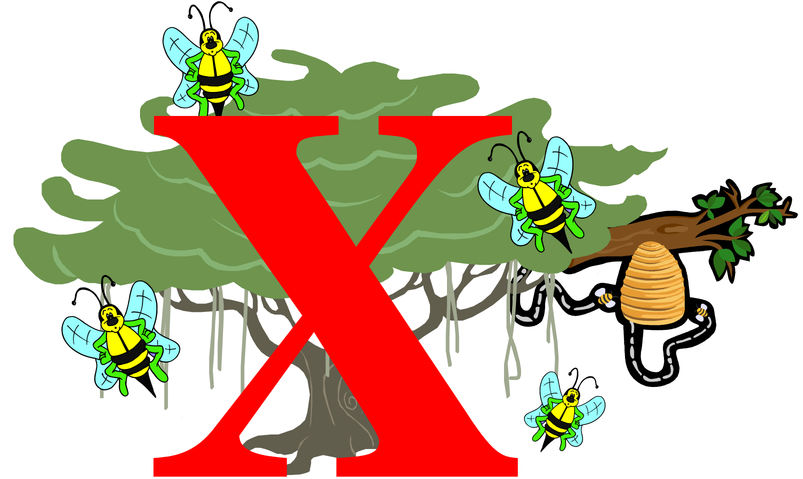
CSE332: Data Abstractions

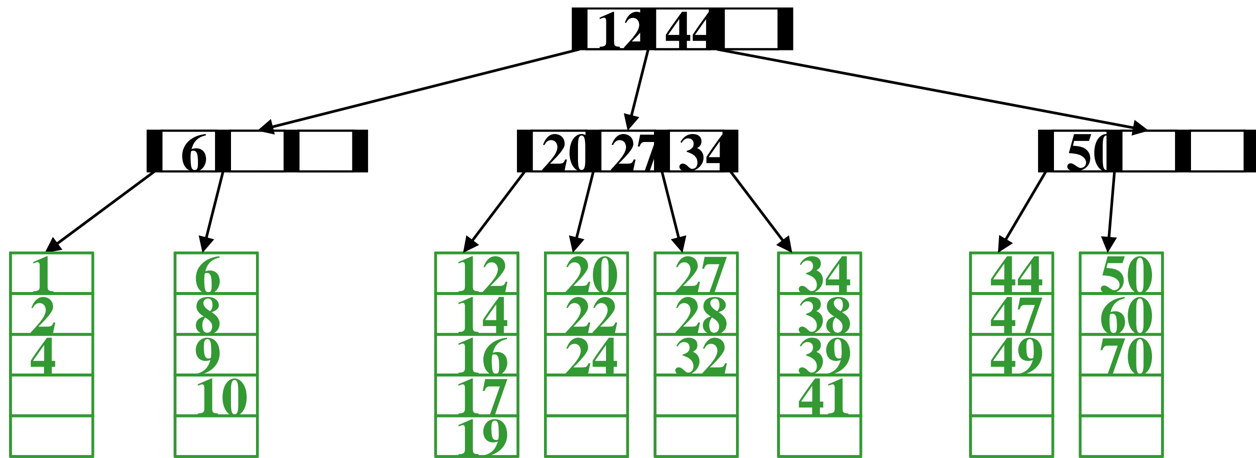
Lecture 9: BTrees

Tyler Robison

Summer 2010

BTrees



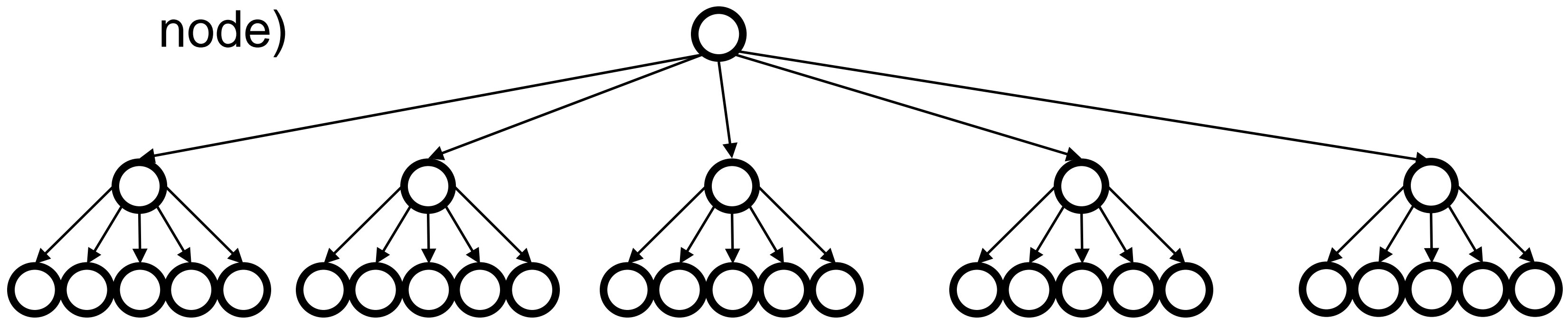


Our goal

- Problem: What if our dictionary has so much data, most of it resides on disk; very slow to access
- Say we had to do a disk access for each node access
  - Unbalanced BST with n nodes: n disk accesses (worst case)
  - AVL tree: $\log_2 n$ accesses (worst case)
    - $\log_2(2^{30})=30$ disk accesses still a bit slow
    - An improvement, but we can do better
- Idea: A balanced tree (logarithmic height) that is even shallower than AVL trees so that we can minimize disk accesses and exploit disk-block size
  - Increase the branching factor to decrease the height
  - Gives us height $\log_3 n$, $\log_{10} n$, $\log_{50} n$ etc., based on branching factor
  - Asymptotically still O(logn) height though

M-ary Search Tree

- Build some kind of search tree with branching factor *M*:
  - Say, each node has an array of M sorted children (`Node[]`)
  - Choose *M* to fit node snugly into a disk block (1 access per node)



