CSE 332: Data Structures and Parallelism

Fall 2022

Richard Anderson

Lecture 9: 2-3 Trees and B-Trees

Announcements

AVL Trees

- Binary Search Tree with O(log n) height guarantee
- Structural Invariants
- Operations to maintain invariants on updates

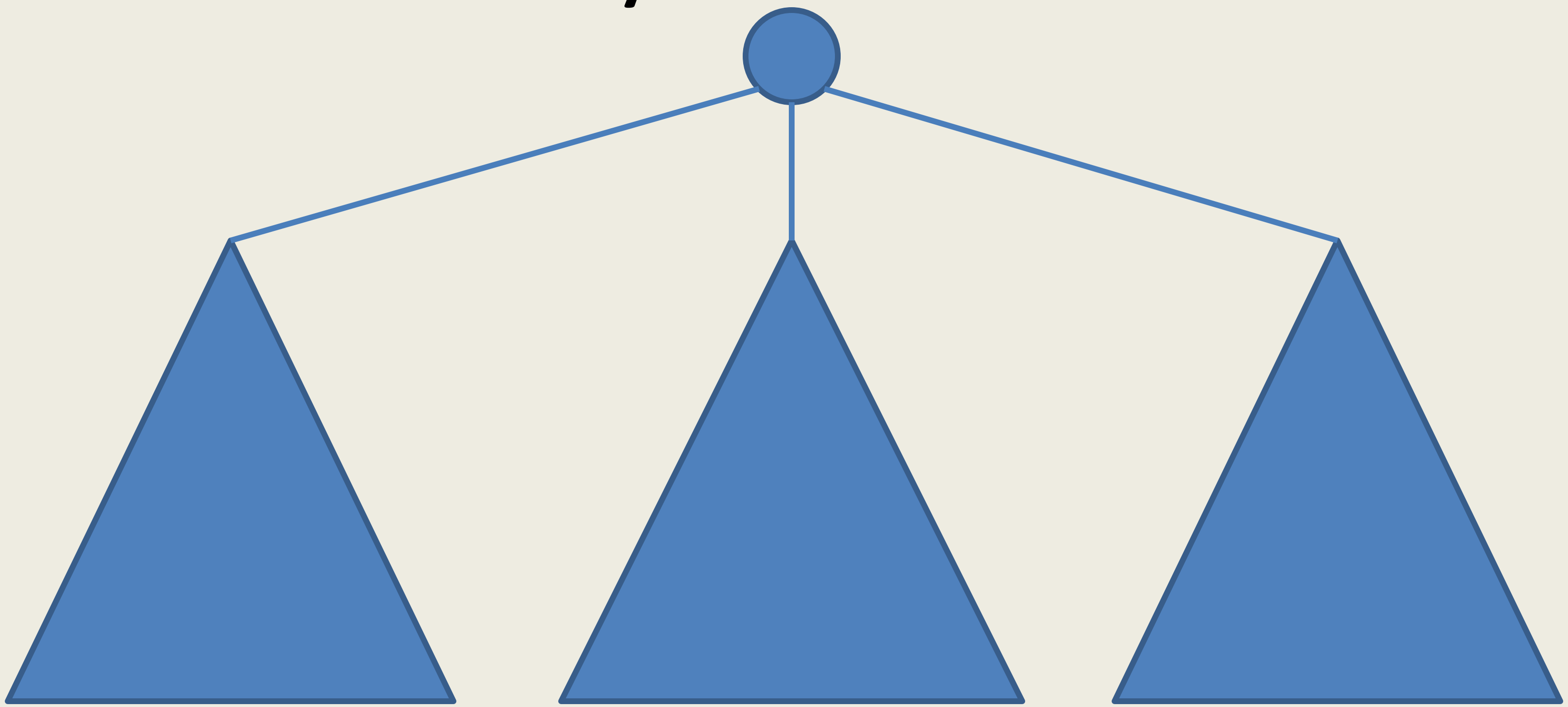
Lectures 9 & 10

- Computation Trees
- 2-3 trees as another O(log n) search tree
- Changing the rules of computation to model external storage
- B-trees: high degree generalization of 2-3 trees

One of the fundamental ideas of computing

- Problem division
- Reduce a problem to smaller and/or simpler problems
- Applies to both data and computation
- Often there is an exponential reduction
- Trees often capture this process
  - Branching factor
  - Workload associated with nodes

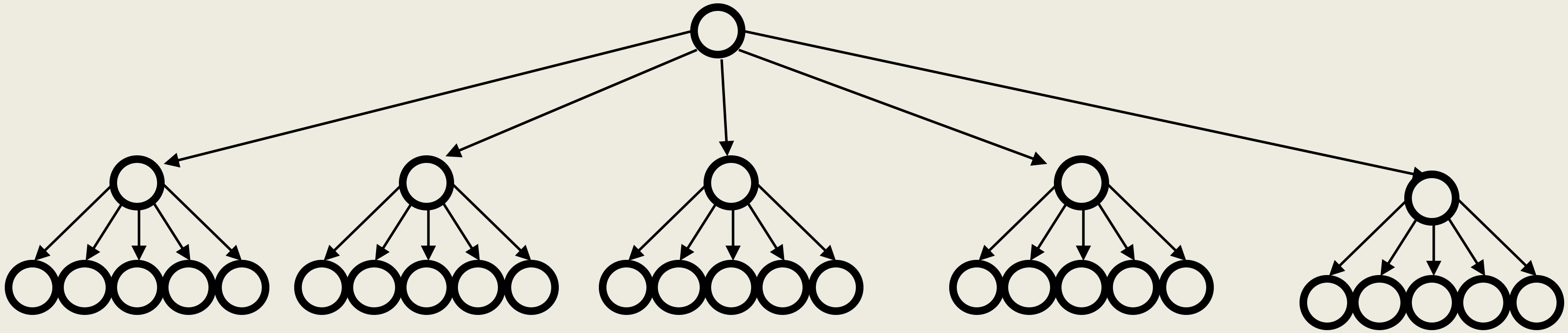
Trinary search trees



- How are BST invariants modified
- How are BST operations modified

M-ary Search Tree

Consider a search tree with branching factor *M*:



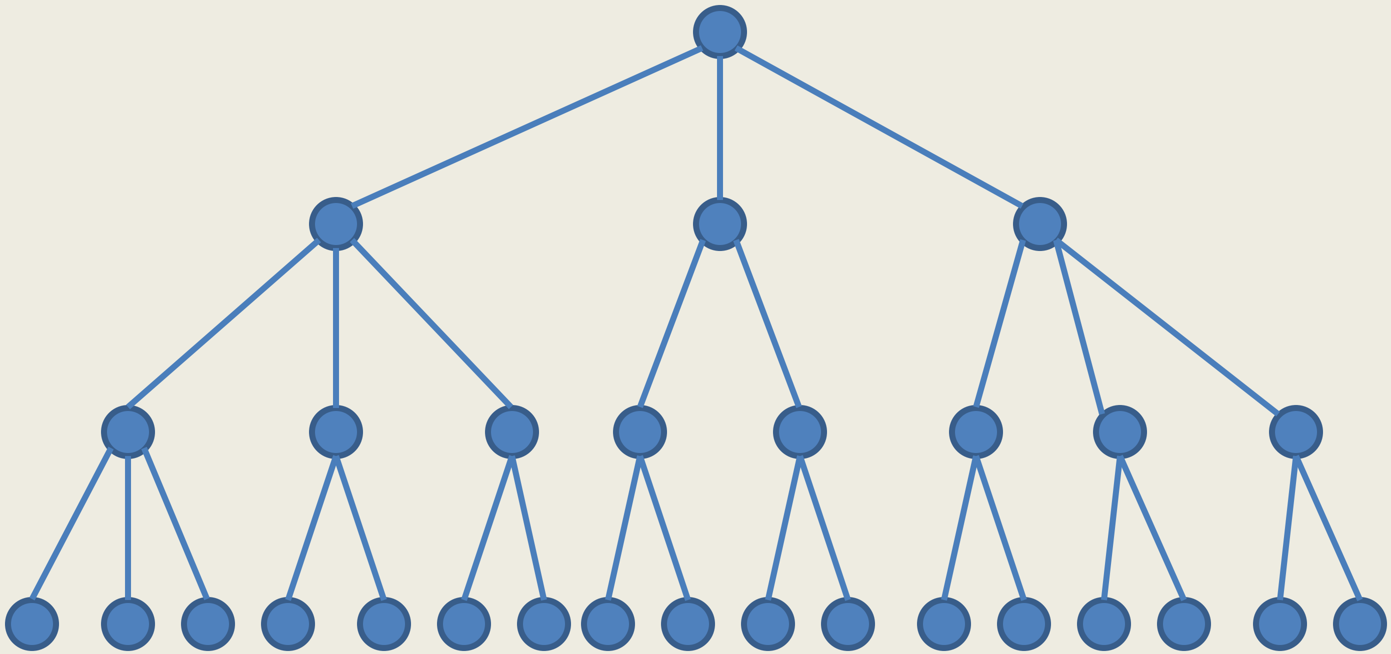
- Complete tree has height:

