CSE332: Data Abstractions

Lecture 7: B Trees

James Fogarty

Winter 2012

The Dictionary (a.k.a. Map) ADT

- Data:
  - *Set* of (key, value) *pairs*
  - keys must be *comparable*

- Operations:
  - **`insert(key,value)`**
  - **`find(key)`**
  - **`delete(key)`**
  - …

insert(jfogarty, ….) →

- **jfogarty**
  James
  Fogarty
  …
- **hchwei90**
  Haochen
  Wei
  …
- **trobison**
  Tyler
  Robison
  …
- **jabrah**
  Jenny
  Abrahamson
  …

← **find(trobison)**

Tyler, Robison, …

We will tend to emphasize the keys, don't forget about the stored values

Comparison: The Set ADT

The *Set* ADT is like a Dictionary without any values

- A key is *present* or not (i.e., there are no repeats)

For **`find`**, **`insert`**, **`delete`**, there is little difference

- In dictionary, values are "just along for the ride"
- So *same data structure ideas* work for dictionaries and sets

But if your Set ADT has other important operations this may not hold

- **`union`**, **`intersection`**, **`is_subset`**
- Notice these are binary operators on sets
- There are other approaches to these kinds of operations

Dictionary Data Structures

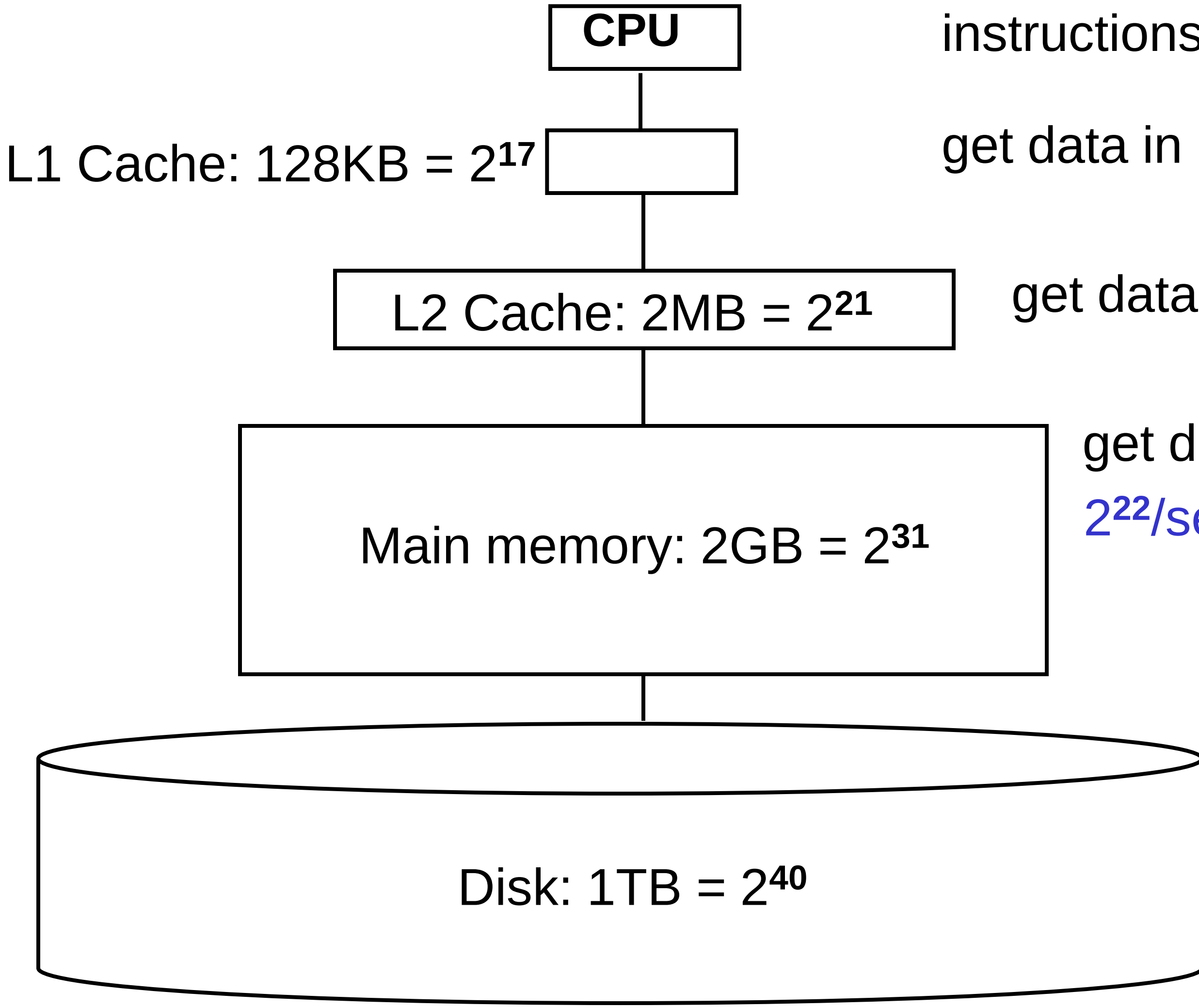
We will see three different data structures implementing dictionaries

1. AVL trees
   - Binary search trees with *guaranteed balancing*
2. B-Trees
   - Also always balanced, but different and shallower
3. Hashtables
   - Not tree-like at all

Skipping: Other balanced trees (e.g., red-black, splay)

A Typical Hierarchy

A plausible configuration …



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

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

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

get data in main memory:
2^{22}/sec = 250 insns

get data from
"new place" on disk:
2^{7}/sec =8,000,000 insns

"streamed": 2^{18}/sec

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 makes lower levels *relatively slower*

Block and Line Size

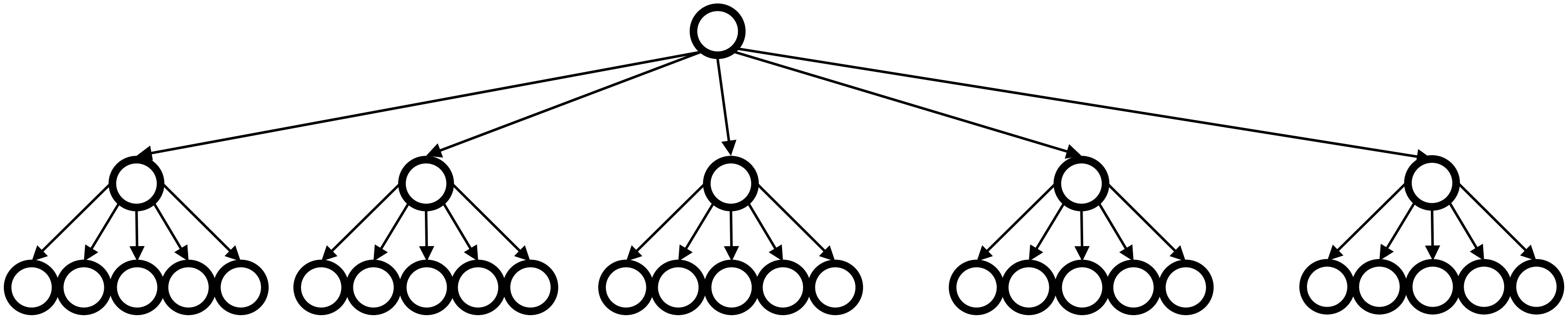
- Moving data up the memory hierarchy is slow because of *latency*
  - Might as well send more, just in case
  - Send nearby memory because:
    - It is easy, we are here anyways
    - And likely to be asked for soon (locality of reference)

- Amount moved from disk to memory is called “block” or “page” size
  - Not under program control

- Amount moved from memory to cache is called the “line” size
  - Not under program control

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)

\# hops for **`find`**: If balanced, using $\log_M n$ instead of $\log_2 n$

- If M=256, that's an 8x improvement
- If $n = 2^{40}$ that's 5 levels instead of 40 (i.e., 5 disk accesses)

Runtime of **`find`** if balanced: $O(\log_2 M \log_M n)$

(binary search children) (walk down the tree)

Problems with M-ary Search Trees

- What should the order property be?

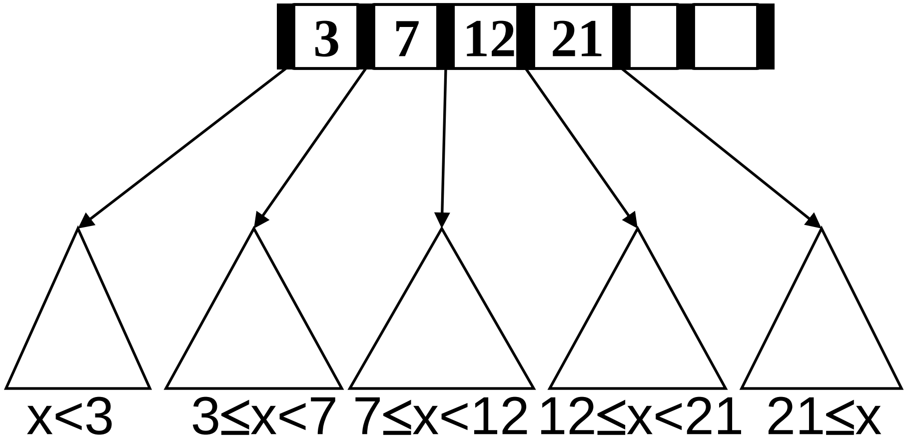
- How would you rebalance (ideally without more disk accesses)?

