CSE 332: Data Structures & Parallelism

Lecture 9: B Trees

Ruth Anderson

Winter 2018

Today

- Finish up AVL Trees
- The Memory Hierarchy and you (briefly)
- Dictionaries
  - B-Trees

Now what?

- We have a data structure for the dictionary ADT (AVL tree) that has worst-case $O(\texttt{log}\ n)$ behavior
  - One of several interesting/fantastic balanced-tree approaches

- We are about to learn another balanced-tree approach: B Trees

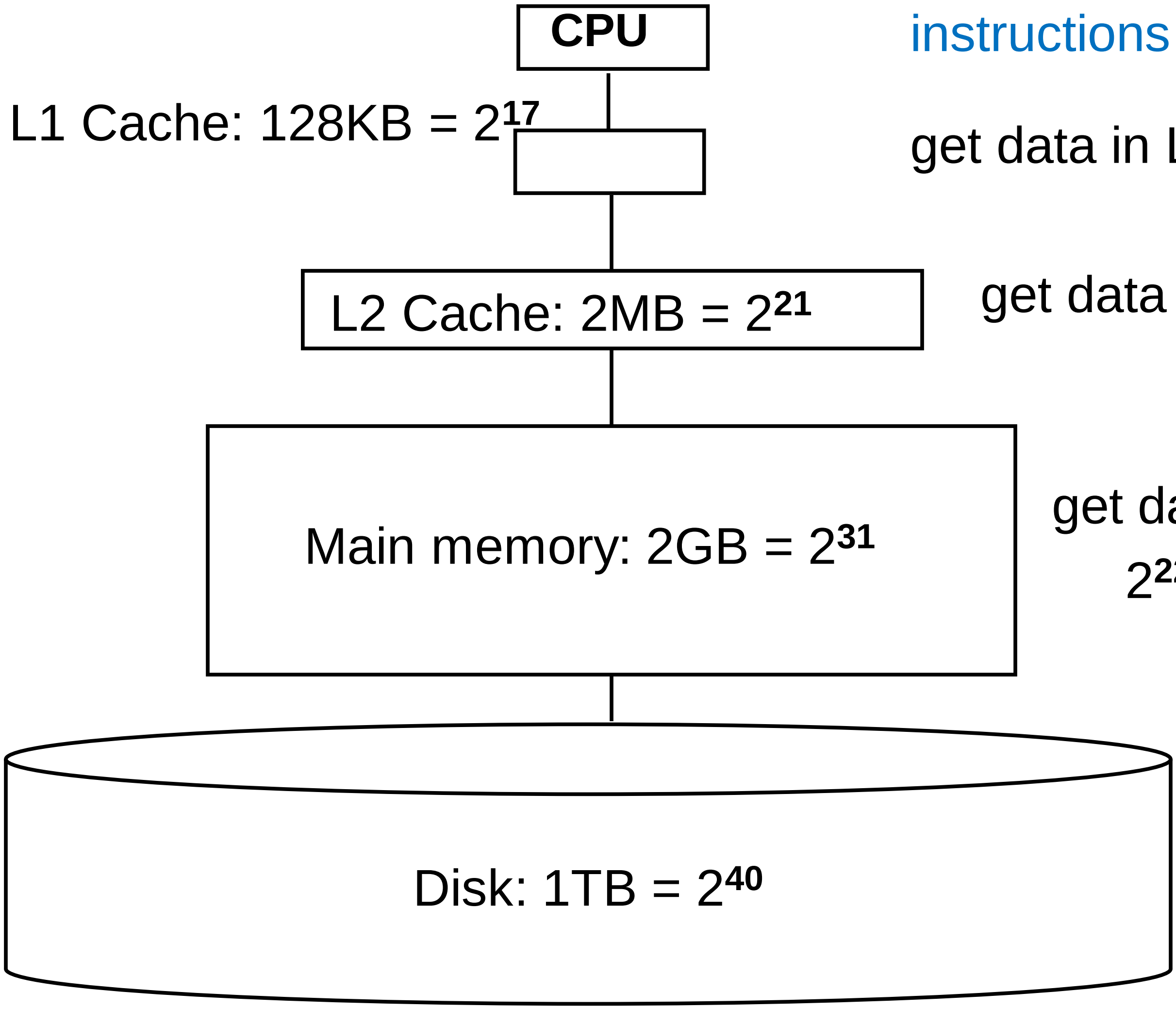
- First, to motivate why B trees are better for really large dictionaries (say, over 1GB = 2^{30} bytes), need to understand some ***memory-hierarchy basics***
  - Don't always assume "every memory access has an unimportant *O(1)* cost"
  - Learn more in CSE351/333/471, focus here on relevance to data structures and efficiency

Why do we need to know about the memory hierarchy?

- One of the assumptions that Big-Oh makes is that all operations take the same amount of time.
- Is that really true?

A typical hierarchy

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



instructions (e.g., addition): 2^{30}/sec

get data in L1: 2^{29}/sec = 2 instructions

get data in L2: 2^{25}/sec = 30 instructions

get data in main memory:

2^{22}/sec = 250 instructions

get data from "new place" on disk:

2^{7}/sec =8,000,000 instructions

Morals

It is much faster to do:	Than:
5 million arithmetic ops	1 disk access
2500 L2 cache accesses	1 disk access
400 main memory accesses	1 disk access

Why are computers built this way?

- Physical realities (speed of light, closeness to CPU)
- Cost (price per byte of different technologies)
- Disks get much bigger not much faster
  - Spinning at 7200 RPM accounts for much of the slowness and unlikely to spin faster in the future
- Speedup at higher levels (e.g. a faster processor) makes lower levels *relatively slower*
- Later in the course: more than 1 CPU!

“Fuggedaboutit”, usually

The hardware automatically moves data into the caches from main memory for you

- Replacing items already there
- So algorithms much faster if “data fits in cache” (often does)

Disk accesses are done by software (e.g., ask operating system to open a file or database to access some data)

So most code “just runs” but sometimes it’s worth designing algorithms / data structures with knowledge of memory hierarchy

- And when you do, you often need to know one more thing…

How does data move up the hierarchy?

- Moving data up the memory hierarchy is slow because of *latency* (think distance-to-travel)
  - Since we're making the trip anyway, may as well carpool
    - Get a block of data in the same time it would take to get a byte
  - Sends *<u>nearby memory</u>* because:
    - It's easy
    - And likely to be asked for soon (think fields/arrays)

Spatial Locality

- Side note: Once a value is in cache, may as well keep it around for awhile; accessed once, *<u>a particular value</u>* is more likely to be accessed again in the near future (more likely than some random other value)

Temporal locality

Locality

Temporal Locality (locality in time) – If an address is referenced, ***it*** will tend to be referenced again soon.

Spatial Locality (locality in space) – If an address is referenced, ***addresses that are close by*** will tend to be referenced soon.

Arrays vs. Linked lists

- Which has the potential to best take advantage of spatial locality?

Block/line size

- The amount of data moved from disk into memory is called the "**block**" size or the "**page**" size
  - Not under program control
- The amount of data moved from memory into cache is called the cache "**line**" size
  - Not under program control

Connection to data structures

- An array benefits more than a linked list from block moves
  - Language (e.g., Java) implementation can put the list nodes anywhere, whereas array is typically contiguous memory
- Suppose you have a queue to process with 2^{23} items of 2^{7} bytes each on disk and the block size is 2^{10} bytes
  - An **array** implementation needs 2^{20} disk accesses
    - If "perfectly streamed", > 4 seconds
    - If "random places on disk", 8000 seconds (> 2 hours)
  - A **list** implementation in the worst case needs 2^{23} "random" disk accesses (> 16 hours) – probably not that bad

- Note: "array" doesn't necessarily mean "good"
  - Binary heaps "make big jumps" to percolate (different block)

BSTs?

- Looking things up in balanced binary search trees is $O(\texttt{log}\ n)$, so even for $n = 2^{39}$ (512GB) we need not worry about minutes or hours

- Still, number of disk accesses matters:
  - Pretend for a minute we had an AVL tree of height 55
  - The total number of nodes could be?__________
  - Most of the nodes will be on disk: the tree is shallow, but it is still many gigabytes big so the entire *tree* cannot fit in memory
    - Even if memory holds the first 25 nodes on our path, we still potentially need 30 disk accesses if we are traversing the entire height of the tree.

Note about numbers; moral

- **Note:** All the numbers in this lecture are “ballpark” “back of the envelope” figures

- **Moral**: Even if they are off by, say, a factor of 5, the moral is the same:

  If your data structure is mostly on disk, you want to minimize disk accesses

- A better data structure in this setting would exploit the block size and relatively fast memory access to ***avoid disk accesses***…

Trees as Dictionaries

(N= 10 million) [Example from Weiss]

In worst case, each node access is a disk access, number of accesses:

<u># Disk accesses</u>

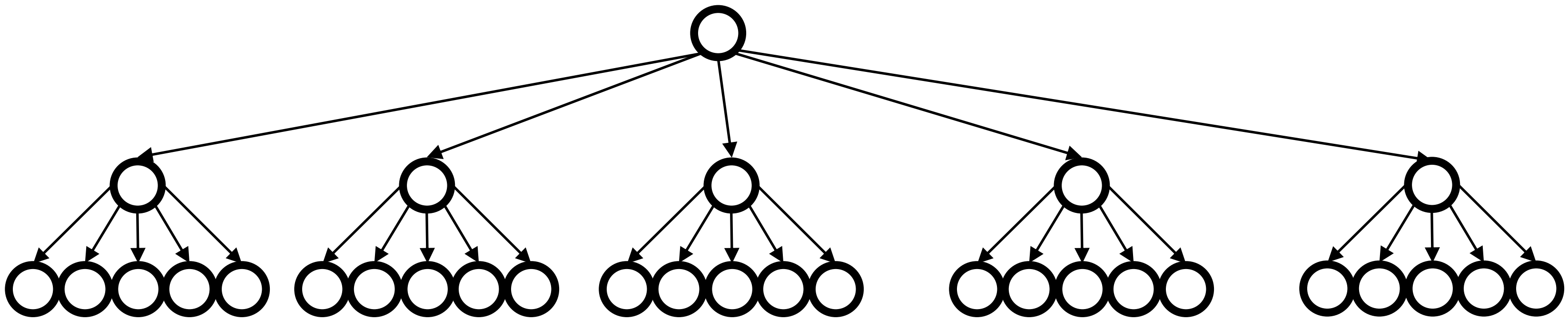
- BST
- AVL
- B Tree

Our goal

- **Problem**: A dictionary with so much data *most of it is on disk*
- **Desire**: A balanced tree (logarithmic height) that is even shallower than AVL trees so that we can minimize disk accesses and exploit disk-block size
- **A key idea**: Increase the branching factor of our tree

