B-Trees (cont)

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

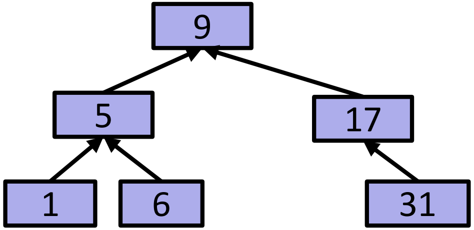
Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- In a B-tree, what is M? What is L? How does it relate to the two node types?

- What constraints are placed on the keys in the subtree rooted at the 5, and the keys on the subtree rooted at the 6?



Announcements

- Expected turnaround time for quiz and project grading: ~1.5w
- We are *not* moving the p2 deadlines
  - … which includes the checkpoint deadline, tomorrow!

Lecture Outline

- B-Trees
  - **Review and B+ Tree Add**
  - B+ Tree Remove
  - Wrapup
- Balanced Tree Wrapup

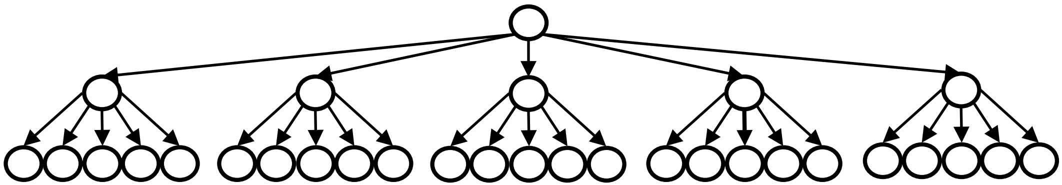
B-Tree Reminder: "Just Another Dictionary"

- Keep in mind how we got here:
  - Large data sets won't fit entirely in memory
  - Disk access is slooooooooooooooooooowwwwwwwww
  - Design a search tree so we do one disk access per node
  - Then, our goal becomes: keep this search tree as shallow as possible
    - … which minimizes disk accesses

Reminder: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

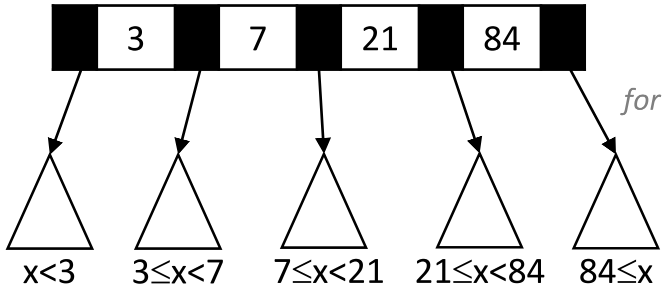
Decision #1: M-ary Search Tree

- A search tree with branching factor M (instead of 2)
  - Each node has a sorted array of M-1 children: `Node[]`



for M = 5

  - Together, M-1 children define the M ranges that we search through



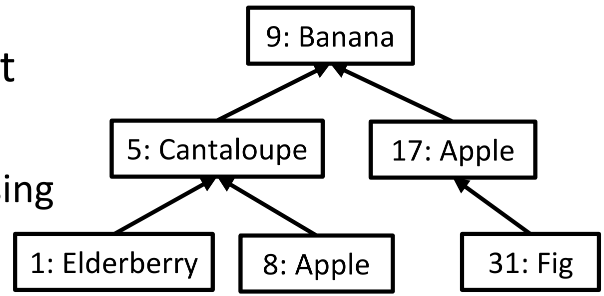
for M = 5

  - Choose M to fit into a disk block: only 1 disk access for entire array!

