CSE332: Data Abstractions

Lecture 10: More B-Trees

Tyler Robison

Summer 2010

B-Tree Review: Another dictionary

- Overall idea:
  - Large data sets won't fit entirely in memory
  - Disk access is slow
  - Set up tree so we do one disk access per node in tree
  - Then our goal is to keep tree shallow as possible
  - Balanced binary tree is a good start, but we can do better than $\log_2 n$ height
  - In an M-ary tree, height drops to $\log_M n$
    - Why not set M really really high? Height 1 tree…
    - Instead, set M so that each node fits in a disk block

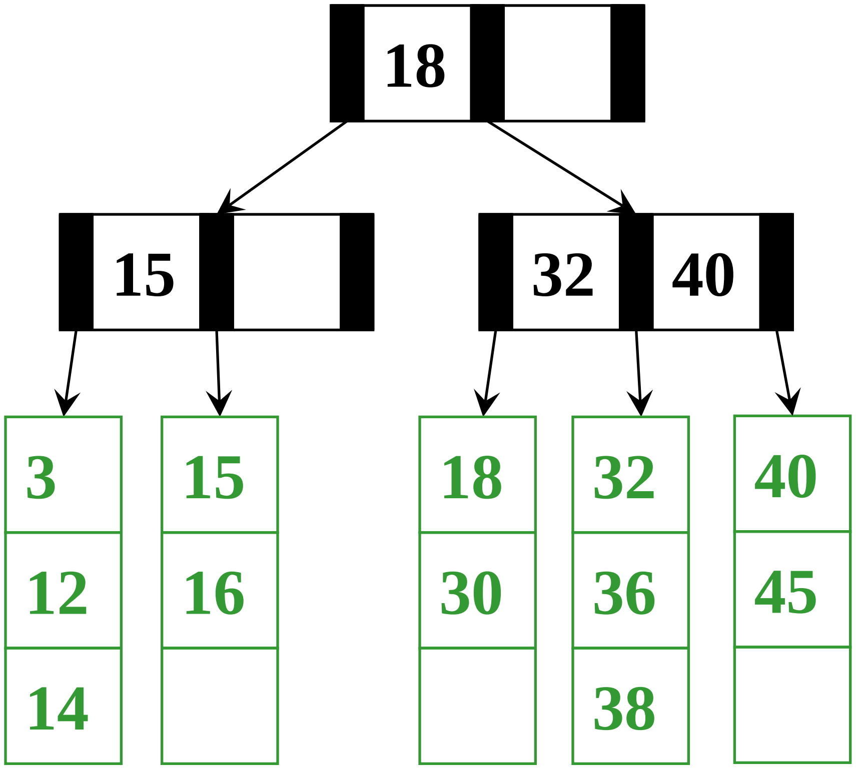
B-Tree Review

There are different variants of B-Trees you can use (adoption, etc.)

- M-ary tree with room for L data items at each leaf
- All data kept at leaves
- Order property:

  Subtree **between** keys *x* and *y* contains only data that is ≥ ***x*** and < ***y*** (notice the ≥)
- Balance property:

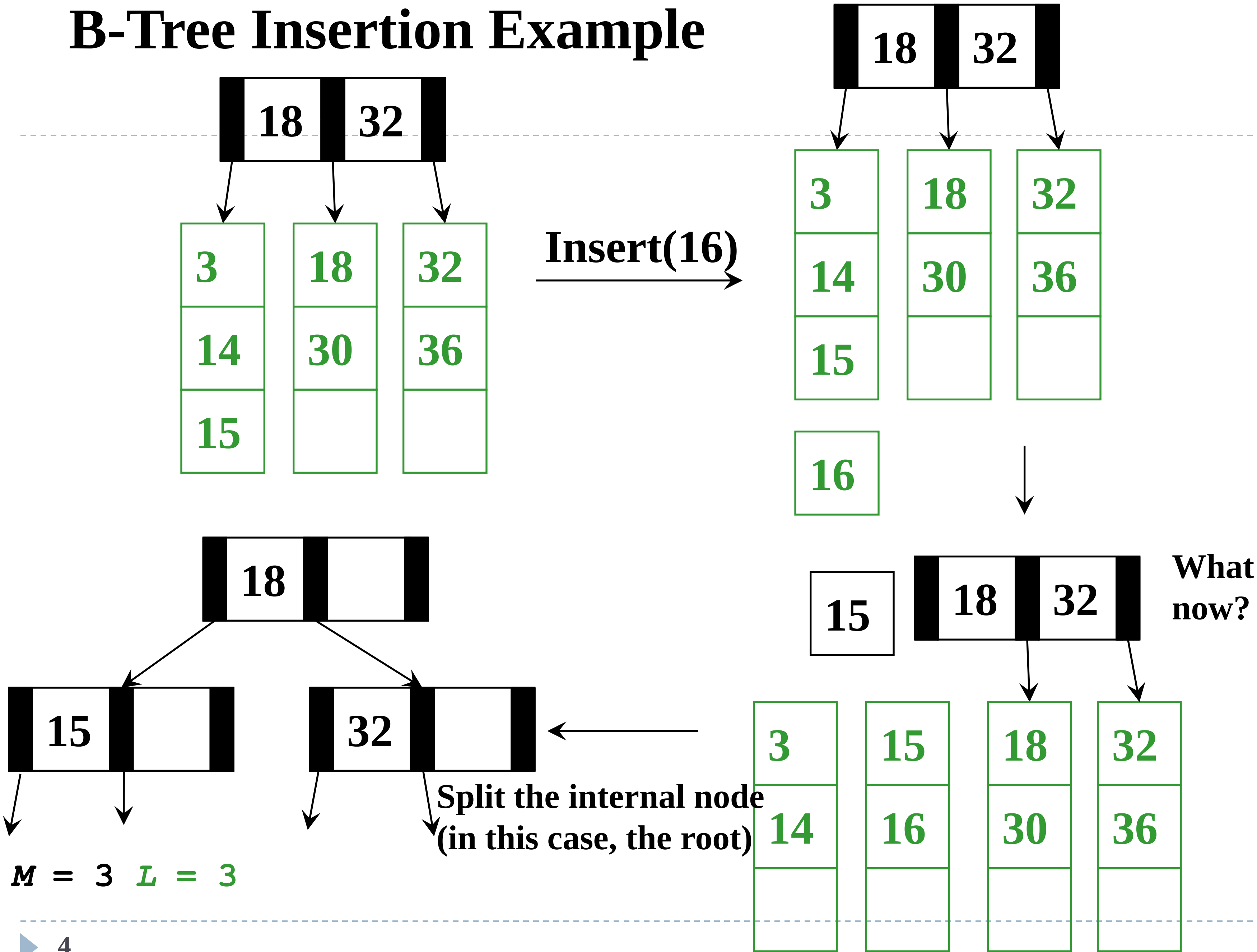
  All nodes and leaves at least half full, and all leaves at same height
- **`find`** and **`insert`** efficient
  - **`insert`** uses *splitting* to handle overflow, which may require splitting parent, and so on recursively



