

# CSE 332: Data Structures and Parallelism

Spring 2022

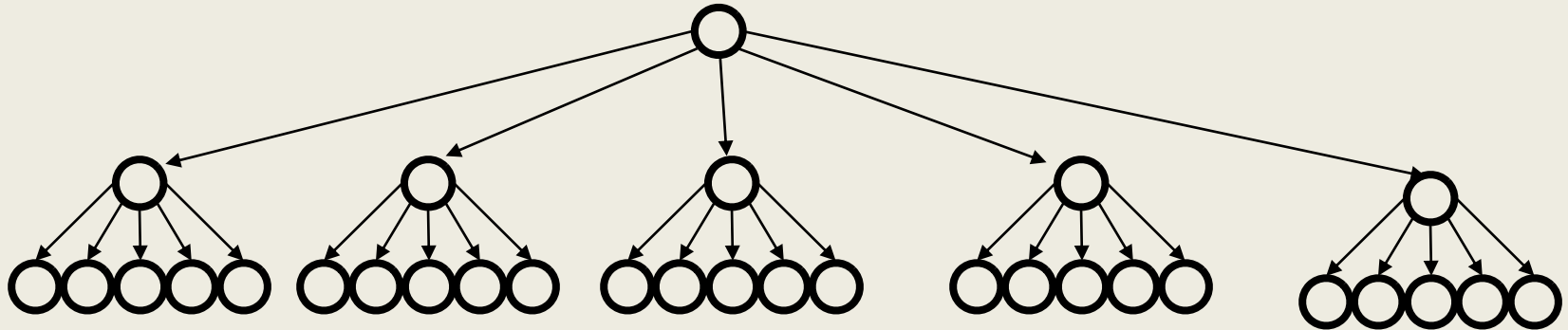
Richard Anderson

Lecture 10: B-Trees

# Announcements

# M-ary Search Tree

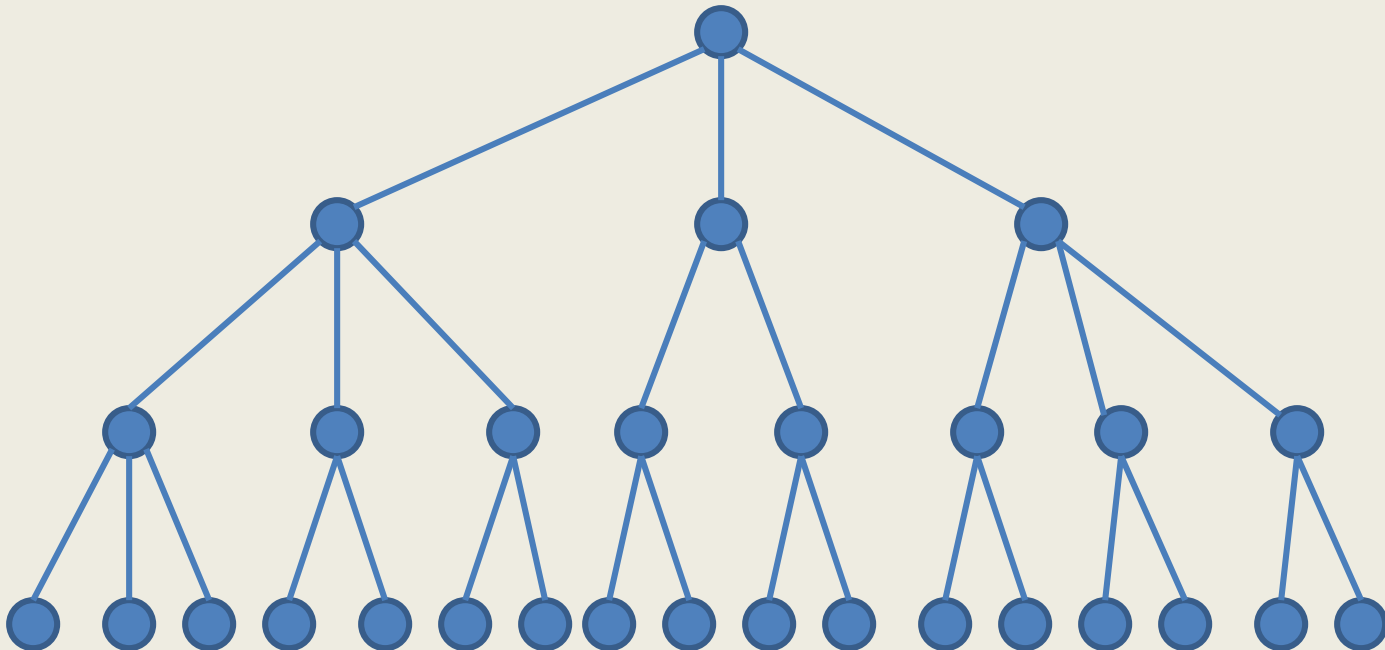
Consider a search tree with branching factor  $M$ :



- Complete tree has height:
- # hops for *find*:
- Runtime of *find*:

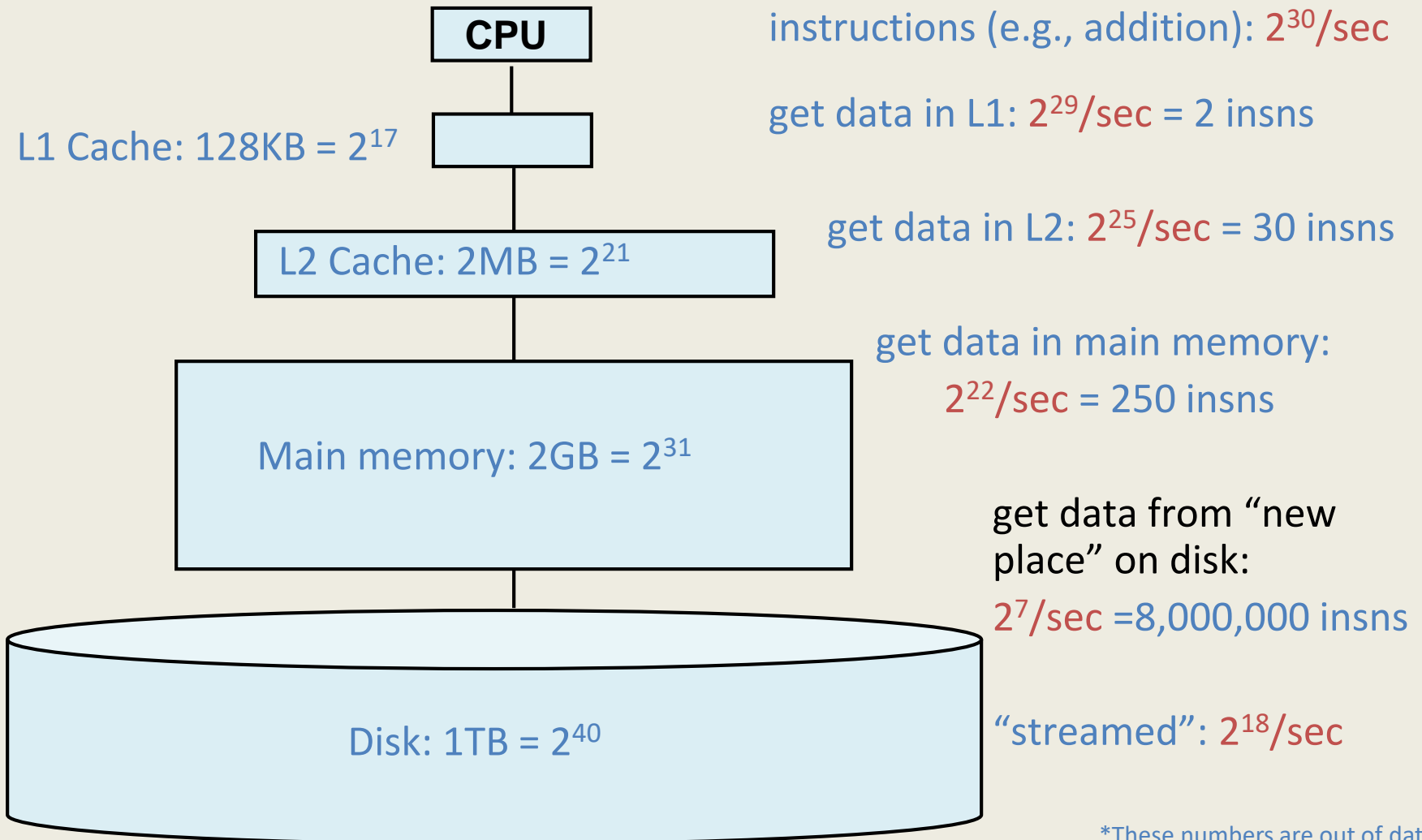
# B Trees

- Can balance a tree by varying the depth of the leaves, or by varying the number of children of the nodes



# A typical hierarchy

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



\*These numbers are out of date

It is much faster to do:

5 million arithmetic ops

2500 L2 cache accesses

400 main memory accesses

Than:

1 disk access

1 disk access

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*

# Usually, it doesn't matter . . .

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...

# Block/line size

- Moving data up the memory hierarchy is slow because of *latency* (think distance-to-travel)
  - May as well send more than just the one int/reference asked for (think “giving friends a car ride doesn’t slow you down”)
  - Sends nearby memory because:
    - It is easy
    - Likely to be used soon (think fields/arrays)
- Amount of data moved from disk into memory called the “block” size or the “page” size
  - Not under program control
- Amount of data moved from memory into cache called the “line” size
  - Not under program control

Principle of *Locality*

# 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 mean “good”
  - Binary heaps “make big jumps” to percolate (different block)

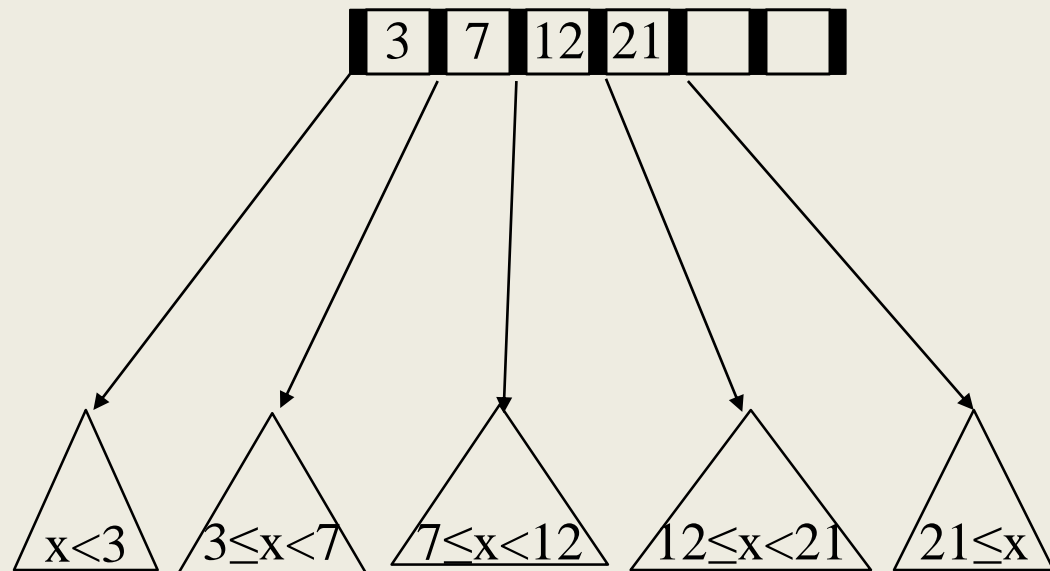
# BSTs?

- Looking things up in balanced binary search trees is  $O(\log n)$ , so even for  $n = 2^{39}$  (512GB) we need not worry about minutes or hours
- Still, number of disk accesses matters
  - AVL tree could have height of 55
  - So each **find** could take about 0.5 seconds or about 100 finds a minute
  - Most of the nodes will be on disk: the tree is shallow, but it is still many gigabytes big so the *tree* cannot fit in memory
    - Even if memory holds the first 25 nodes on our path, we still need 30 disk accesses

# 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

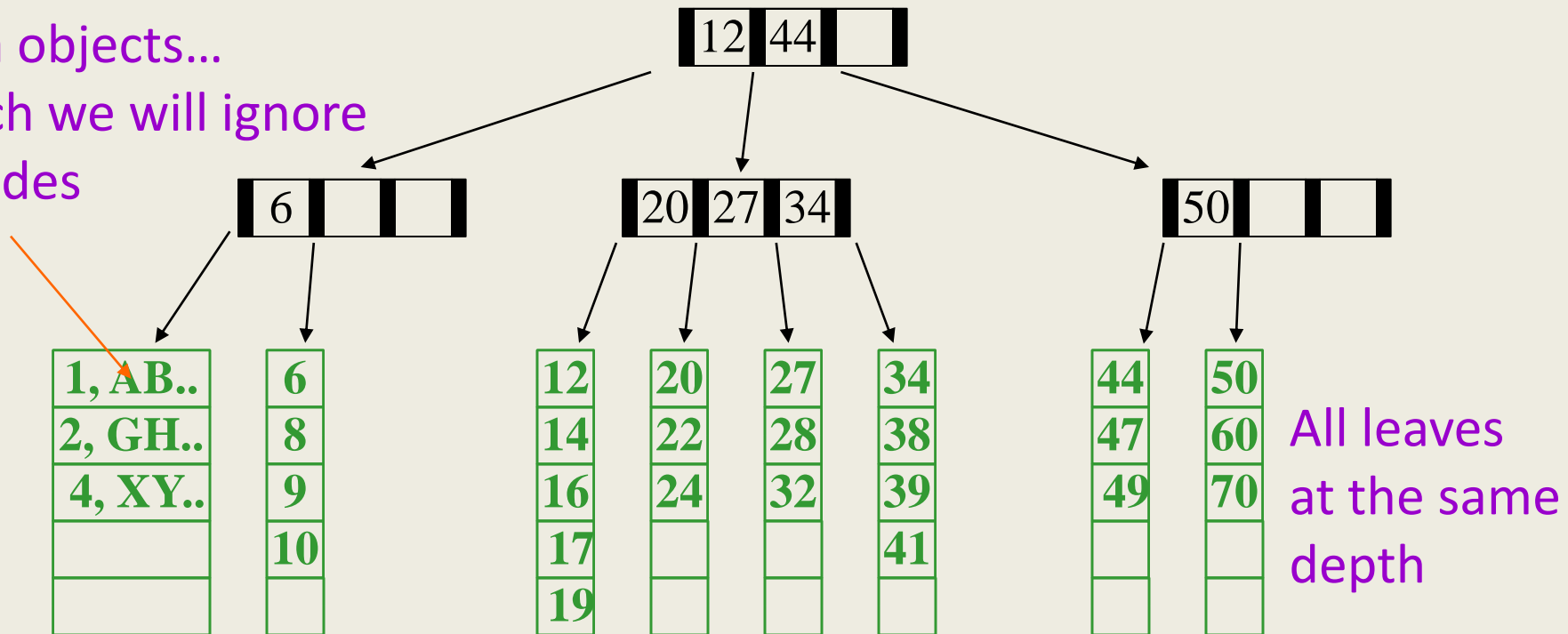
## Root (special case)