- # hops for `find`: If balanced, using $\log_M n$ instead of $\log_2 n$
  - If *M*=256, that's an 8x improvement
  - Example: $M = 256$ and $n = 2^{40}$ that's 5 instead of 40
- To decide which branch to take, divide into portions
  - Binary tree: Less than node value or greater?
  - M-ary: In range 1? In range 2? In range 3?... In range M?
- Runtime of `find` if balanced: $O(\log_2 M \log_M n)$
  - Hmmm… $\log_M$ n is the height we traverse. Why the $\log_2 M$ multiplier?
  - $\log_2 M$: At each step, find the correct child branch to take using binary search

Problems with how to proceed

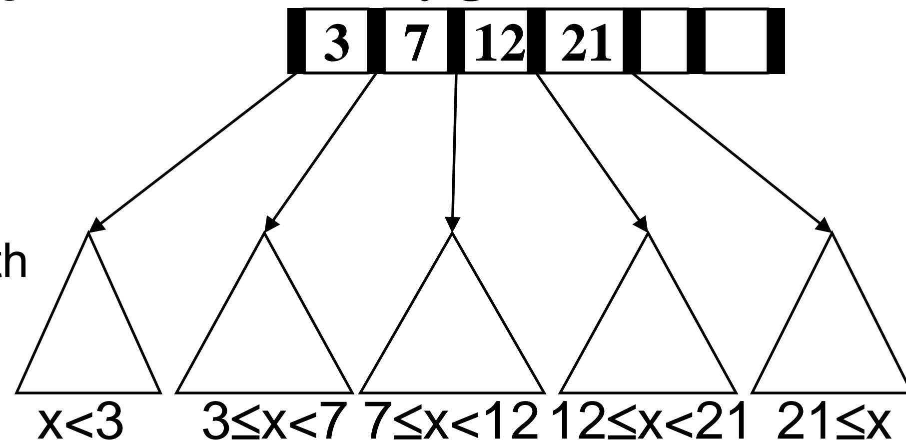
- What should the order property be? How do we decide the 'portion' each child will take?
- How would you rebalance (ideally without more disk accesses)?
- Storing real data at inner-nodes (like in a BST) seems kind of wasteful…
  - To access the node, will have to load data from disk, even though most of the time we won't use it

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
  - In example on right, M=7
  - No other data; all data at the leaves!
  - Think of M-1 keys stored in internal nodes as ‘signposts’ used to find a path to a leaf
- Order property:

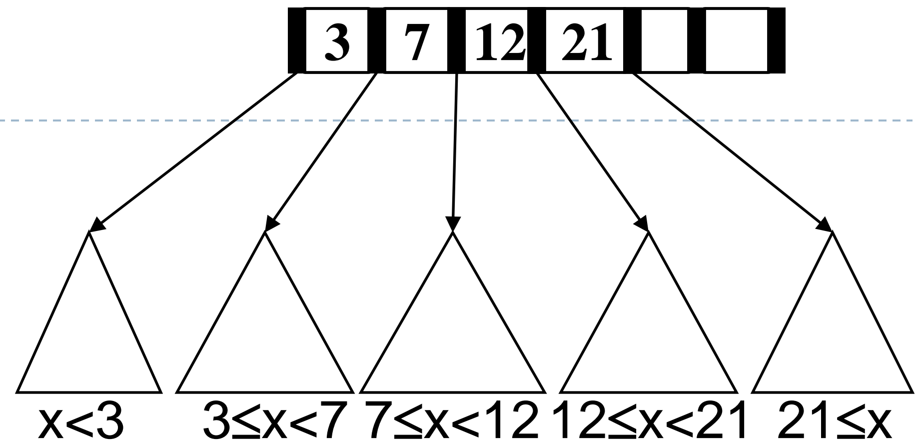
  Subtree **between** keys *x* and *y* contains only data that is ≥ ***x*** and < ***y*** (notice the ≥)
- Leaf nodes (not shown here) have up to *L* sorted data items
- Remember:
  - Leaves store data
  - Internal nodes are ‘signposts’

Empty cells at the end are currently unused, but may get filled in later



What’s the ‘B’ for? Wikipedia quote from 1979 textbook:
The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.

Find



- Different from BST in that we don't store values at internal nodes

- But **`find`** is still an easy recursive algorithm, starting at root
  - At each internal-node do binary search on the (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

Remember: M=max # children
L=max items in leaf

- Non-root Internal nodes
  - Have between $\lceil M/2 \rceil$ and M children (inclusive), i.e., at least half full
- Leaf nodes
  - All leaves at the same depth
  - Have between $\lceil L/2 \rceil$ and L data items (inclusive), i.e., at least half full
- Root (special case)
  - If tree has $\leq L$ items, root is a leaf (occurs when starting up; otherwise unusual); can have any number of items $\leq L$
  - Else has between 2 and M children

(Any $M > 2$ and L will work; picked based on disk-block size)

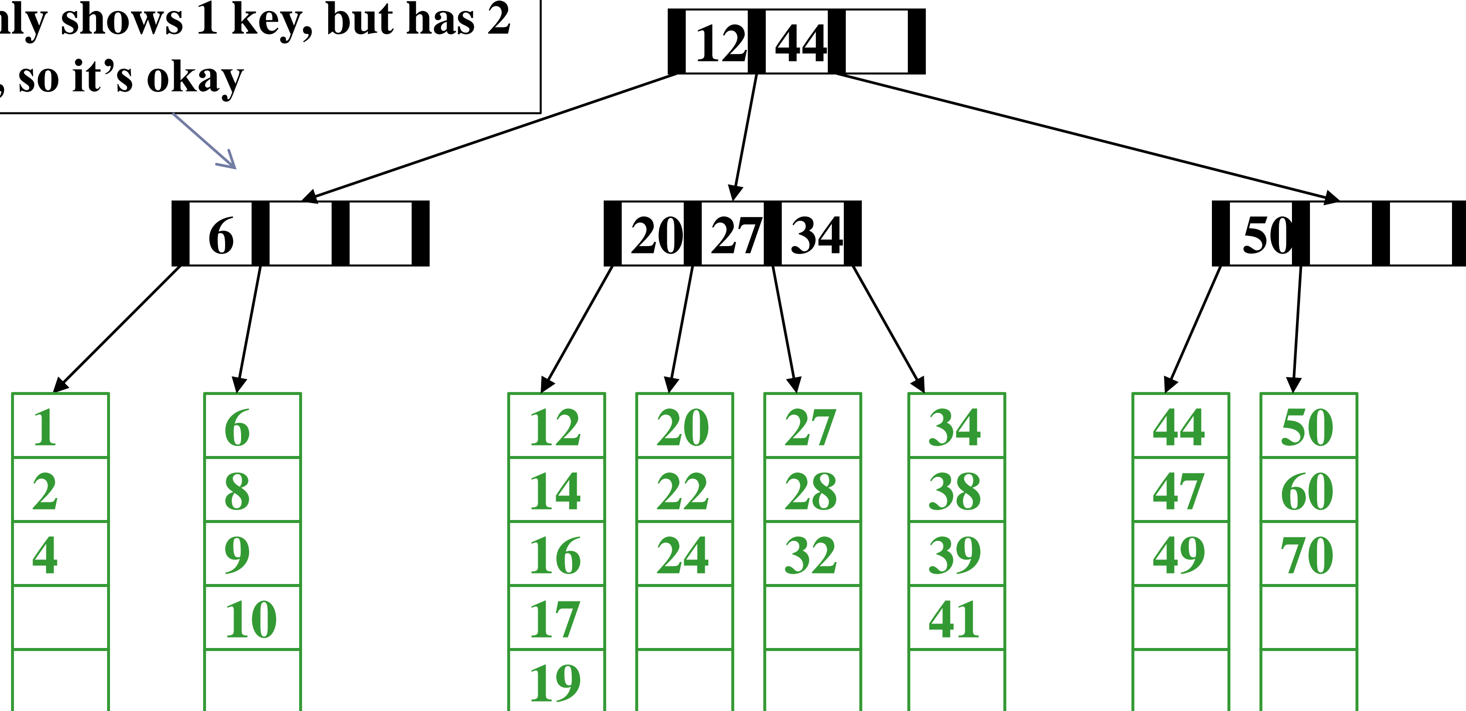
- Uh, why these bounds for internal nodes & children?
  - Upper bounds make sense: don't want one long list; pick to fit in block
  - Why lower bounds?
    - Ensures tree is sufficiently 'filled in': Don't have unnecessary height, for instance
    - Guarantees 'broadness'
    - We'll see more when we cover insert & delete