- # hops for *find*:

- Runtime of *find*:

2-3 Trees

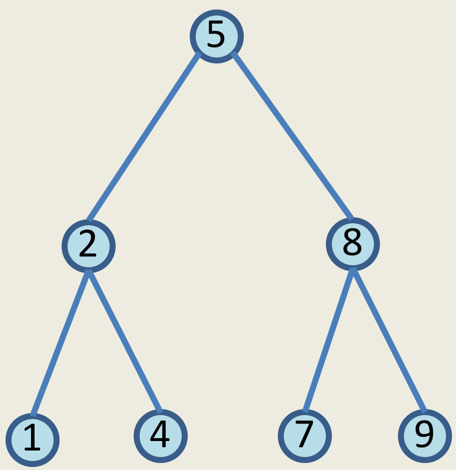
- Can balance a tree by varying the depth of the leaves, or by varying the number of children of the nodes
- 2-3 trees have all internal nodes of degree 2 or 3

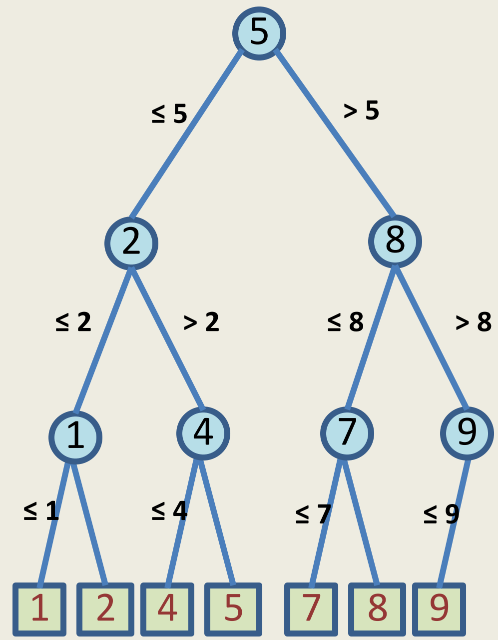


2-3 Tree basics

- Search trees
- Invariants
  - Every internal node has degree 2 or 3
  - All leaves at the same depth
- Height bound
- B-trees, generalization to high degree trees

Detail: Keys vs. Values stored at nodes



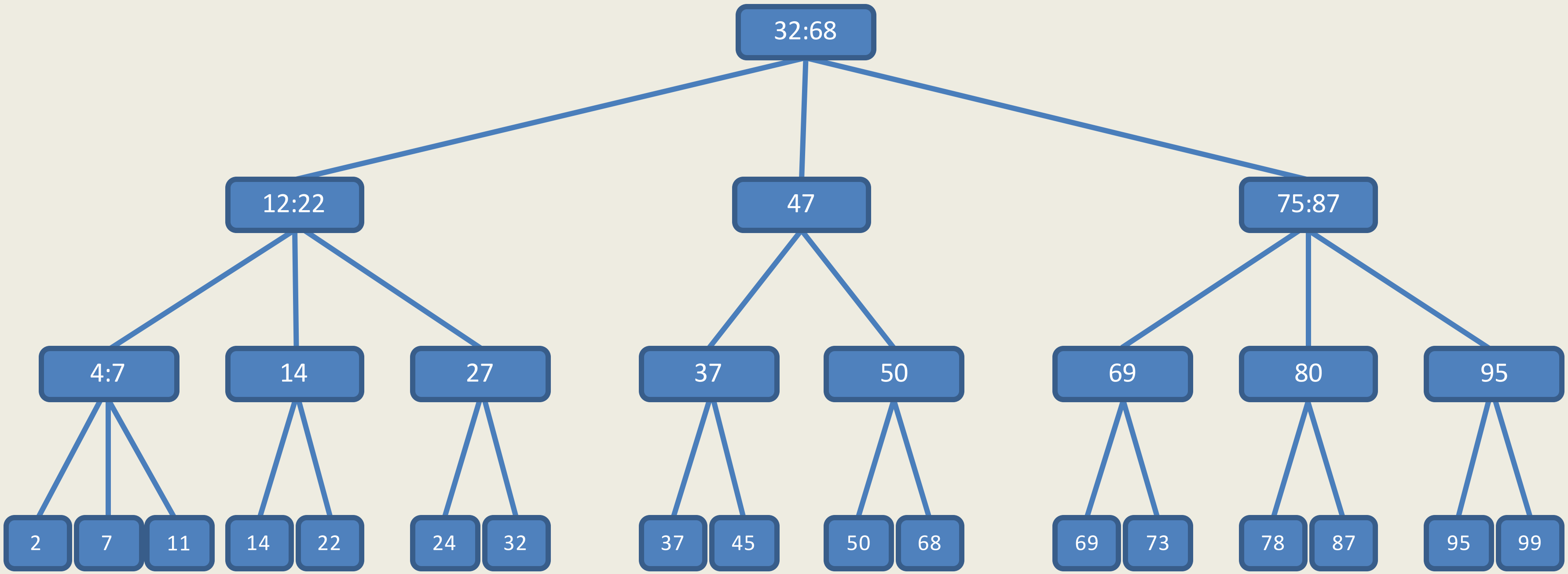


For 2-3 trees, we will consider the version with values stored in leaves

Each internal node can have 1 or 2 keys, each leaf has one value

Assume distinct keys

2-3 Tree Example



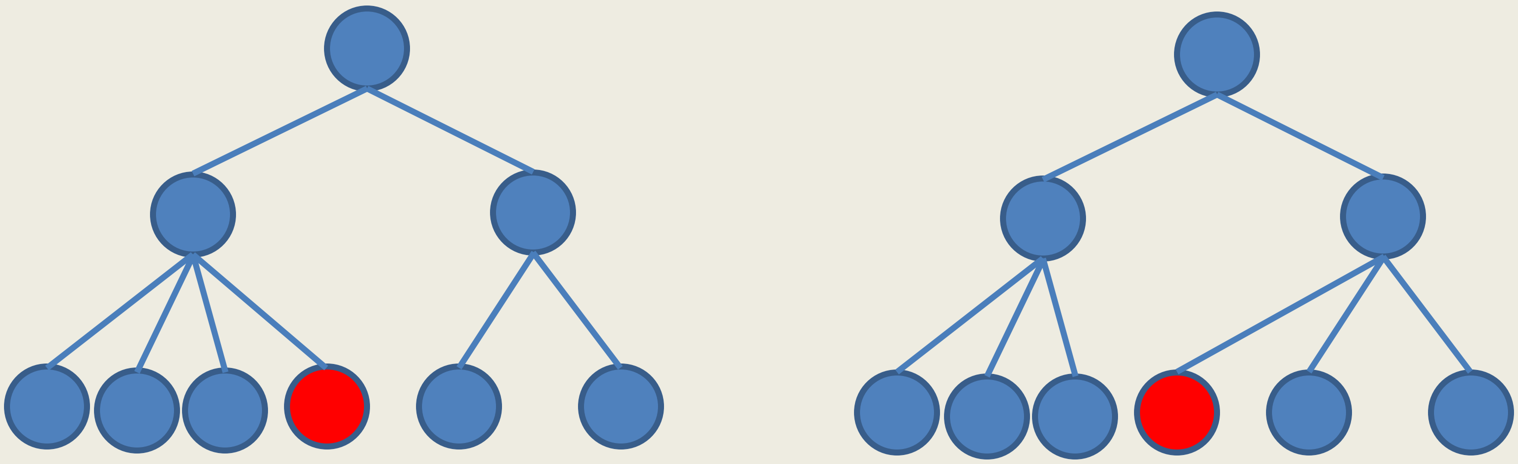
Inserts

- Need to maintain invariants
  - Internal nodes of degree 2 or 3
  - All leaves at the same level
- Trees of height 0:
- Trees of height 1:

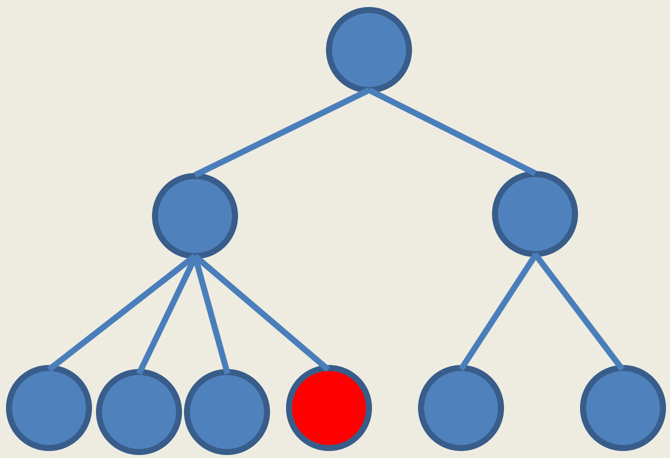
General case

- Insert happens at a leaf
- Easy case, parent has two children

- Three child case, option 1, rebalance children



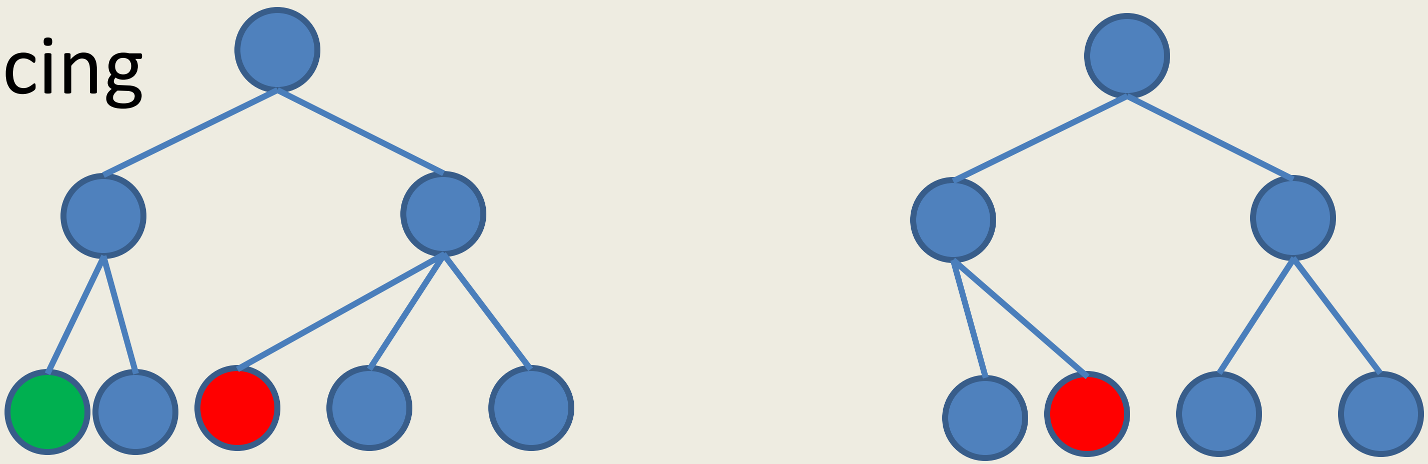
Option 2, parent splitting



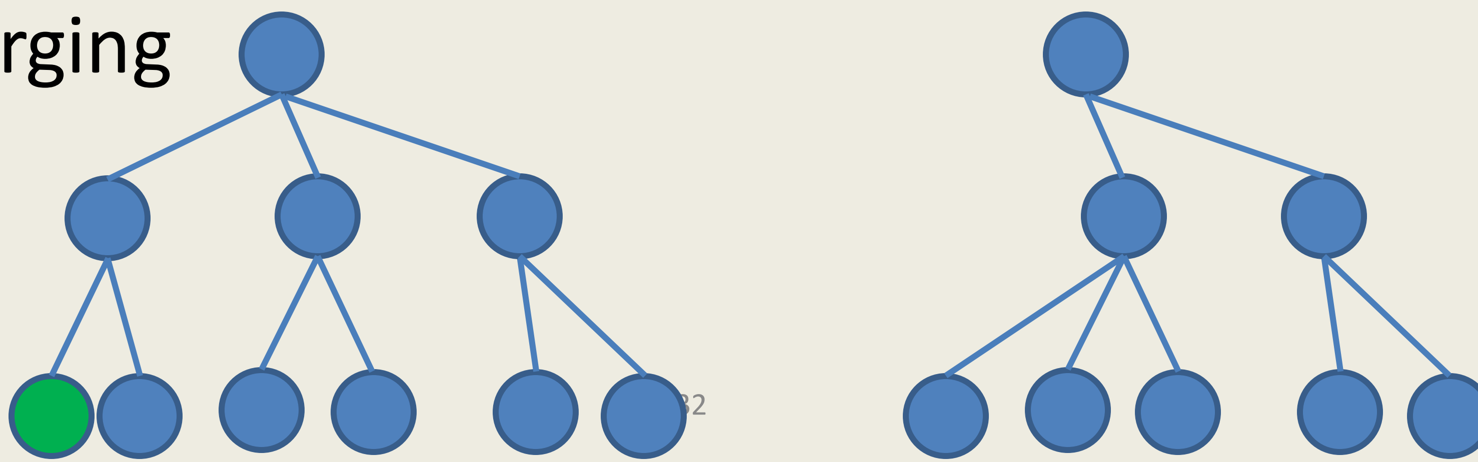
- But what if the grand parent already has three children?

Deletes (not being lazy)

- Easy case is a parent with three children
- Rebalancing



- Merging



Thinking about computation

- Algorithmic view
  - Computation is a sequence of primitive operations
  - Abstract machine
  - Various approaches
    - Runtime as a function of input size
    - Asymptotic view
  - This approach has been very successful
    - Basic understanding for implementation of algorithms
    - Foundation for mathematical theory of computation

Where does this model break?

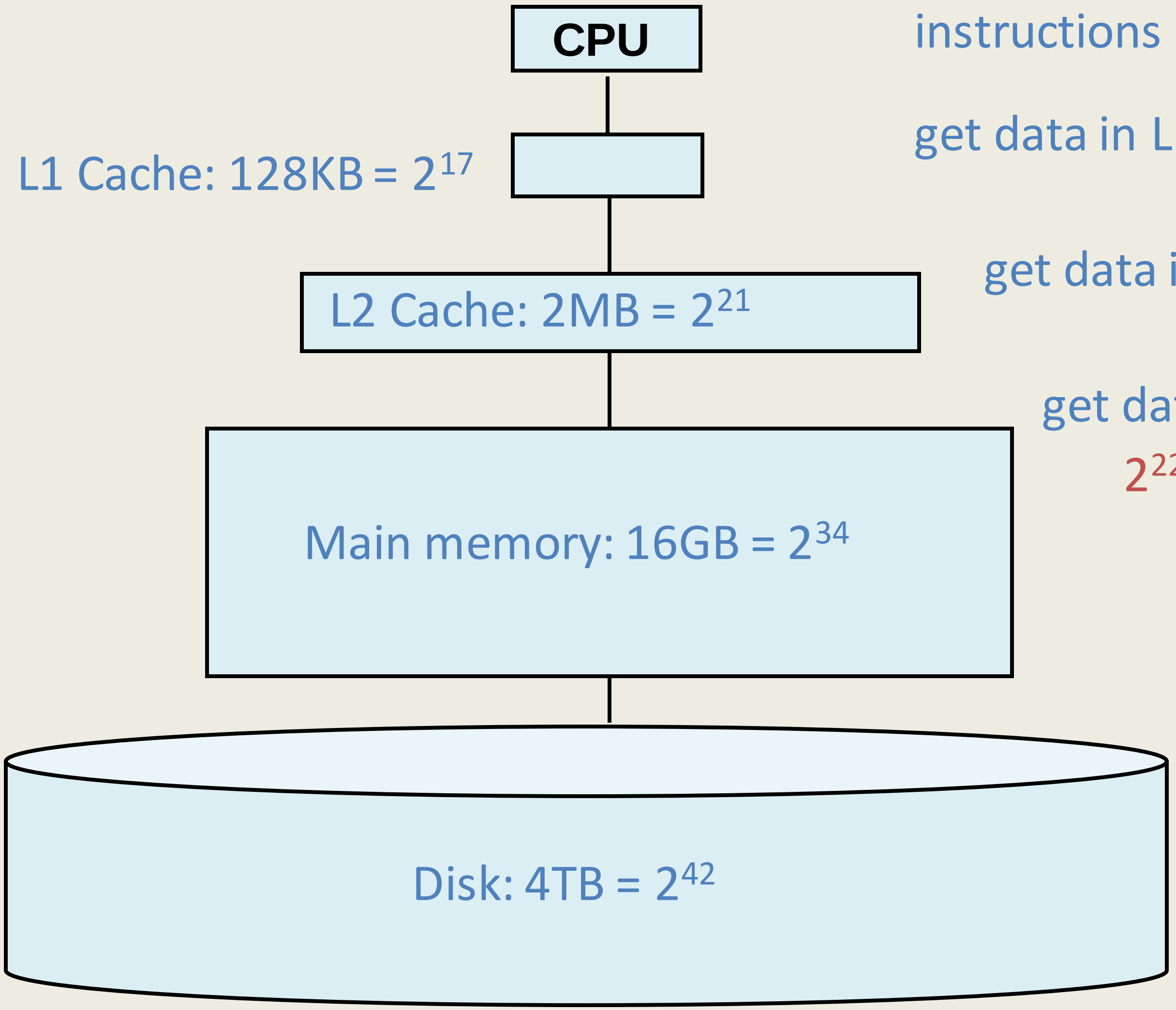
- Model: sequence of operations of roughly equal cost
- Model breaks if it does not suggest appropriate implementation techniques
- When is "roughly equal cost" wrong?