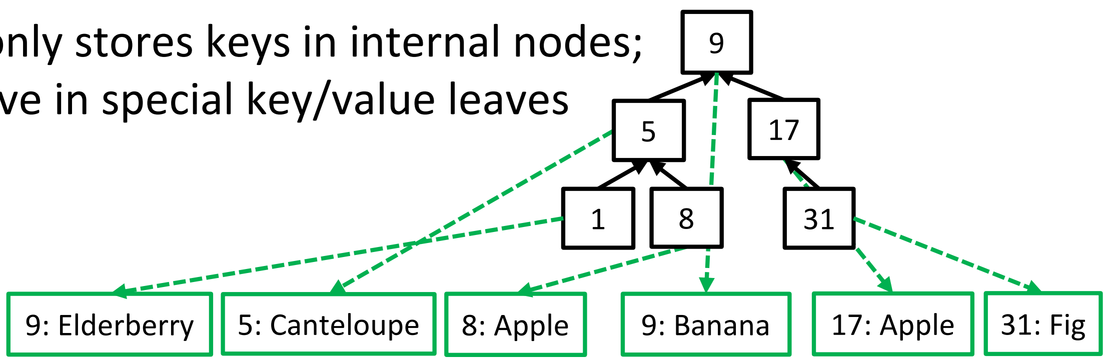
Decision #2: Key-only Internal Nodes

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

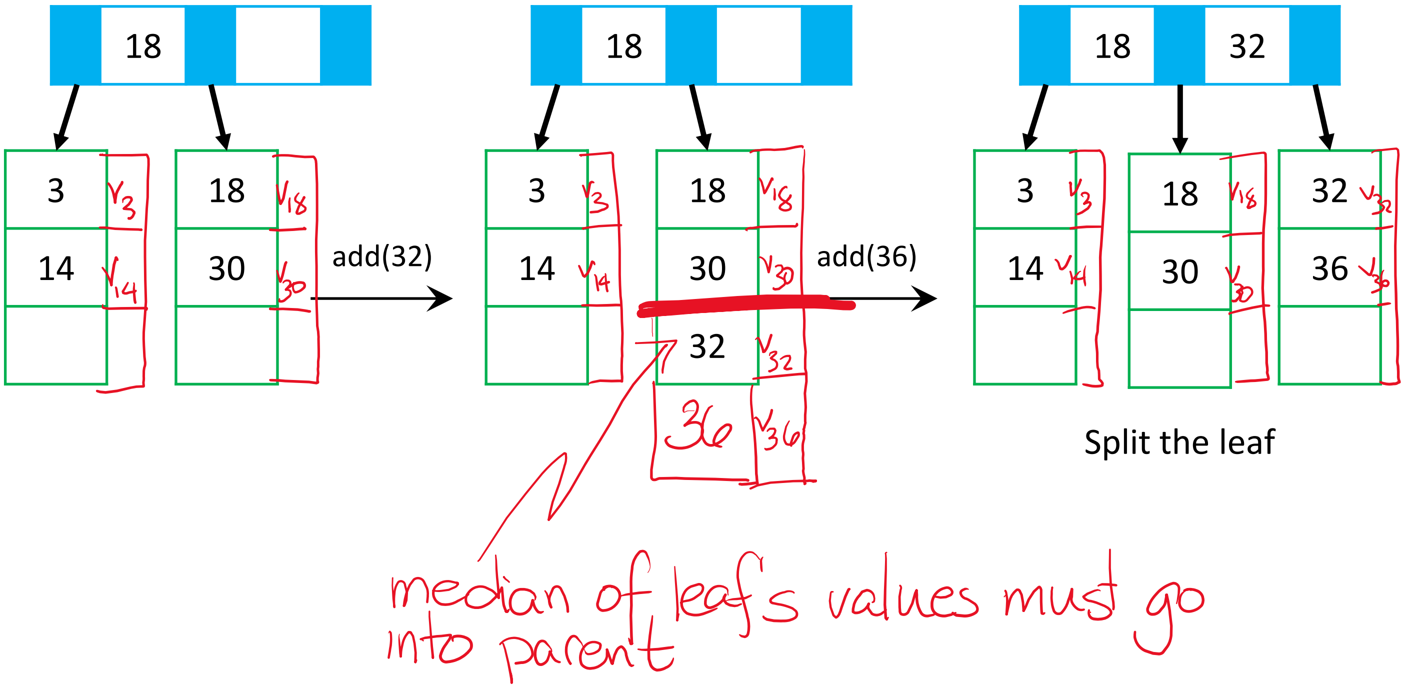
- BST stores value alongside the key at every node
  - Loads entire node even if we are "passing through" to find a different key



- B-Tree only stores keys in internal nodes; values live in special key/value leaves

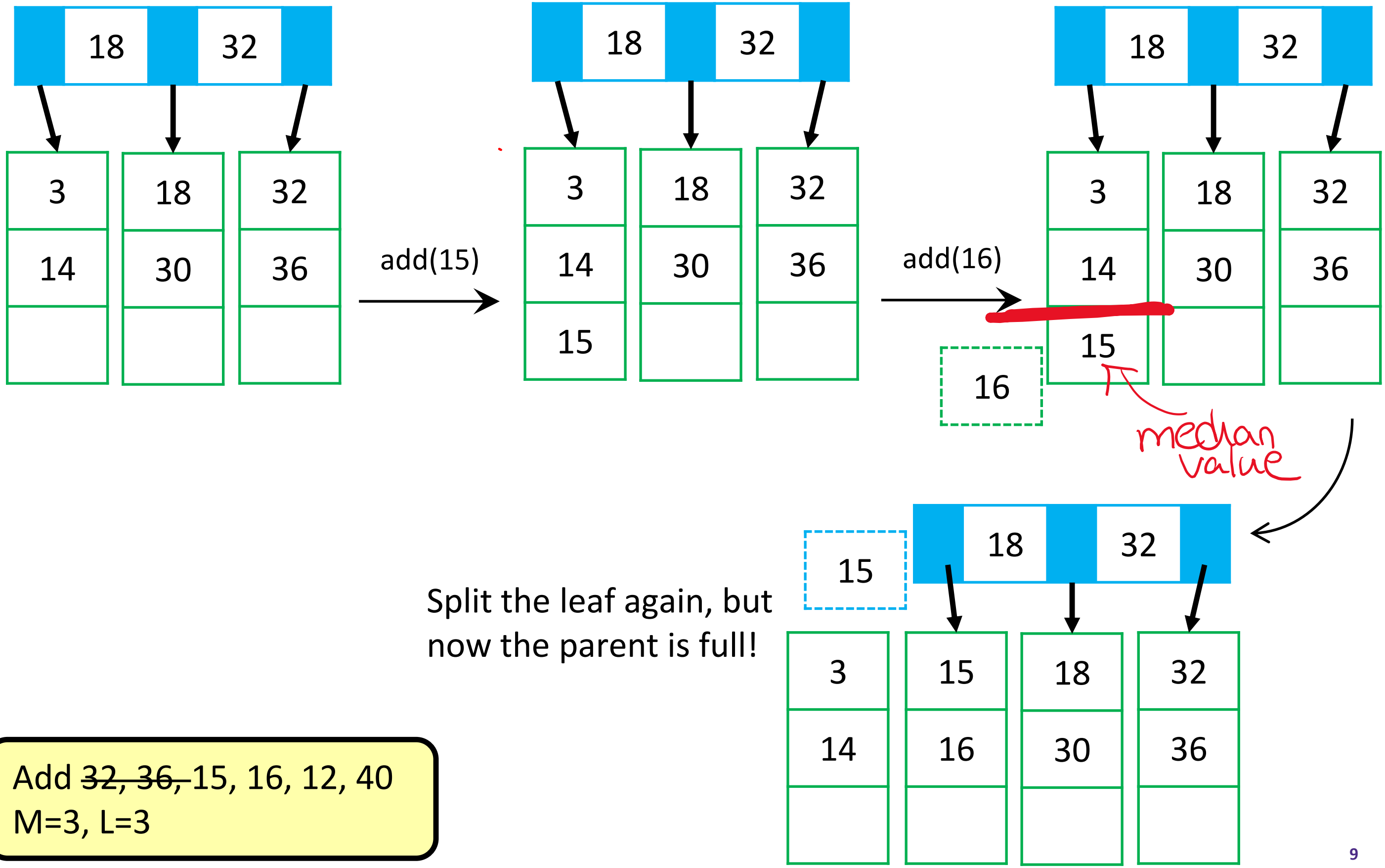


Add Example (1 of 4)



