CSE 332
Data Structures & Parallelism

B Trees 2

Melissa Winstanley
Spring 2024

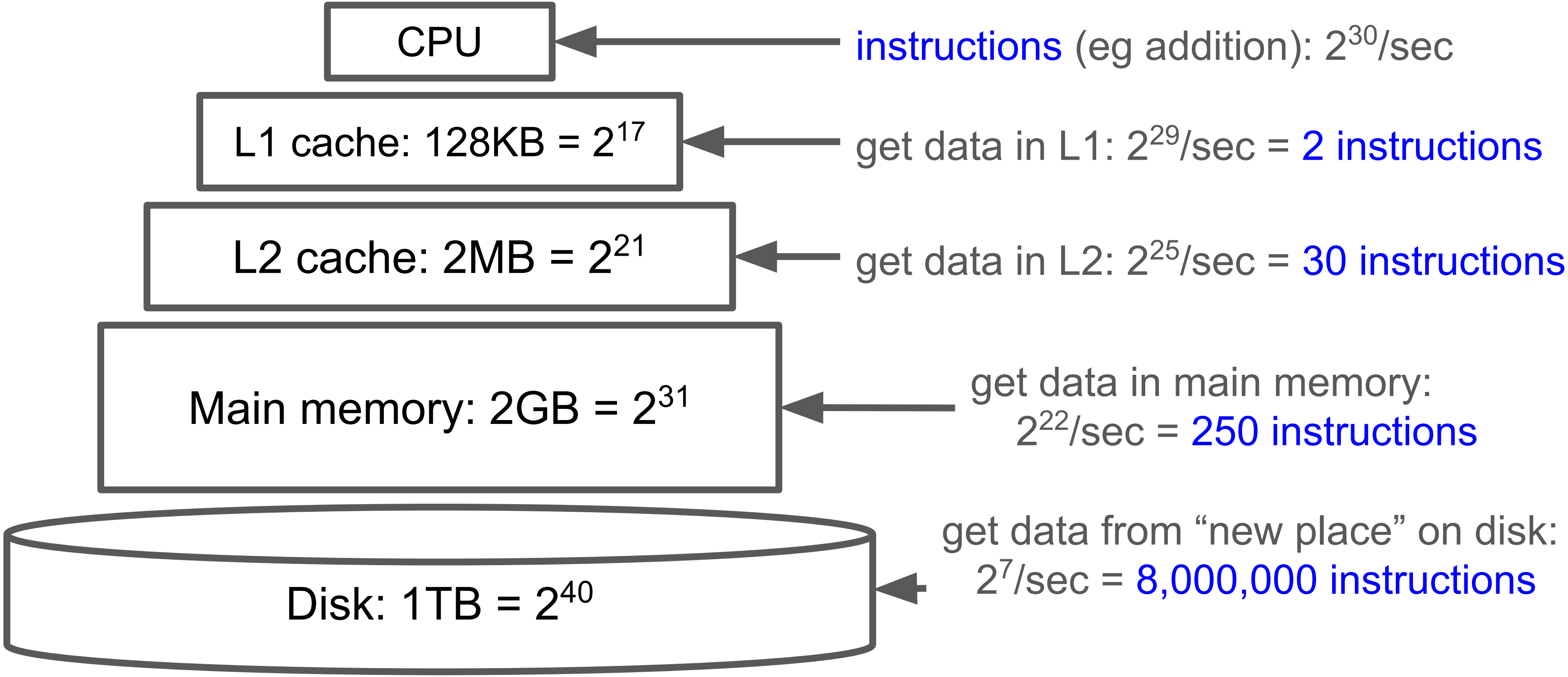
Question Time

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

Is that really true?

A Typical Memory Hierarchy

"Every desktop/laptop/server is different" but here's a plausible configuration



“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 nearby memory because:
        - It's easy
        - And likely to be asked for soon

Spatial locality

- Side note: Once a value is in cache, may as well keep it around for awhile; accessed once, a particular value 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.

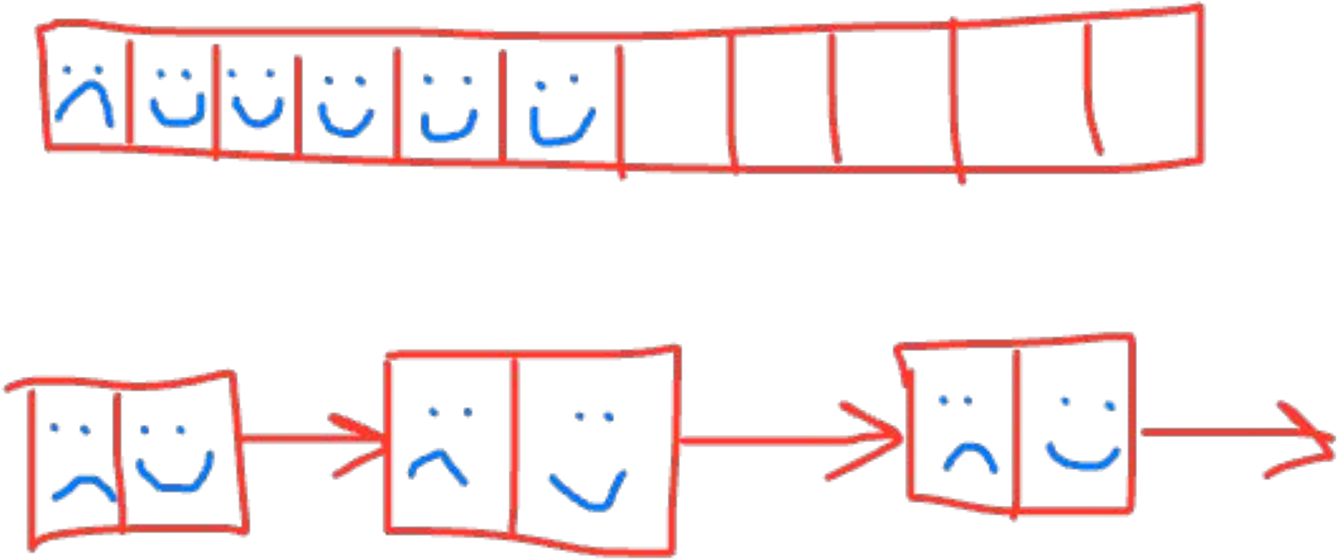
Eg a loop index, size field of a data structure

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

Eg elements in an array

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

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:
  - Pretend for a minute we had an AVL tree of height 55
  - The total number of nodes could be? $2^{56} - 1$
  - 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

- **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:

	Worst case big-O	**# Disk accesses**
• BST	O(n)	10M
• AVL	O(log n)	~25
• B Tree	O(log n)	3-4

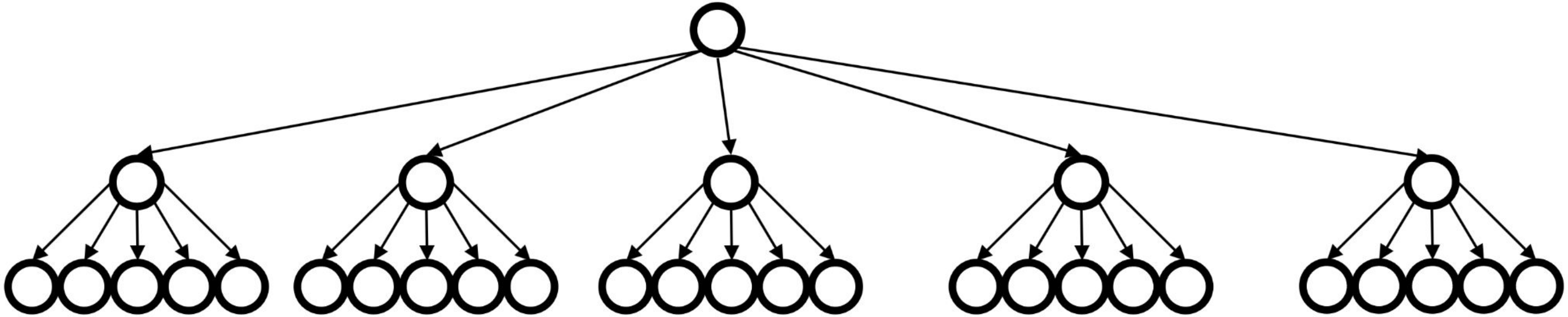
Our goal

- **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)

