CSE 373
Data Structures & Algorithms

Lecture 07

B-Trees

Announcements

- Homework 2 due on Friday
  - Turning hard copy at the beginning of class OR
  - Submit online before 12:30
- Midterm next Friday
  - Start reading the book !!!

Time Complexity

Suppose we had very many pieces of data (as we would in a database), such as $n = 2^{30} \approx 10^9$.

How many (worst case) hops through the tree to find a node?

- BST

List tree: 10^9

- AVL

$\log_\phi 10^9 =$
$1.44 \log_2 2^{30} = 43$

Space Complexity

What is in a tree node? In an object?

```
Node:
    Object obj;
    Node left;
    Node right;
Object:
    Key key;
    …data…
```

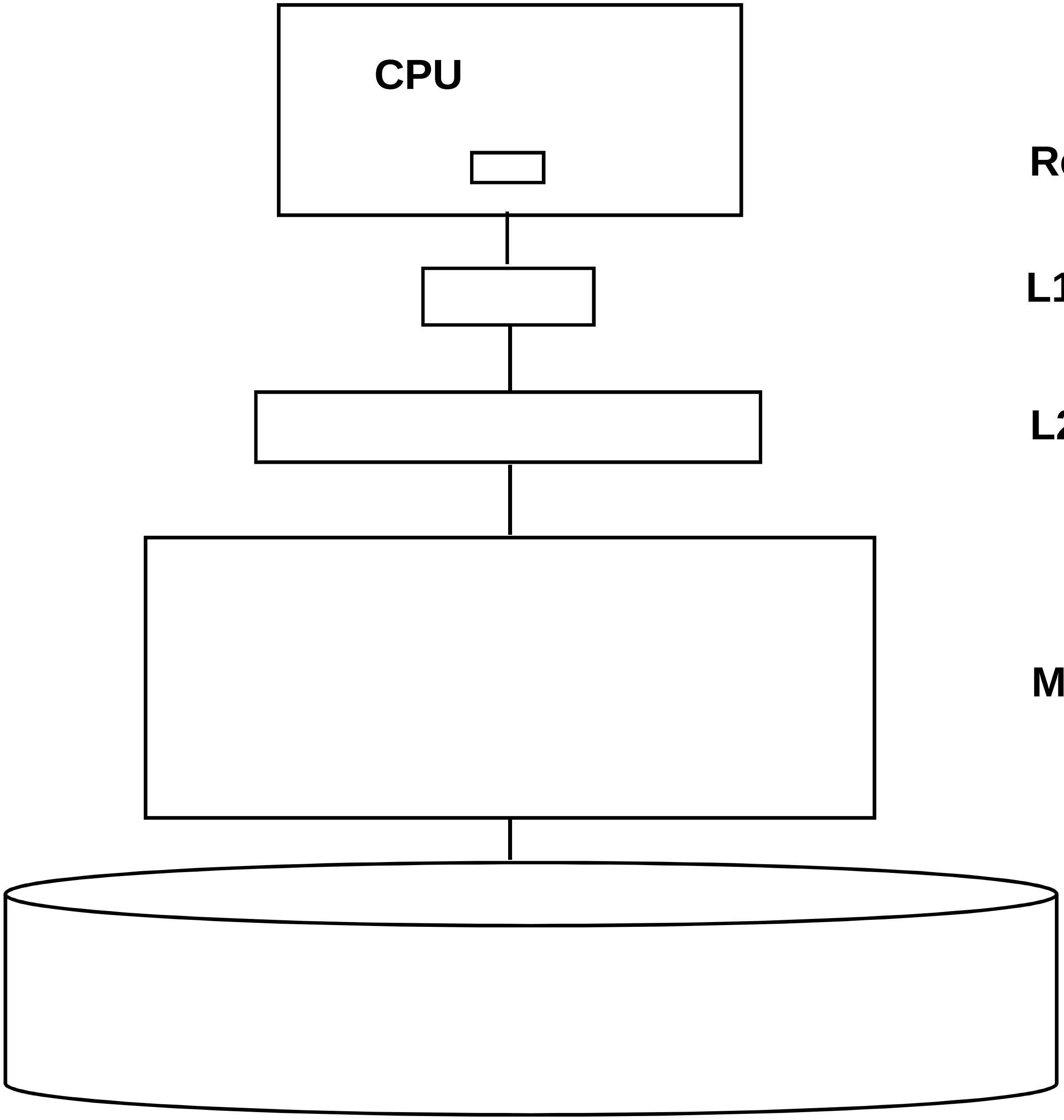
Suppose the data is 1KB.

How much space does the tree take?

10^9 KB = 1TB

How much of the data can live in 1GB of RAM?

10^6 items,
1/1000th of data



CPU

Cycles to access:

Registers	1
L1 Cache	2
L2 Cache	30
Main memory	250

Disk

Random: 30,000,000

Streamed: 5000

Minimizing random disk access

Almost all of our data structure is on disk.

Thus, hopping around in a tree amounts to random accesses to disk.

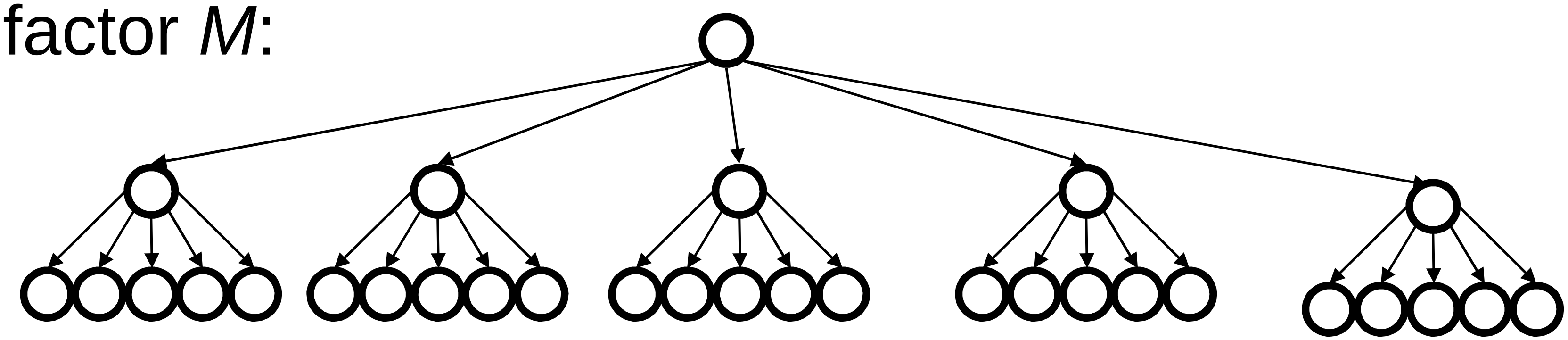
They are really, really painful.

How can we address this problem?

Big branching factor
Implemented using arrays of children at each node
Store keys in nodes, data at leaves

M-ary Search Tree

Suppose we devised a search tree with branching factor *M*:



Complete tree has height: $O(\log_M n) = O(\log_M n/\log_2 M)$

# Hops for *find*: $O(\log_M n)$

Runtime of *find*: $O(\log_2 M * \log_M n) = O(\log_2 n)$

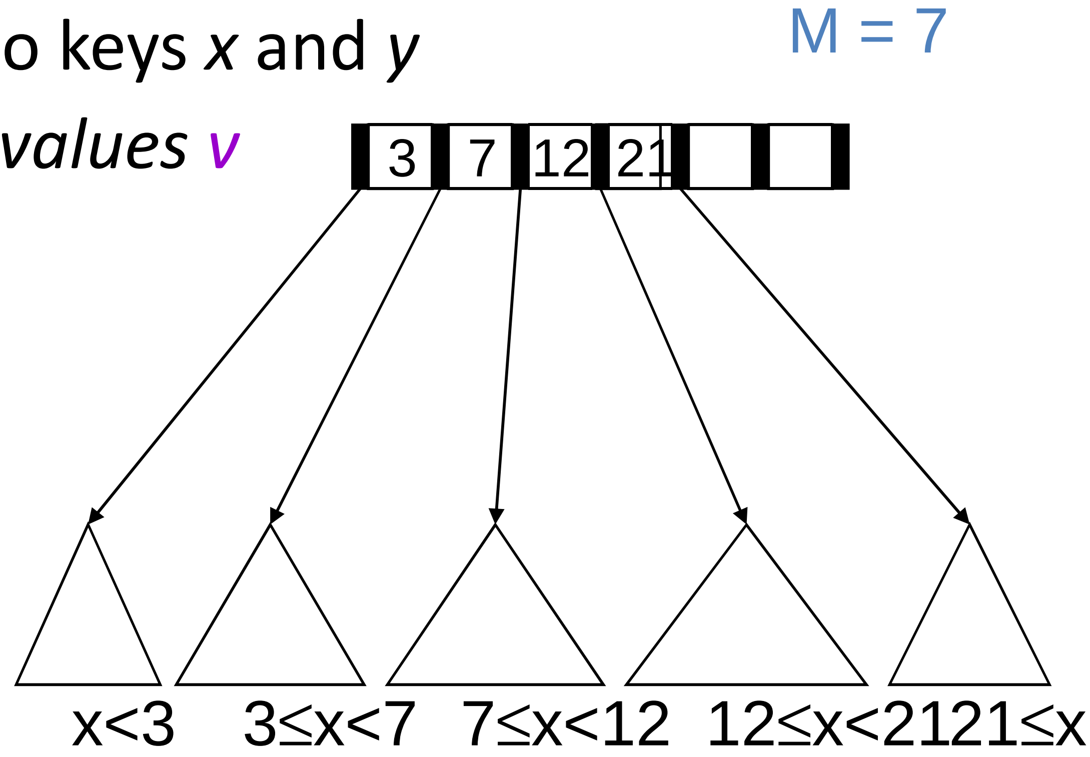
Binary search at a node takes $O(\log_2 M)$ time.
Asymptotically same as AVL, but better disk access

B Trees

- Each internal still has (up to) *M*-1 keys:

- Order property:
  - subtree between two keys *x* and *y* contain leaves with *values* *v* such that *x* ≤ *v* < *y*
  - Note the "≤"

- Leaf nodes contain up to *L* sorted keys.