- Any “useful” data at the internal nodes takes up disk-block space without being used by finds moving past it

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 will just say "B Trees")

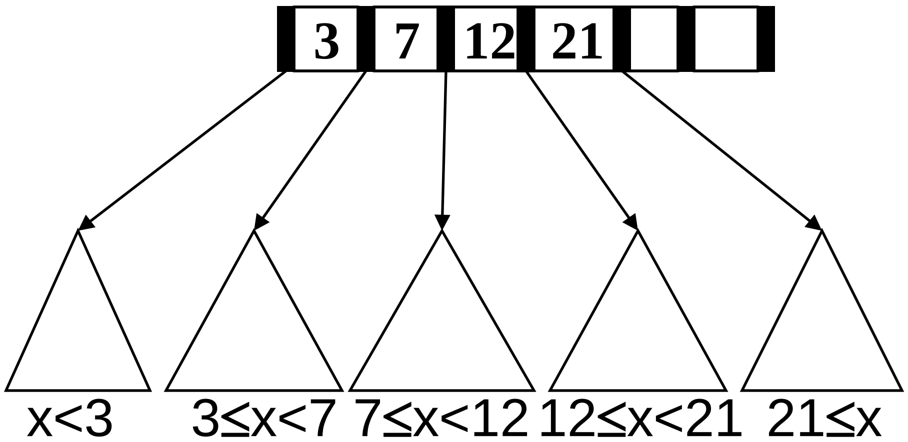
- Two types of nodes:
  - internal nodes and leaf nodes
- Each internal node has room for up to *M-1* keys and *M* children
  - no data; all data at the leaves!
- Order property:
  - Subtree between *x* and *y*
    - Data that is ≥ ***x*** and < ***y***
  - Notice the ≥
- Leaf has up to *L* sorted data items



As usual, we will ignore the presence of data in our examples

Remember it is actually not there for internal nodes

Find



- We are accustomed to data at internal nodes

- But **`find`** is still an easy root-to-leaf recursive algorithm
  - At each internal node do binary search on the ≤ M-1 keys
  - At the leaf do binary search on the ≤ L data items

- 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 very 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)

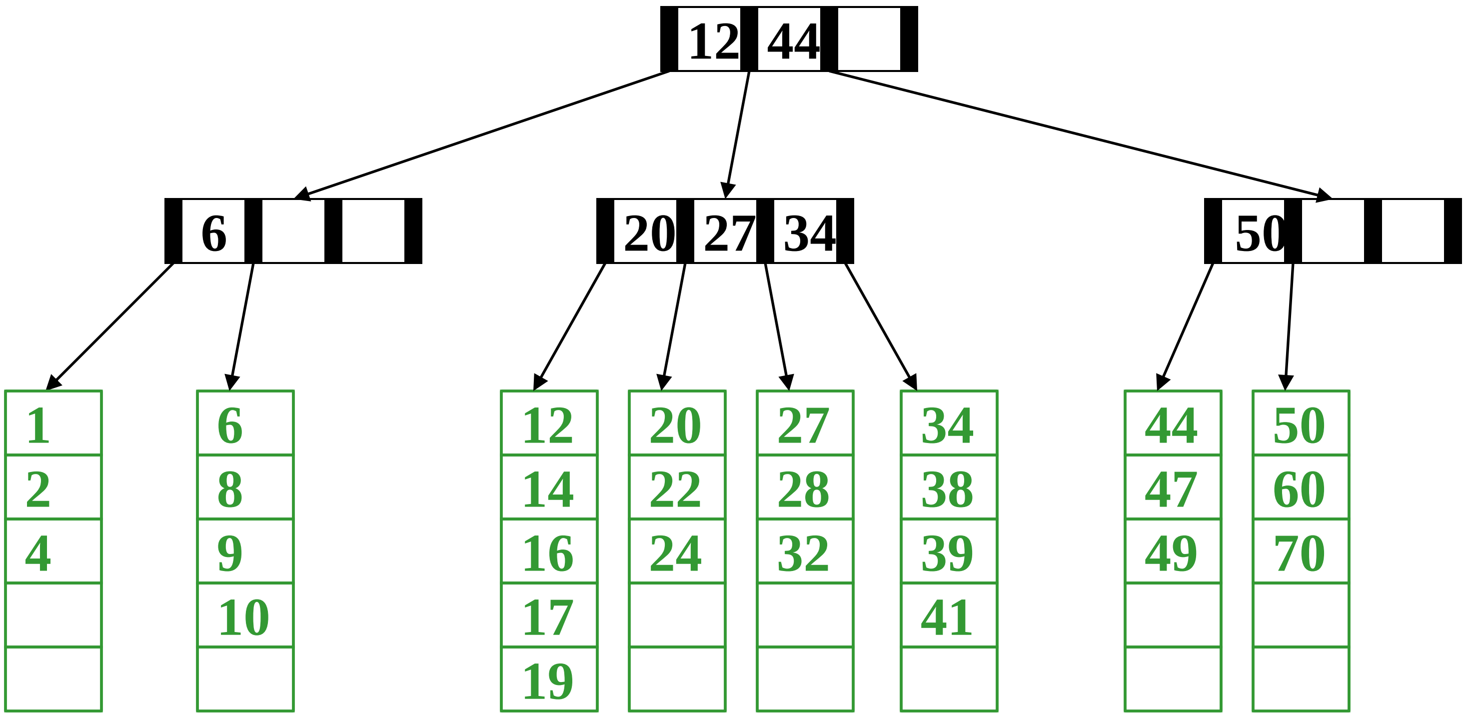
(Any $M > 2$ and L will work; ***picked based on disk-block size***)

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

Example

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

- All **internal nodes** have at least 2 children
- All **leaves** at same depth, have at least 3 data items



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, which is 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…

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

Exponential in height because $\lceil M/2 \rceil > 1$

Disk Friendliness

What makes B trees so disk friendly?

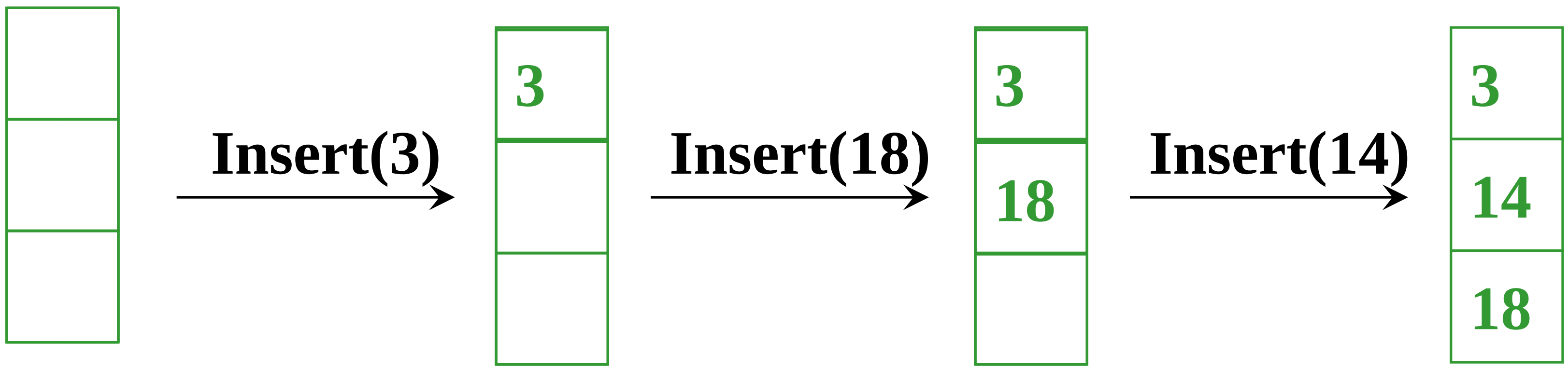
- Many keys stored in one **internal node**
  - All brought into memory in one disk access
  - But only if we pick *M* wisely
  - Makes the binary search over *M*-1 keys totally worth it (insignificant compared to disk access times)

- **Internal nodes** contain only keys
  - Any **`find`** wants only one data item; wasteful to load unnecessary items with internal nodes
  - Only bring one **leaf** of data items into memory
  - Data-item size does not 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

Building a B-Tree

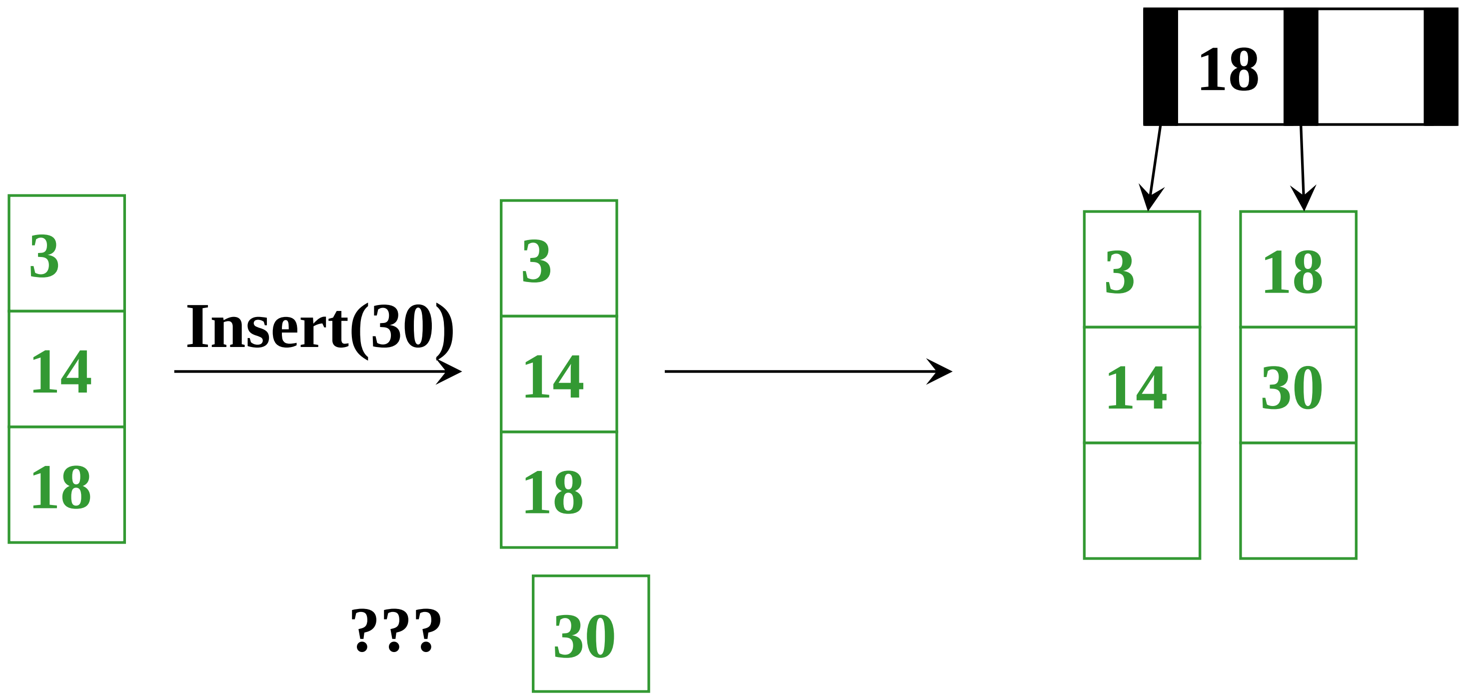


The empty B-Tree (the root will be a leaf at the beginning)