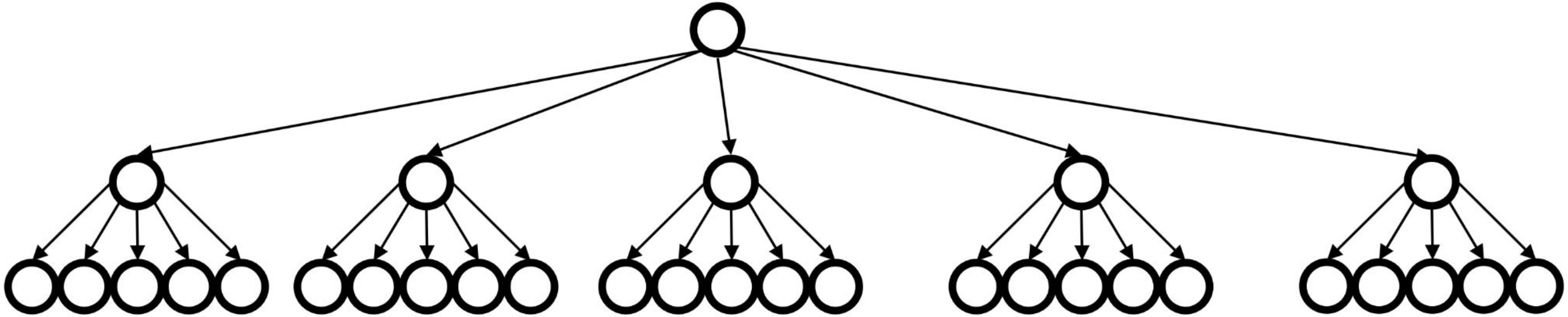


Perfect tree of height h has $(M^{h+1}-1)/(M-1)$ nodes (textbook, page 4)

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

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

Complexity of Find in M-ary Search Tree



How many **hops**?

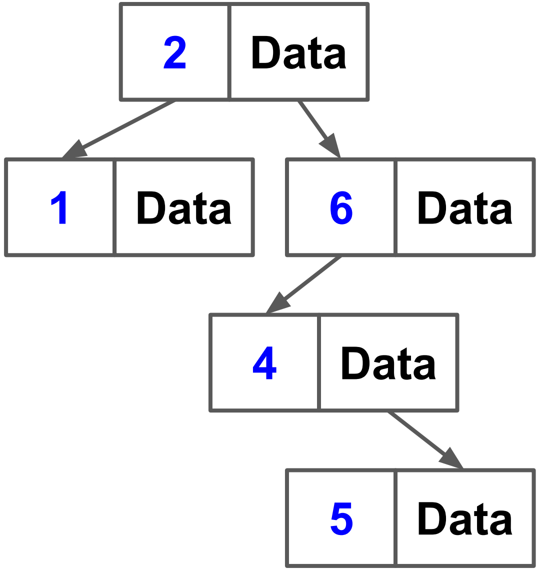
How much **work** at each level?

(find which child to take)

Overall complexity?

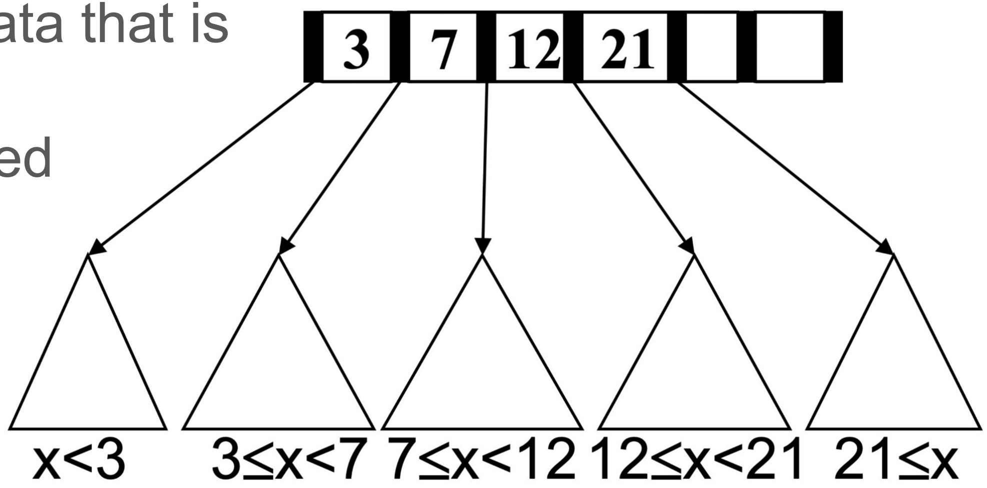
Questions about M-ary search trees

- 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 **≥ a** and **< b** (notice the **≥**)
- **Leaf** nodes have up to L sorted data items
- As usual, we'll ignore the "along for the ride" data

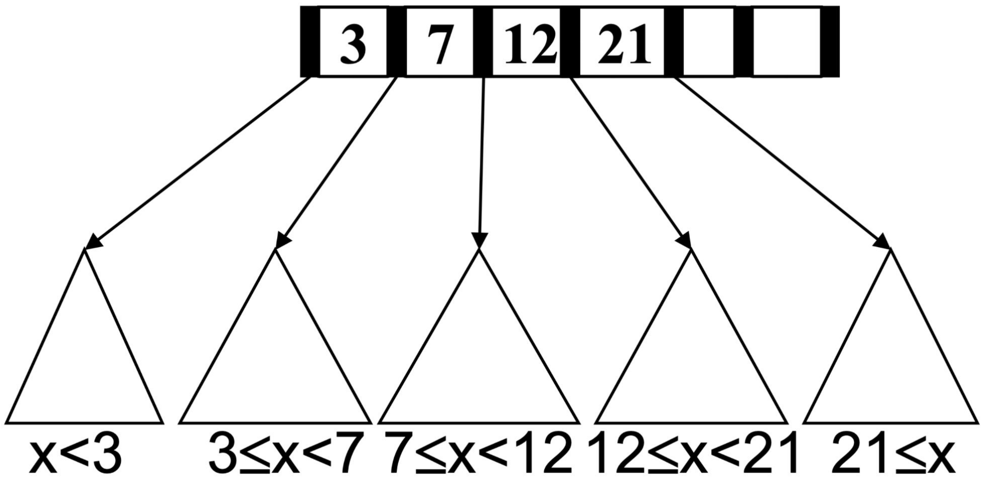


B Trees: Leaves vs Internal Nodes

Remember:

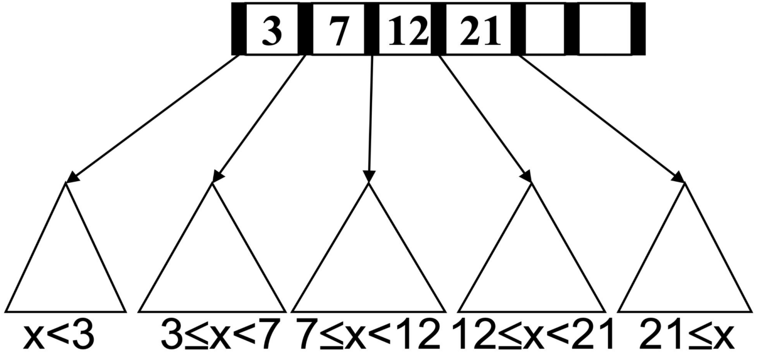
- **Leaves** store data
- **Internal nodes** are 'signposts'

- There are different ways to implement these - pay attention to how we talk about them!