M-ary Search Tree

- Build some sort of search tree with branching factor *M*:
  - Have an array of sorted children (`Node[]`)
  - Choose *M* to fit snugly into a disk block (1 access for array)

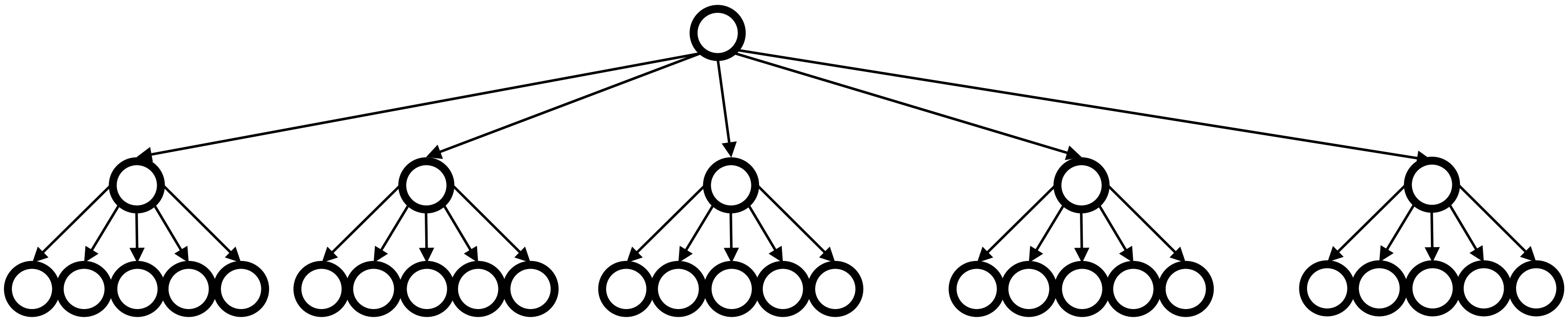


Perfect tree of height h has $(M^{h+1}-1)/(M-1)$ nodes (textbook, page 4)

What is the **height** of this tree?

What is the worst case running time of **find**?

M-ary Search Tree



- # hops for **`find?`**
  - If we have a balanced M-ary tree:
  - Approx. $\log_M n$ hops instead of $\log_2 n$ (for balanced BST)
  - Example: $M = 256$ (=2^8) and $n = 2^{40}$ that's 5 hops instead of 40 hops
- Sounds good, but how do we decide which branch to take?
  - Binary tree: Less than/greater than node value?
  - M-ary: In range 1? In range 2? In range 3?... In range M?
- Runtime of **`find`** if balanced: $O(\log_2 M \log_M n)$
  - $\log_M$ n is the height we traverse.
  - $\log_2 M$: At each step, find the correct child branch to take using binary search among the M options!

Questions about M-ary search trees

- What should the **order** property be?
- How would you **rebalance** (ideally without more disk accesses)?
- Storing **real data** at inner-nodes (like we do in a BST) seems kind of wasteful…
  - To access the node, will have to load the **data** from disk, *even though most of the time we won't use it!!*
  - Usually we are just "passing through" a node on the way to the value we are actually looking for.

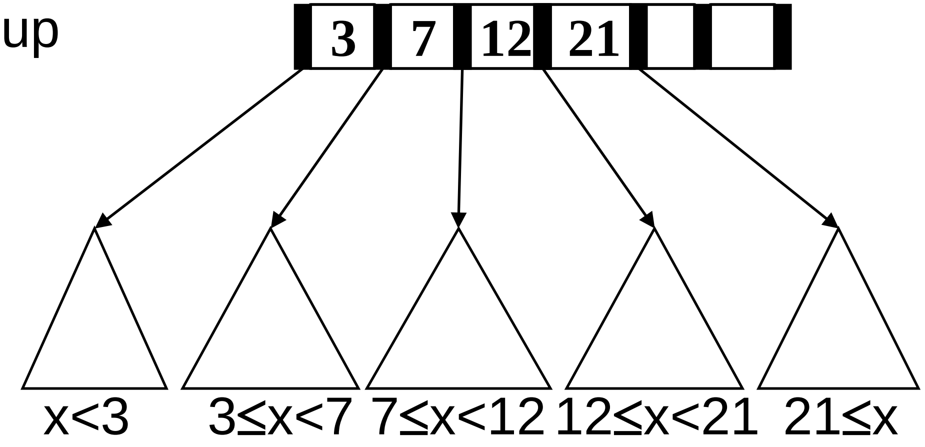
So let's use the branching-factor idea, but for a different kind of balanced tree:

- **Not** a binary *search tree*
- But still logarithmic height for any $M > 2$

B+ Trees (we and the book say "B Trees")

- Two types of nodes: **internal nodes** & **leaves**
- Each **internal node** has room for up to *M-1* keys and *M* children
  - No other data; all data at the leaves!
- Order property:

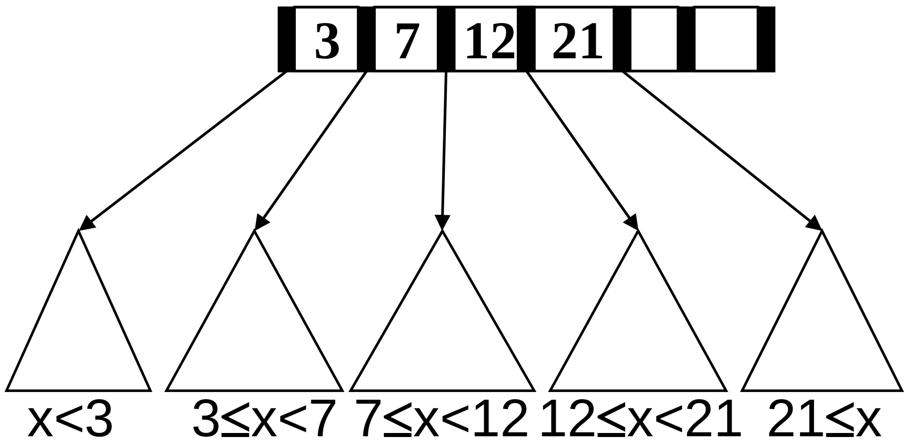
  Subtree **between** keys *a* and *b* contains only data that is $\geq$ ***a*** and $<$ ***b*** (notice the $\geq$)
- **Leaf** nodes have up to *L* sorted data items
- As usual, we'll ignore the "along for the ride" data in our examples
  - Remember no data at non-leaves



Remember:

- **Leaves** store data
- **Internal nodes** are 'signposts'

Find



- Different from BST in that we *<u>don't store data at 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 the (up to) L data items

- But to get logarithmic running time, we need a balance condition…

Structure Properties

- **Root** (special case)
  - If tree has $\leq L$ items, root is a leaf (occurs when starting up, otherwise unusual)
  - Else has between 2 and M children

- **Internal nodes**
  - Have between $\lceil M/2 \rceil$ and M children, i.e., at least half full

- **Leaf nodes**
  - All leaves at the same depth
  - Have between $\lceil L/2 \rceil$ and L data items, i.e., at least half full

Any $M > 2$ and L will work, but:

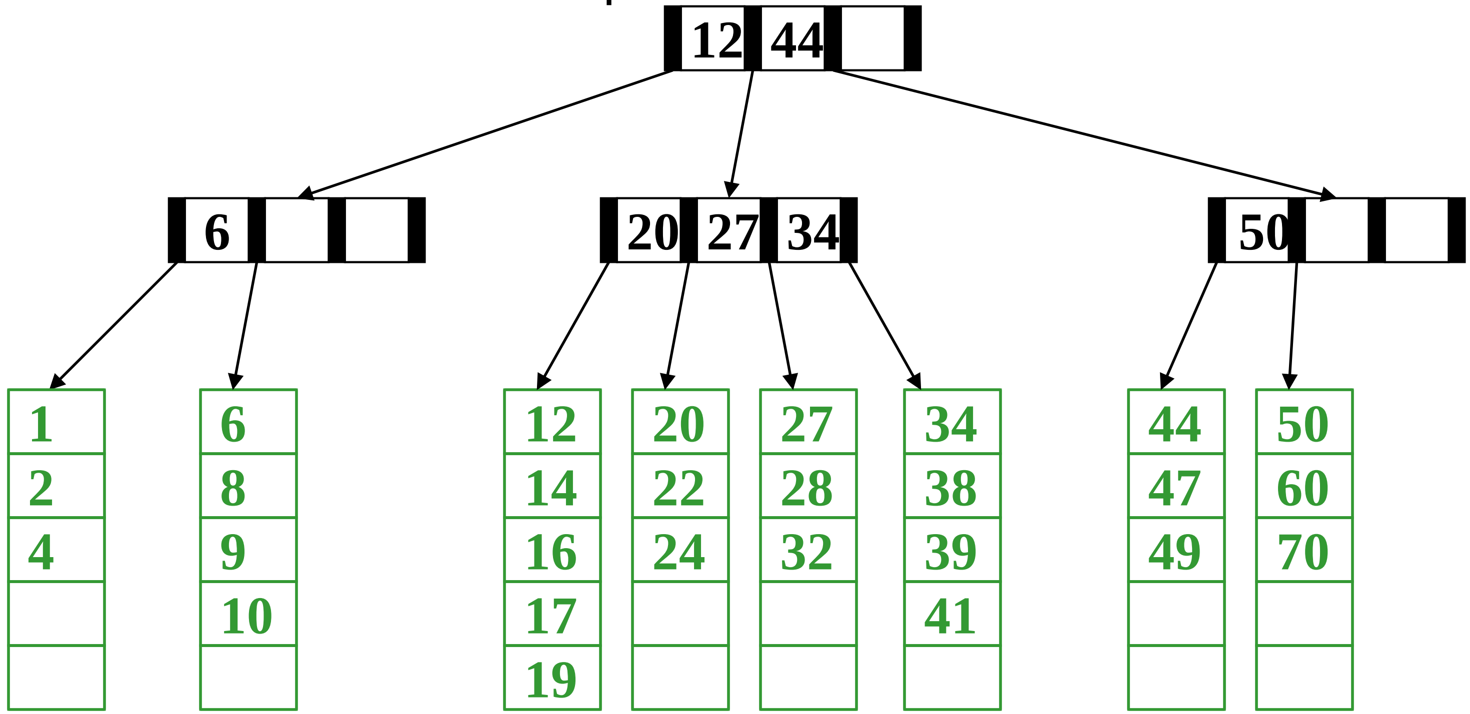
We pick M and L ***based on disk-block size***

Note on notation: Inner nodes drawn horizontally, leaves vertically to distinguish. Include empty cells

Example

Suppose *M*=4 (max # pointers in **internal node**)
and *L*=5 (max # data items at **leaf**)

- All **internal nodes** have at least 2 children
- All **leaves** have at least 3 data items (only showing keys)
- All **leaves** at same depth



Balanced enough

Not hard to show height *h* is logarithmic in number of data items *n*

- Let *M* > 2 (if *M* = 2, then a list tree is legal – no good!)
- Because all nodes are at least half full (except root may have only 2 children) and all leaves are at the same level, the minimum number of data items *n* for a height *h>0* tree is…

$$n \geq \underbrace{2 \lceil M/2 \rceil^{h-1}}_{\text{minimum number of leaves}} \underbrace{\lceil L/2 \rceil}_{\text{minimum data per leaf}}$$

Example: B-Tree vs. AVL Tree

Suppose we have 100,000,000 items

- Maximum height of AVL tree?