M=3, L=3

Horizontal: Internal, Vertical: Leaf

B-Tree Insertion Example



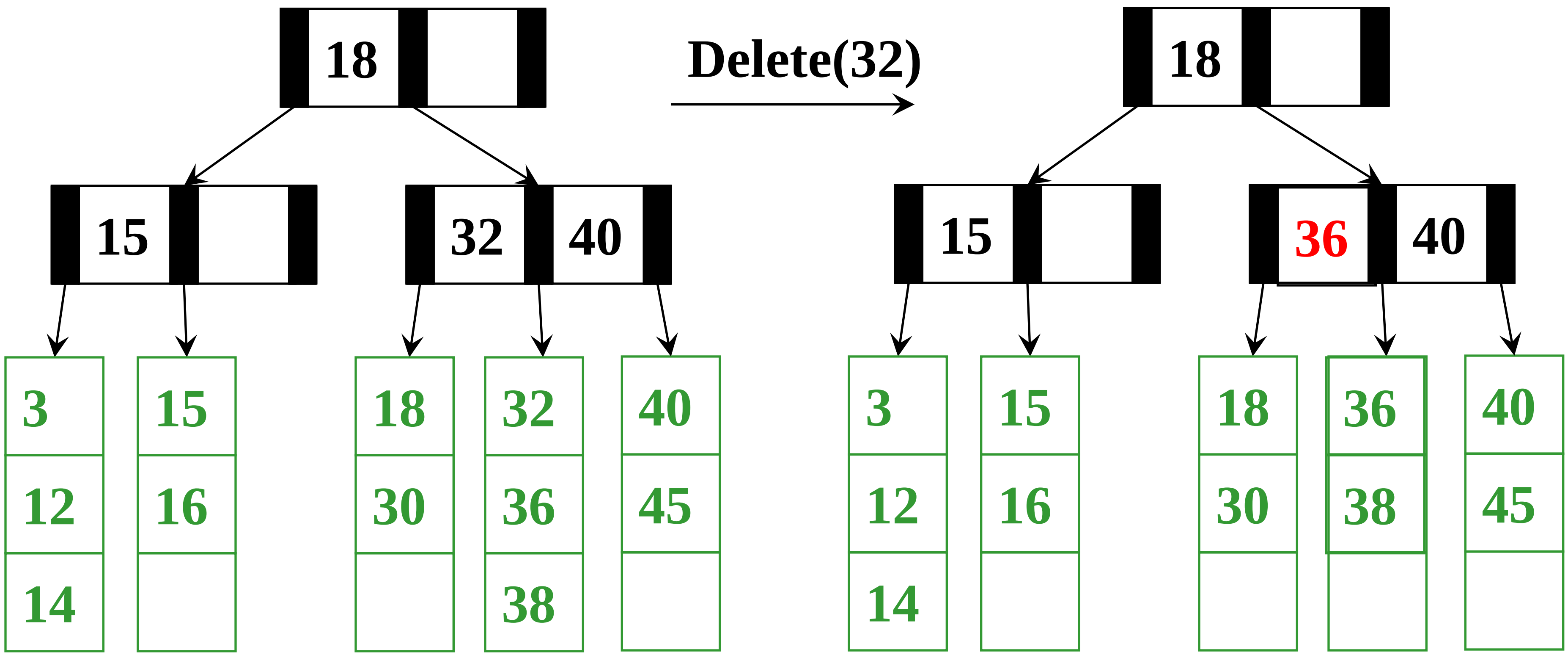
B-Tree Insertion Algorithm Overview

1. Traverse from the root to the proper leaf. Insert the data in its leaf in sorted order
2. If the leaf now has L+1 items, *overflow!*
   - Split the leaf into two leaves:
   - Attach the new child to the parent
3. If an internal node has M+1 children, *overflow!*
   - Split the node into two nodes
   - Attach the new child to the parent

Splitting at a node (step 3) could make the parent overflow too

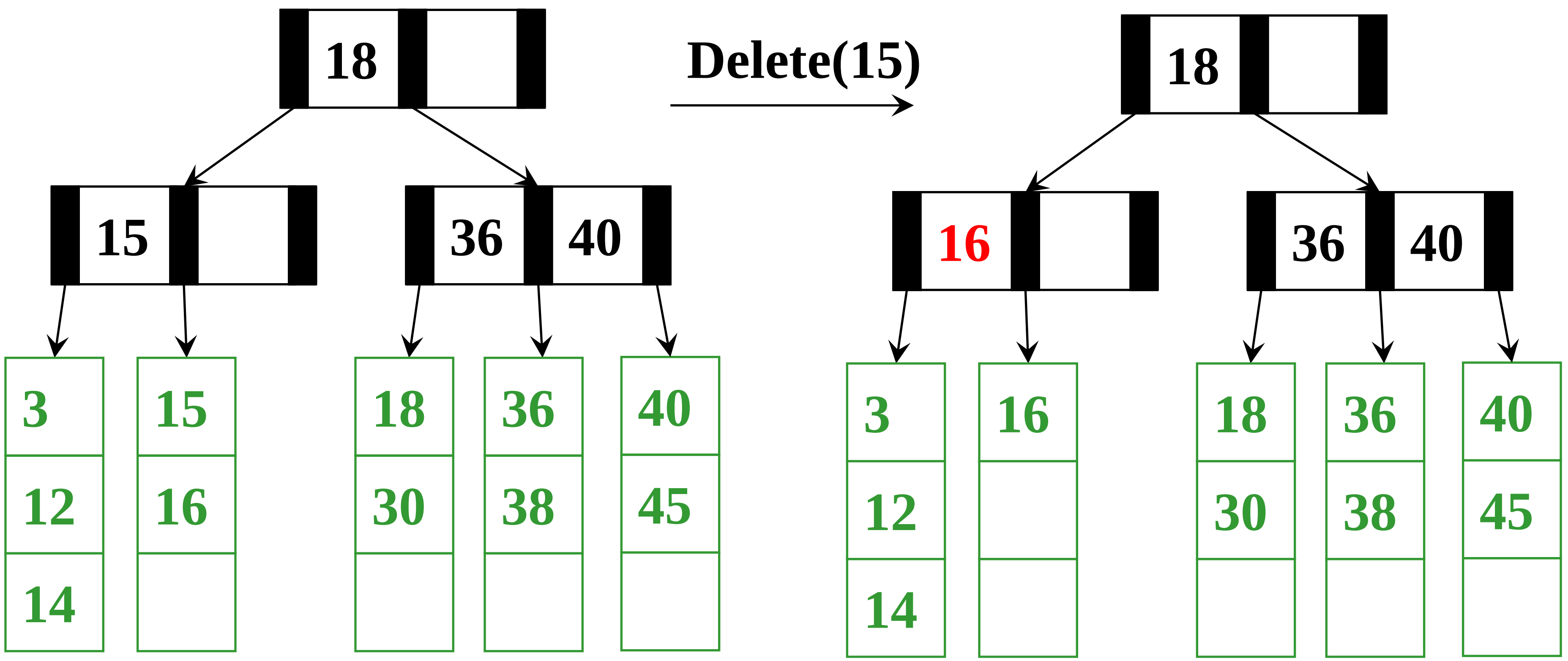
- So repeat step 3 up the tree until a node doesn't overflow
- If the root overflows, make a new root with two children