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…



B Tree Structure Properties

- **Internal nodes**
  - Have between ⌈M/2⌉ and M children, i.e., at least half full
- **Leaf nodes**
  - All leaves at the same depth
  - Have between ⌈L/2⌉ and L data items, i.e., at least half full
- **Root** (special case)
  - If tree has ≤ L items, root is a leaf (occurs when starting up, otherwise unusual)
  - Else has between 2 and M children
- Any M > 2 and L will work, but:

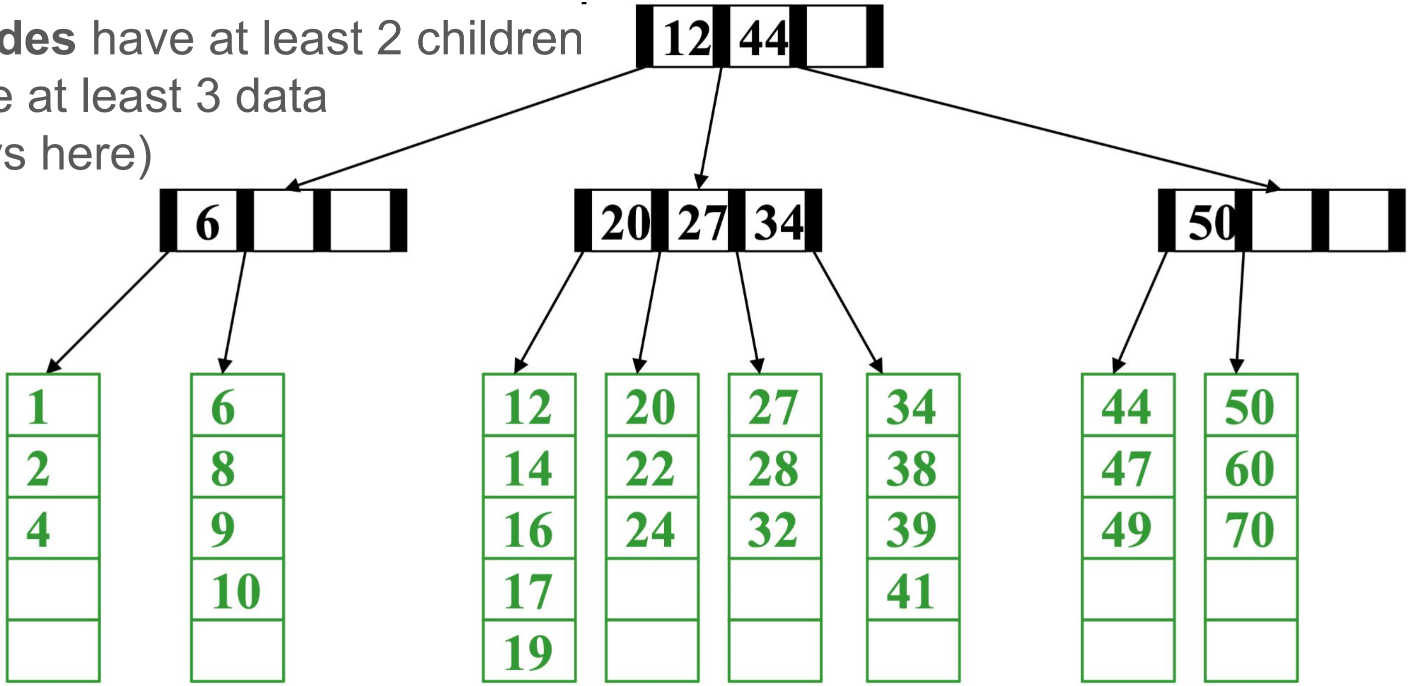
  We pick M and L ***based on disk-block size***

Example

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

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

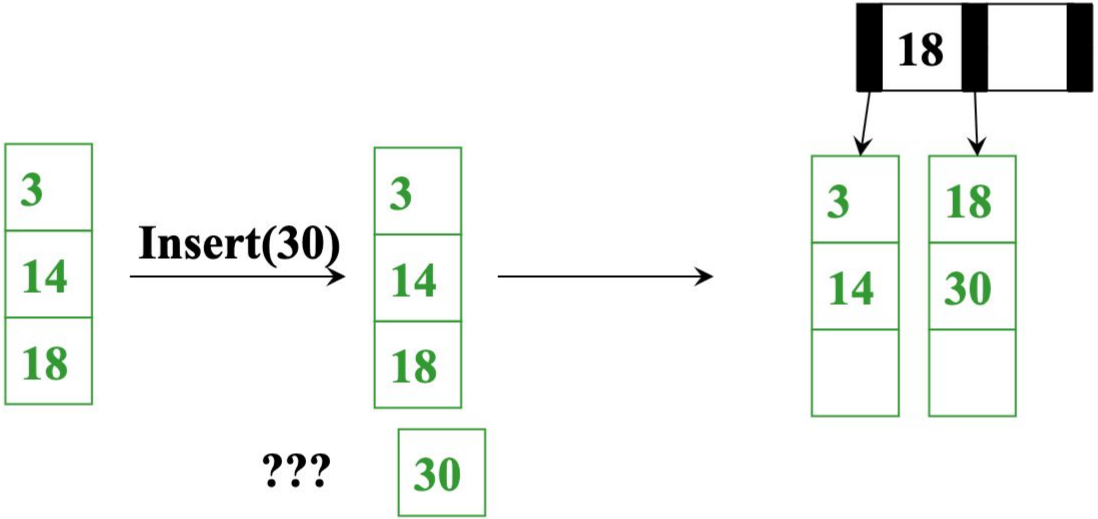
- All **internal nodes** have at least 2 children
- All **leaves** have at least 3 data items (only keys here)
- All **leaves** at same depth



find(28)

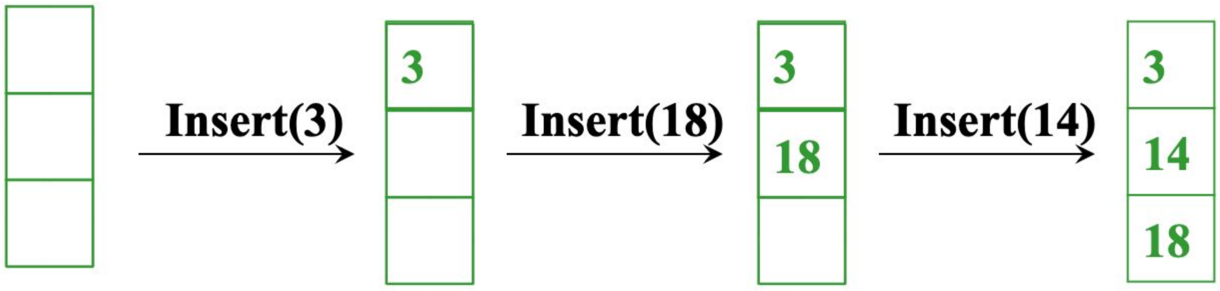
How many disk blocks did we touch?

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

Building a B-Tree (insertions)

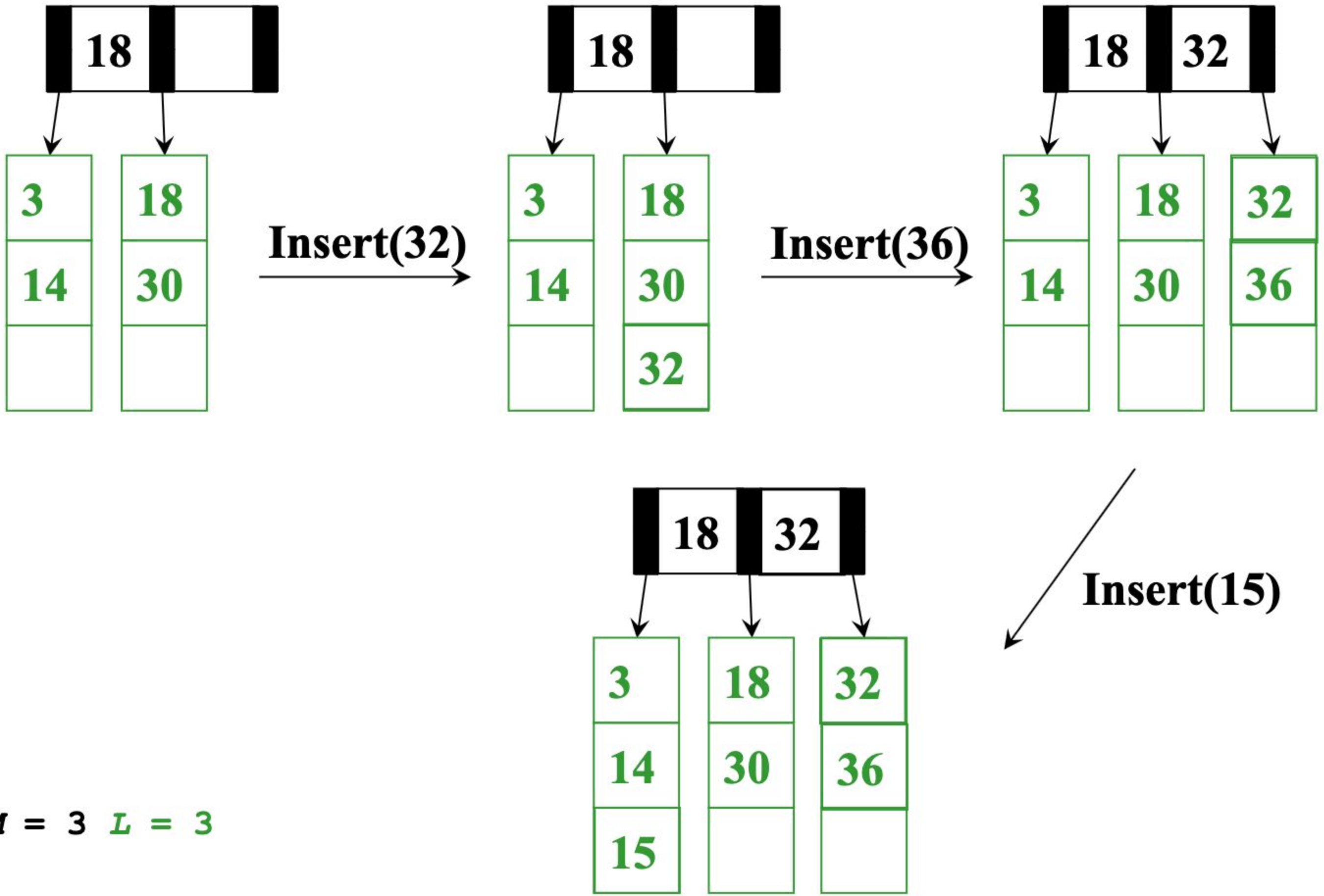


The empty B-Tree (the root will be a leaf at the beginning)

