CSE 332: Data Structures & Parallelism
Lecture 8: B-Trees

Arthur Liu

Summer 2022

Announcements

- In-class Midterm Next Monday!
  - Be here on time, we will start promptly at 9:40-10:40am to give you the full hour
  - Midterm Resources will be posted later today
  - Midterm Review Session on Friday time TBD

Deletion – The Two Child Case

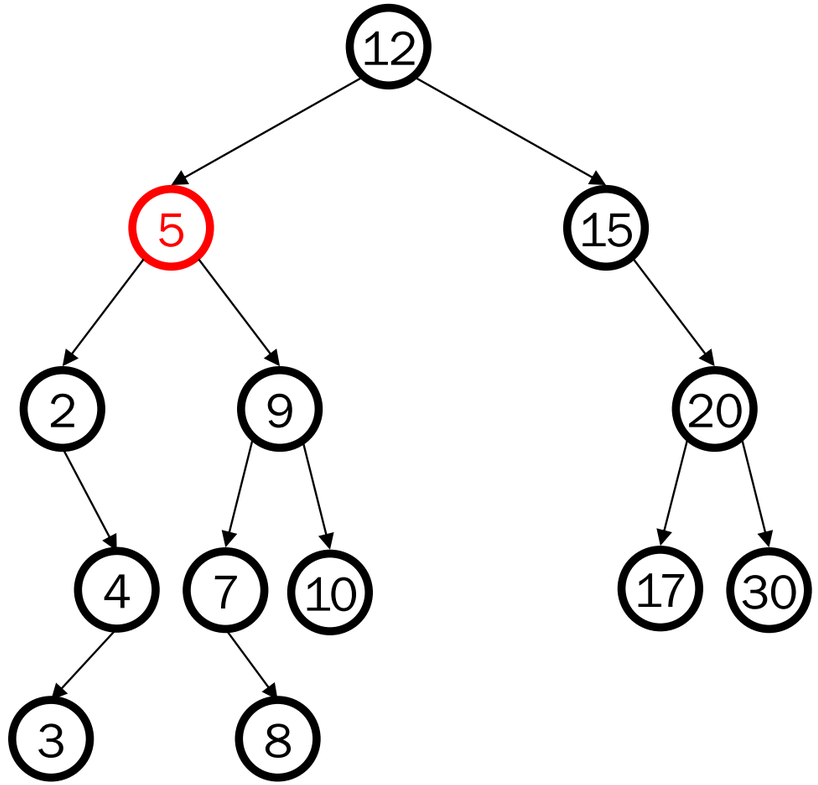
`delete(5)`

The easiest solution involves replacing the deleted node with its successor (7) or predecessor (4) value.

- “Easiest” because we know exactly where the rest of the subtrees should go.

Cases:

- Replacement node has no children -> then nothing to worry about!
- Replacement node has 1 children -> just replace the *replacement* node with that child
- Replacement node has 2 children(?) (This case cannot happen!)



Outline for Today

- Finish AVL
  - How to handle insertion (4 cases)
  - AVL size math (and height guarantees)
- B-Trees
  - Motivating B-Trees, a memory perspective
  - B-Tree structure
  - B-Tree methods
    - Insertion AND deletion

Dictionaries Review

We've talked about 1.5 "standard" dictionaries:

AVL Trees – O(log n) find, insert, and delete

- Optimize for worst case
- BST can only give average case logarithmic

Today: A New Problem

Design Goal: Optimize behavior for huge datasets

- Need to understand how memory works

A false assumption

We said “every operation takes about the same time”

In order to do big-O analysis

Most of the time this is true.

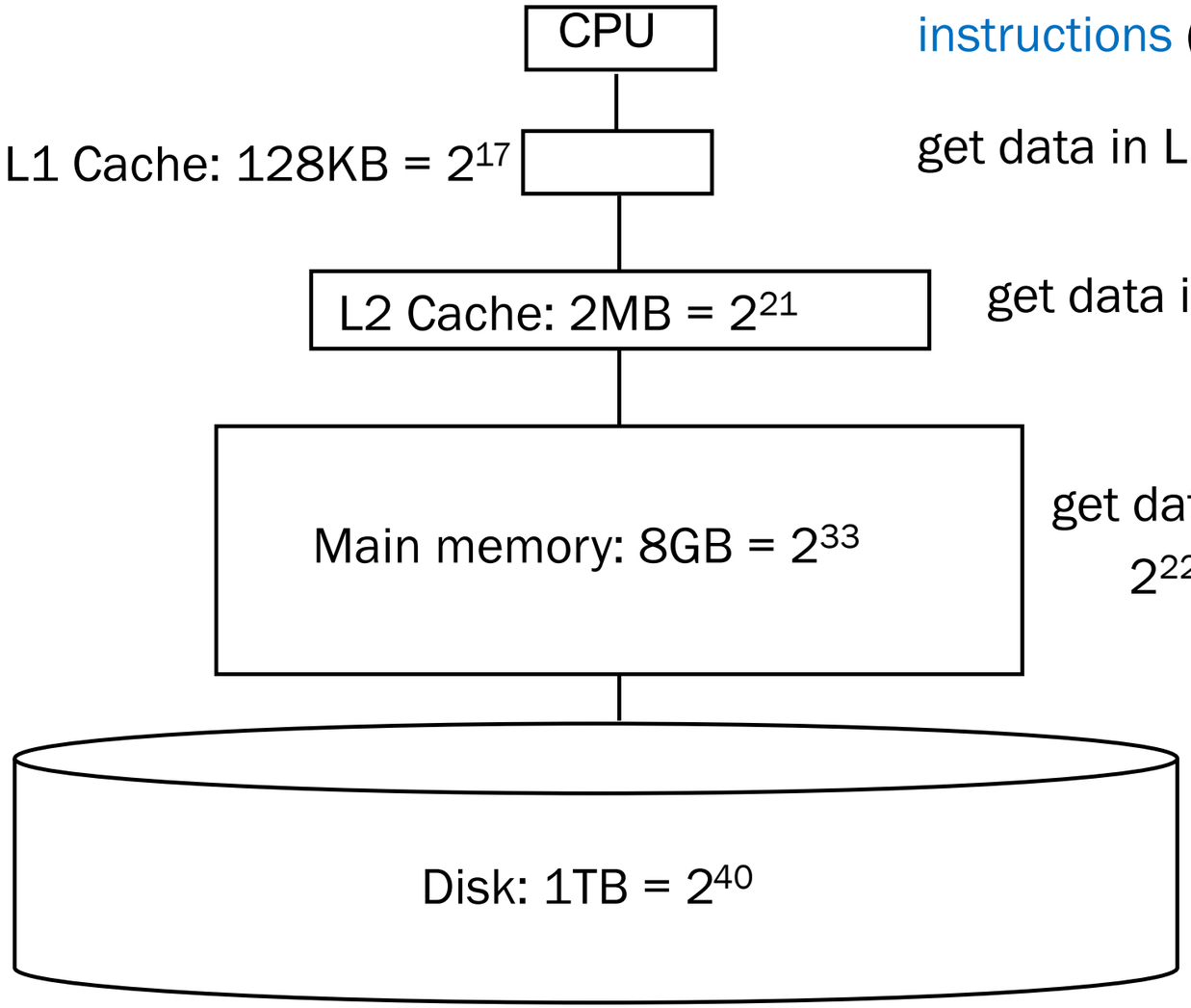
It’s always true up to a constant factor

- But what if that constant is 5,000,000?

Today: What to do when your dictionary is huge

A typical hierarchy

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



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

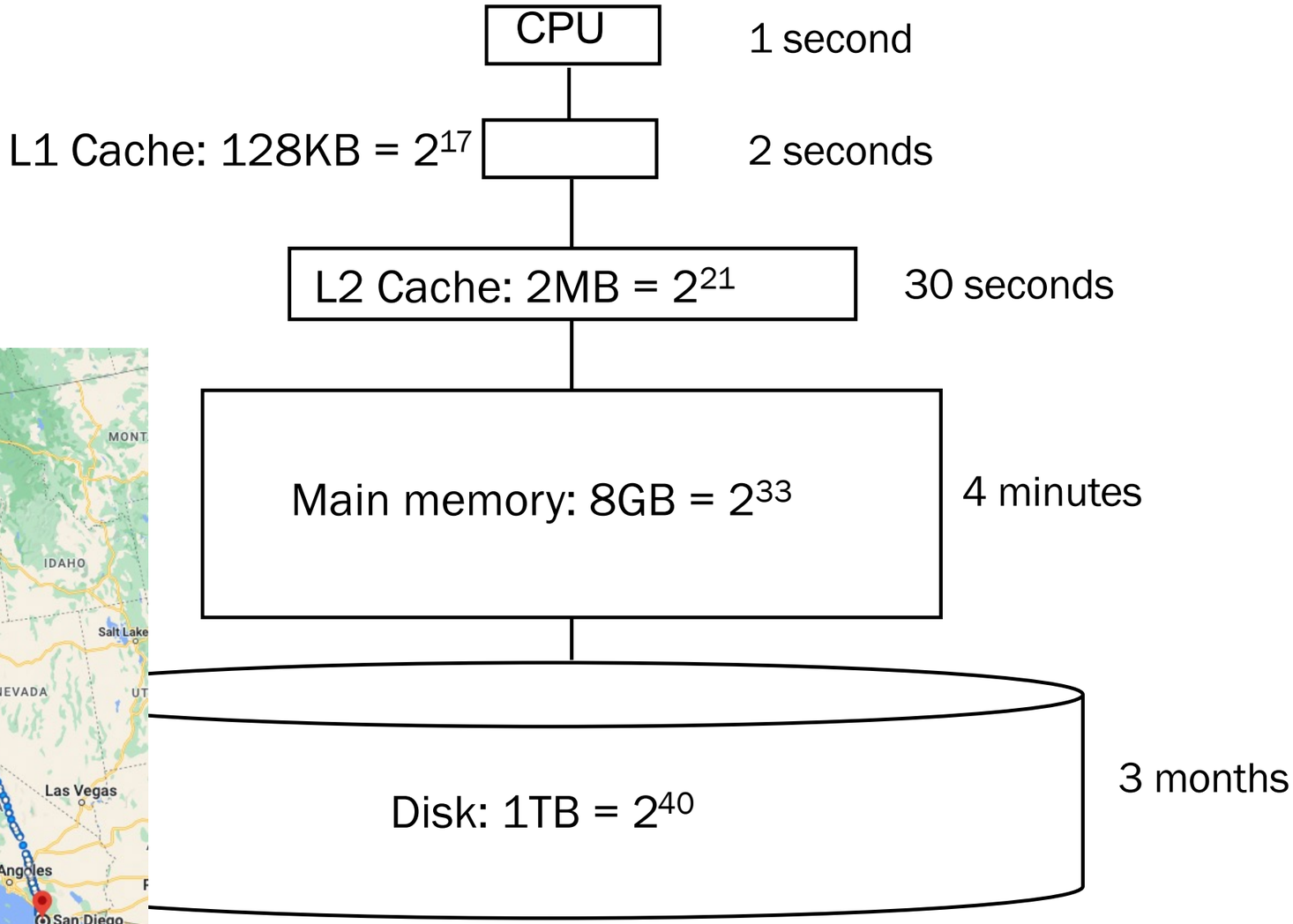
get data in L1: 2^{29}/sec = 2 instructions