B Tree Structure Properties

Root (special case)

- has between 2 and **M** children (or could be a leaf)

Internal nodes

Nodes are at least ½ full

- store up to **M**-1 keys
- have between ⌈**M**/2⌉ and **M** children

Leaf nodes

- where data is stored

Leaves are at least ½ full

- contain between ⌈**L**/2⌉ and **L** data items

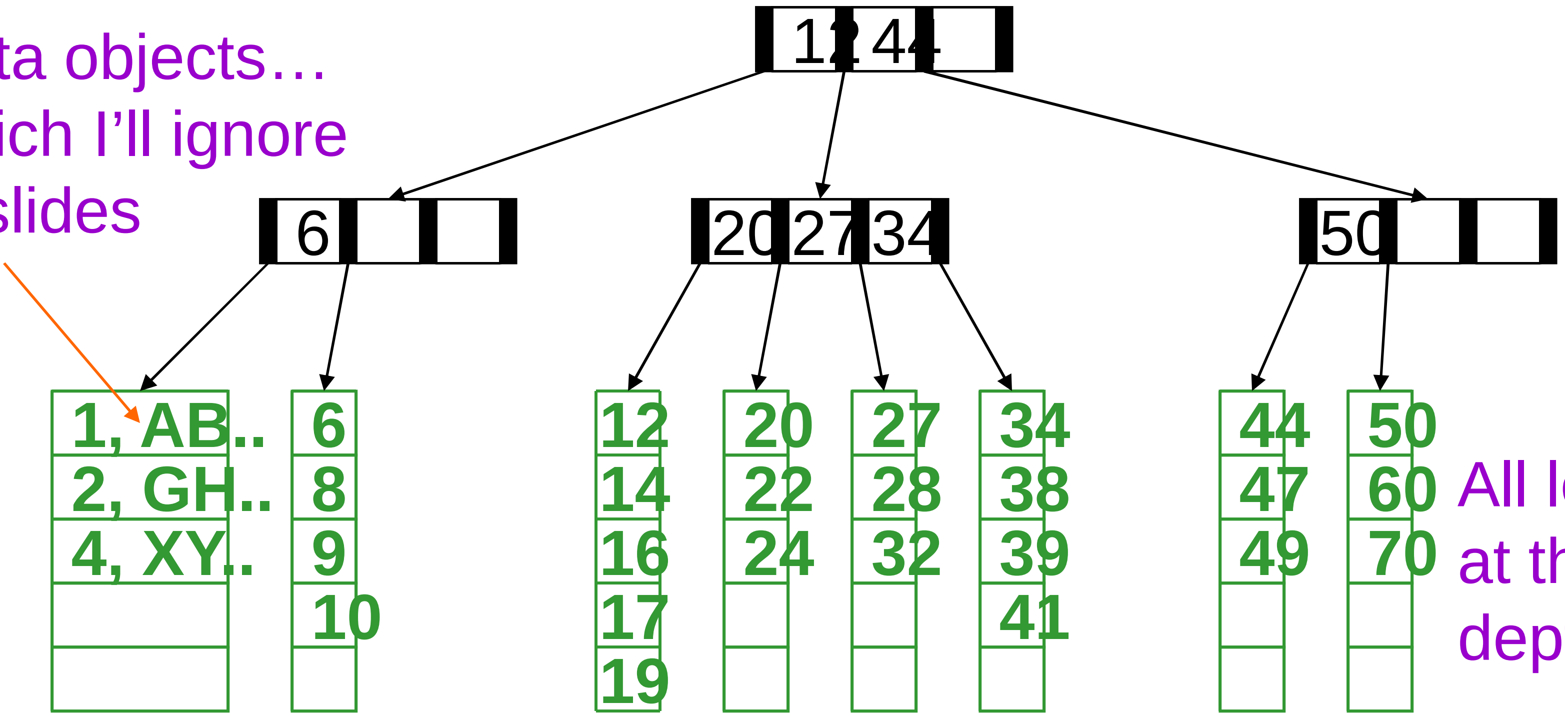
The tree is ***perfectly balanced*** !

B Tree: Example

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

and L = 5 (# data items in leaf)

Data objects… which I'll ignore in slides



All leaves at the same depth

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

Disk Friendliness

What makes B trees disk-friendly?

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

2. Internal nodes contain *only* keys;
   Only leaf nodes contain keys and actual *data*
   - Much of tree structure can be loaded into memory irrespective of data object size
   - Data actually resides in disk

B Trees in Practice

- Typical order: M=200. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

B trees vs. AVL trees

Suppose again we have $n = 2^{30} \approx 10^9$ items:

- Depth of AVL Tree

  43

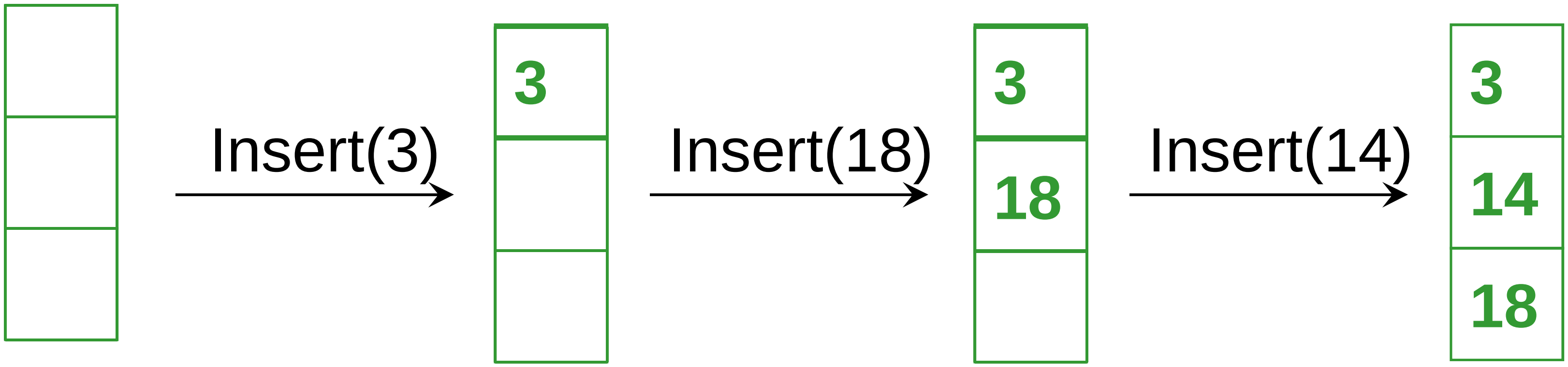
- Depth of B Tree with M = 256, L = 256

  $\text{Log}_{128} 10^9 = 4.3$

So let's see how we do this...