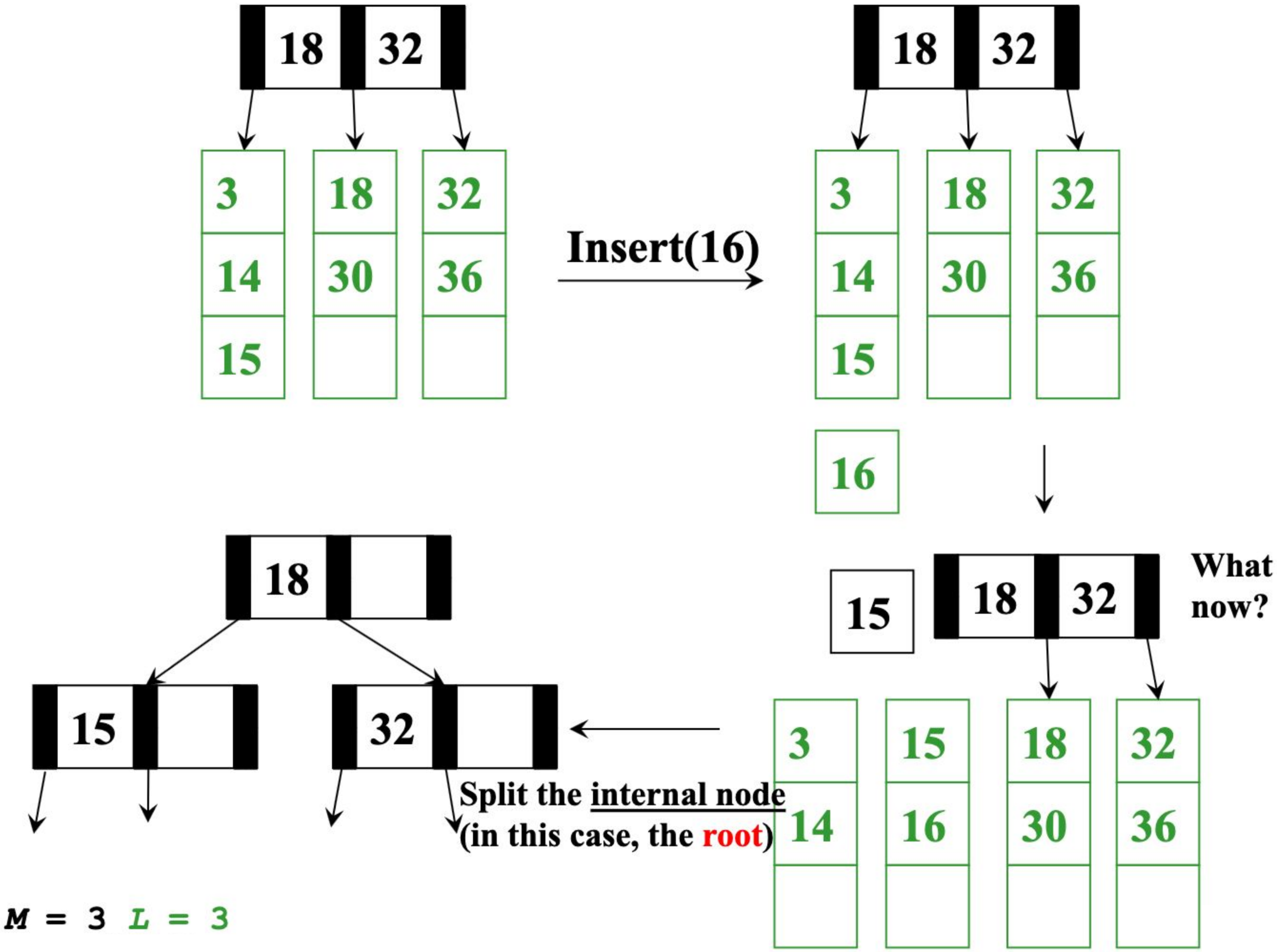
Just need to keep data in order

$M = 3$ $L = 3$

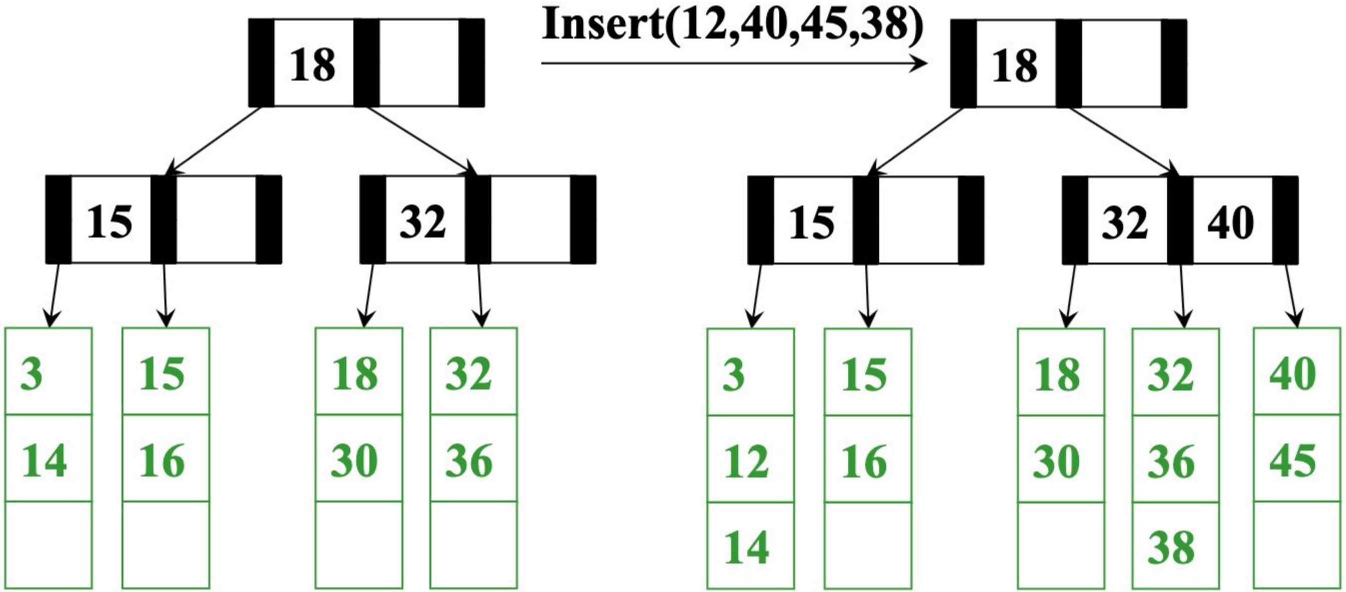
Split leaf again



M = 3 *L* = 3



18 32
3 14 15
18 30
32 36
Insert(16)
18 32
3 14 15
18 30
32 36
16
15
18 32
What now?
3 14
15 16
18 30
32 36
18
15
32
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

Insertion Algorithm continued

3. If step (2) caused the **internal node** parent to have M+1 children,
   - Split the **node** into **two nodes**
     - Original **node** with $\lceil \mathtt{(M+1)/2} \rceil$ smaller items
     - New **node** with $\lfloor \mathtt{(M+1)/2} \rfloor = \lceil \mathtt{M/2} \rceil$ 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

Worst-Case 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: $O(M \log_M n)$