And Now for Deletion…



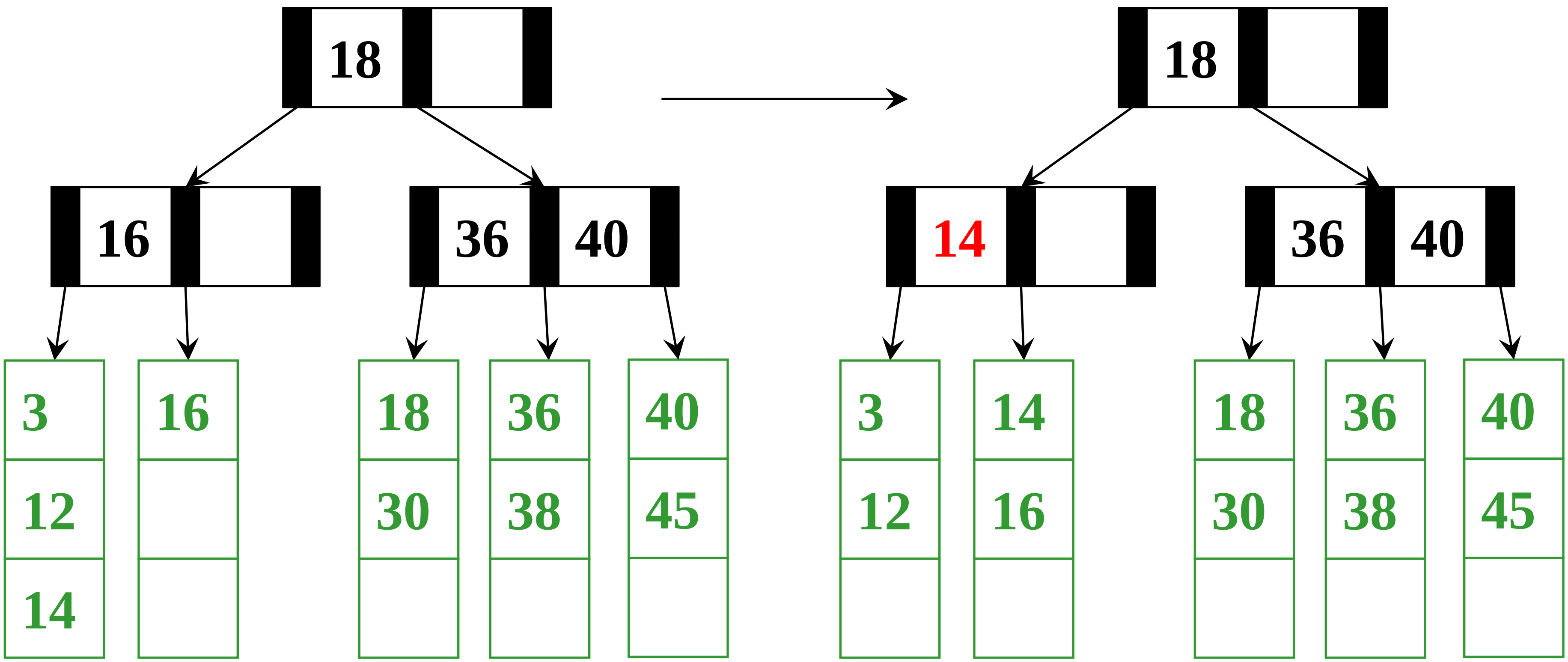
Easy case: Leaf still has enough data; just remove