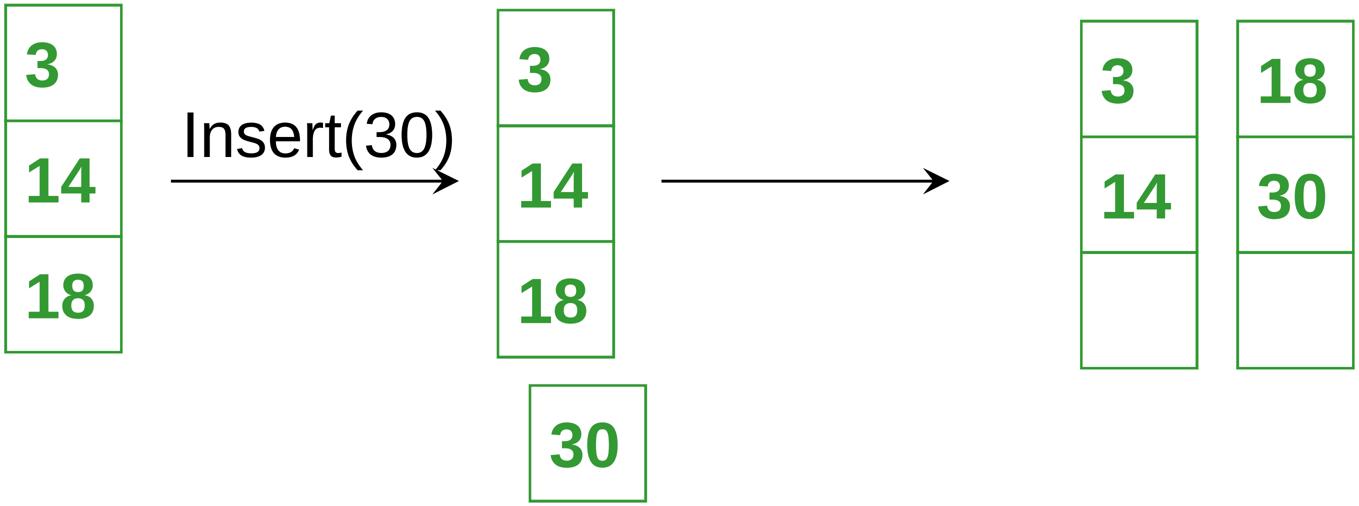
13

Building a B Tree with Insertions



The empty B-Tree

M = 3 **L = 3**



3
14
18

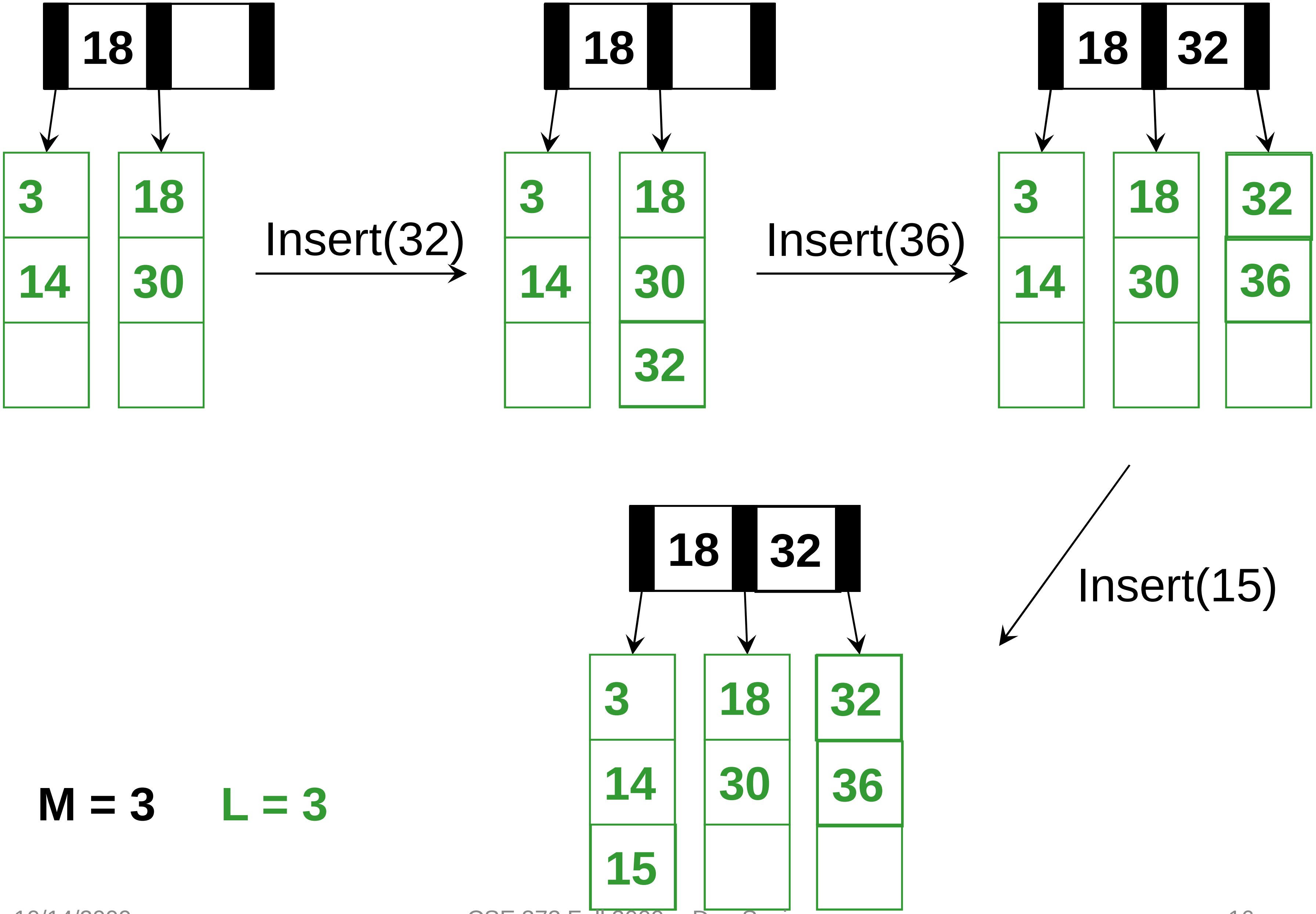
Insert(30)

