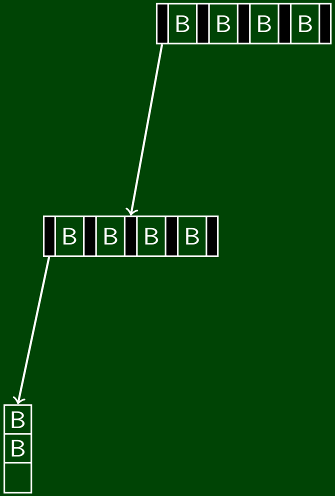
CSE 332

Data Structures and Parallelism

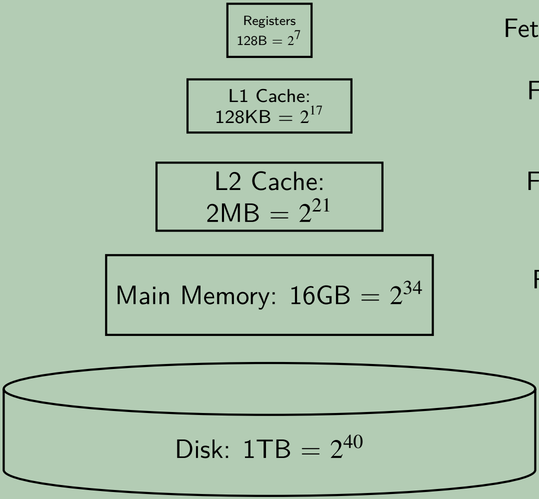
B-Trees



Outline

A New Model?

We've been assuming that **all memory accesses** are the same. In practice, this isn't true. The memory hierarchy looks something like this:



Fetch from Register: "free"

Fetch from L1 Cache:
0.5 nanoseconds

Fetch from L2 Cache:
7 nanoseconds

Fetch from Main Memory:
100 nanoseconds

Fetch from Disk:
8,000,000 nanoseconds

The take-away is that **disk accesses** are very expensive.

A New Model?

Why do we care how the machine works?

Big-Oh is just an abstraction that says "all memory fetches are equal"... but in practice, some memory fetches are more equal than others. (**The disk is prohibitively slow**.)

AVL Trees: Big-Oh vs. Practice

We've seen that AVL Trees are $\mathcal{O}(\lg n)$ which is great, but what if we account for disk accesses?
Consider an AVL Tree of height **40** where each node is b bytes.

- How many nodes in the tree? $\lg n = 40 \to n = 2^{40}$. So, we need about

  b **terabytes**

  for the tree. This means **an overwhelming majority is on the disk**.
- How many disk accesses does a `find` take? It could take none (**3 nanoseconds**) or it could take 40 (**0.3 seconds**). That's a difference of:

  100,000,000

If the data structure is mostly on disk, yes, we still need a data structure that is $\mathcal{O}(\lg n)$, but **it's not enough anymore**!

Problem

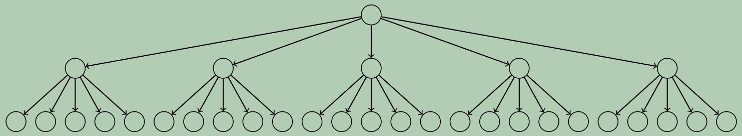
A dictionary with so much data most of it is on disk

Goal

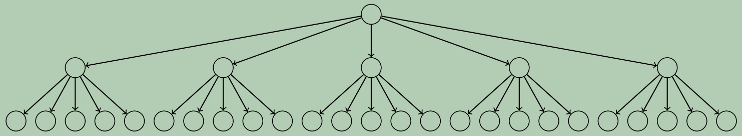
A balanced tree (logarithmic height) that is even shallower than AVL trees so that we can minimize disk accesses and exploit disk-block size

The Idea

Increase the branching factor of our tree



M-ary Search Tree



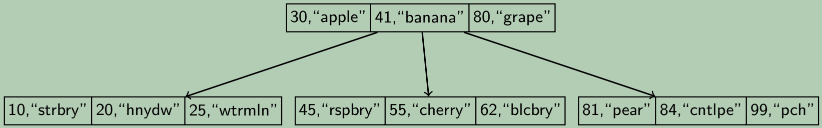
Like a binary tree, but with M branches instead of two.