M = 3 L = 3



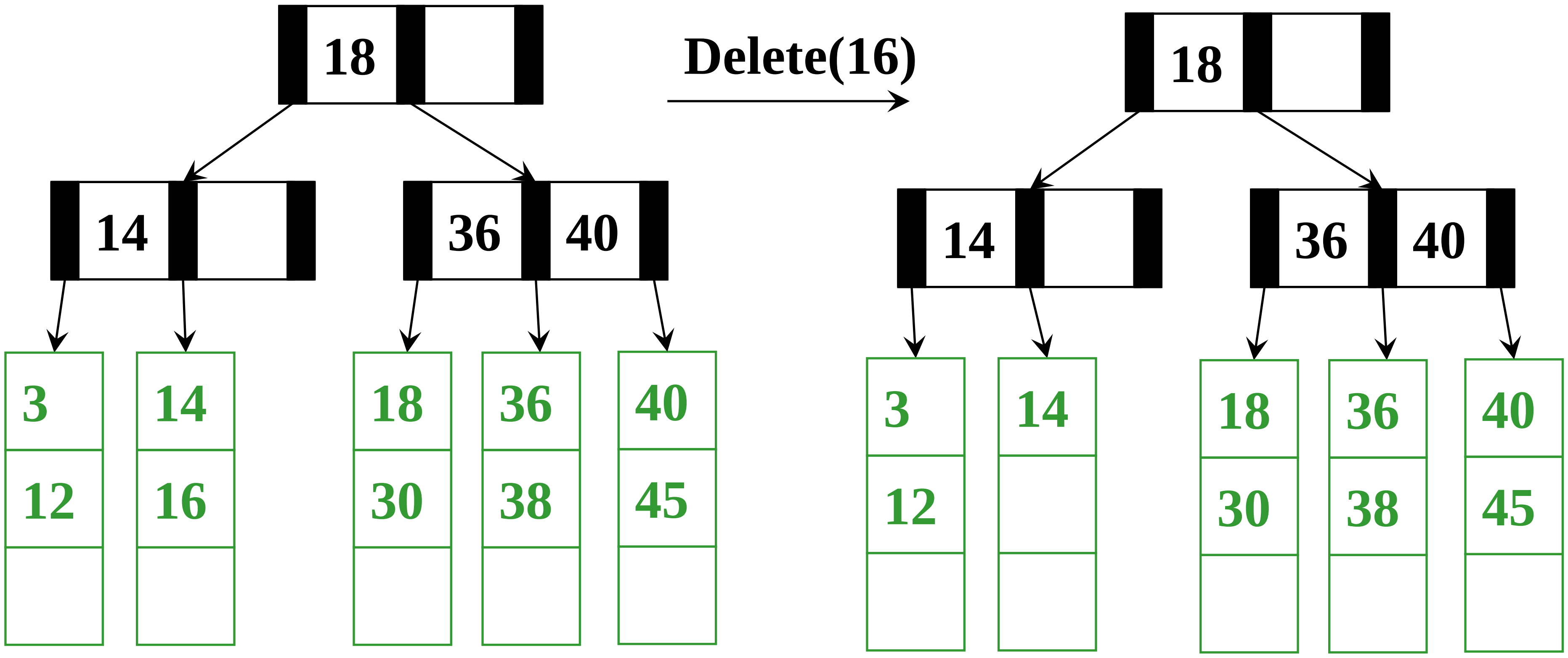
Underflow in the leaf

M = 3 L = 3



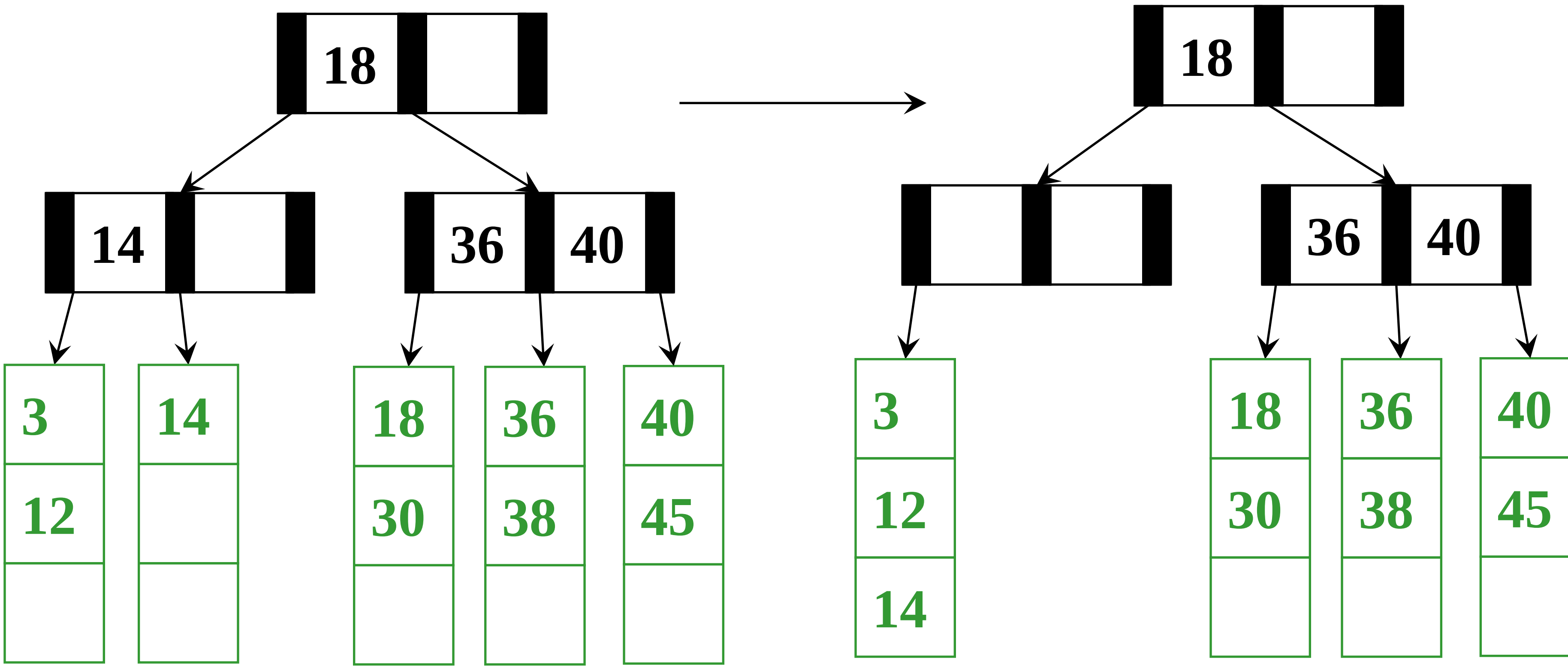
Adoption: grab a data item from neighboring leaf

M = 3 *L* = 3



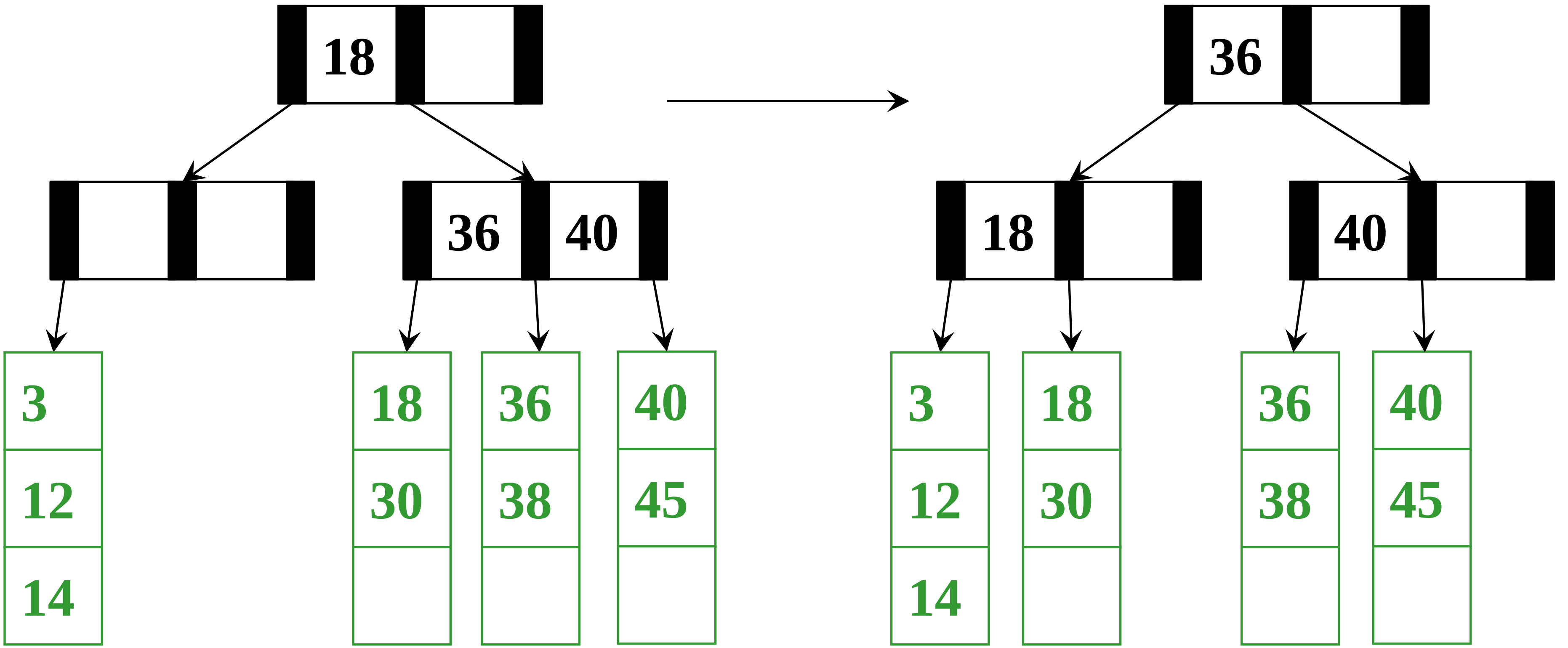
Uh-oh, neighbors at their minimum!