Example

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

Suppose M=4 (max children) and L=5 (max 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



Balanced enough

Not too difficult 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…

$$n \geq 2 \lceil M/2 \rceil^{h-1} \lceil L/2 \rceil$$

$2 \lceil M/2 \rceil^{h-1}$: minimum number of leaves

$\lceil L/2 \rceil$: minimum data per leaf

Exponential in height because $\lceil M/2 \rceil > 1$

Disk Friendliness

What makes B trees so disk friendly?

- Many keys stored in one node
  - All brought into memory in one disk access
    - *IF* we pick *M* wisely
  - Makes the binary search over *M*-1 keys totally worth it (insignificant compared to disk access times)
- Internal nodes contain only keys
  - Any `find` wants only one data item; wasteful to load unnecessary items with internal nodes
  - So only bring one leaf of data items into memory
  - Data-item size doesn't affect what *M* is

Maintaining balance

- So this seems like a great data structure (and it is)

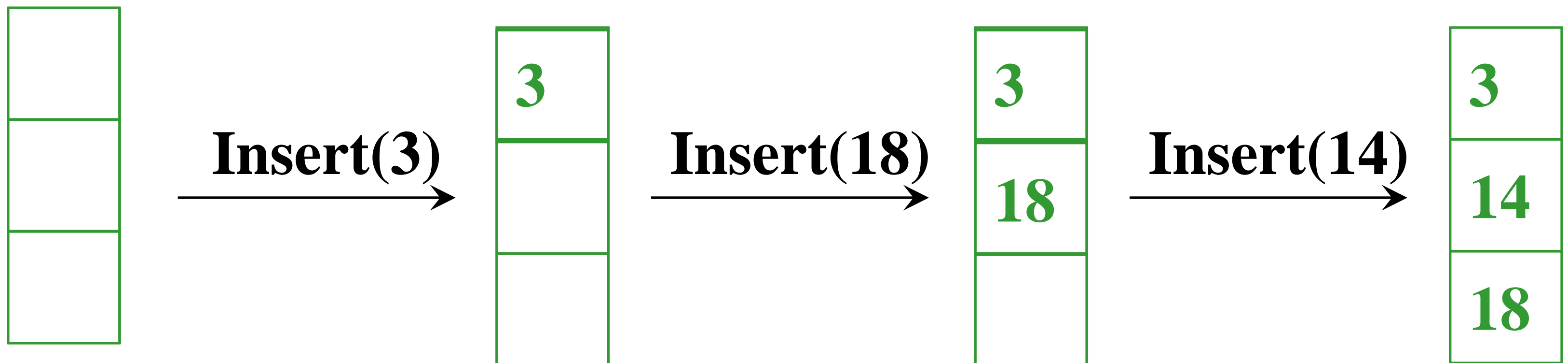
- But we haven't implemented the other dictionary operations yet
  - `insert`
  - `delete`

- As with AVL trees, the hard part is maintaining structure properties
  - Example: for **`insert`**, there might not be room at the correct leaf
  - Unlike AVL trees, there are no rotations ☺

Building a B-Tree (insertions)