Computer Architecture

- CPU – collection of highly engineered computational gadgets
- Dominant consideration – keeping the CPU fed with data to keep all operations running
- Memory access costs
  - The closer data is to the CPU the faster it is to access
  - Different technologies in hierarchy change costs

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 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

*These numbers are out of date

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
- 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…

Model of data access

- Two separate issues
  - What is the latency
  - How much data is delivered at a time
- Buying in bulk
- Natural size of data delivery (page)
- External storage boundary most important to consider

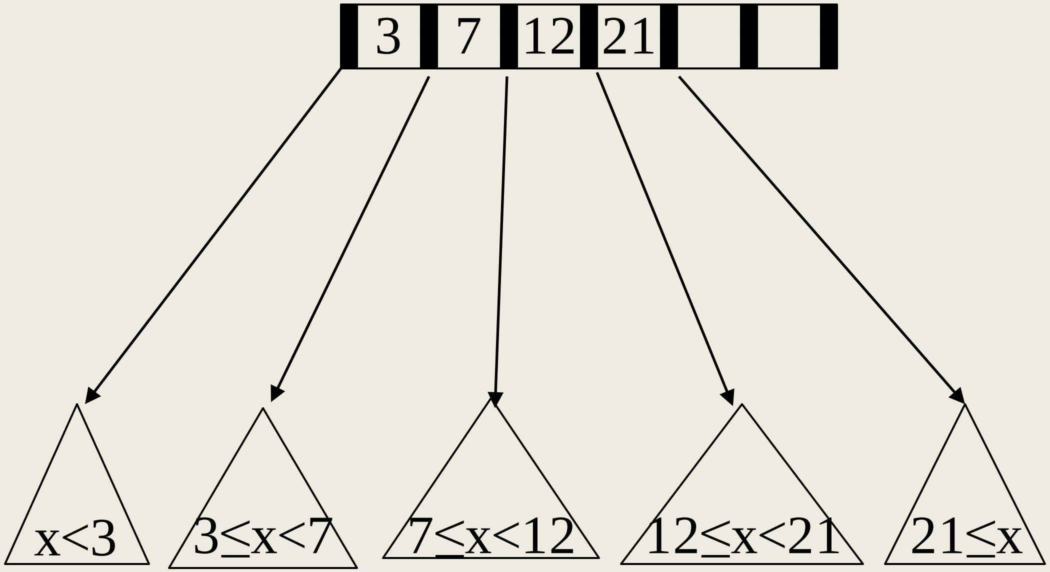
BSTs?

- 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

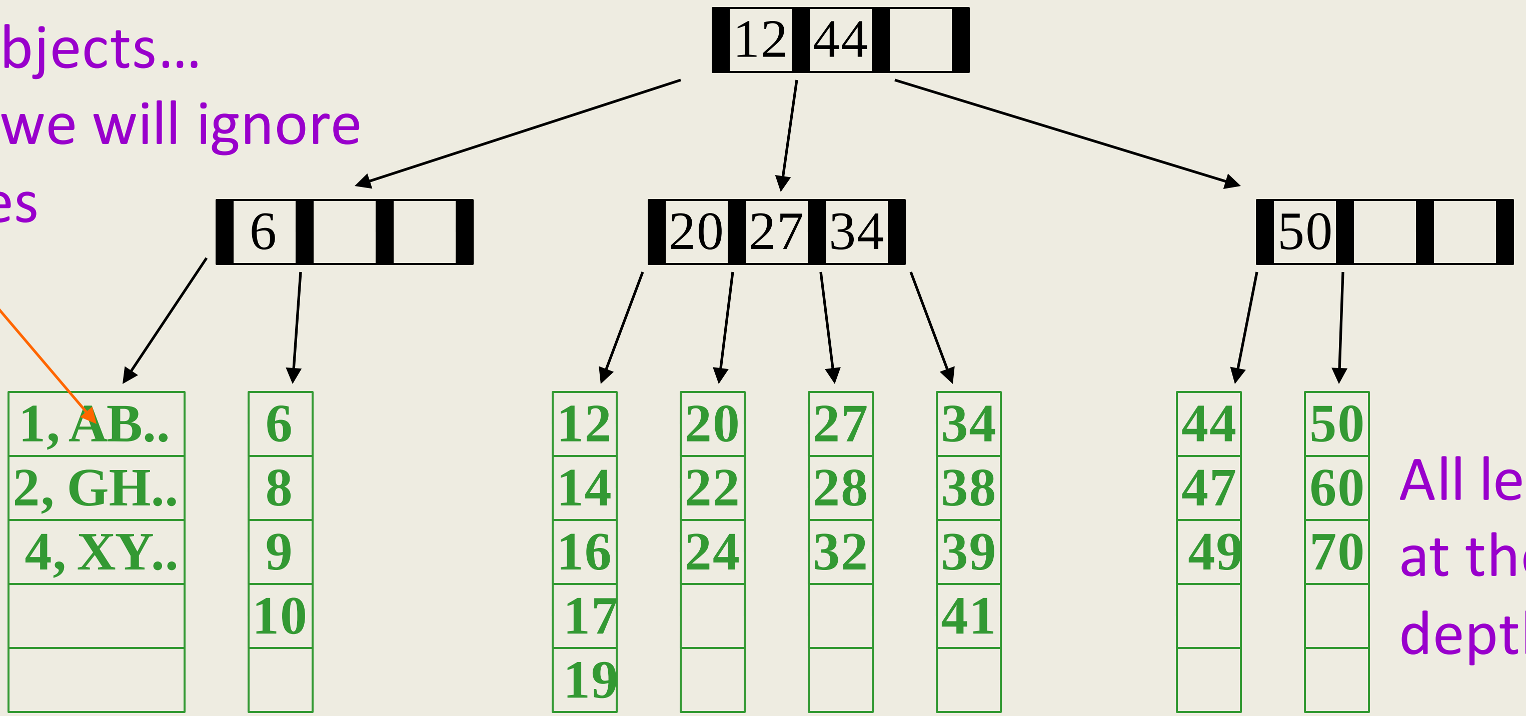
Root (special case)

- 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

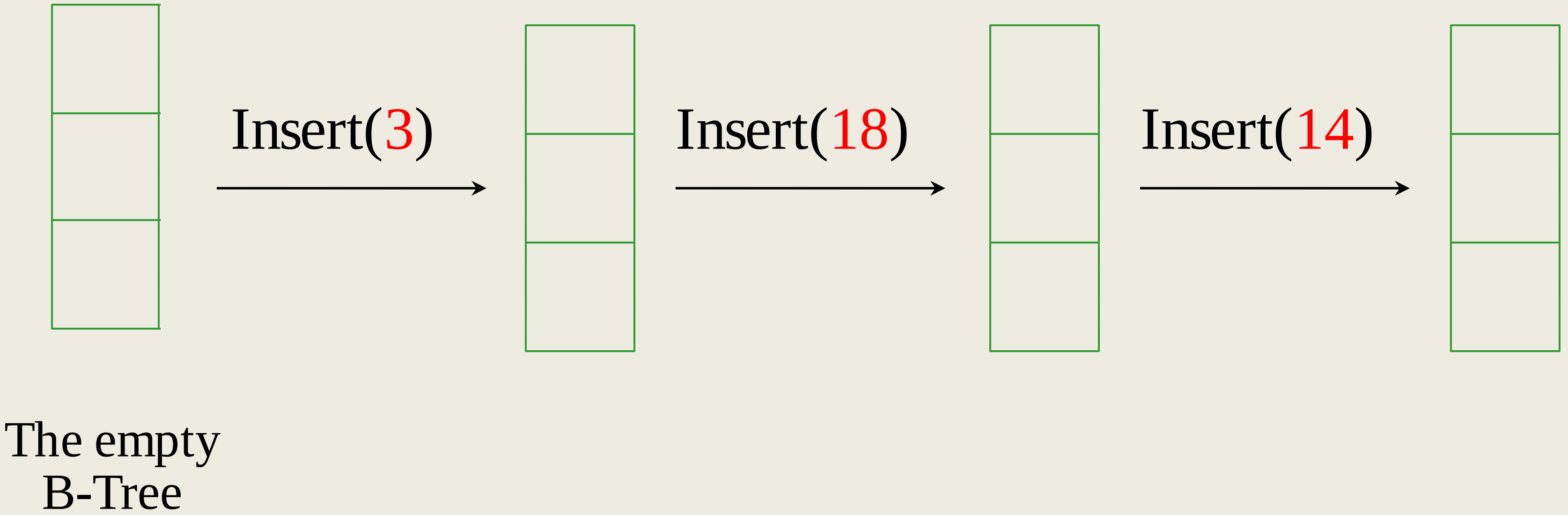
- Suppose again we have $n = 2^{30} \approx 10^9$ items:

- Depth of AVL Tree

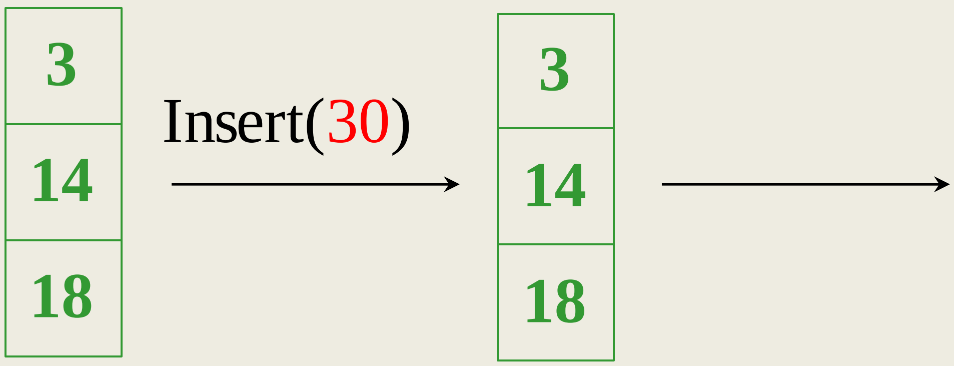
- Depth of B+ Tree with M = 256, L = 256