`M = 3 L = 3`



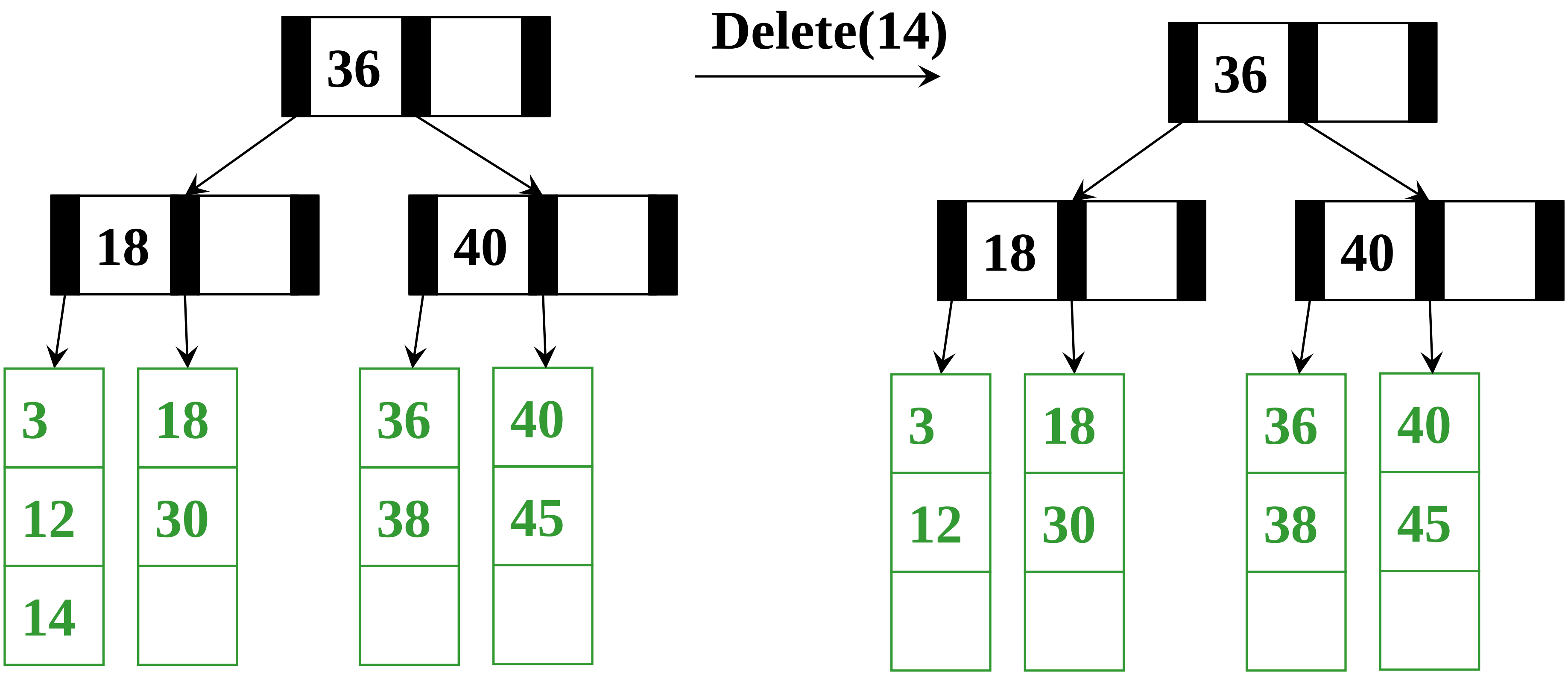
Merge the two nodes together. This causes underflow in the parent