Simply need to keep data sorted

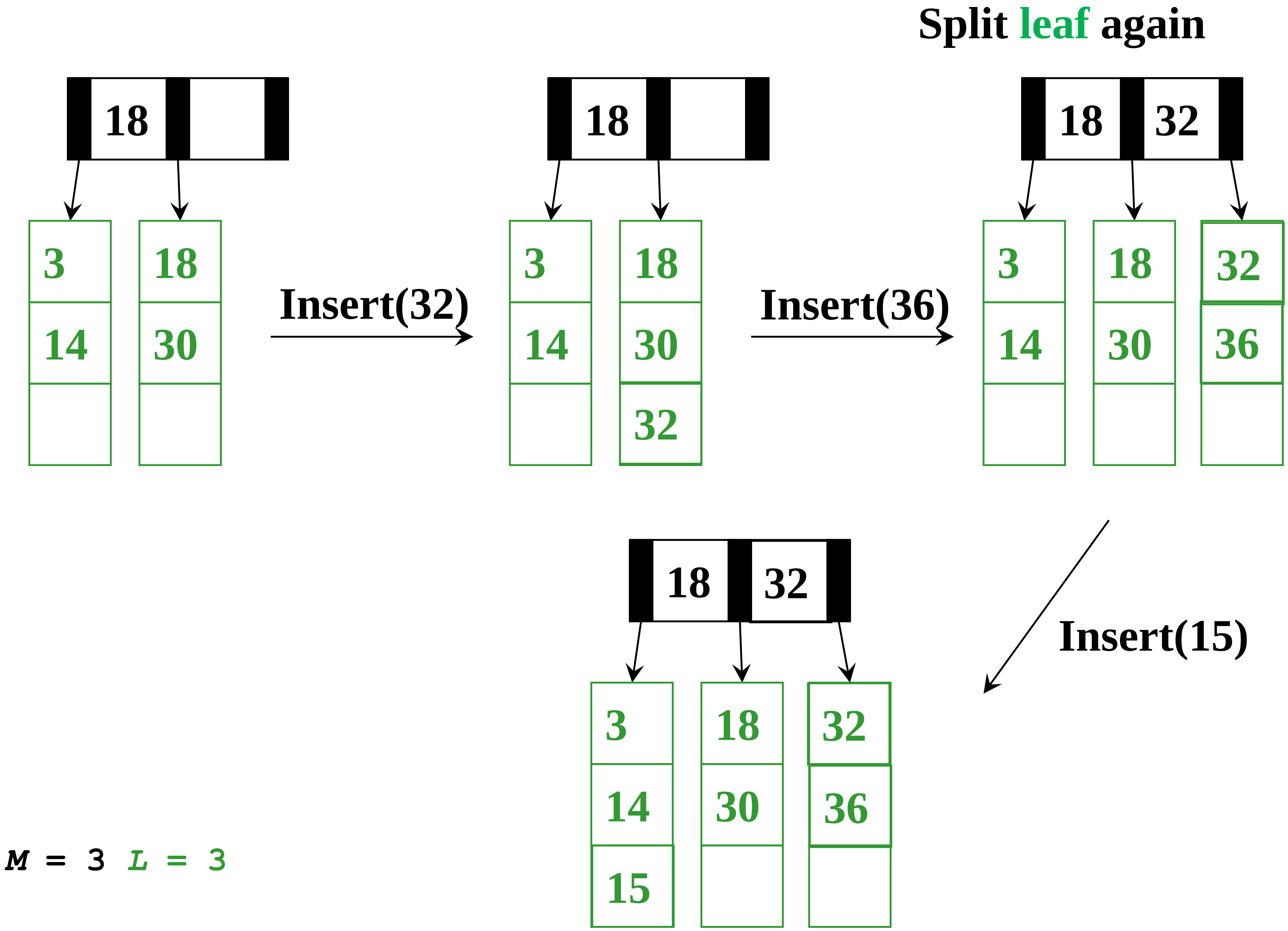
$M = 3$ $L = 3$

$M = 3$ $L = 3$

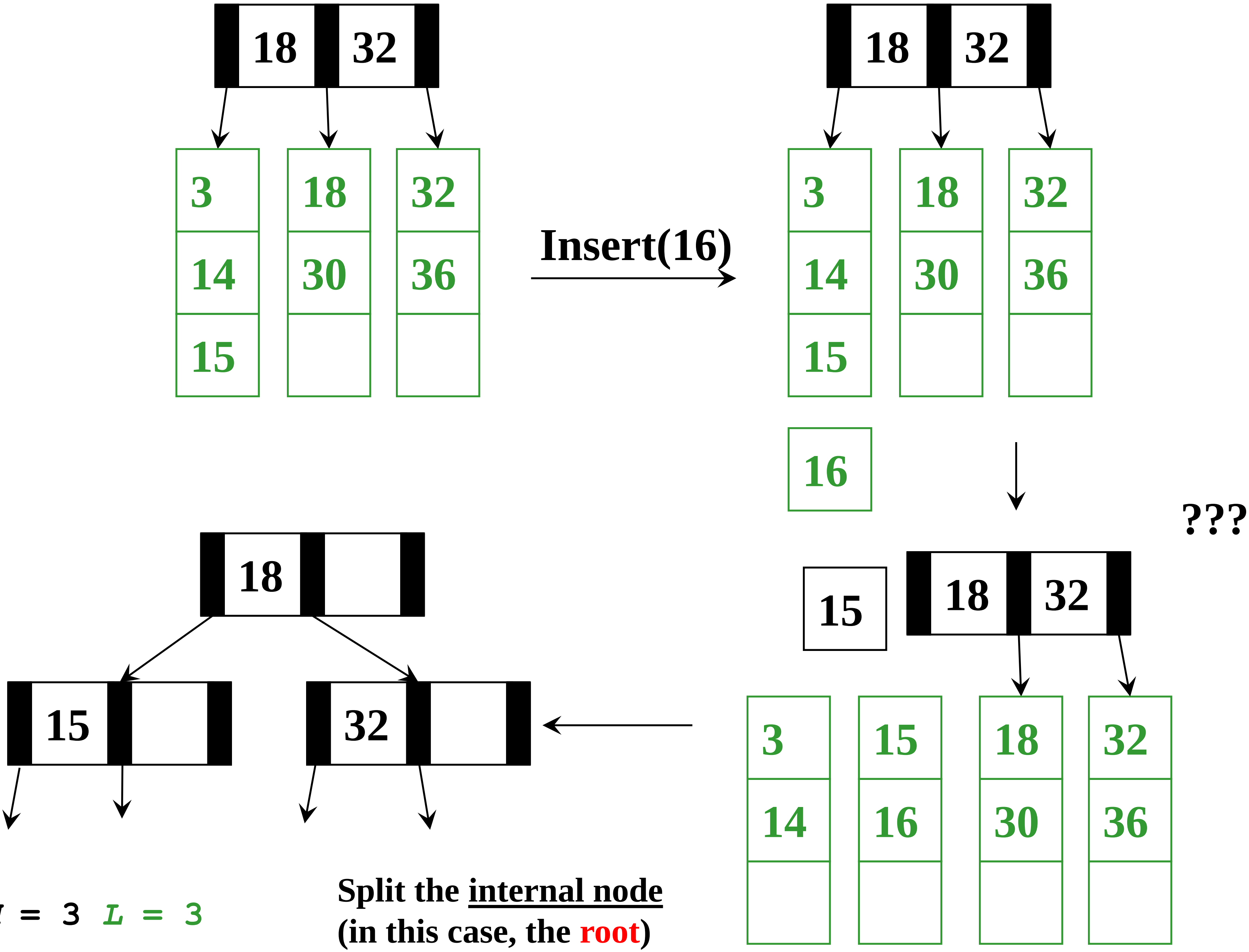


- When we ‘overflow’ a leaf, we split it into 2 leaves
- Parent gains another child
- If there is no parent, we create one