- 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 we will 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;

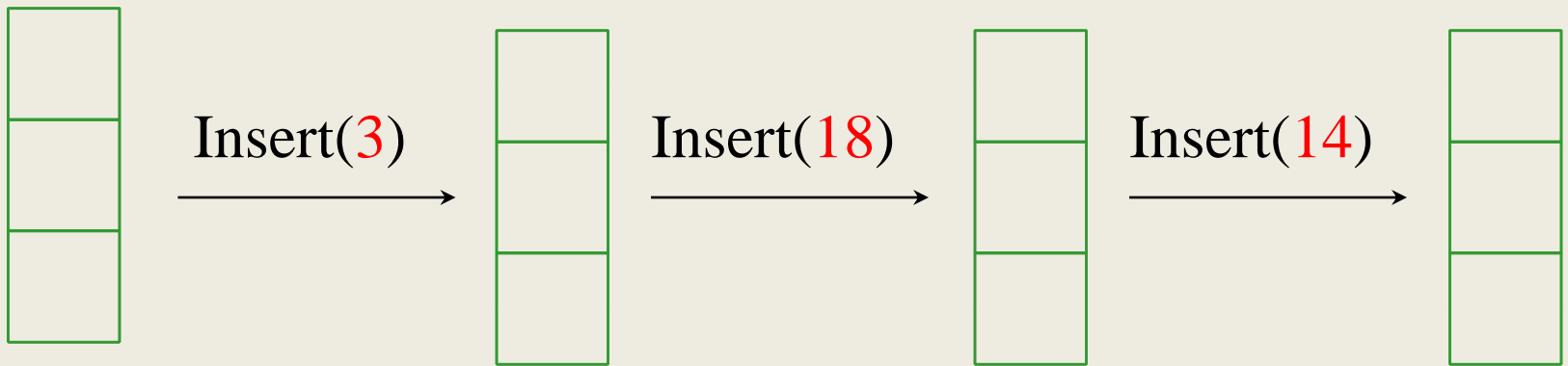
### **Only leaf nodes contain keys and actual *data***

- Much of tree structure can be loaded into memory irrespective of data object size
- Data actually resides in disk

# B+ trees vs. AVL trees

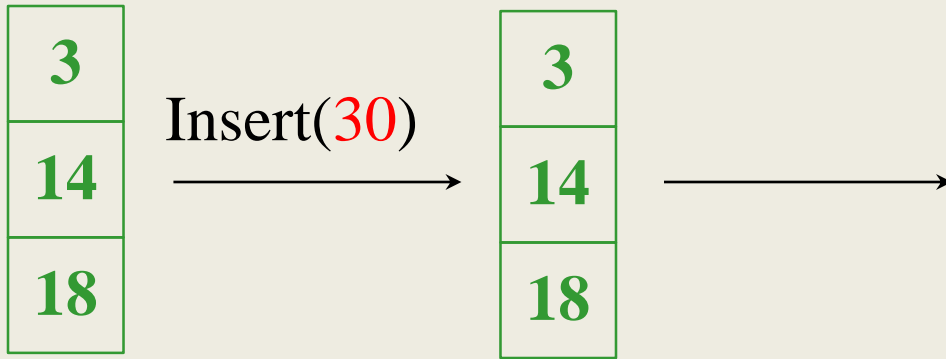
- Suppose again we have  $n = 2^{30} \approx 10^9$  items:
- Depth of AVL Tree
- Depth of B+ Tree with  $M = 256$ ,  $L = 256$
- Great, but how to we actually make a B+ tree and keep it balanced...?

# Building a B+ Tree with Insertions

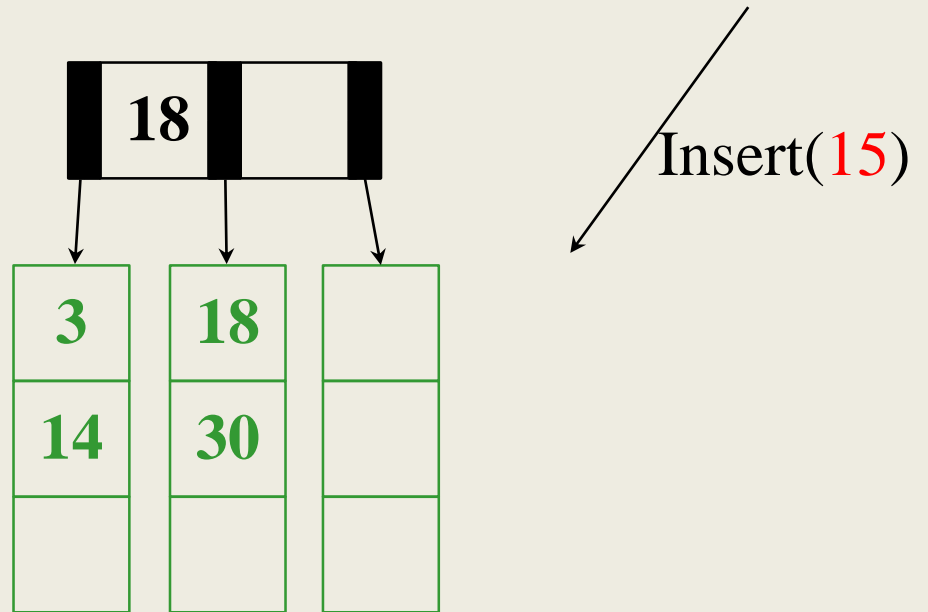
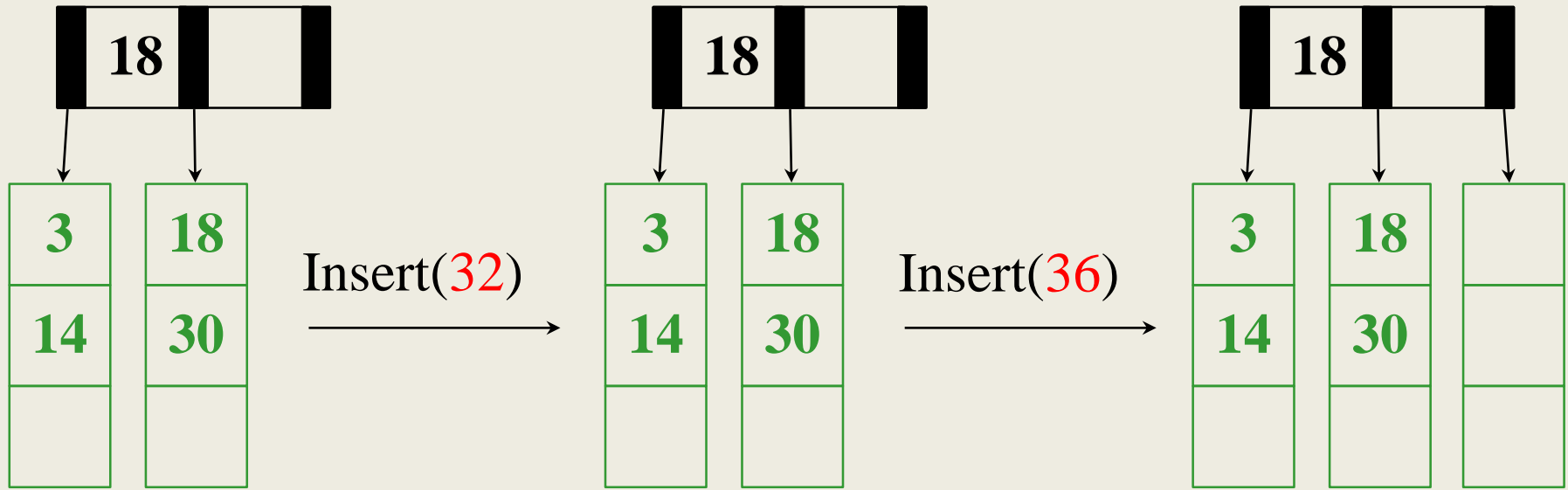