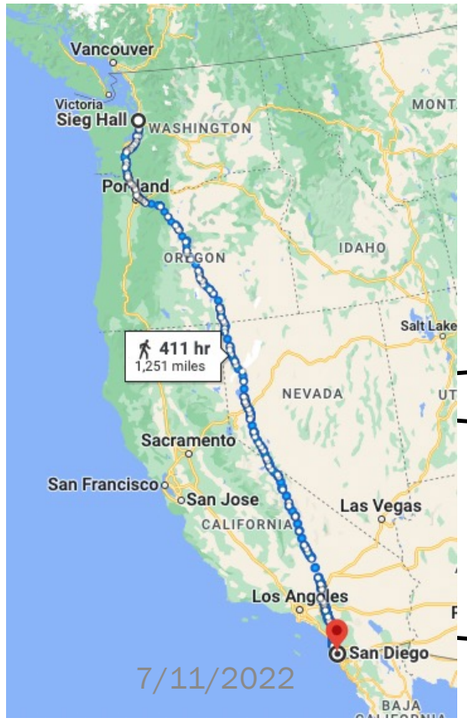
get data in L2: 2^{25}/sec = 30 instructions

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

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

A typical hierarchy





Why have we not noticed this?

Very smart people work on algorithms to quickly decide what memory to put in caches (in case you need it again).

If your data fits in the cache, you'll probably never notice.

Compiler optimizations can sometimes ask for data before you need it.

When you ask for one piece of data, the OS will give you that, and everything near it.

- It's likely to be used soon (think arrays).

Once you use a value, the OS will keep it in a close by cache.

Two Principles

Temporal Locality:

- If you use some piece of memory, you are likely to use that exact data again pretty soon.

Spatial Locality:

- If you use some piece of memory, you are likely to use nearby data pretty soon.

OS accomplishes this by:

Keeping recently used memory in the cache

Moving memory in “pages” (blocks/pages/lines) – if you access one data point, you move everything nearby with it

Pause to Ponder

pollev.com/artliu

We’ve seen two dictionary types:

Search tree-based dictionaries

Array-based dictionaries

Which one should we use when we have a huge dataset?

Memory Accesses – Dictionaries

Suppose we have a dictionary with about 2^{50} elements.

Could store as an AVL tree of height about 50.

How many disk accesses might it take to `find`?

What if we made the tree shorter?

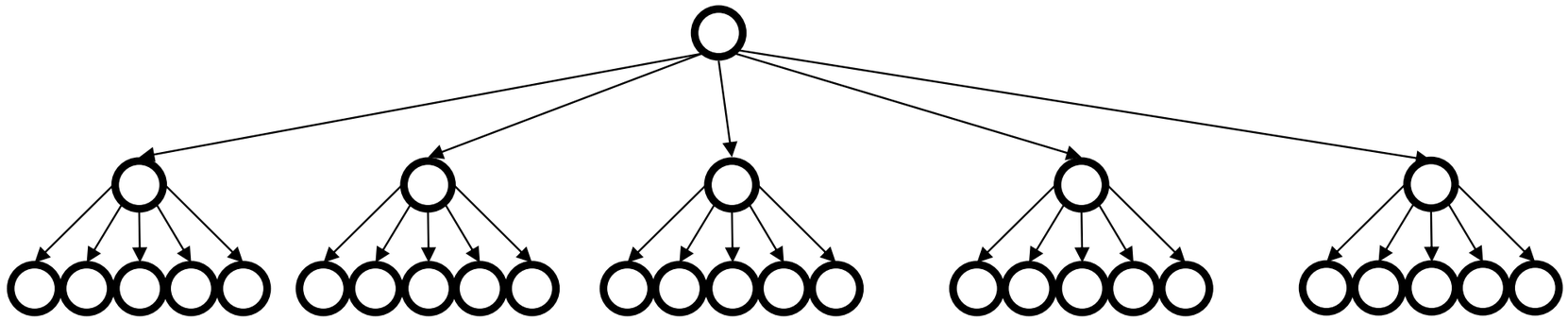
Make it an M-ary tree, not a binary tree.

Disk accesses? Only $50 \log_M 2$

If $M = 30$, we'll cut the memory accesses to about 10.

M will often be even bigger.

M is not entirely free



- # hops for find?
  - Approx. $\log_M n$ hops instead of $\log_2 n$ (for balanced BST)

- But how do we decide which branch to take?
  - Use binary search at every branch node $\log_2 m$
  - Need to pick an M so binary search happens within one memory block

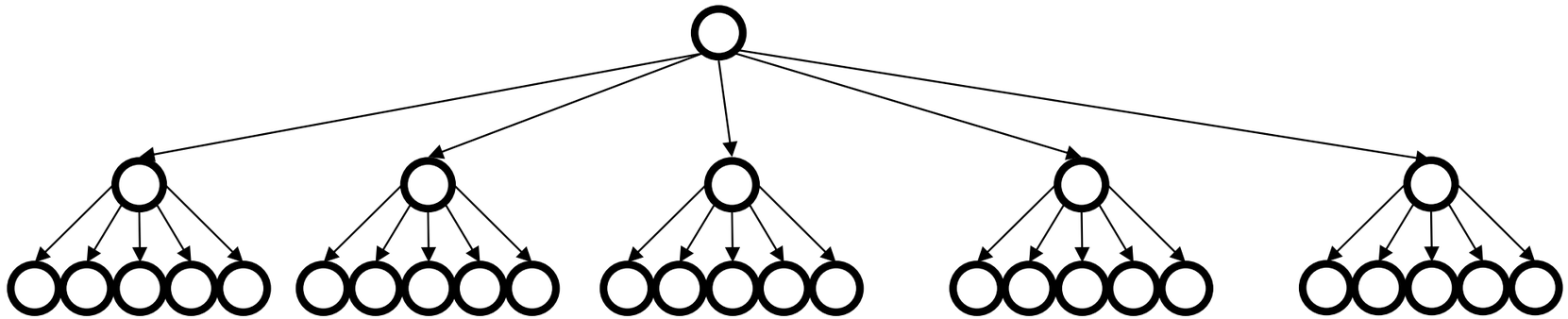
- Runtime of `find` if balanced: $O(\log_2 M \log_M n)$
  - $\log_M$ n is the height we traverse, $\log_2 M$ is the cost at every node

Some flaws of just an M-ary tree

- Storing real data at inner-nodes seems wasteful
  - A lot of the times we are just “passing through” a node on the way to the node that we are actually looking for

Disk Block:

Key: 7
Value:
<TheBeeMovie.mp
4TheBeeMovie.mp
4TheBeeMovie.mp
4>
Children Pointers:
Ptr1, ptr2, ptr3,
ptr4, ptr5



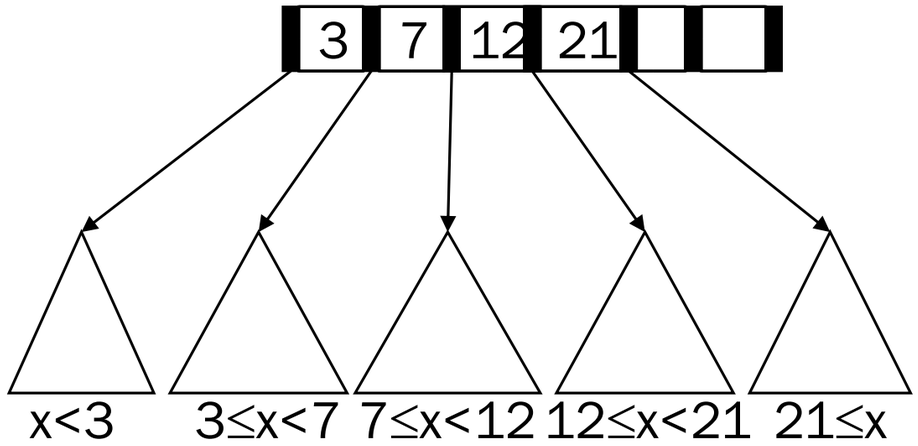
Outline for Today

- Finish AVL
  - How to handle insertion (4 cases)
  - AVL size math (and height guarantees)
- B-Trees
  - Motivating B-Trees, a memory perspective
  - B-Tree structure
  - B-Tree methods
    - Insertion AND deletion

B+ Trees Parameters (we and the book say “B Trees”)

- Two types of nodes: **internal nodes** & leaves

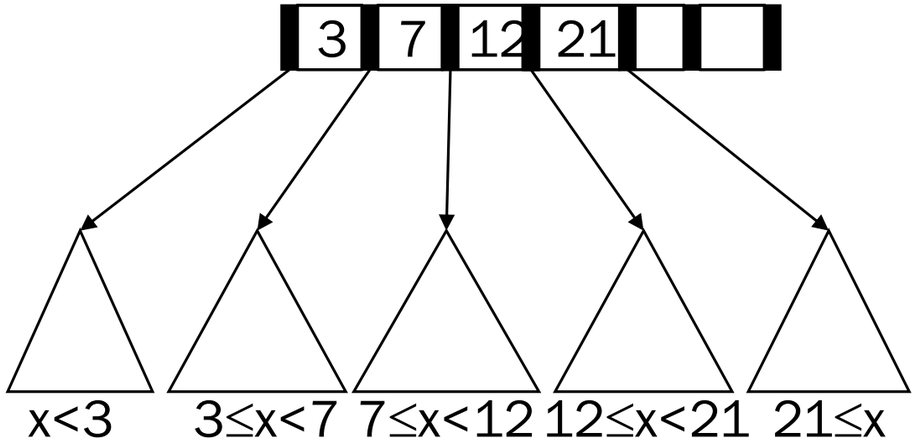
- Each **internal node** has room for up to *M-1* keys and *M* children
  - Does not store values
  - Function as “sign-posts”



