CSE 332
Data Abstractions
B-Trees

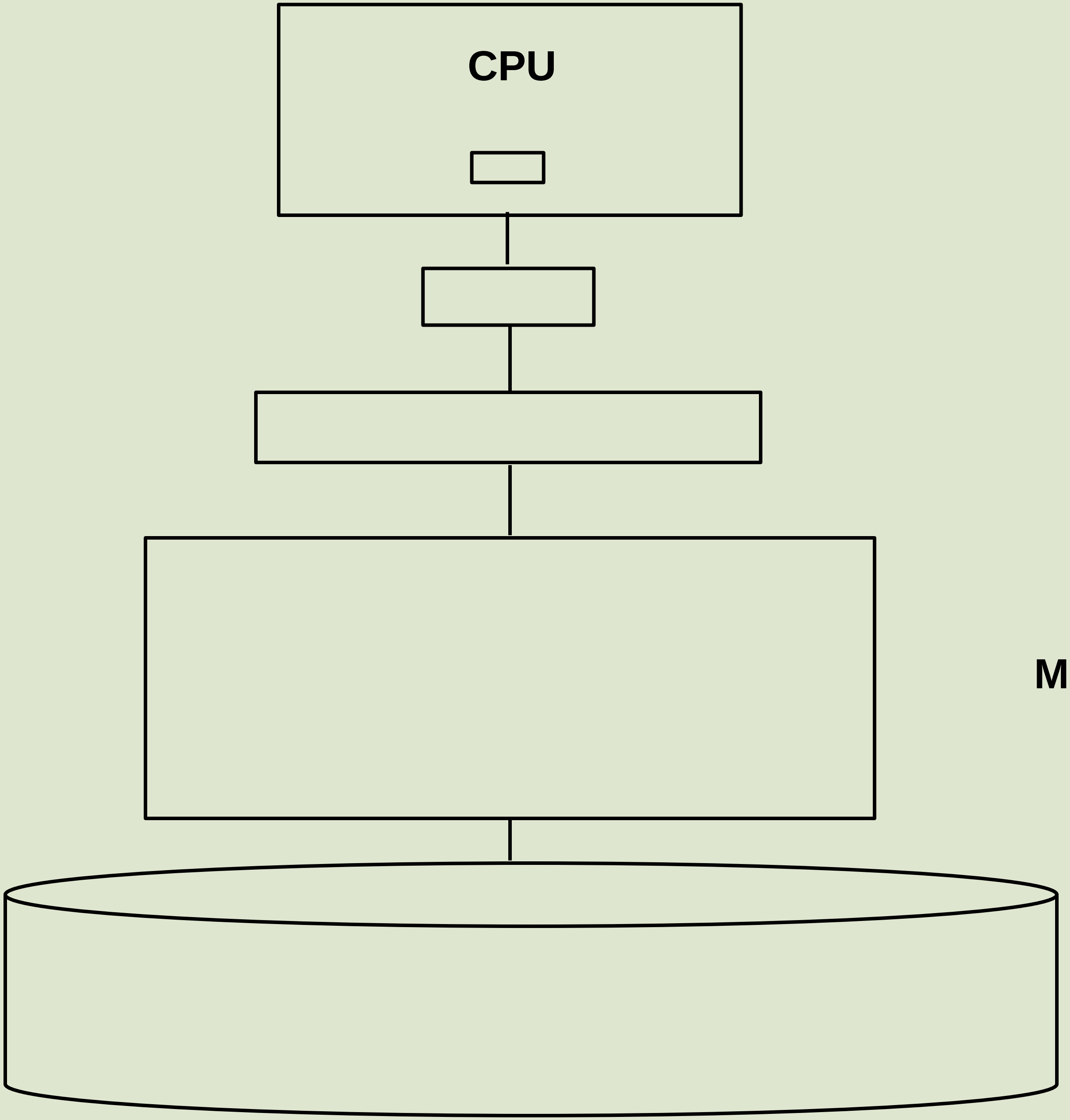
Richard Anderson
Spring 2016

Announcements

- Next two weeks: Hashing and sorting

- Upcoming dates
  - Friday, April 29. Midterm

Cycles to access:



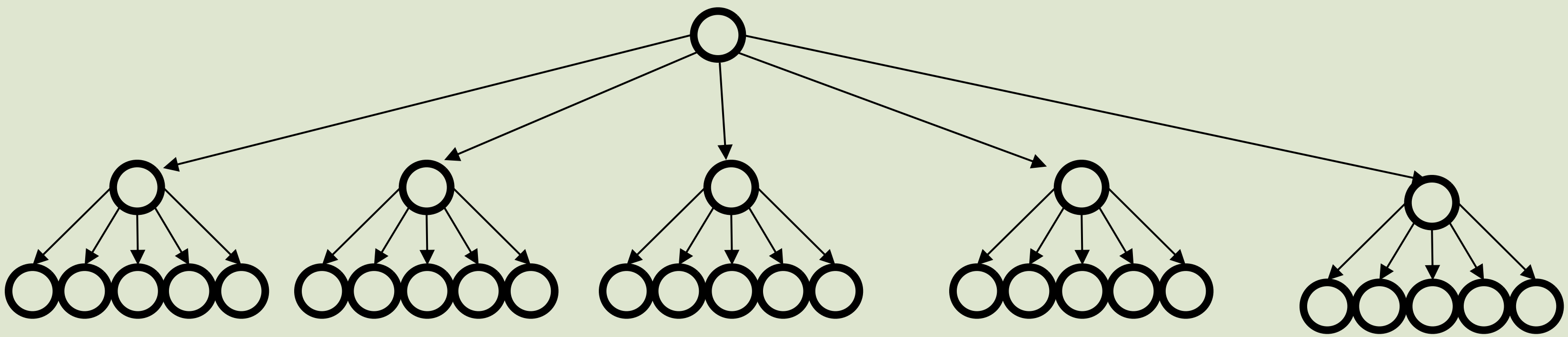
Registers	**1**
L1 Cache	**2**
L2 Cache	**30**
Main memory	**250**
Disk	

Random: 30,000,000

Streamed: 5000

M-ary Search Tree

Consider a search tree with branching factor *M*:



Complete tree has height:

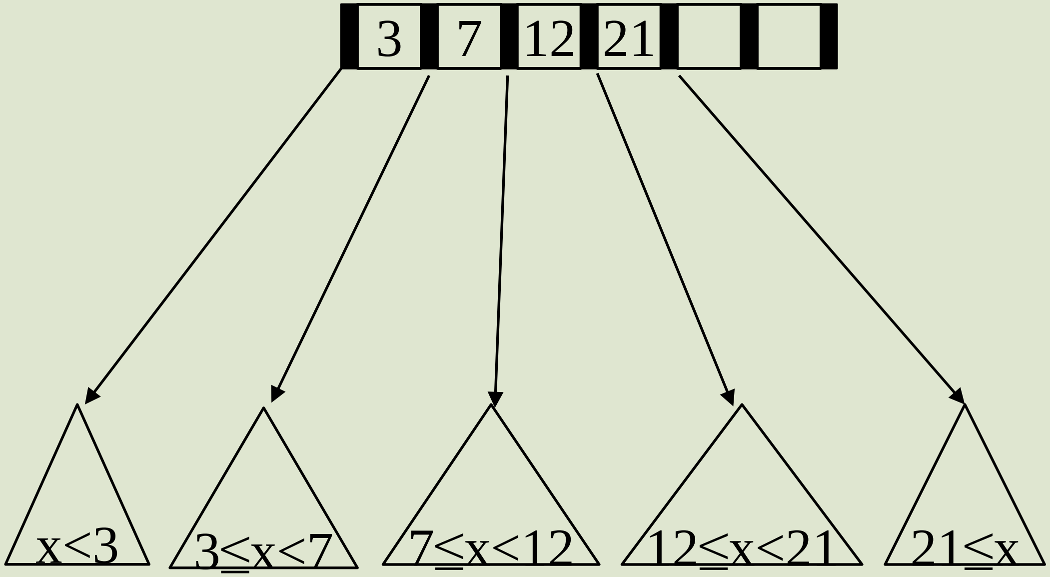
hops for *find*:

Runtime of *find*:

B+ Trees

(book calls these B-trees)

- Each internal node has (up to) M-1 keys:
- Order property:
  - subtree between two keys x and y contain leaves with *values* v such that $x \leq v < y$
  - Note the "$\leq$"
- Leaf nodes have up to L **sorted** keys.



B+ Tree Structure Properties

Internal nodes

- store up to M-1 keys
- have between $\lceil M/2 \rceil$ and M children

Leaf nodes

- where data is stored
- all at the same depth
- contain between $\lceil L/2 \rceil$ and L data items

Root (special case)

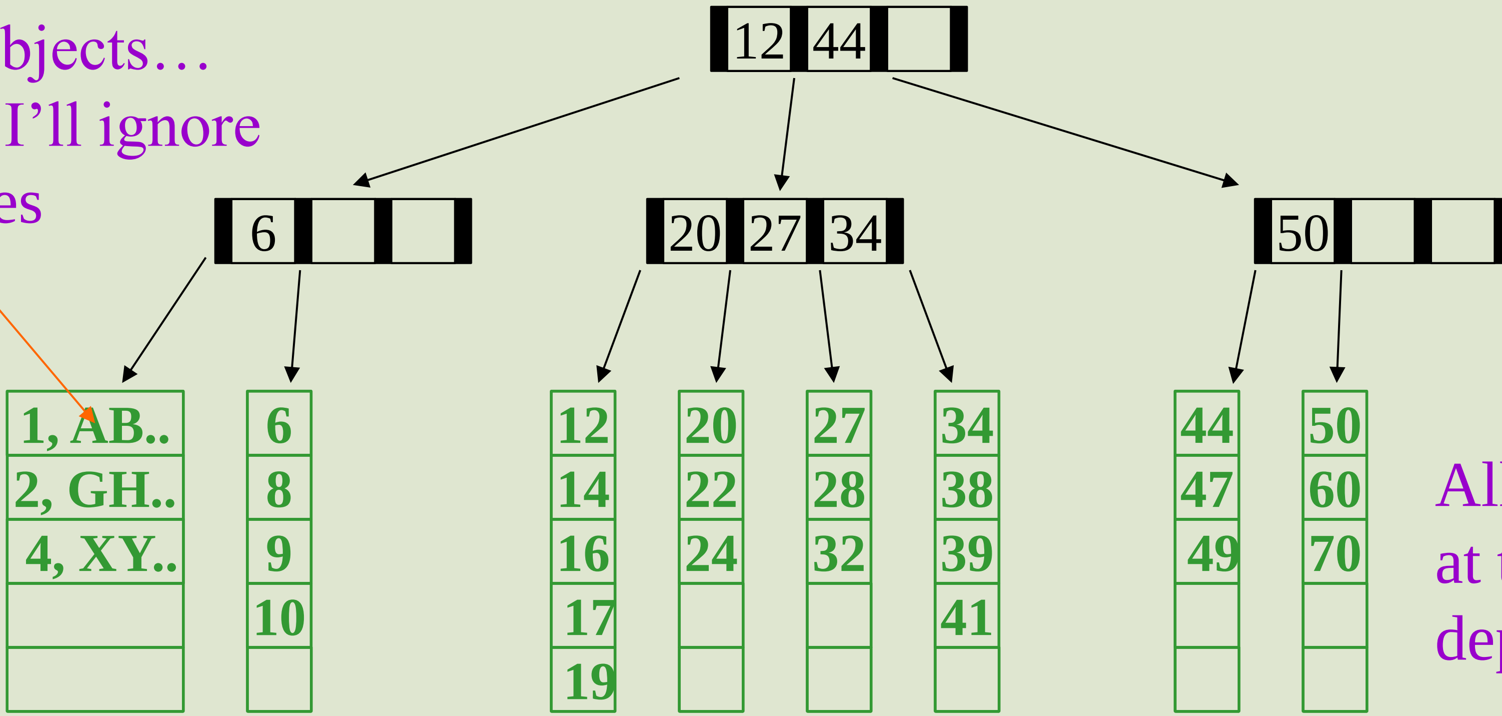
- has between 2 and ***M*** children (or root could be a leaf)