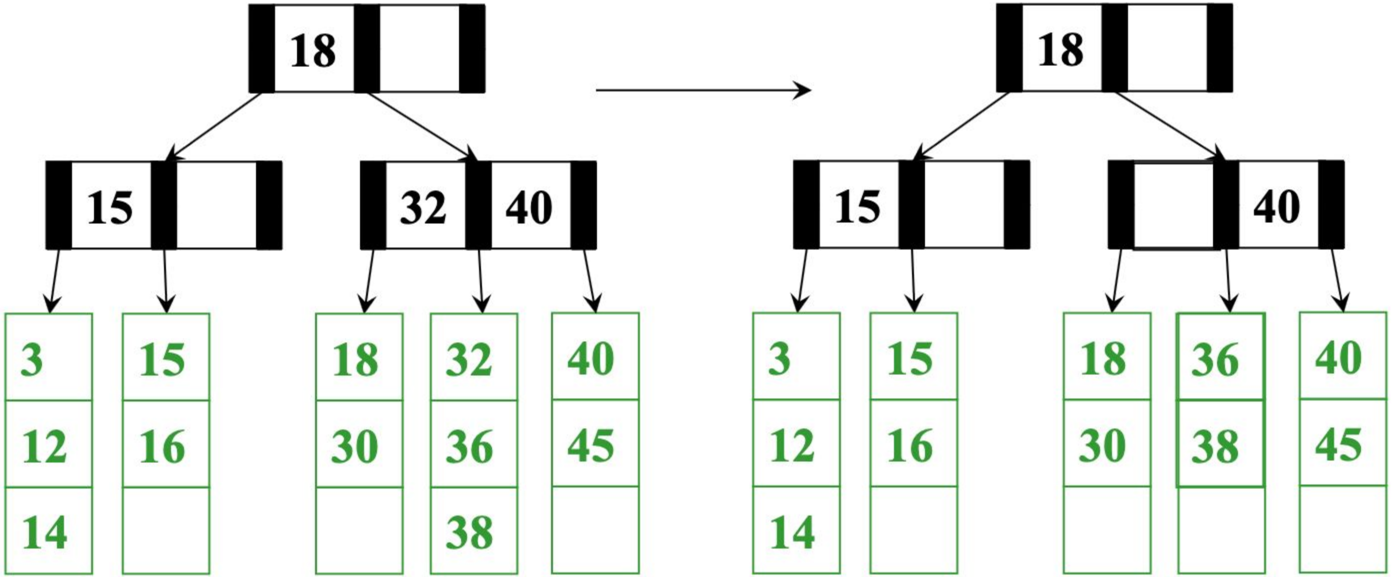
Total: **$O(L + M \log_M n)$**

But it's not that bad:

- Splits are not that common (M & L are likely to be large)
- Disk accesses are the name of the game: $O(\log_M n)$

And Now for Deletion…

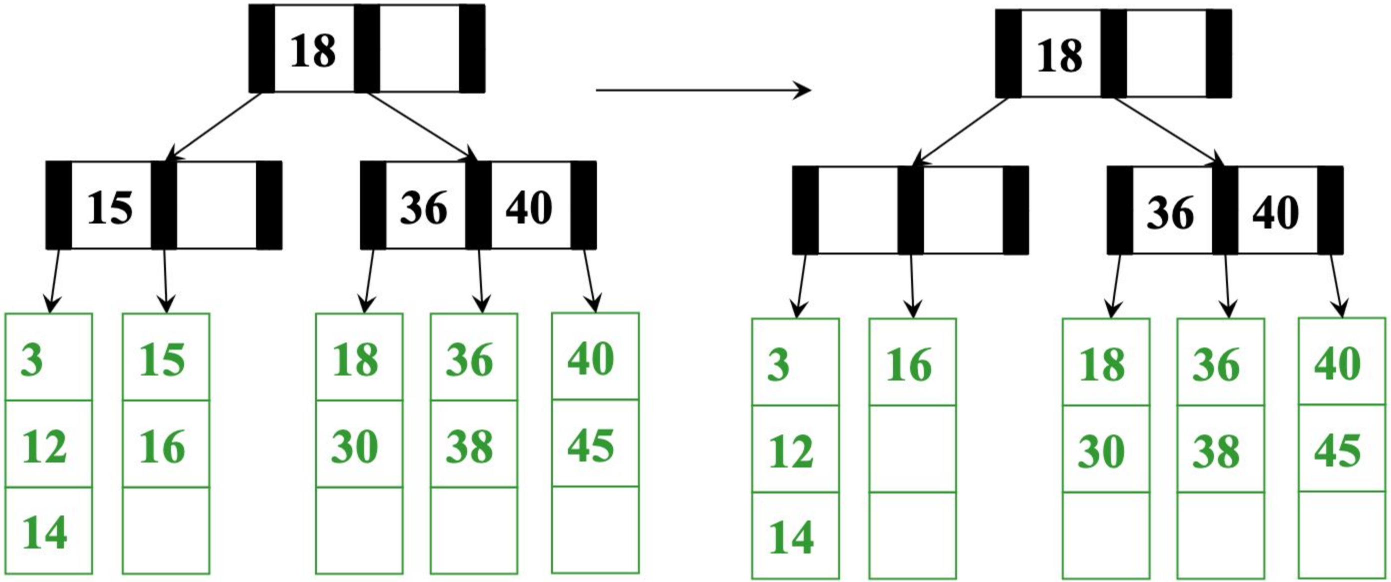
Delete(32)



Easy case: Leaf still has enough data; just remove

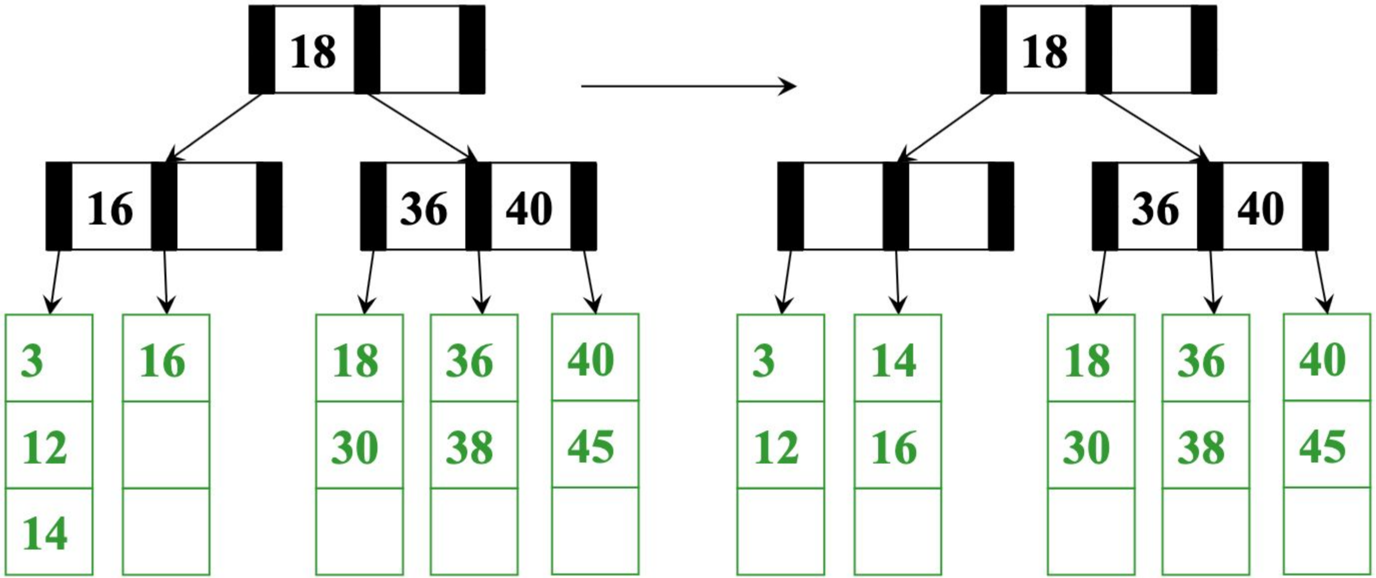
`M = 3 L = 3`

Delete(15)



Is there a problem?