3
14
18
30

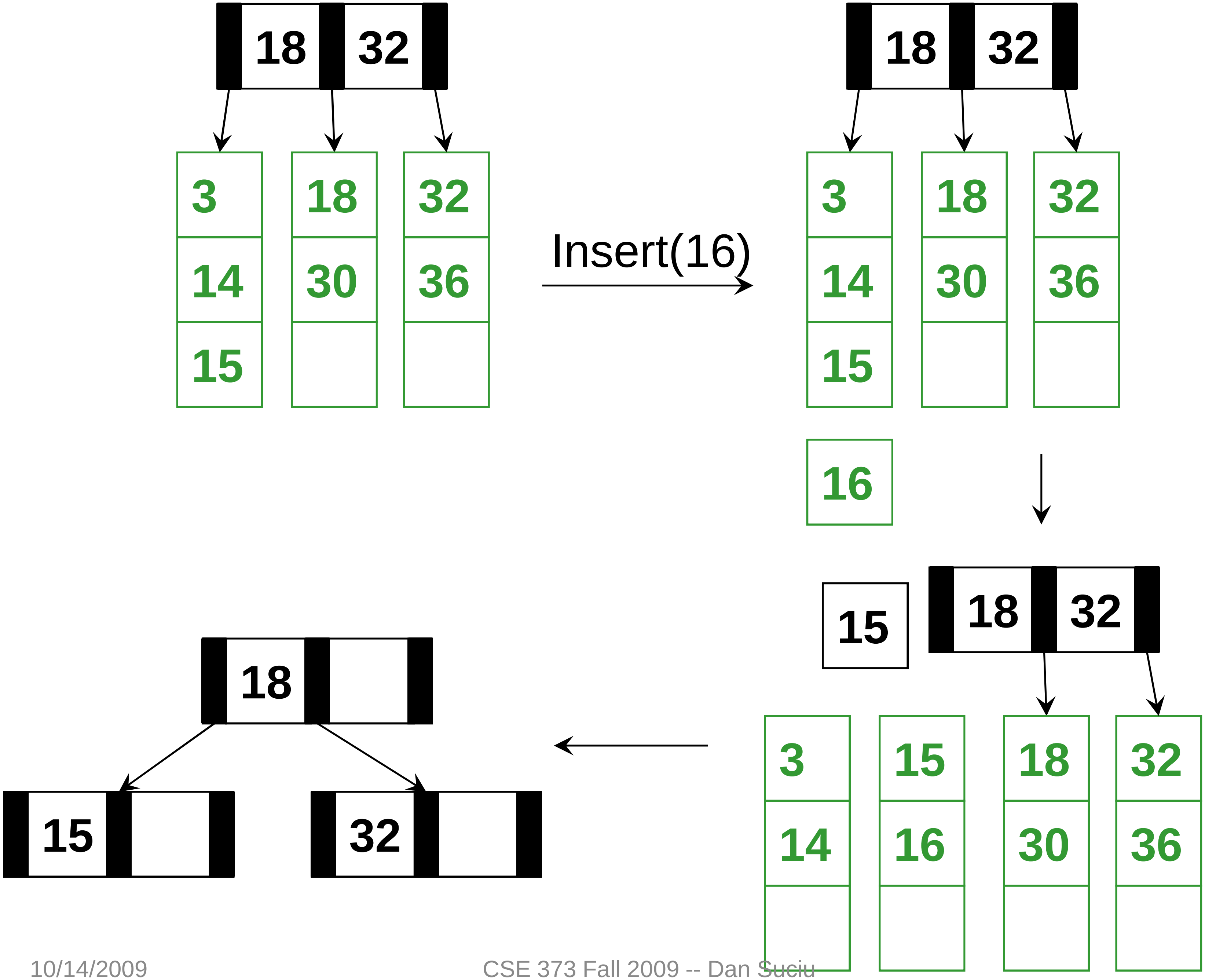
3
14

18
30

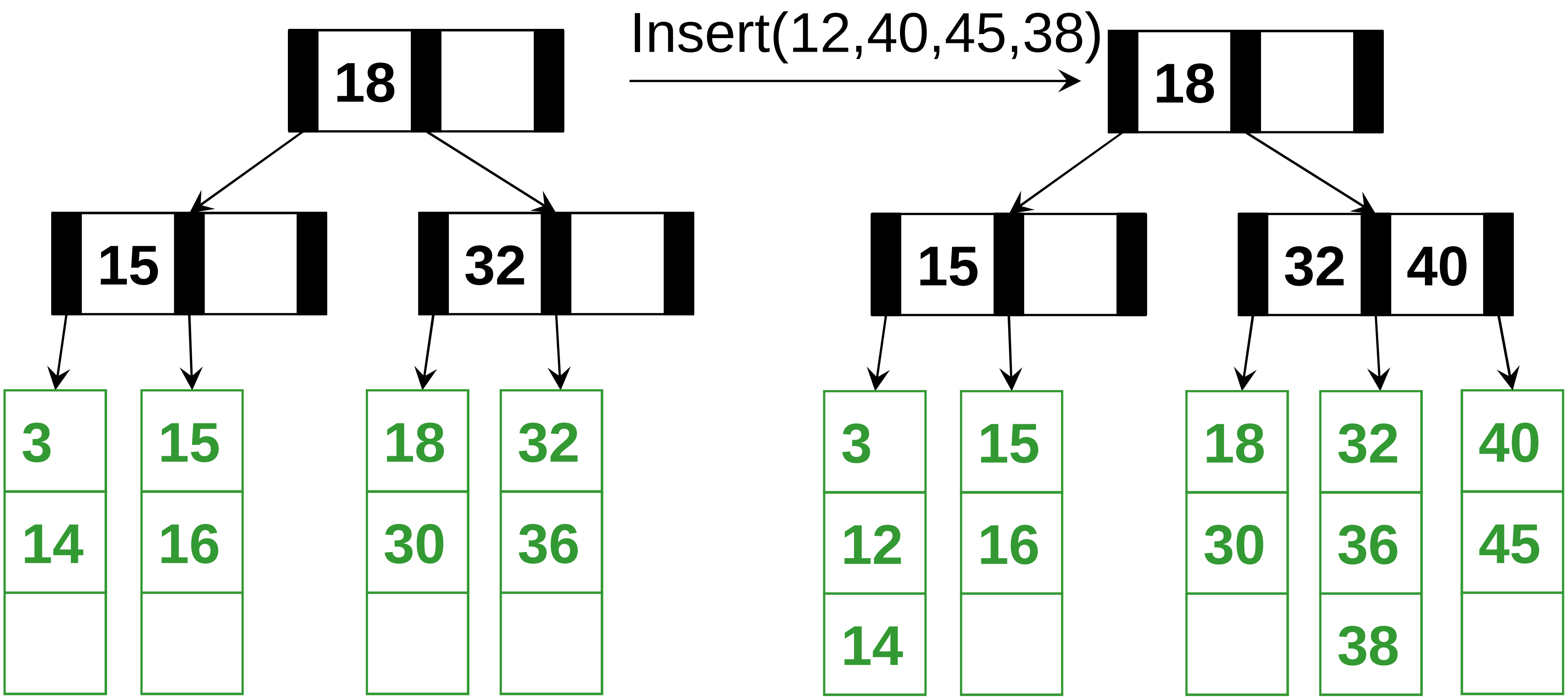
M = 3 **L = 3**


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

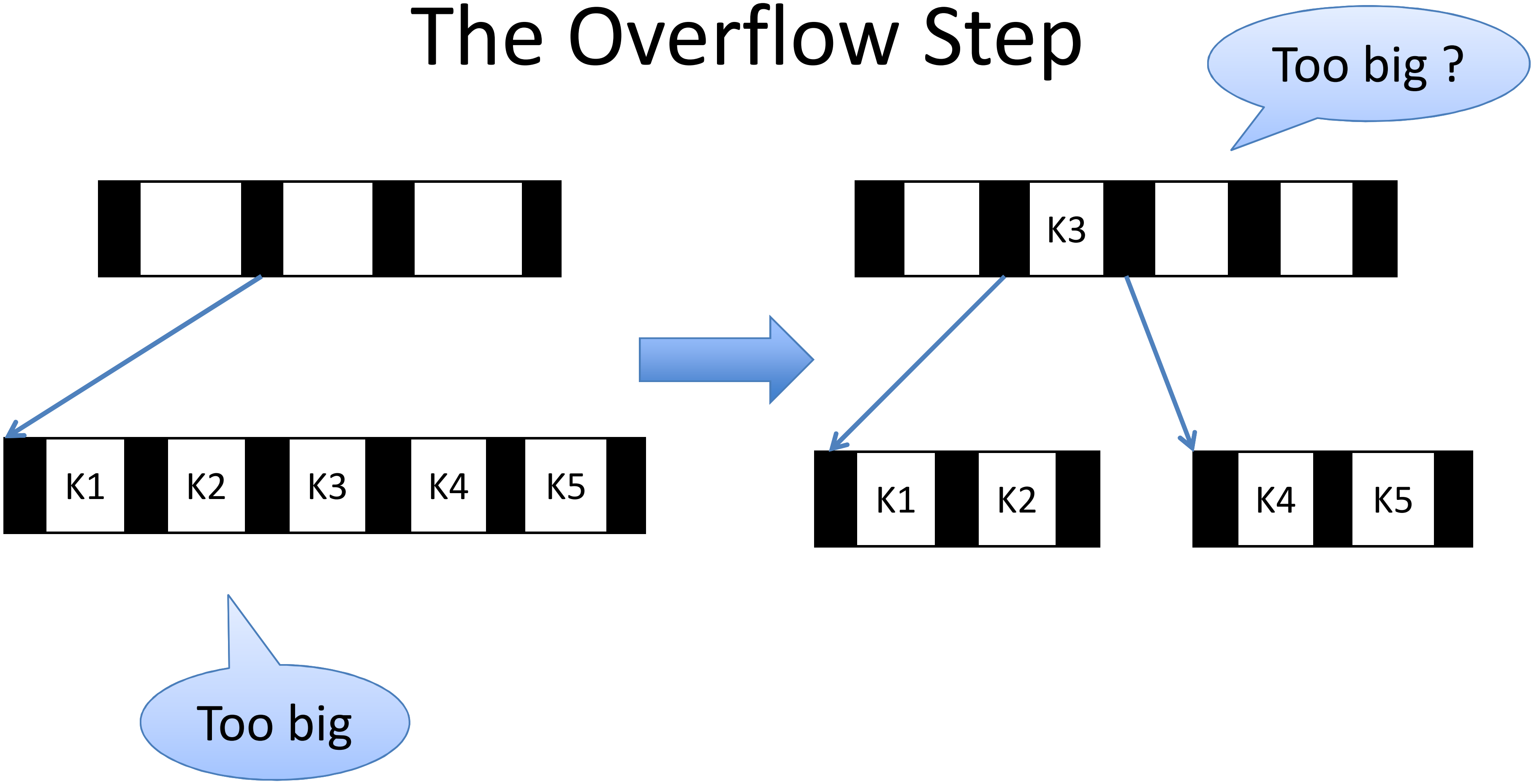
M = 3 **L = 3**


18
32
3
14
15
18
30
32
36
Insert(16)
18
32
3
14
15
18
30
32
36
16
15
18
32
3
14
15
16
18
30
32
36
18
15
32

M = 3 **L = 3**


Insert(12,40,45,38)
18
15
32
3
14
15
16
18
30
32
36
18
15
32
40
3
12
14
15
16
18
30
32
36
38
40
45

Insertion Algorithm: The Overflow Step



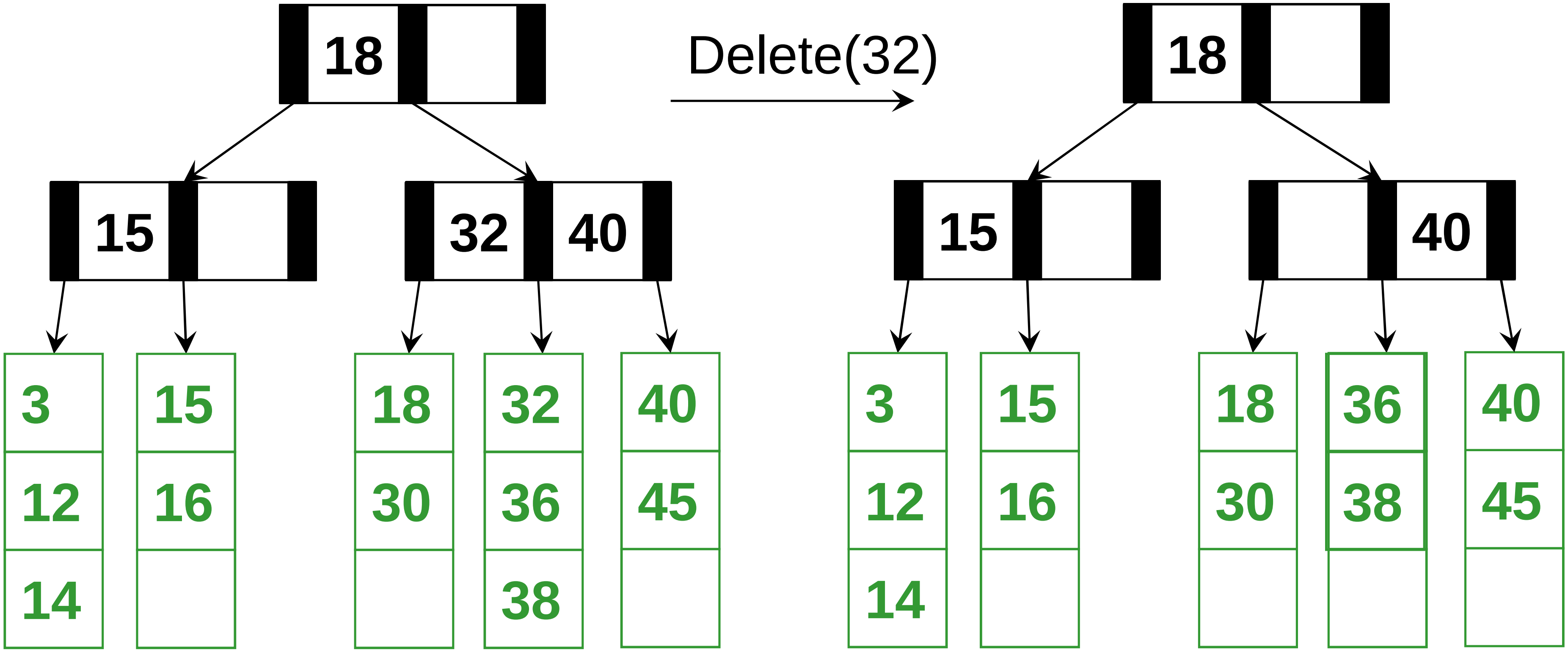
M=5

Insertion Algorithm

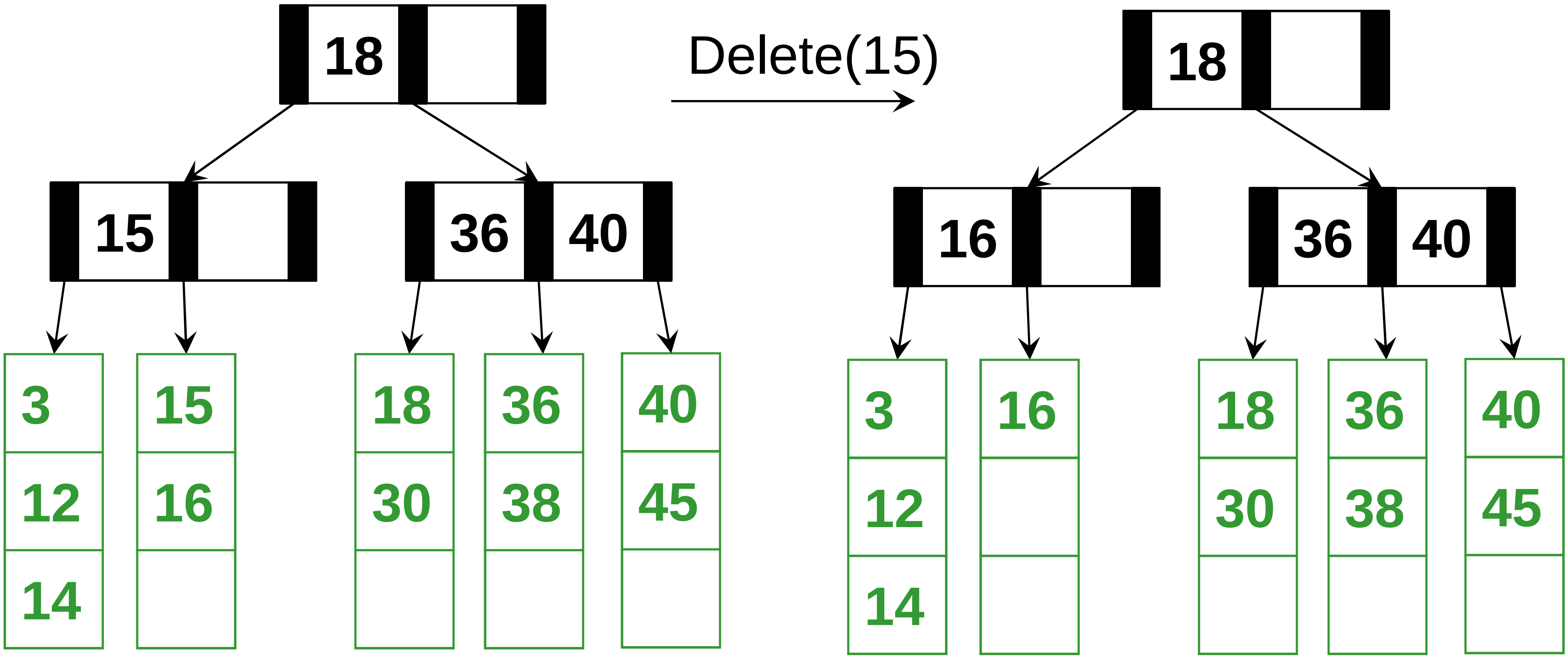
1. Insert the key in its leaf in sorted order
2. If the leaf ends up with ***L+1*** items, **overflow**!
   - Split the leaf into two nodes:
     $\lceil (L+1)/2 \rceil$ smaller keys
     $\lfloor (L+1)/2 \rfloor$ larger keys
   - Add the new child to the parent
   - If the parent ends up with ***M*+1** children, **overflow**!
3. If an internal node ends up with ***M+1*** children, **overflow**!
   - Split the node into two nodes:
     $\lceil (M+1)/2 \rceil$ children with smaller keys
     $\lfloor (M+1)/2 \rfloor$ children with larger keys
   - Add the new child to the parent
   - If the parent ends up with ***M*+1** items, **overflow**!
4. If the root ends up with ***M+1*** children, split it in two, and create new root with two children

This makes the tree deeper!