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 vs. AVL trees

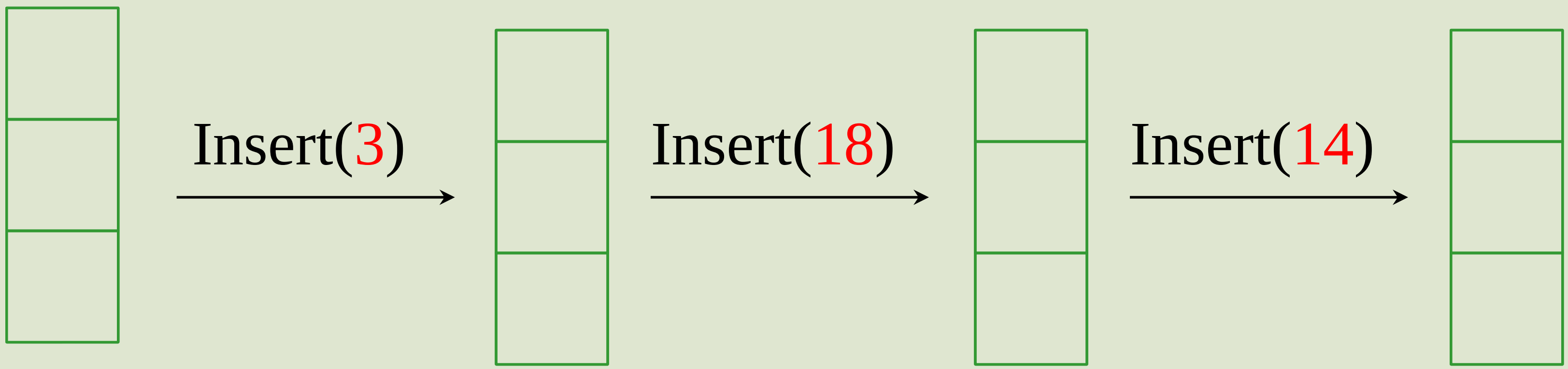
Suppose again we have $n = 2^{30} \approx 10^9$ items:

- Depth of AVL Tree

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

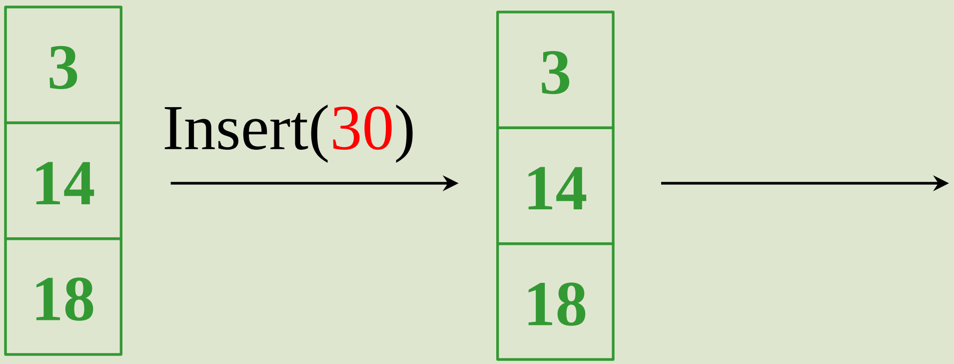
Great, but how to we actually make a B+ tree and keep it balanced…?