- Leaf nodes have up to *L* *sorted* data items

3	“cat”
14	“apple”
15	“purple”
21	“ideas”

Find



- Different from BST in that we *don't store data at internal nodes*
- 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…

Structure Properties

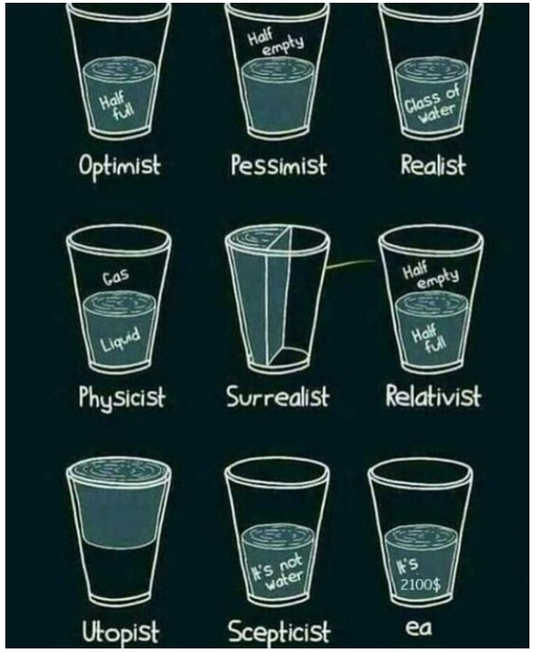
- Root (special case)
  - If tree has $\leq L$ items, root is a leaf (if almost empty – this should be very rare)
  - Else has between 2 and M children

- 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, but:

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

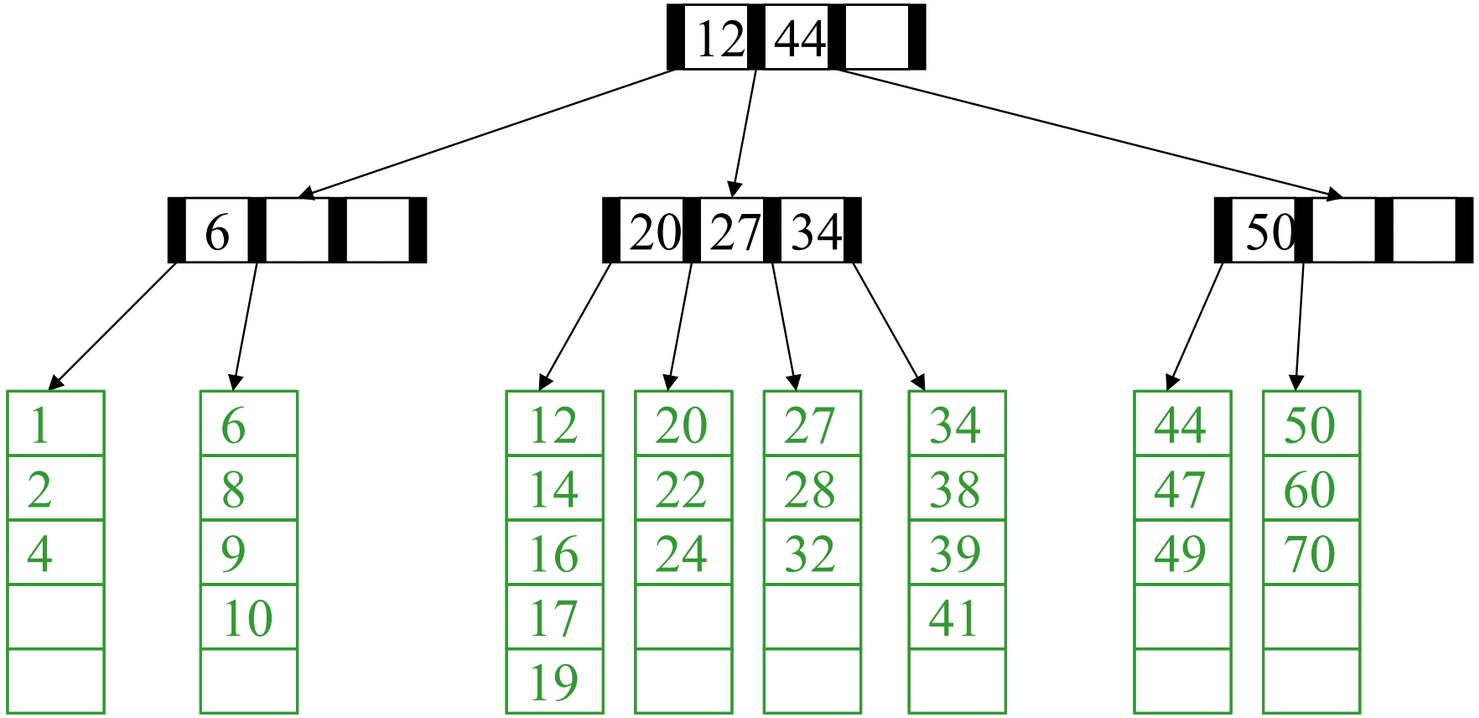


Example

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 showing keys)
- All leaves at same depth



Is this balance condition good 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 – 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…

Minimum branching at root

$$n \geq 2 \left\lceil \frac{M}{2} \right\rceil^{h-1} \left\lceil \frac{L}{2} \right\rceil$$

minimum number of leaves

minimum data per leaf

Example Scenario

pollev.com/artliu

$$n \geq 2 \left\lceil \frac{M}{2} \right\rceil^{h-1} \left\lceil \frac{L}{2} \right\rceil$$

Suppose we have 100,000,000 items

Maximum height of AVL tree?

Maximum height of B tree with M = 128 and L = 64?

Example Scenario

$$n \geq 2 \left\lceil \frac{M}{2} \right\rceil^{h-1} \left\lceil \frac{L}{2} \right\rceil$$

Suppose we have 100,000,000 items

Maximum height of AVL tree?

37 (Recall: S(h) = 1 + S(h-1) + S(h-2)

Maximum height of B tree with M = 128 and L = 64?

5 (Recall: $n \geq 2\lceil M/2 \rceil^{h-1} \lceil L/2 \rceil$)

How do we pick M and L?

We need to make sure M is as large as possible (to make tree as shallow as possible)

But…

We need to also make sure that internal node and leaf node both completely fit inside a page block. (Otherwise, binary searching within a singular node can be really expensive!)

Some Formulas You Should Be Able to Derive

Size of Internal Node:

$$(M-1)\cdot key + M\cdot pointer \leq page$$

Size of Leaf Node:

$$L\cdot data \leq page$$

Rearranging for M and L

$$M = \left\lfloor \frac{page+key}{pointer+key} \right\rfloor \qquad L = \left\lfloor \frac{page}{data} \right\rfloor$$

B-Trees and Disks <3

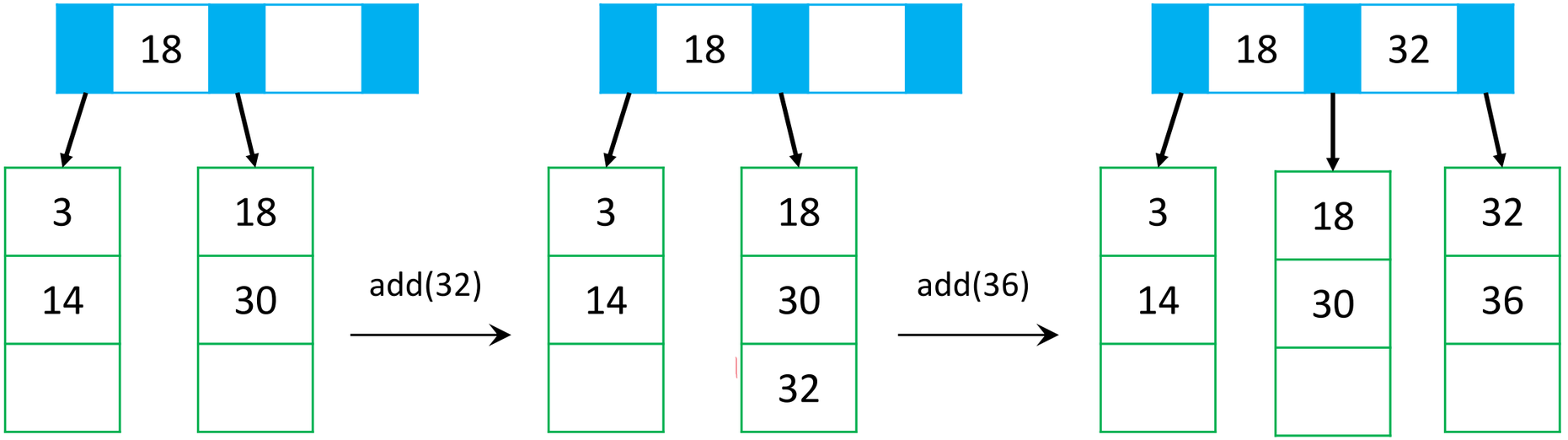
What makes B-trees so disk friendly?

- Many keys stored in one internal node
  - All brought into memory in one disk access
- Internal nodes contain only keys
  - Any **`find`** wants only one data item; wasteful to load unnecessary items with internal nodes
  - Data-item size doesn't affect what *M* is

Outline for Today

- Finish AVL
  - How to handle insertion (4 cases)
  - AVL size math (and height guarantees)
- B-Trees
  - Motivating B-Trees, a memory perspective
  - B-Tree structure
  - B-Tree methods
    - Insertion AND deletion