M-ary Search Tree Properties

- Height (if balanced)? $\mathcal{O}(\log_M(n))$
- Ordering Property?
  - Binary Tree: smaller on the left, larger on the right
  - M-ary Tree: split the range into M equal sized groups
- Runtime of `find` (if balanced)? $\mathcal{O}(h \lg M) = \mathcal{O}(\log_M(n) \lg M)$
  - h possible nodes to visit: $\log_M(n)$
  - **Binary Search** on each node: $\lg M$

Good start, but. . .

M-ary Search Tree Example?



Some Questions

- What should the order property be?
- How would re-balancing work? **We DON’T want to do more disk accesses!**

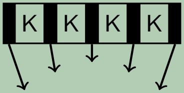
Some Thoughts

- We will have to load the **values** (e.g., fruits) for all the internal nodes. This is very wasteful!
- Usually we are just “passing through” a node on the way to the value we are actually looking for.

Two Types of Nodes

Internal Nodes

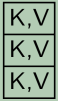
("sign posts")



An internal node has $M-1$ **sorted** keys and M pointers to children

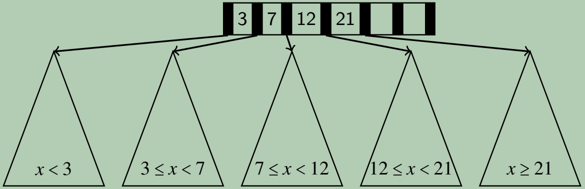
Leaf Nodes

("real data")



A leaf node has L **sorted** key/value pairs

B-Tree Order Property



Subtree between a and b contains all data x where $a \leq x < b$

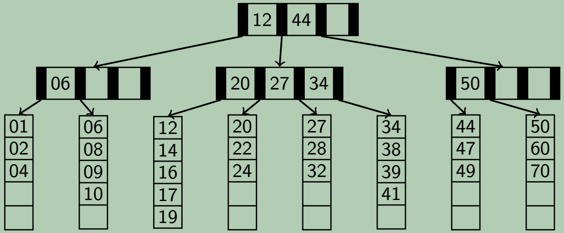
First, choose $M > 2$ and any L. (Here $M = 4, L = 5$.)

Very Few Nodes

If $n \leq L$, the ROOT is a LEAF:

Otherwise, the root must have between 2 and M children

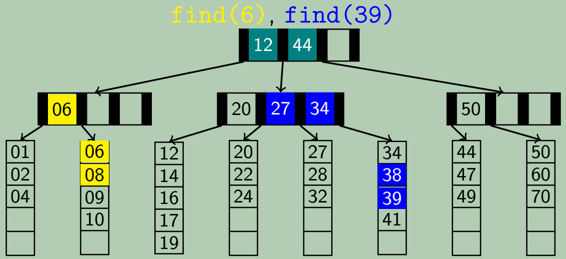
B-Tree Example



Internal Nodes must have between $\left\lceil \frac{M}{2} \right\rceil$ and M children (i.e., half full).

Leaf Nodes must have between $\left\lceil \frac{L}{2} \right\rceil$ and L children (i.e., half full).

