32
36
16
15
18
32
What now?
3
14
15
16
18
30
32
36
Split the internal node
(in this case, the root)
18
15
32

$M = 3$ $L = 3$



***M* = 3** ***L* = 3**

Note: Given the leaves and the structure of the tree, we can always fill in internal node keys;
'the smallest value in my right branch'

Insertion Algorithm

1. Insert the key in its leaf in sorted order
2. If the leaf ends up with L+1 items, **overflow**!
   - Split the leaf into two nodes:
     - original with $\lceil (L+1)/2 \rceil$ smaller keys
     - new one with $\lfloor (L+1)/2 \rfloor$ larger keys
   - Add the new child to the parent
   - If the parent ends up with $M+1$ children, **overflow**!

This makes the tree deeper!

3. If an internal node ends up with M+1 children, **overflow**!
   - Split the node into two nodes:
     - original with $\lceil (M+1)/2 \rceil$ children with smaller keys
     - new one with $\lfloor (M+1)/2 \rfloor$ children with larger keys
   - Add the new child to the parent
   - If the parent ends up with $M+1$ items, **overflow**!
4. Split an overflowed root in two and hang the new nodes under a new root
5. Propagate keys up tree.

Efficiency of Insert

- Find correct leaf: $O(\log_2 M \log_M n)$ [binary search on each node along the path]
- Insert in leaf: $O(\log_2 L + L)$ [binary search + move elements by one spot]
- Split leaf: O(L) [requires creating a new leaf node – O(L/2) keys + initialization]
- Split parents all the way up to root: $O(M \log_M n)$ [splitting may be needed all the way up to the root + per-split requires two new nodes with O(M) children nodes – initialization]

Total: $O(L + M \log_M n)$

But it's not that bad:

- Splits are not that common (only required when a node is FULL, M and L are likely to be large, and after a split, will be half empty)
- Splitting the root is extremely rare
- Remember disk accesses are key: $O(\log_M n)$

B-Tree Reminder: Another dictionary

- Before we talk about deletion, just keep in mind the overall idea:
  - Large data sets won't fit entirely in memory and disk access is slow
  - Set up tree so we do (at most) one disk access per node in tree
  - Then our goal is to keep tree shallow as possible
  - Balanced BSTs are a good start, but we can do better than $\log_2 n$
  - In an M-ary tree, height drops to $\log_M n$
    - Why not set M really high? Height 1 tree…
    - Instead, set M so that each node fits in a disk block

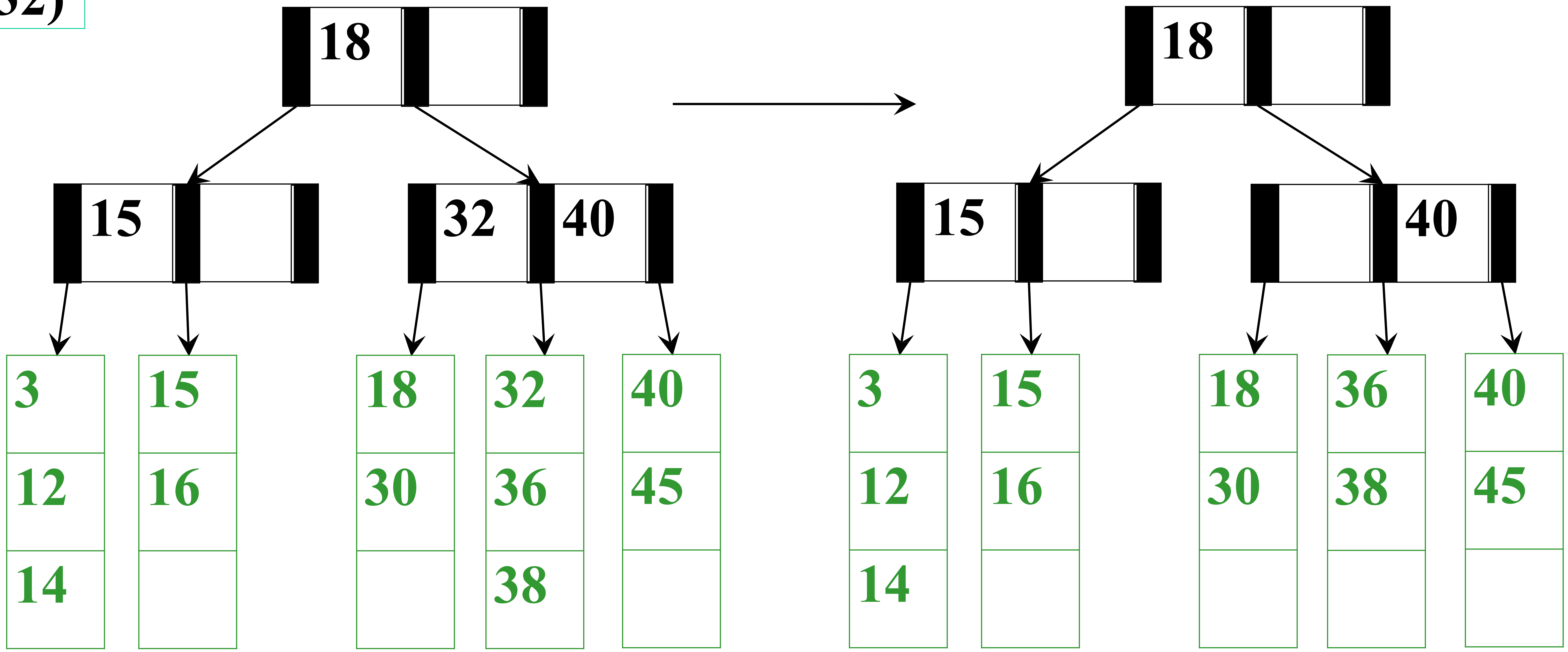
B+ Tree Deletion

Deletion: Basic Idea

- Remove the data from the correct leaf
- If the leaf has too few elements,
  - Adopt one from a neighbor (if it doesn't result in an underflow)
  - Otherwise, merge with the neighbor
- Recursively underflow up to root if necessary