- Maximum height of B tree with M=128 and L=64?

Example: B-Tree vs. AVL Tree

Suppose we have 100,000,000 items

- **Maximum height of AVL tree**?
  - Recall S(h) = 1 + S(h-1) + S(h-2)
  - lecture8.xlsx reports: **37**

- **Maximum height of B tree** with *M*=128 and *L*=64?
  - Recall $(2 \lceil M/2 \rceil^{h-1}) \lceil L/2 \rceil$
  - lecture9.xlsx reports: **5** (and 4 is more likely)
  - Also not difficult to compute via algebra

Disk Friendliness

What makes B trees so disk friendly?

- Many keys stored in one **internal node**
  - All brought into memory in one disk access
    - *IF* we pick *M* wisely
  - Makes the binary search over *M*-1 keys totally worth it (insignificant compared to disk access times)

- **Internal nodes** contain only keys
  - Any **`find`** wants only one data item; wasteful to load unnecessary items with internal nodes
  - So only bring one **leaf** of data items into memory
  - Data-item size doesn't affect what *M* is

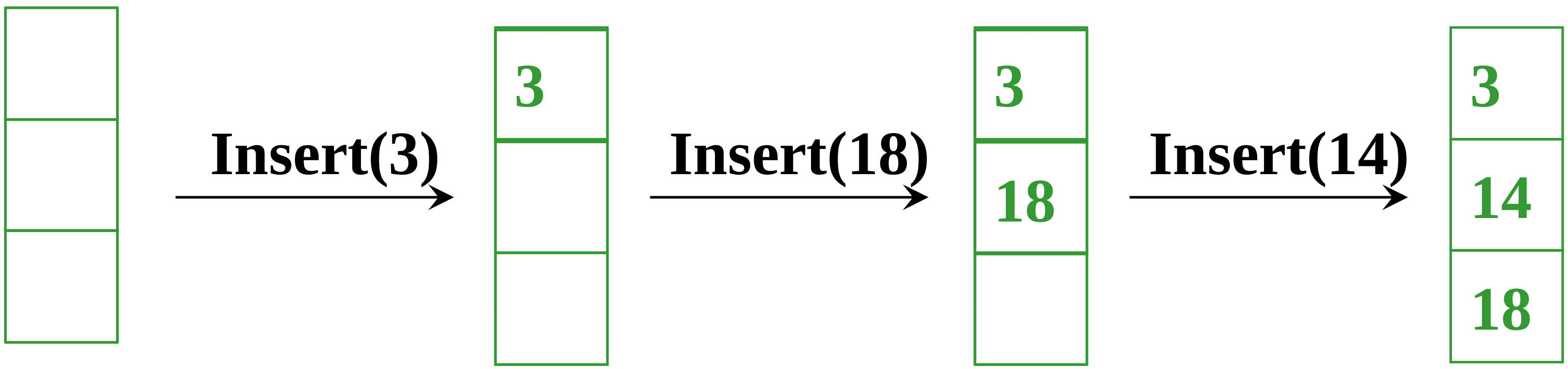
Maintaining balance

- So this seems like a great data structure (and it is)

- But we haven't implemented the other dictionary operations yet
  - **`insert`**
  - **`delete`**

- As with AVL trees, the hard part is maintaining structure properties
  - Example: for **`insert`**, there might not be room at the correct leaf

Building a B-Tree (insertions)

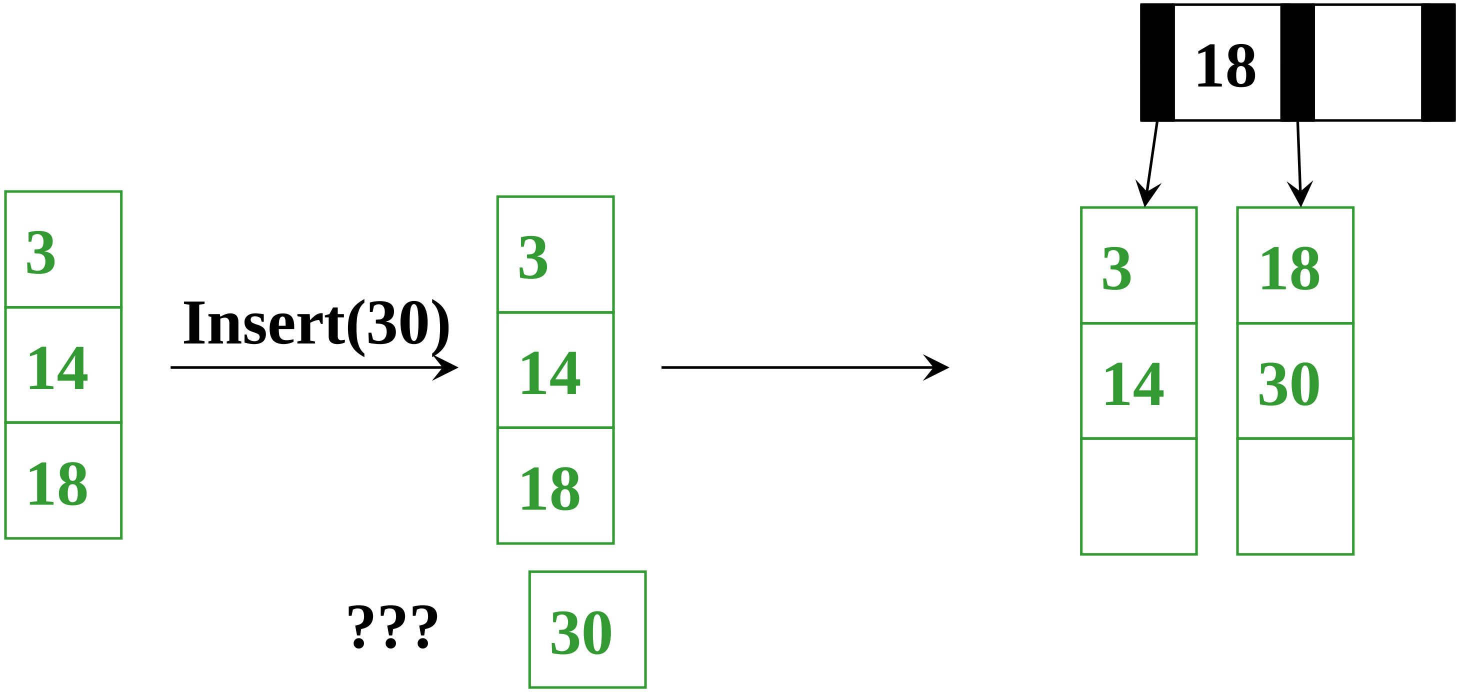


The empty B-Tree (the root will be a leaf at the beginning)

Just need to keep data in order

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

$M = 3$ $L = 3$



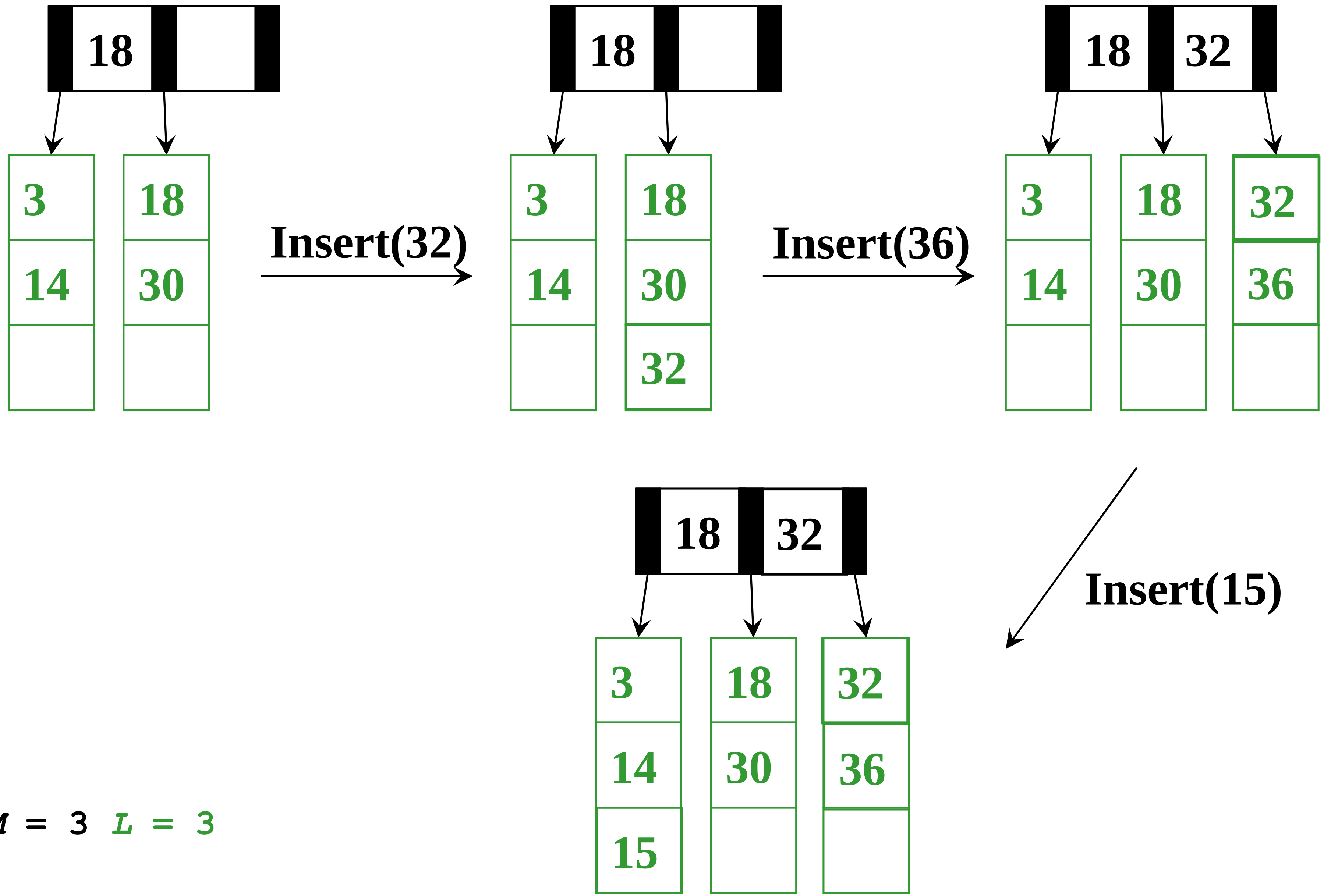
•When we ‘overflow’ a leaf, we split it into 2 leaves

