B+ Trees
CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

- Thank you SO MUCH for your patience this past week

- Exercise 6 out today: VerifyAVL
  - Due Sunday, 11:59 PM (released late)
  - Ex 7 & 8 will come out on time this weekend, they are on the easier side.

- Midterm coming next week!
  - Reminder: Non-traditional midterm

Lecture Outline

- **Recap**

- B+ Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Find
  - B+ Tree Add
  - B+ Tree Remove

Recap of weekend Lecture

- Our data structures so far have assumed O(1) time for basic operations, reads and writes
- When our data structures are big enough, reads and writes may trigger a disk load (takes a LONG time)
- To mitigate this, we rely on locality
  - Spatial Locality
  - Temporal Locality
- We want a data structure that is specifically designed to take full advantage of locality and minimize disk accesses

Lecture Outline

- Recap
- B+ Trees
  - **Goals and Design**
  - B+ Tree Structure
  - B+ Tree Find
  - B+ Tree Add
  - B+ Tree Remove

Goal of the B+ Tree

- **Problem**: A dictionary with so many items *most of it is on disk*

- **Goal**: A balanced tree (logarithmic height) that minimizes disk accesses

- **Concept**: Increase the branching factor of our tree
  - Minimize number of nodes to traverse

- **Disclaimer:** You will not have to implement this structure!!
  - Requires more control over memory than Java allows

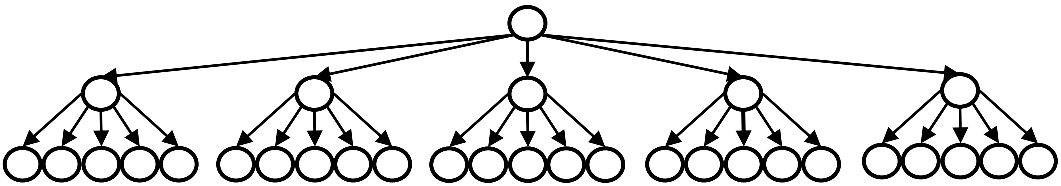
How do we minimize disk accesses?

- Increase size of each node in our tree
  - … to the size of a full disk block
  - For a dictionary, this would mean many key/value pairs and pointers per node

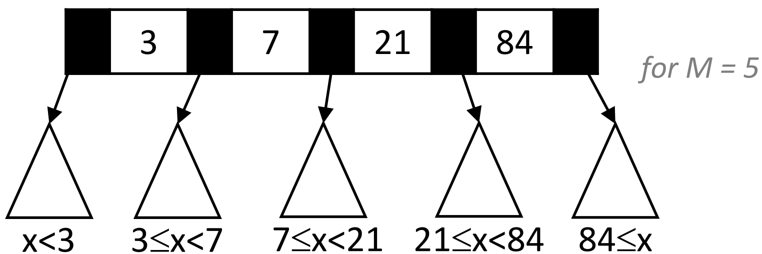
- Worst case number of nodes that we look at for a find in any tree will be bounded by height
  - So let's try to maximize how efficiently we use the height
  - Higher branching factor than 2

Increasing efficiency: M-ary Search Tree

- A search tree with branching factor M (instead of 2)
  - Each node has a key-sorted array of M children: `Node[]`



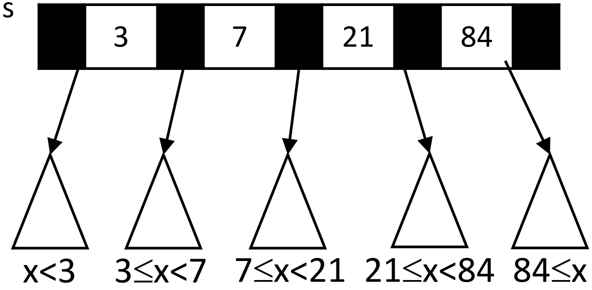
  - Ordering property similar to BST



  - M-1 keys define the M subtrees (ie, ranges) that we search through
- Choose M such that the node size = disk block size

M-ary Performance?

- Runtime for `find` = NumHops * WorkPerHop
  - **Balanced** tree height is: $\log_M n$ (M-ary) vs $\log_2 n$ (binary)
    - Eg: M = 256 (=2^8) and n = 2^{40}, M-ary makes 5 hops vs binary makes 40 hops
  - For each internal node, how to decide which child to take?
    - Binary: Less than vs greater than node's single key? 1 comparison
    - M-ary: In range 1? In range 2? In range 3?... In range M?
      - Linear search the Node[]: M comparisons
      - Binary search the Node[]: $\log_2 n$ comparisons



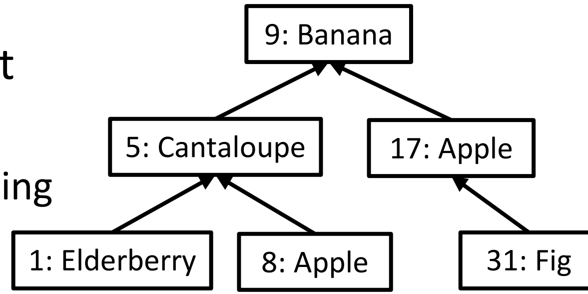
- Runtime for M-ary `find`:
  - $O(\log_2 M \log_M n)$

- Note: a "hop" here means following a pointer to another node

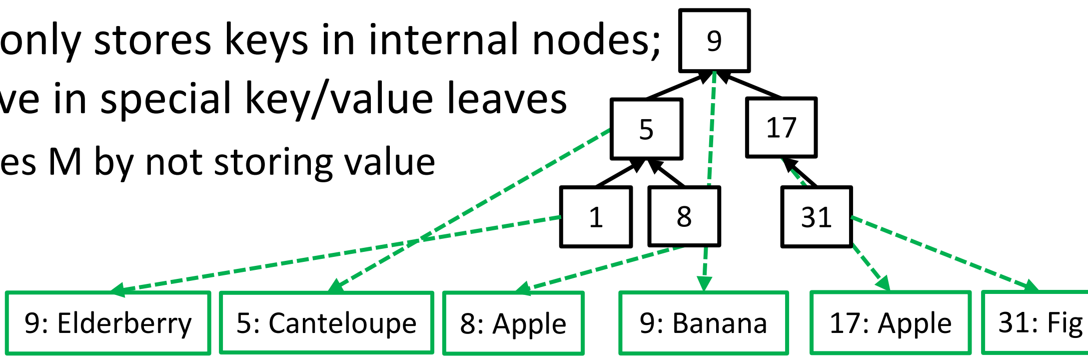
Design Decision: Key-only Internal Nodes

- A Dictionary ADT stores key->value pairs; where should we store a key's value?