Building a B+ Tree with Insertions

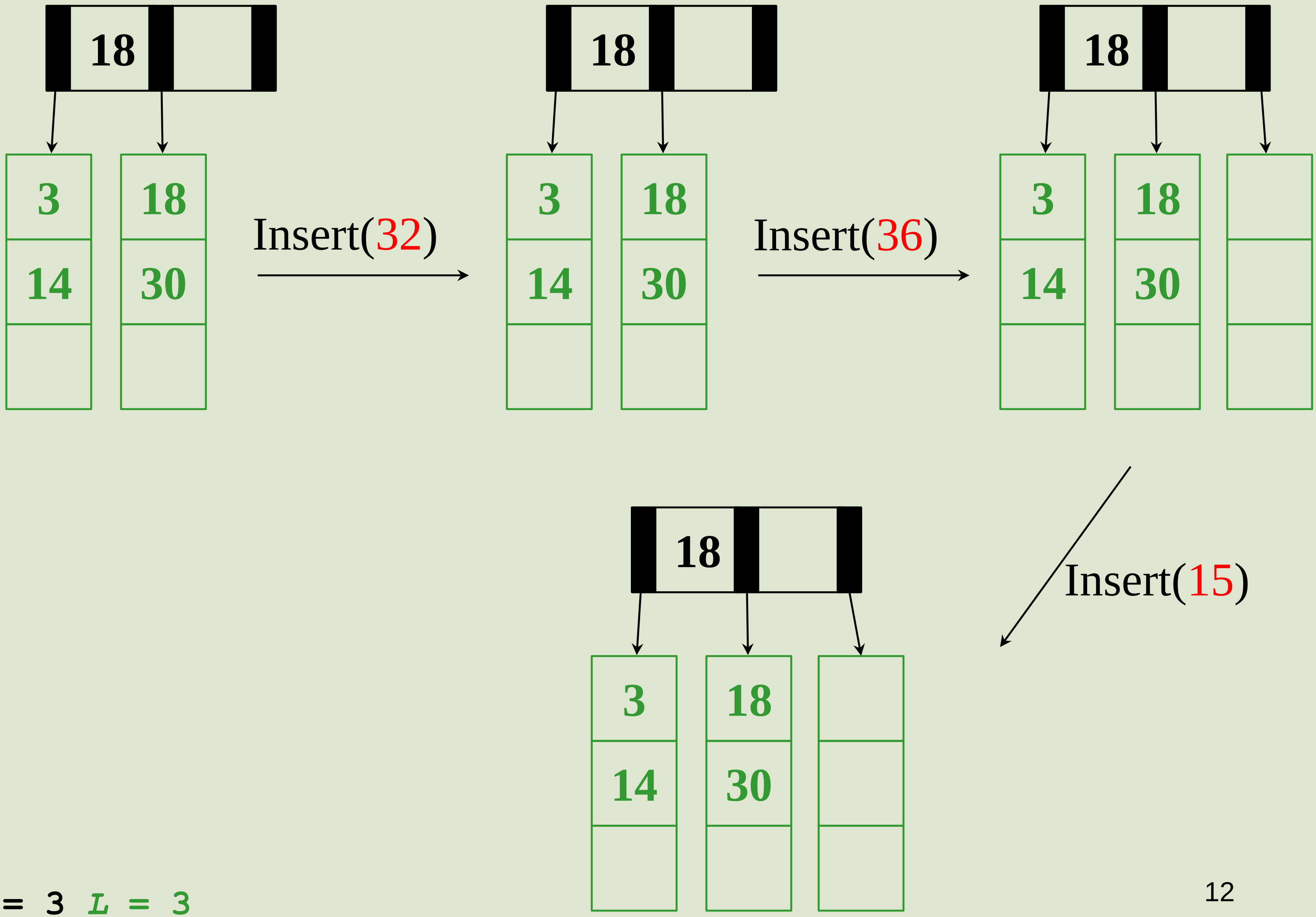


The empty
B-Tree

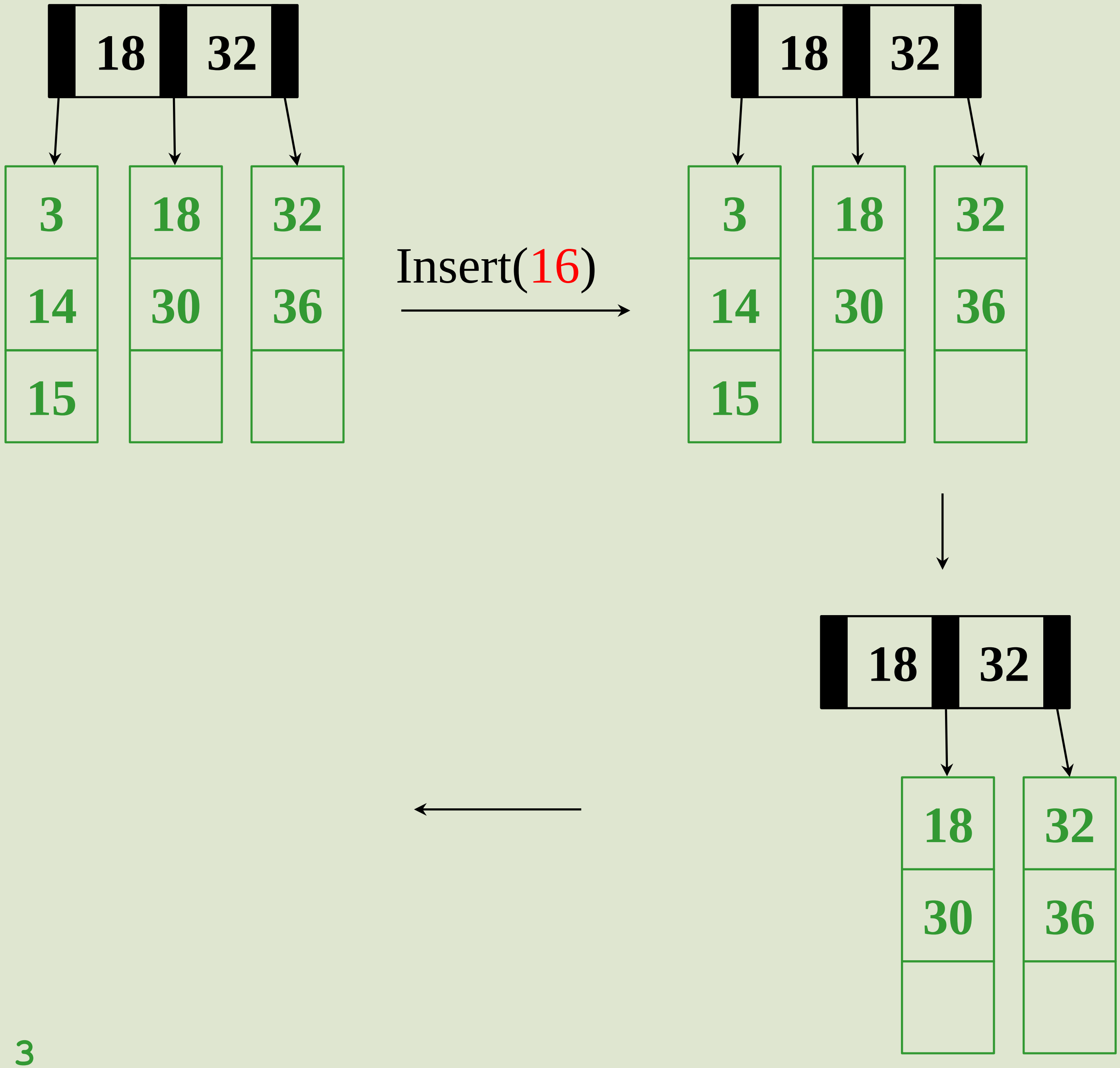
M **= 3** ***L*** **= 3**



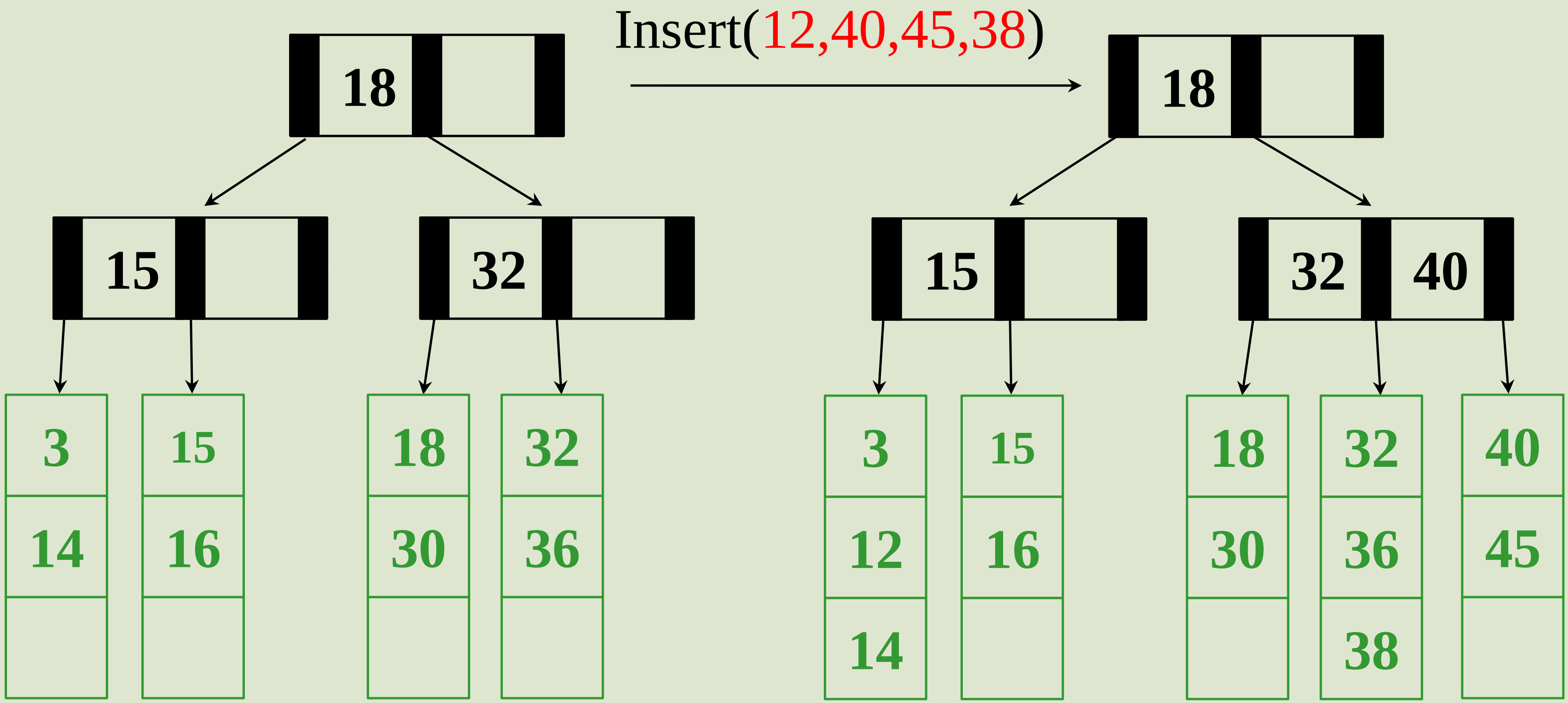
$M = 3$ $L = 3$


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

M = 3 *L* = 3