- How do we pick the new key?
  - Smallest element in right tree

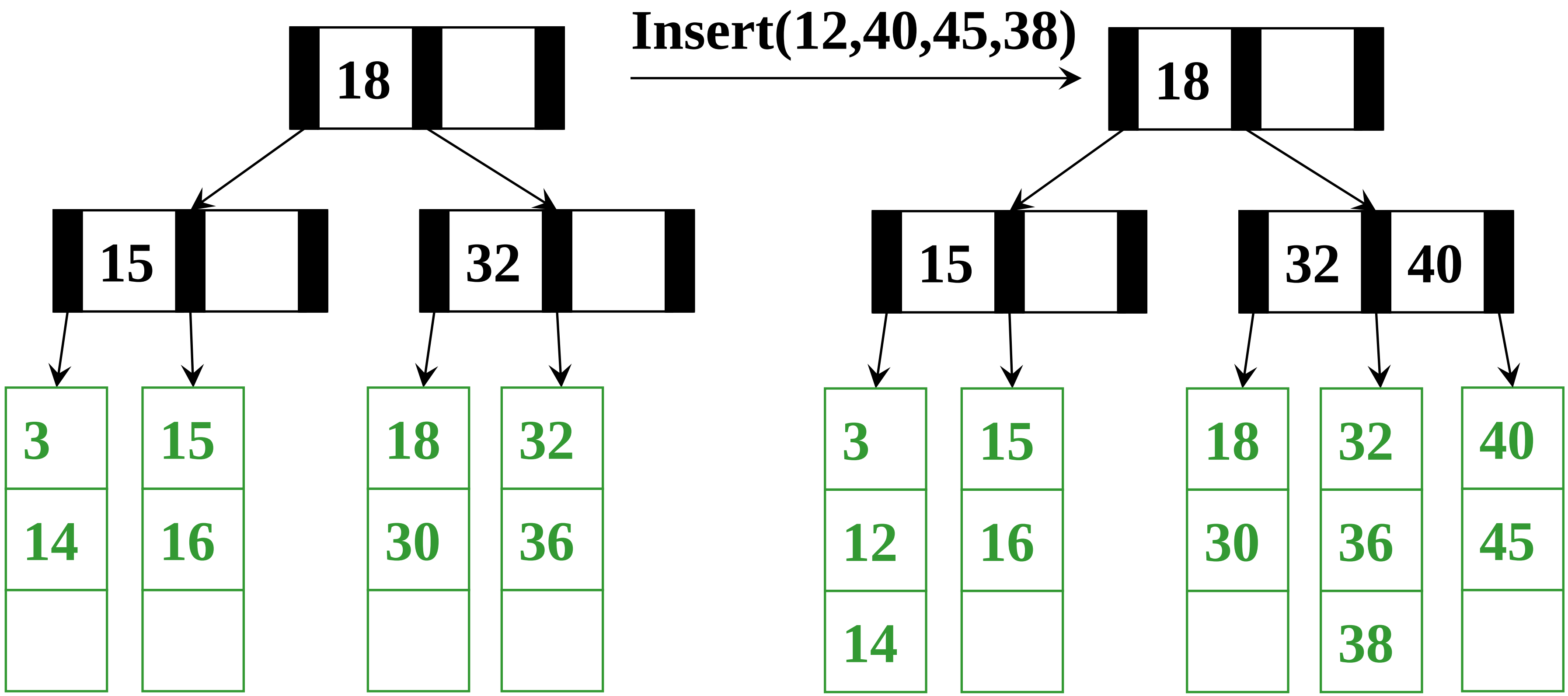


Split leaf again
18
3
14
18
30
Insert(32)
18
3
14
18
30
32
Insert(36)
18
32
3
14
18
30
32
36
Insert(15)
18
32
3
14
15
18
30
32
36
M = 3 L = 3



Split the <u>internal node</u>
(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

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

Step 3 splitting could make the parent overflow too

- *So repeat step 3 up the tree until a node does not overflow*
- If the **root** overflows, make a new **root** with two children
  - This is the only case that increases the tree height

Worst-Case Efficiency of Insert

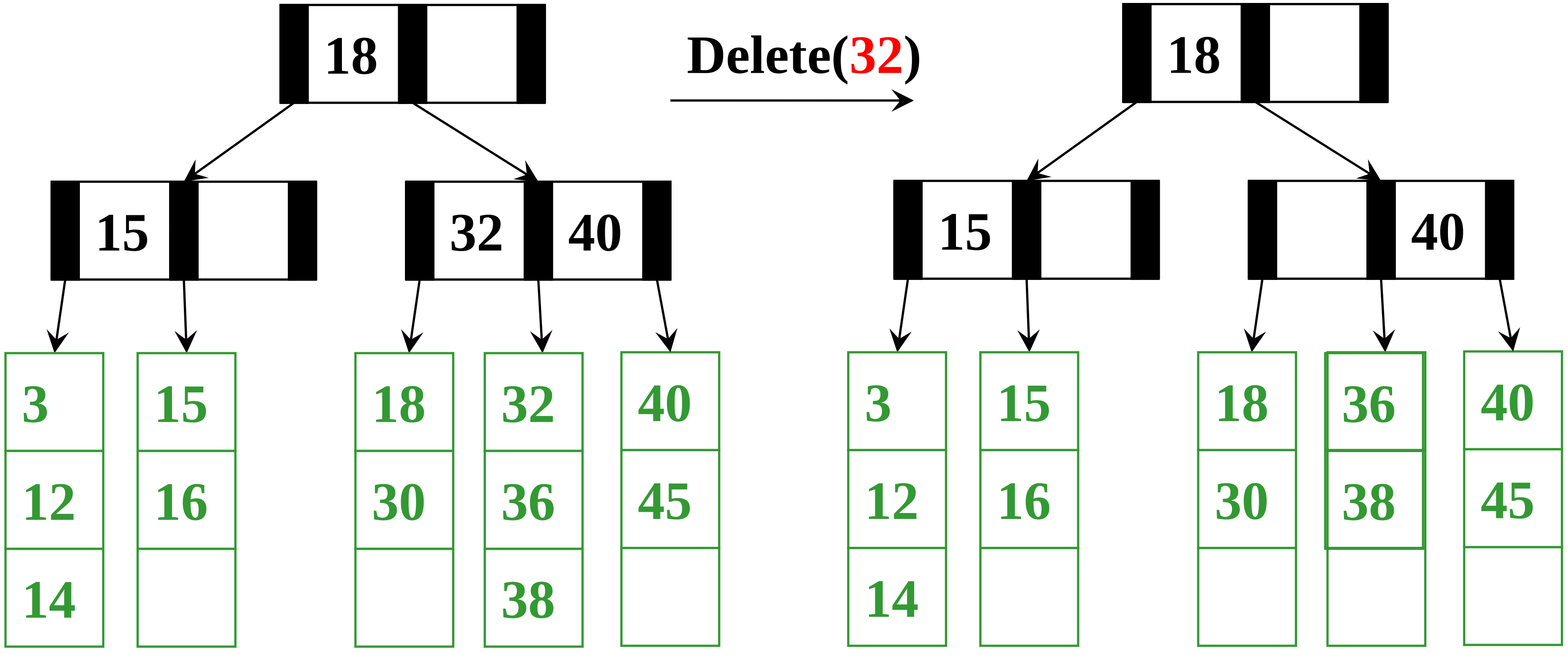
- Find correct leaf: $\boldsymbol{O}(\mathtt{log}_2 \boldsymbol{M}\ \mathtt{log}_{\boldsymbol{M}} \boldsymbol{n})$
- Insert in leaf: $\boldsymbol{O(L)}$
- Split leaf: $\boldsymbol{O(L)}$
- Split parents all the way up to root: $\mathbf{O}(\boldsymbol{M}\ \mathtt{log}_{\boldsymbol{M}} \boldsymbol{n})$

Total: $\boldsymbol{O}(\boldsymbol{L} + \boldsymbol{M}\ \mathtt{log}_{\boldsymbol{M}} \boldsymbol{n})$

But it's not that bad:

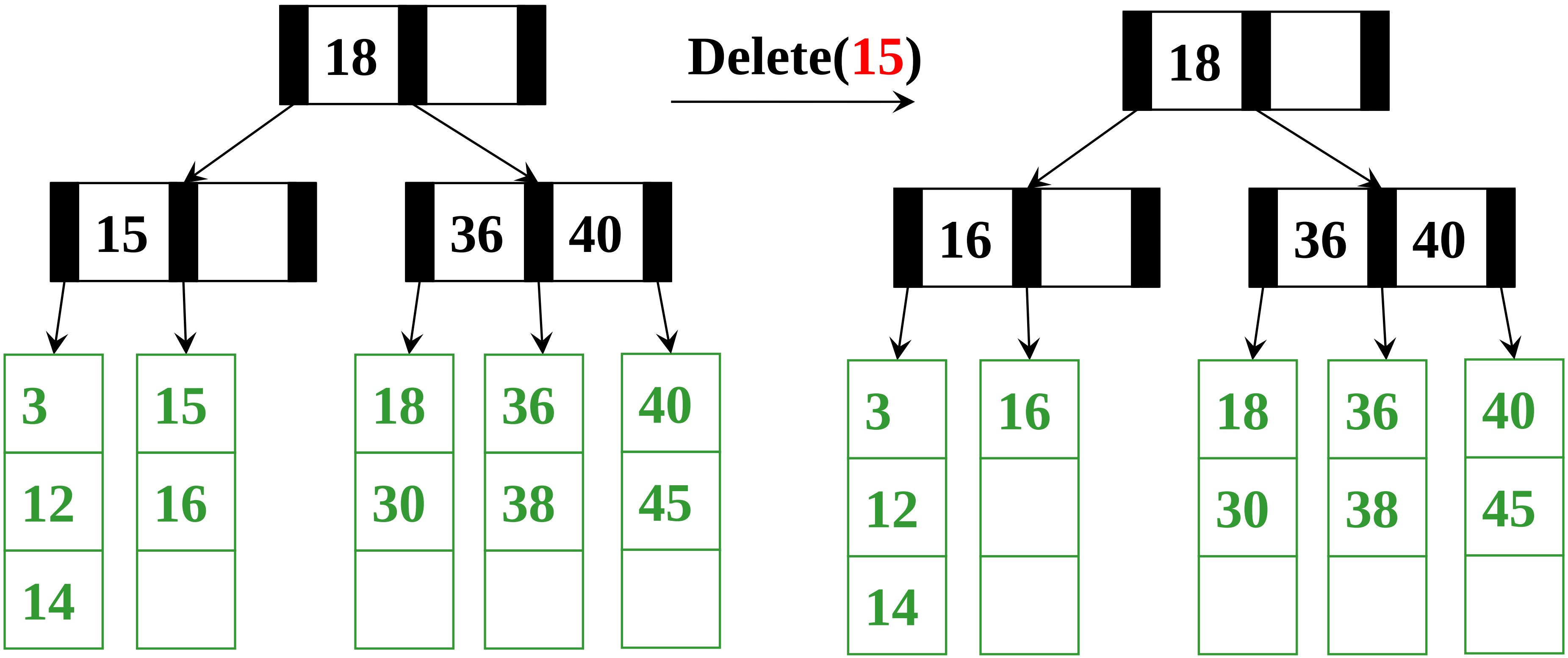
- Splits are not that common (only required when a node is FULL, M and L are likely to be large, and after a split will be half empty)
- Splitting the **root** is extremely rare
- Remember disk accesses is name of the game: $O(\mathtt{log}_M n)$

Deletion



Let them eat cake!