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

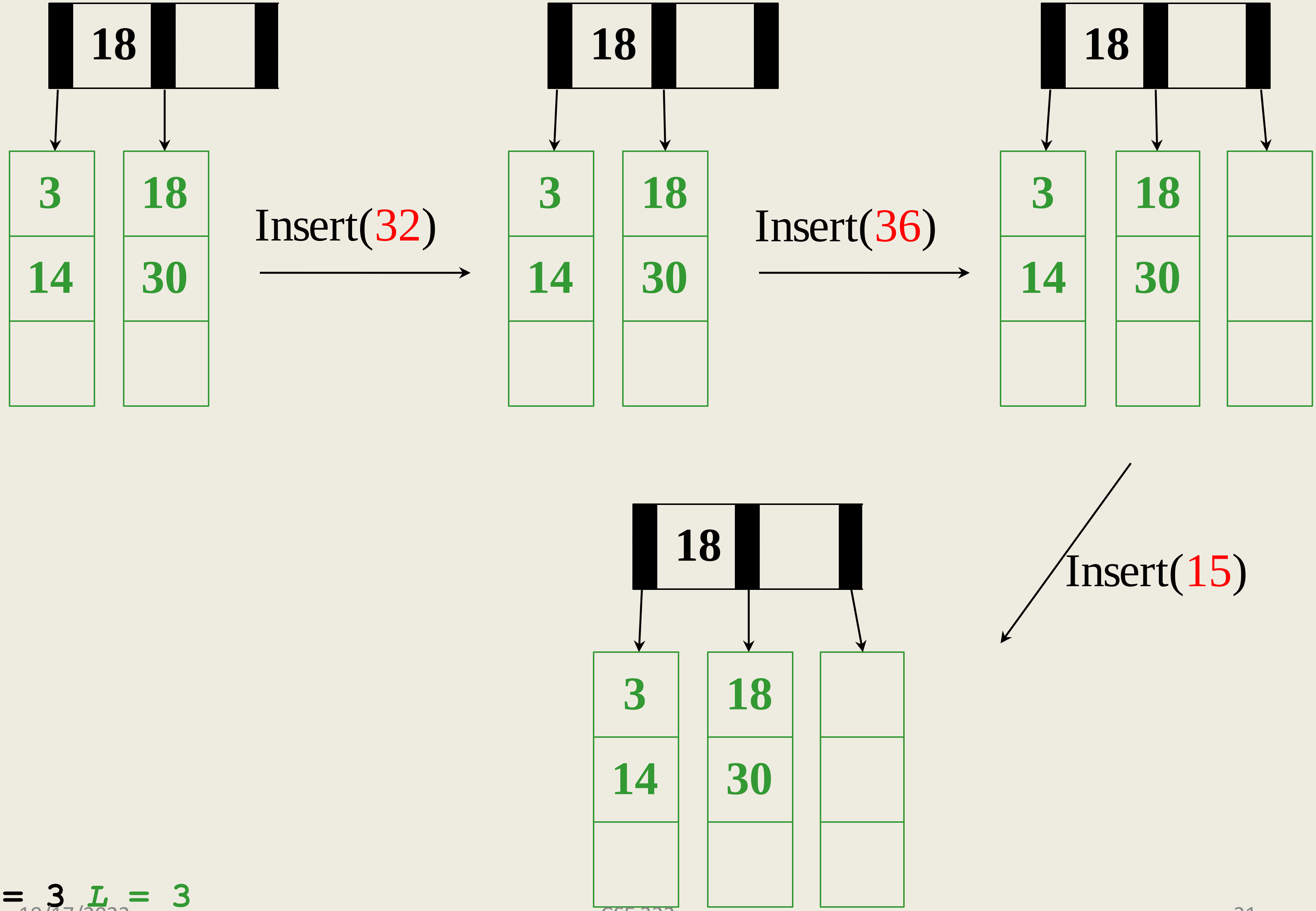
Building a B+ Tree with Insertions



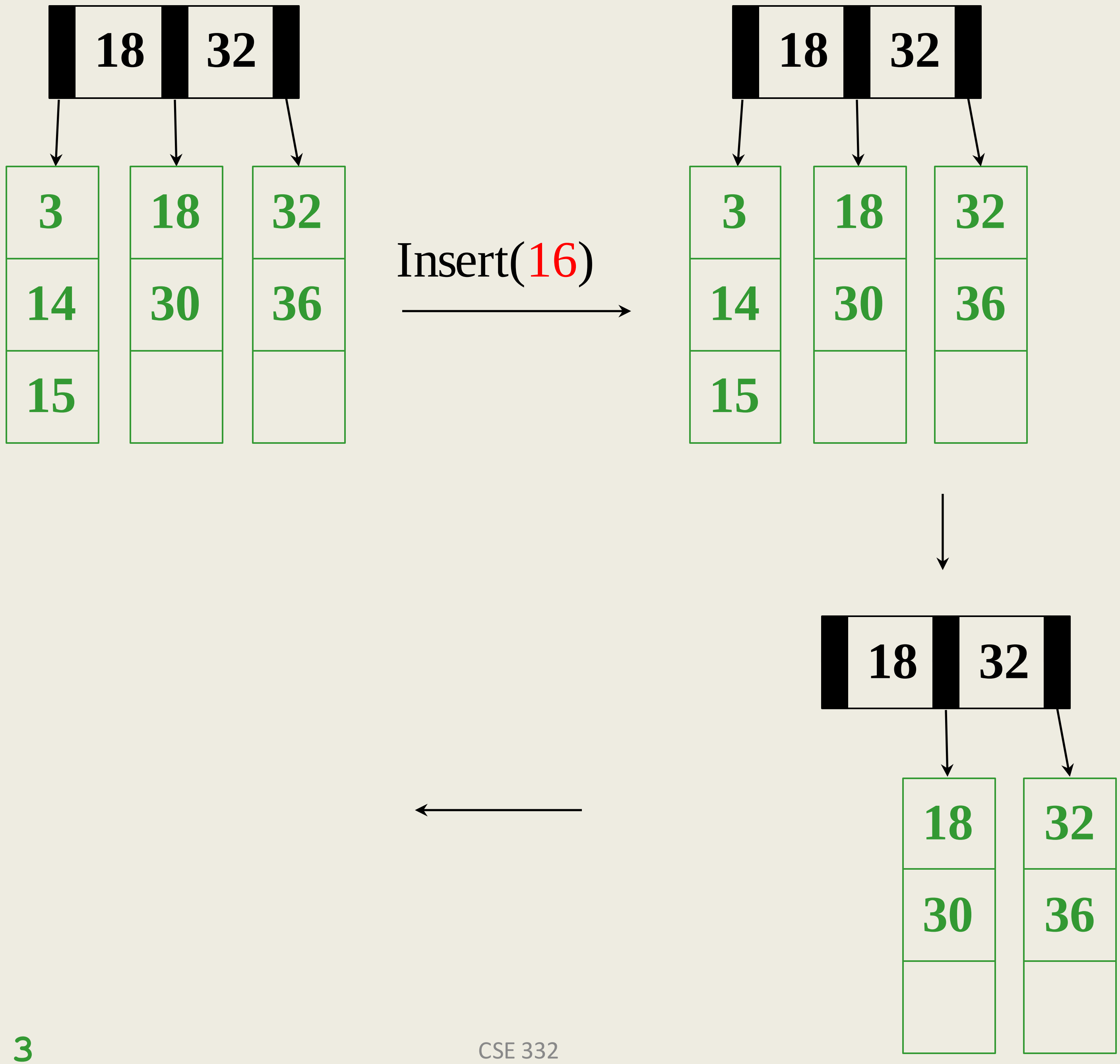
M = 3 **L = 3**



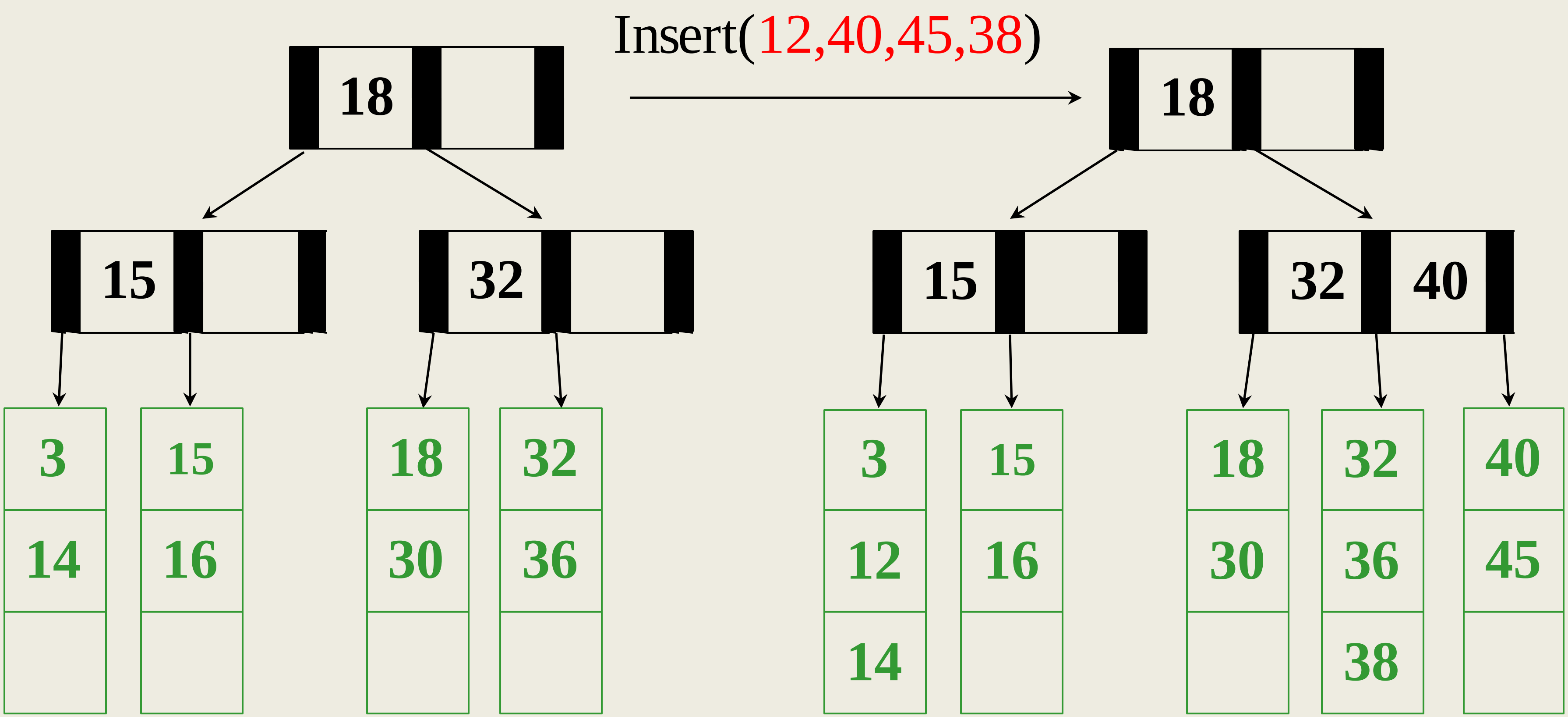
M = 3 *L* = 3


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

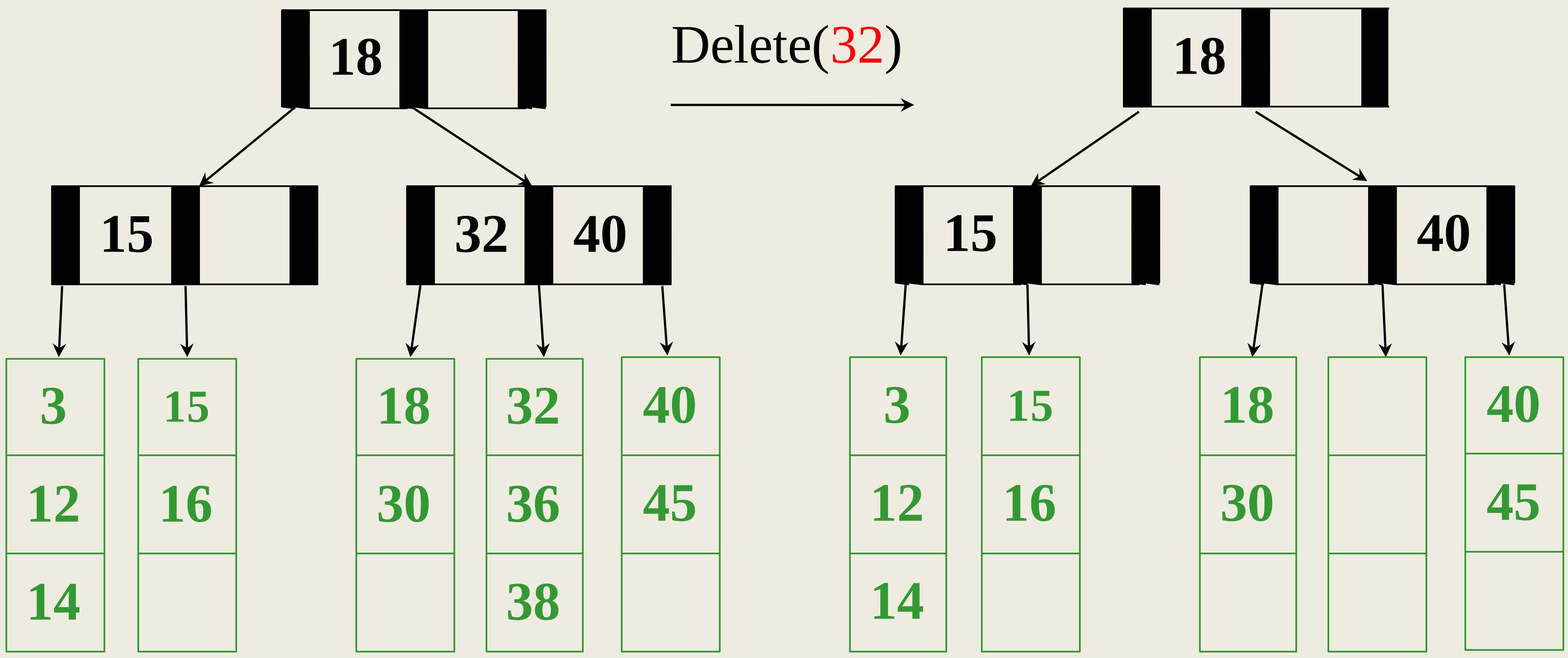
Insertion Algorithm

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**!

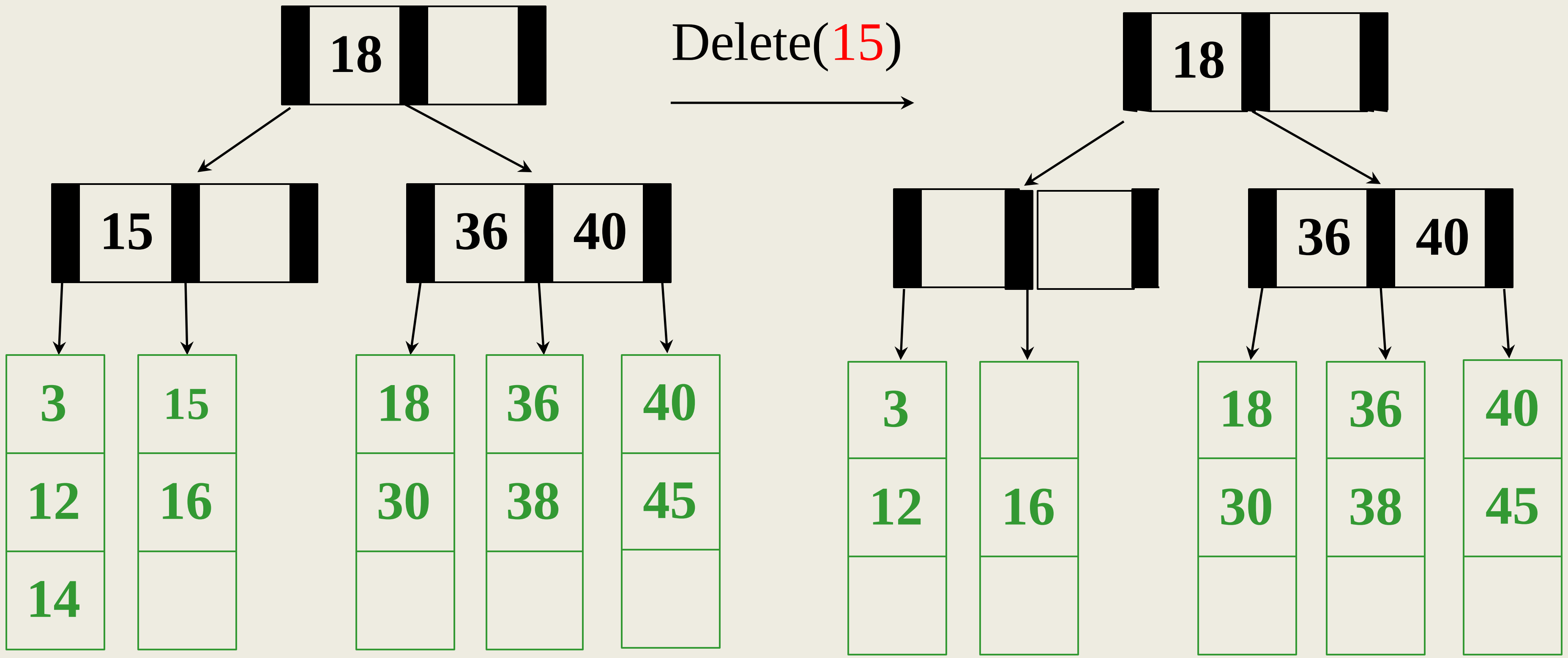
This makes the tree deeper!