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

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

$M = 3$ $L = 3$

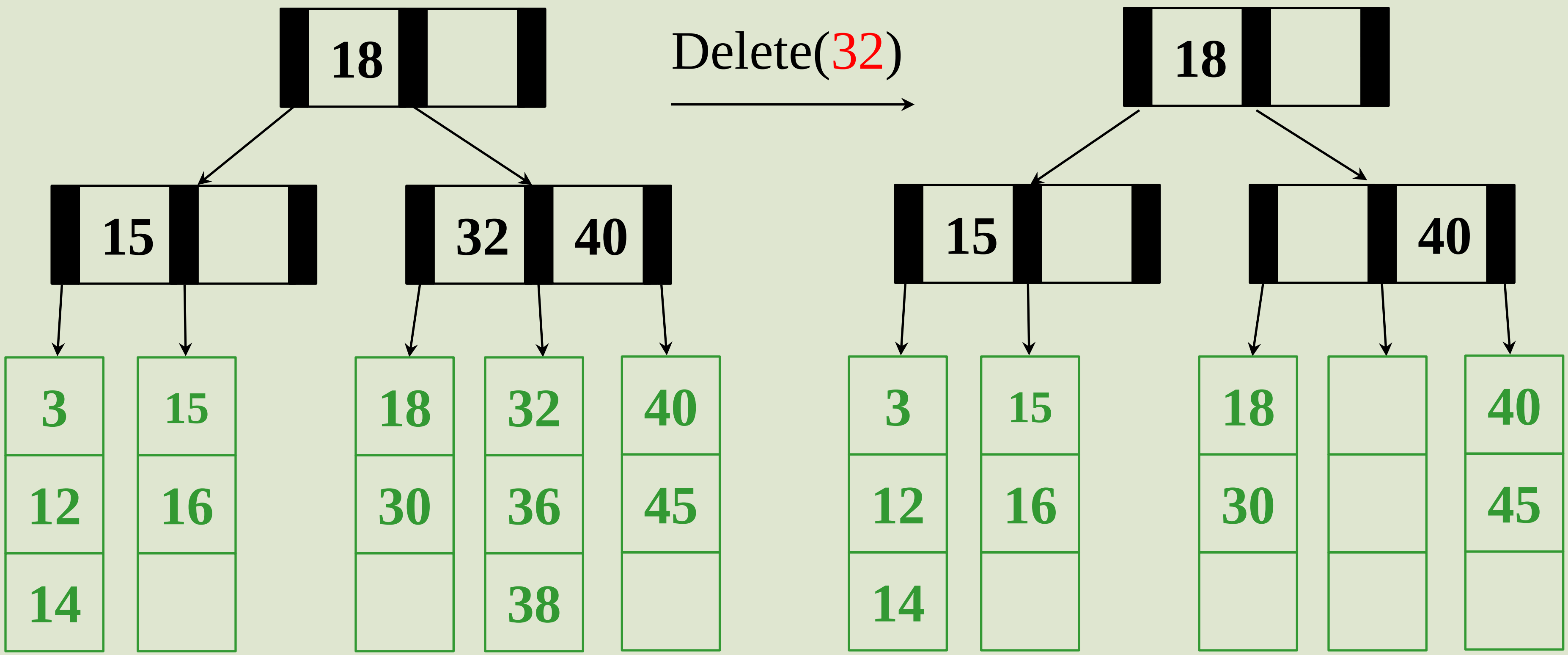
Insertion Algorithm

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**!

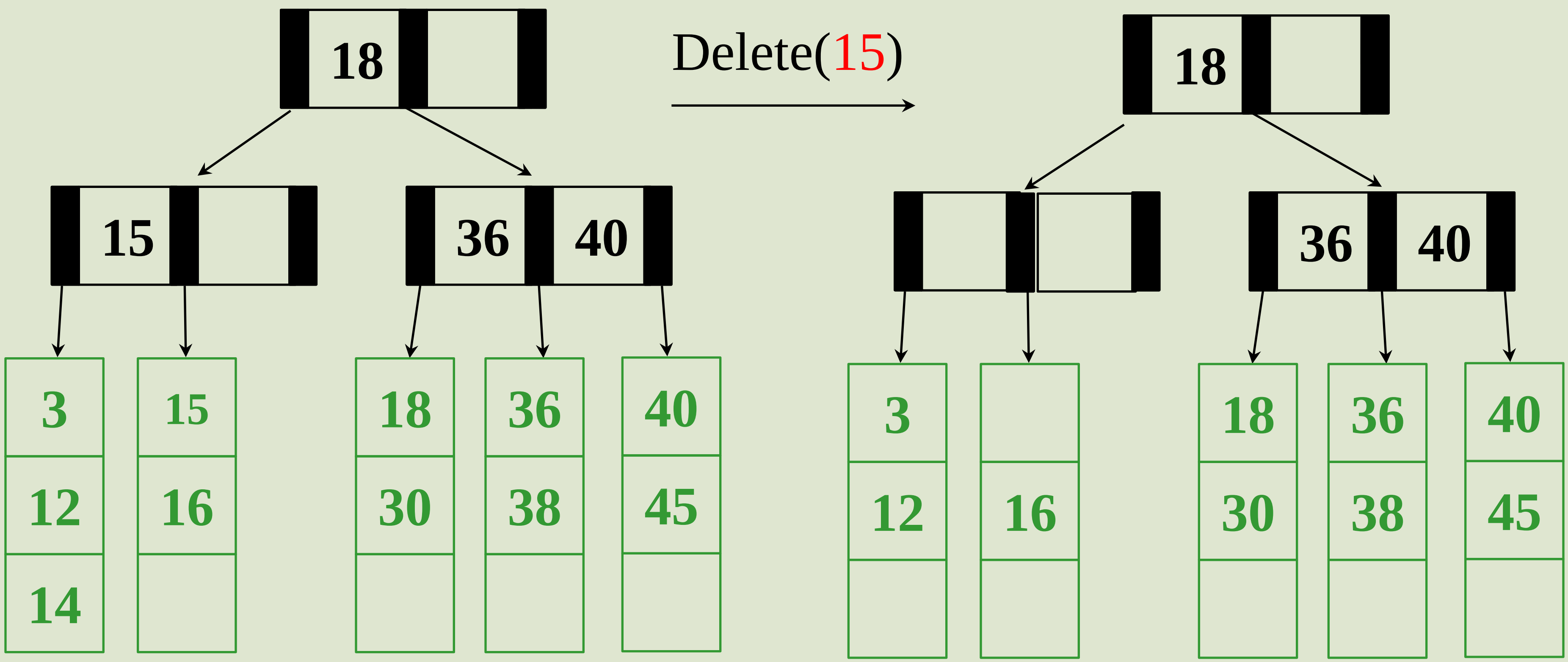
This makes the tree deeper!