- BST stores value alongside the key at every node
  - Loads entire node even if we are "passing through" to find a different key



- B+ Tree only stores keys in internal nodes; values live in special key/value leaves
  - Increases M by not storing value



Lecture Outline

- Recap

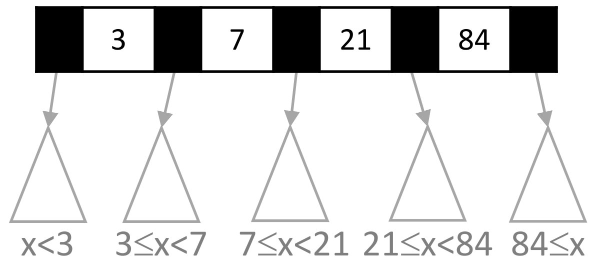
- B+ Trees
  - Goals and Design
  - **B+ Tree Structure**
  - B+ Tree Find
  - B+ Tree Add
  - B+ Tree Remove

B+ Tree Node Structure

Both the textbook and we refer to "B+ Trees" as "B-Trees", but "B-Trees" actually encompass several variants

- Two node types: **internal** and **leaf**

- Each **internal node** contains up to *M-1* keys (for up to *M* children)
  - Does not store values, only keys
  - Function as "signposts"



- Each **leaf node** contains up to *L* items
  - Stores (key, value) pairs
  - As usual, we'll ignore the "along for the ride" value in our examples

3	"cat"
7	"apple"
21	"purple"
84	"ideas"

B+ Tree Parameters

- Two parameters, one for each type of node:
  - ***M*** = # of children in an **internal** node
    - The ranges are defined by M-1 keys
  - ***L*** = # of items in a **leaf** node

k_1	k_2	…	k_{m-1}	
ptr_1	ptr_2	…	ptr_{m-1}	ptr_m

(sorted by key)

- Picking *M* and *L* based on disk-block size maximizes B+ Tree's efficiency
  - Recommend M* ≈ diskBlockSize/keySize
  - Recommend L = diskBlockSize/(keySize + valueSize)
  - In practice, ***M*** ≫ ***L***
    - Since typically sizeof(key) ≫ sizeof(keyvaluepair)

k_1	v_1
k_2	v_2
…	…
k_L	v_L

(sorted by key)

** More precisely, we recommend M = (diskBlockSize + keySize)/(keySize + pointerSize)*

B+ Tree Structure

- **Internal nodes**
  - Have between $\lceil M/2 \rceil$ and M children; i.e., at least half full
  - *Reminder: no values, just keys*
- **Leaf nodes**
  - All leaves at the same depth
  - Have between $\lceil L/2 \rceil$ and L items; i.e., at least half full
  - *Reminder: keys **and** values*
- **Root node** – A Special Case!
  - If tree has $\leq L$ items, root is a **leaf node**
    - Unusual; only occurs when starting up
  - Else, root is an **internal node** and has between 2 and M children
    - i.e., the "at least half full" condition does not apply

B+ Trees are Balanced (Enough)

- Not hard to show height *h* is logarithmic in number of items *n*
  - Let $M > 2$ (if $M = 2$, then a "linked list tree" is legal – no good!)
  - Because all nodes are at least half full *(except possibly the root)* and all leaves are at the same level, the minimum number of items *n* for a height *h>0* tree is

$$n \geq \underbrace{2\lceil M/2 \rceil^{h-1}}_{\text{minimum number of leaves}} \underbrace{\lceil L/2 \rceil}_{\text{minimum items per leaf}}$$

B+ Trees are Shallower than AVL Trees

- Suppose we have 100,000,000 items

- Maximum height of AVL tree?
  - Recall S(h) = 1 + S(h-1) + S(h-2)
  - h = **37**

- Maximum height of B+ Tree with *M*=128 and *L*=64?
  - Recall $n \geq 2\lceil M/2 \rceil^{\mathbf{h-1}} \lceil L/2 \rceil$
  - h = **5**

B+ Trees are Disk Friendly (1 of 2)

- Reduces number of disk accesses during `find`
  - Large *M* = shallower tree = potentially fewer accesses
  - Requires that ***we pick M wisely***
    - Too large: multiple disk accesses to load a single **internal** node
    - Too small: tree could've been shallower
  - Time for binary search over *M*-1 keys insignificant compared to disk access

- Reduces unnecessary data transferred from disk
  - `find` wants *one value*; doesn't load "incorrect" values into memory
  - Only one disk access to bring (the single correct) value into memory: when we find the correct **leaf node**

B+ Trees are Disk Friendly (2 of 2)

- Maximizes temporal locality
  - B+ Tree-style **internal** nodes are used more often (they differentiate between a larger fraction of keys) than BST-style nodes, and therefore are more likely to be held in memory by the OS

Lecture Outline

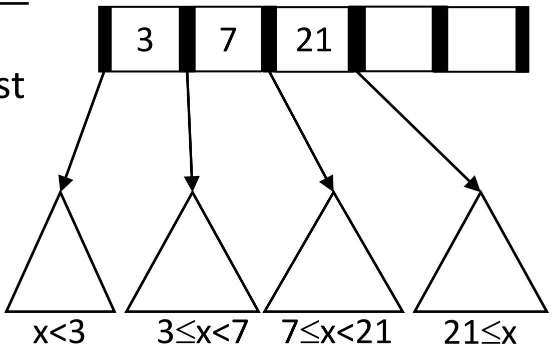
- Recap

- B+ Trees
  - Goals and Design
  - B+ Tree Structure
  - **B+ Tree Find**
  - B+ Tree Add
  - B+ Tree Remove

B+ Tree Find/Contains

- M-way extension of a BST's root-to-leaf recursive algorithm
  - At each **internal** node, do binary search on (up to) *M*-1 keys to determine which branch to take
  - At the **leaf** node, do binary search on the (up to) *L* items
  - *Requires that keys are sorted in both* ***internal*** *and* ***leaf*** *nodes!*

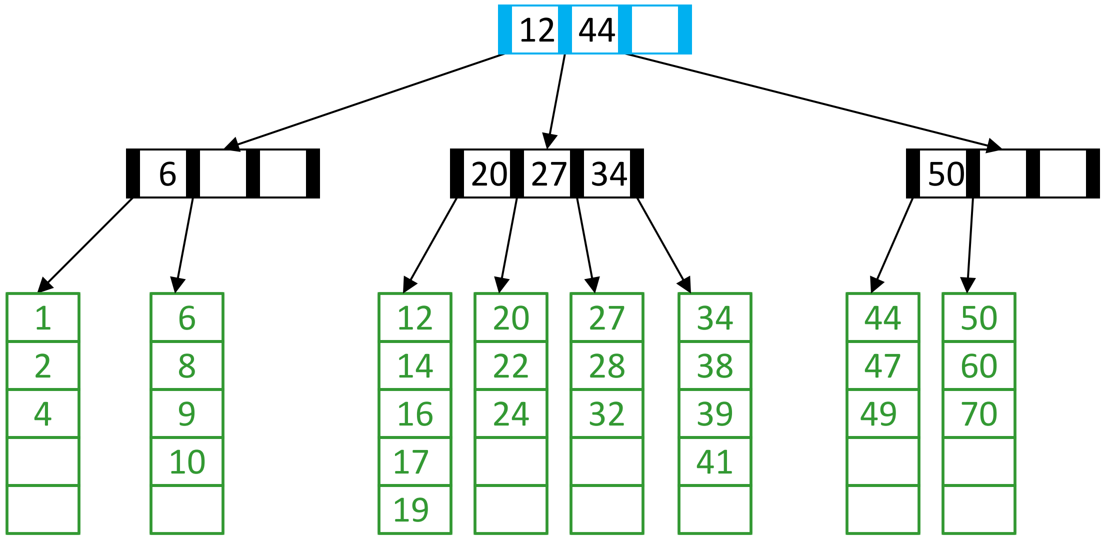
- Difference:
  - Since we *don't store value at internal nodes*, we will never find our value in the internal nodes; must always traverse to the bottom of B+ Tree



Find/Contains Example

Notation:
- Internal nodes drawn horizontally
- Leaf nodes drawn vertically
- All nodes include empty cells

- Tree with M=4 (max # pointers in **internal node**) and L=5 (max # items in **leaf node**)
  - All **internal nodes** must have ≥2 children
  - All **leaf nodes** must have ≥3 items (but we are only showing keys)



Lecture Outline

- Recap

- B+ Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Find
  - **B+ Tree Add**
  - B+ Tree Remove

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 $\lceil (L+1)/2 \rceil$ smaller items
     - New **leaf** with $\lfloor (L+1)/2 \rfloor = \lceil L/2 \rceil$ larger items
   - Attach the new **leaf** to its parent
     - 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)