Find



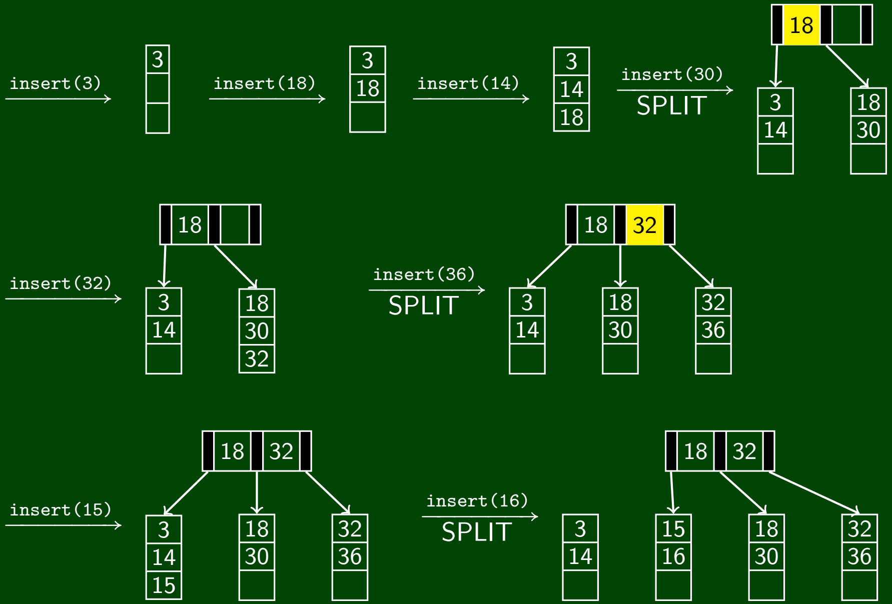
Balanced Enough!

Let $M > 2$. Since all nodes are at least half full (ignoring the root), we have:

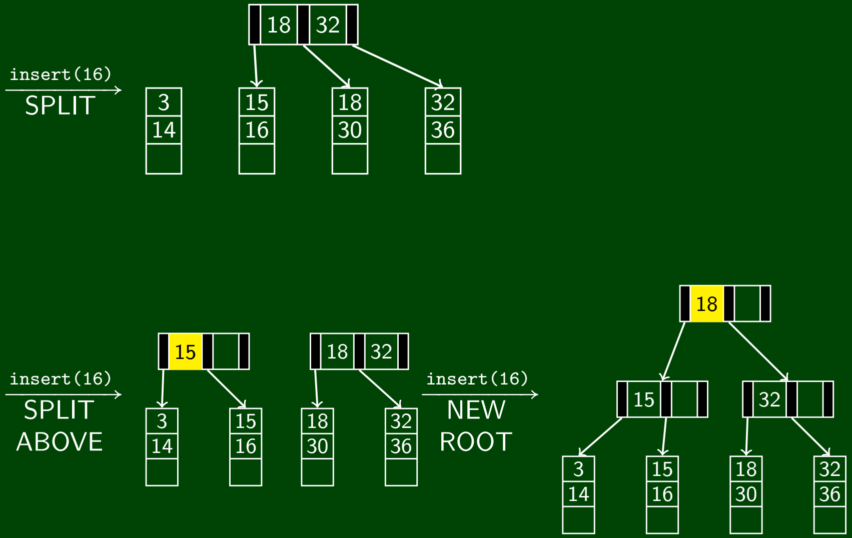
$$2\left\lceil \frac{M}{2} \right\rceil^{h-1} \text{ leaves, and each leaf has at least } \left\lceil \frac{L}{2} \right\rceil \text{ data items}$$

So, $n \geq 2\left\lceil \frac{M}{2} \right\rceil^{h-1} \times \left\lceil \frac{L}{2} \right\rceil$. So, the height h is logarithmic in the number of data items n.

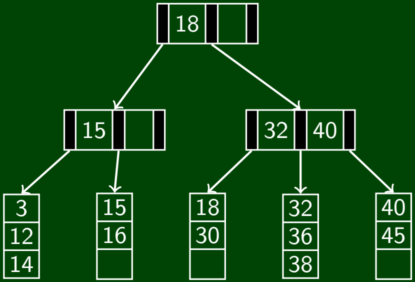
B-Tree Insertion



B-Tree Insertion (Continued)



`insert(12)`, `insert(40)`, `insert(45)`, `insert(38)`



Always fill the "signpost" with the smallest value to my right!