M = 3 *L* = 3



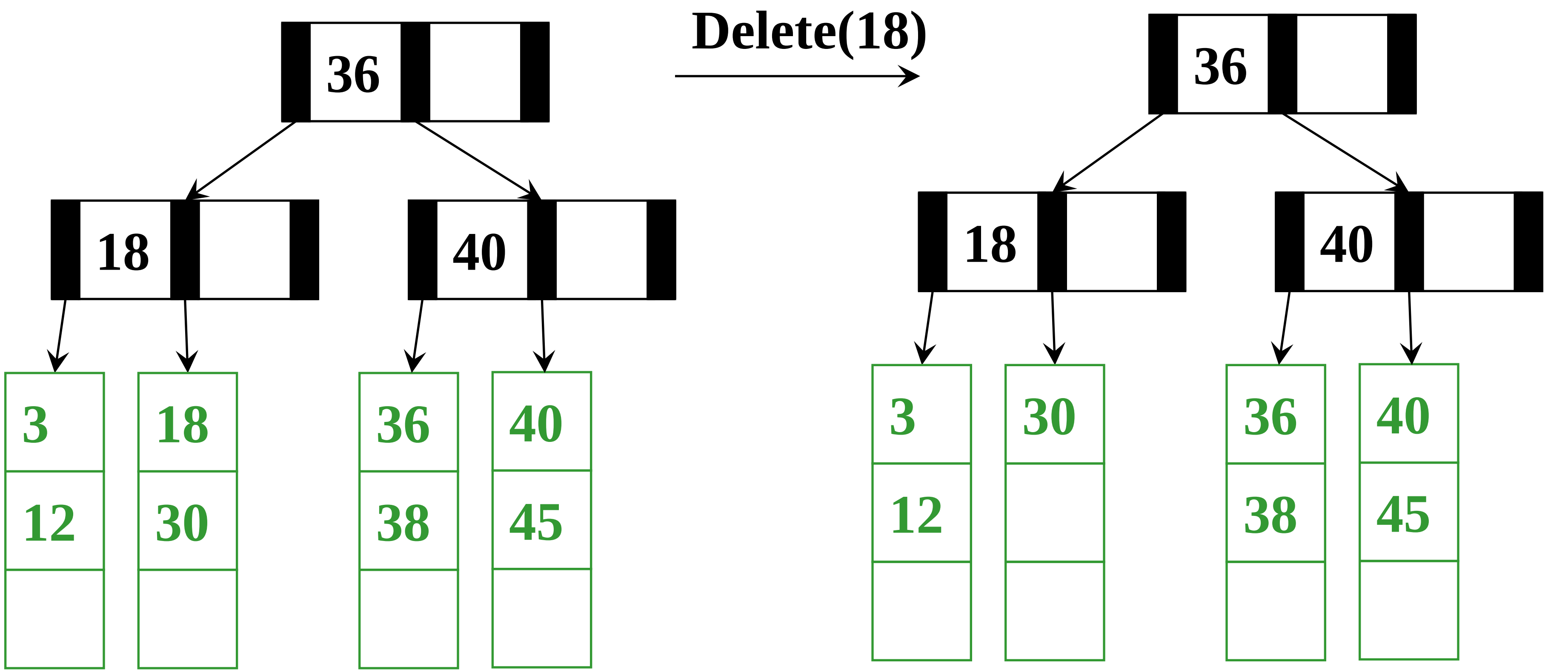
Now grab a leaf node from parent's neighbor