The empty  
B-Tree

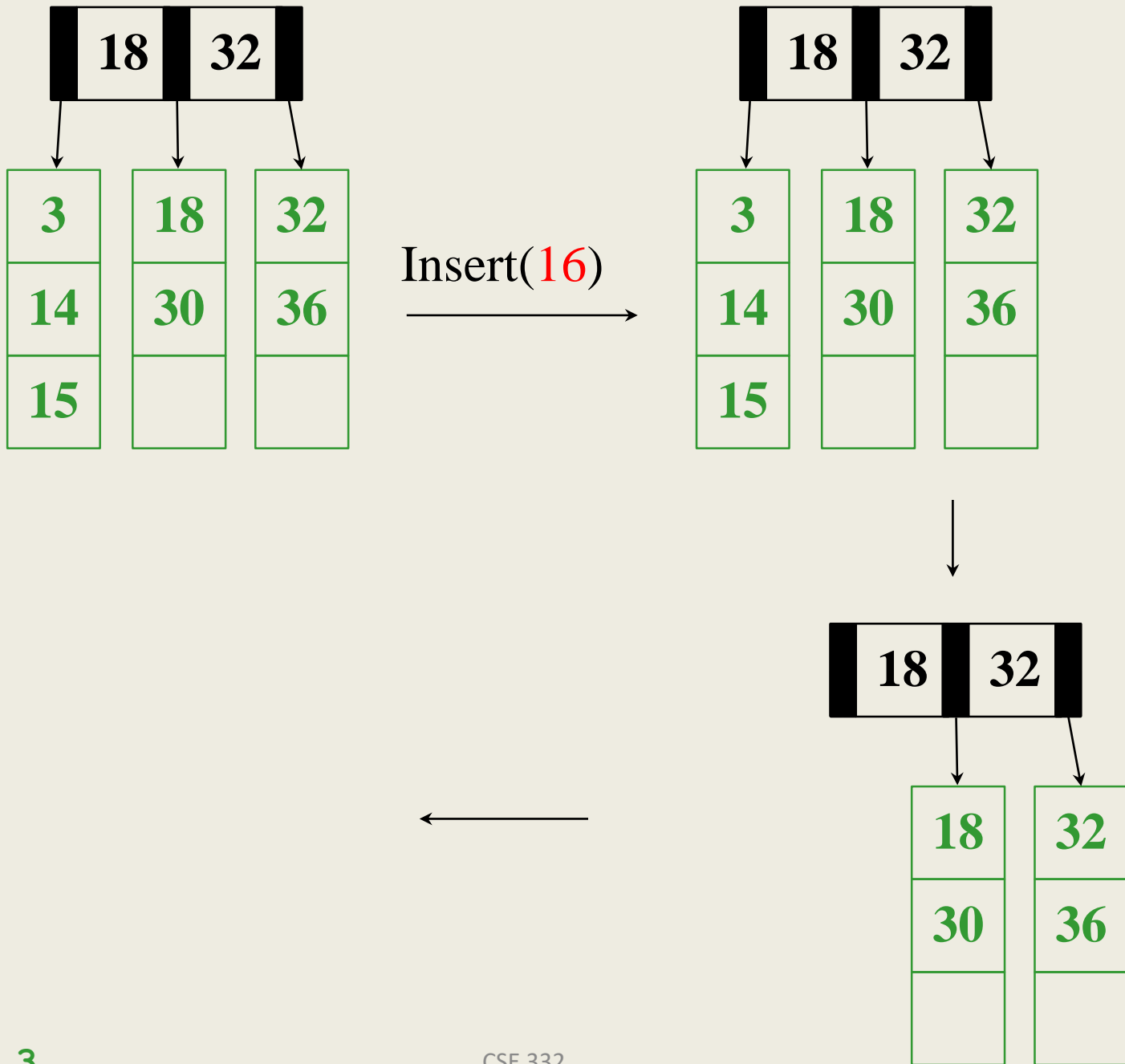
$M = 3$   $L = 3$

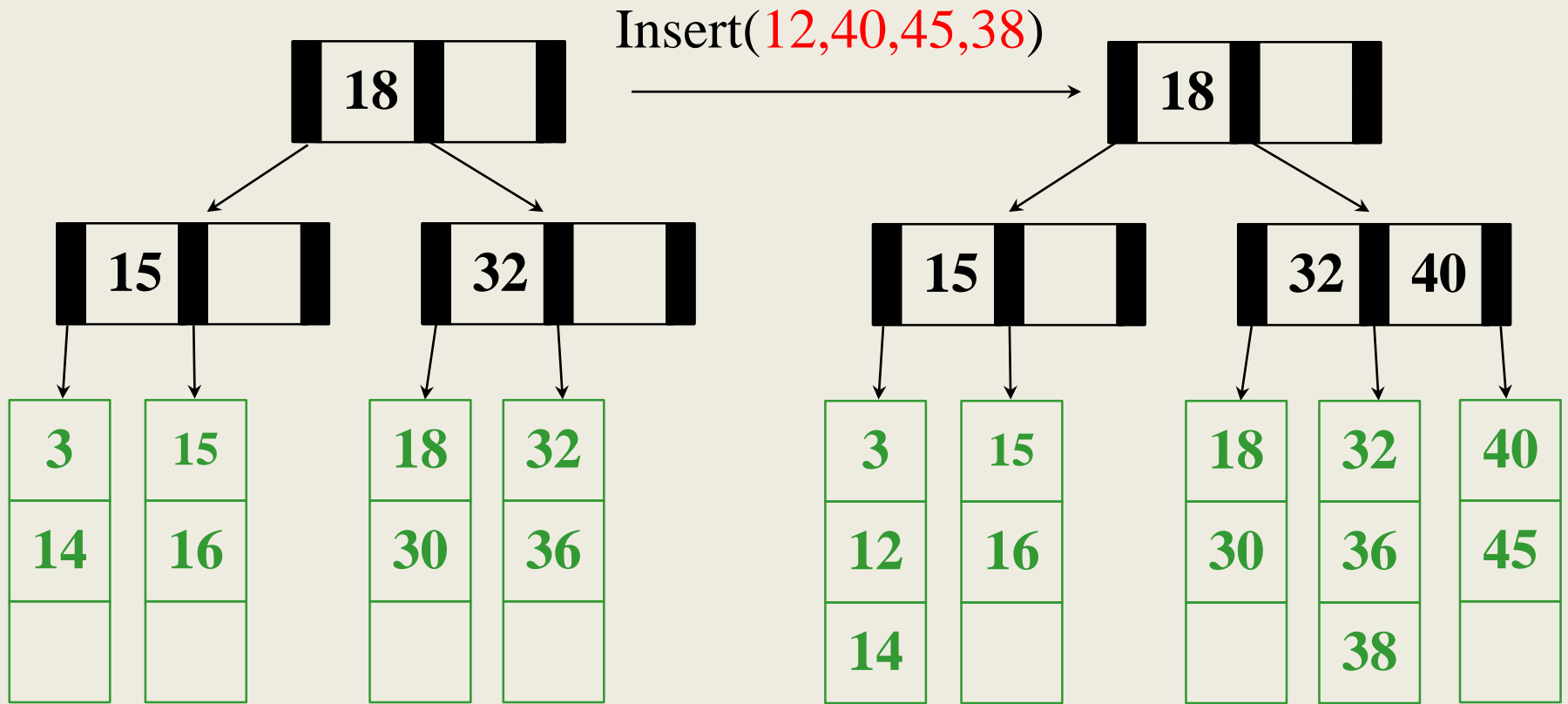


$M = 3$ ,  $L = 3$   
4/18/2022



$M = 3, L = 3$   
4/18/2022





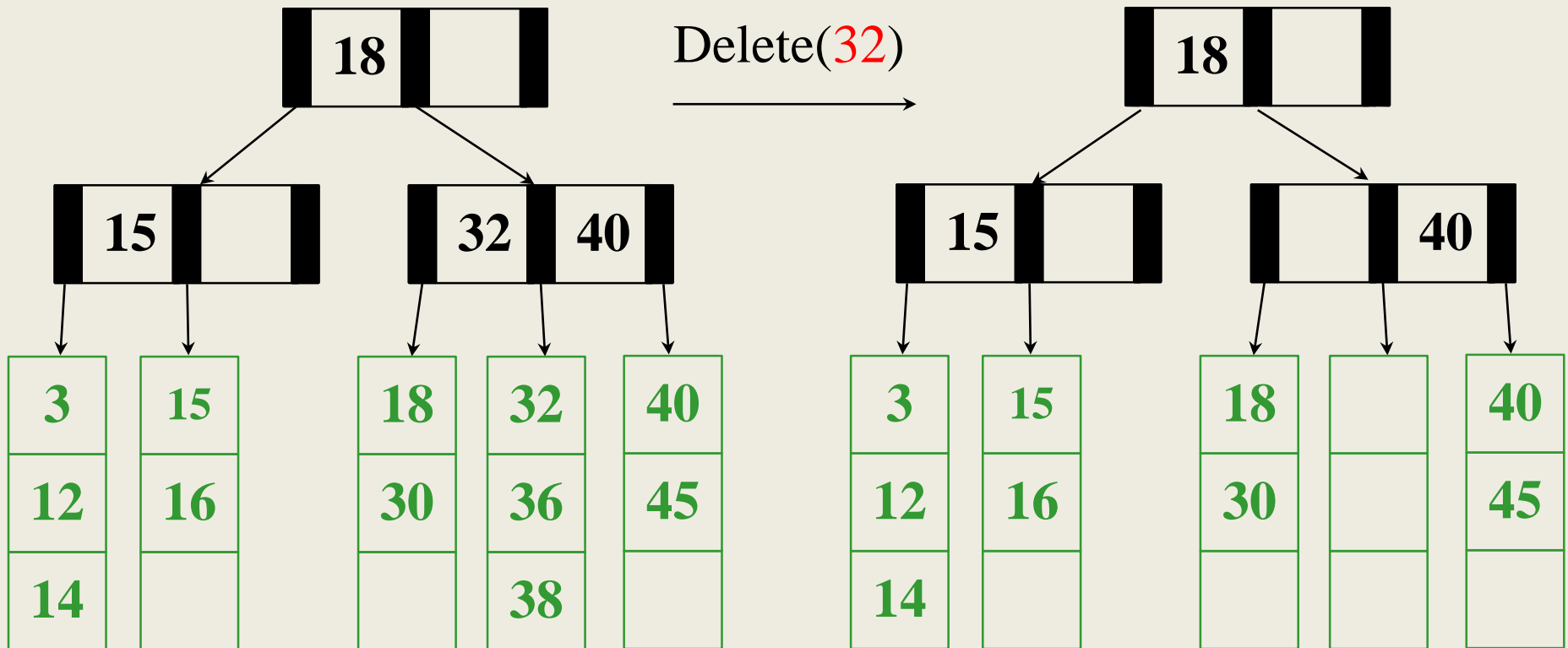