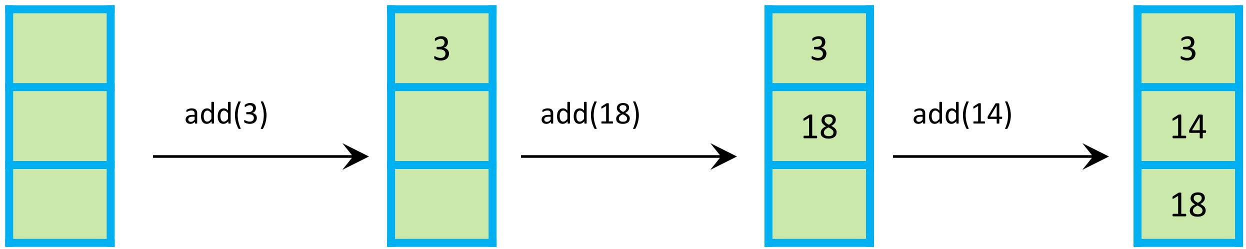
3. If step (2) caused an **internal node** to have M+1 children
   - Split the **internal node** into two nodes
     - Original **node** with $\lceil (M+1)/2 \rceil$ smaller keys
     - New **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger keys
   - Attach the new **internal node** to 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!*

Add Example:

- Add 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45, 38
- M=3, L=3

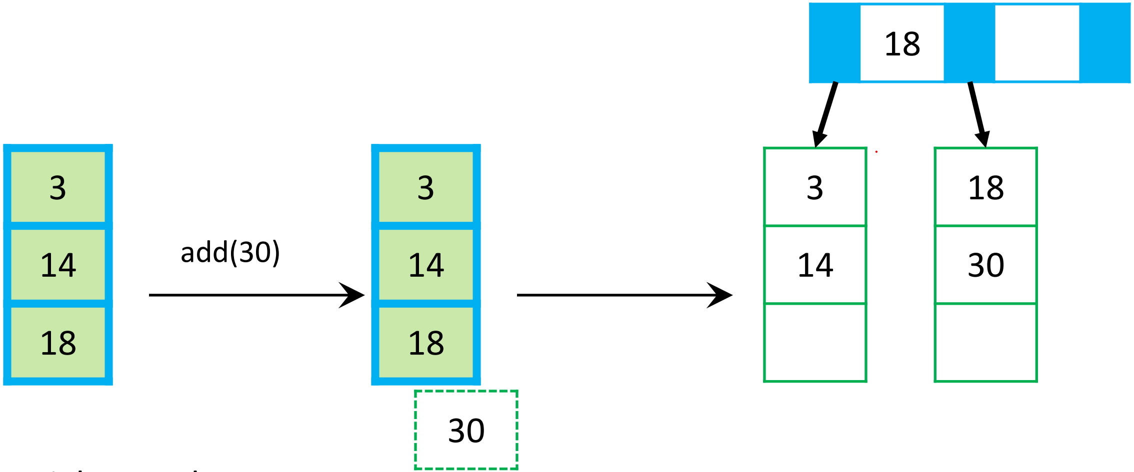
Add Example: Answer (1 of 7)



Special case: the
root is a **leaf node**

Add 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45, 38
M=3, L=3

Add Example: Answer (2 of 7)

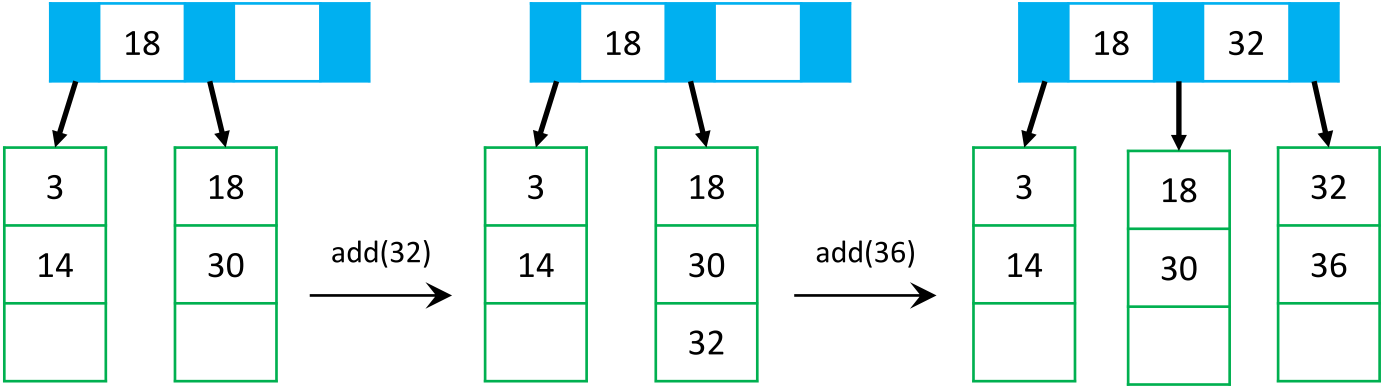


Special case: the **root** is a **leaf node**

Add ~~3, 18, 14,~~ 30, 32, 36, 15, 16, 12, 40, 45, 38
M=3, L=3

- When we "overflow" a leaf, it is split and the parent gains another key (to select between the two leaves)
- Parent's new key is the smallest element in the <u>right</u> child
- If there is no parent, create one

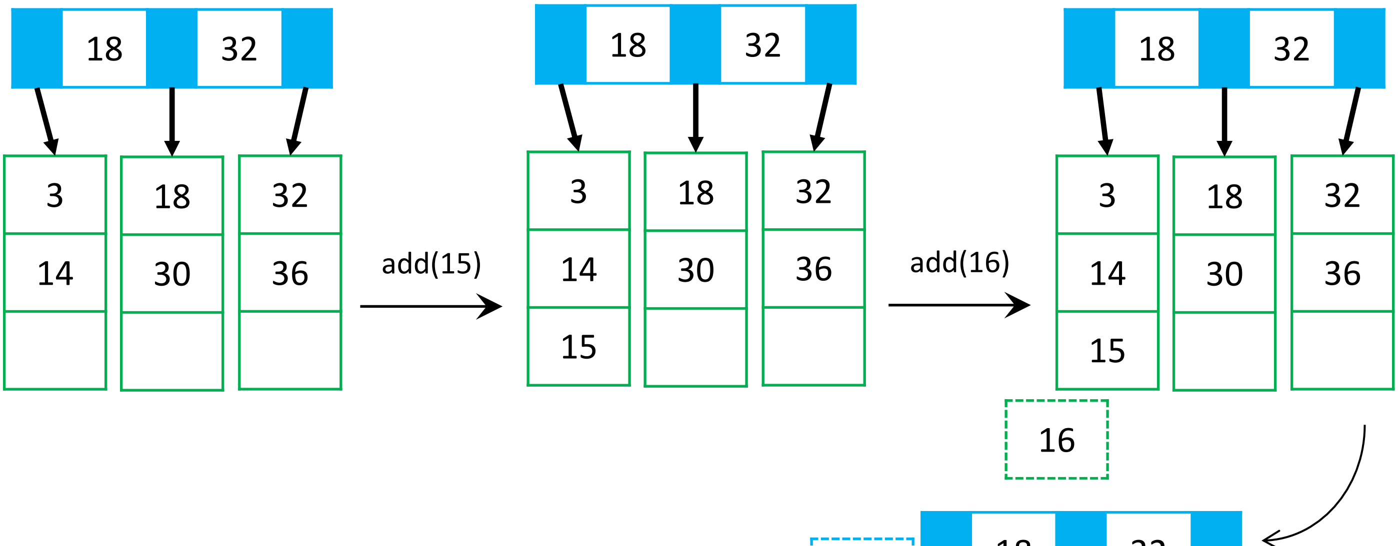
Add Example: Answer (3 of 7)