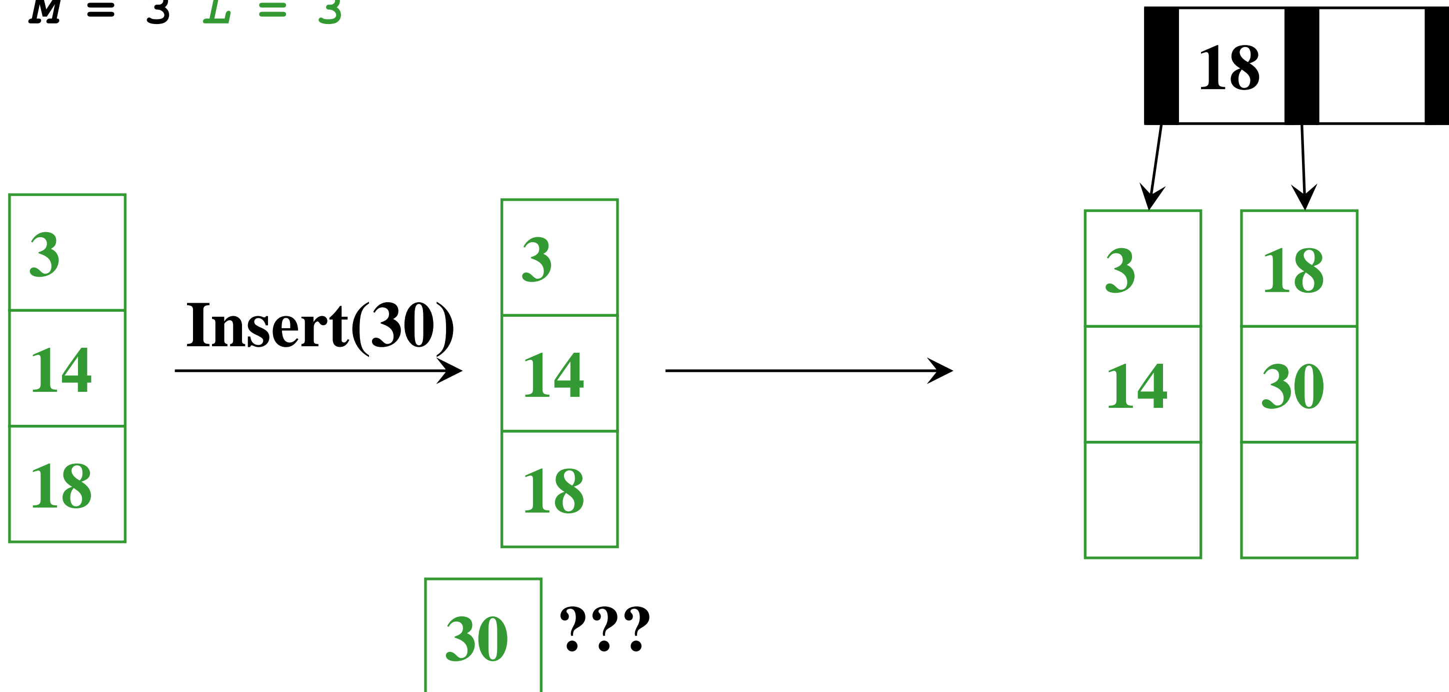
M = 3 *L* = 3

Remember: Horizontal=internal node; Vertical=leaf



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

Just need to keep data in order

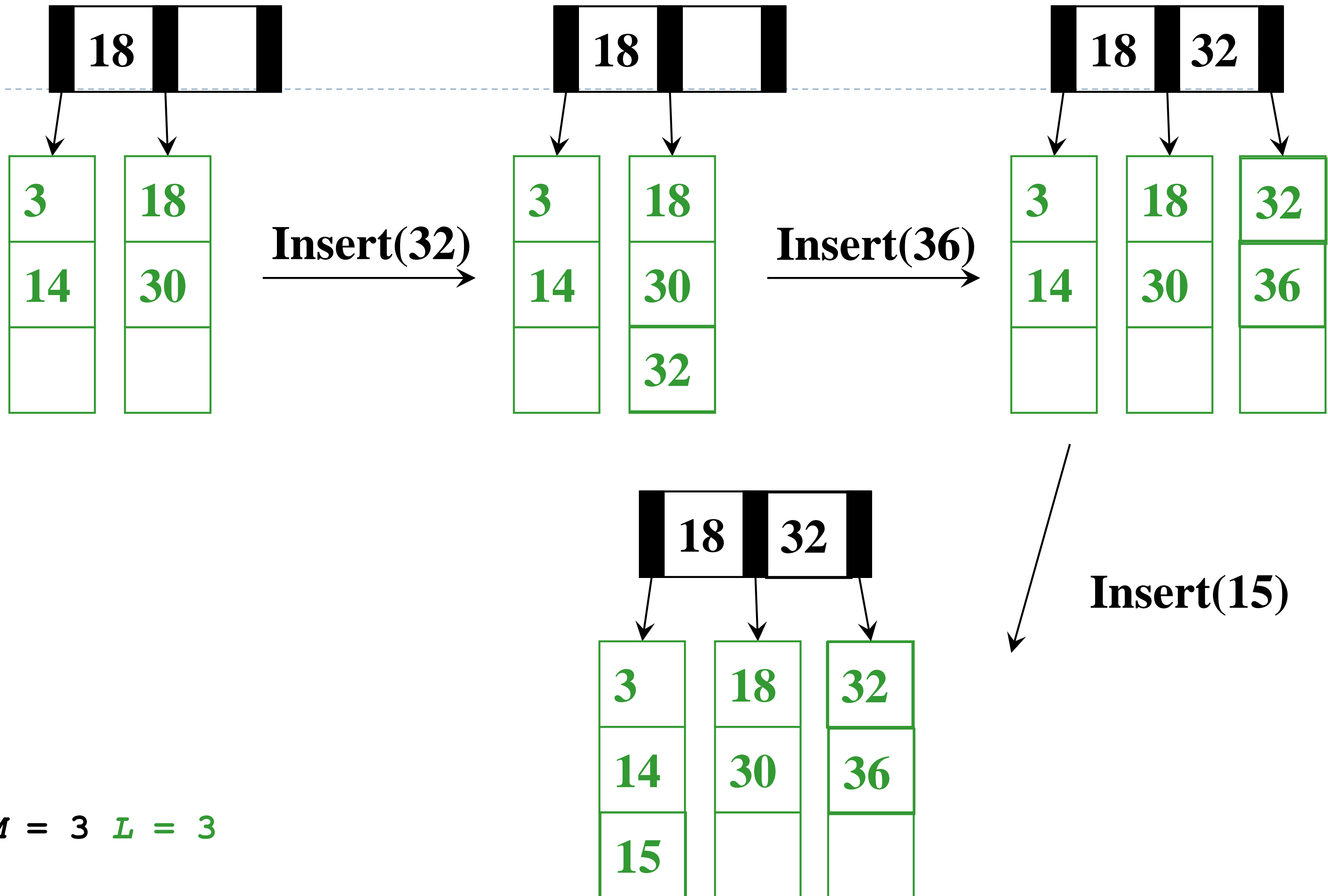