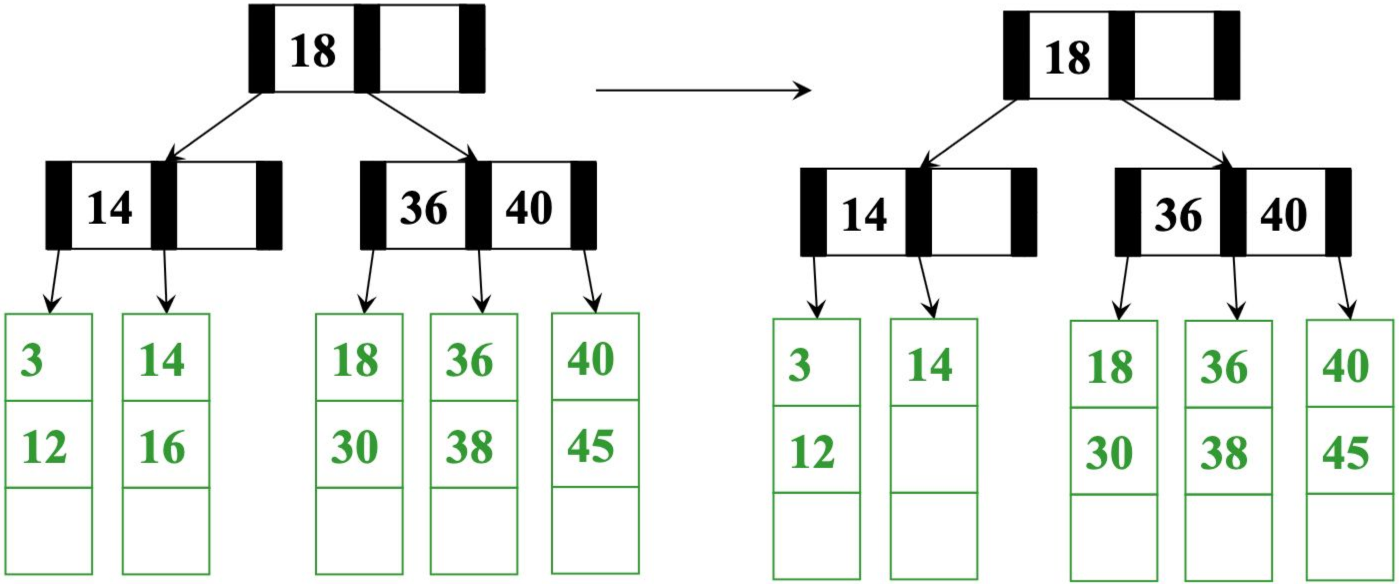
$M = 3$ $L = 3$



$M = 3$ $L = 3$

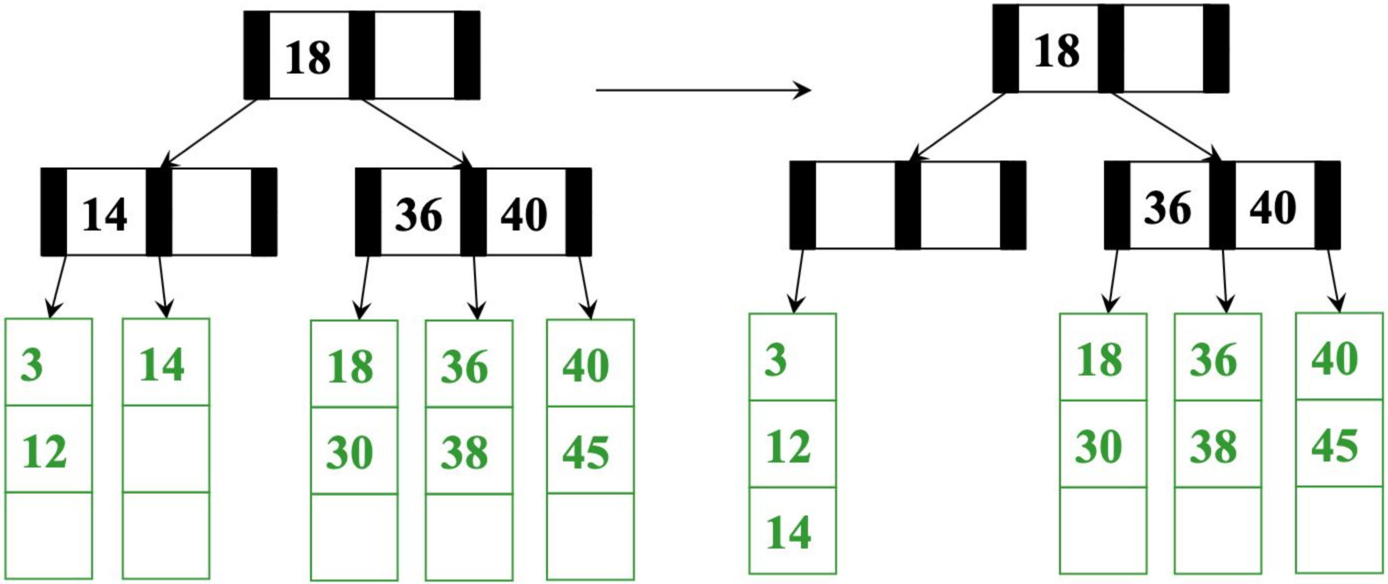
Adopt from neighbor!

Delete(16)



$M = 3$ $L = 3$

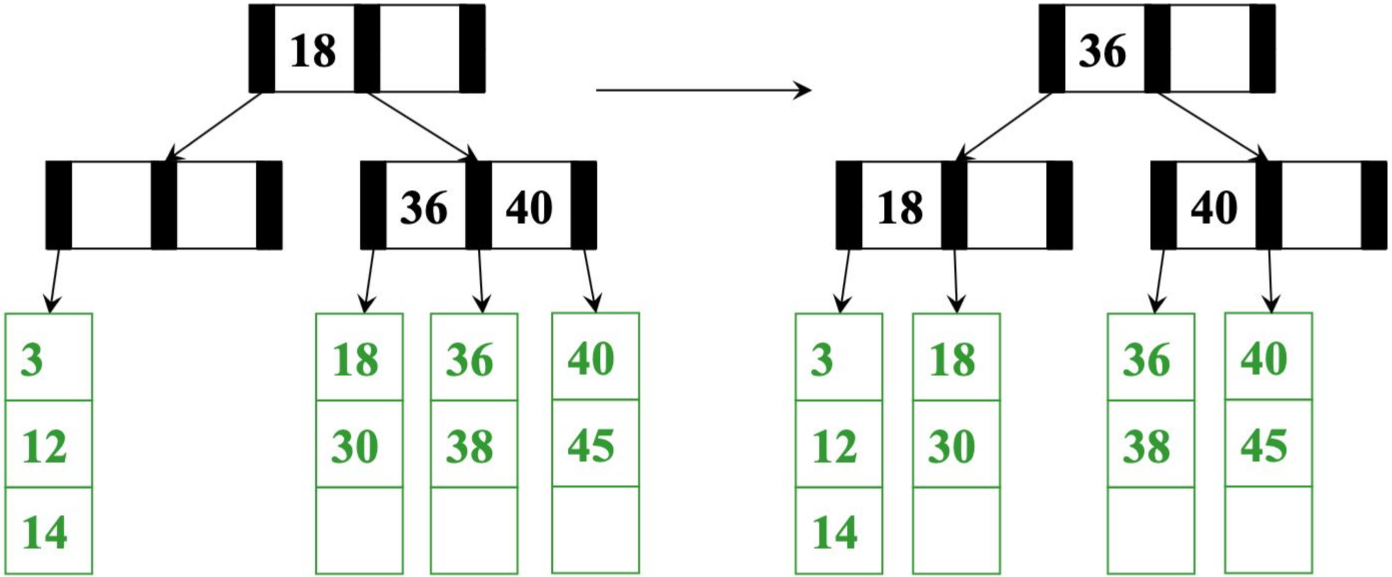
Is there a problem?



Merge with neighbor!