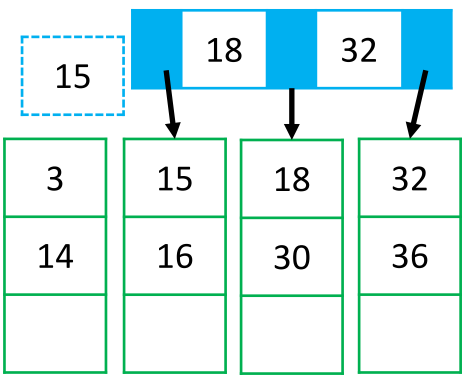
Split the leaf again

Add ~~3, 18, 14, 30,~~ 32, 36, 15, 16, 12, 40, 45, 38
M=3, L=3

Add Example: Answer (4 of 7)

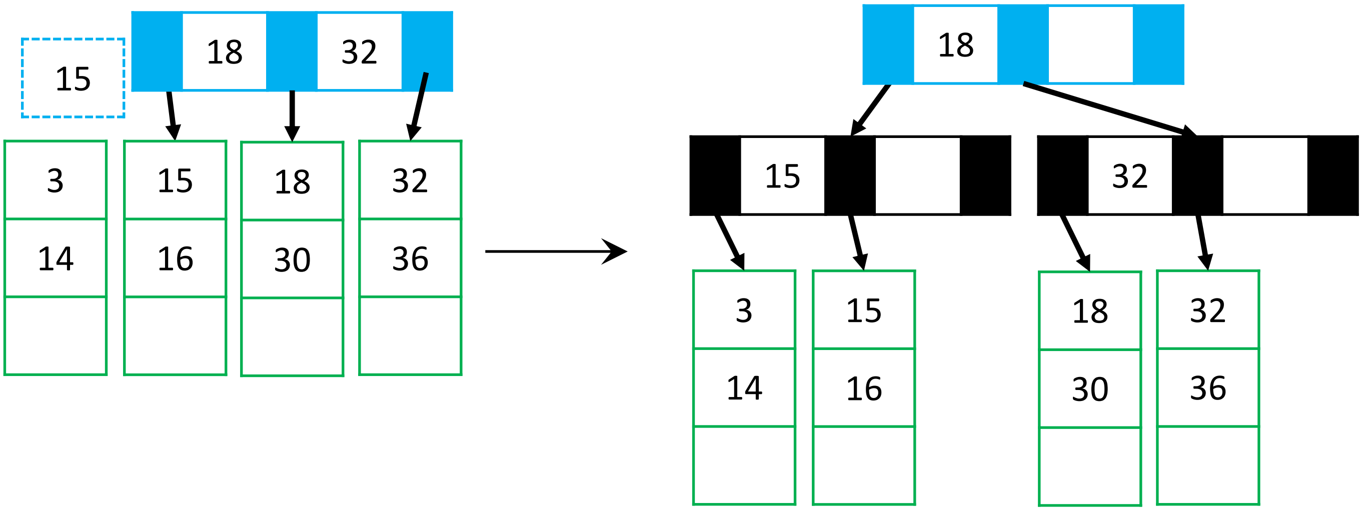


Split the leaf again, but now the parent is full!



Add ~~3, 18, 14, 30, 32, 36,~~ 15, 16, 12, 40, 45, 38
M=3, L=3

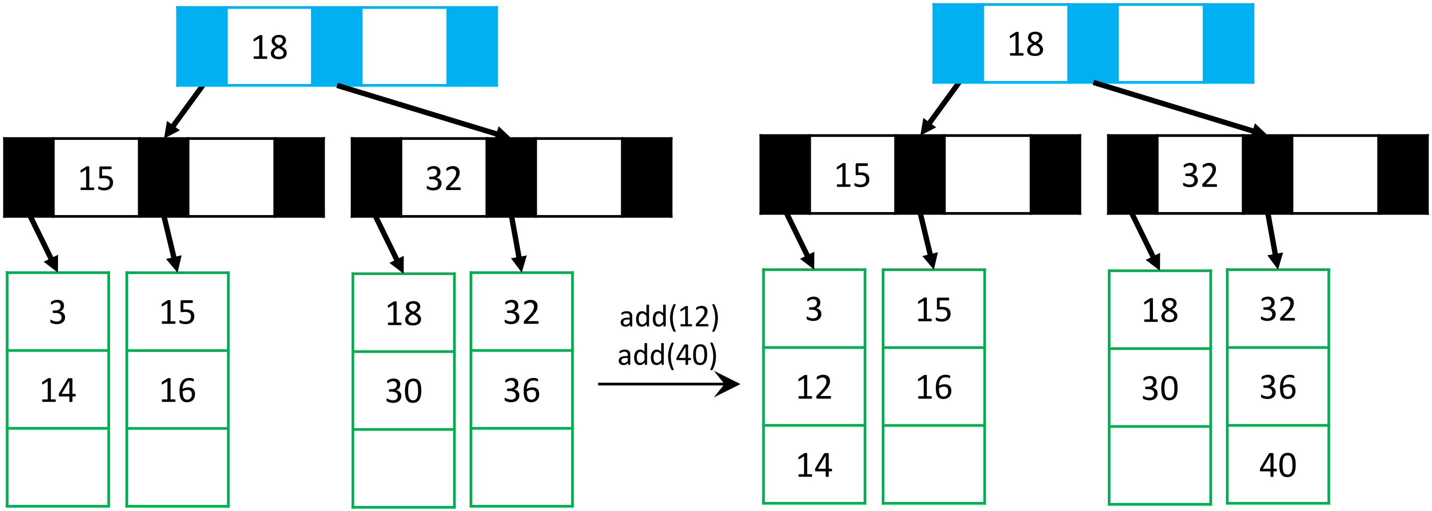
Add Example: Answer (5 of 7)



Split the parent (in this case, the root).
Note that the median key **moves** into the parent (vs being copied)

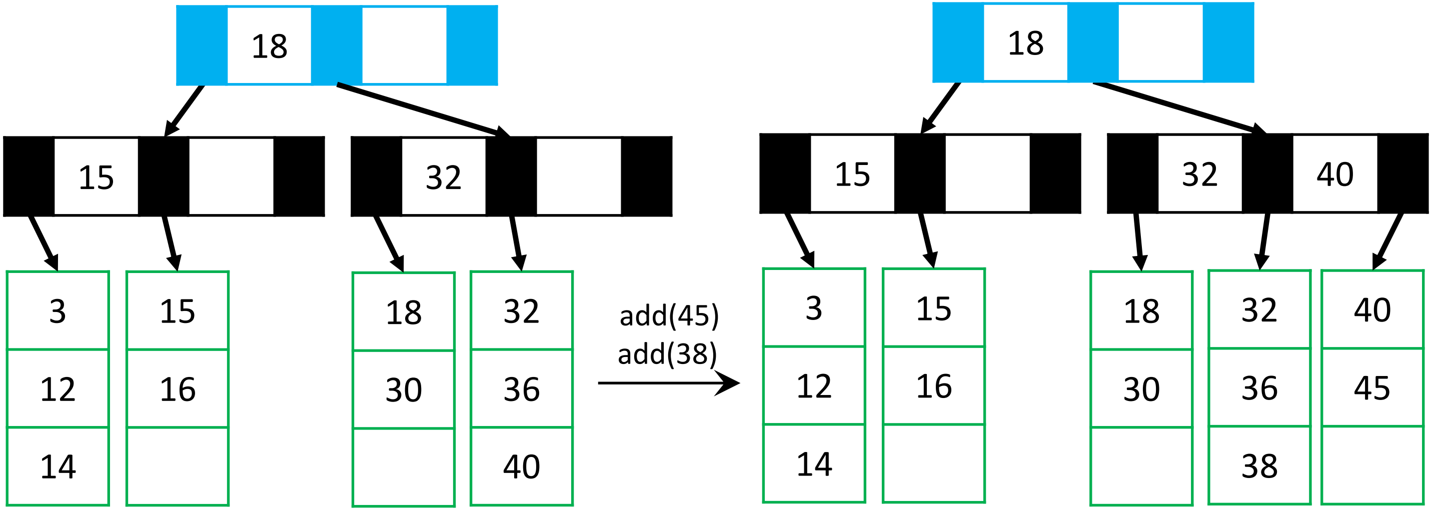
Add ~~3, 18, 14, 30, 32, 36, 15,~~ 16, 12, 40, 45, 38
M=3, L=3