Insertion Algorithm

- Insert the data in the correct leaf **in sorted order**.

- If the leaf has $L+1$ items, overflow:
  - Split the leaf into two new nodes:
    - Original leaf with $\left\lceil \frac{L+1}{2} \right\rceil$ smaller items
    - New leaf with $\left\lceil \frac{L}{2} \right\rceil$ larger items
  - Attach the new child to the parent
  - Add the new key to the parent in sorted order

- Recursively continue overflowing if necessary. Noting that on the internal nodes we split using M instead of L.

- In the case where the **root** overflows, make a new root.

How Efficient is Insert?

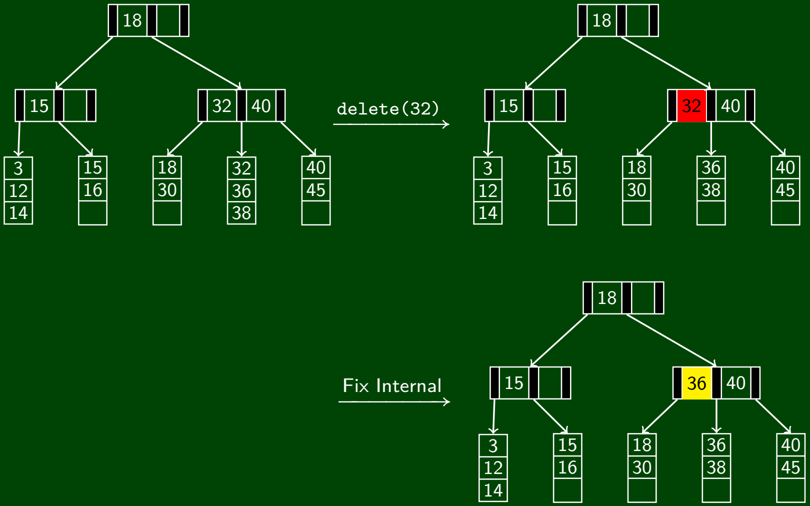
- Find the correct leaf: $\mathcal{O}(\lg(M)\log_M(n))$
- Insert in the leaf: $\mathcal{O}(L)$
- Split leaf: $\mathcal{O}(L)$
- Split parents all the way up to the root: $\mathcal{O}(M\log_M(n))$

In total, this gives us $\mathcal{O}(L+M\log_M(n))$.

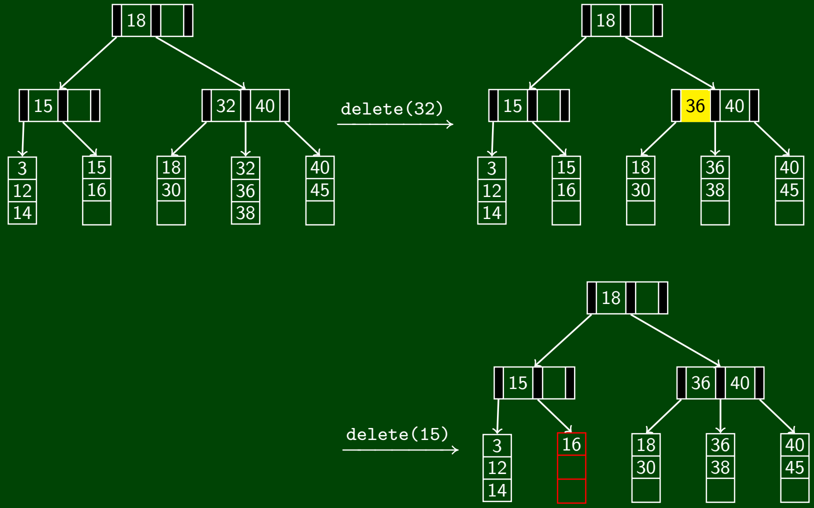
But It's Actually Pretty Good!