M = 3 L = 3



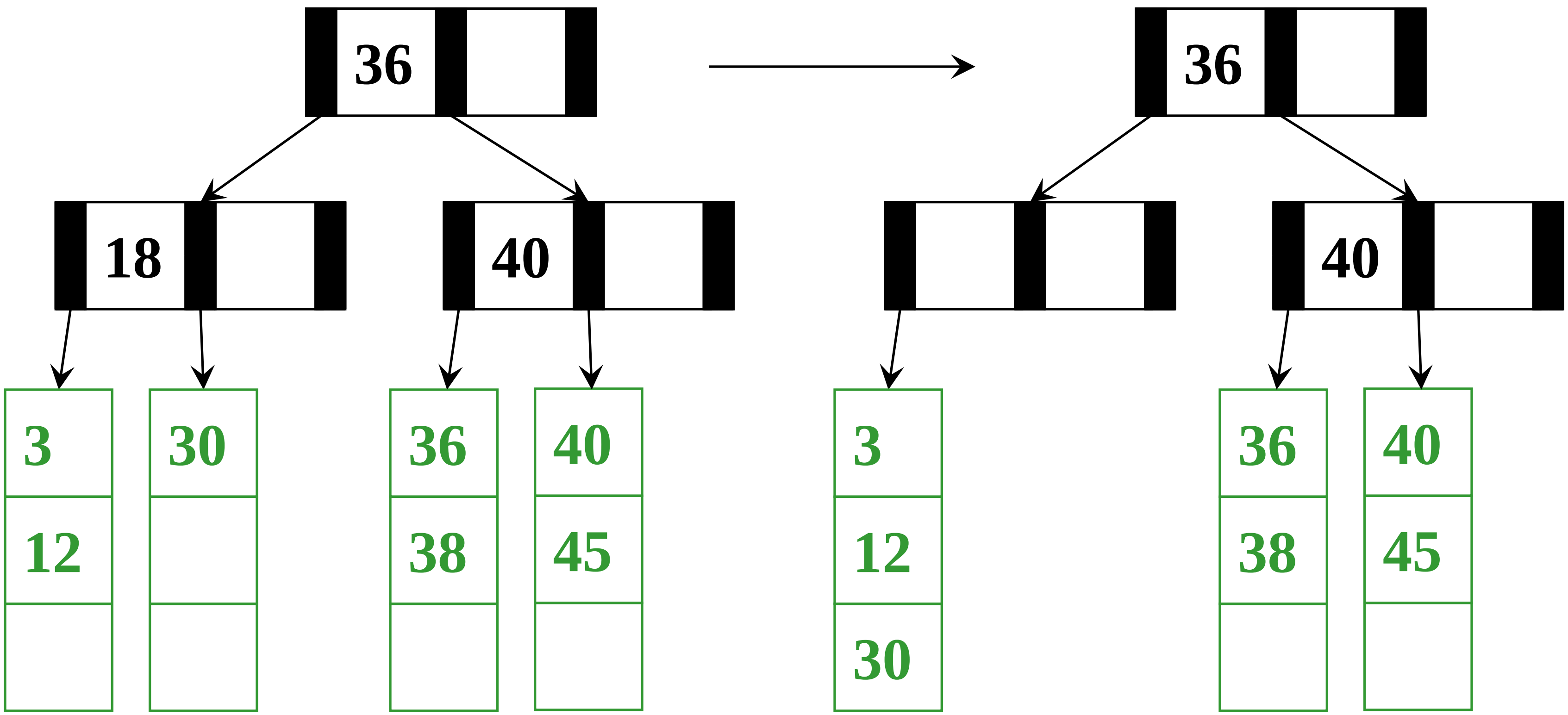
Easy case again

`M = 3 L = 3`



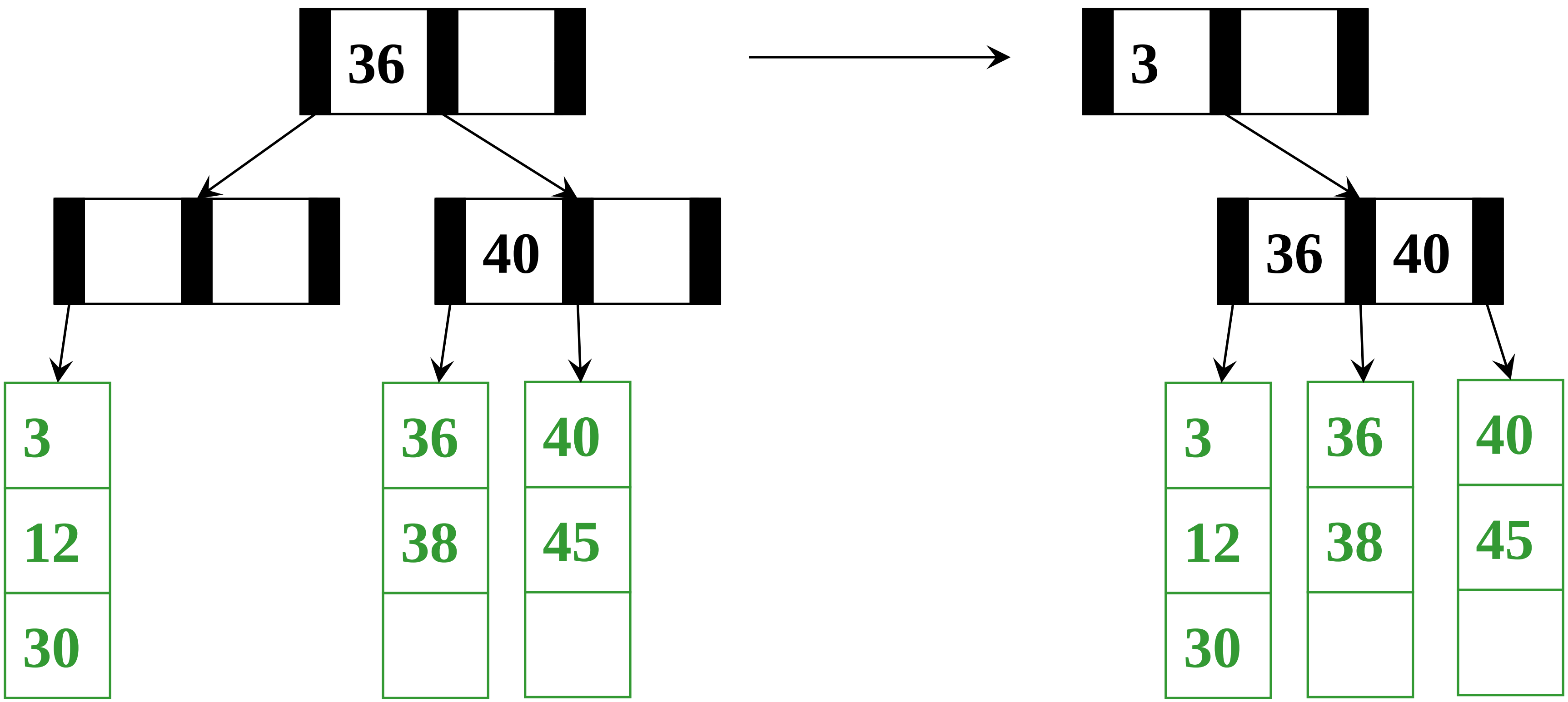
Leaf underflow; no neighbors with enough to steal from…

$M = 3$ $L = 3$



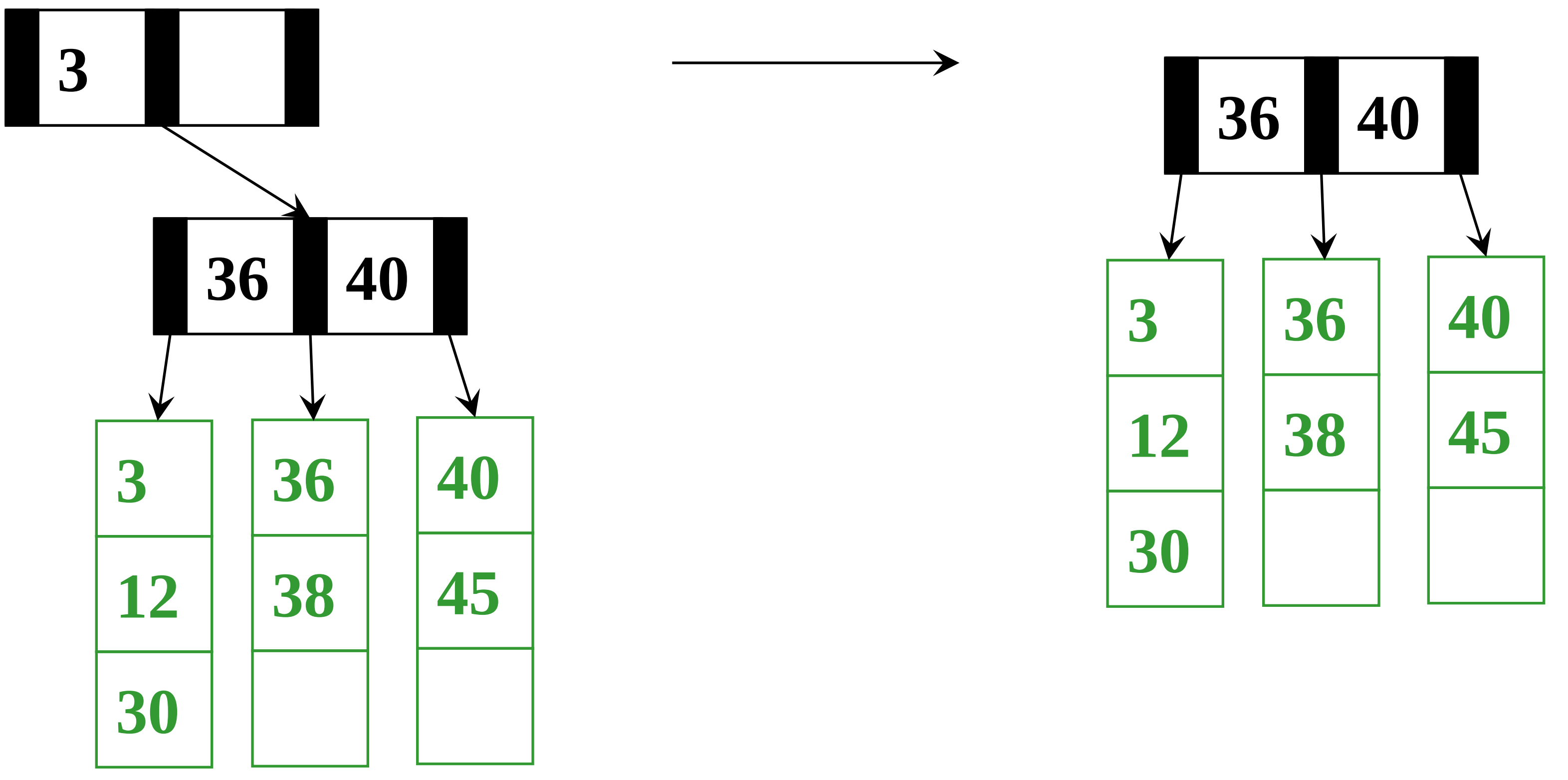
Merge leaves…

$M = 3$ $L = 3$



Can't steal leaf from parent's neighbor; too few leaves. Instead merge parent w/ parent's neighbor