And Now for Deletion…



M = 3 L = 3

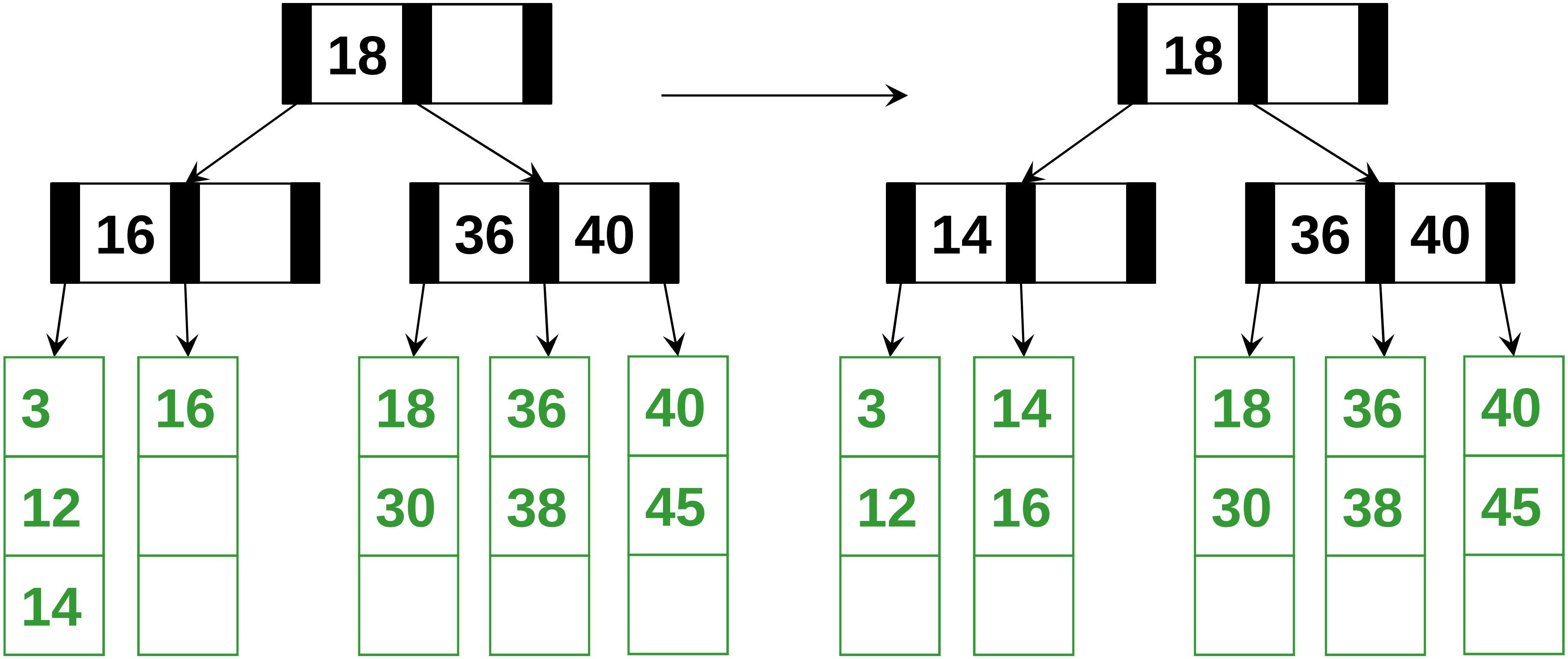


Are we okay?

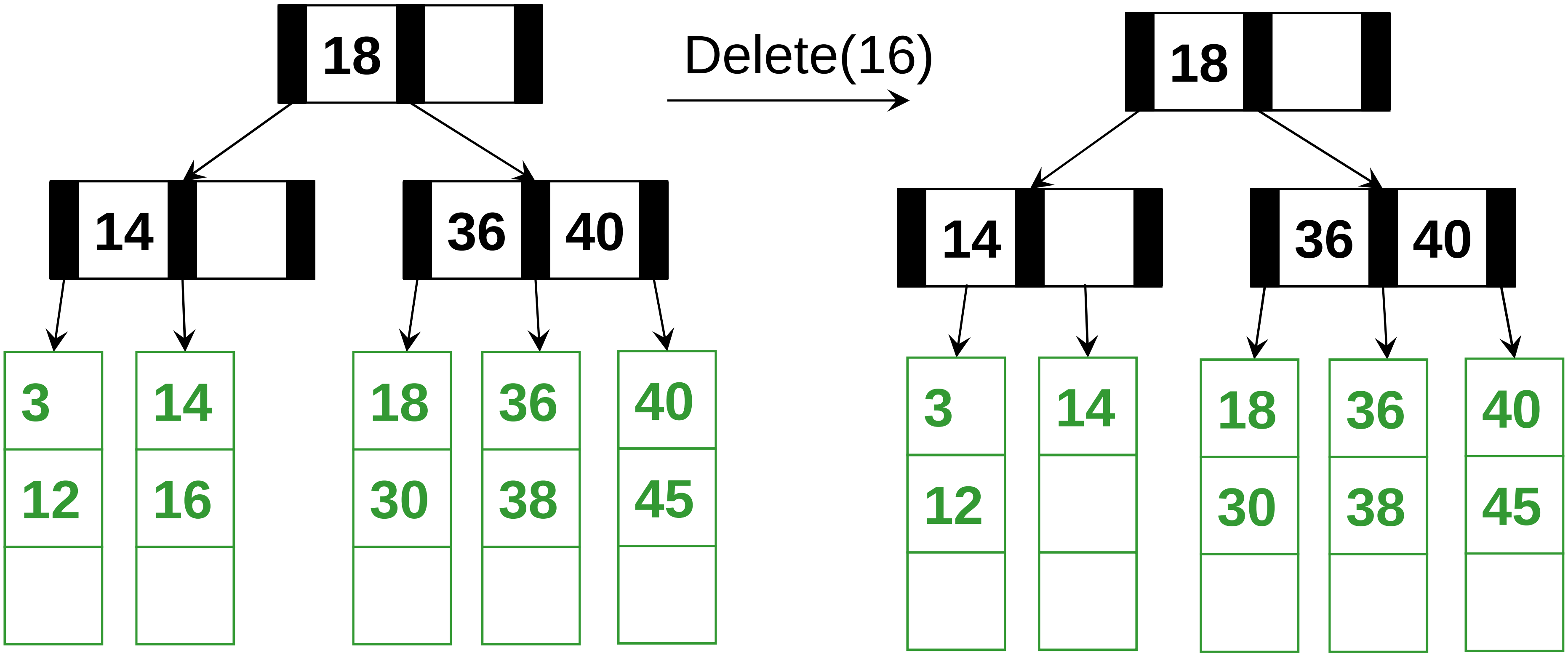
M = 3 L = 3

Dang, not half full

Are you using that 14?
Can I borrow it?


18
16
36
40
3
12
14
16
18
30
36
38
40
45
18
14
36
40
3
12
14
16
18
30
36
38
40
45

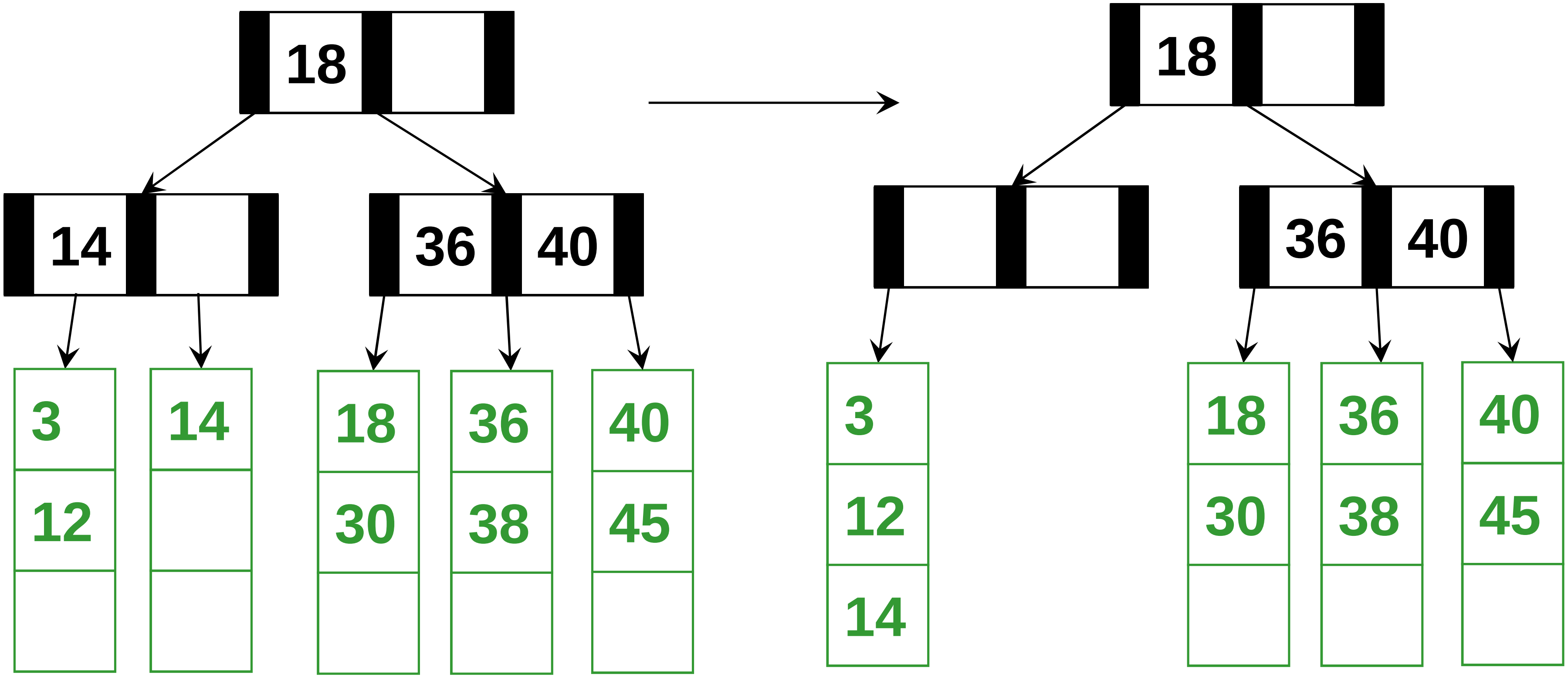
M = 3 L = 3



Delete(16)