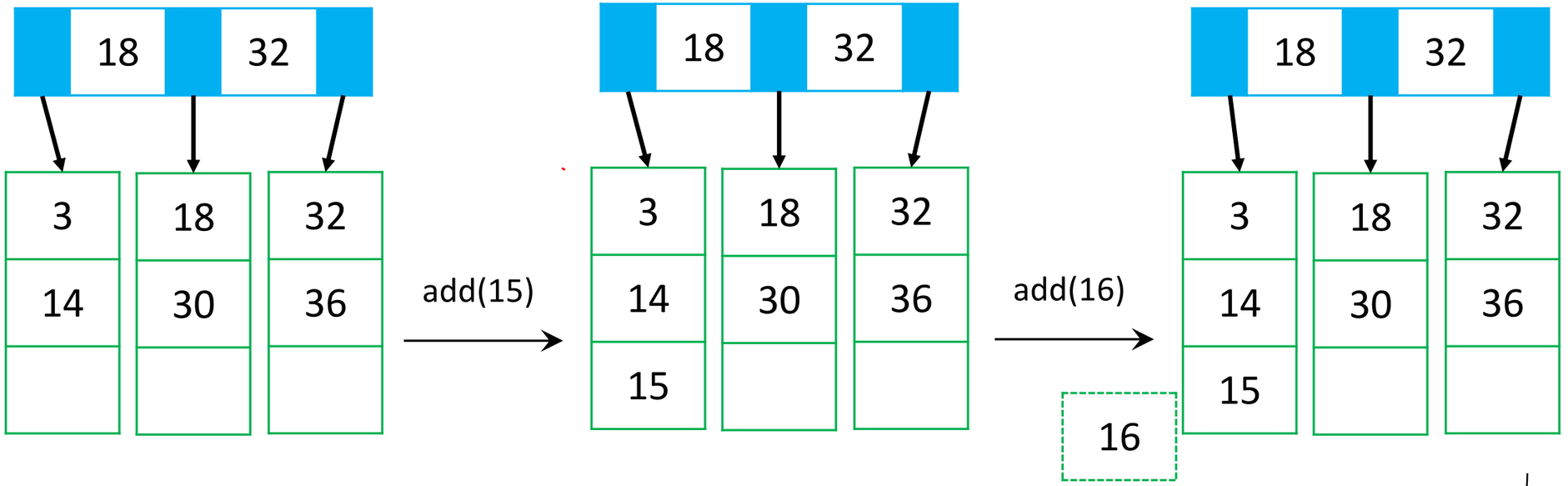
Add Example (1 of 4)



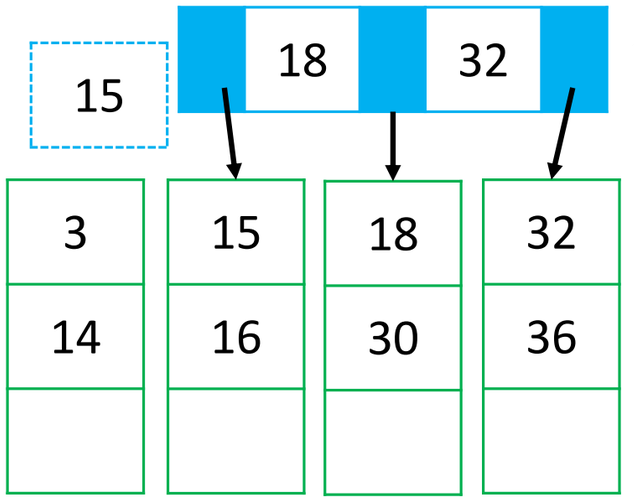
Split the leaf

Add 32, 36, 15, 16, 12, 40
M=3, L=3

Add Example (2 of 4)

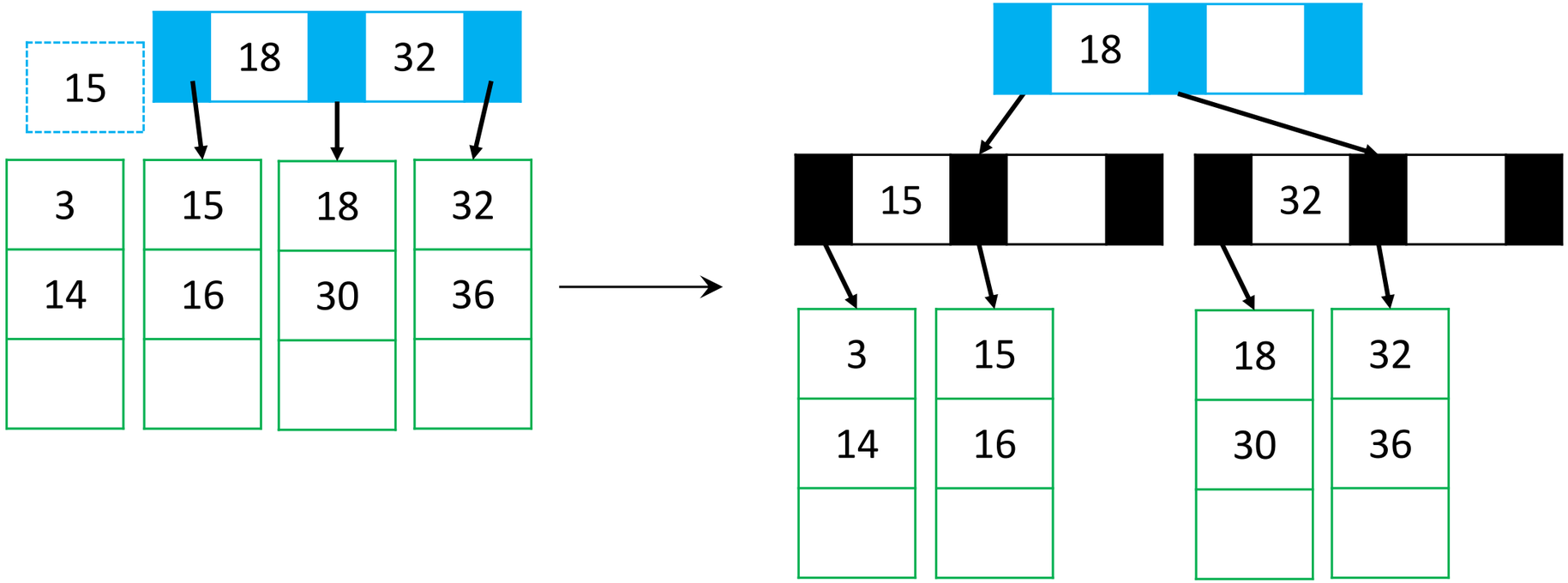


Split the leaf again, but now the parent is full!



Add ~~32, 36~~, 15, 16, 12, 40
M=3, L=3

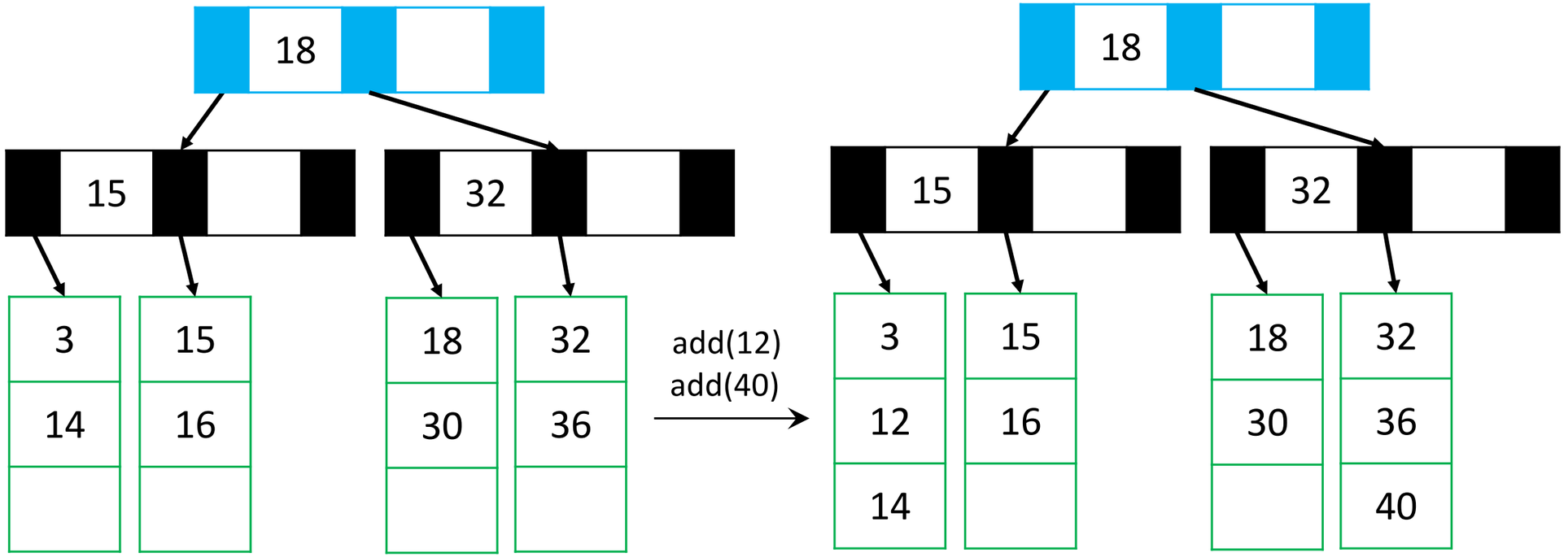
Add Example (3 of 4)



Split the parent (in this case, the root)

Add ~~32, 36, 15,~~ 16, 12, 40
M=3, L=3

Add Example (4 of 4)



Add ~~32, 36, 15, 16,~~ 12, 40
M=3, L=3

B+ Tree Add Algorithm (1 of 3)

1. Add the value to its leaf in key-sorted order

2. If the leaf now has L+1 items, *overflow:*
   - Split the leaf into two leaves:
     - Original leaf with $\lceil L/2 \rceil$ smaller items
     - New leaf with $\lfloor L/2 \rfloor = \lceil L/2 \rceil$ larger items
   - Attach the new leaf to its parent
     - Add a new key (smallest key in new leaf) to parent in sorted order

If step (2) caused the parent to have M+1 children, …

B+ Tree Add Algorithm (2 of 3)

3. If step (2) caused an **internal node** to have M+1 children
   - Split the **internal node** into two nodes
     - Original **node** with $\lceil (M+1)/2 \rceil$ smaller keys
     - New **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger keys
   - Attach the new **internal node** to its parent
     - Move the median key (smallest key in new node) to parent in sorted order
   - If step (3) caused the parent to have M+1 children, repeat step (3) on the parent

4. If step (3) caused the root to have M+1 children
   - Split the old root into two **internal nodes**, then add them to a newly-created root as described in step (3)
   - *This is the only case that increases the tree height!*

B+ Tree Add Algorithm (3 of 3)

Split the **leaf** into two leaves:
- 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 **leaf** to its parent
- Copy a new key (smallest key in new **leaf**) to the parent in sorted order

Split the **internal node** into two leaves:
- 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 **internal node** to its parent
- Move the median key (smallest key in new **node**) to the parent in sorted order

Split the **root** into two **internal nodes**:
- Left **node** with $\lceil (M+1)/2 \rceil$ smaller items
- Right **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger items

Attach the **internal nodes** to the new **root**
- Move the median key (smallest key in new right **node**) to the **root**

B+ Tree Add: Efficiency (1 of 2)

- Find correct leaf: $O(\log_2 M \log_M n)$
- Add (key, value) pair to leaf: $O(L)$
  - Why?
- Possibly split leaf: $O(L)$
  - Why?
- Possibly split parents all the way up to root: $O(M \log_M n)$
  - Why?