M = 3 *L* = 3

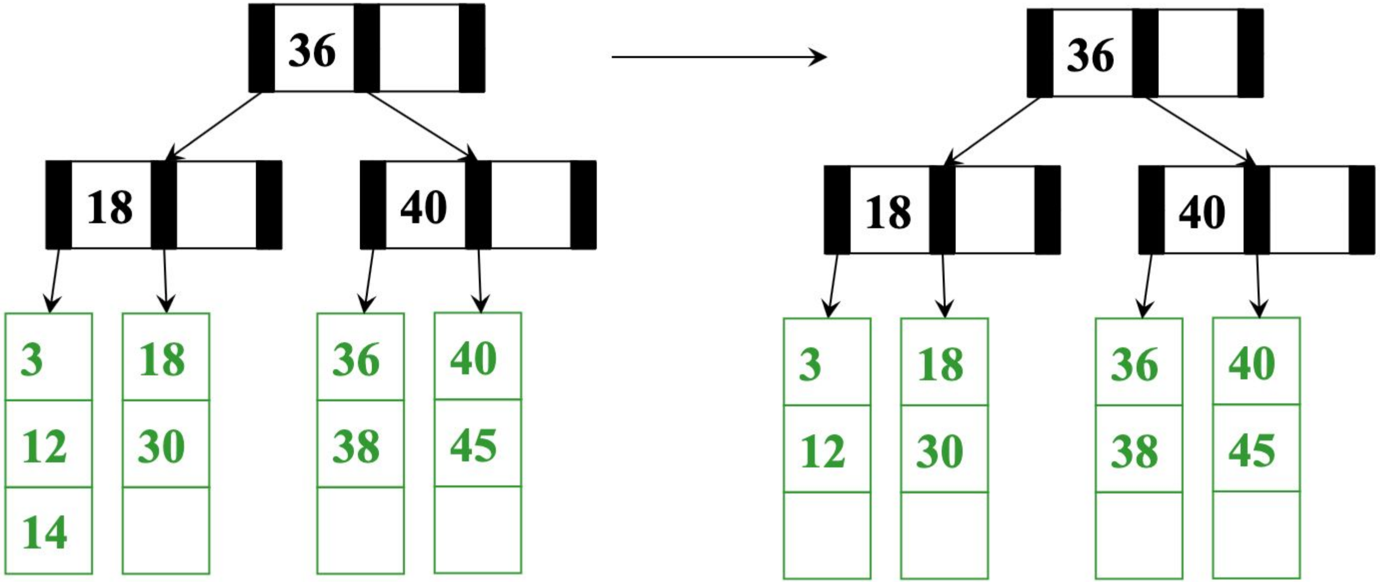
But hey, Is there a problem?



$M = 3$ $L = 3$

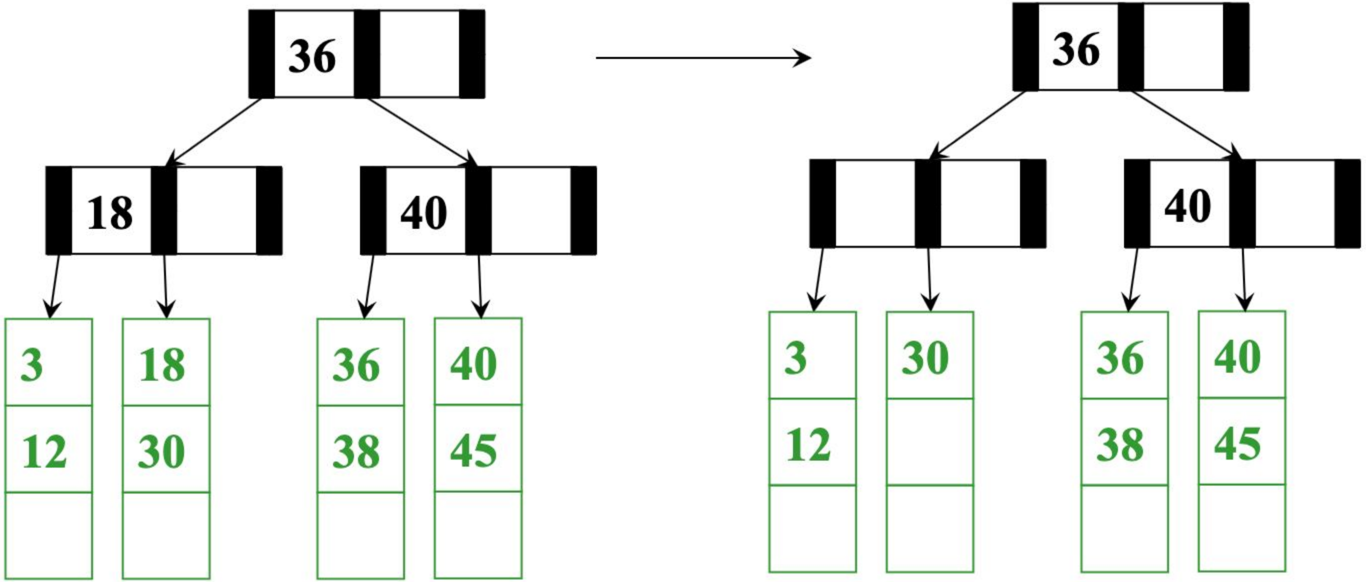
Adopt from neighbor!

Delete(14)



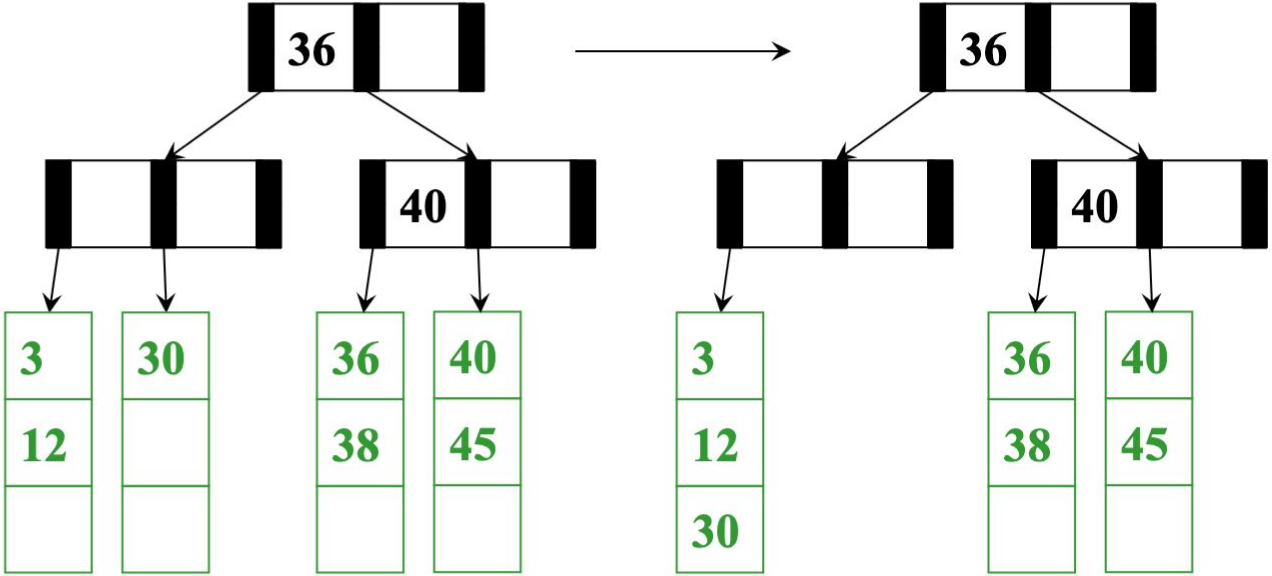
M = 3 *L* = 3

Delete(18)



Is there a problem?

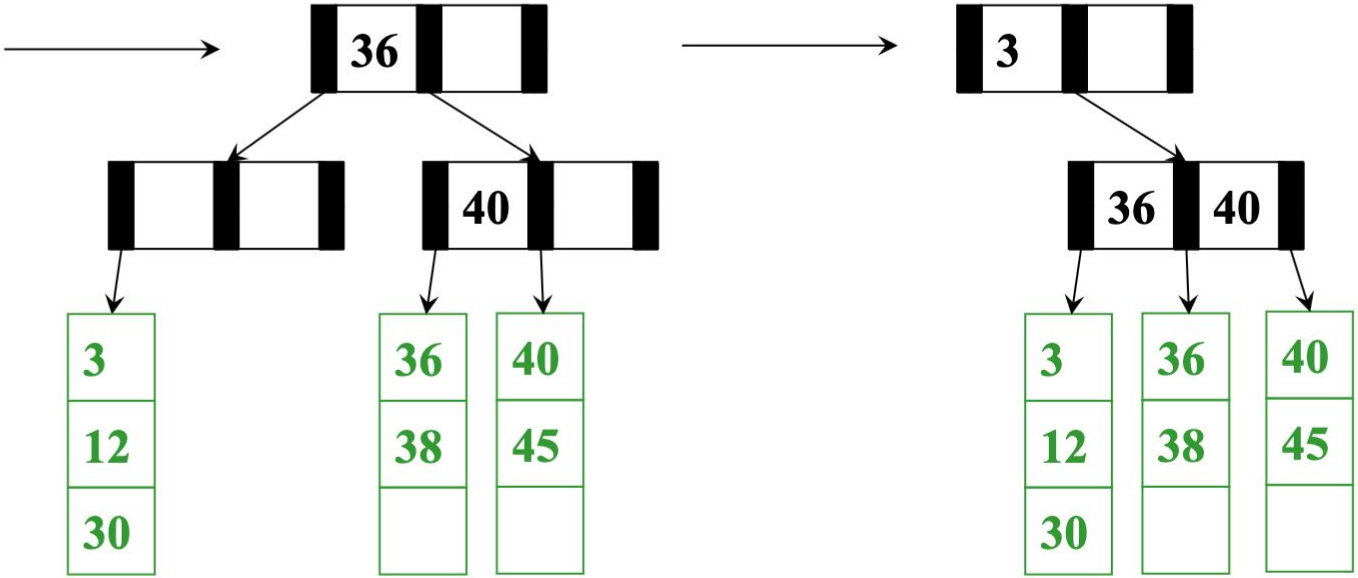
$M = 3$ $L = 3$



Merge with neighbor!