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.

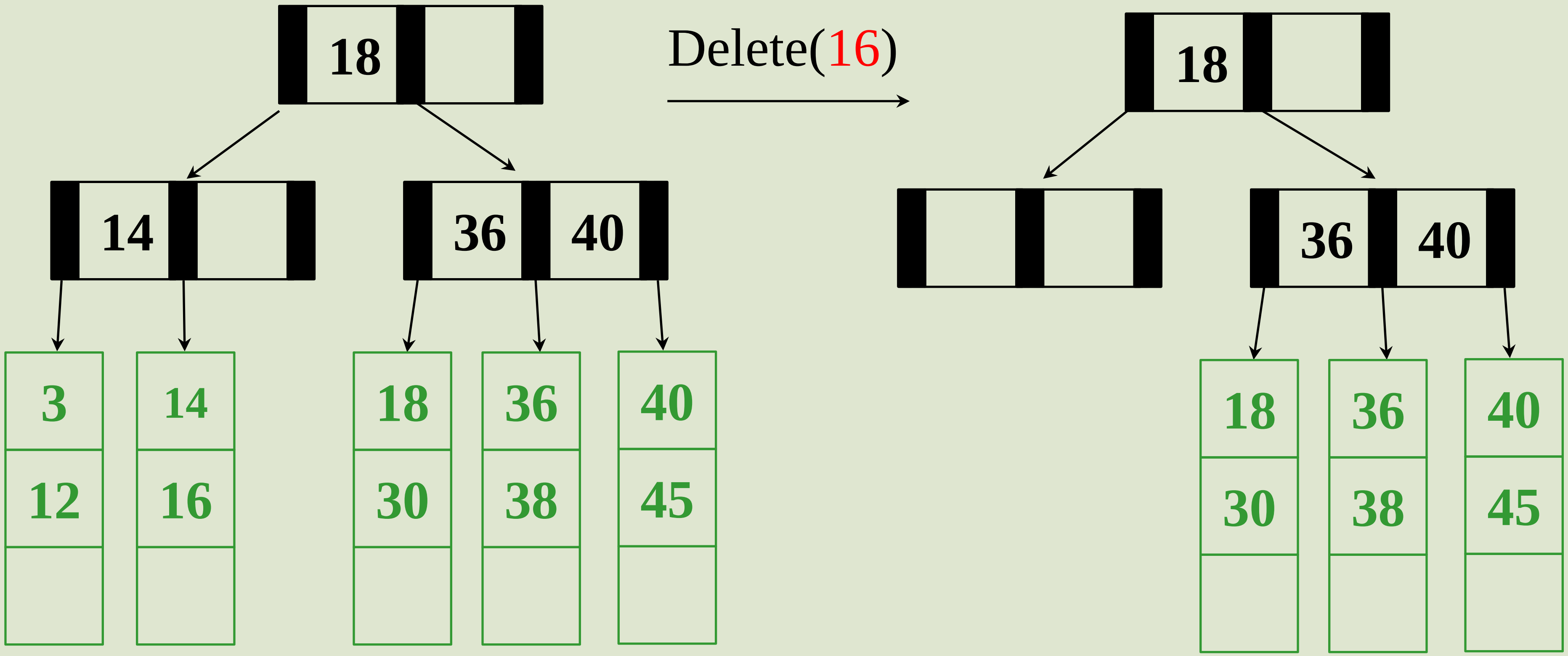
And Now for Deletion…



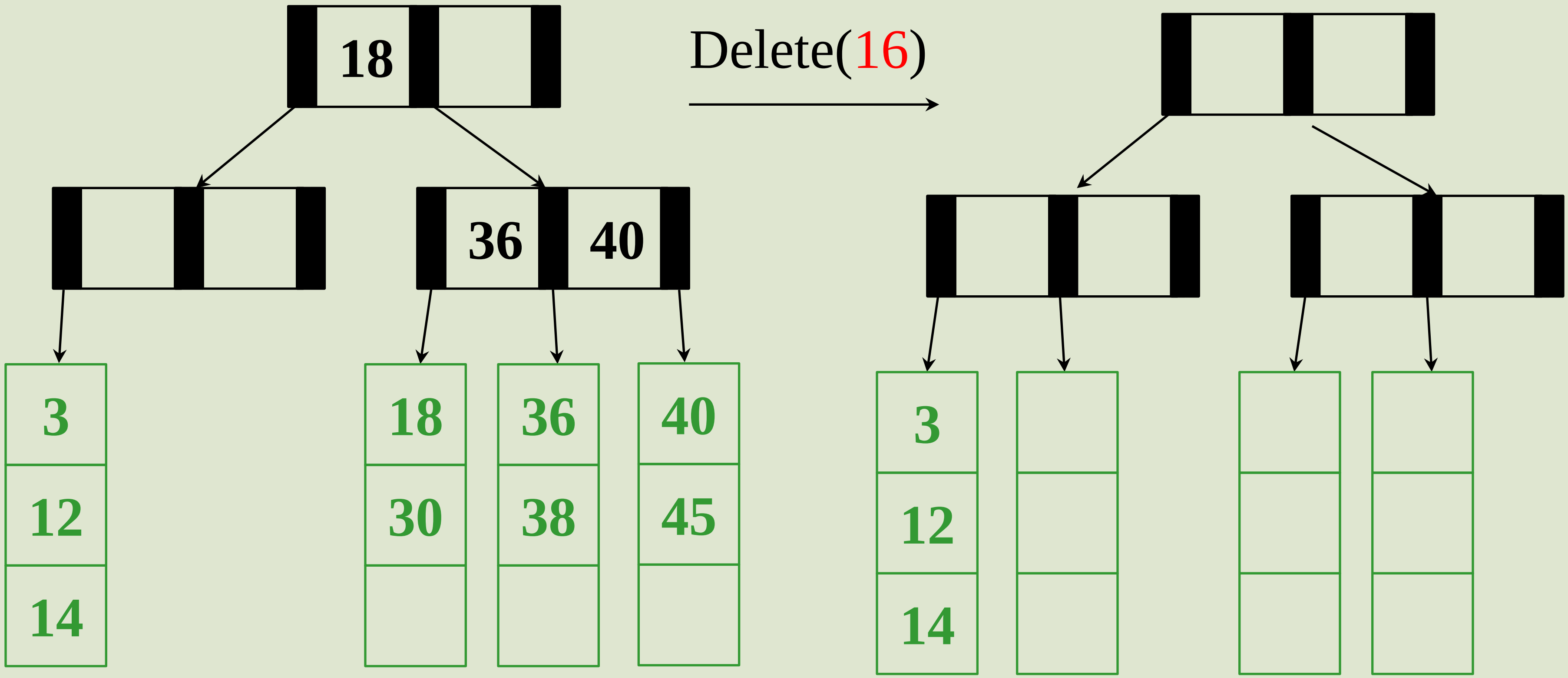
M = 3 *L* = 3


18
15
36
40
3
12
14
15
16
18
30
36
38
40
45
Delete(15)
18
36
40
3
12
16
18
30
36
38
40
45