$M = 3$ $L = 3$

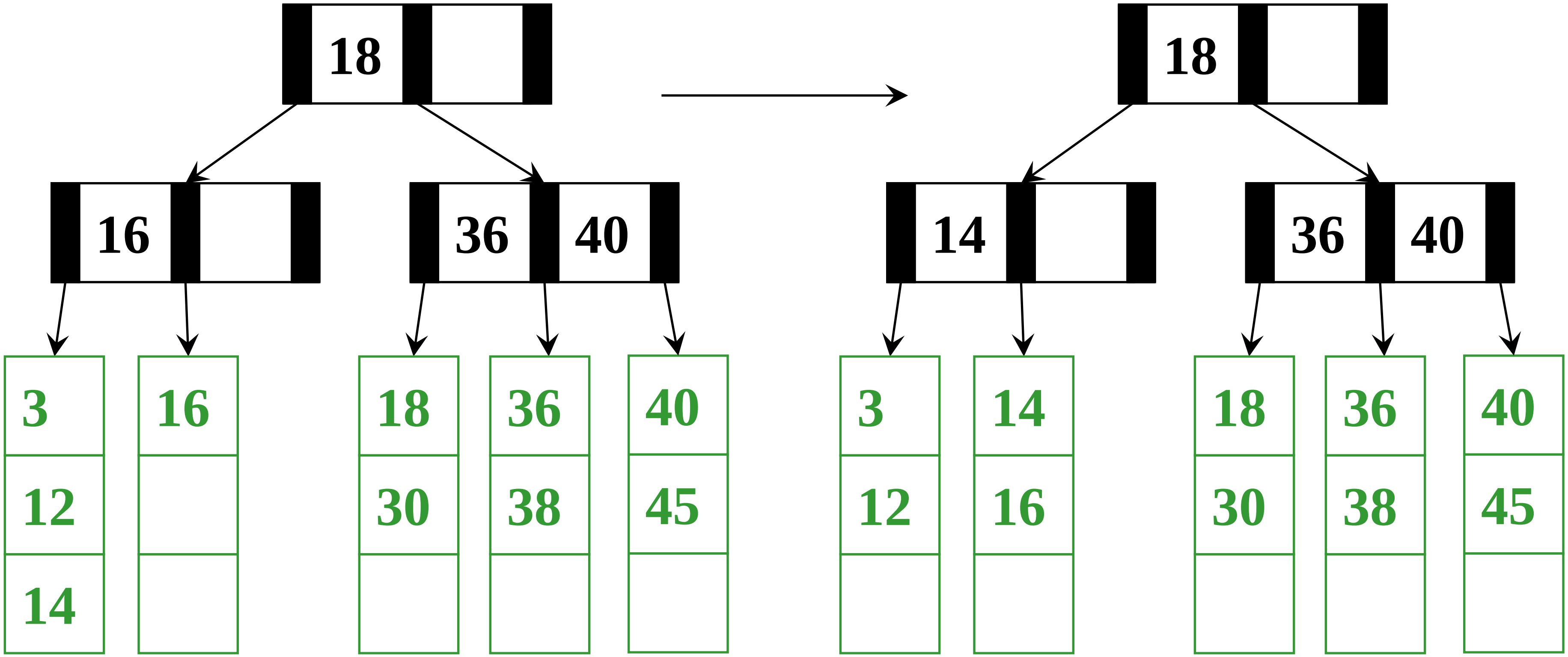


Are we okay?

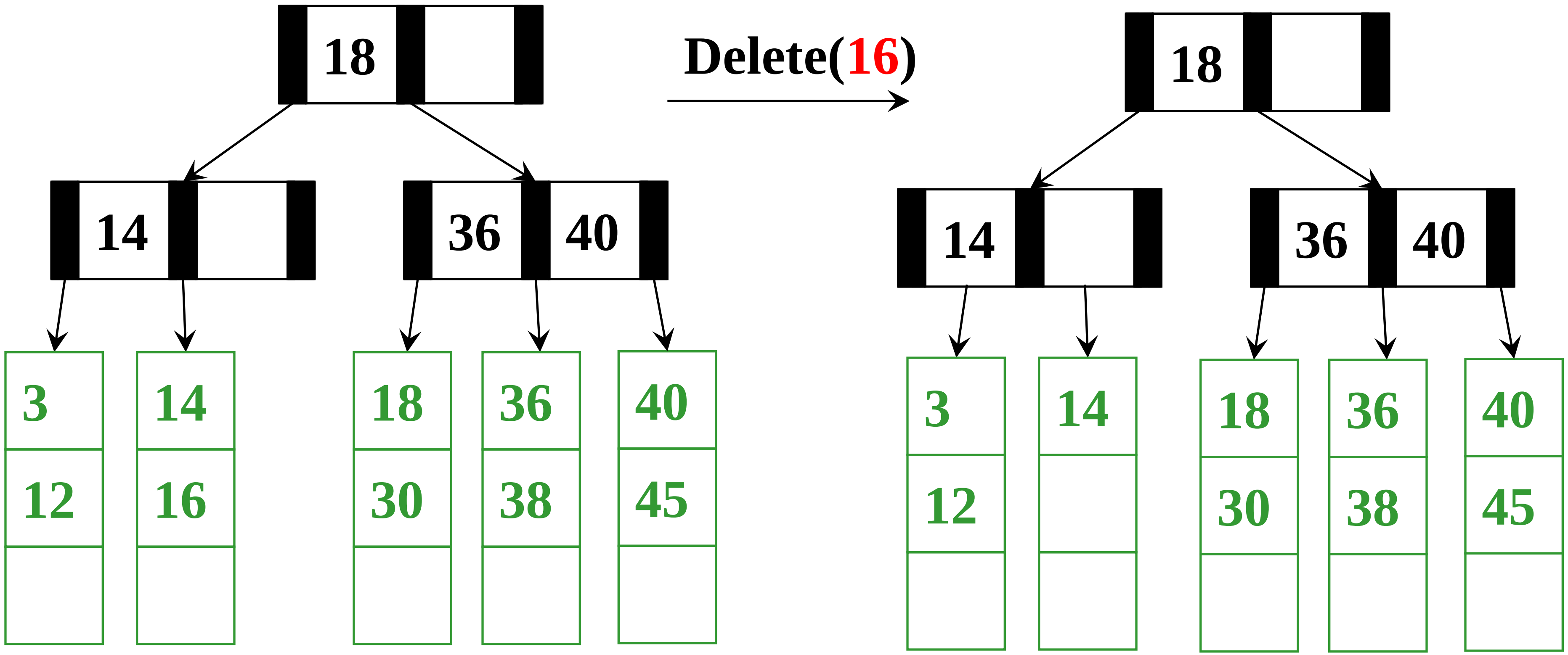
Are you using that 14?
Can I borrow it?

Dang, not half full

M = 3 L = 3


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

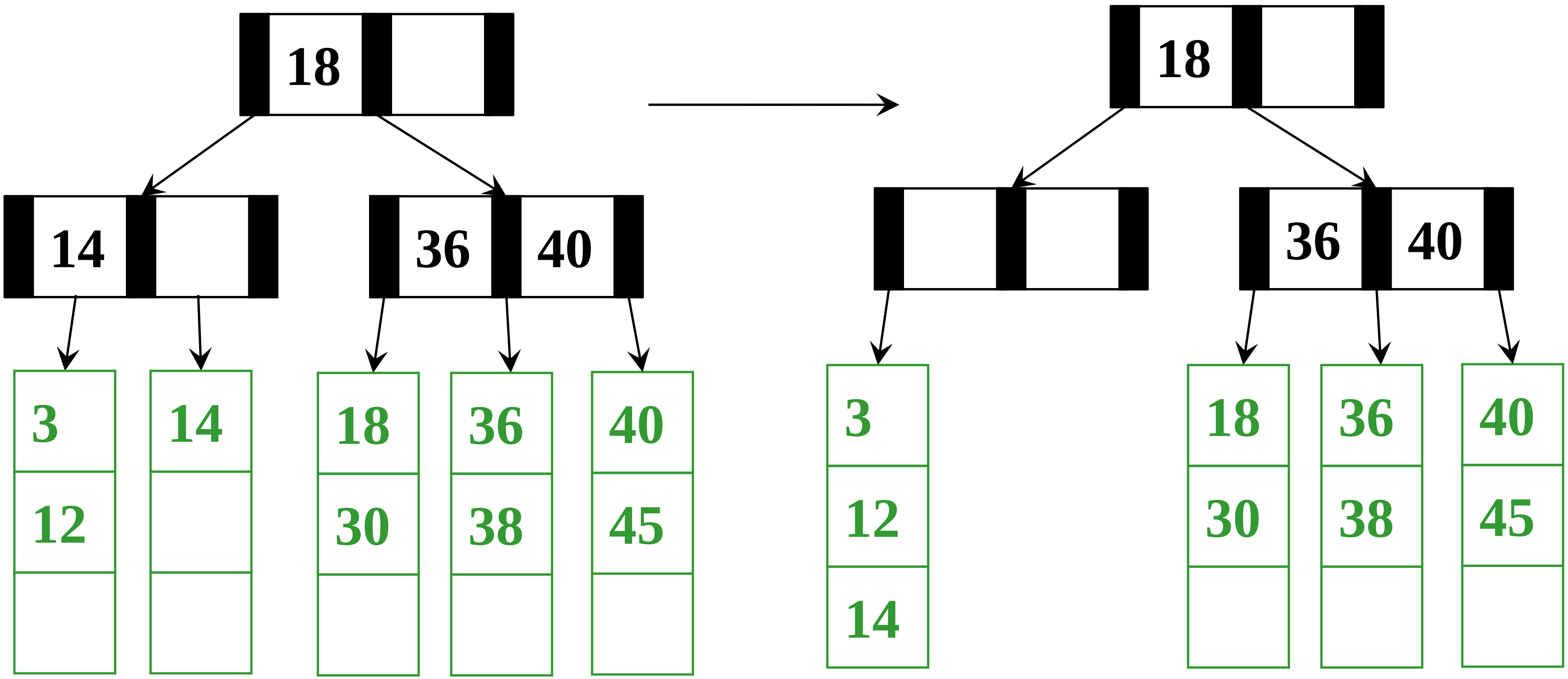
`M = 3 L = 3`



Are you using that 12?