# Insertion Algorithm

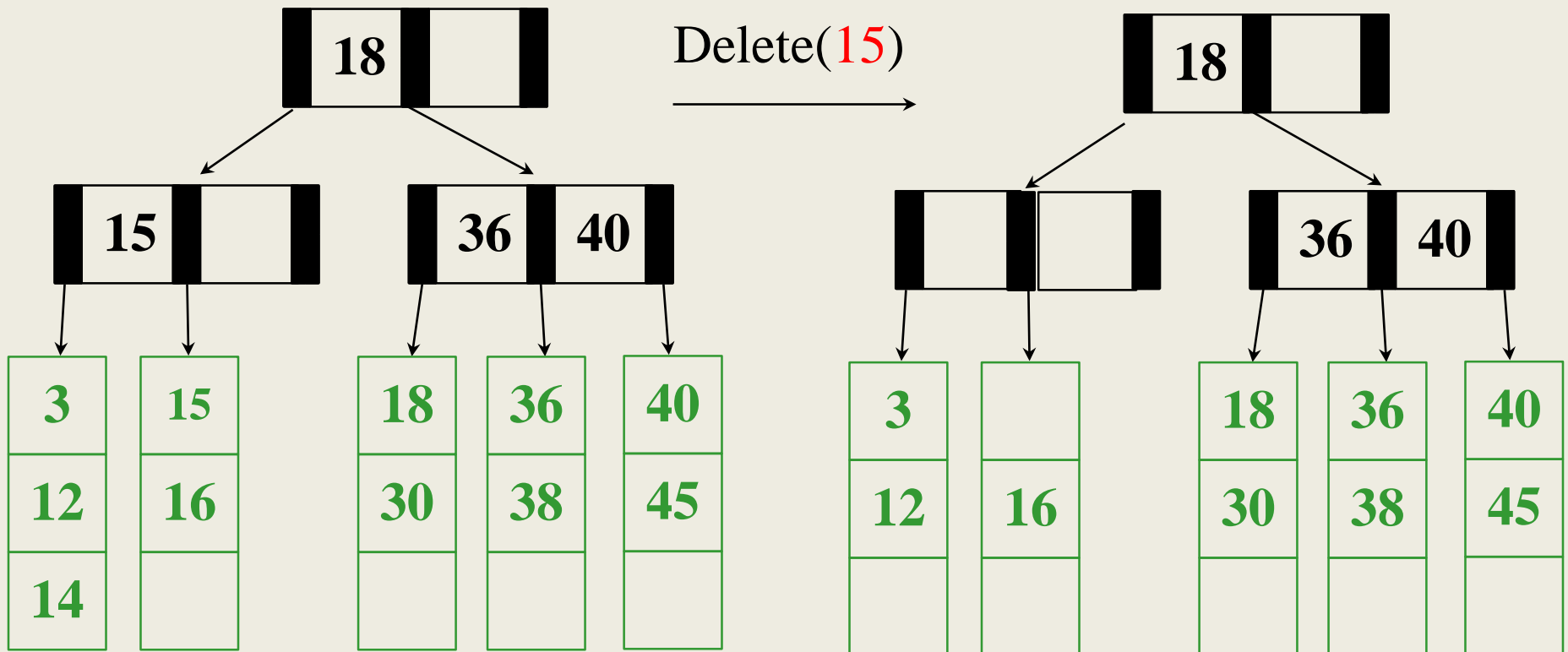
1. Insert the key in its leaf in sorted order
2. If the leaf ends up with  $L+1$  items, **overflow!**
  - Split the leaf into two nodes:
    - original with  $\lceil (L+1)/2 \rceil$  smaller keys