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

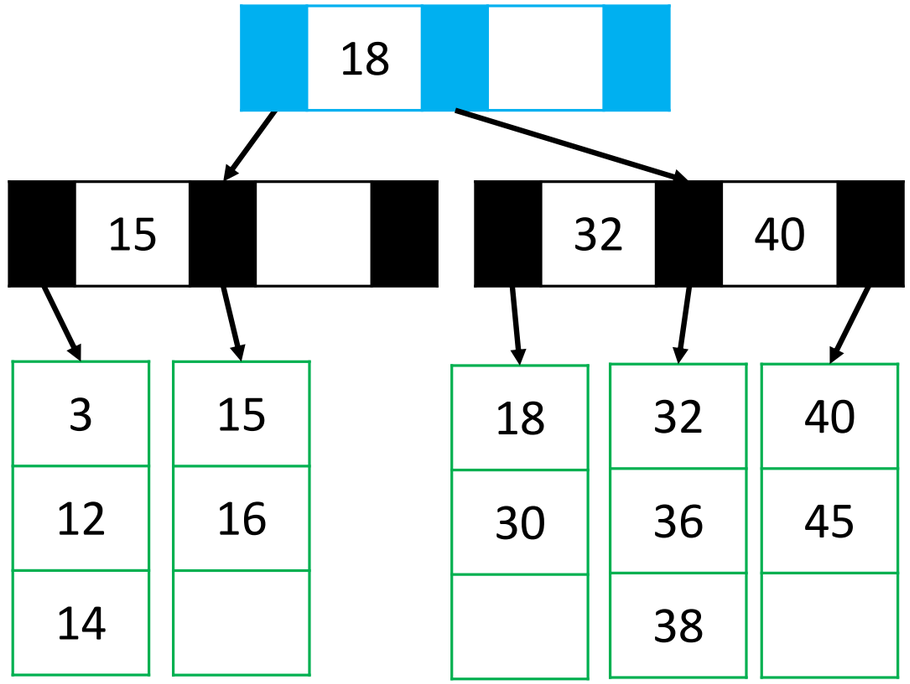
B+ Tree Add: Efficiency (2 of 2)

- Worst-case runtime is $O(L + M \log_M n)$!

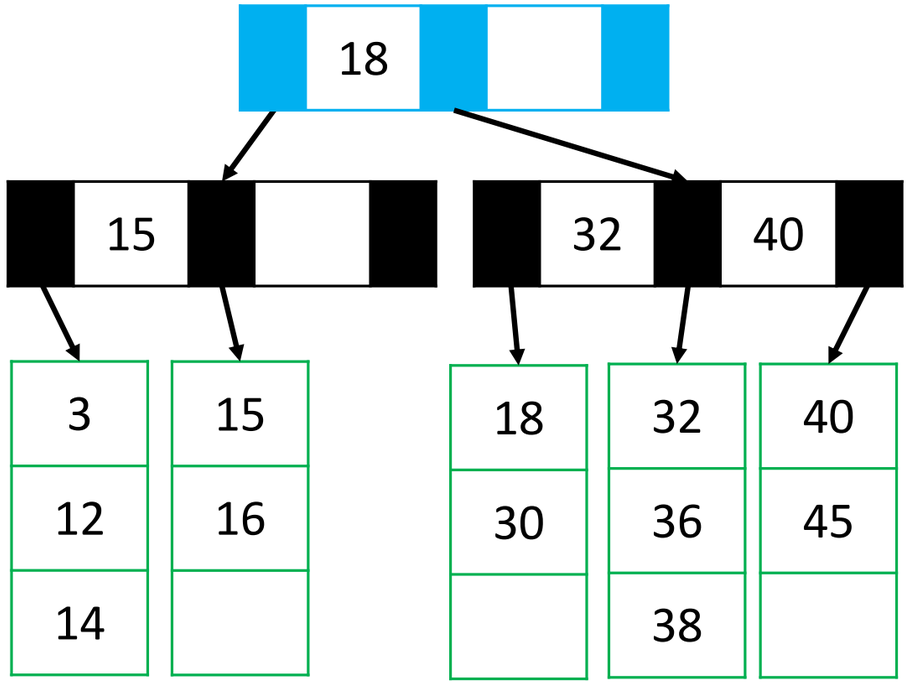
- But the worst-case isn't that common!
  - Splits are uncommon
    - Only required when a node is *full*
    - M and L are likely to be large and, after a split, nodes will be half empty
  - Splitting the root is extremely rare
  - Remember that our goal is minimizing disk accesses! Disk accesses are still bound by $O(\log_M n)$

Remove Example:

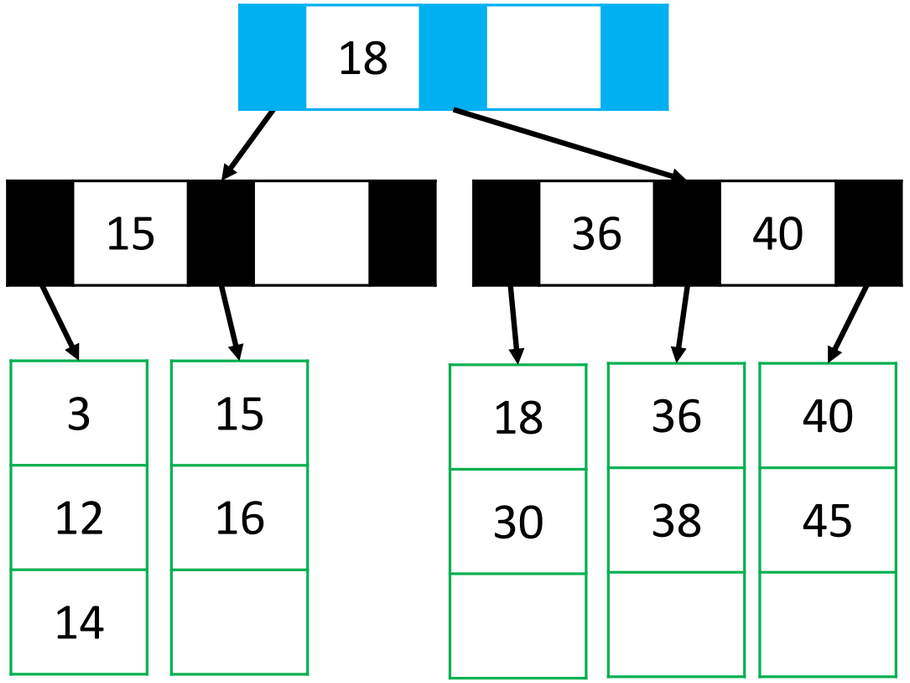
- Remove 32, 15, 16, 14, 18
- M=3, L=3
  - Min #children = 2
  - Min #items = 2



Remove Example: Answer (1 of 8)

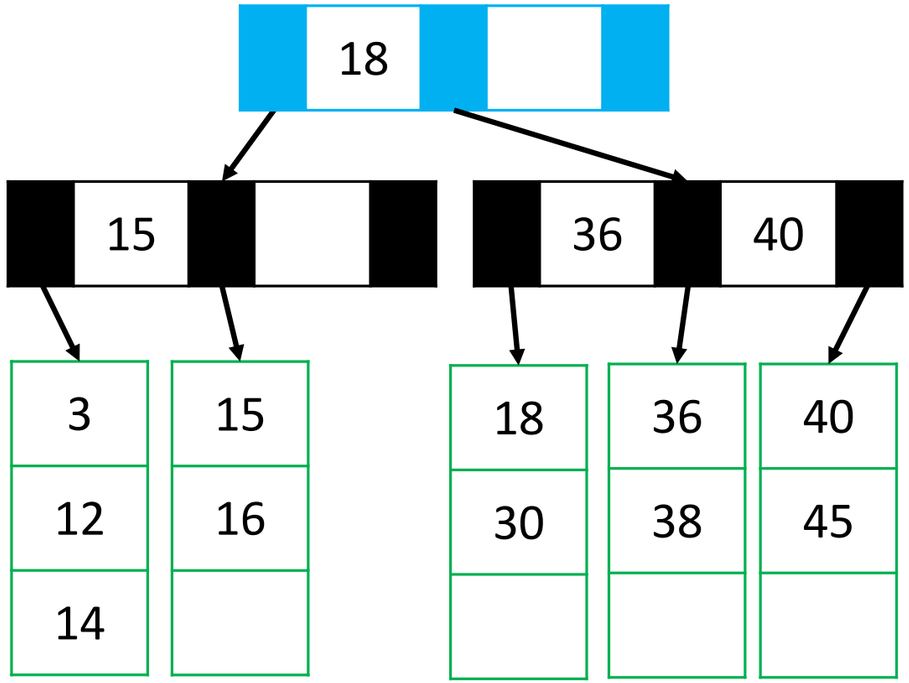


remove(32)



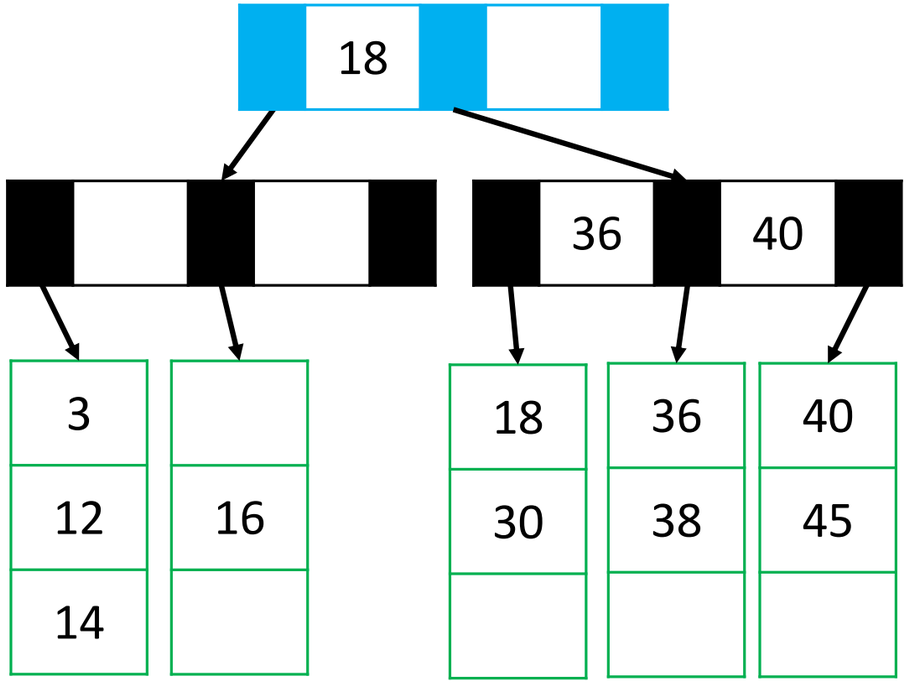
Remove 32, 15, 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (2 of 8)