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

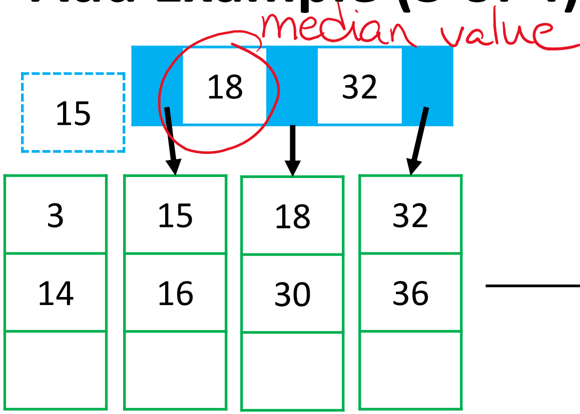
Add Example (2 of 4)

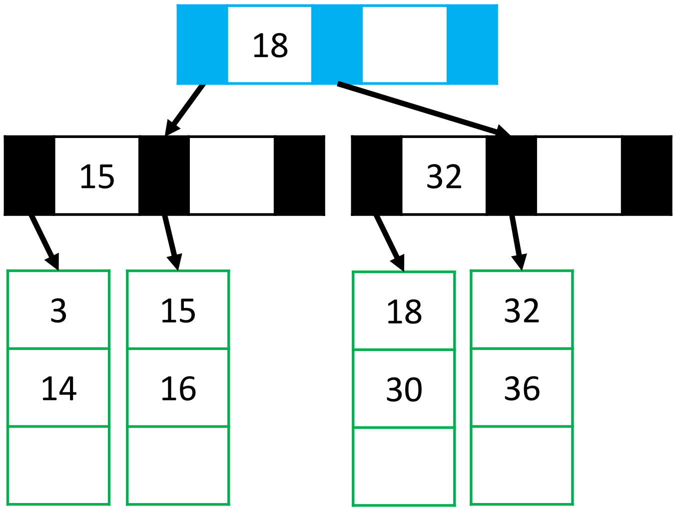


Split the leaf again, but now the parent is full!

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

Add Example (3 of 4)

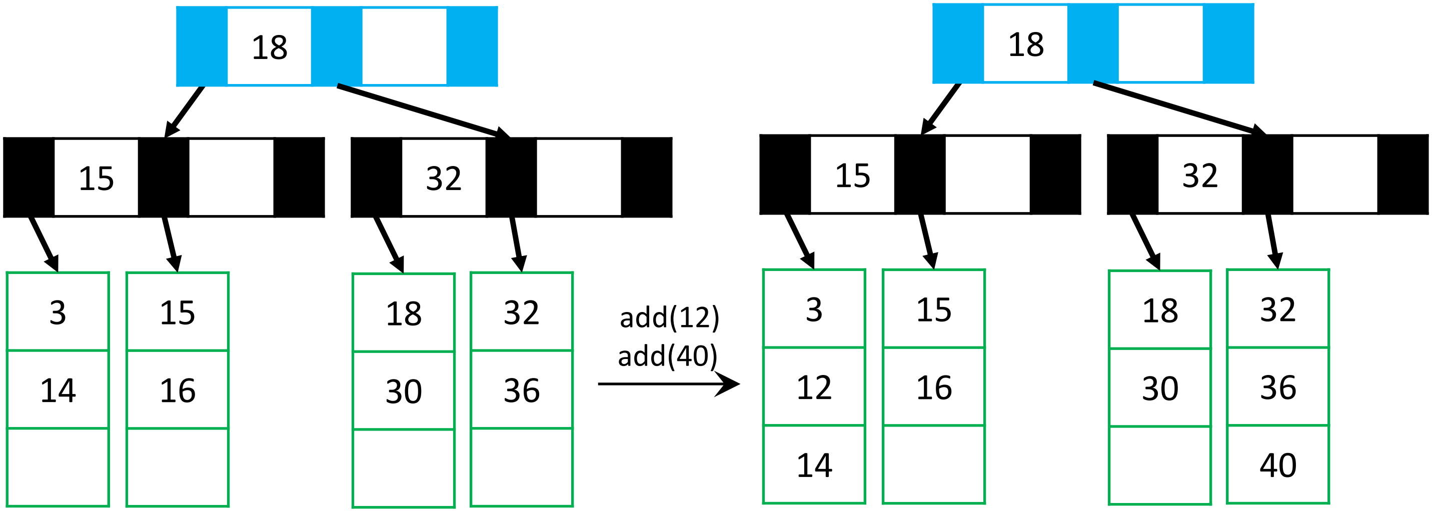




Split the parent (in this case, the root)

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)

- 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/256241

- 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)

per-node binary search — number of nodes traversed (aka tree height)

- Find correct **leaf**: $O(\log_2 M \log_M n)$
- Add (key, value) pair to **leaf**: $O(L)$ — L operations
  - Why? Shift keys/values for insert
- Possibly split **leaf**: $O(L)$ — $\frac{L}{2}$ operations
  - Why? Copy keys/values into split leaf
- Possibly split parents all the way up to **root**: $O(M \log_M n)$?
  - Why? $\frac{M}{2}$ $\log_M n$ (tree height) — copy keys in each split node
- Total: $O(L + M \log_M n)$?

$$\left(\log_2 M \log_M n + L + \frac{L}{2} + \frac{M}{2}\log_M n\right)$$

find — add to leaf — split's copies — split parents

B+ Tree Add: Efficiency (2 of 2)