    - new one with  $\lfloor (L+1)/2 \rfloor$  larger keys
  - Add the new child to the parent
  - If the parent ends up with  $M+1$  children, **overflow!**
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.

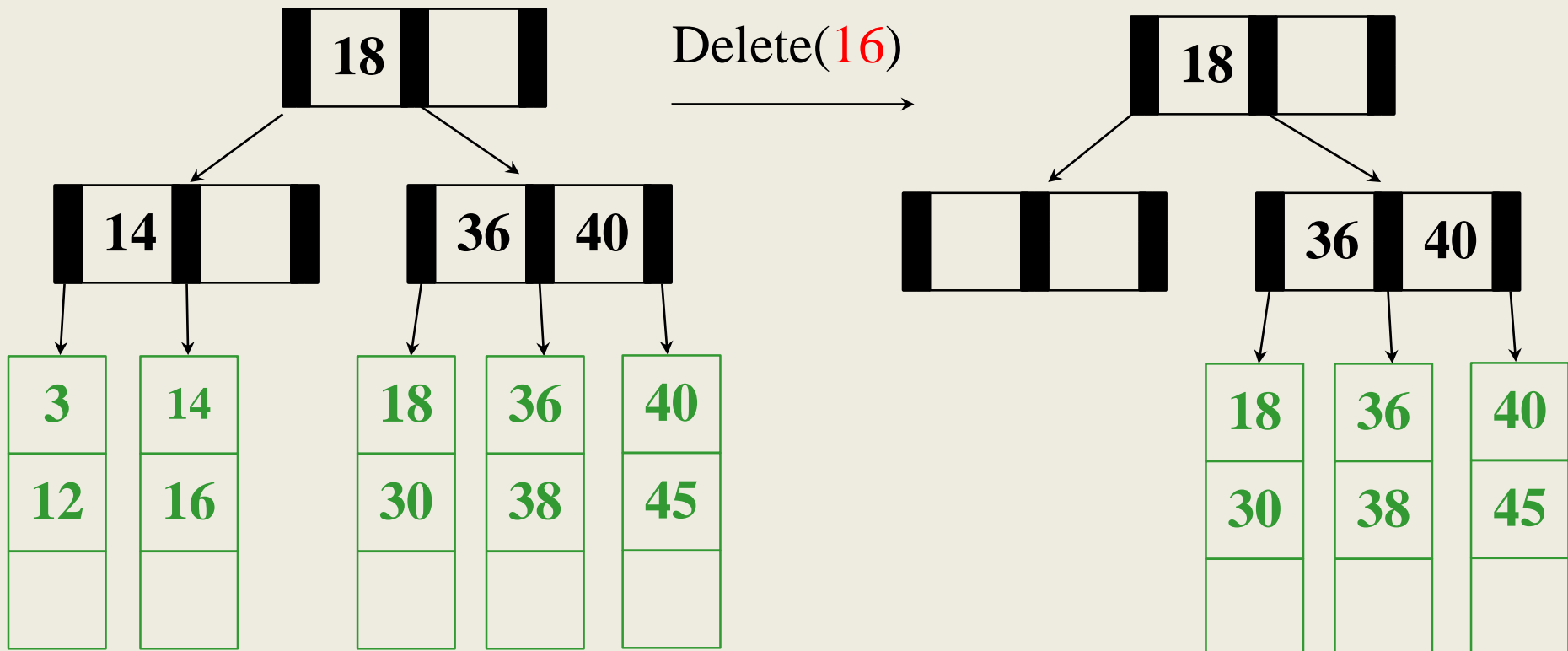
This makes the tree deeper!