•Parent gains another child

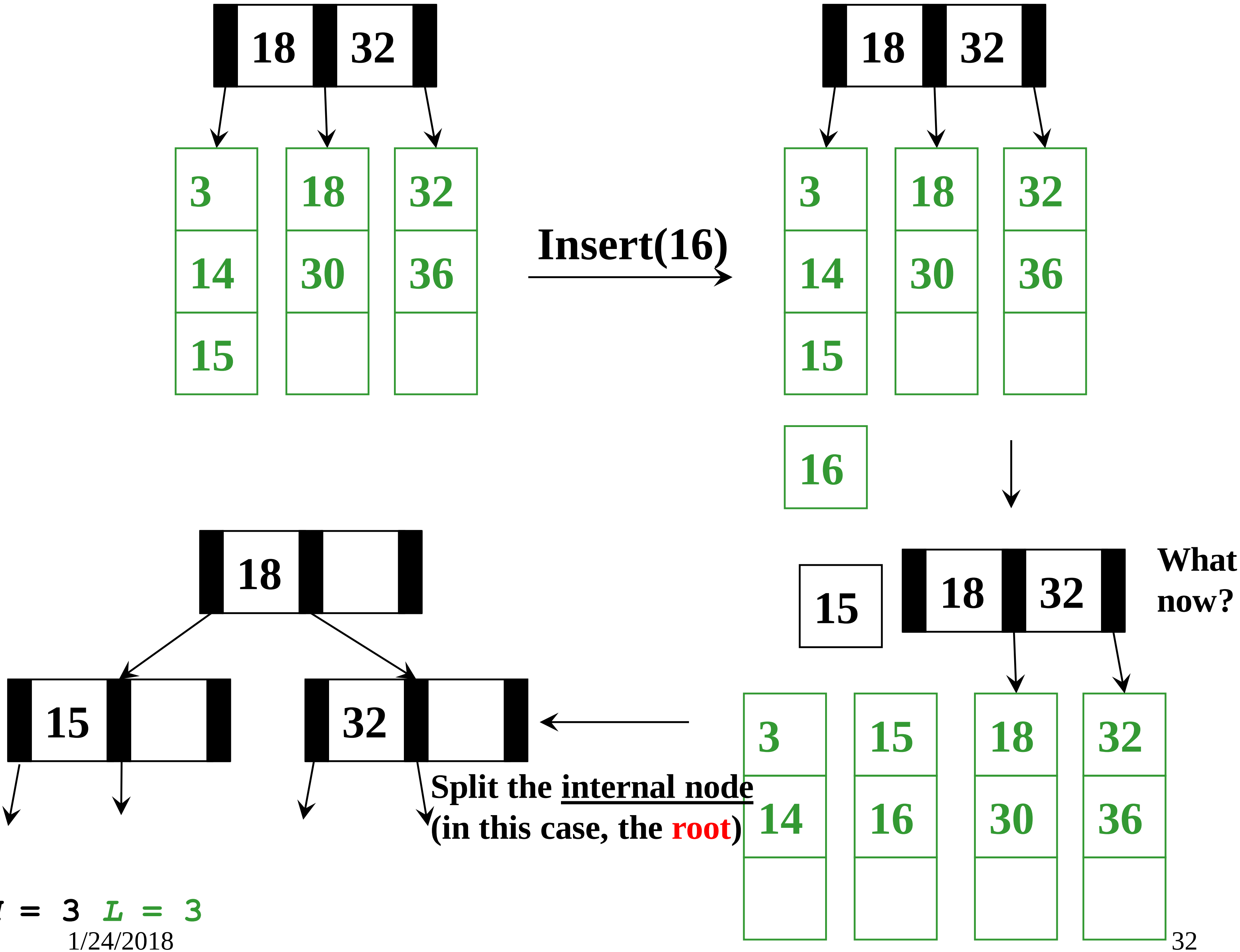
•If there is no parent (like here), we create one; how do we pick the key shown in it?

- •Smallest element in right tree

Split leaf again

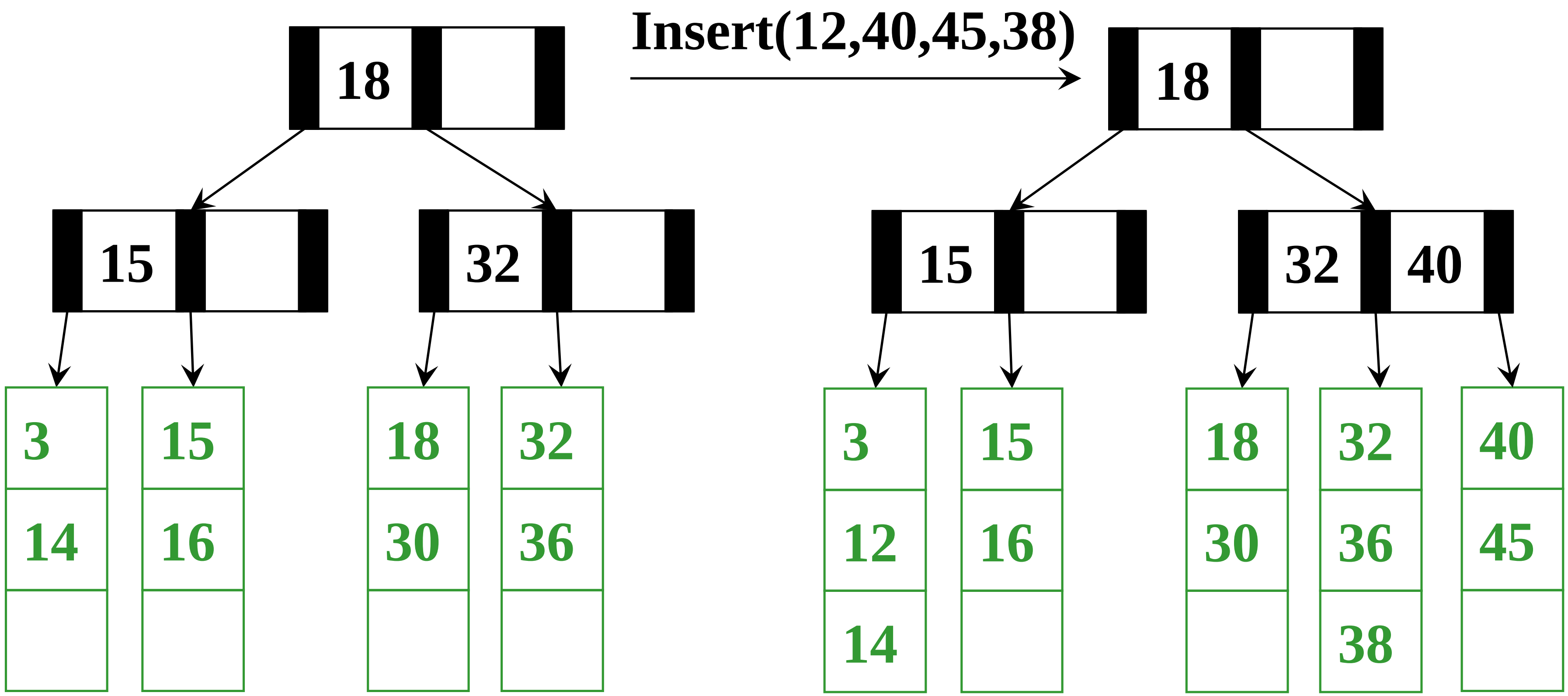


M = 3 **L = 3**



Insert(16)
What now?
Split the internal node
(in this case, the root)

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 data in its **leaf** in sorted order

2. If the **leaf** now has L+1 items, *overflow!*
   - Split the **leaf** into two nodes:
     - Original **leaf** with $\lceil (L+1)/2 \rceil$ smaller items
     - New **leaf** with $\lfloor (L+1)/2 \rfloor = \lceil L/2 \rceil$ larger items
   - Attach the new child to the parent
     - Adding new key to parent in sorted order

3. If step (2) caused the parent to have M+1 children, *overflow!*
   - …

Insertion algorithm continued

3. If an **internal node** has M+1 children
   - Split the **node** into **two nodes**
     - Original **node** with $\lceil (M+1)/2 \rceil$ smaller items
     - New **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger items
   - Attach the new child to the parent
     - Adding new key to parent in sorted order

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
  - This is the only case that increases the tree height

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

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

But it's not that bad:

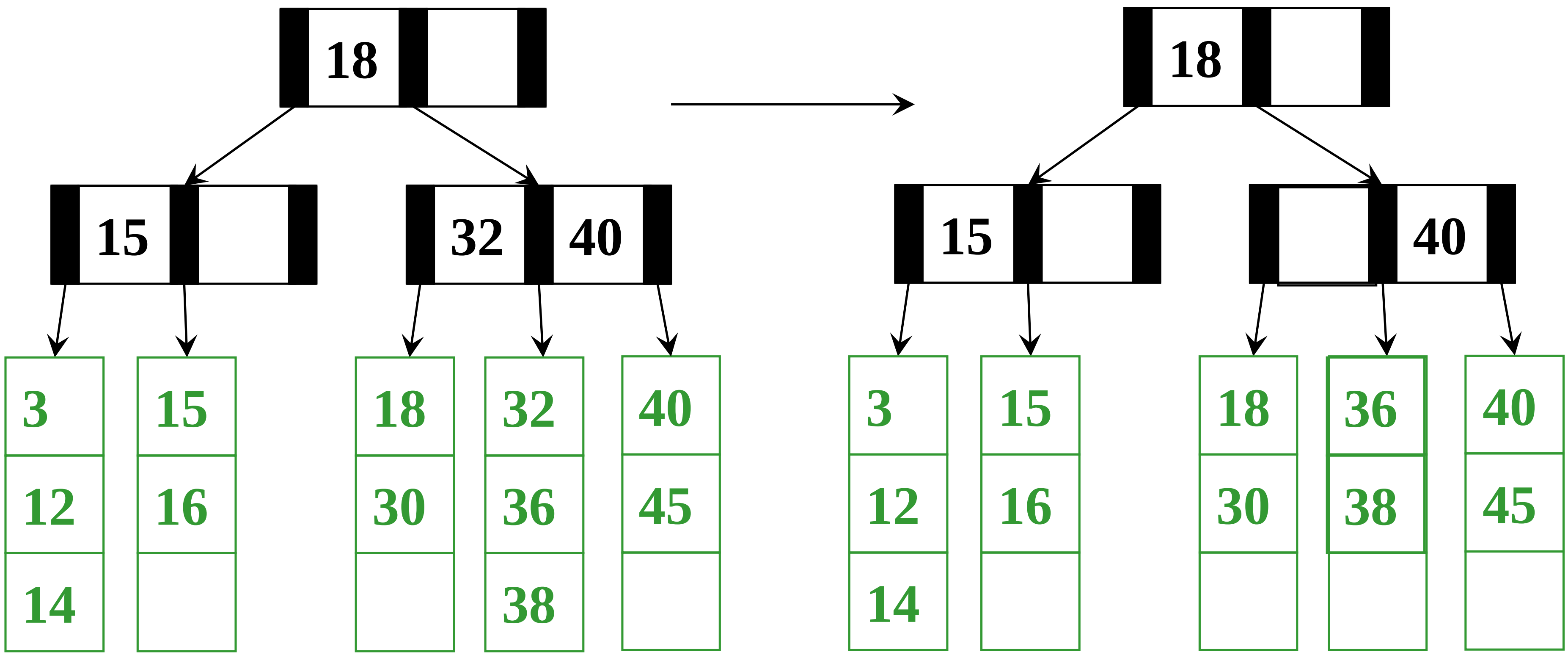
- Splits are not that common (only required when a node is FULL, M and L are likely to be large, and after a split, will be half empty)
- Splitting the **root** is extremely rare
- Remember disk accesses were the name of the game: $O(\log_M n)$

B-Tree Reminder: Another dictionary

- Before we talk about deletion, just keep in mind 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

And Now for Deletion…

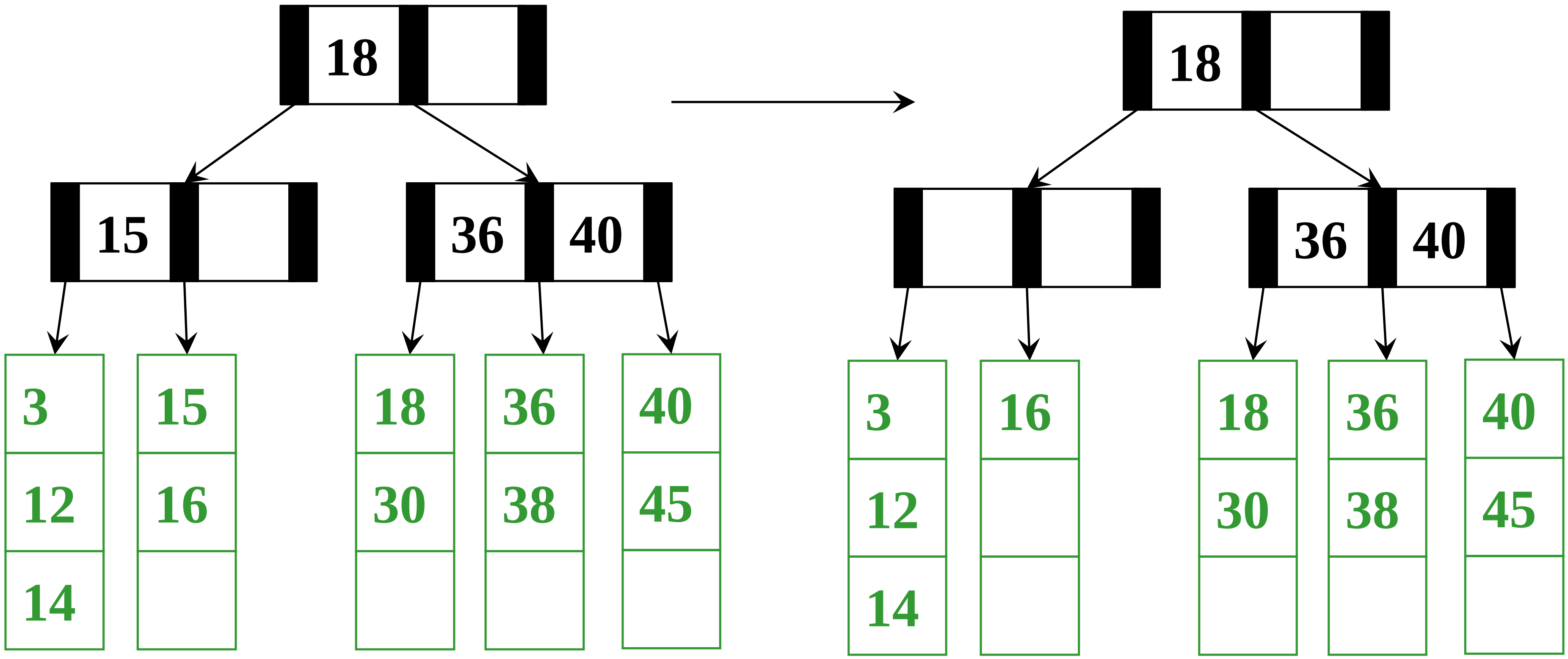
Delete(32)



Easy case: Leaf still has enough data; just remove

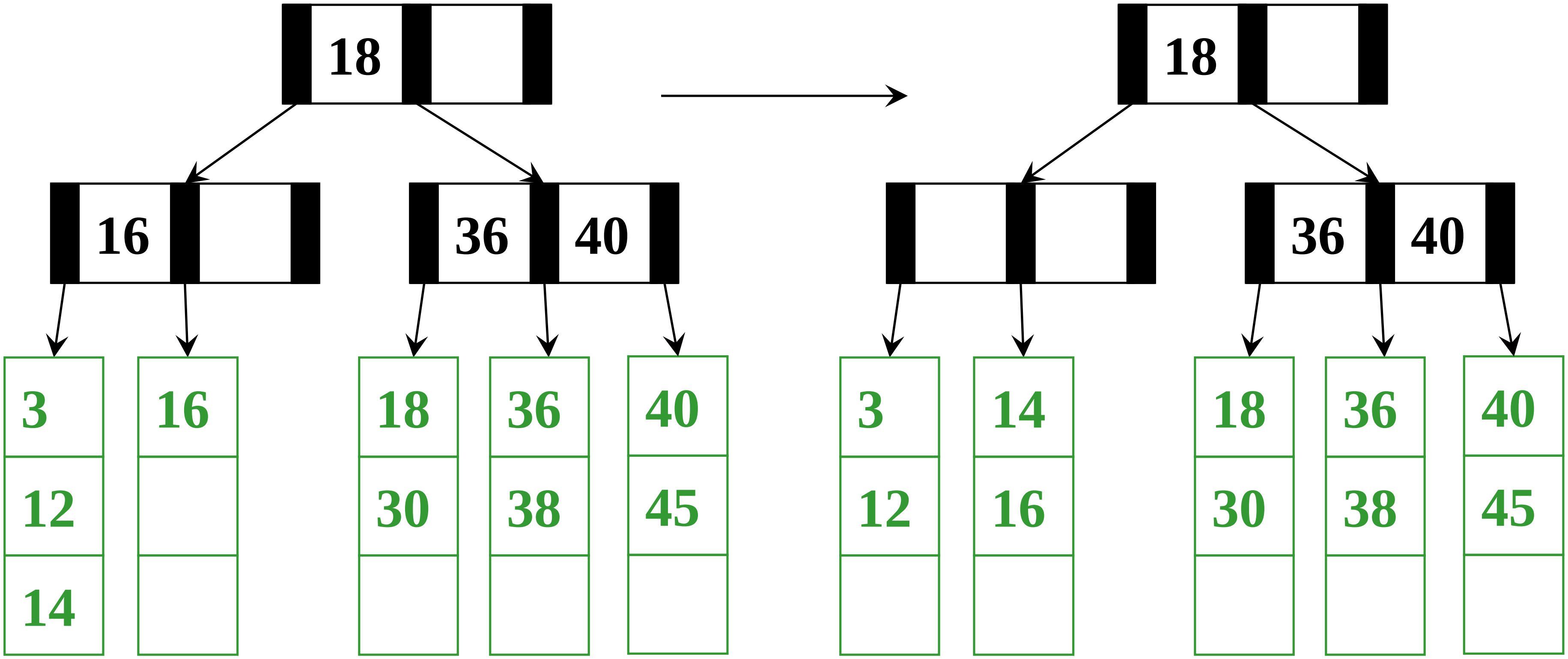
***M* = 3 *L* = 3**

Delete(15)



Is there a problem?