Add Example: Answer (6 of 7)



Add ~~3, 18, 14, 30, 32, 36, 15, 16,~~ 12, 40, 45, 38
M=3, L=3

Add Example: Answer (7 of 7)



Split the leaf again

Add ~~3, 18, 14, 30, 32, 36, 15, 16, 12, 40,~~ 45, 38
M=3, L=3

B+ Tree Add Algorithm (3 of 3)

- Note the similarities between the overflow steps:

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
- Add 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

- But also the difference when overflowing a root:

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**

gradescope

gradescope.com/courses/275833

- When splitting nodes in a B+ Tree, why do we need to *copy* keys out of leaves but *move* keys out of internal nodes?

B+ Tree Add: Efficiency (1 of 2)

- Find correct **leaf**: $O(\mathtt{log}_2 M \mathtt{log}_M n)$
- Add (key, value) pair to **leaf**: $O(L)$
  - Why?
- Possibly split **leaf**: $O(L)$
  - Why?
- Possibly split parents all the way up to **root**: $O(M \mathtt{log}_M n)$
  - Why?

- Total: $O(L + M \mathtt{log}_M n)$

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 $O(\log_M n)$

Lecture Outline

- Recap

- B+ Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Find
  - B+ Tree Add
  - **B+ Tree Remove**

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!*

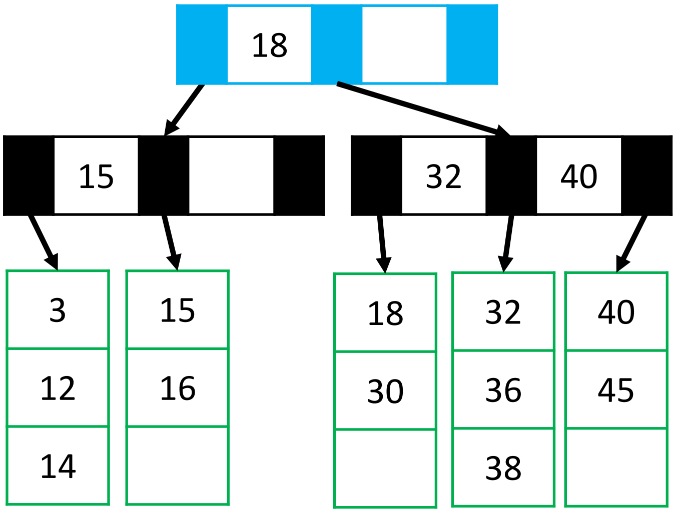
B+ Tree Remove Algorithm (3 of 3)

- Again, note the similarities between the underflow steps:

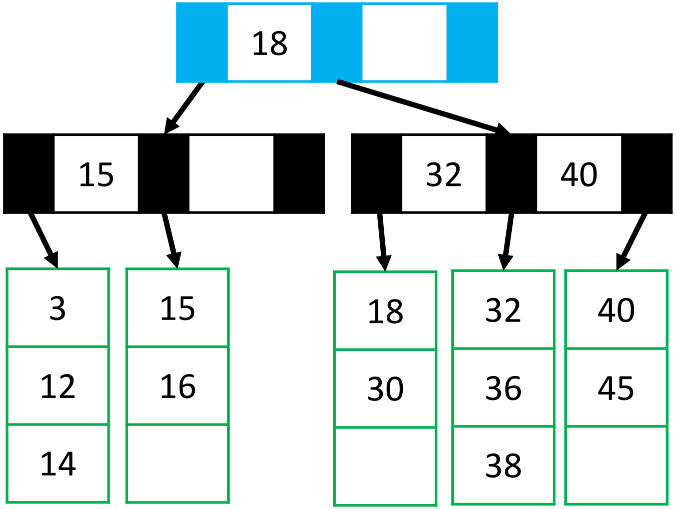
If a neighbor **leaf** has > $\lceil L/2 \rceil$ items, *adopt*: Move parent's key down, and neighbor's adjacent key up Else *merge* **leaf** with neighbor: Guaranteed to have a legal number of items Remove parent's key and move grandparent's key down Parent now has one less **leaf**	If a neighbor **node**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

Remove Example

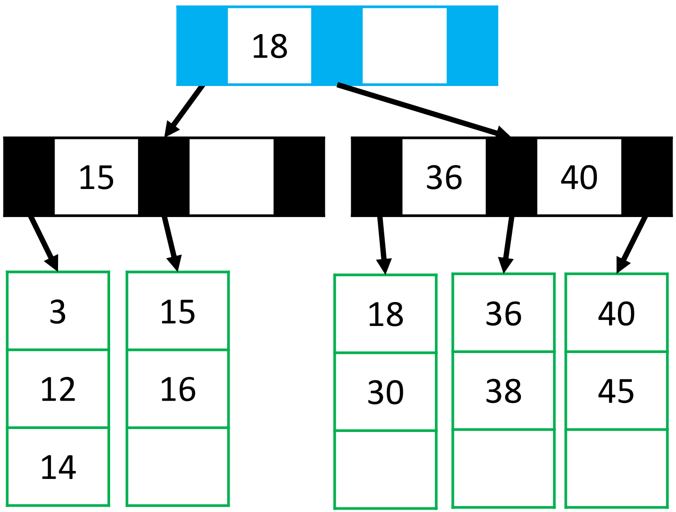
- Remove 32, 15, 16, 14, 18
- M=3, L=3
  - Min #children = 2
  - Min #items = 2
- Gradescope question:
  - How many nodes do we end with?



Remove Example: Answer (1 of 8)

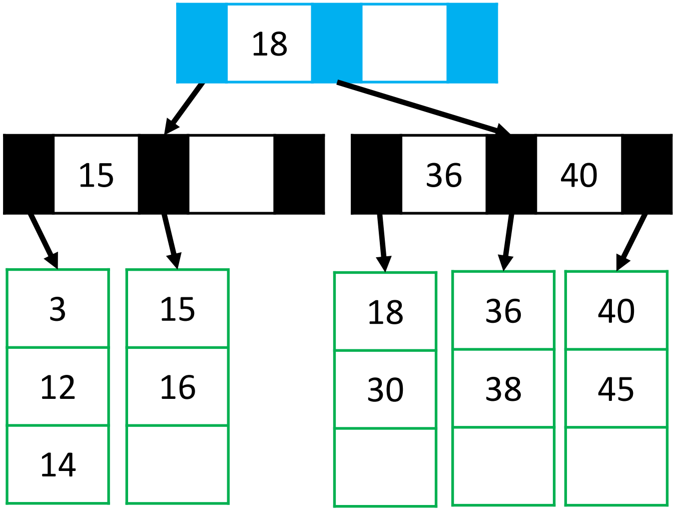


remove(32)