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.

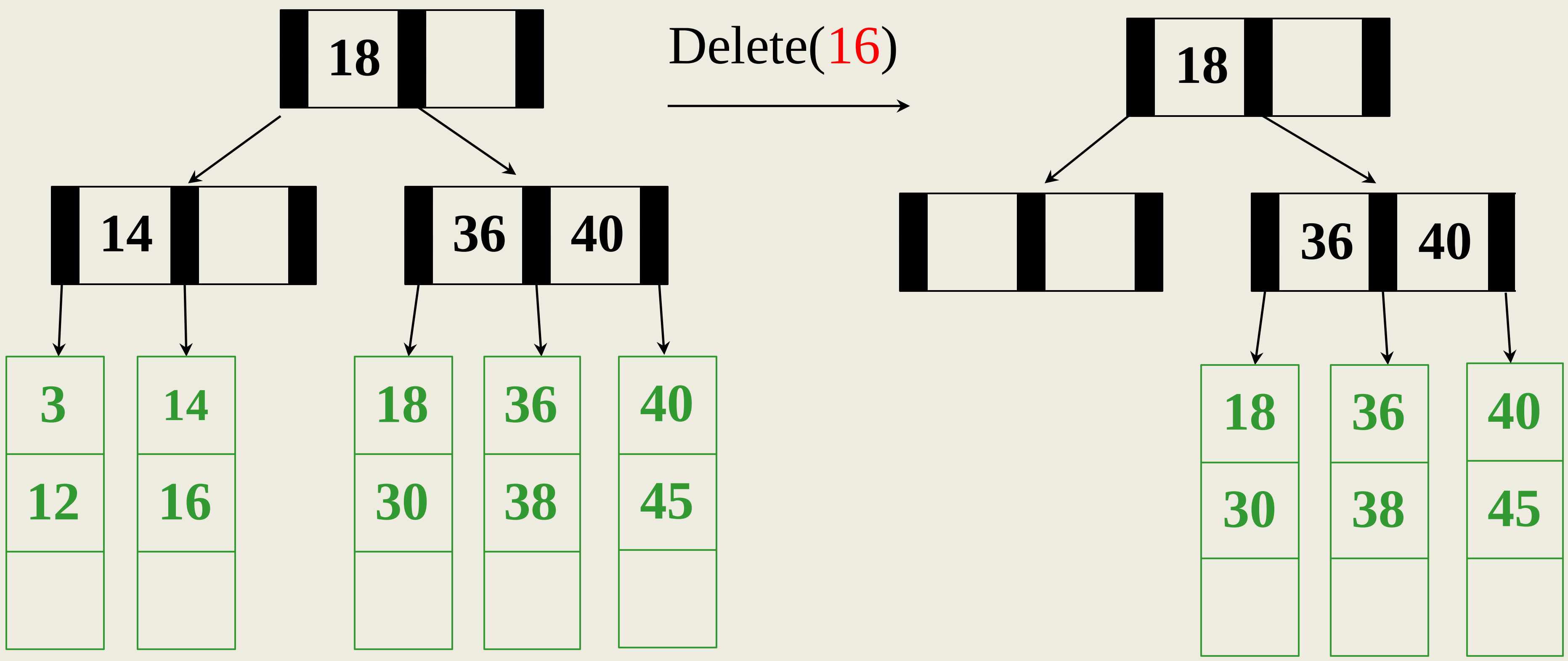
And Now for Deletion…



M = 3 *L* = 3

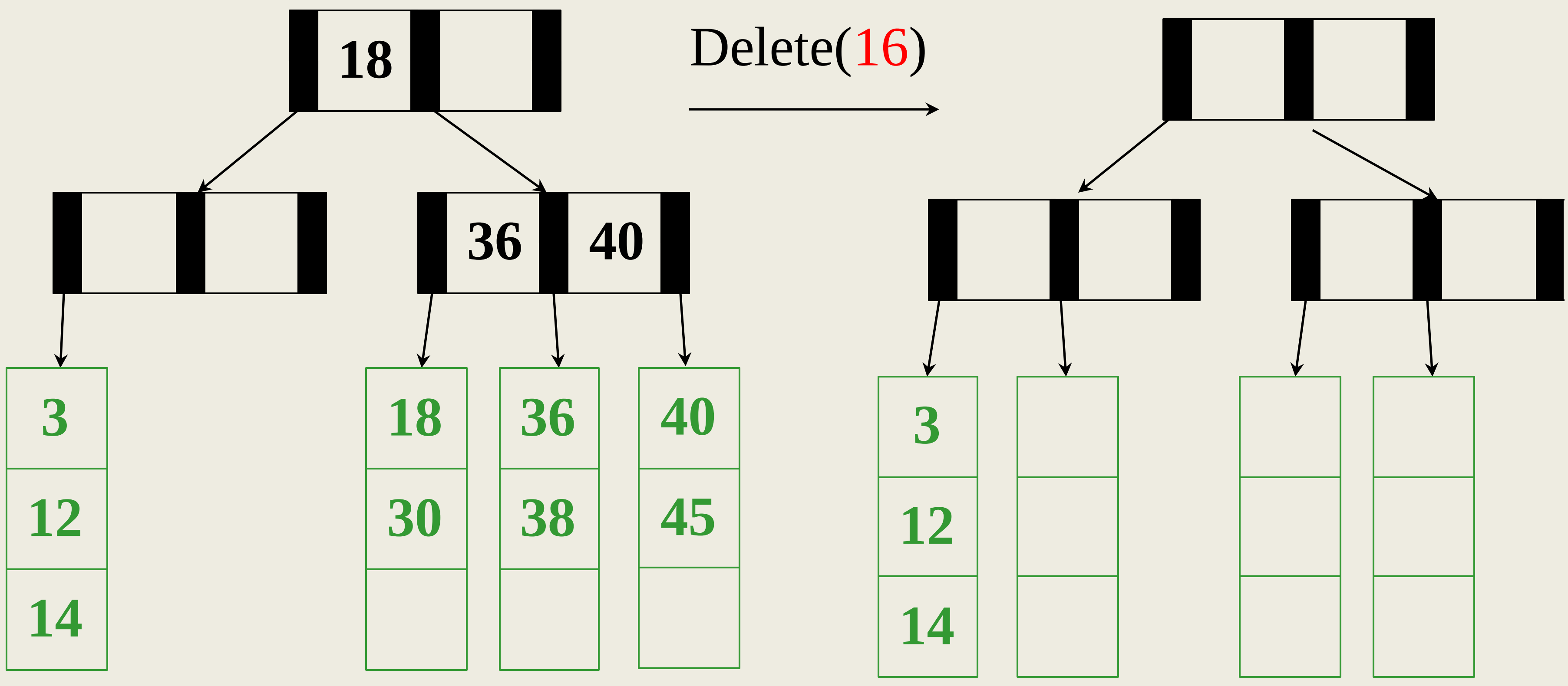

18
15
36
40
3
12
14
15
16
18
30
36
38
40
45
Delete(15)
18
36
40
3
12
16
18
30
36
38
40
45