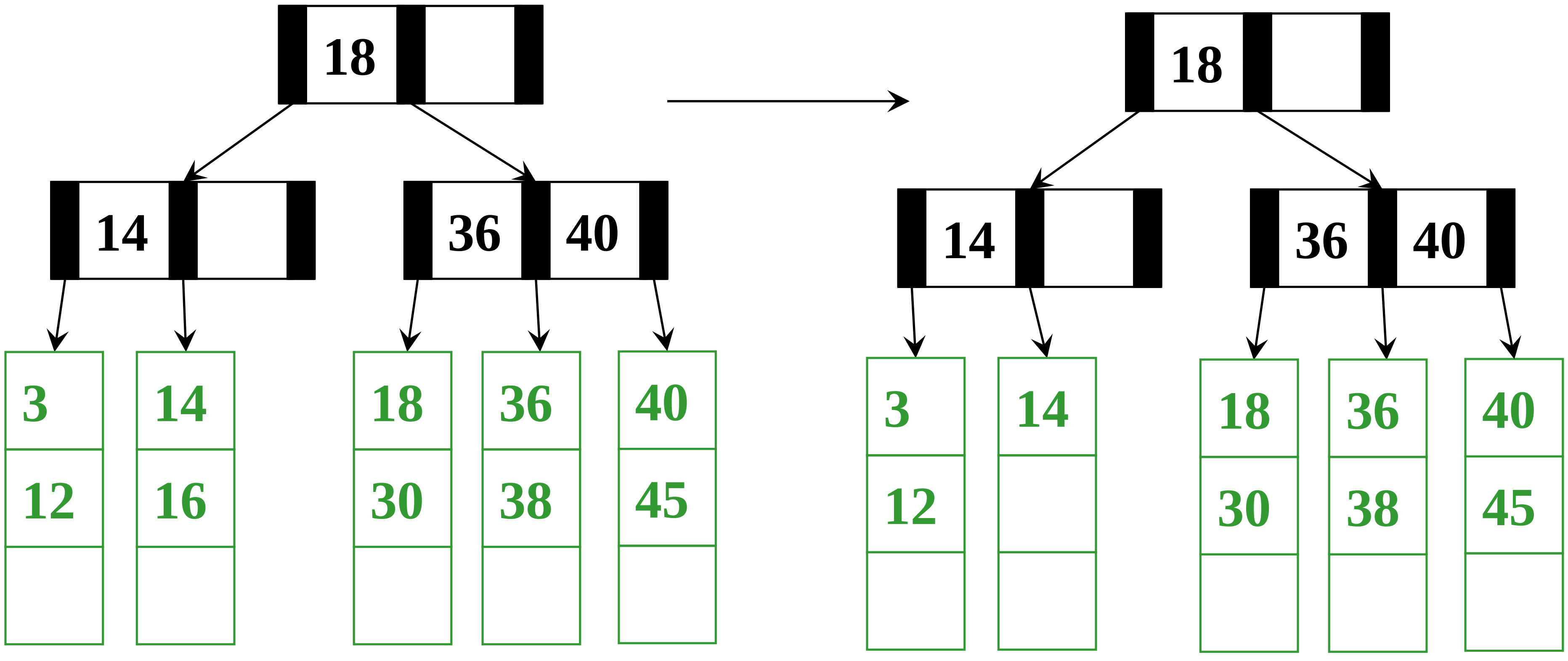
M = 3 L = 3



Adopt from neighbor!

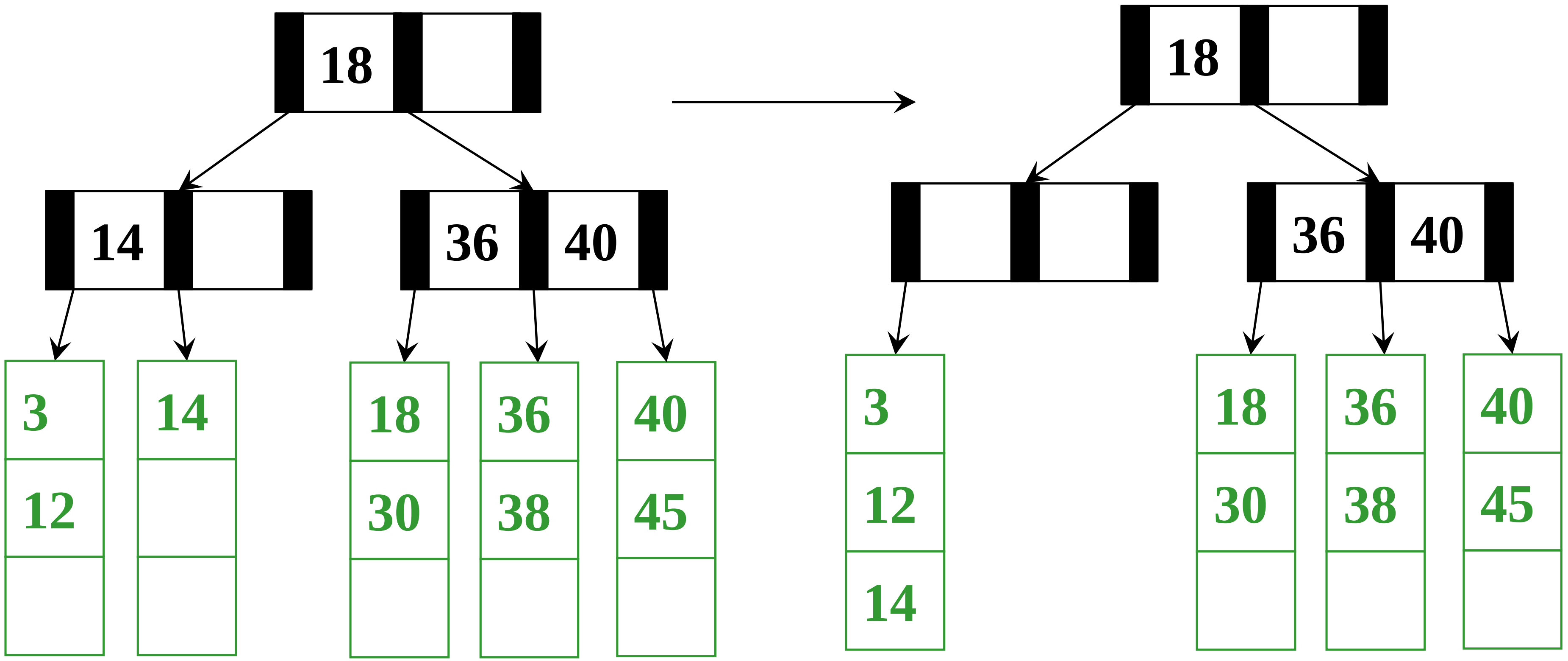
M = 3 *L* = 3

Delete(16)



Is there a problem?

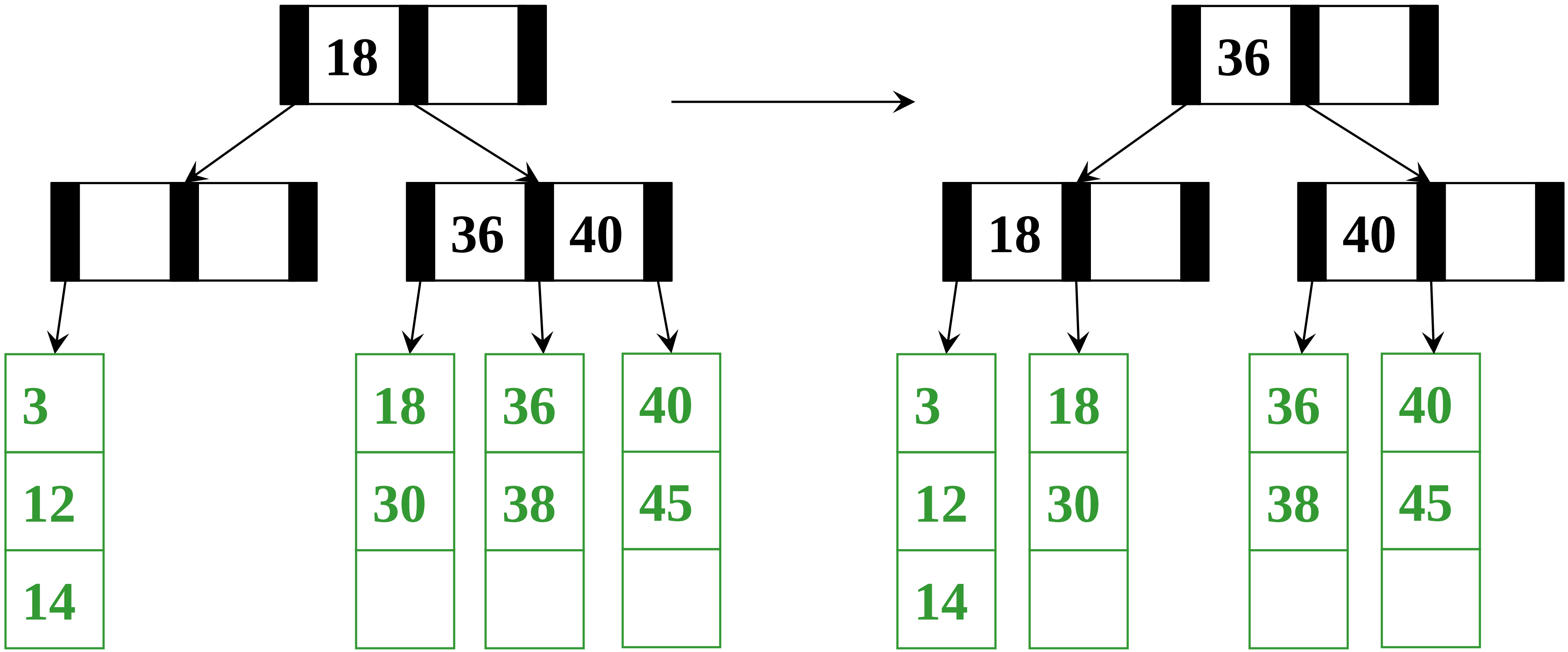
M = 3 L = 3



Merge with neighbor!

But hey, Is there a problem?

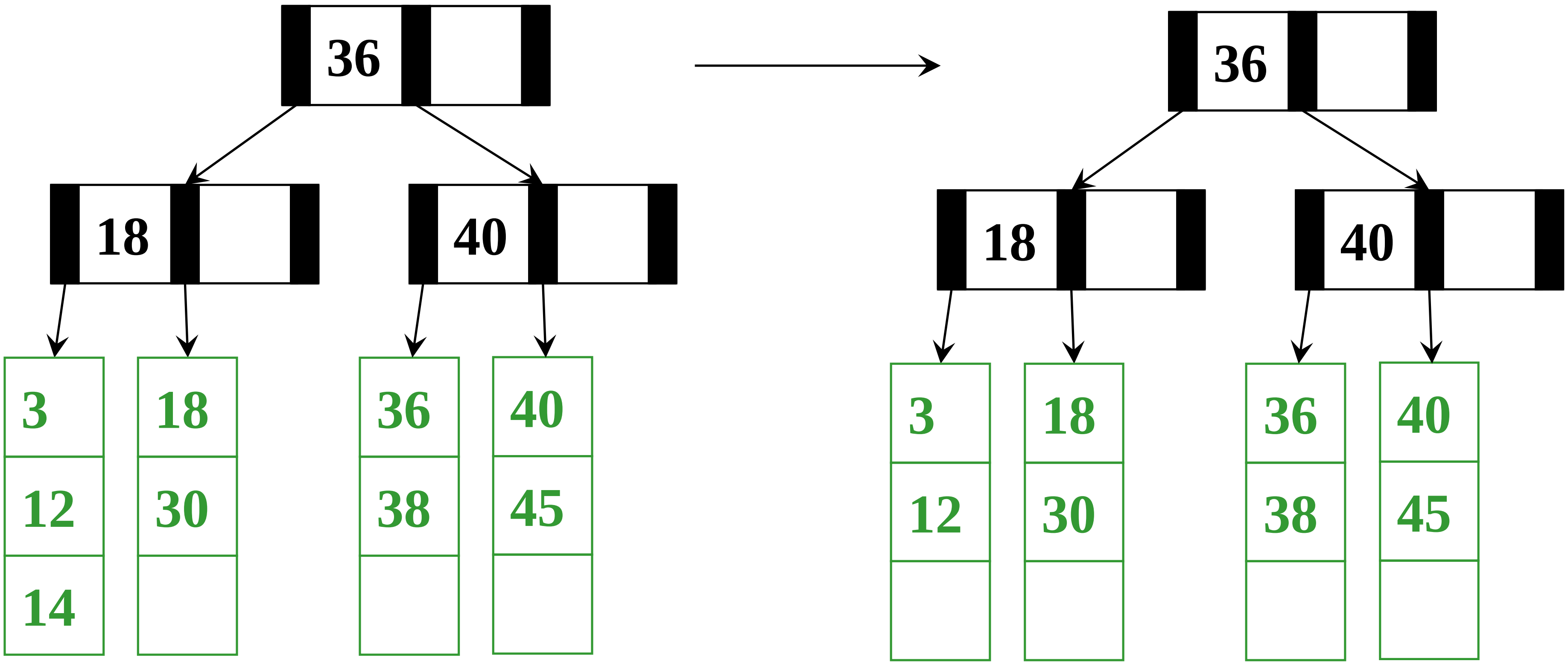
M = 3 *L* = 3



Adopt from neighbor!