And Now for Deletion…

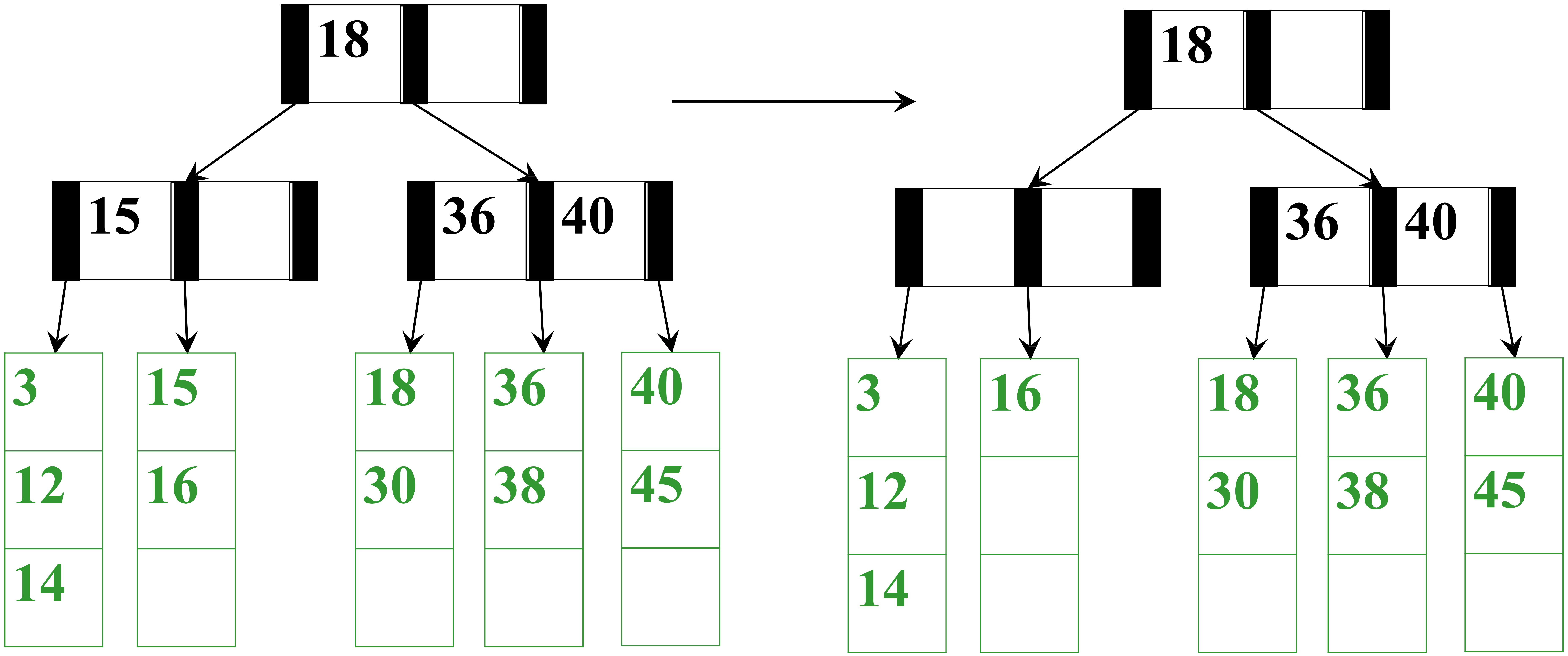
Delete(32)



Easy case: Leaf still has enough data; just remove

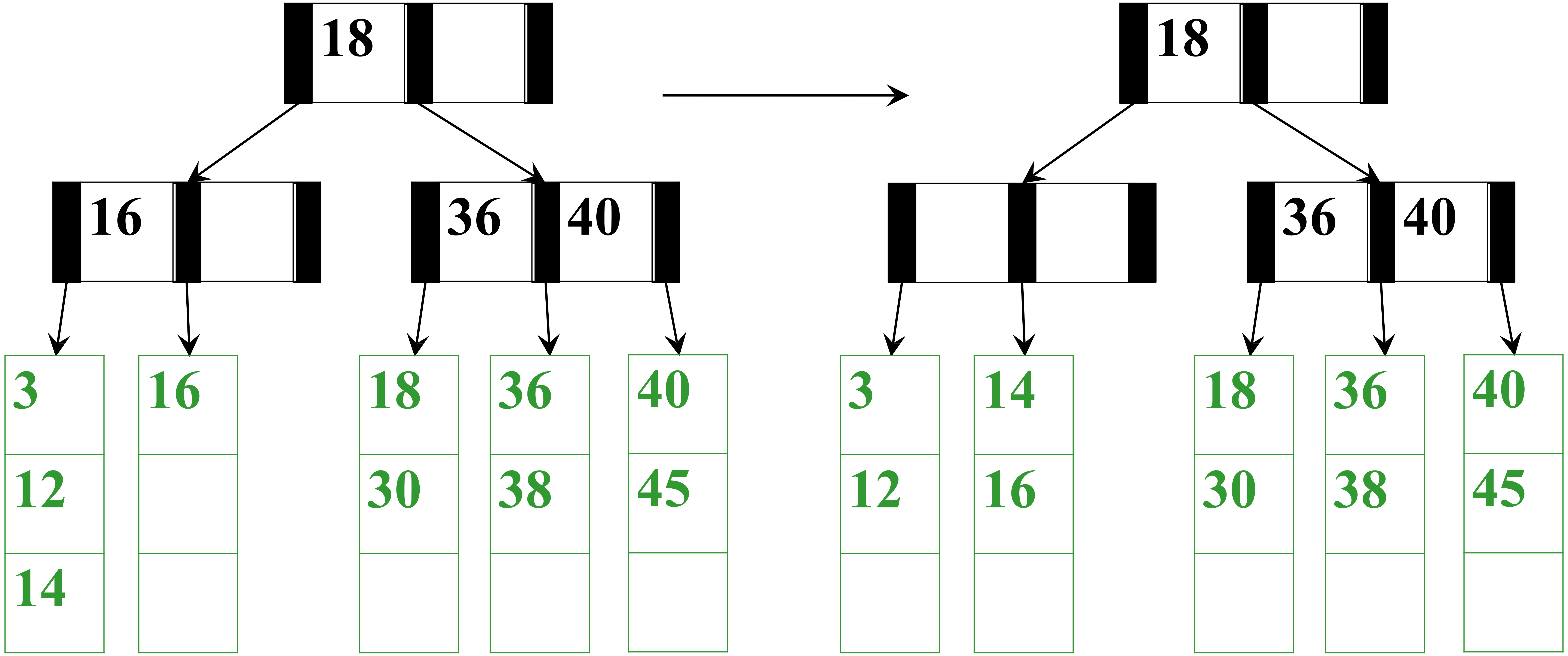
***M* = 3 *L* = 3**

Delete(15)