- Splits are very uncommon (think amortized analysis)
- Splitting the root almost never happens
- We're significantly more concerned about disk accesses than anything else: $\mathcal{O}(\log_M(n))$

B-Tree Deletion

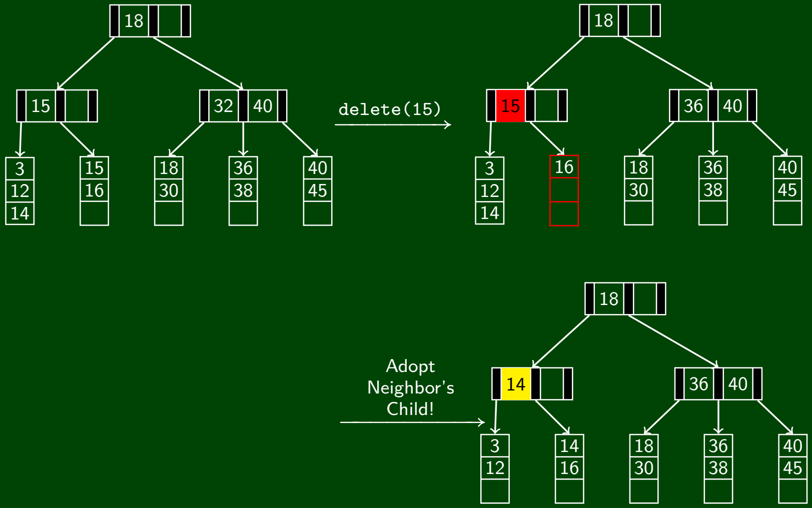


B-Tree Deletion (Continued)

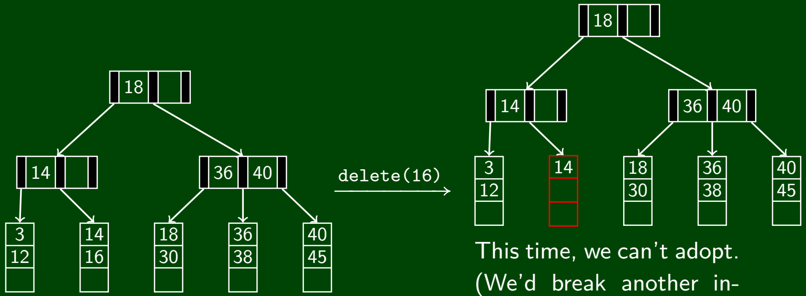


This breaks our invariant. Leaves must have more than one node!

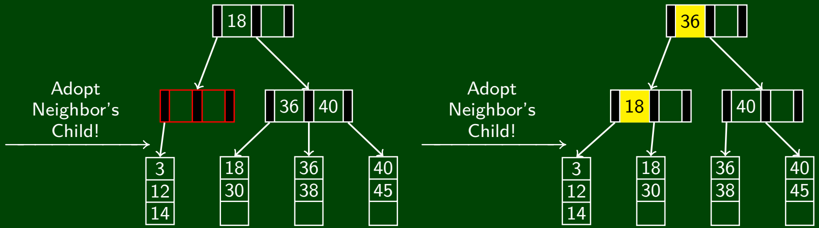
B-Tree Deletion (Continued)