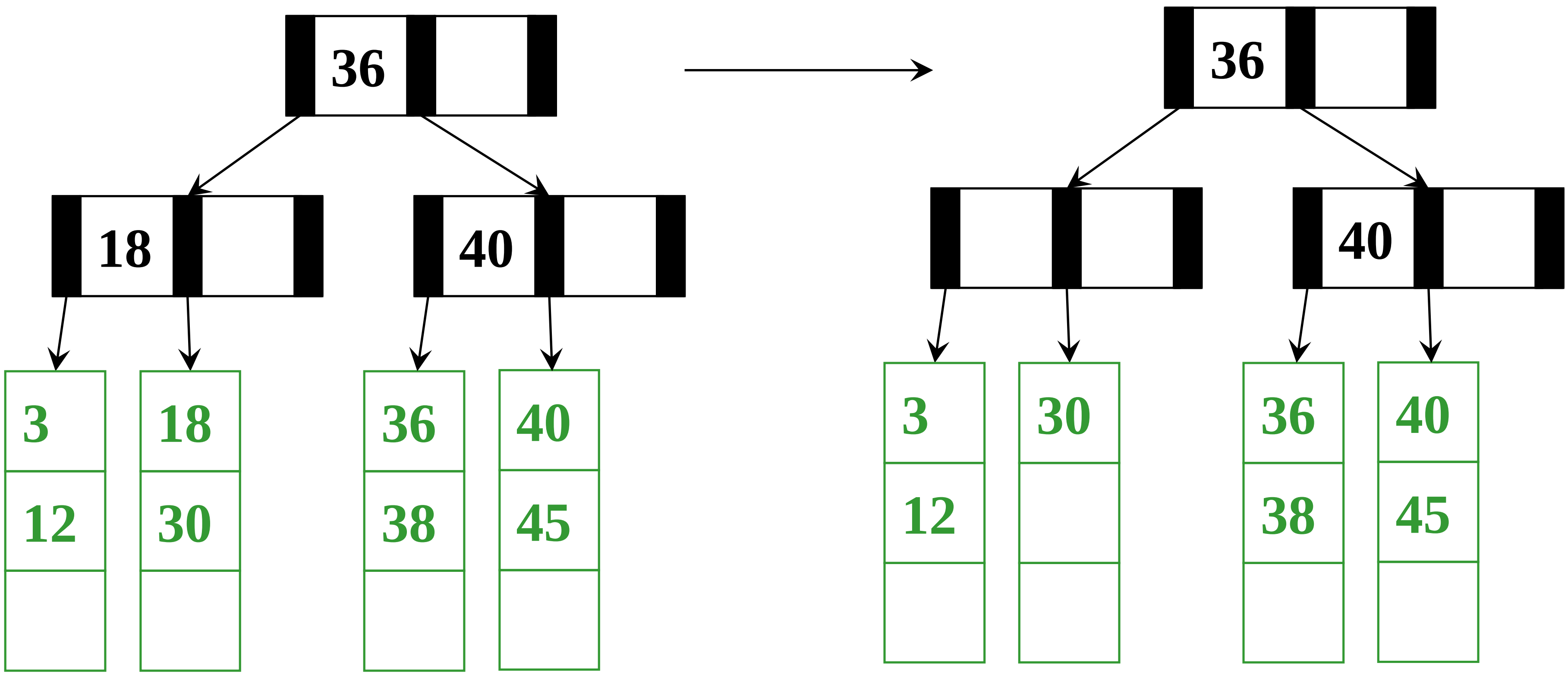
M = 3 *L* = 3

Delete(14)



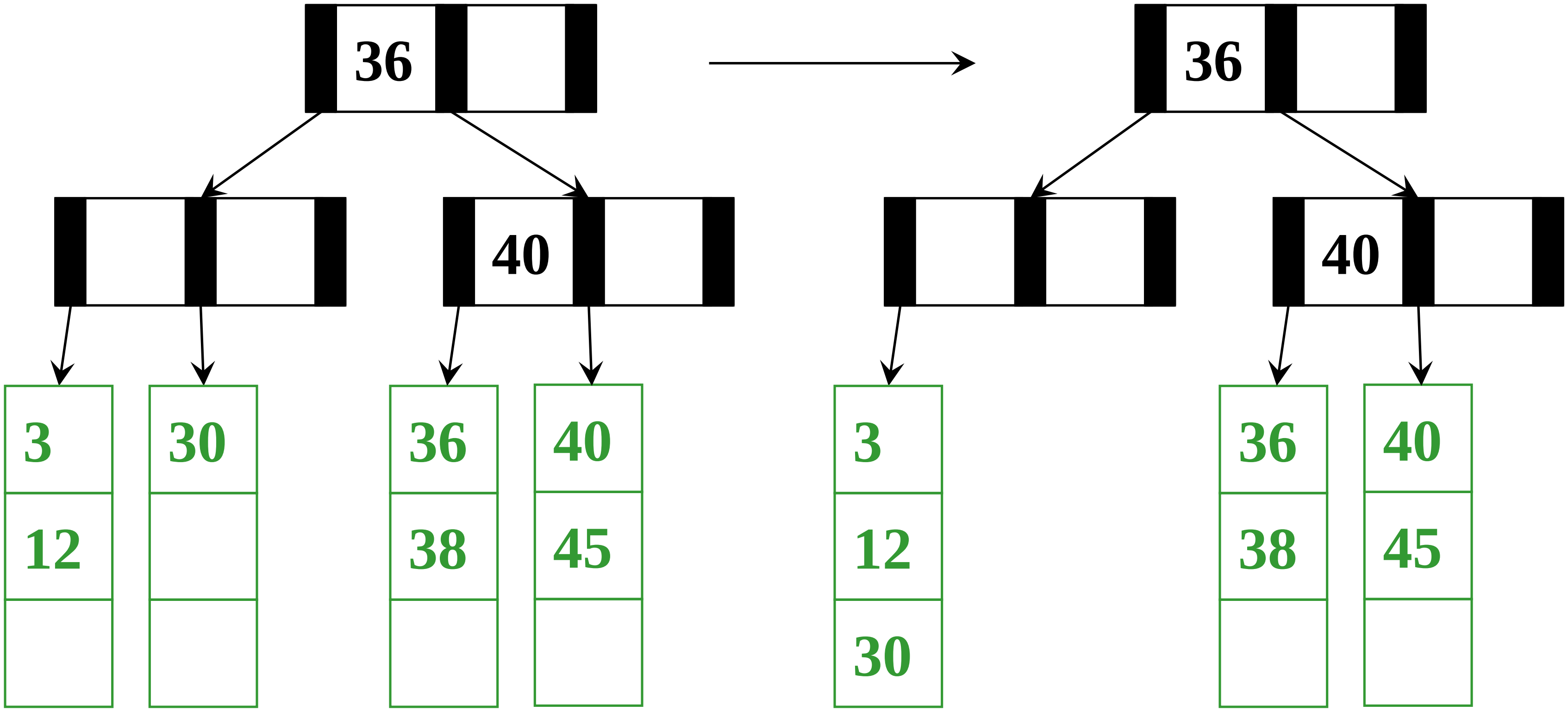
***M* = 3 *L* = 3**

Delete(18)



Is there a problem?

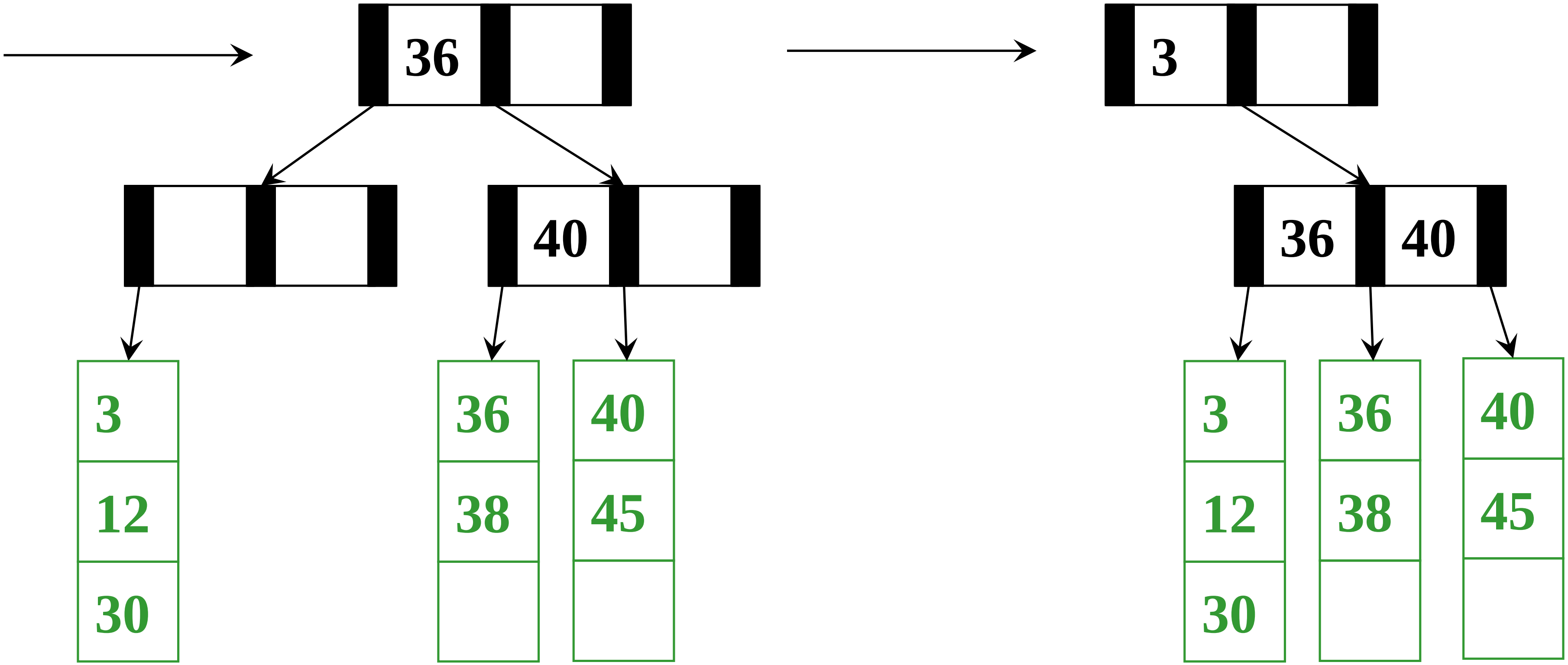
M = 3 L = 3



Merge with neighbor!