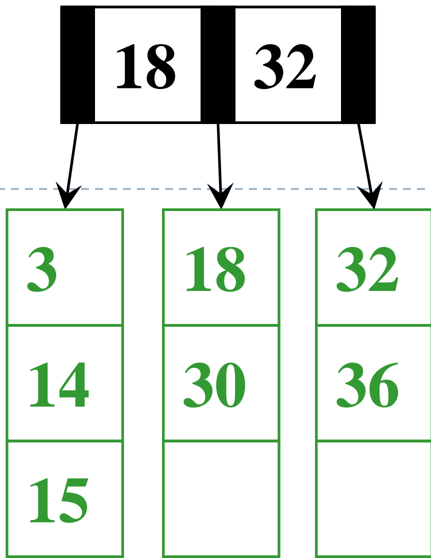
•When we ‘overflow’ a leaf, we split it into 2 leaves

•Parent gains another child

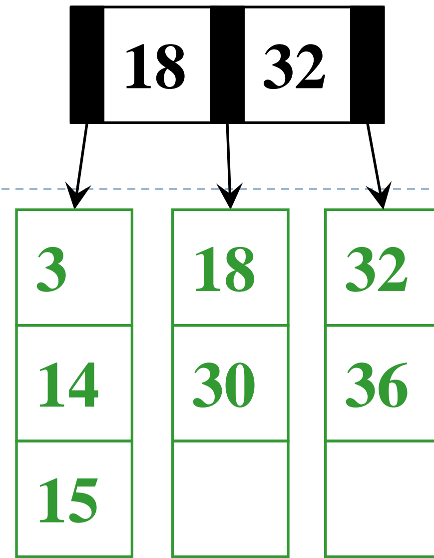
•If there is no parent (like here), we create one; how do we pick the key shown in it?