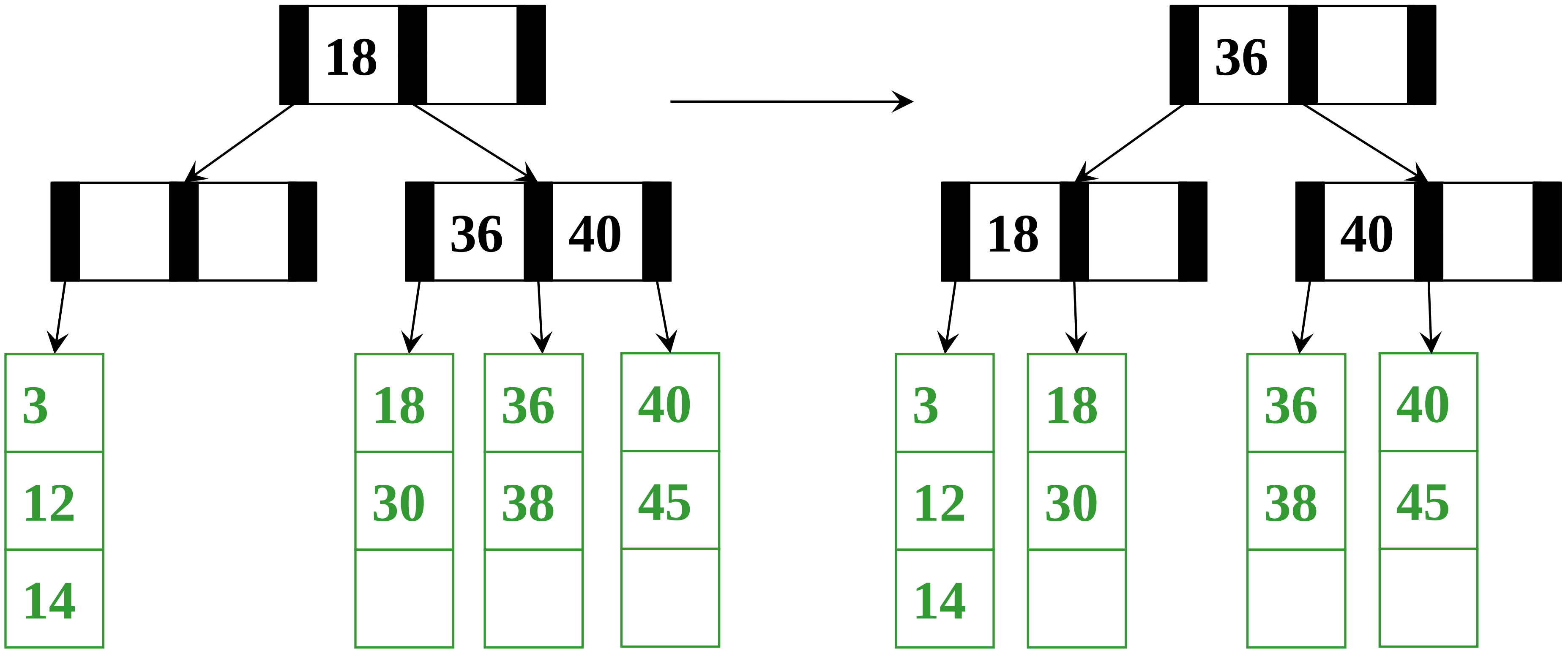
Are you using that 18?

$M = 3$ $L = 3$

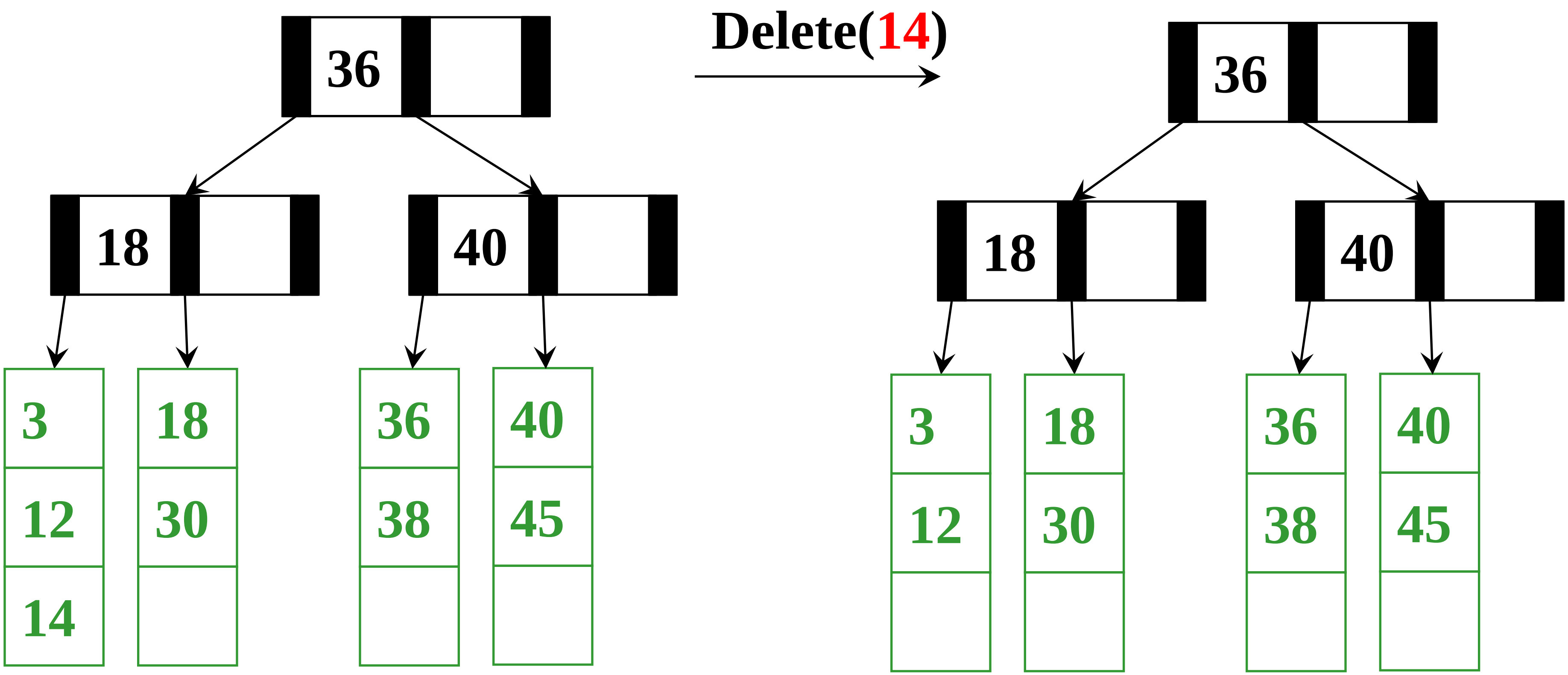


Are you using that 18/30?