$M = 3$ $L = 3$

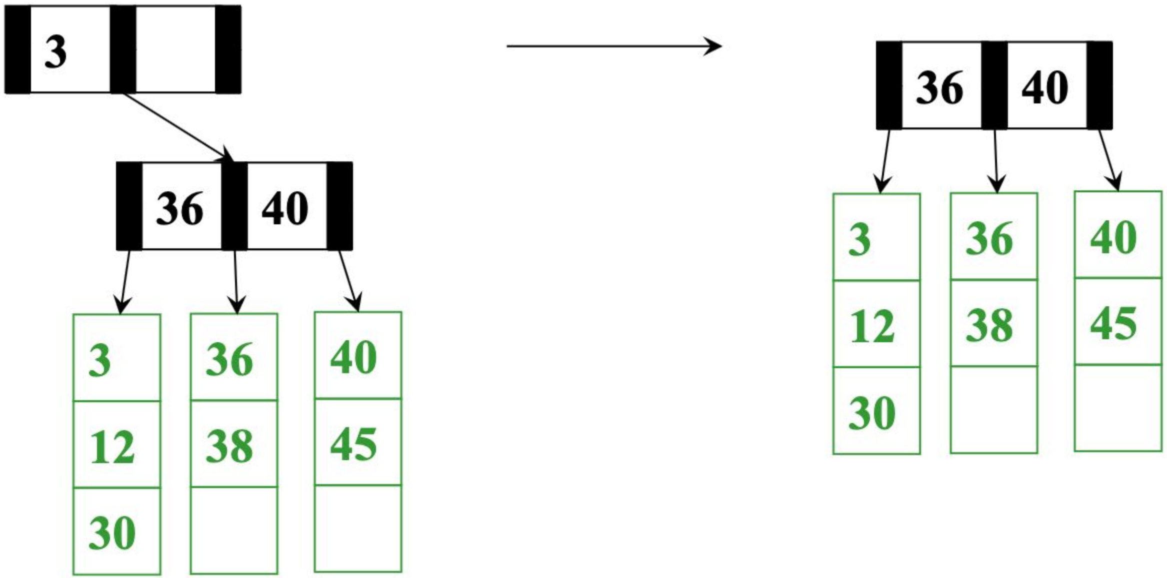
But hey, Is there a problem?



$M = 3$ $L = 3$

Merge with neighbor!

But hey, Is there a problem?



Pull out the root!

$M = 3$ $L = 3$

Deletion Algorithm

1. Remove the data from its **leaf**
2. If the **leaf** now has $\lceil L/2 \rceil - 1$ items, *underflow!*
   - If a neighbor has > $\lceil L/2 \rceil$ items, *adopt* and update parent
   - Else *merge* node with neighbor
     - Guaranteed to have a legal number of items
     - Parent now has one less node

Deletion Algorithm continued

3. If step (2) caused the **internal node** parent to have $\lceil M/2 \rceil - 1$ children, *underflow!*
   - If a neighbor has > $\lceil M/2 \rceil$ items, *adopt* and update parent
   - Else *merge* node with neighbor
     - Guaranteed to have a legal number of items
     - Parent now has one less node

Merging at a node (step 3) could make the parent underflow too

- *So repeat step 3 up the tree until a node doesn't underflow*
- If the **root** went from 2 children to 1, delete the root and make the child the root
  - This is the only case that decreases the 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: $\mathbf{O(L + M \log_M n)}$

But it's not that bad:

- Merges are not that common
- Disk accesses are the name of the game: $O(\log_M n)$

Determining *M* & *L*

Say:

1 disk block	= 1024 bytes
Key	= 8 bytes
Pointer	= 4 bytes
Data(K, V)	= 500 bytes *(includes key)*

Determining *L*: How much data can fit?

L = 1024 / 500 = about 2

Determining *M*: how many interior nodes can fit?

Each interior node has M pointers and M-1 keys.

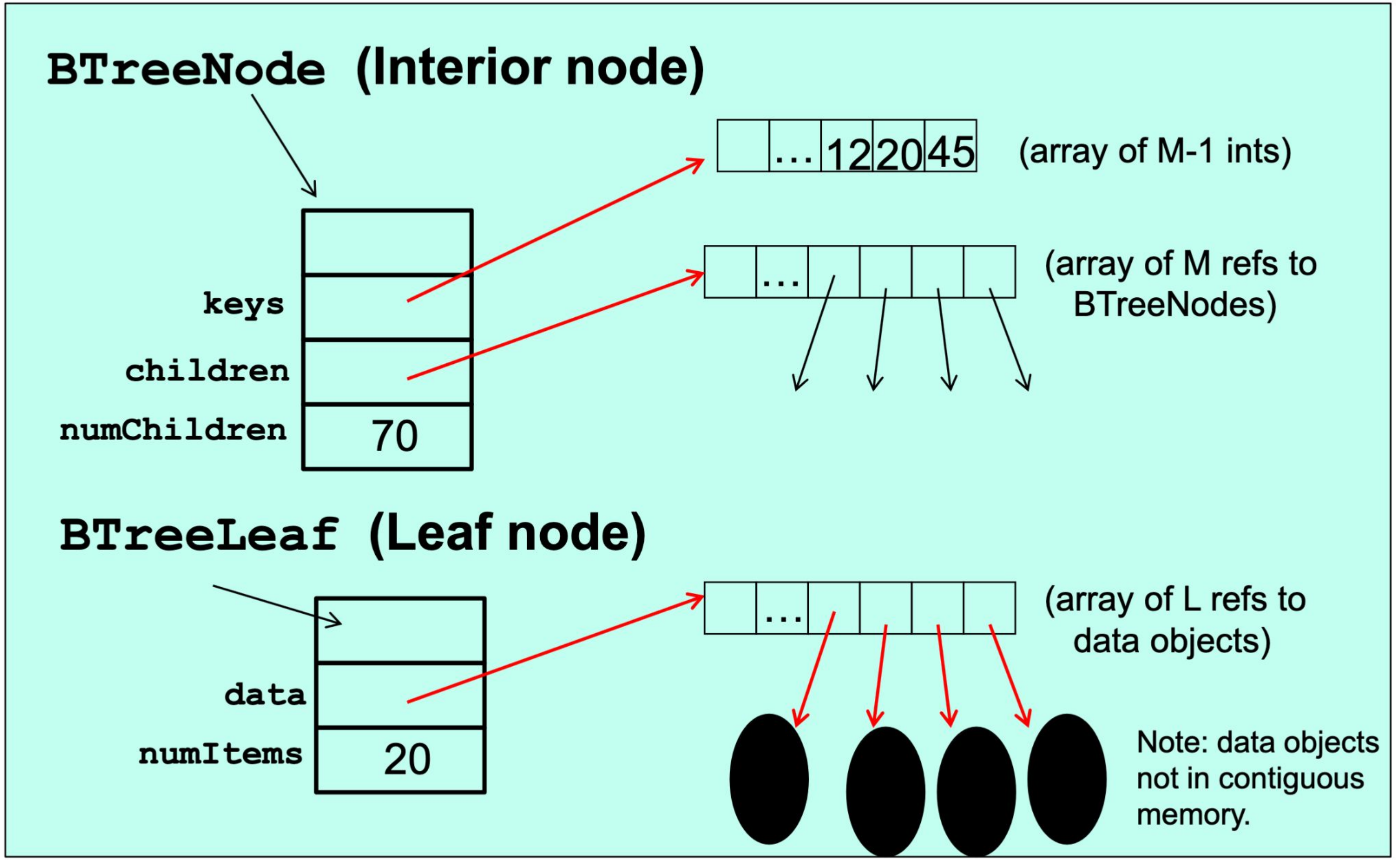
$$1024 \geq 4M + 8(M-1)$$
$$1024 \geq 4M + 8M - 8$$
$$1024 \geq 12M - 8$$
$$1024+8 \geq 12M$$
$$1032/12 \geq M$$
$$M = 86$$

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