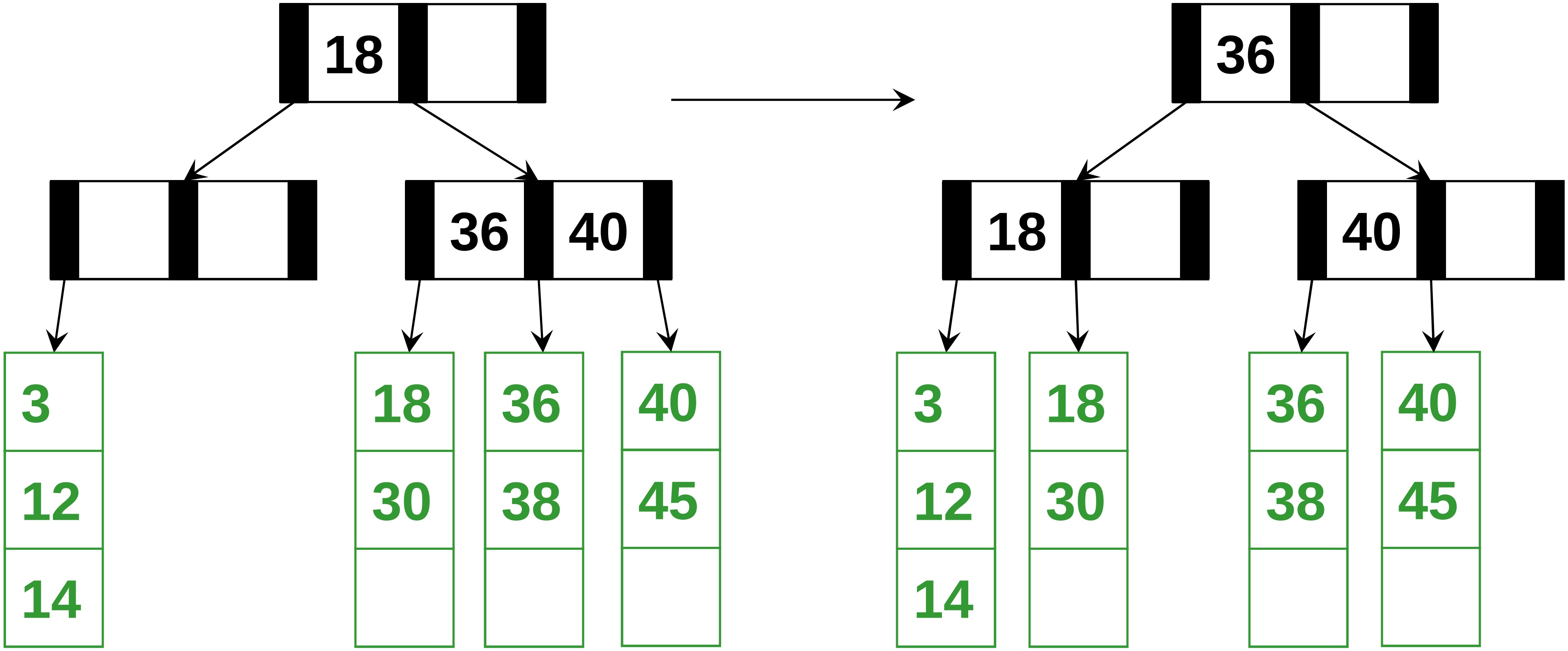
M = 3 L = 3

Are you using that 12?

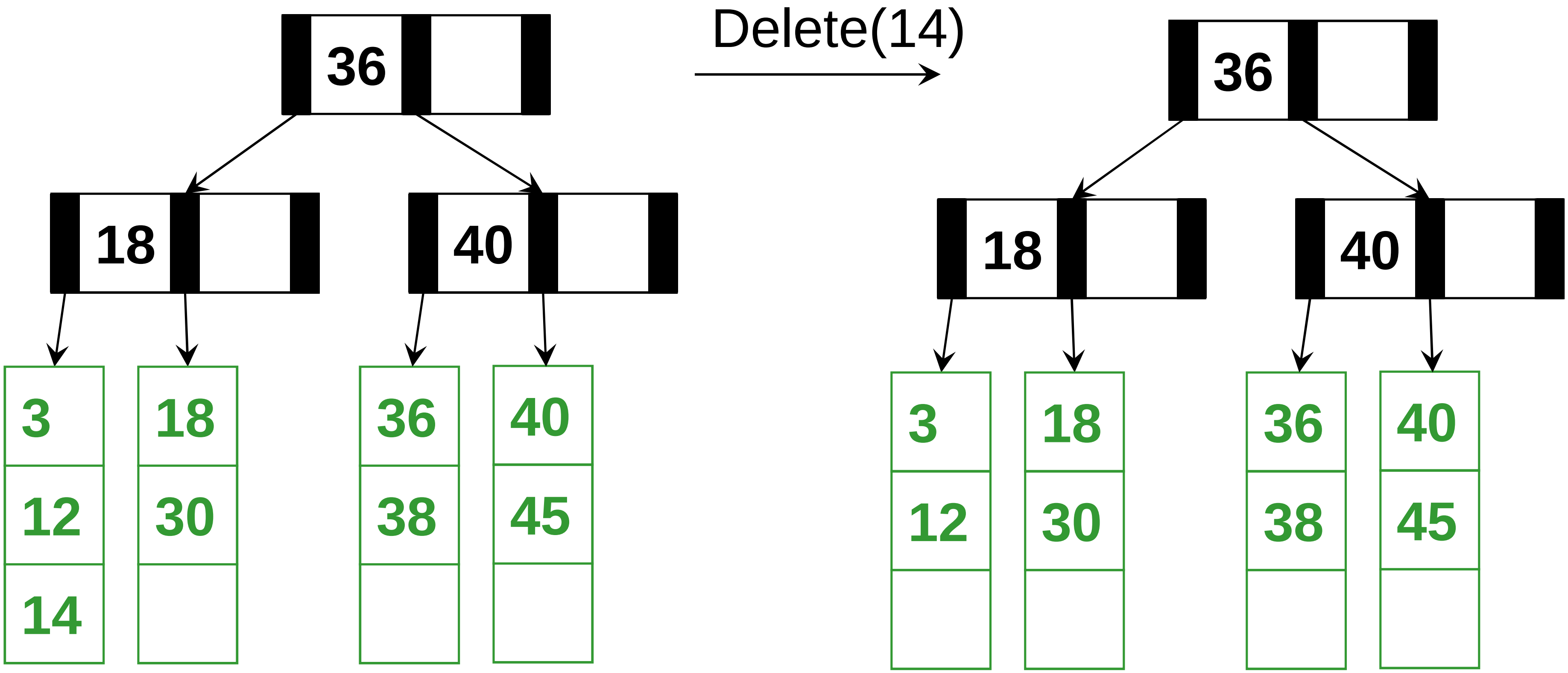


M = 3 L = 3

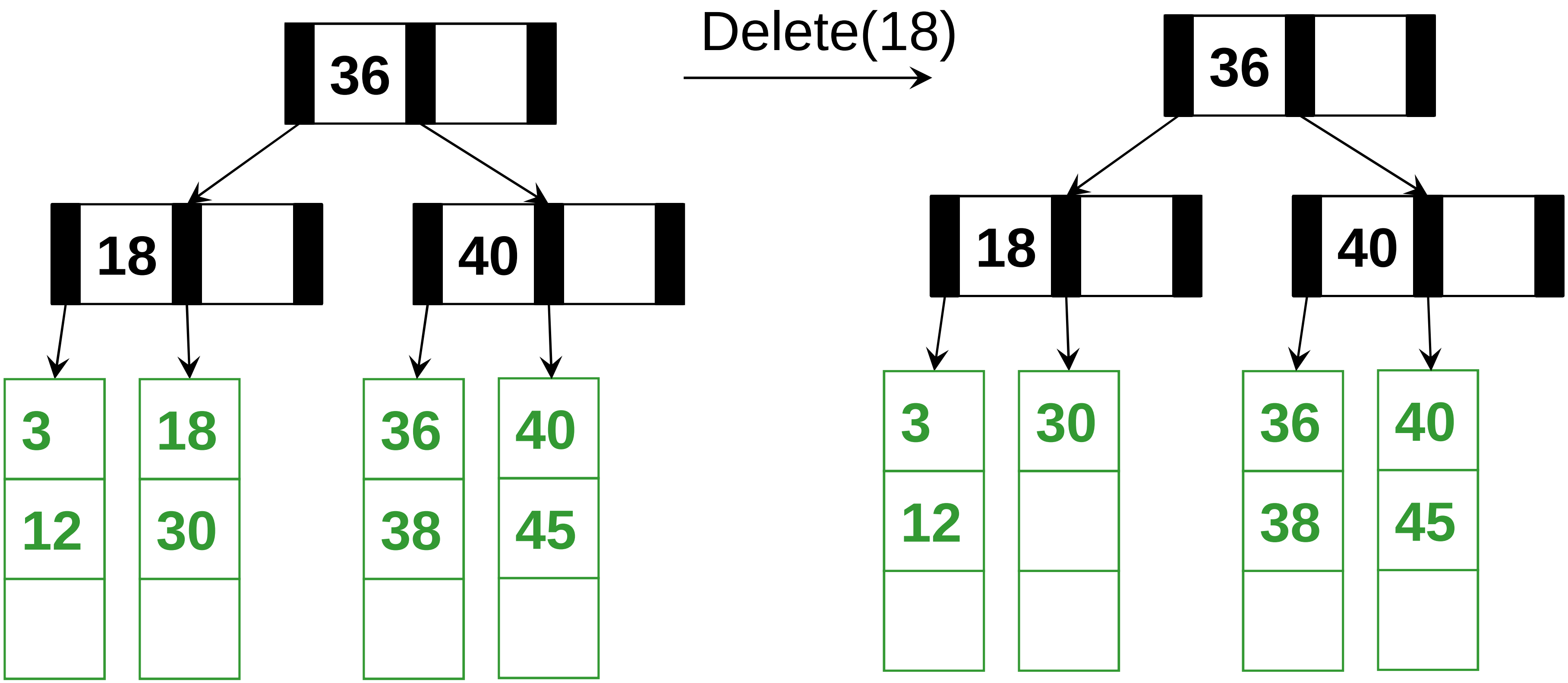
Are you using the node18/30?