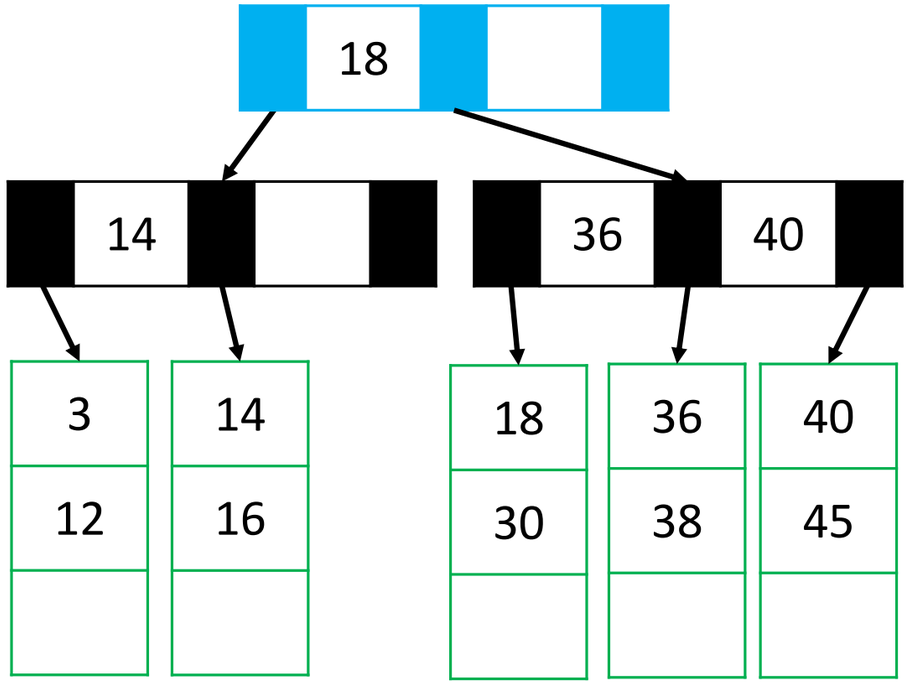
remove(15)

Adopt an item from a neighbor **leaf**



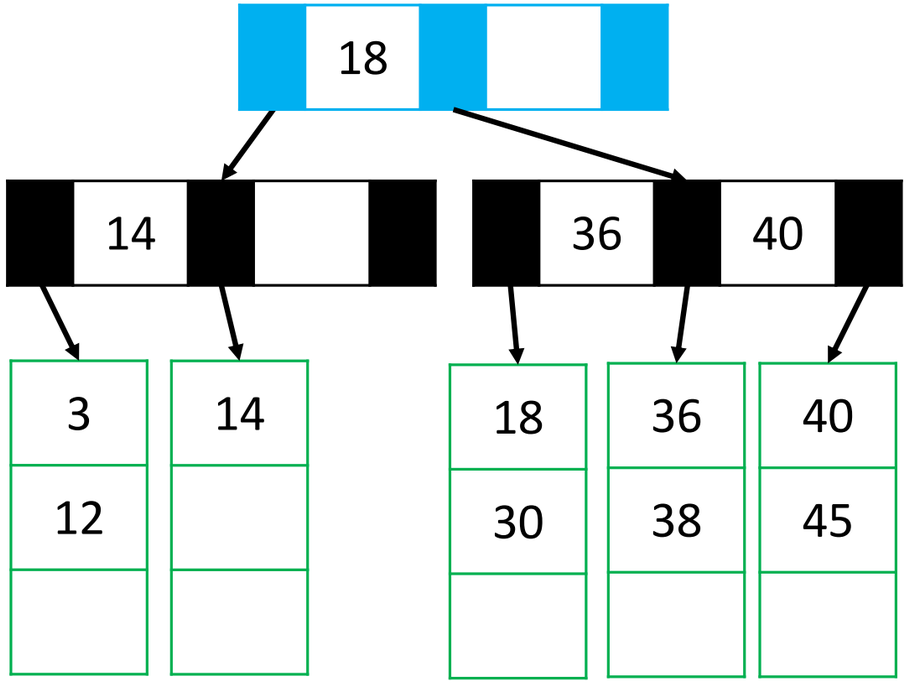
Remove ~~32,~~ 15, 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (3 of 8)



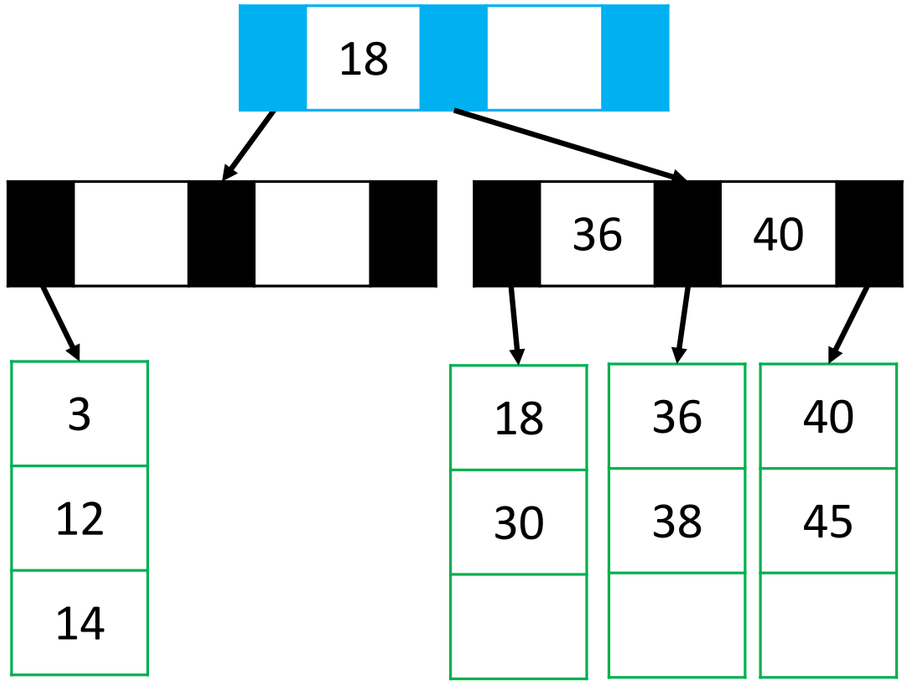
remove(16)

Merge with a neighbor **leaf**

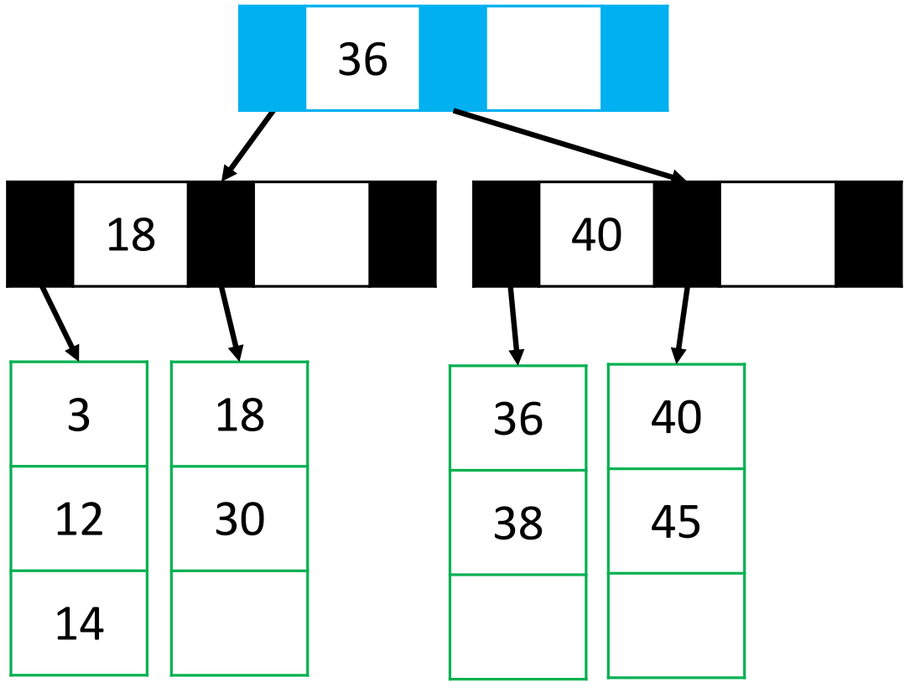


Remove ~~32, 15,~~ 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (4 of 8)