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

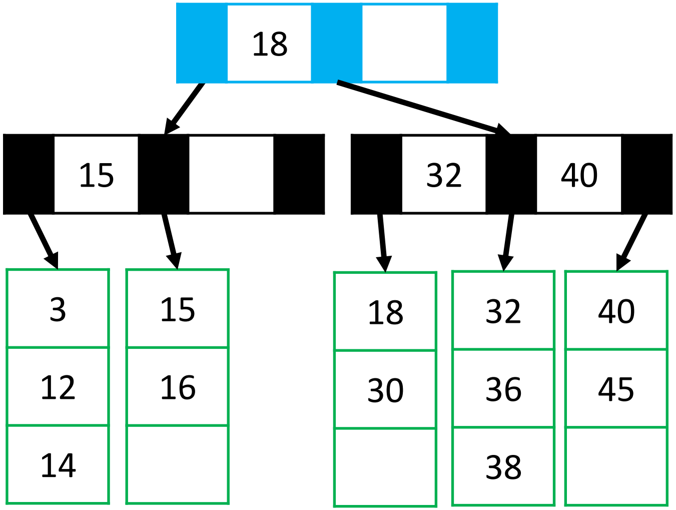
- But the worst-case isn't that common!
  - Splits are uncommon
    - Only required when a node is ***full***
    - M and L are likely to be large and, after a split, nodes will be half empty
  - Splitting the **root** is extremely rare
  - Remember that our goal is minimizing disk accesses! Disk accesses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

Lecture Outline

- B-Trees
  - Review and B+ Tree Add
  - **B+ Tree Remove**
  - Wrapup
- Balanced Tree Wrapup

Remove Example:

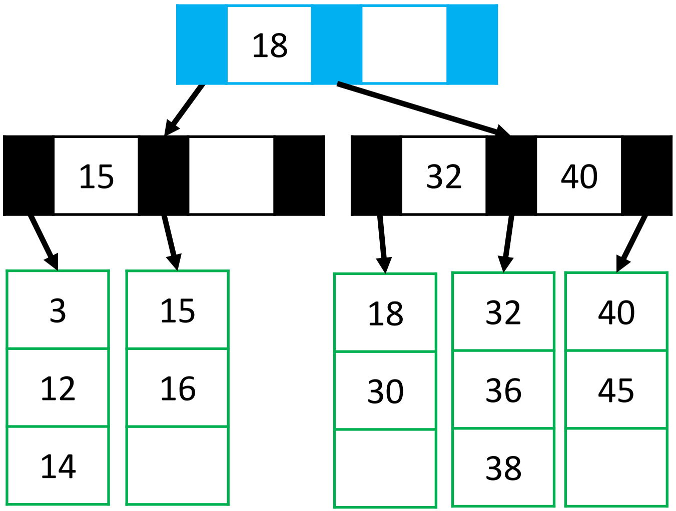
- Remove 32, 15, 16, 14, 18
- M=3, L=3
  - Min #children = 2
  - Min #items = 2