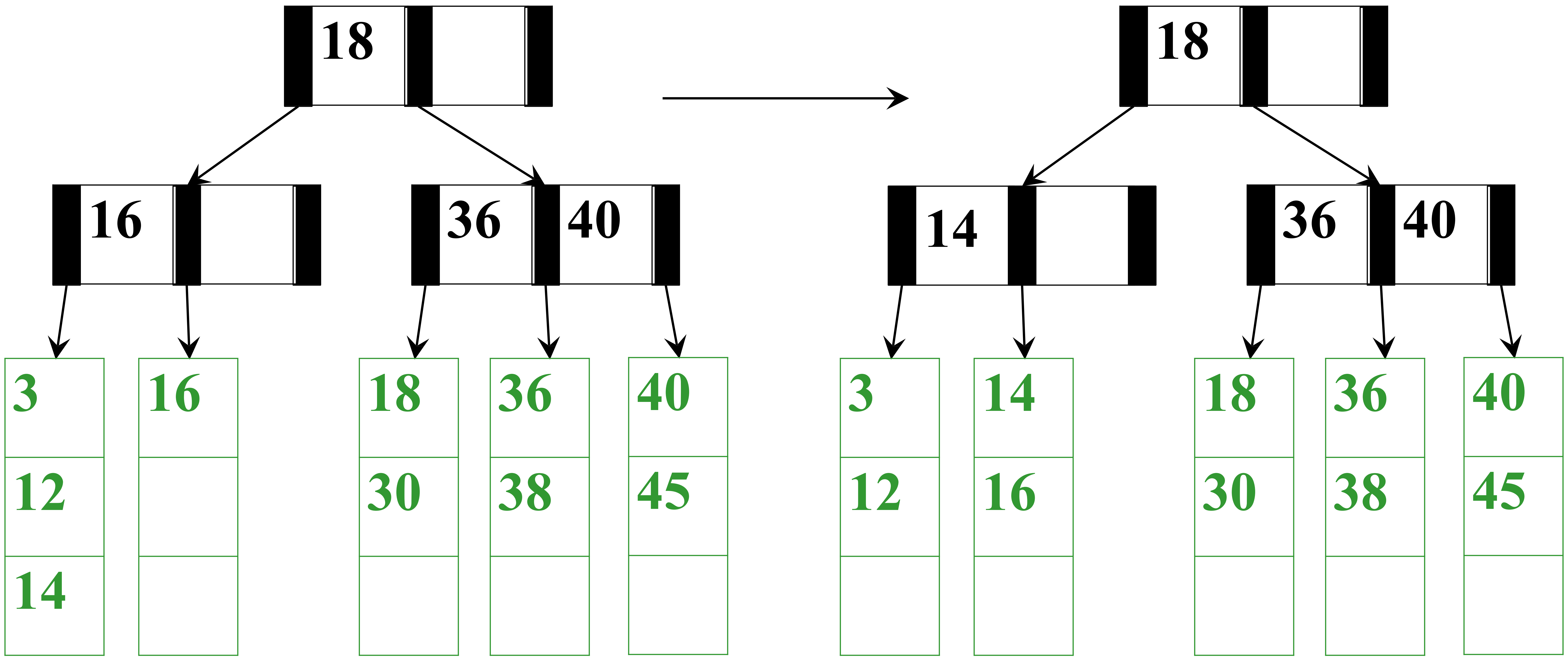
Is there a problem?

$M = 3$ $L = 3$



$M = 3$ $L = 3$

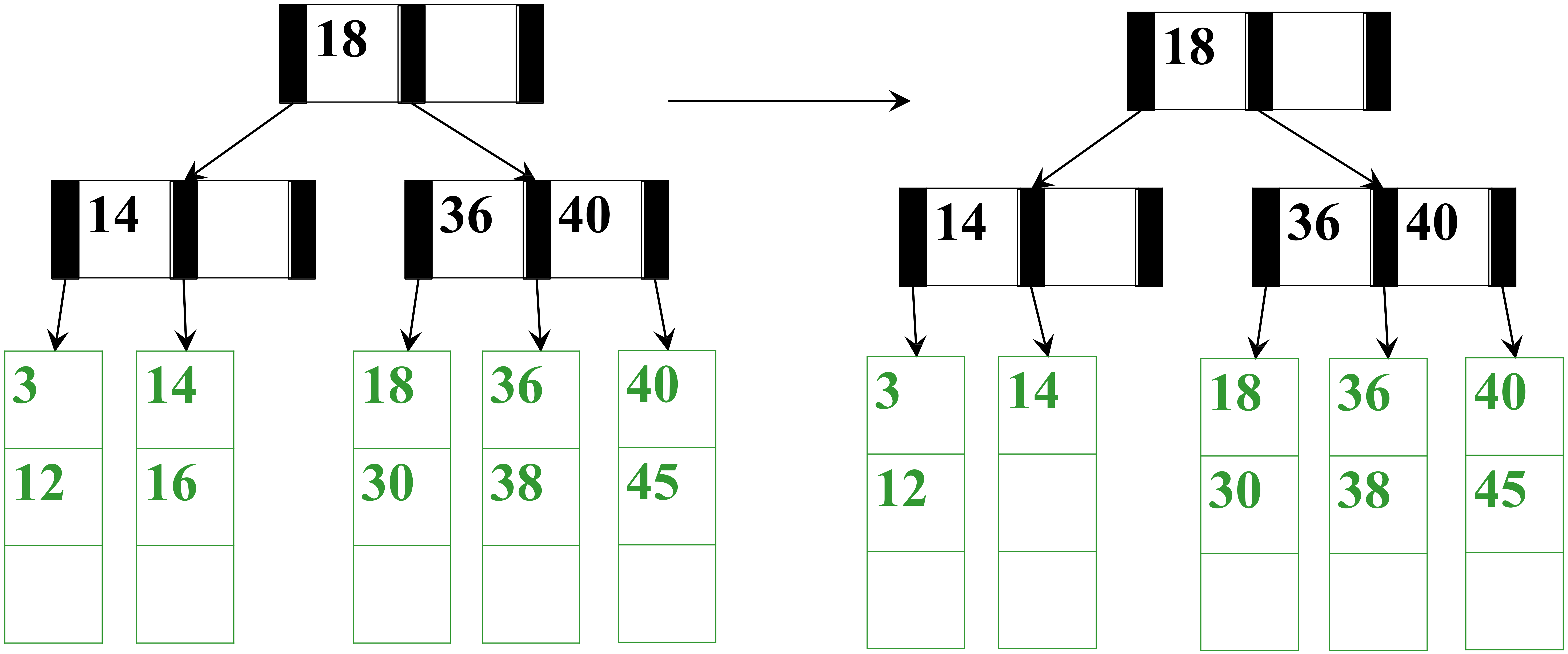
Adopt from neighbor!