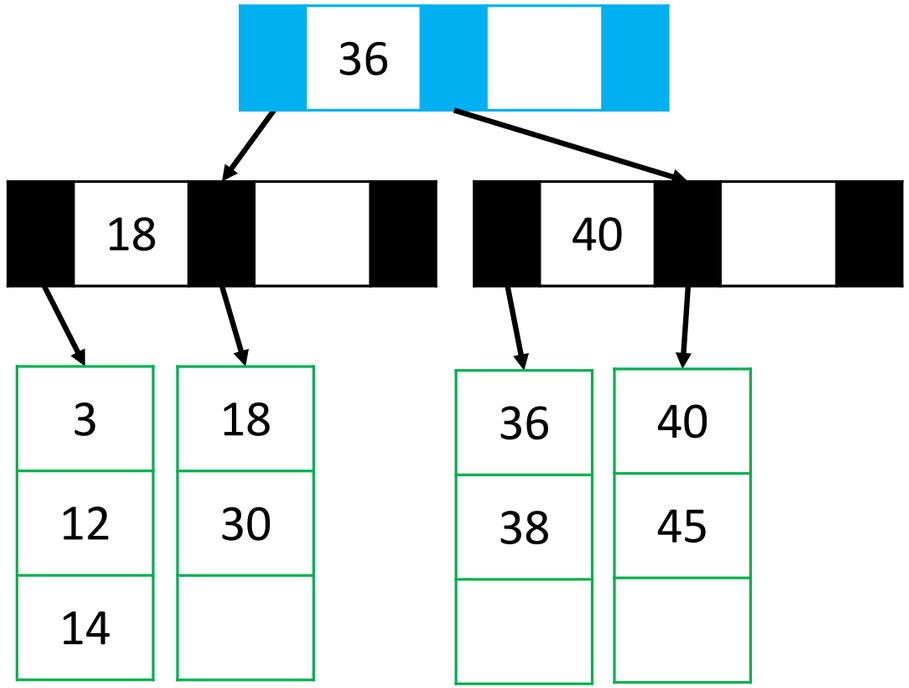
Adopt from a neighbor **node**



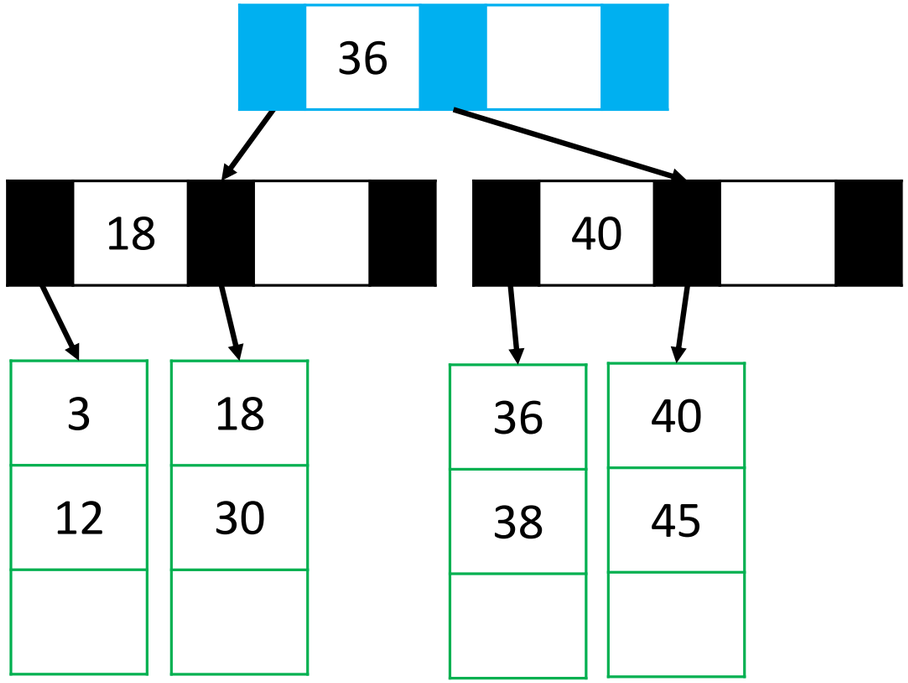
Remove ~~32, 15,~~ 16, 14, 18

M=3, L=3; min children=2, min items=2

Remove Example: Answer (5 of 8)

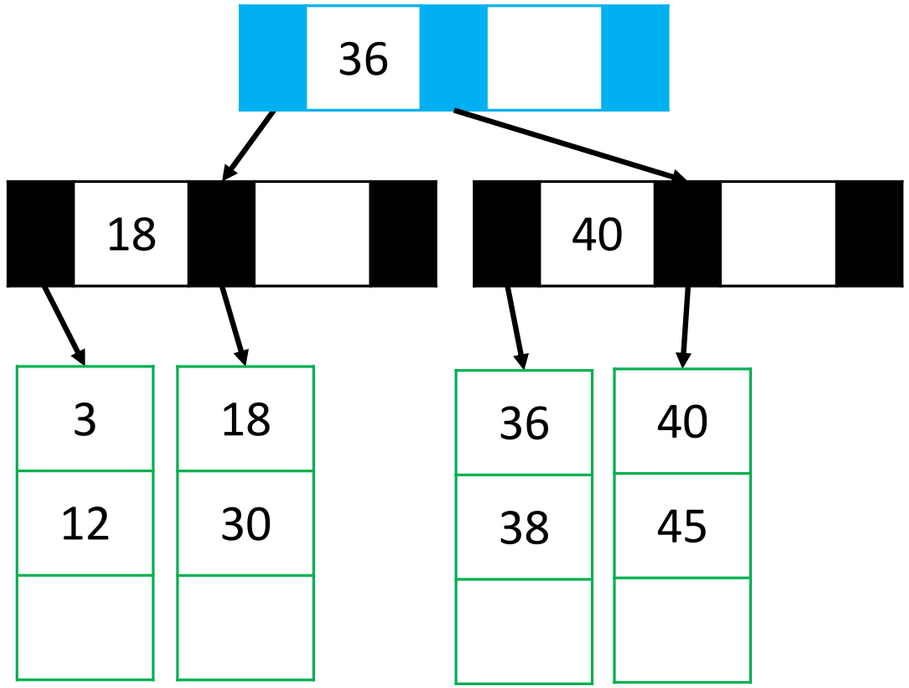


remove(14)



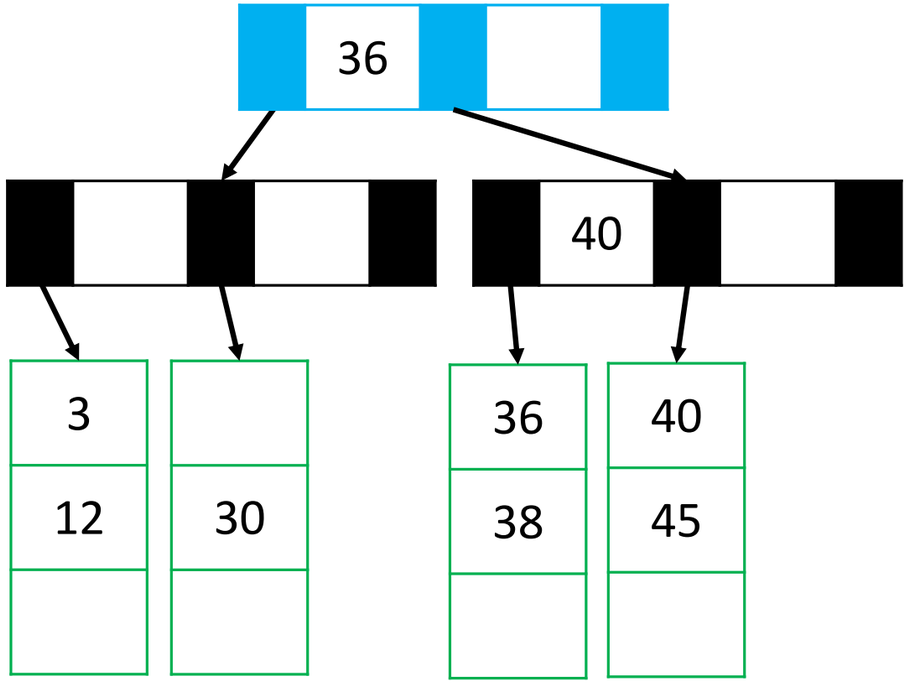
Remove ~~32, 15, 16,~~ 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (6 of 8)



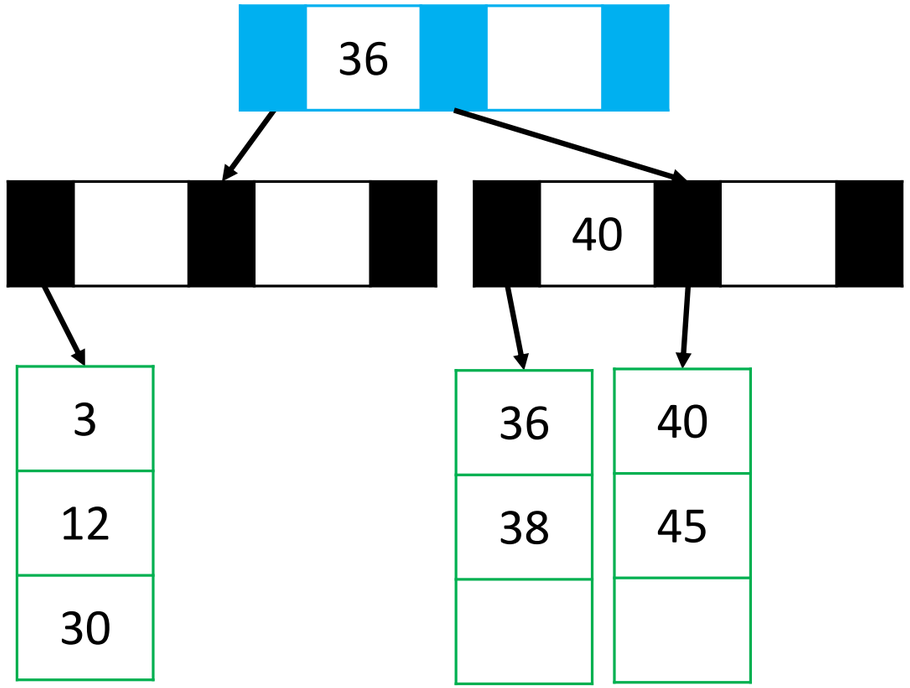
remove(18)

Merge with a neighbor **leaf**

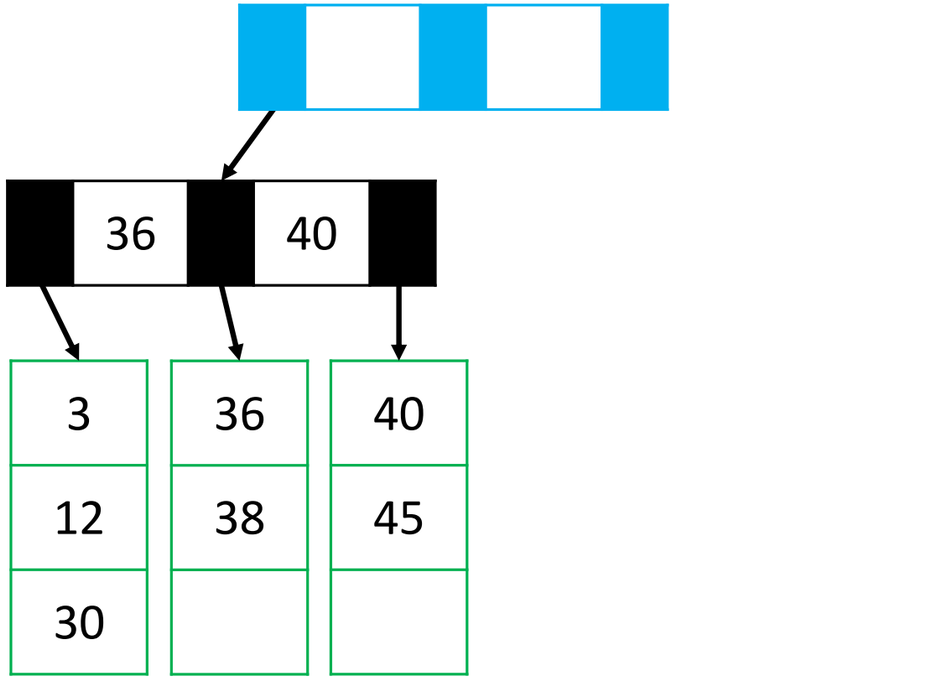


Remove ~~32, 15, 16, 14,~~ 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (7 of 8)