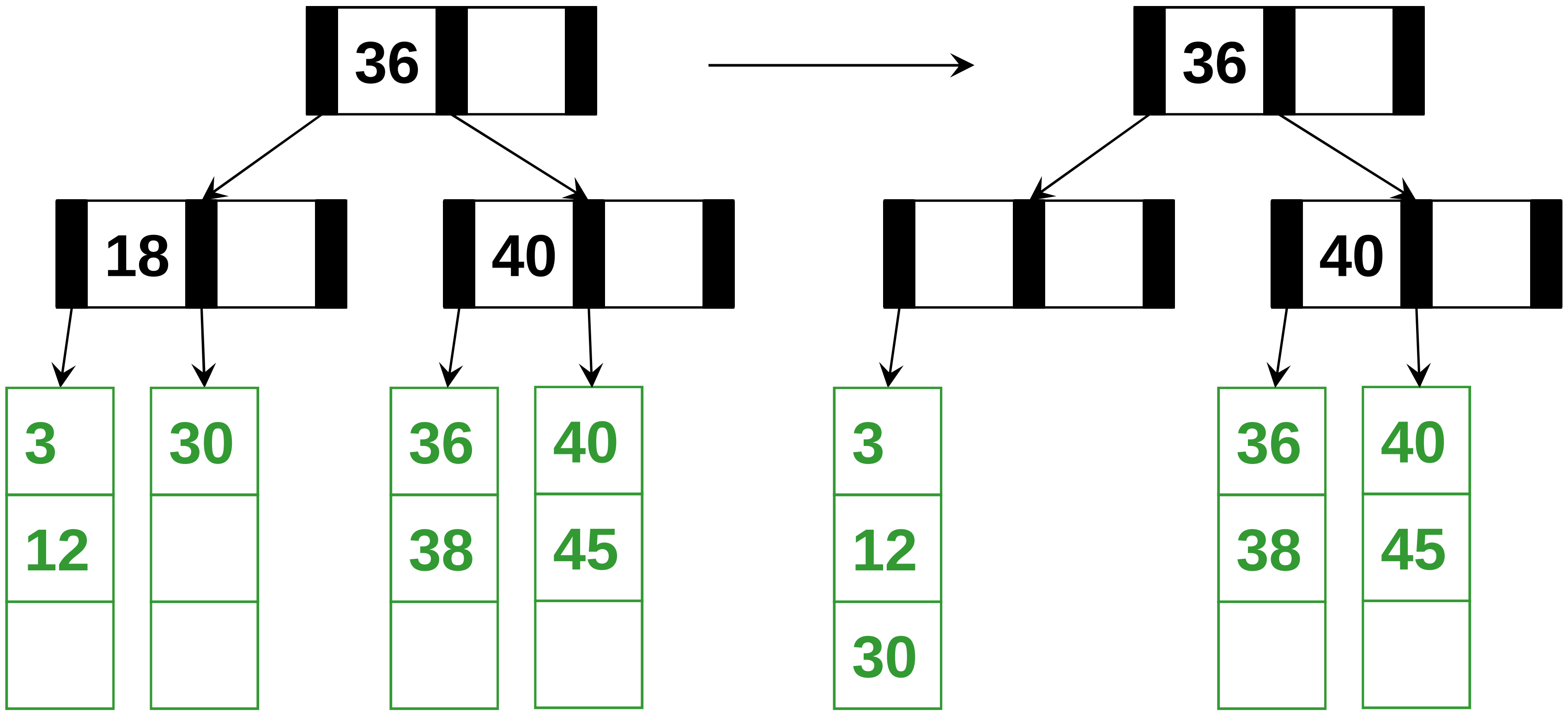
M = 3 L = 3


36
18
40
3
12
14
18
30
36
38
40
45
Delete(14)
36
18
40
3
12
18
30
36
38
40
45

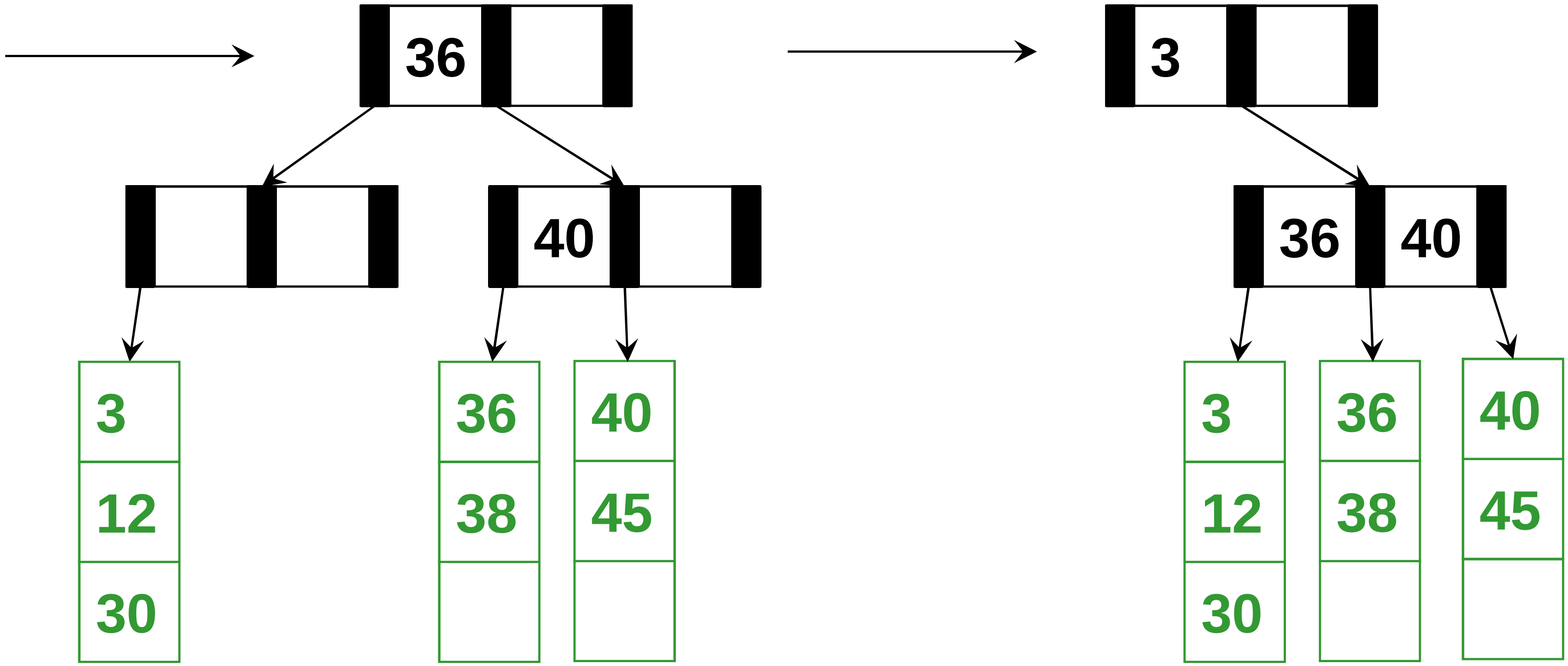
M = 3 **L = 3**



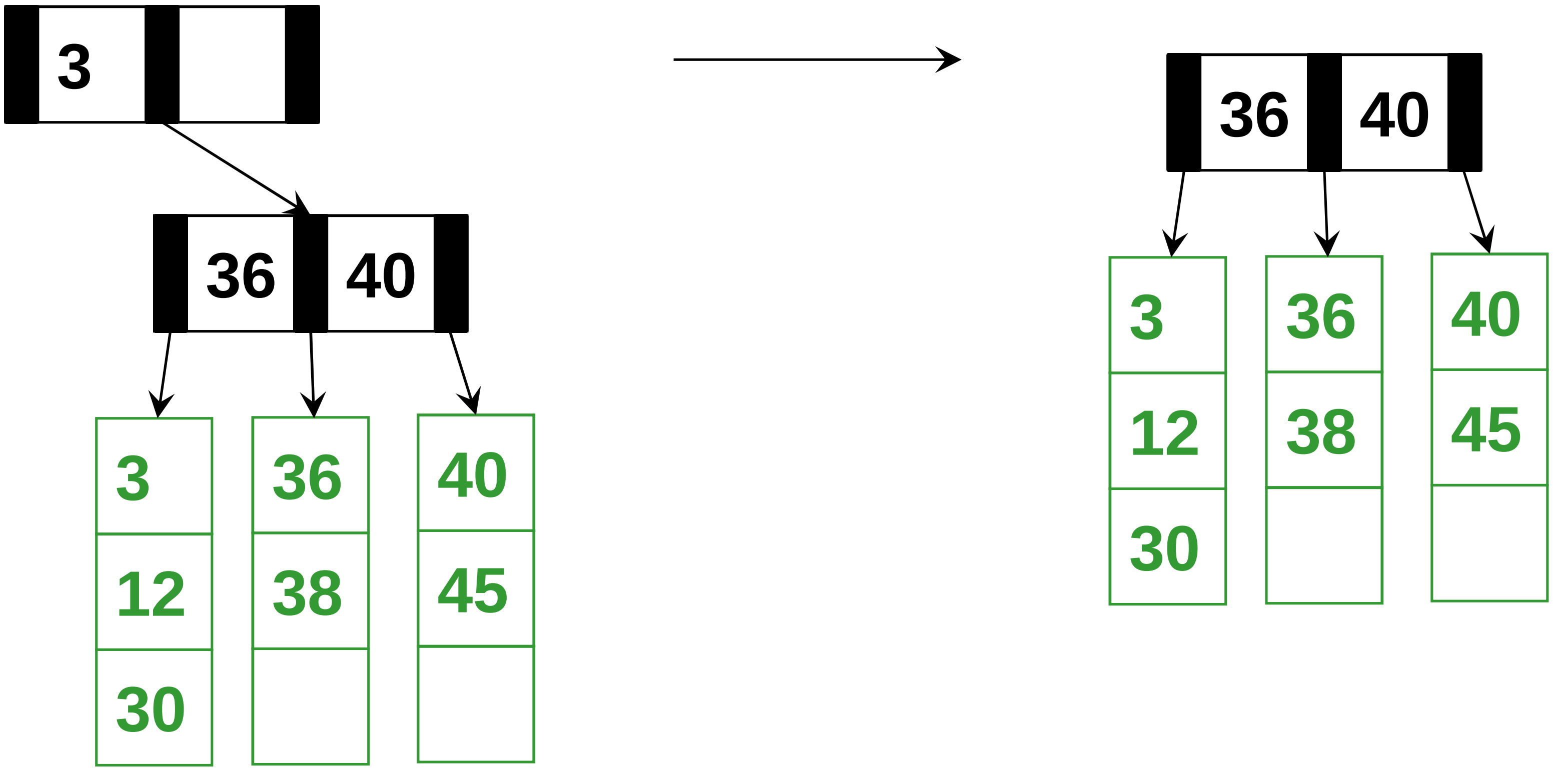
M = 3 L = 3



M = 3 L = 3

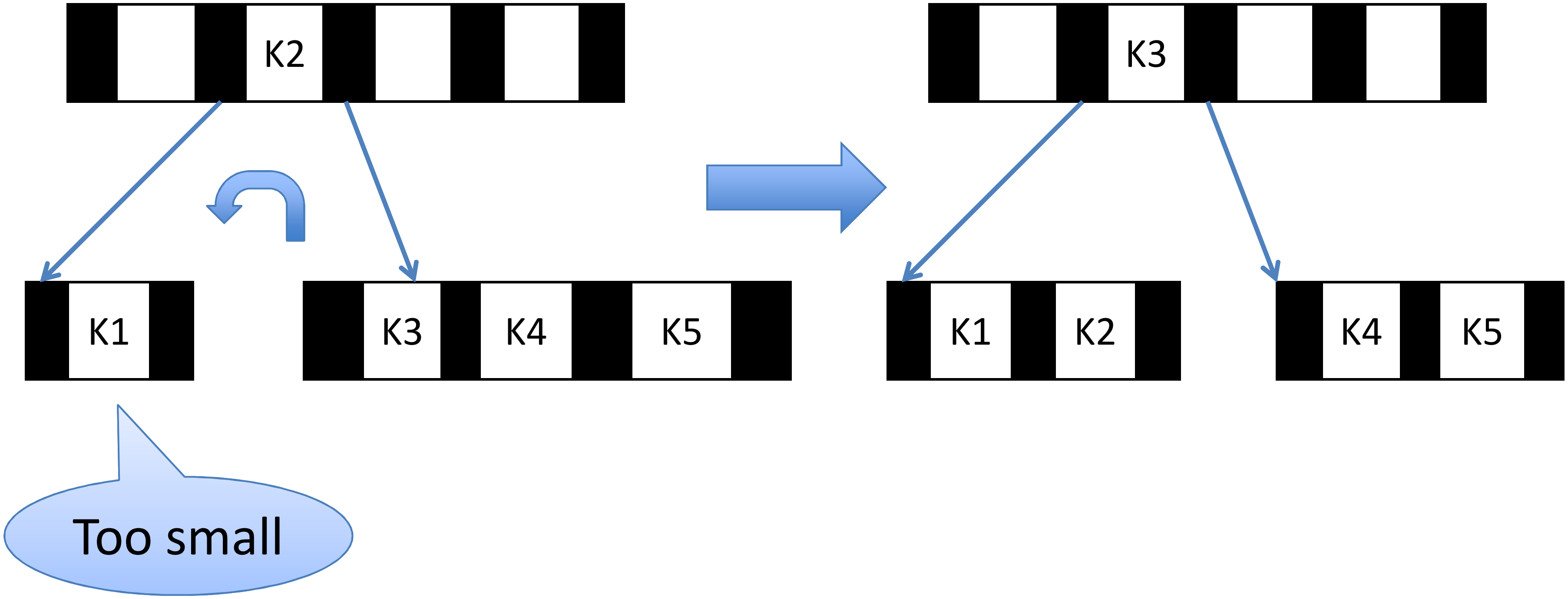


M = 3 **L = 3**



M = 3 **L = 3**

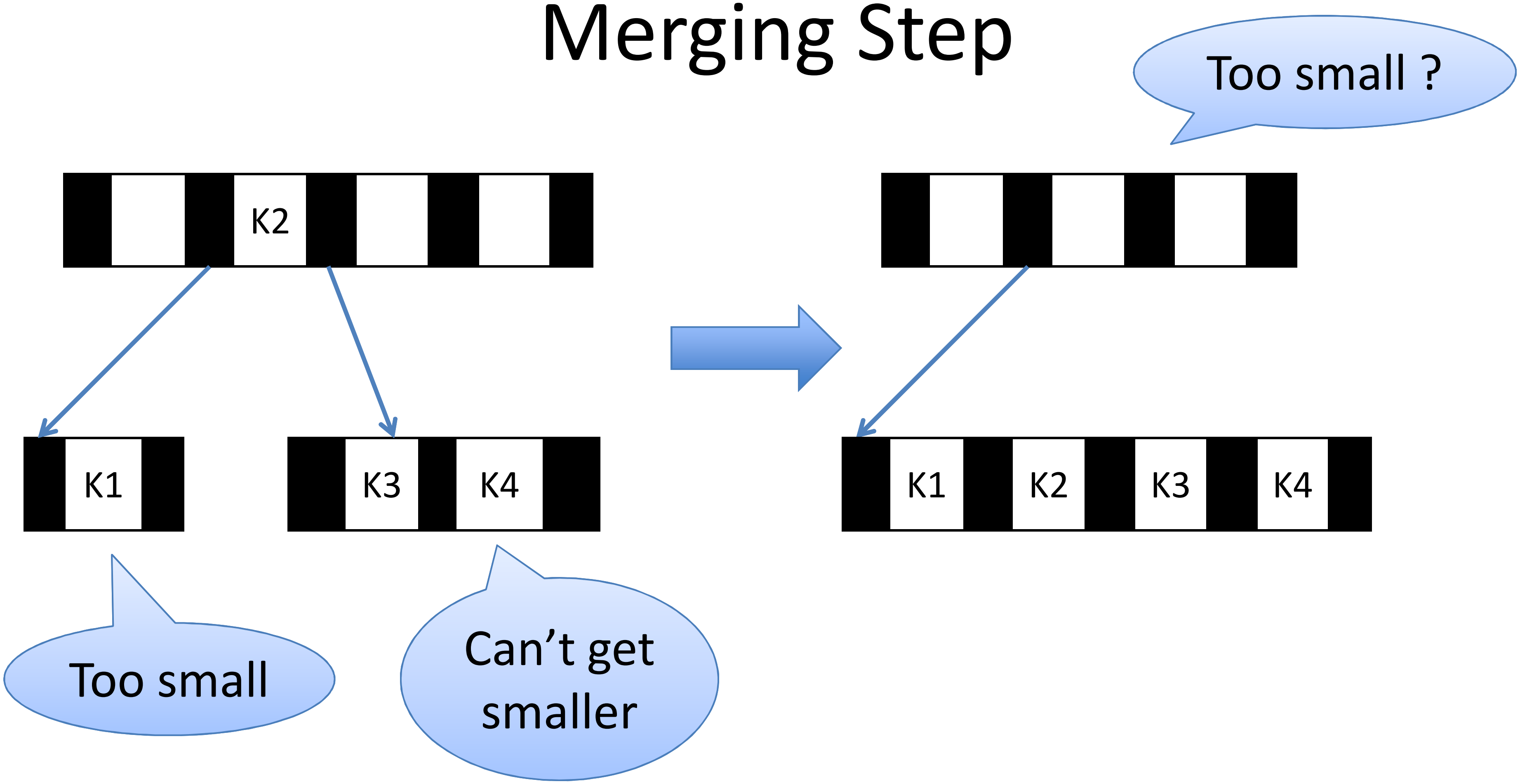
Deletion Algorithm: Rotation Step



M=5

This is *left* rotation. Similarly, *right* rotation

Deletion Algorithm: Merging Step



M=5

Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than $\lceil \boldsymbol{L/2} \rceil$ items, **underflow**!
   - Try a left rotation
   - If not, try a right rotation
   - If not, merge, then check the parent node for underflow

Deletion Slide Two

3. If an internal node ends up with fewer than $\lceil M/2 \rceil$ children, **underflow**!
   - Try a left rotation
   - If not, try a right rotation
   - If not, merge, then check the parent node for underflow

4. If the root ends up with only one child, make the child the new root of the tree

This reduces the height of the tree!

Complexity

- Find: $O(\log_2 M \log_M n)$
- Insert:
  - find: $O(\log_2 M \log_M n)$
  - Insert in leaf: $O(M)$
  - split/propagate up: $O(M \log_M n)$

- O(M) costs are negligible, it’s disk that kills us