$M = 3$ $L = 3$

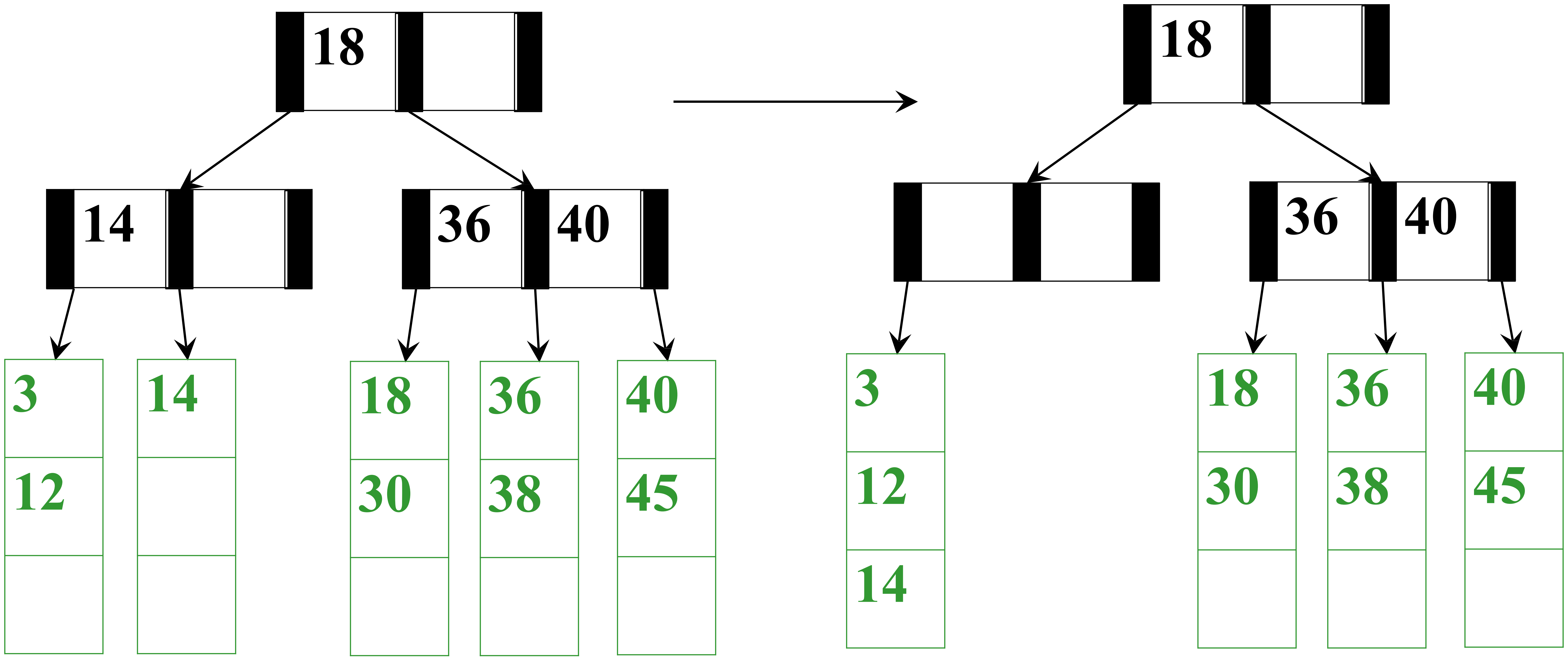
Adopt from neighbor!

Delete(16)



Is there a problem?