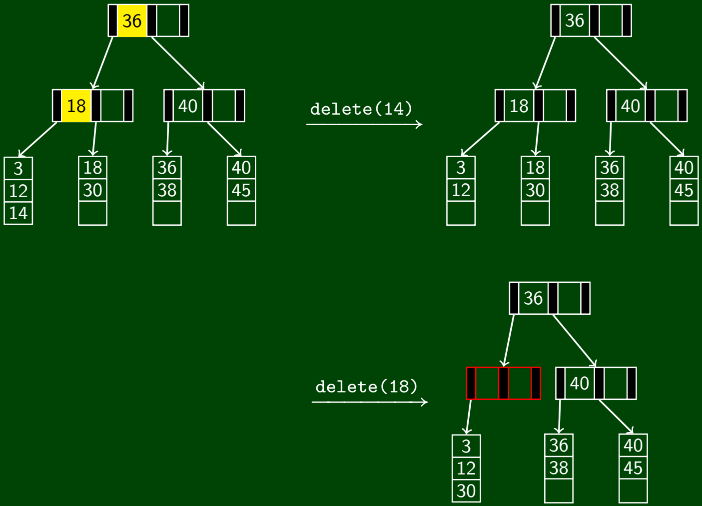
B-Tree Deletion (Continued)



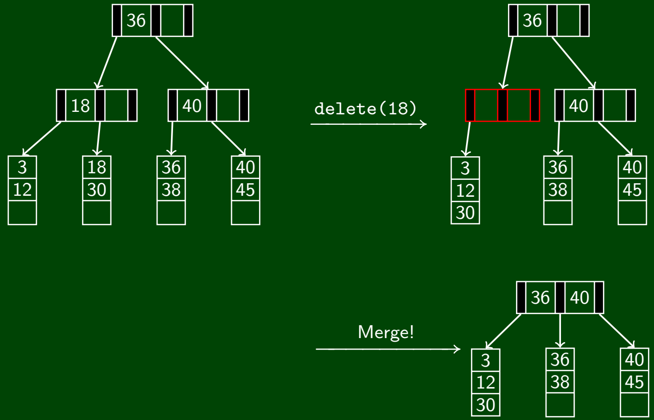
This time, we can't adopt. (We'd break another invariant.) The solution is to adopt **recursively**.



B-Tree Deletion (Continued)



B-Tree Deletion (Continued)



- Remove the data from correct leaf.
- If the leaf has $\left\lceil \frac{L}{2} \right\rceil - 1$ items, underflow:
  - If a neighbor has more than $\left\lceil \frac{L}{2} \right\rceil$, adopt one!
  - Otherwise, **merge** with a neighbor (parent will now have one fewer node)
- Recursively continue underflowing if necessary. Noting that on the internal nodes we split using M instead of L.
- If we merge all the way up to the root and the root went from $2 \to 1$ children, then delete the root and make the child the root.

How Efficient is Delete?

- Find the correct leaf: $\mathcal{O}(\lg(M)\log_M(n))$
- Remove from the leaf: $\mathcal{O}(L)$
- Adopt/Merge with neighbor: $\mathcal{O}(L)$
- Merge parents all the way up to the root: $\mathcal{O}(M\log_M(n))$

In total, this gives us $\mathcal{O}(L+M\log_M(n))$.

But It's Actually Pretty Good!

- Merges are very uncommon (think amortized analysis)
- We're significantly more concerned about disk accesses than anything else: $\mathcal{O}(\log_M(n))$

What makes B-Trees so disk friendly?

- Many keys stored in one internal node: all brought into memory in one disk access
- Makes the binary search over $M-1$ keys totally worth it (insignificant compared to disk access times)
- Internal nodes contain only keys (it's a waste to load all the values)

We take advantage of the choice of M and L to ensure good behavior!

We want each of M and L to fit as best as possible in the **page size**.

Say we know the following:

- 1 page on disk is p bytes
- Keys are k bytes
- Pointers are t bytes
- Key/Value pairs are v bytes

Then, we should choose the following:

- $p \geq M \times (\text{size of a pointer}) + (M-1) \times (\text{size of a key}) = Mt + (M-1)k$.
  So, $M = \left\lfloor \frac{p+k}{t+k} \right\rfloor$.
- $p \geq L \times v$. So, $L = \left\lfloor \frac{p}{v} \right\rfloor$.

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