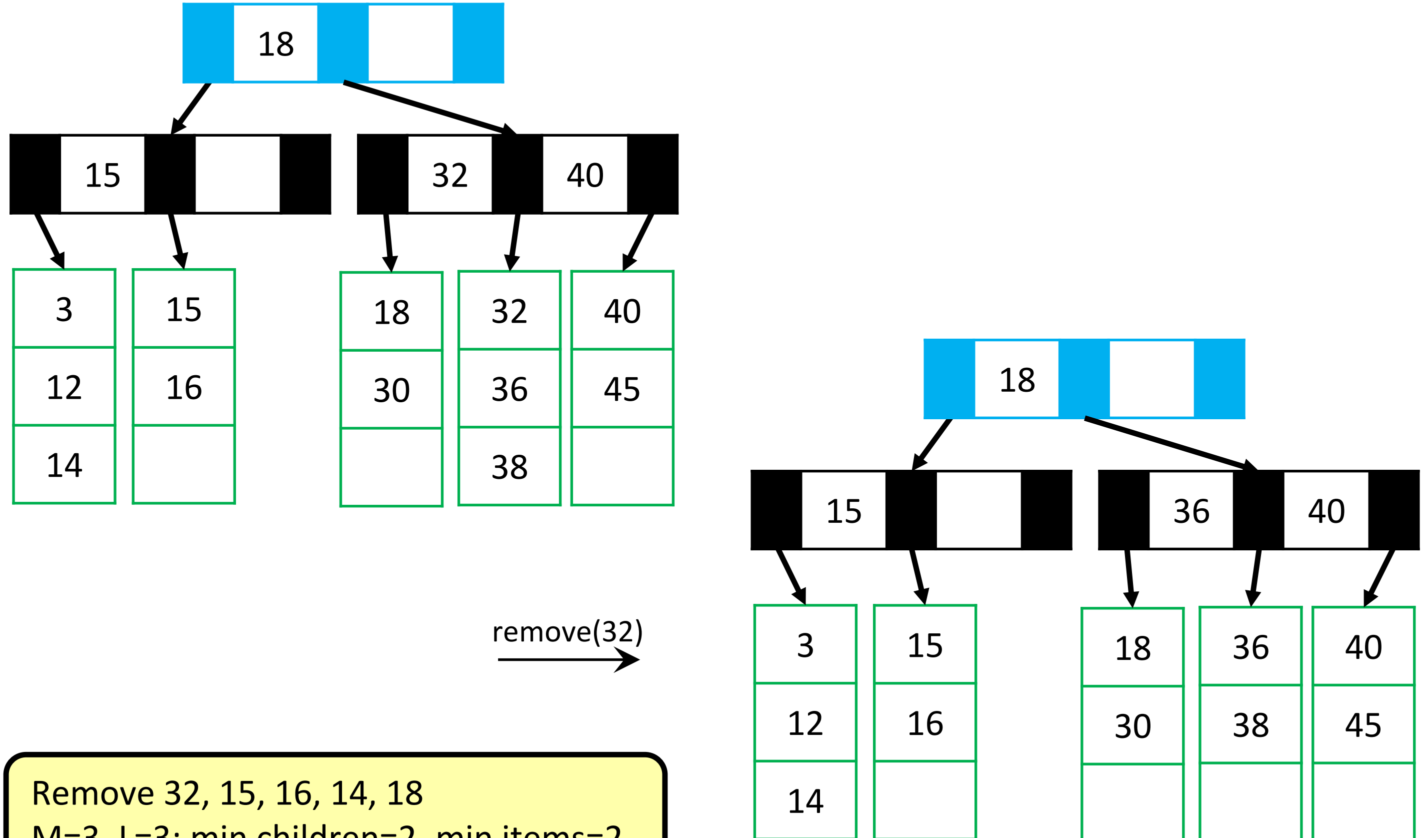
gradescope

gradescope.com/courses/256241

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

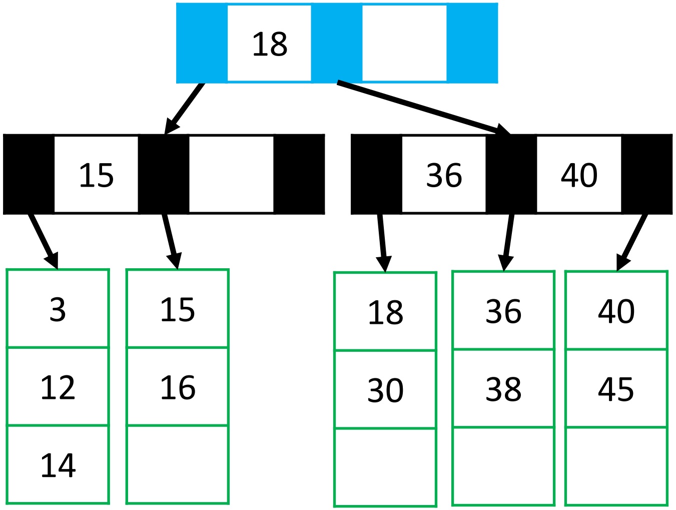


Remove Example: Answer (1 of 8)



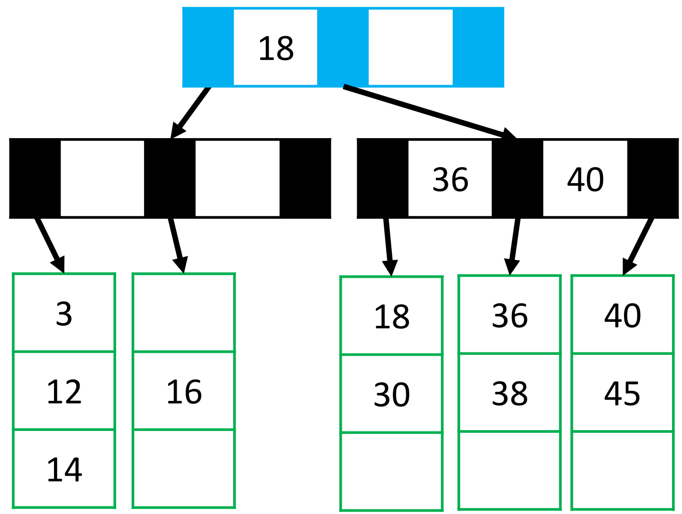
Remove 32, 15, 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (2 of 8)



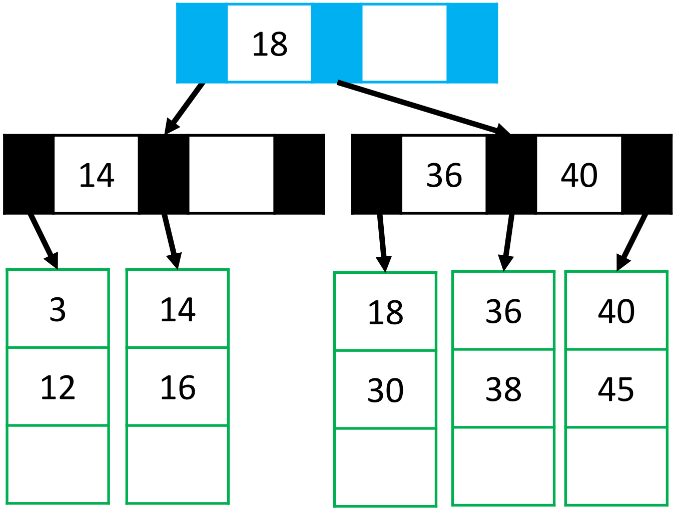
remove(15) →

Adopt an item from a neighbor **leaf**



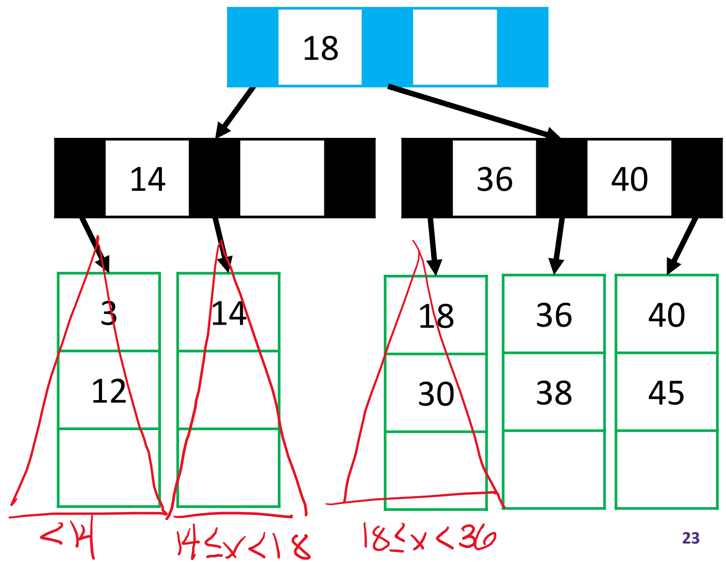
Remove ~~32,~~ 15, 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (3 of 8)