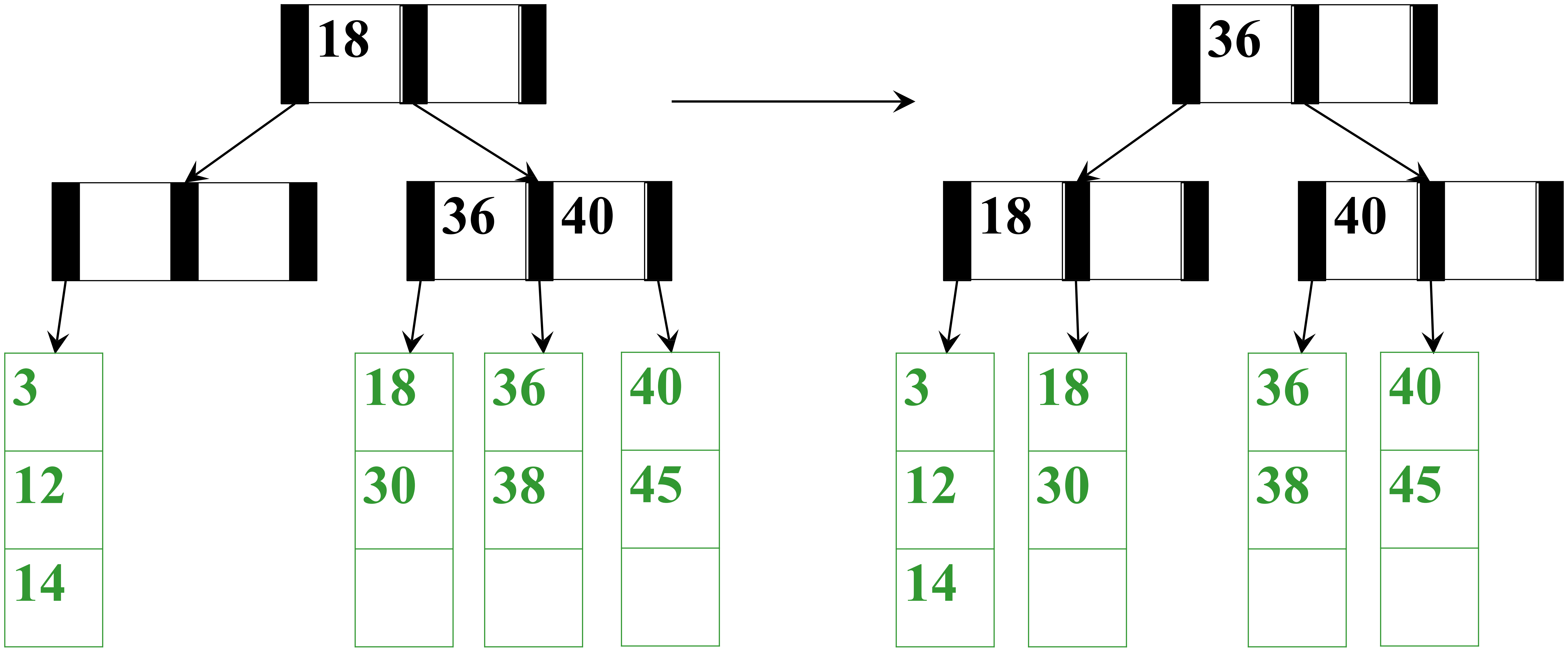
$M = 3$ $L = 3$



$M = 3$ $L = 3$

Merge with neighbor!

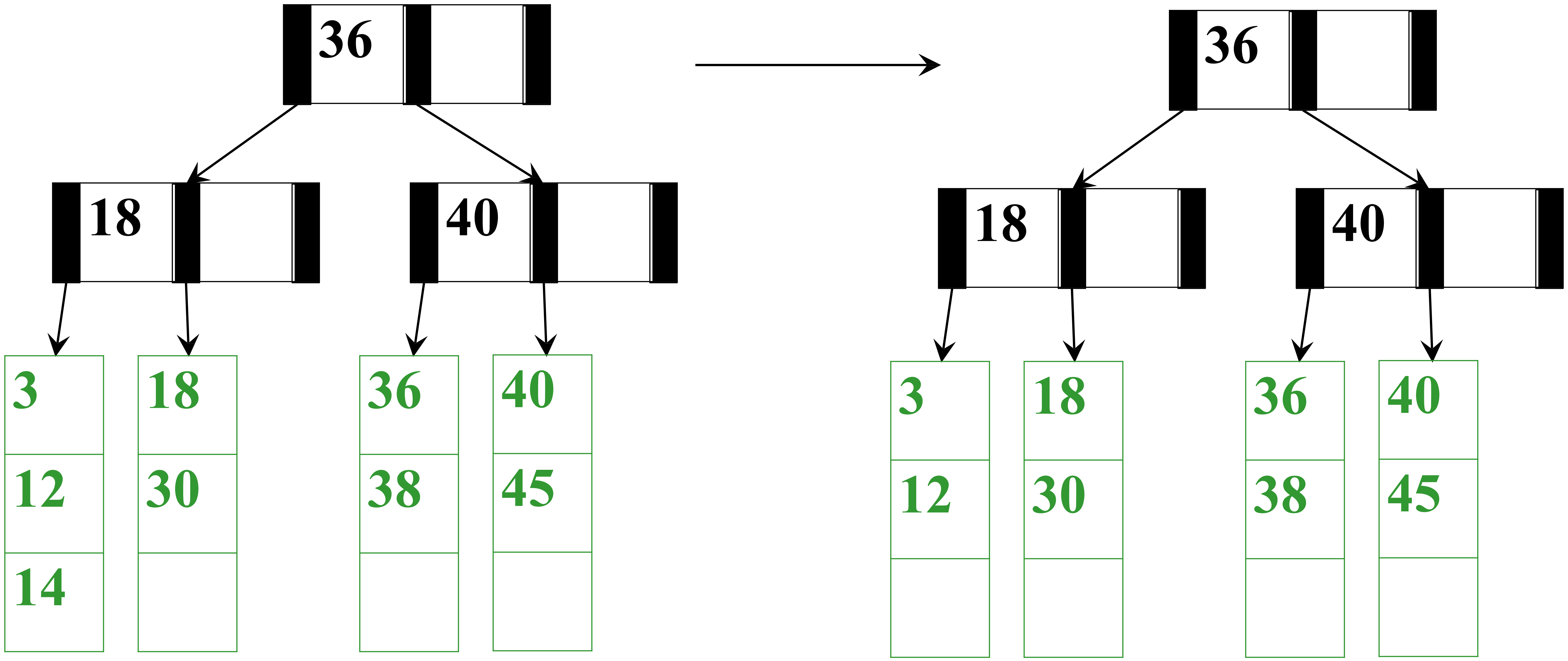
But hey, Is there a problem?