$M = 3$ $L = 3$

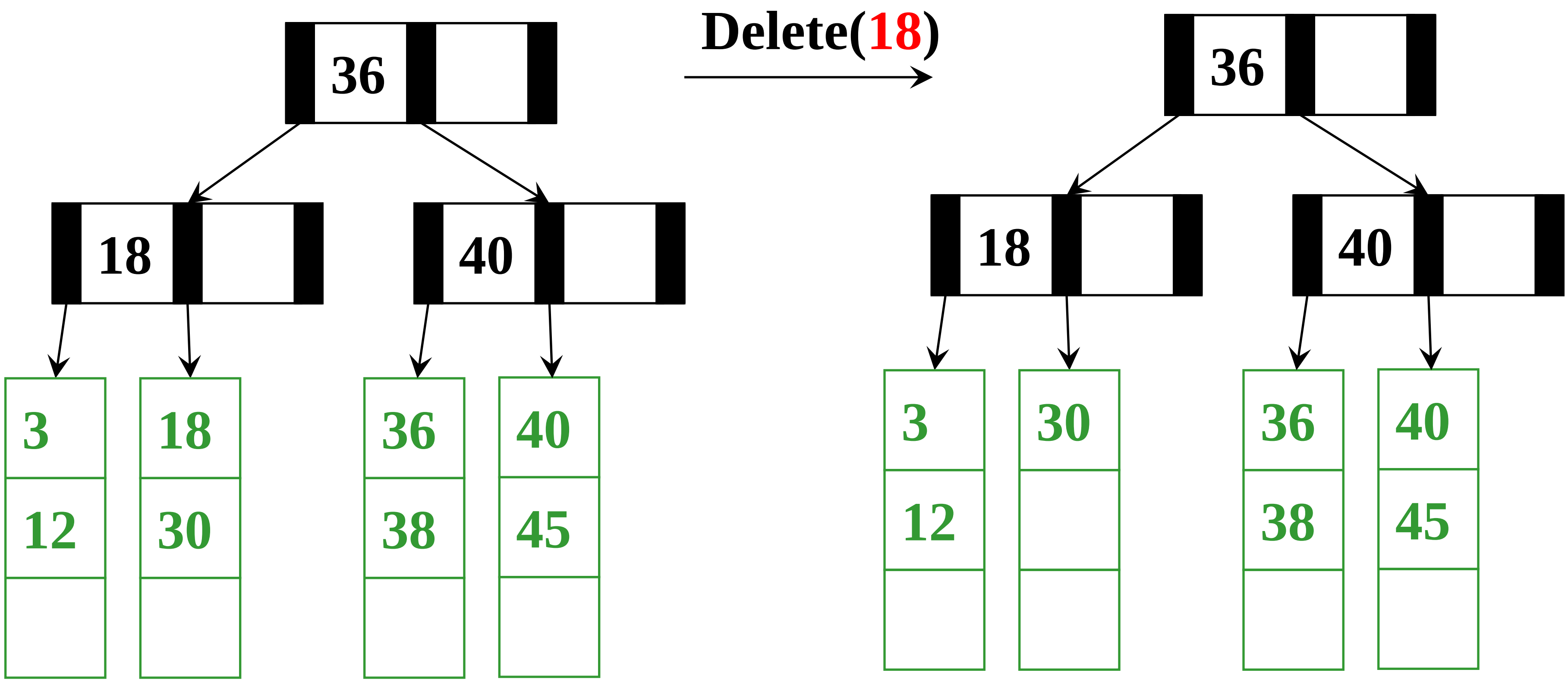


$M = 3$ $L = 3$

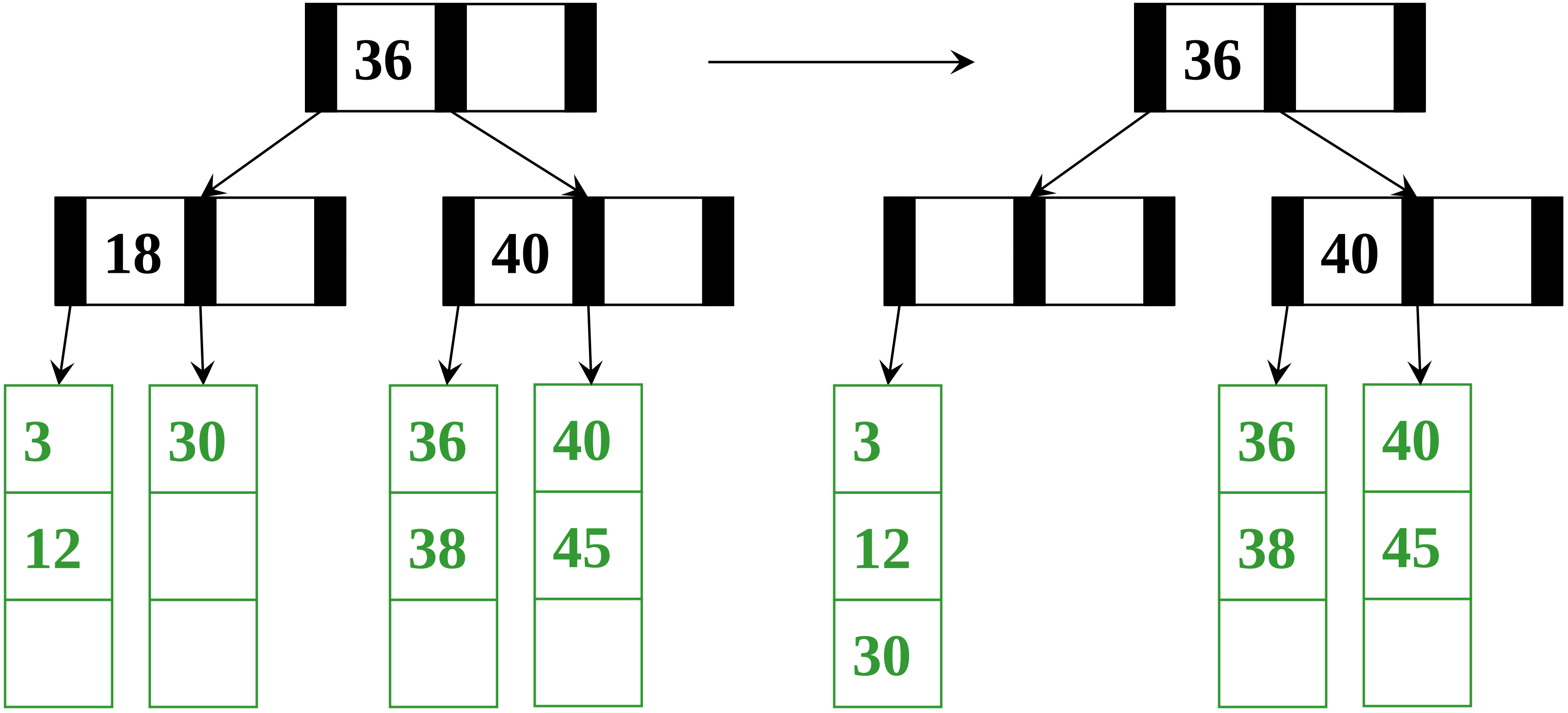

36
18
40
3
12
14
18
30
36
38
40
45
Delete(14)
36
18
40
3
12
18
30
36
38
40
45


easy

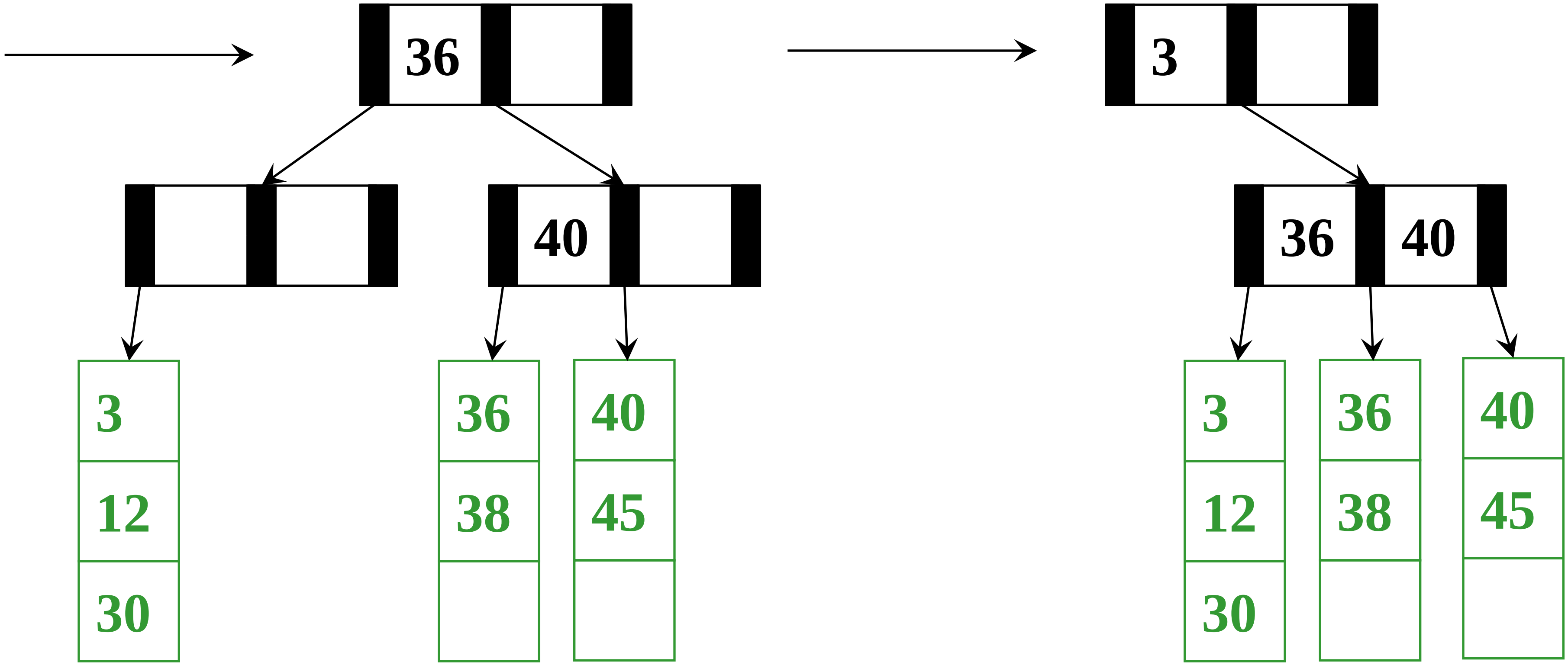
M = 3 *L* = 3



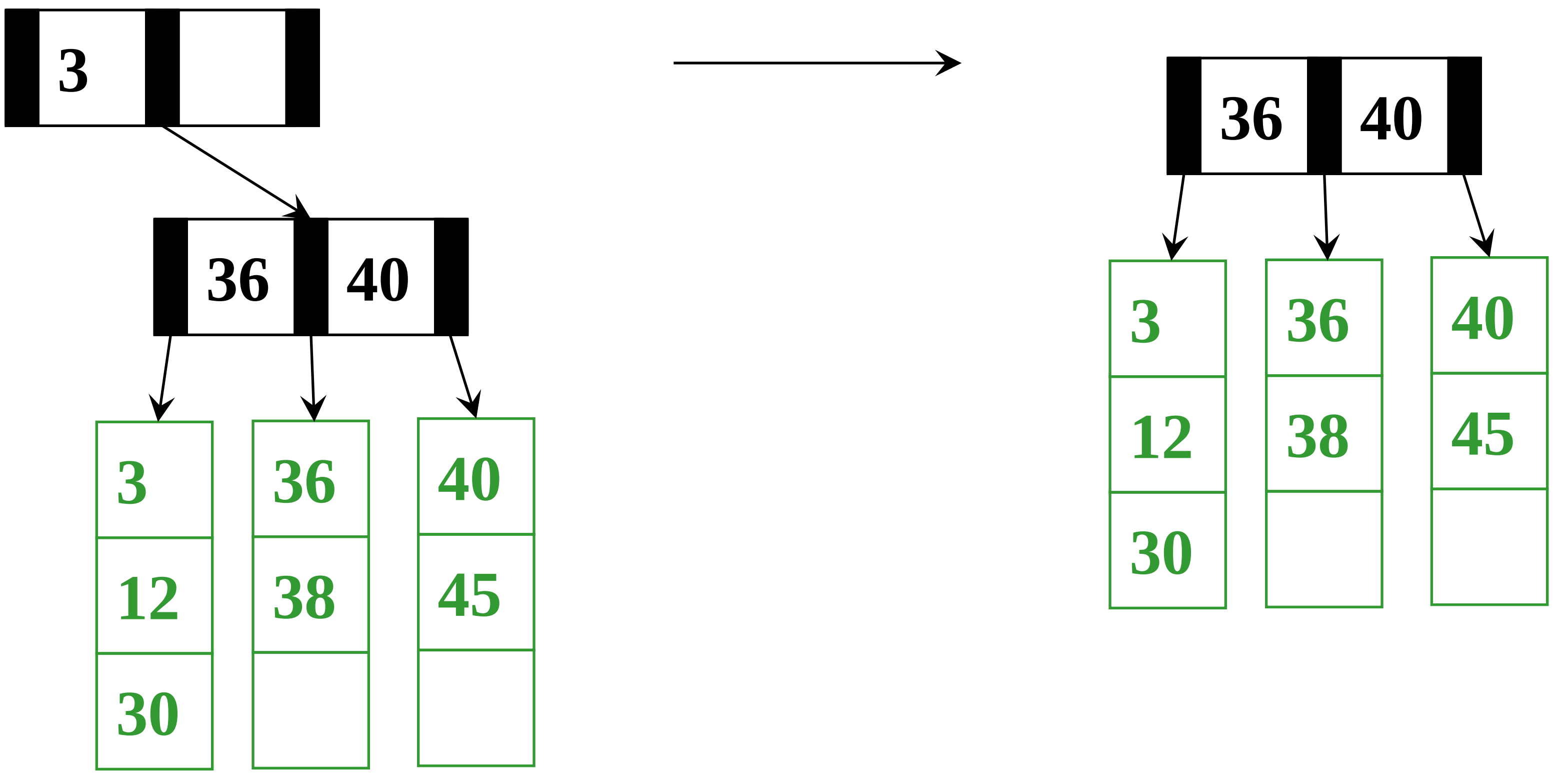
$M = 3$ $L = 3$



M = 3 L = 3



M = 3 L = 3



$M = 3$ $L = 3$

Deletion Algorithm

1. Remove the data from its leaf

2. If the leaf now has $\lceil \boldsymbol{L/2} \rceil$ **- 1**, *underflow!*
   – If a neighbor has > $\lceil \boldsymbol{L/2} \rceil$ 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 parent to have $\lceil \boldsymbol{M/2} \rceil$ **- 1** children, *underflow!*

Deletion Algorithm

3. If an internal node has $\lceil \boldsymbol{M/2} \rceil$ **– 1** children
   - If a neighbor has > $\lceil \boldsymbol{M/2} \rceil$ 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 underflowing up the tree

Fine if we merge all the way up through the root

- 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
- Remember disk access is the name of the game: $O(\log_M n)$