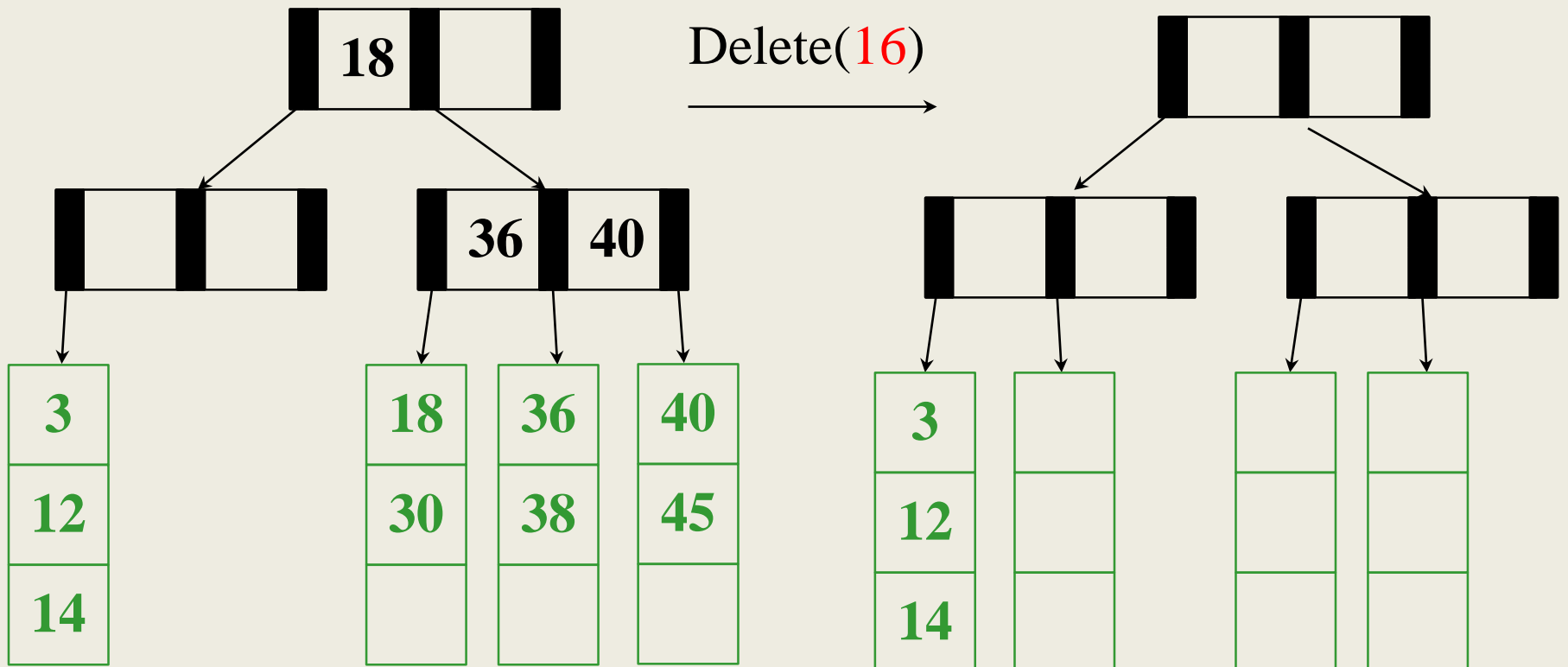


# And Now for Deletion...

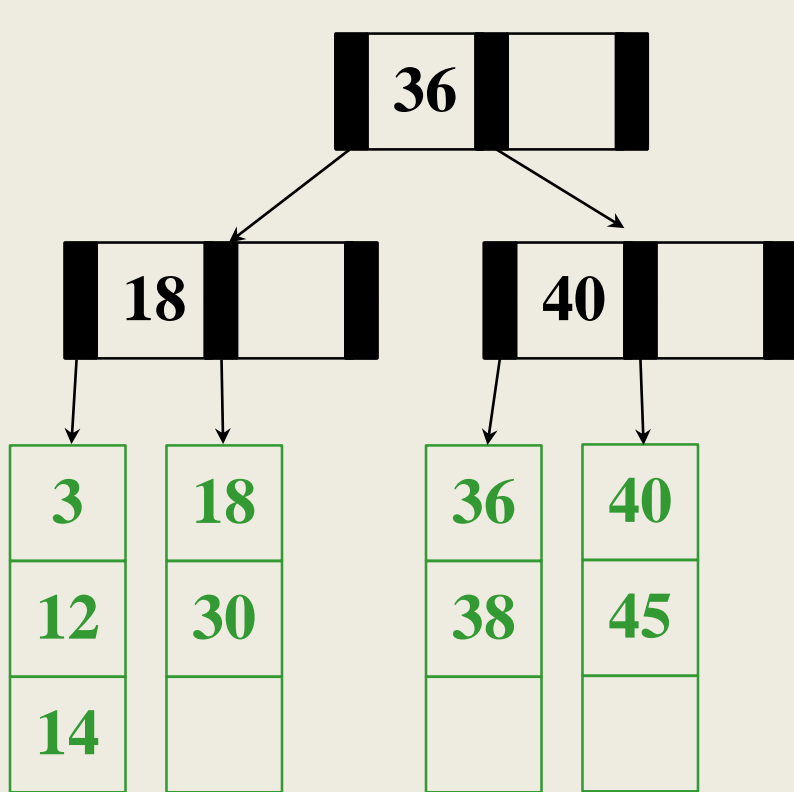




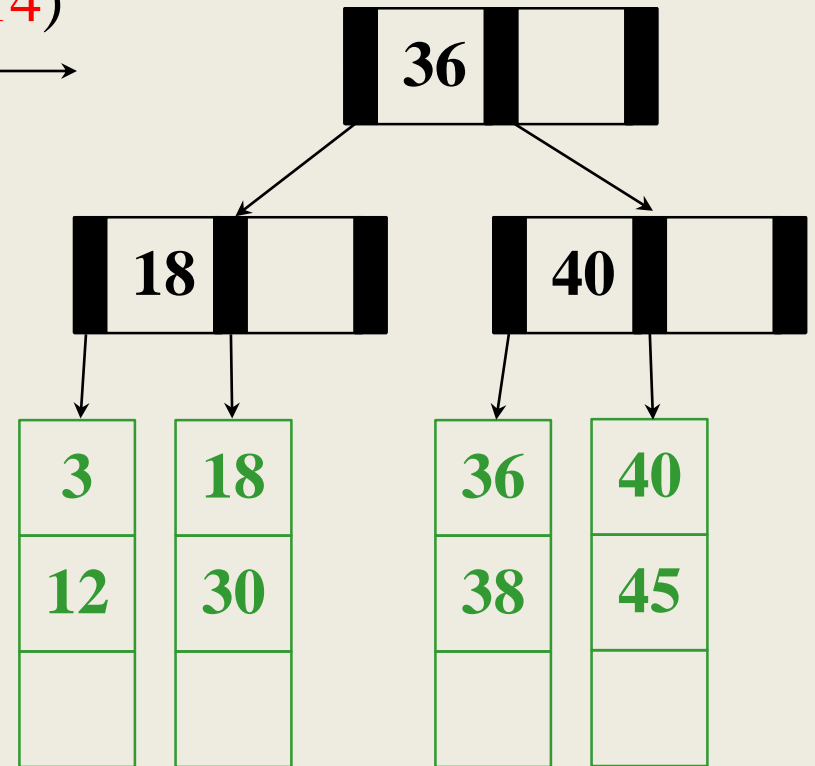


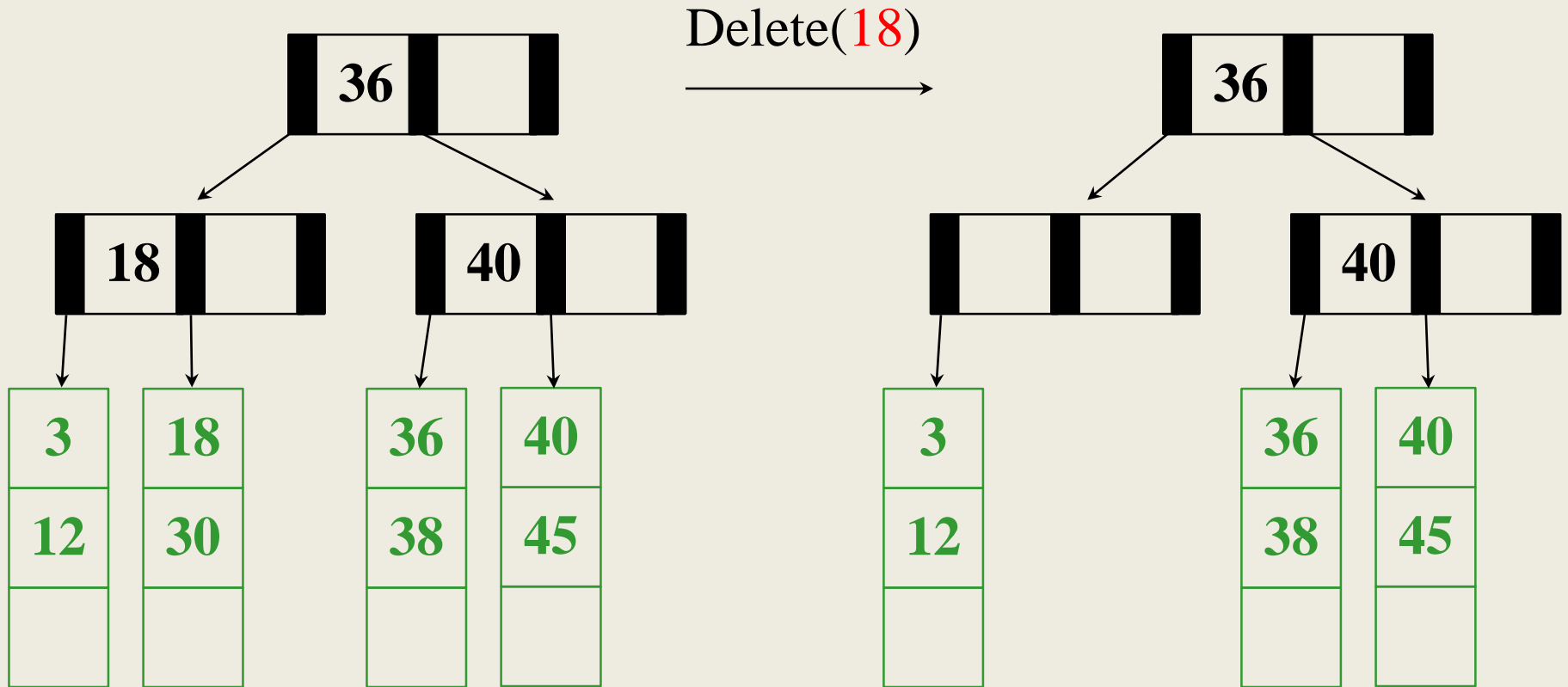