$M = 3$ $L = 3$

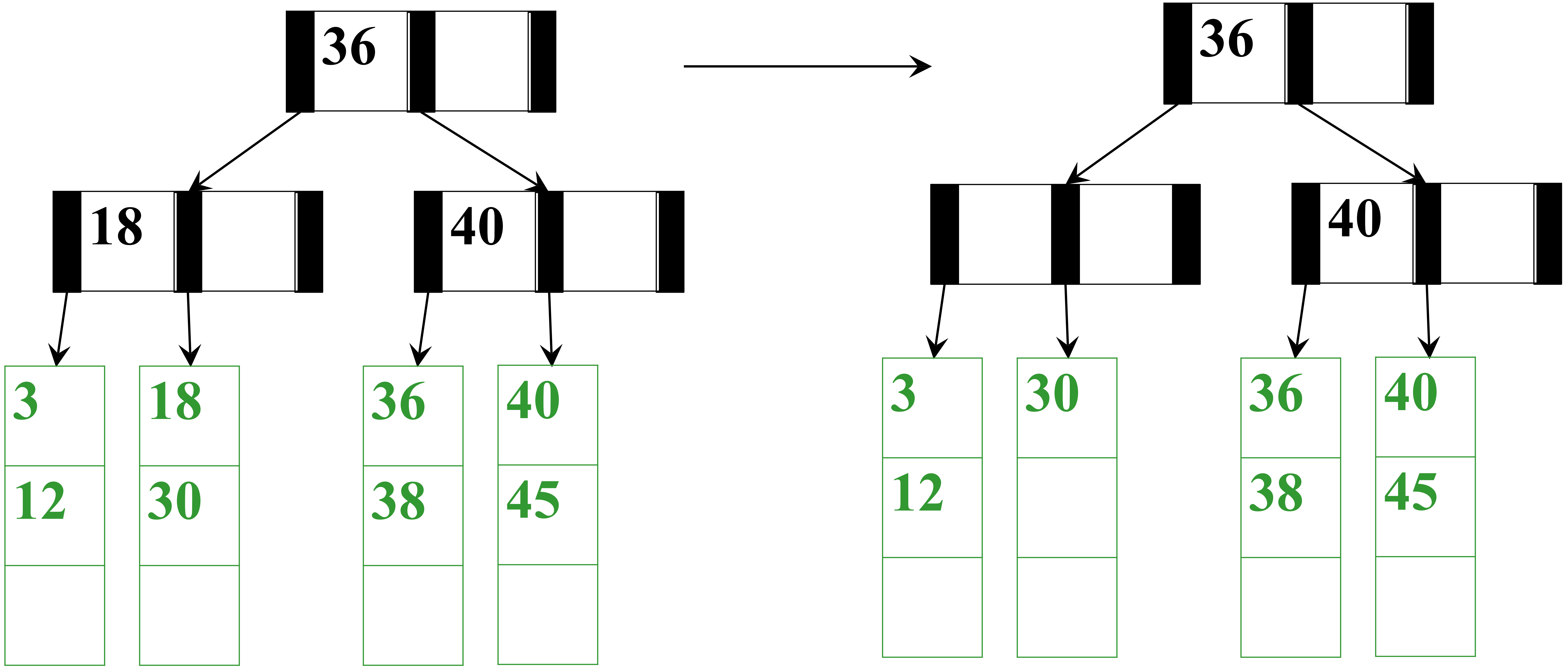
Adopt from neighbor!

Delete(14)



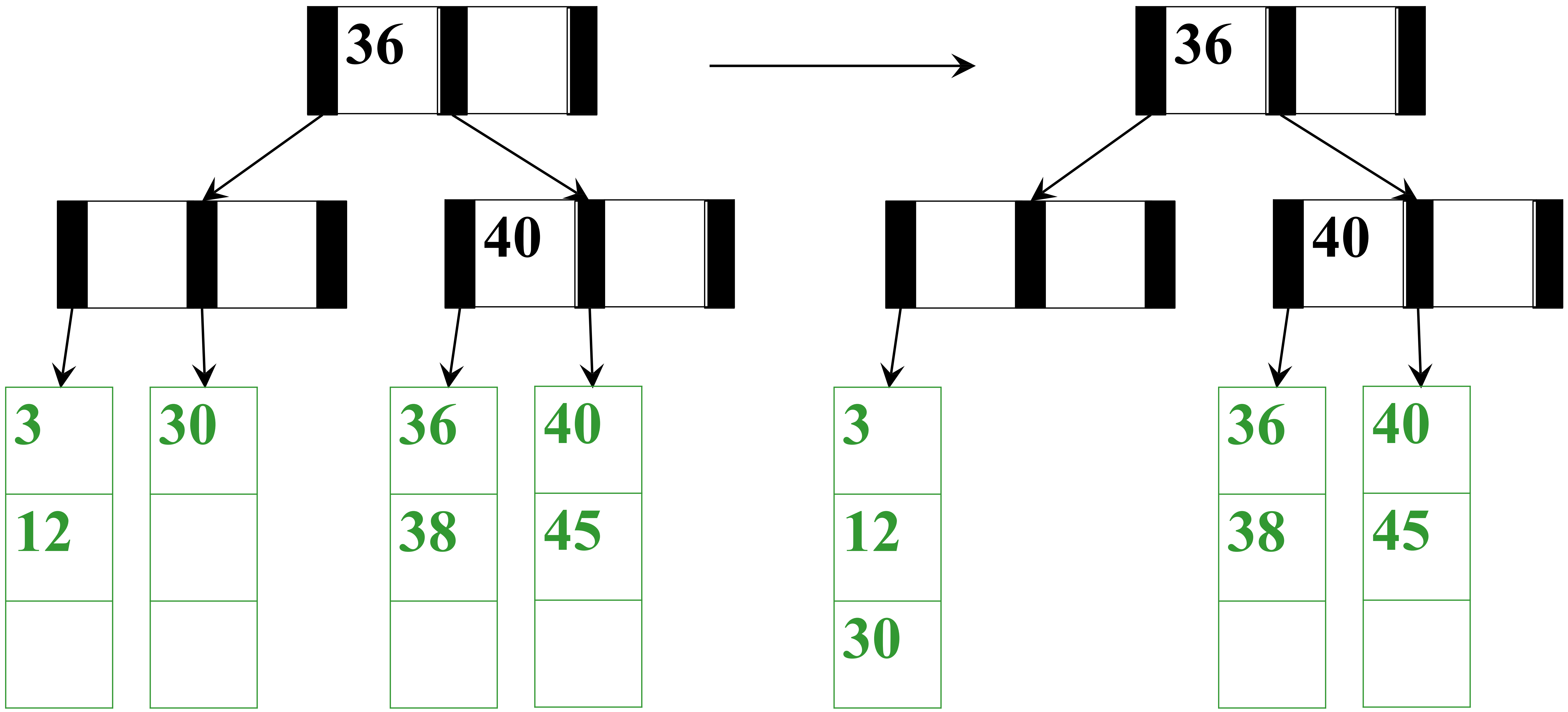
$M = 3$ $L = 3$

Delete(18)



Is there a problem?