`M = 3 L = 3`



Which causes an underflow in root; replace root

$M = 3$ $L = 3$

Deletion Algorithm

1. Remove the data from its leaf

2. If the leaf now has $\lceil \mathbf{\textit{L}/2} \rceil$ **– 1**, *underflow!*
   - If a neighbor has > $\lceil \mathbf{\textit{L}/2} \rceil$ 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 $\lceil \mathbf{\textit{M}/2} \rceil$ **– 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

Efficiency of 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: $O(M \log_M n)$

Worst-case Delete: $O(L + M \log_M n)$

But it's not that bad:

- Merges are not that common
- Remember disk accesses were 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: $O(M \log_M n)$

Aside: Limitations of B-Trees in Java

For most of our data structures, we have encouraged writing high-level, reusable code, such as in Java with generics

It is worth knowing enough about "how Java works" to understand why this is probably a bad idea for B-Trees

- Assuming our goal is efficient number of disk accesses
- Java has many advantages, but it wasn't designed for this
- If you just want a balanced tree with worst-case logarithmic operations, no problem

The problem is extra *levels of indirection*…

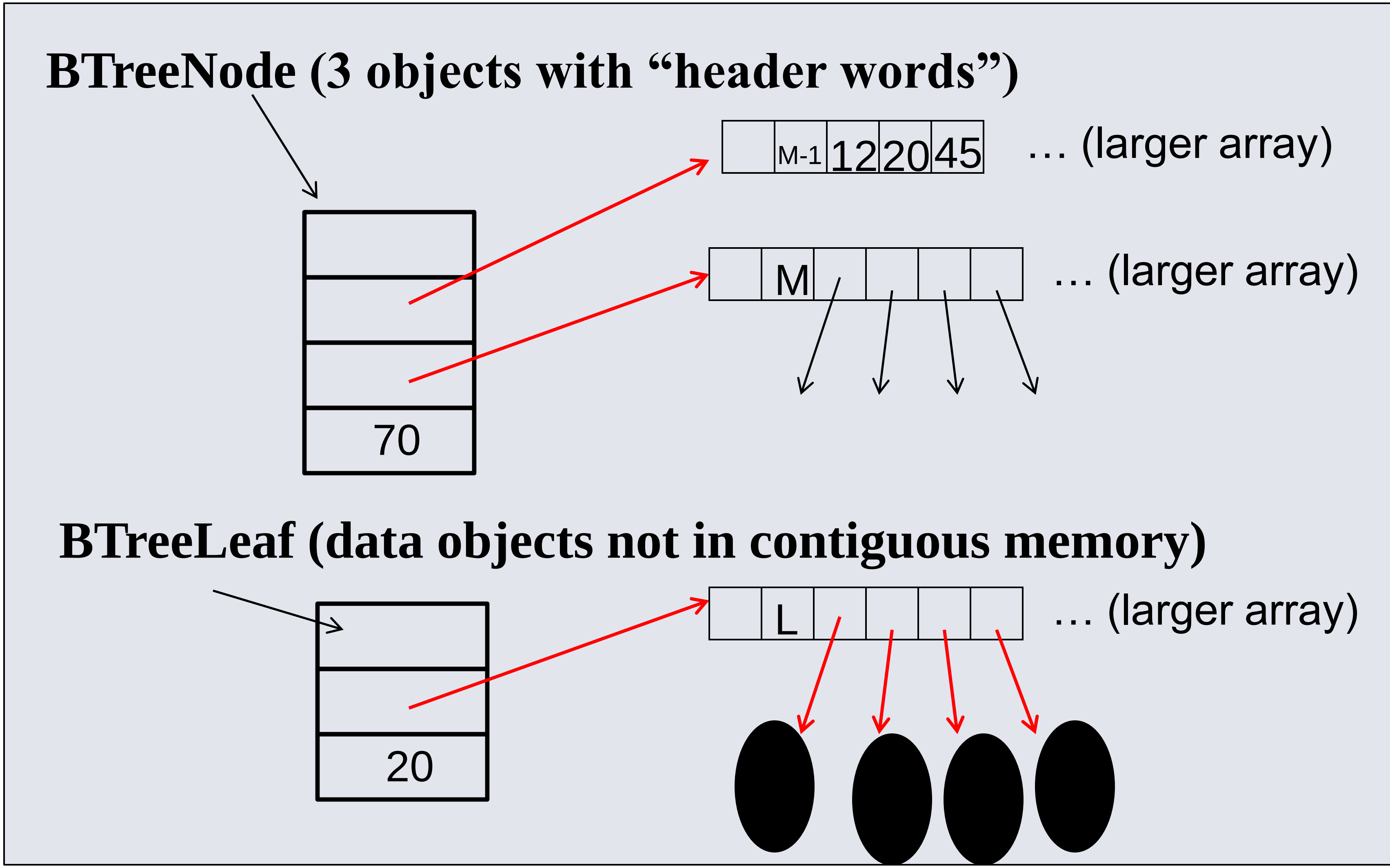
One approach

Even if we assume data items have **`int`** keys, you cannot get the data representation you want for “really big data”

```
interface Keyed<E> {
  int key(E);
}
class BTreeNode<E implements Keyed<E>> {
  static final int M = 128;
  int[] keys = new int[M-1];
  BTreeNode<E>[] children = new BTreeNode[M];
  int numChildren = 0;
  …
}
class BTreeLeaf<E> {
  static final int L = 32;
  E[] data = (E[])new Object[L];
  int numItems = 0;
  …
}
```

What that looks like

All the red references indicate unnecessary indirection



The moral

- The whole idea behind B trees was to keep related data in contiguous memory
- But that's "the best you can do" in Java
  - Again, the advantage is generic, reusable code
  - But for your performance-critical web-index, not the way to implement your B-Tree for terabytes of data
- C# may have better support for "flattening objects into arrays"
  - C and C++ definitely do
- Levels of indirection matter!

Conclusion: Balanced Trees

- *Balanced* trees make good dictionaries because they guarantee logarithmic-time **`find`**, **`insert`**, and **`delete`**
  - Essential and beautiful computer science
  - But only if you can maintain balance within the time bound
- 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
- Other great balanced trees (see text for details)
  - Splay trees: self-adjusting; amortized guarantee; no extra space for height information
  - Red-black trees: all leaves have depth within a factor of 2