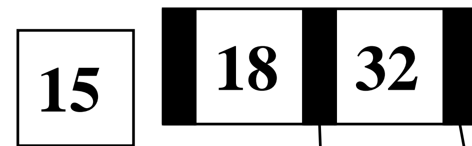
  •Smallest element in right tree


Split leaf again
18
3
14
18
30
Insert(32)
18
3
14
18
30
32
Insert(36)
18
32
3
14
18
30
32
36
Insert(15)
18
32
3
14
15
18
30
32
36
M = 3 L = 3

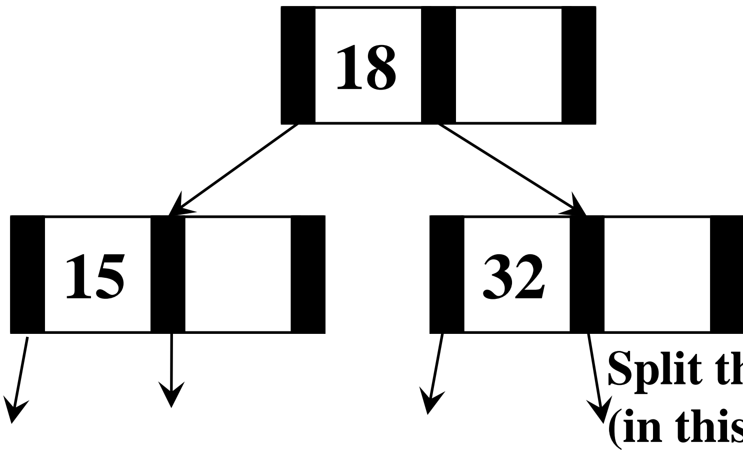


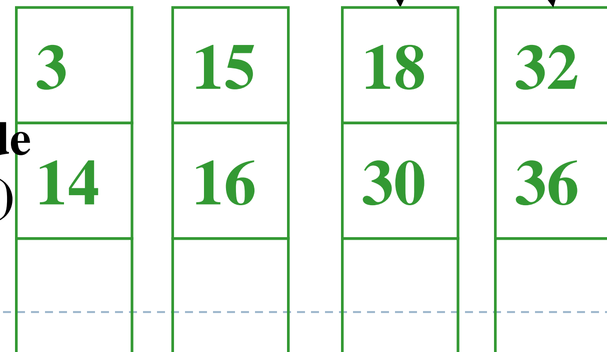
Insert(16)





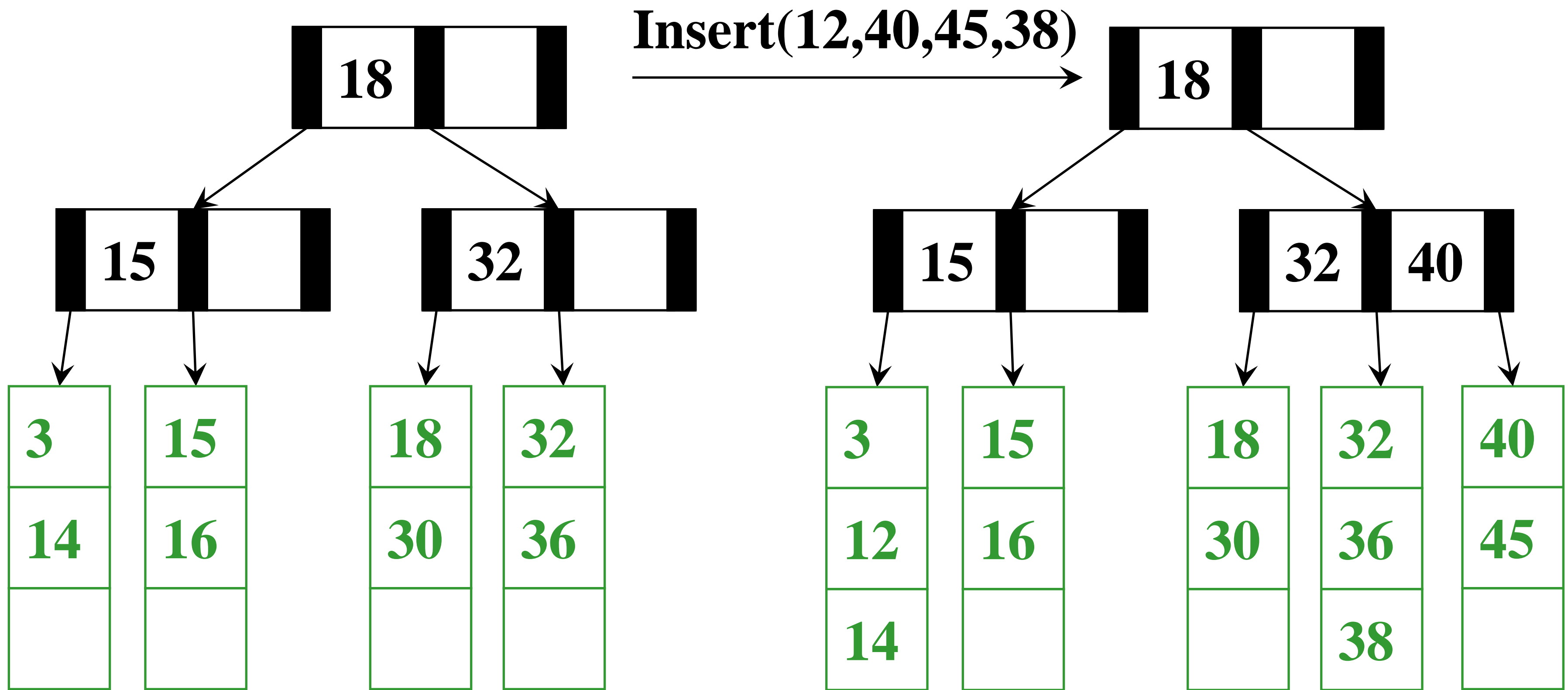
What now?