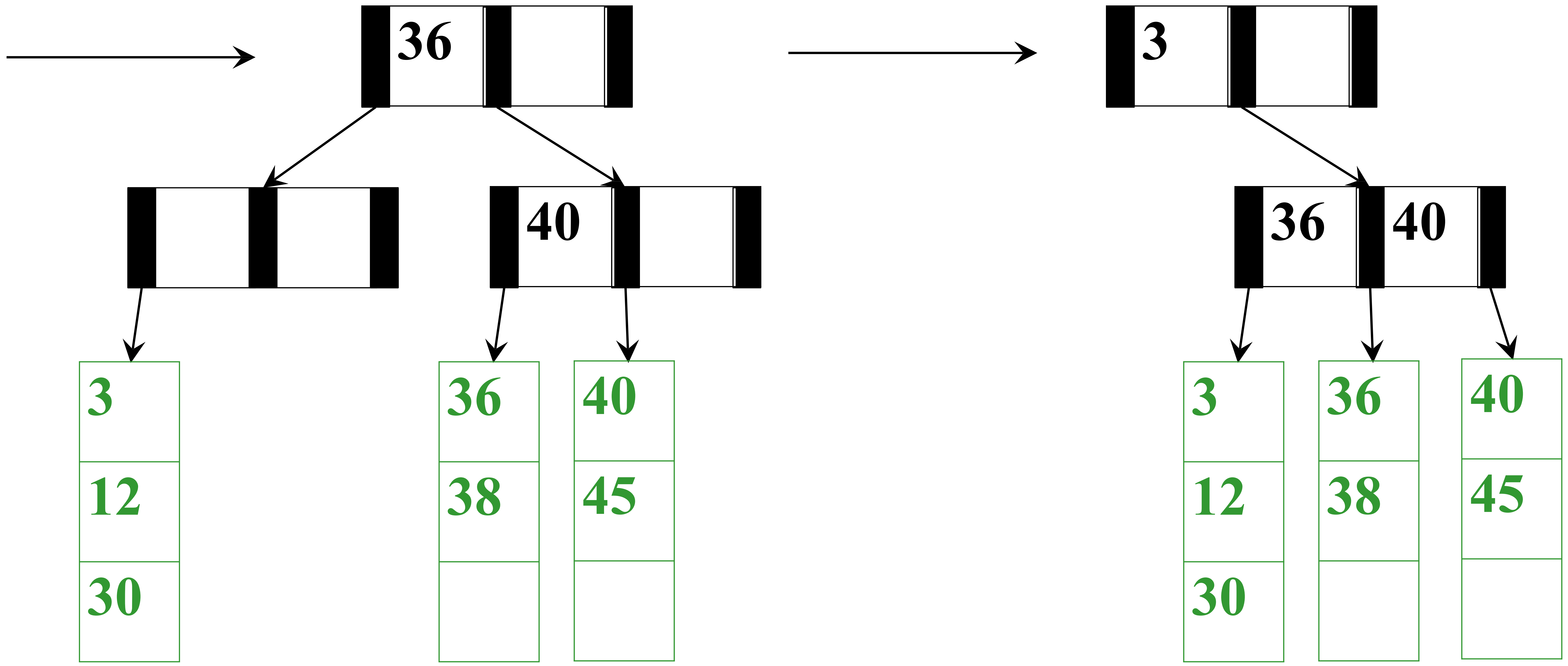
M = 3 *L = 3*



$M = 3$ $L = 3$

Merge with neighbor!

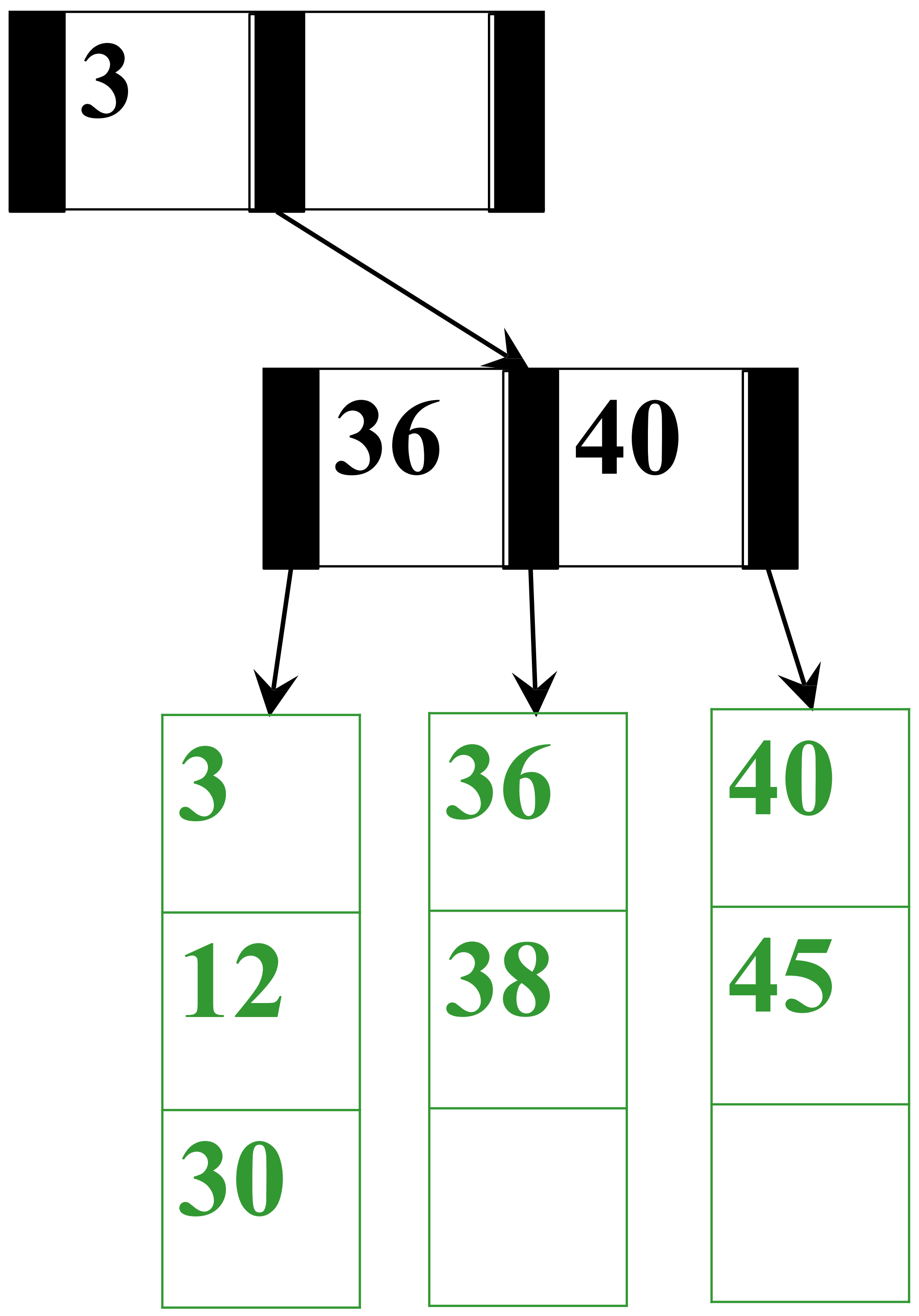
But hey, Is there a problem?