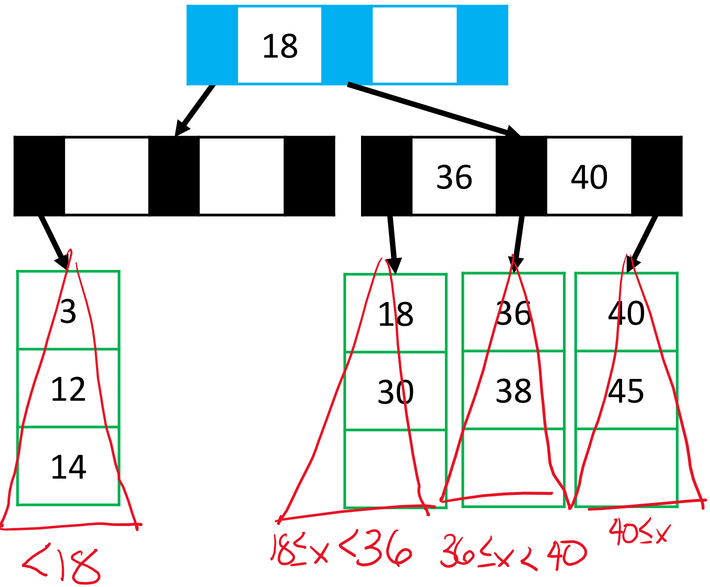
remove(16) →

Merge with a neighbor **leaf**

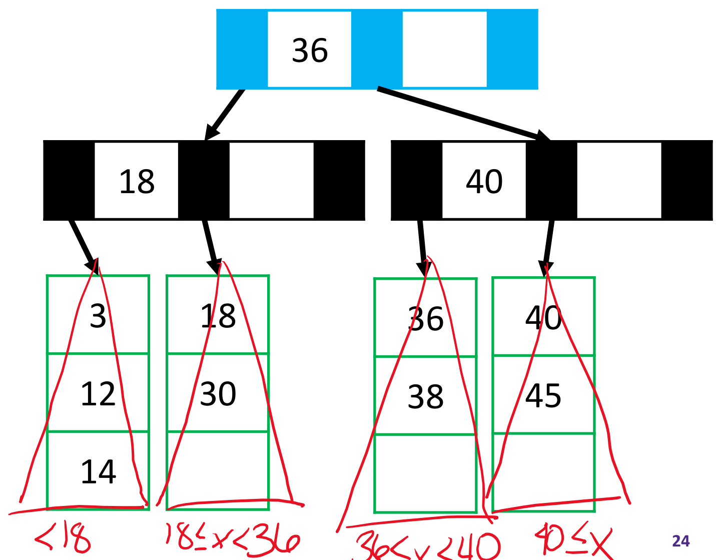


Remove ~~32, 15,~~ 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (4 of 8)