Split the internal node (in this case, the root)

`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'

`M = 3` `L = 3`

Insertion Algorithm

1. Traverse from the root to the proper leaf. Insert the data in its leaf in sorted order

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

Insertion algorithm continued

3. If an internal node has M+1 children, *overflow!*
   - Split the node into two nodes
     - Original node with $\lceil (M+1)/2 \rceil$ children
     - New node with $\lfloor (M+1)/2 \rfloor$ children
   - 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

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: $O(M \log_M n)$

Worst-case for insert: $O(L + M \log_M n)$

But it’s not that bad:

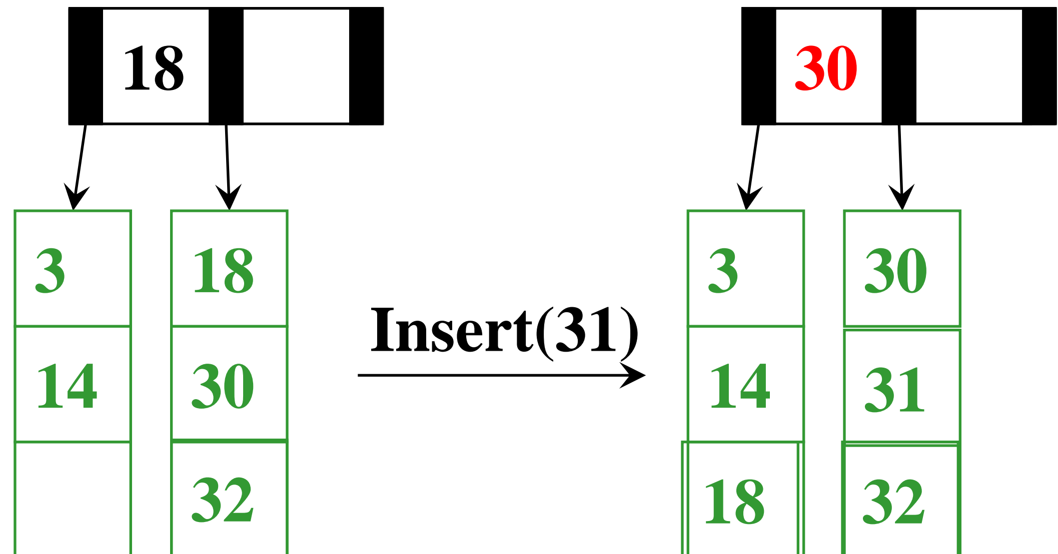
- Splits are not that common (have to fill up nodes)
- Splitting the root is extremely rare
- Remember disk accesses were the name of the game: $O(\log_M n)$

Another option (we won't use)

- Adoption
  - When leaf gains L+1 items, instead of splitting, try to put one 'up for adoption'
    - Check neighboring leaves; if they have space, they can take it
    - If not, will have to split anyway
  - Doesn't change worst-case asymptotic run-time
  - Can also use for internal node; pass off child pointers

Adoption example

- Adoption
  - If overflow, then try to pass on to neighbor
  - If no neighbor has space, split



Same example, with splitting

- For this class, we'll stick with splitting
  - No adoption
  - But good to be aware of alternatives

Splitting