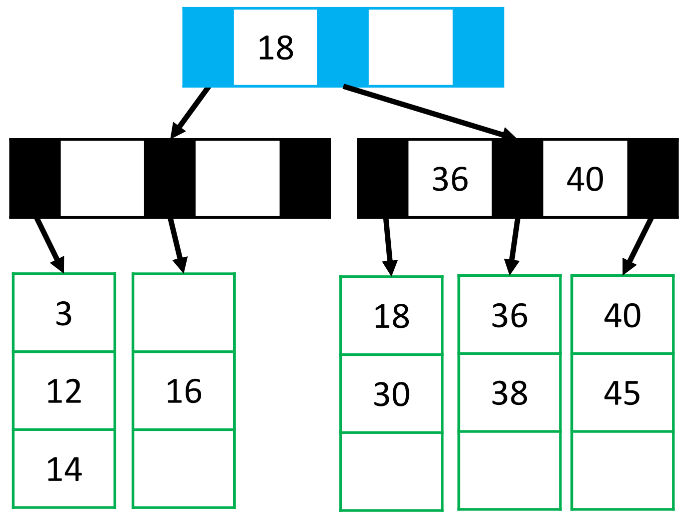
Remove 32, 15, 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (2 of 8)



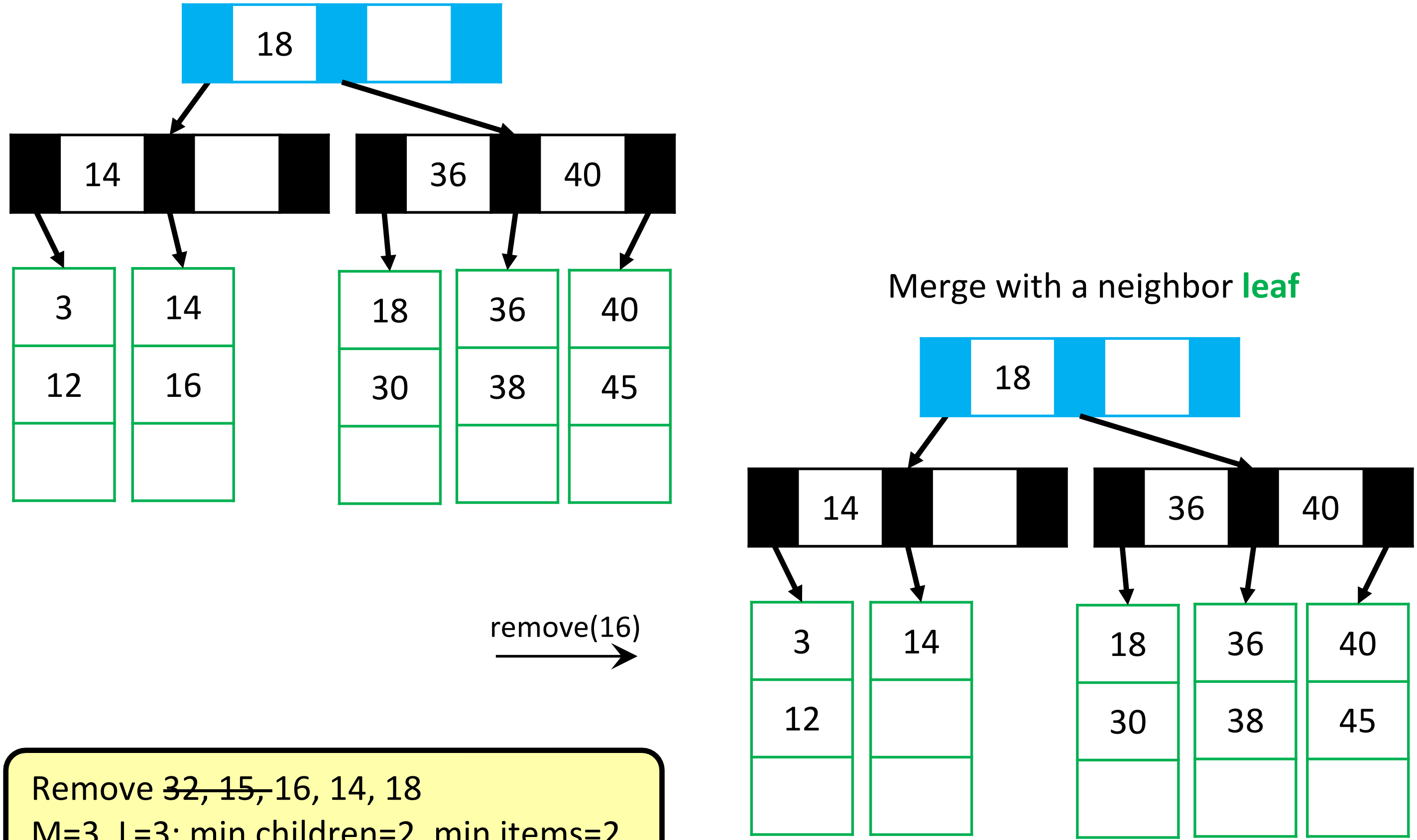
Adopt an item from a neighbor **leaf**

remove(15)



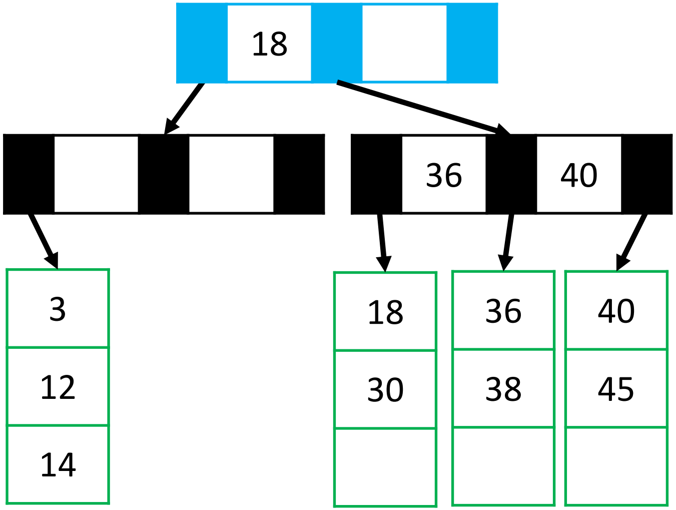
Remove ~~32,~~ 15, 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (3 of 8)

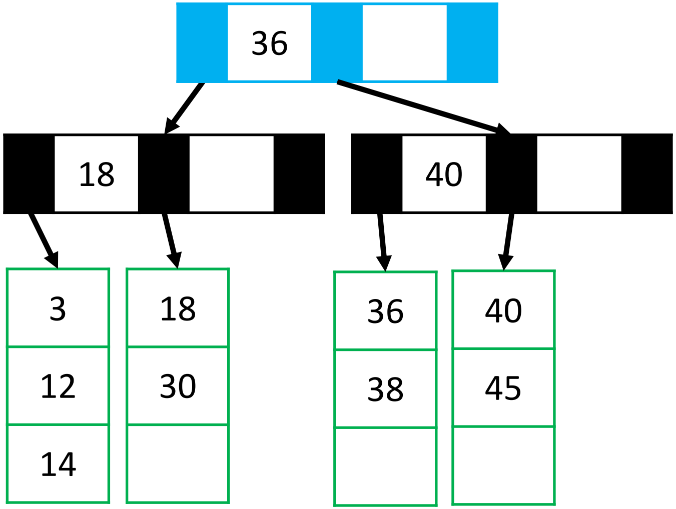


Remove ~~32, 15,~~ 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (4 of 8)