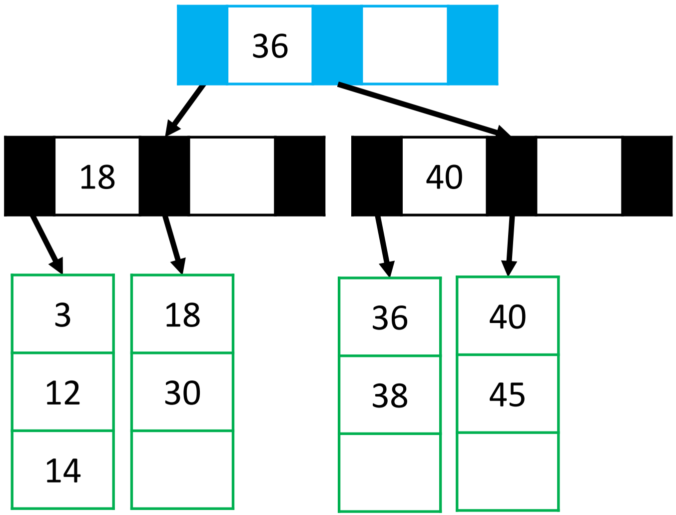


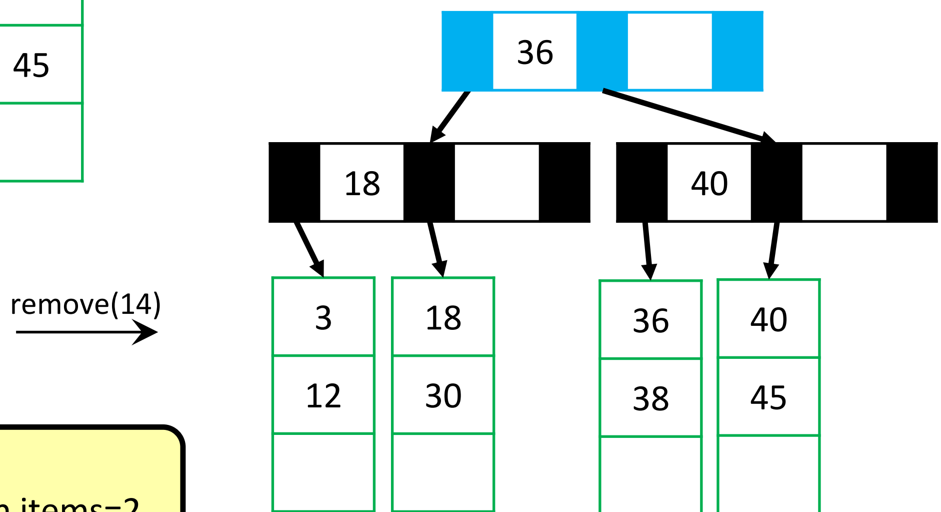
Adopt from a neighbor **node**



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

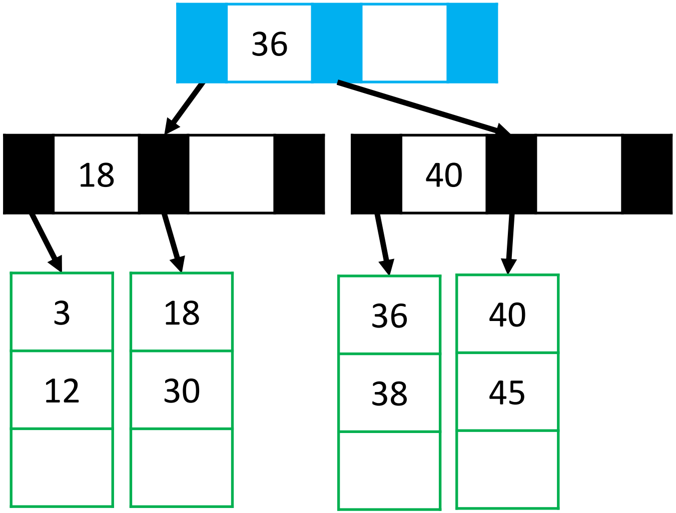
Remove Example: Answer (5 of 8)





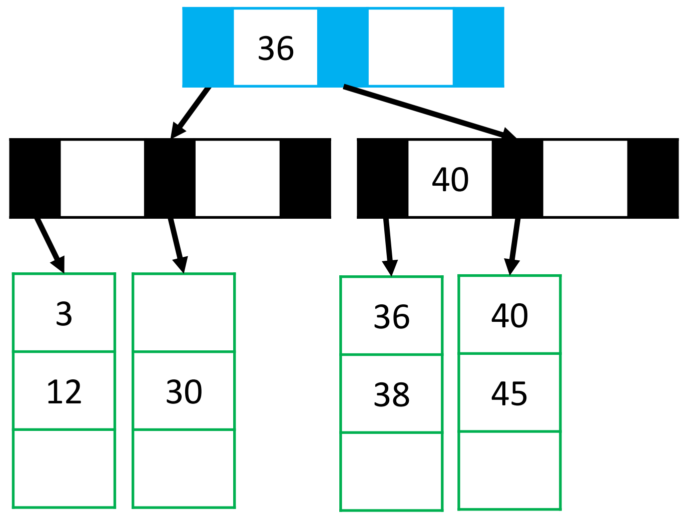
Remove ~~32, 15, 16,~~ 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (6 of 8)



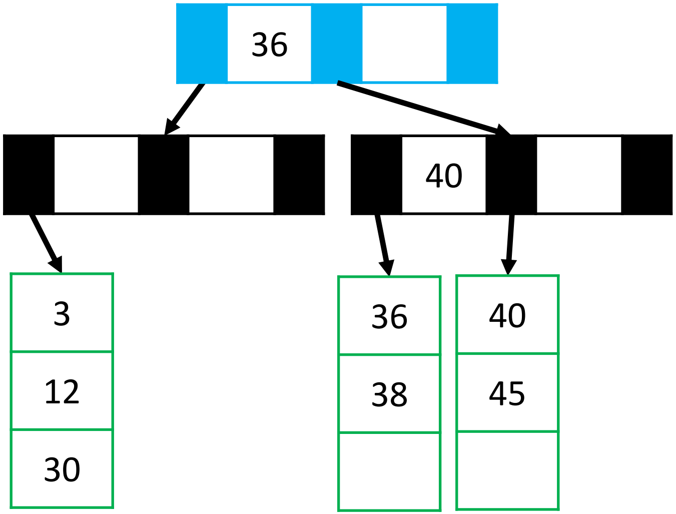
remove(18)

Merge with a neighbor **leaf**

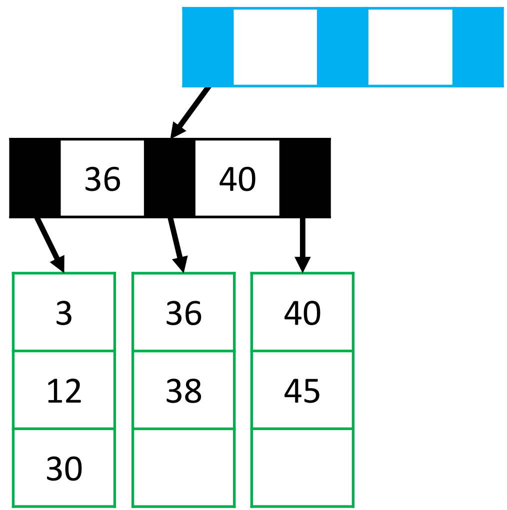


Remove ~~32, 15, 16, 14,~~ 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (7 of 8)

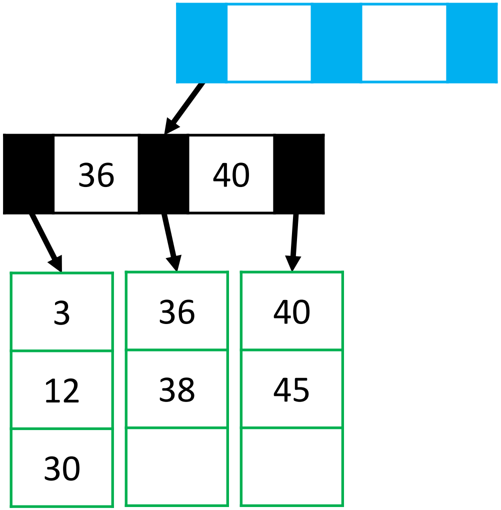


Merge with a neighbor **node**

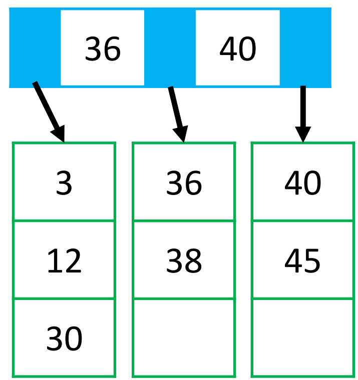


Remove ~~32, 15, 16, 14,~~ 18
M=3, L=3; min children=2, min items=2

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)