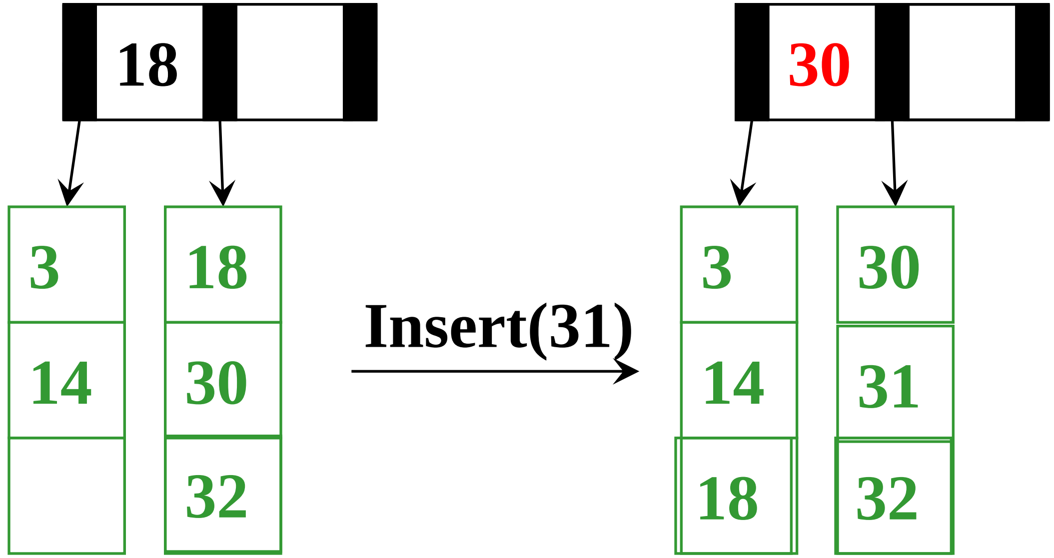
Adoption for Insert

But can sometimes avoid splitting via *adoption*

- Change what leaf is correct by changing parent keys
- This is simply “borrowing” but “in reverse”
- Not necessary

Example:

Adoption



B Trees in Java?

Remember you are learning deep concepts, not just trade skills

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

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

- If you just want balance with worst-case logarithmic operations
  - No problem, M=3 is a 2-3 tree, M=4, is 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

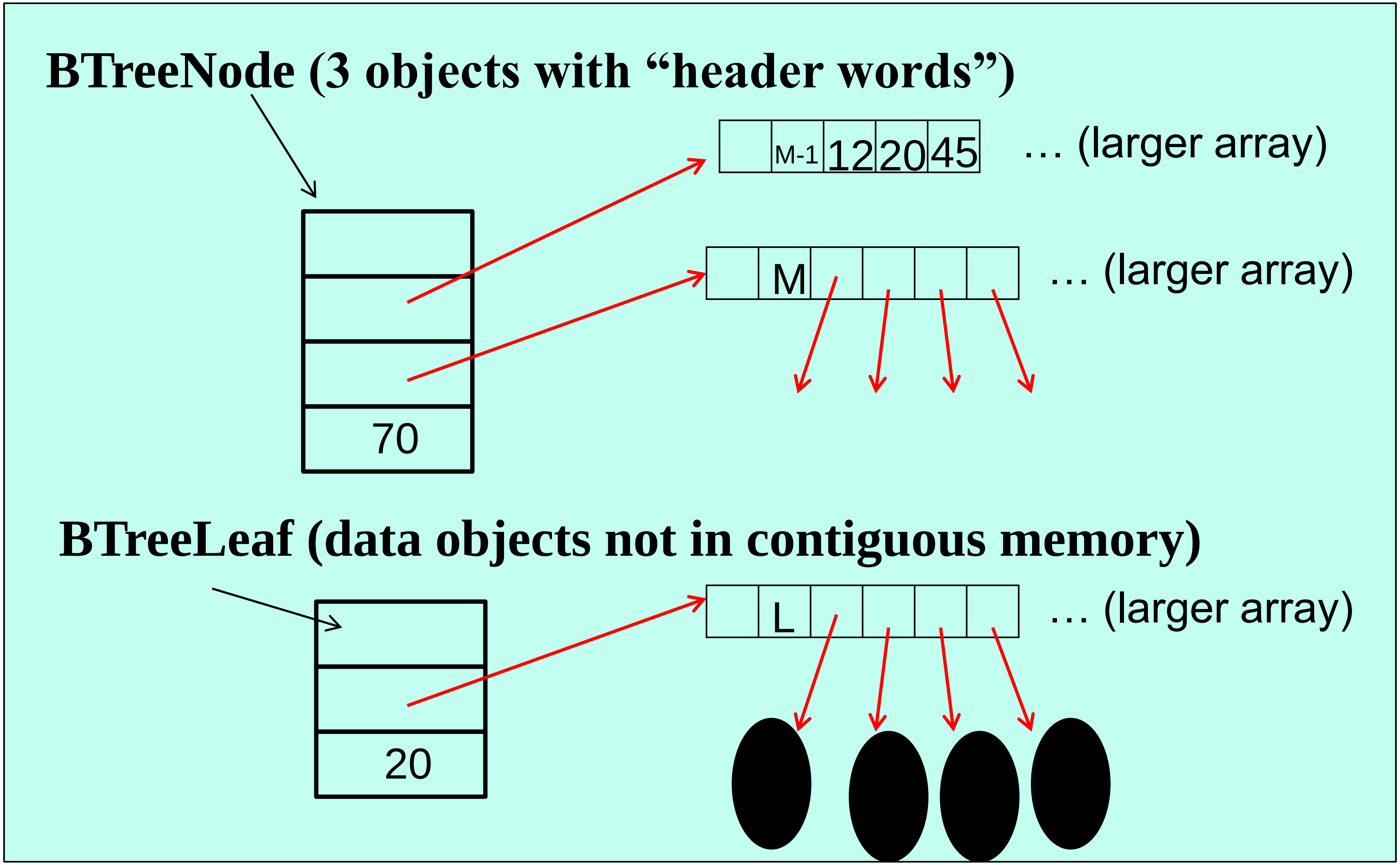
The key issue is extra *levels of indirection*…

Naïve Approach

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



The moral

- The point of B trees is to keep related data in contiguous memory
- All the red references on the previous slide are inappropriate
  - As minor point, beware the extra “header words”
- But that is “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 better support “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