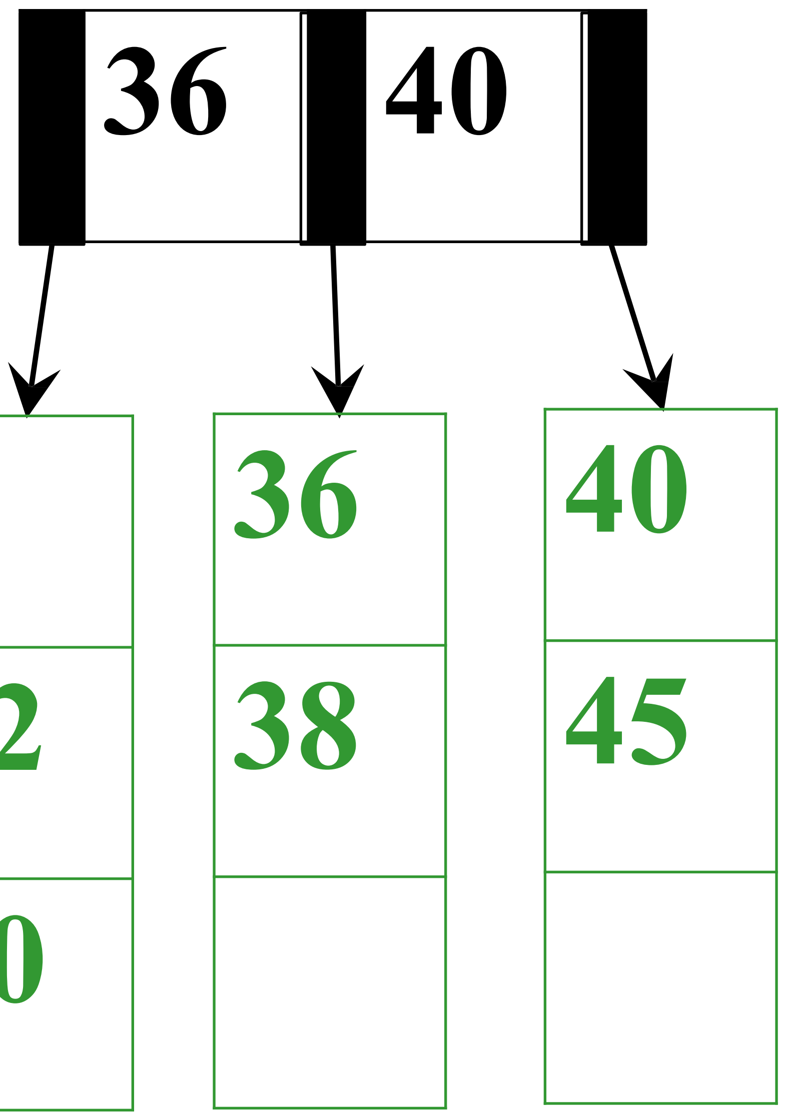


$M = 3$ $L = 3$

Merge with neighbor!

But hey, Is there a problem?





Pull out the root!

M = 3 L = 3

Deletion Algorithm, part 1

1. Remove the data from its leaf

2. If the leaf now has $\lceil \mathtt{L/2} \rceil - 1$, *underflow!*
   - If a neighbor has > $\lceil \mathtt{L/2} \rceil$ items, *adopt* and update parent
   - Else *merge* node with neighbor
     - Guaranteed to have a legal number of items
     - Parent now has one less node

3. If step (2) caused the parent to have $\lceil \mathtt{M/2} \rceil - 1$ children, *underflow!*
   - …

Deletion Algorithm (continued)

3. If an internal node has $\lceil M/2 \rceil - 1$ children
   - If a neighbor has > $\lceil M/2 \rceil$ items, *adopt* and update parent
   - Else *merge* node with neighbor
     - Guaranteed to have a legal number of items
     - Parent now has one less node, may need to continue up the tree

If we merge all the way up through the root, that's fine unless the root went from 2 children to 1

- In that case, delete the root and make child the root
- This is the only case that decreases tree height

Worst-Case Efficiency of Delete

- Find correct leaf: $O(\log_2 M \log_M n)$
- Remove from leaf: $O(L)$
- Adopt from or merge with neighbor: $O(L)$
- Adopt or merge all the way up to root: $O(M \log_M n)$

Total: $O(L + M \log_M n)$

But it's not that bad:

- Merges are not that common
- Disk accesses are the name of the game: $O(\log_M n)$

Insert vs Delete Comparison

Insert

- Find correct leaf: $O(\log_2 M \log_M n)$
- Insert in leaf: $O(L)$
- Split leaf: $O(L)$
- Split parents all the way up to root: $O(M \log_M n)$

Delete

- Find correct leaf: $O(\log_2 M \log_M n)$
- Remove from leaf: $O(L)$
- Adopt/merge from/with neighbor leaf: $O(L)$
- Adopt or merge all the way up to root: $O(M \log_M n)$

Conclusion: Balanced Trees

- Balanced trees make good dictionaries because they guarantee logarithmic-time for find, insert, and delete
- AVL trees maintain balance by tracking height and allowing all children to differ in height by at most 1
- B+ trees maintain balance by keeping nodes at least half full and all leaves at same height

Thank you!