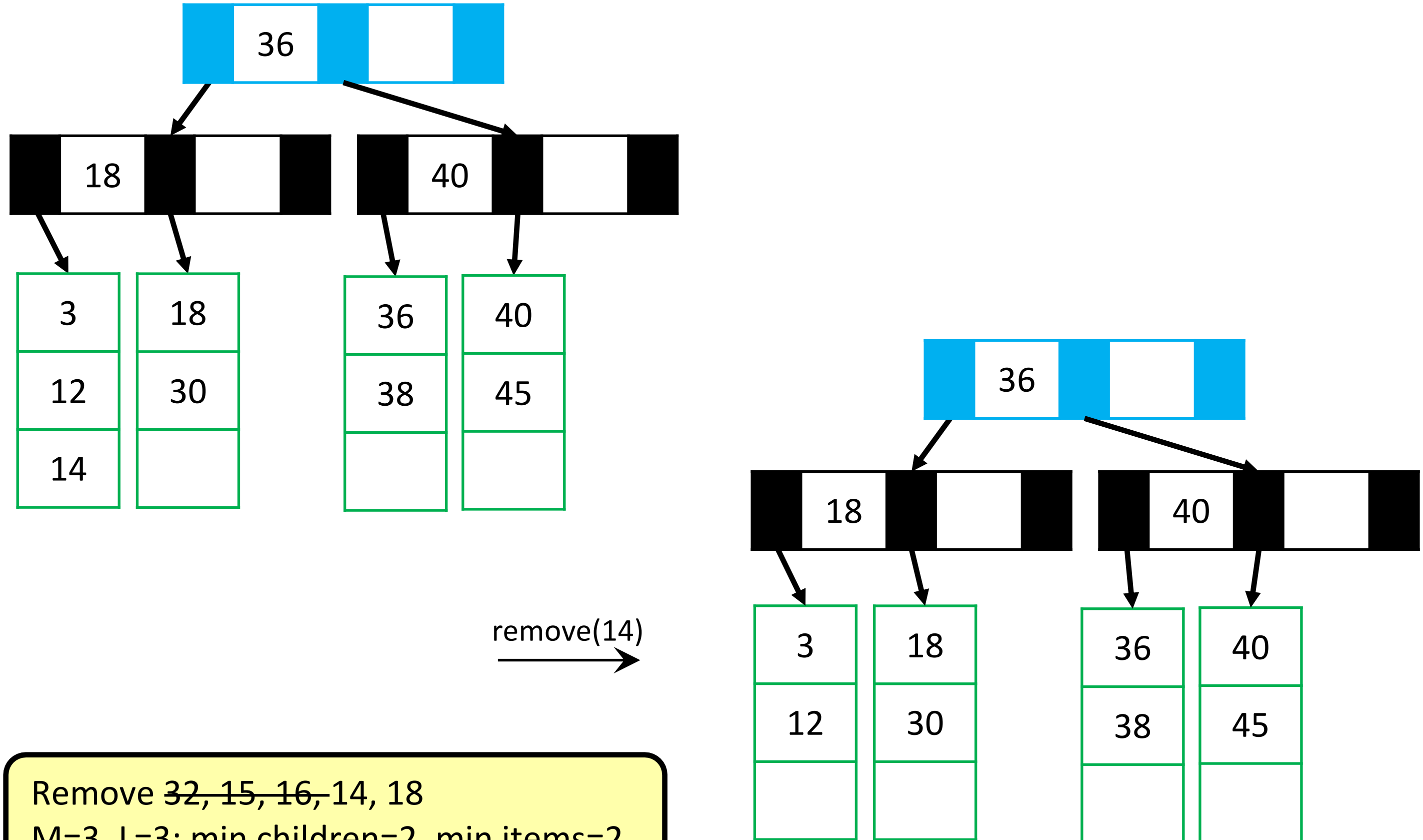


Adopt from a neighbor **node**



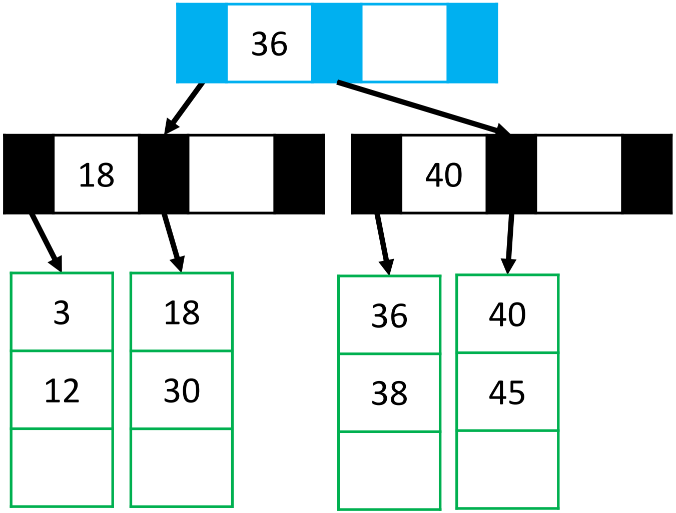
Remove ~~32, 15,~~ 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (5 of 8)



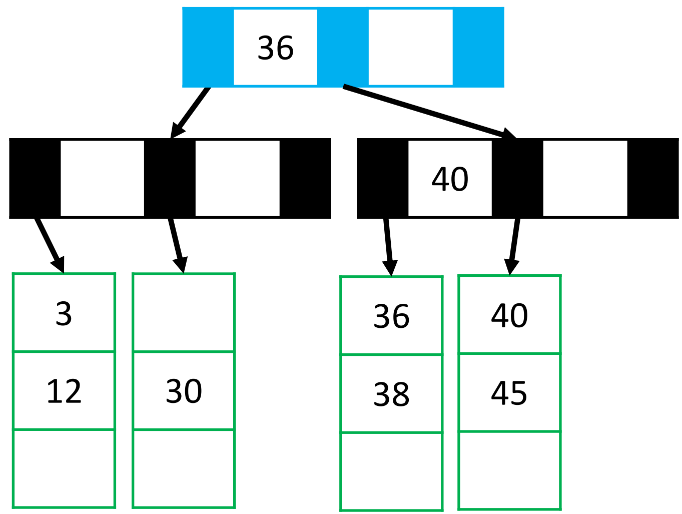
Remove ~~32, 15, 16,~~ 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (6 of 8)



remove(18)