$M = 3$ $L = 3$

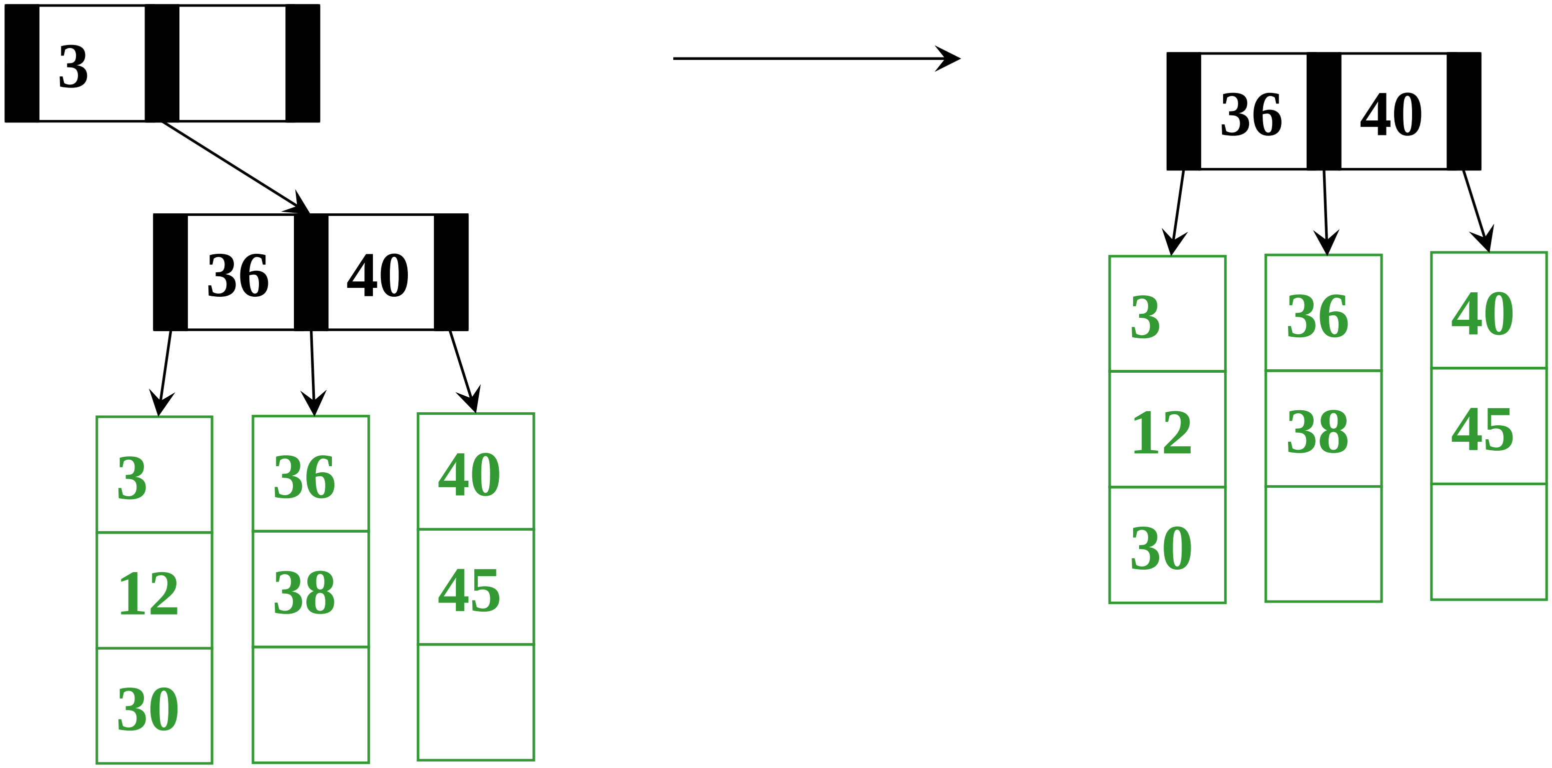
But hey, Is there a problem?



$M = 3$ $L = 3$

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 $\lceil \mathbf{L/2} \rceil$ **- 1**, *underflow!*
   - If a neighbor has > $\lceil \mathbf{L/2} \rceil$ 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 $\lceil \mathbf{M/2} \rceil$ **- 1** children, *underflow!*
   - …

Deletion algorithm (continued)

3. If an internal node has $\lceil \mathbf{M/2} \rceil$ **- 1** children
   - If a neighbor has > $\lceil \mathbf{M/2} \rceil$ 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: $O(M \log_M n)$

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 worthwhile to know enough about "how Java works" to understand why this is probably a bad idea for B trees

- If you just want a balanced tree with worst-case logarithmic operations, no problem
  - If *M*=3, this is called a 2-3 tree
  - If *M*=4, this is called a 2-3-4 tree
- Assuming our goal is efficient number of disk accesses
  - Java has many advantages, but it wasn't designed for this

The key issue is extra *levels of indirection*…

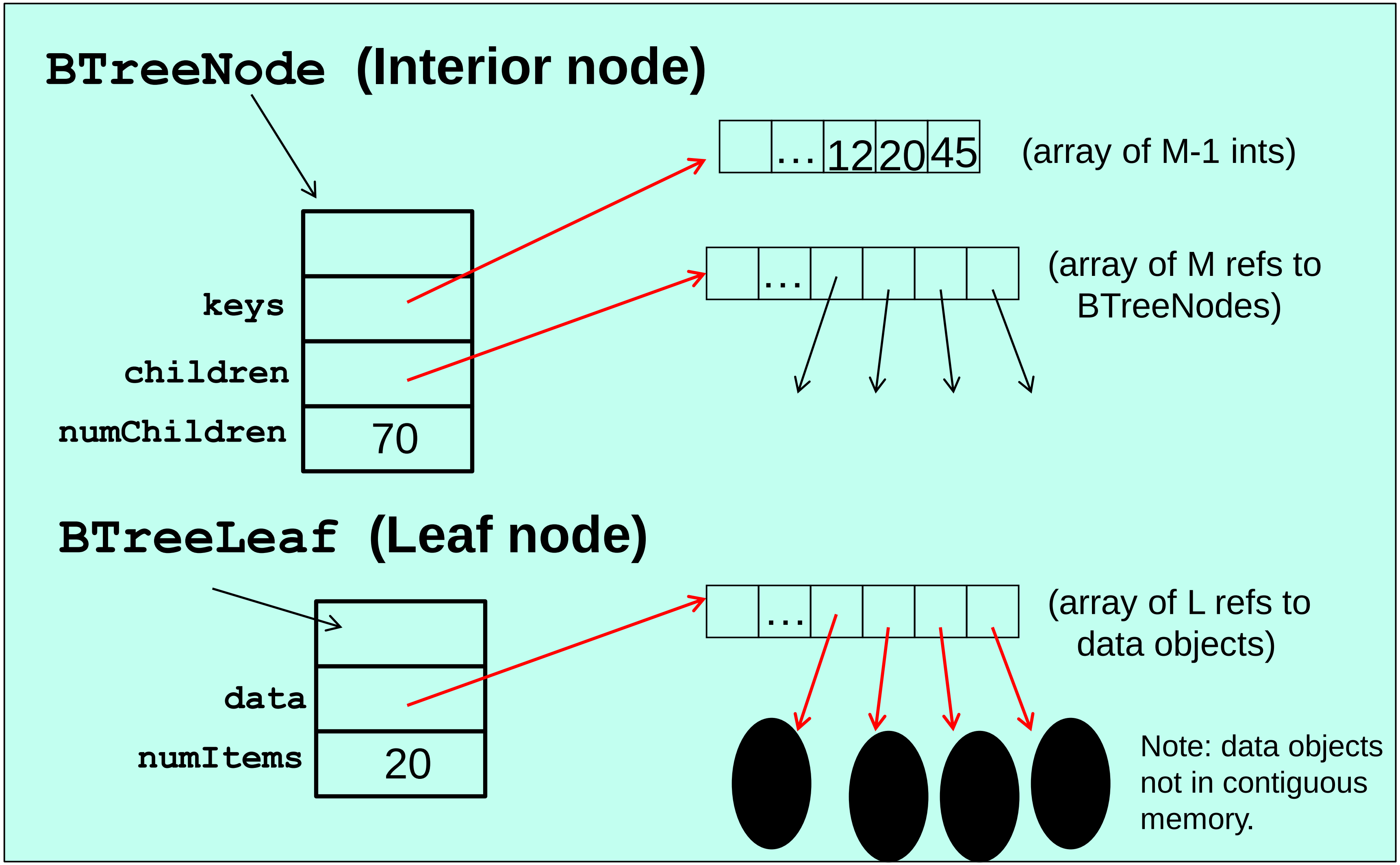
Naïve approach in Java

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

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

What that looks like in Java

All the red references indicate "unnecessary" indirection that might be avoided in another programming language.



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
- Other languages (e.g., C++) have better support for "flattening objects into arrays"
- 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; worth knowing they exist)
  - Red-black trees: all leaves have depth within a factor of 2
  - Splay trees: self-adjusting; amortized guarantee; no extra space for height information