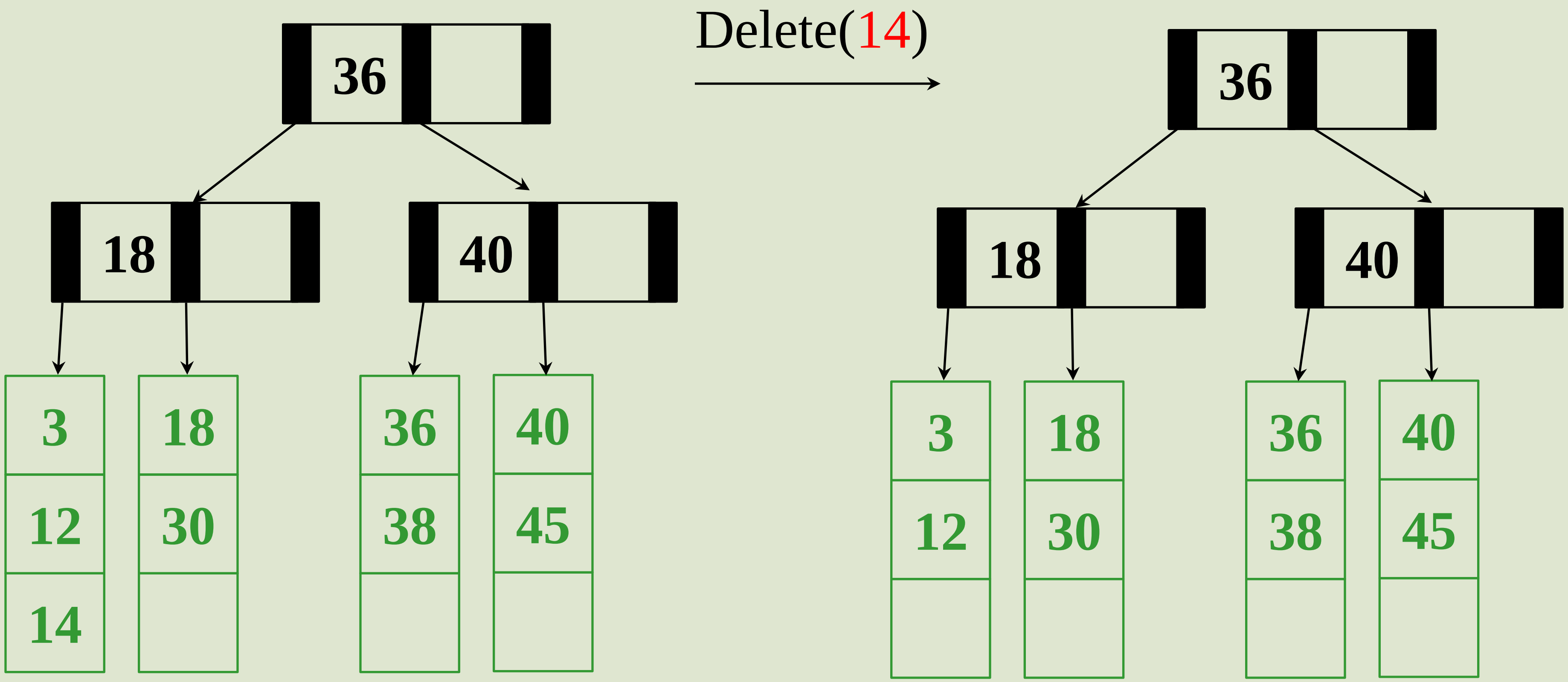
$M = 3$ $L = 3$


18
Delete(16)
14
36 40
3
12
14
16
18
30
36
38
40
45
18
36 40
18
30
36
38
40
45

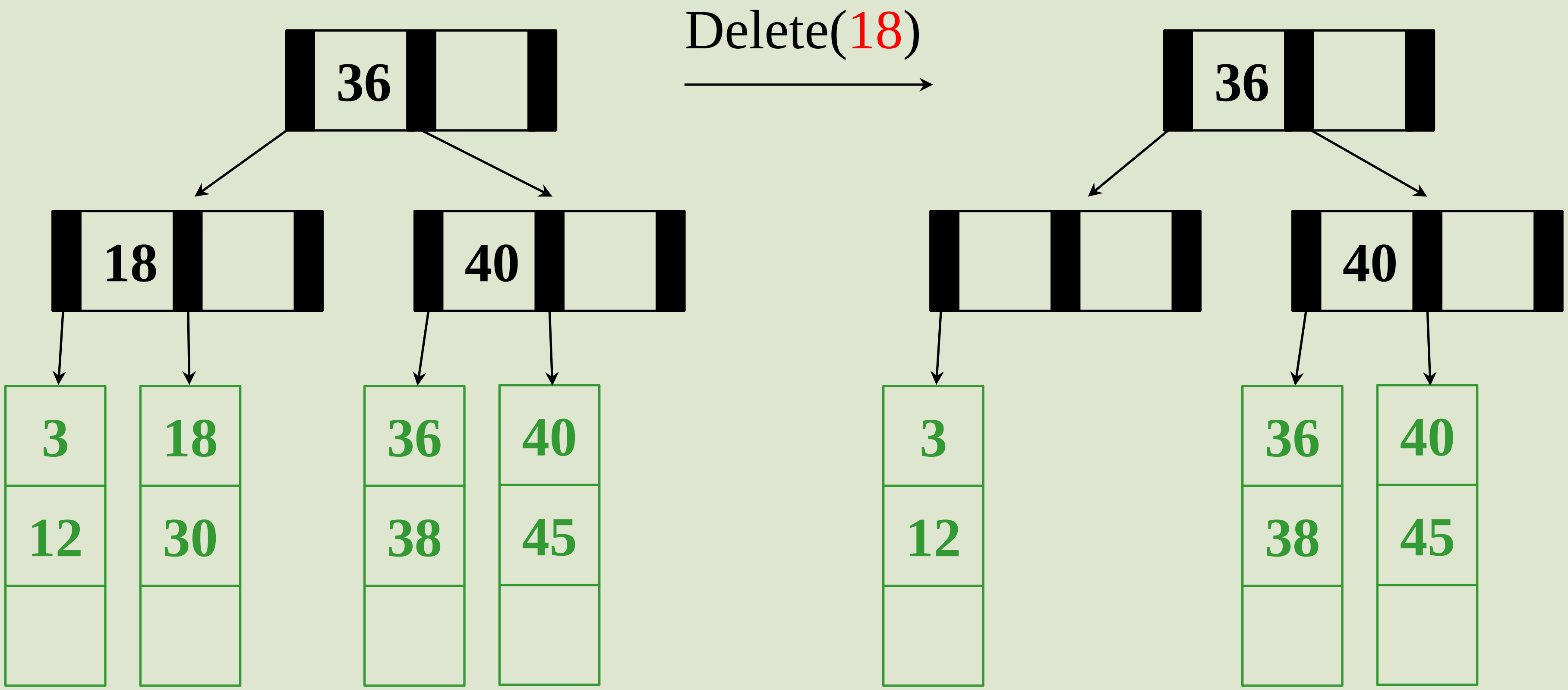
M = 3 *L* = 3



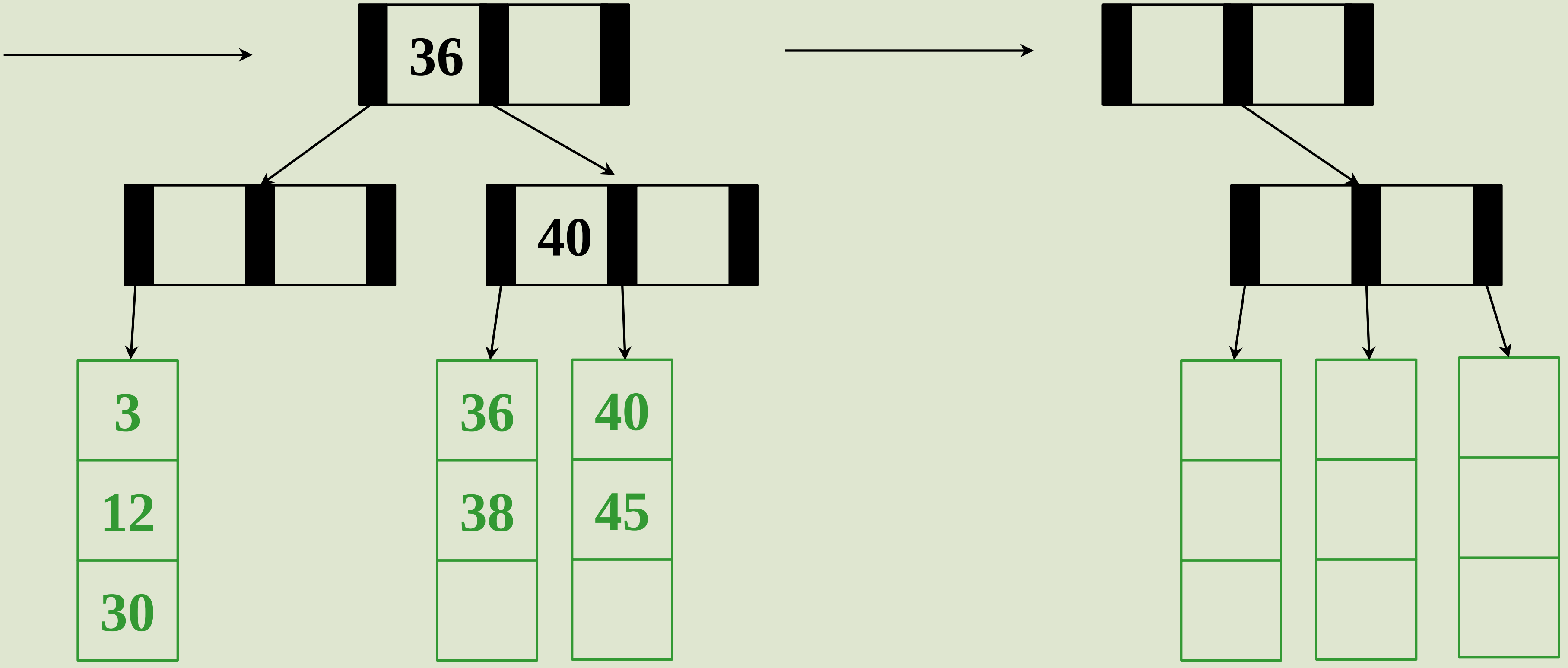
$M = 3$ $L = 3$


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

M = 3 *L* = 3



M = 3 L = 3



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

Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than $\lceil L/2 \rceil$ items, **underflow**!
   - Adopt data from a neighbor; update the parent
   - If adopting won't work, delete node and merge with neighbor
   - If the parent ends up with fewer than $\lceil M/2 \rceil$ children, **underflow**!

Deletion Slide Two

3. If an internal node ends up with fewer than $\lceil M/2 \rceil$ children, **underflow**!
   – Adopt from a neighbor; update the parent
   – If adoption won't work, merge with neighbor
   – If the parent ends up with fewer than $\lceil M/2 \rceil$ children, **underflow**!

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

5. Propagate keys up through tree.

This reduces the height of the tree!

Thinking about B+ Trees

- B+ Tree insertion can cause (expensive) splitting and propagation up the tree
- B+ Tree deletion can cause (cheap) adoption or (expensive) merging and propagation up the tree
- Split/merge/propagation is rare if ***M*** and ***L*** are large *(Why?)*
- Pick branching factor ***M*** and data items/leaf ***L*** such that each node takes one full page/block of memory/disk.

Complexity

- Find:
- Insert:
  - find:
  - Insert in leaf:
  - split/propagate up:

- Claim: O(M) costs are negligible

Tree Names You Might Encounter

- “B-Trees”
  - More general form of B+ trees, allows data at internal nodes too
  - Range of children is (key1,key2) rather than [key1, key2)
- B-Trees with ***M* = 3**, ***L* = x** are called **2-3 trees**
  - Internal nodes can have 2 or 3 children
- B-Trees with ***M* = 4**, ***L* = x** are called **2-3-4 trees**
  - Internal nodes can have 2, 3, or 4 children