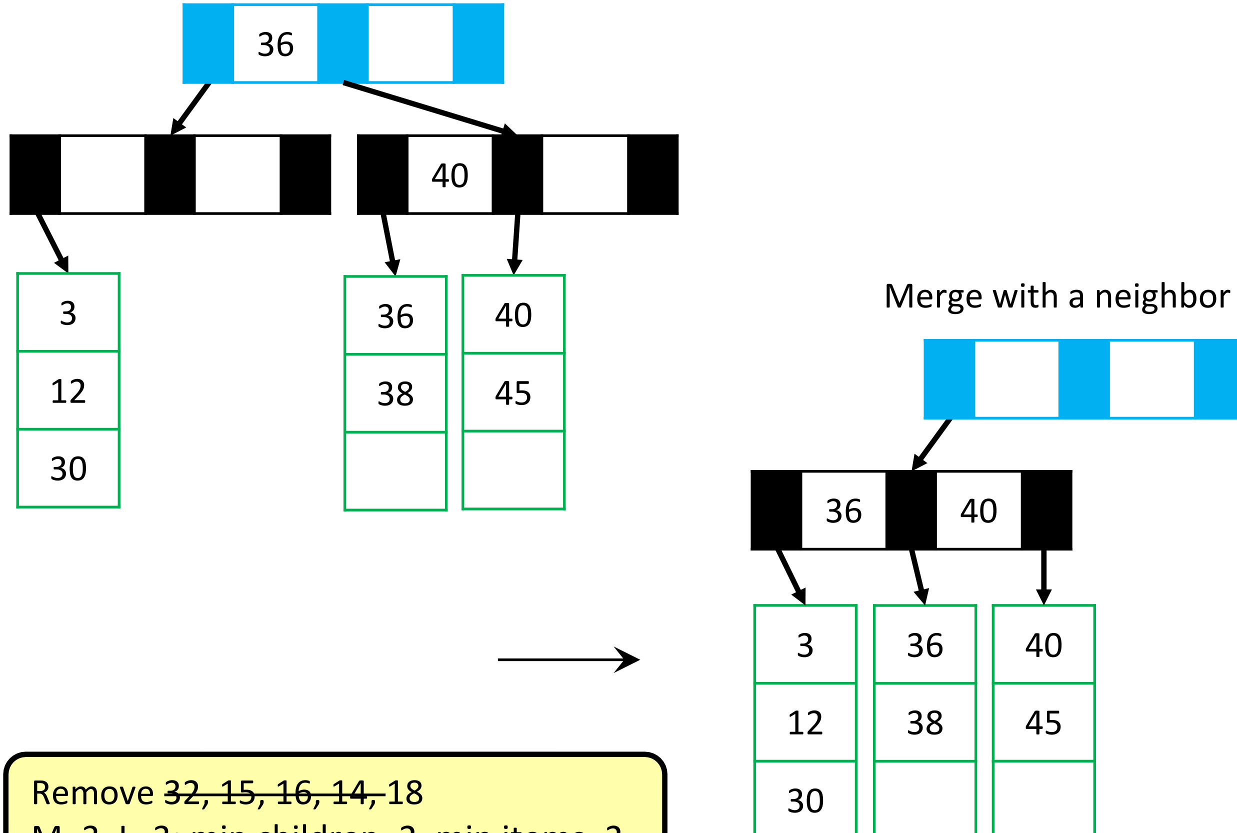
Merge with a neighbor **leaf**



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

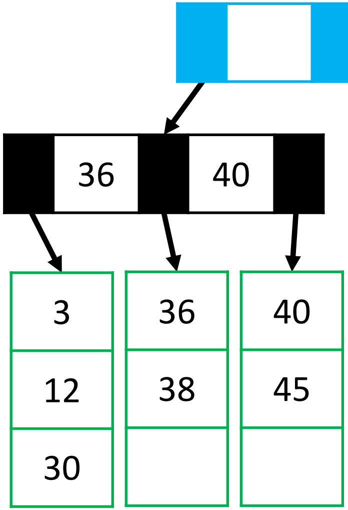
Remove Example: Answer (7 of 8)

Merge with a neighbor **node**

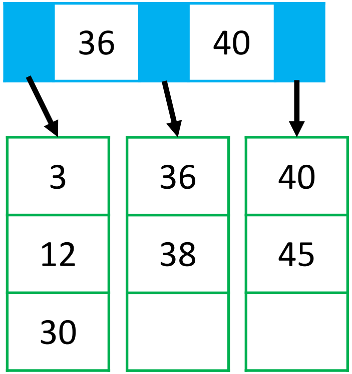


Remove ~~32, 15, 16, 14,~~ 18
M=3, L=3; min children=2, min items=2

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: Efficiency (1 of 2)

- Find correct **leaf**: $O(\mathtt{log}_2 M\ \mathtt{log}_M n)$
- 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**: $O(M\ \mathtt{log}_M n)$
  - Why?

- Total: $O(L + M\ \mathtt{log}_M n)$

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 $O(\log_M n)$

Lecture Outline

- Recap

- B+ Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Find
  - B+ Tree Add
  - B+ Tree Remove
  - **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!
  - 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*…

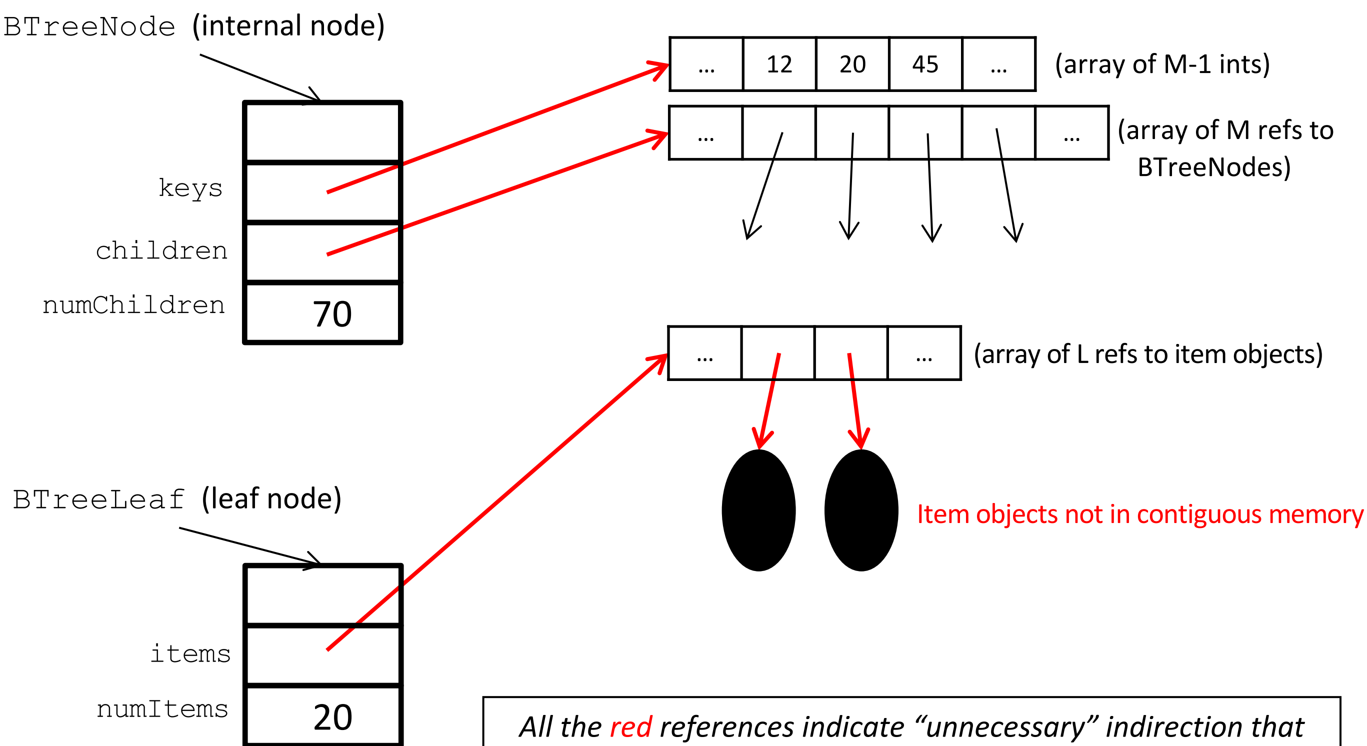
Possible Java Implementation: Code

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;
  …
}
```

Why is the code bad for B+Tree?



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

B+ Trees in Java: Just say no

- 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!

Summary: Search Trees

- **Binary Search Trees** make good dictionaries because they implement `find`, `add`, and `remove` as well as a number of useful operations such as `fl`~~`att`~~`enIntoSortedList` or `successor`
  - Essential and beautiful computer science

- *Balanced* search trees guarantee logarithmic-time operations
  - … 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

- Next up: dictionaries that don't rely on trees at all!