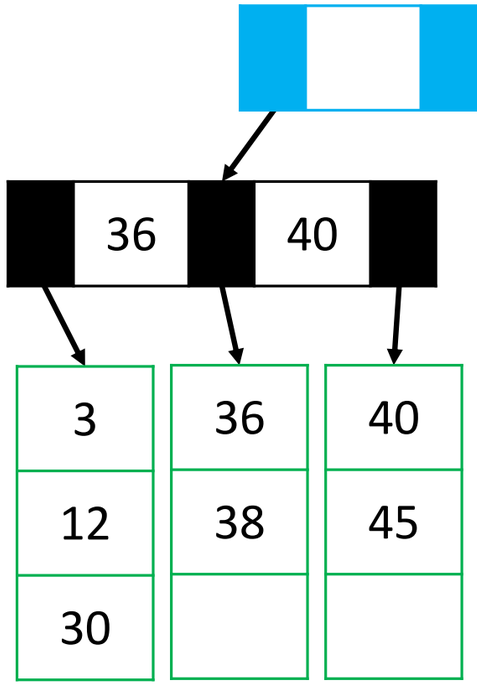


Merge with a neighbor **node**

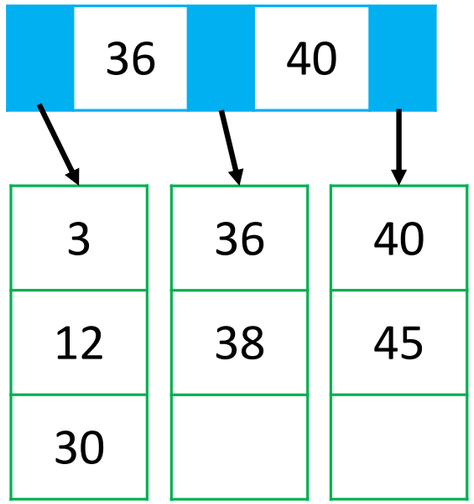


Remove ~~32, 15, 16, 14,~~ 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (8 of 8)



Delete the old **root**



Remove ~~32, 15, 16, 14,~~ 18
M=3, L=3; min children=2, min items=2

B+ Tree Remove Algorithm (1 of 3)

1. Remove the item from its leaf

2. If the leaf now has $\lceil L/2 \rceil - 1$, *underflow:*
   - If a neighbor has $> \lceil L/2 \rceil$ items, *adopt*
     - Move parent's key down, and neighbor's adjacent key up
   - Else, *merge* leaf with neighbor
     - Guaranteed to have a legal number of items
     - Remove parent's key and move grandparent's key down
     - Parent now has one less leaf

If step (2) caused the parent to have $\lceil M/2 \rceil - 1$ children, …

B+ Tree Remove Algorithm (2 of 3)

3. If step (2) caused an **internal node** to have $\lceil M/2 \rceil - 1$ children
   - If a neighbor has $> \lceil M/2 \rceil$ keys, *adopt* and update parent
     - Move parent's key down, and neighbor's adjacent key up
   - Else, *merge* with neighbor node
     - Guaranteed to have a legal number of keys
     - Remove parent's key and move grandparent's key down
     - Parent now has one less node, may need to continue up the tree

4. If step (3) caused the root to have have $\lceil M/2 \rceil - 1$ children
   - If root went from 2 children to 1 child, move key down and make the child the new root
   - *This is the only case that decreases the tree height!*

B+ Tree Remove Algorithm (3 of 3)

If a neighbor **leaf** has $> \lceil L/2 \rceil$ items, *adopt*:
- Move parent's key down, and neighbor's adjacent key up

Else *merge* **leaf** with neighbor:
- Guaranteed to have a legal number of items
- Remove parent's key and move grandparent's key down
- Parent now has one less **leaf**

If a neighbor **node** has $> \lceil M/2 \rceil$ items, *adopt*:
- Move parent's key down, and neighbor's adjacent key up

Else *merge* **node** with neighbor:
- Guaranteed to have a legal number of keys
- Remove parent's key and move grandparent's key down
- Parent now has one less **node**

B+ Tree Remove: Efficiency (1 of 2)

- Find correct leaf: $O(\log_2 M \log_M n)$
- Remove item from leaf: $O(L)$
  - Why?
- Possibly adopt from or merge with neighbor leaf: $O(L)$
  - Why?
- Possibly adopt or merge **parent node** up to root: $O(M \log_M n)$
  - Why?

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

B+ Tree Remove: Efficiency (2 of 2)

- Worst-case runtime is $O(L + M \log_M n)$!

- But the worst-case isn't that common!
  - Merges are uncommon
    - Only required when a node is *half empty*
    - M and L are likely large and, after a merge, nodes will be completely full
  - Shrinking the height by removing the root is extremely rare
  - Remember that our goal is minimizing disk accesses! Disk accesses are still bound by $O(\log_M n)$

Outline for Today

- Finish AVL
  - How to handle insertion (4 cases)
  - AVL size math (and height guarantees)
- B-Trees
  - Motivating B-Trees, a memory perspective
  - B-Tree structure
  - B-Tree methods
    - Insertion AND deletion
  - B-Tree wrap-up

B+ Trees in Java?

- For most of our data structures, we encourage writing high-level, reusable code. Eg, using Java generics in our projects

- It's a bad idea for B+ Trees, however
  - Java can do balanced trees! It can even do other B-Trees, such as the 2-3 tree (which resembles a B+ Tree with M=3)
  - Java wasn't designed for things like managing disk accesses, which is the whole point of B+ Trees
  - The key issue is Java's extra *levels of indirection*…

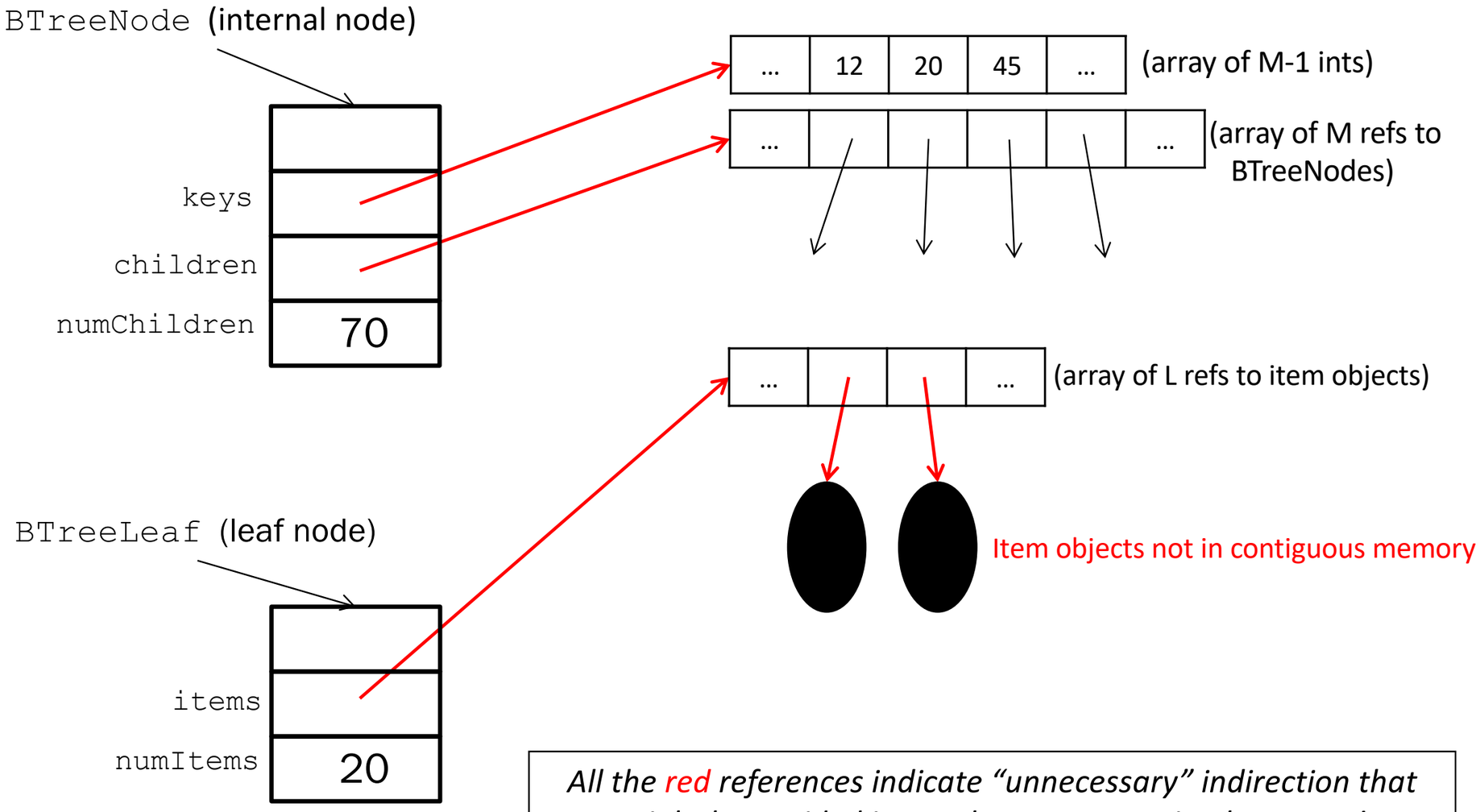
Possible Java Implementation: Code

Even if we assume **`int`** keys, Java's data representation doesn't match what we want out of a B+ Tree

```
class BTreeNode<E> {  // internal node
  static final int M = 128;
  int[]           keys        = new int[M-1];
  BTreeNode<E>[] children    = new BTreeNode[M];
  int             numChildren = 0;
  …
}

class BTreeLeaf<E> {  // leaf node
  static final int L = 32;
  int[] keys     = new int[L-1];
  E[]   items    = new Object[L];
  int   numItems = 0;
  …
}
```

Possible Java Implementation: Box-and-Arrows



All the red references indicate "unnecessary" indirection that might be avoided in another programming language!

B+ Trees in Java: The Moral of the Story

- The whole idea behind B+ trees was to keep related data in contiguous memory

- But this runs counter to the code and patterns Java encourages
  - Java's implementation of generic, reusable code is not want you want for your performance-critical web-index

- Other languages (e.g., C++) have better support for "flattening objects into arrays" in a generic, reusable way

- 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