M = 3 *L* = 3

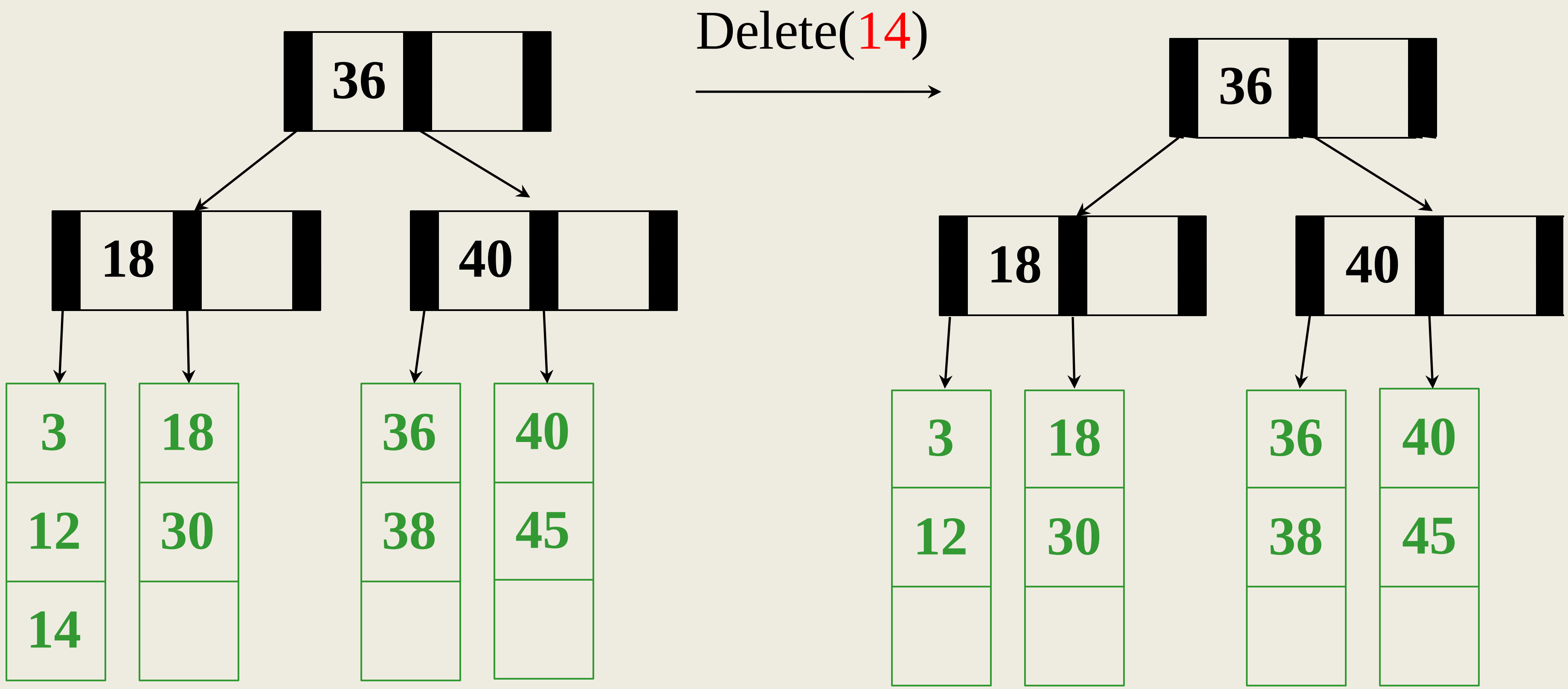


18
14
36
40
3
12
14
16
18
30
36
38
40
45
Delete(16)
18
36
40
18
30
36
38
40
45

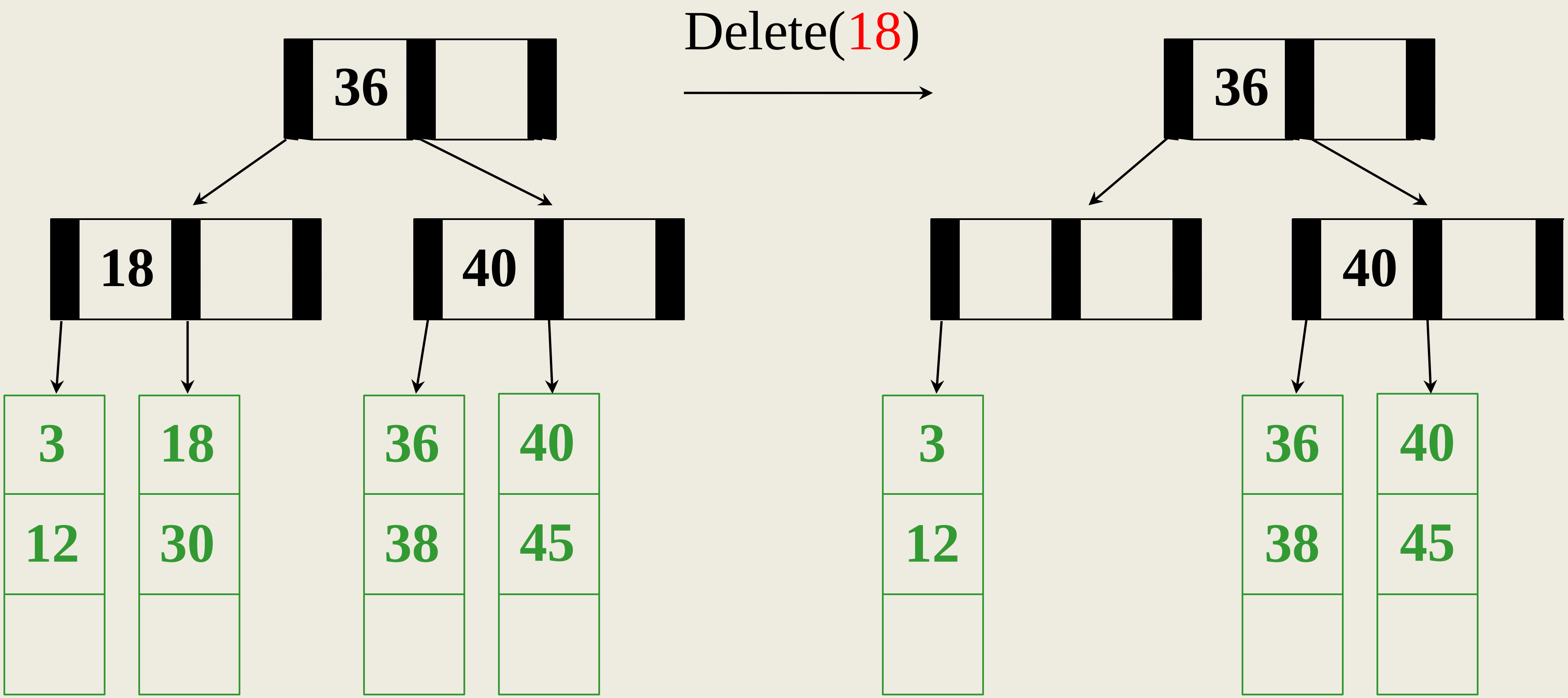
$M = 3$ $L = 3$



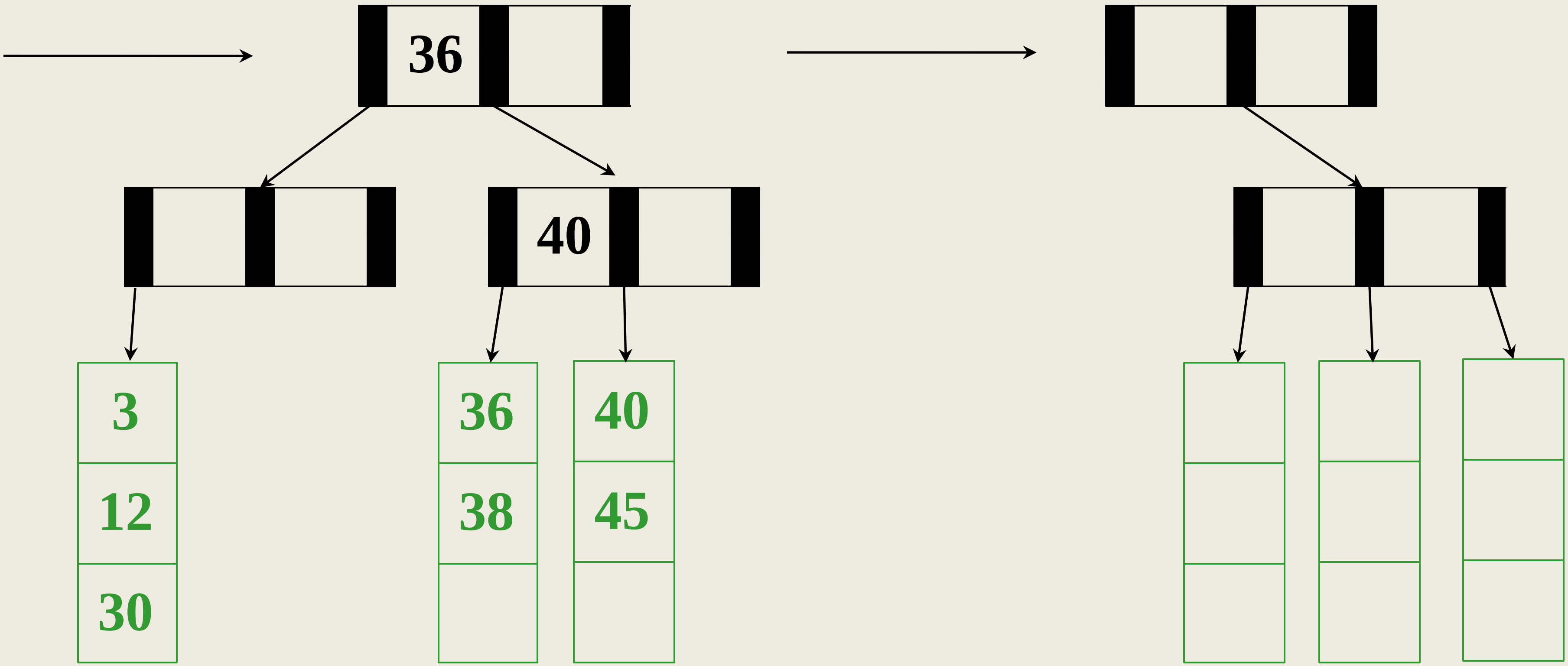
M = 3 L = 3


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