- 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**

B+ Tree Remove: Efficiency (1 of 2)

- Find correct **leaf**: $O(\texttt{log}_2 M \ \texttt{log}_M n)$
- Remove item from **leaf**: $O(L)$
  - Why? shift k/v over the "hole"
- Possibly adopt from or merge with neighbor **leaf**: $O(L)$
  - Why? copy neighbor's values into this leaf
- Possibly adopt or merge **parent node** up to **root**: $O(M \ \texttt{log}_M n)$
  - Why? $\frac{M}{2} \log_2 M$
- Total: $O(L + M \ \texttt{log}_M n)$ Same as add

$$\left(\log_2 M \log_M n + L + \frac{L}{2} + \frac{M}{2}\log_2 M\right)$$

find; leaf shift; k/v copies from adopt/merge; adopt/merge up to root

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*** (🤔 half full?)
    - 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

- B-Trees
  - Review and B+ Tree Add
  - B+ Tree Remove
  - **Wrapup**
- Balanced Tree Wrapup
- Hashing
  - Designing Our Own Hash Function
  - Hashing Applications

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*…

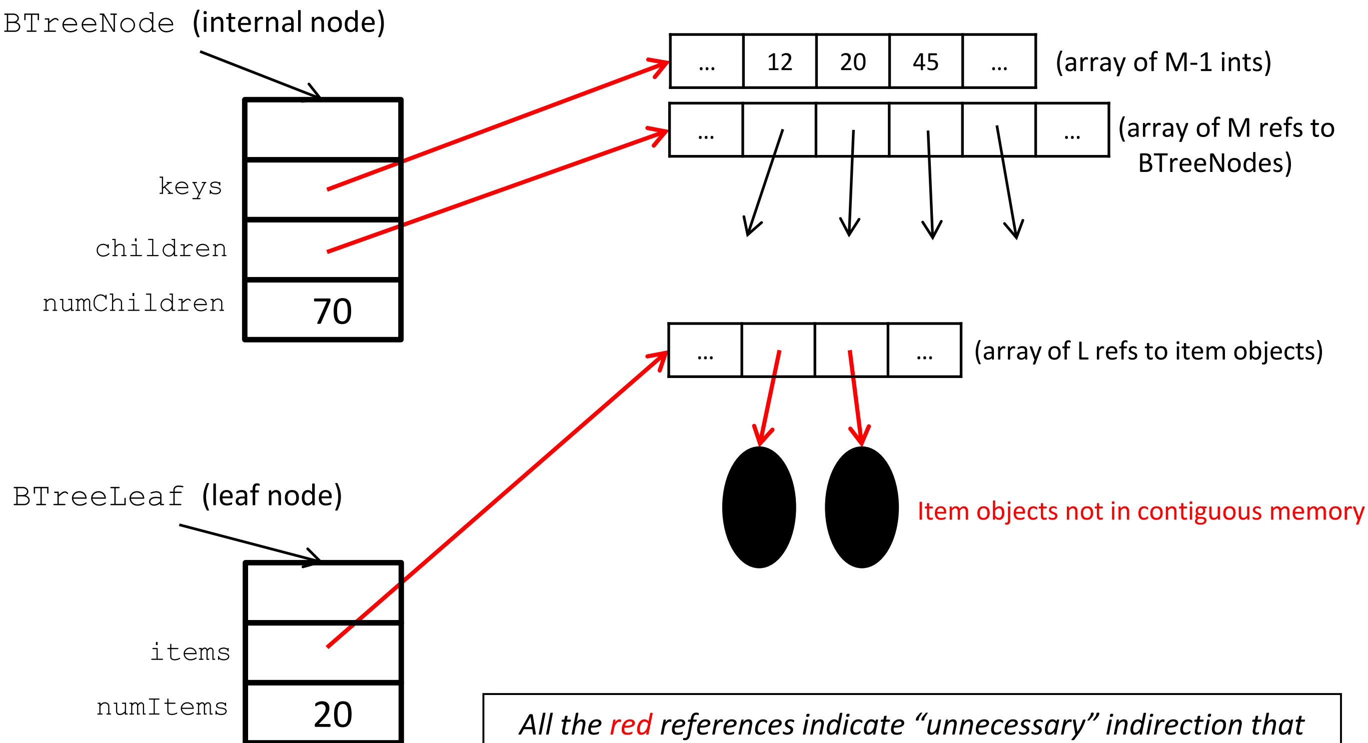
Possible Java Implementation: Code

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!

Lecture Outline

- B-Trees
  - Review and B+ Tree Add
  - B+ Tree Remove
  - Wrapup

- **Balanced Tree Wrapup**

Summary: Search Trees (1 of 2)

- **Binary Search Trees** make good dictionaries because they implement `find`, `add`, and `remove` as well as a number of useful operations such as `flattenIntoSortedList` 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

Summary: Search Trees (2 of 2)

- Most balanced BSTs are **Red-Black trees**
  - No extra space needed: store the (boolean) color in the pointer or as reversed children
  - 1.39x taller than equivalent AVL tree, but still logarithmic in height
  - Deletes are amortized constant
  - Used in linux kernel (scheduler, epoll), C++ and Java libraries

- But difficult to reason about (especially in a lecture), so we use AVL and B+ trees to illustrate the *ideas* and *techniques*
  - Also interesting are **splay trees**: self-adjusting; amortized guarantee; no extra space for height information

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