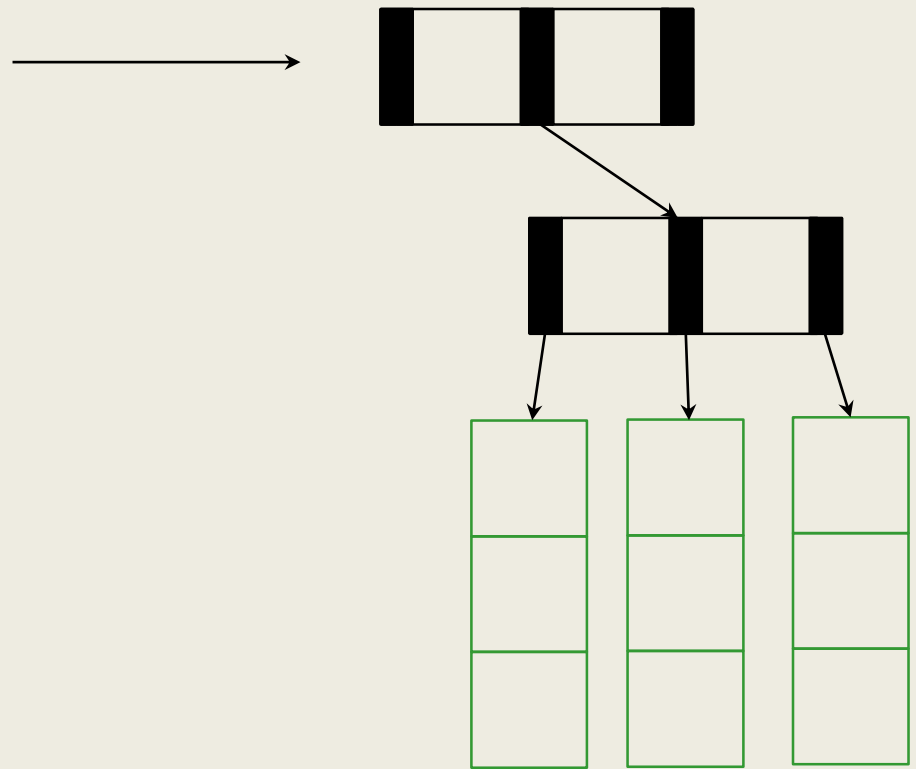
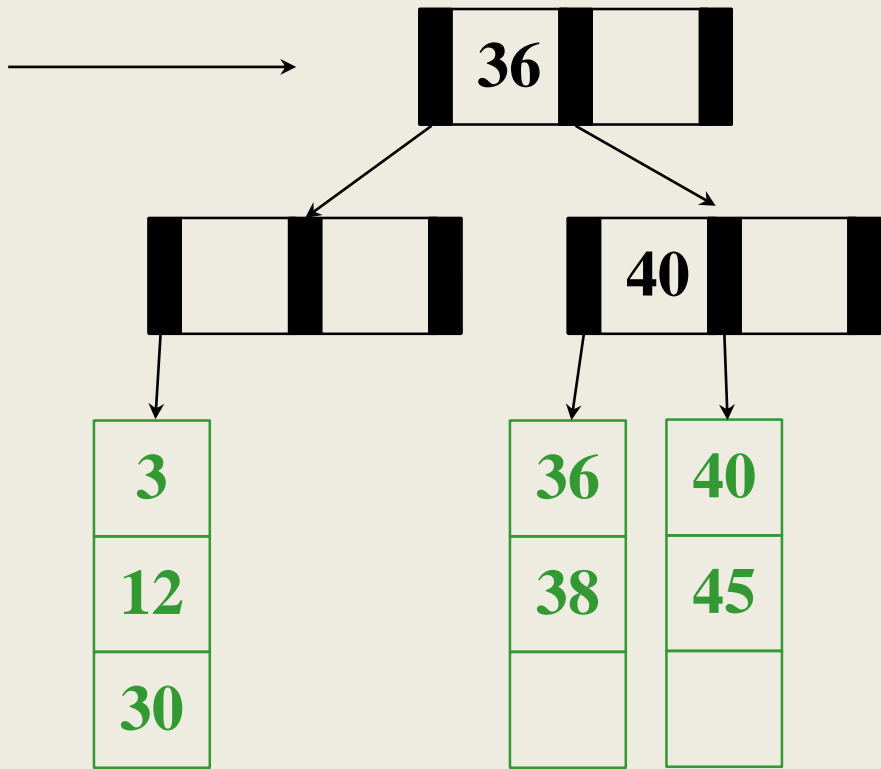
$M = 3, L = 3$   
4/18/2022



Delete(14)







$M = 3, L = 3$   
4/18/2022

# 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