Our goal

- Problem: A dictionary with so much data most of it is on disk
- Desire: A balanced tree (logarithmic height) that is even shallower than AVL trees so that we can minimize disk accesses and exploit disk-block size
- A key idea: Increase the branching factor of our tree

**M-ary Search Tree**

- Build some sort of search tree with branching factor $M$:
  - Have an array of sorted children \( \text{Node[]} \)
  - Choose $M$ to fit snugly into a disk block (1 access for array)

Perfect tree of height $h$ has \( (M^{h+1}-1)/(M-1) \) nodes (textbook, page 4)

# hops for \texttt{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

Runtime of \texttt{find} if balanced: \( O(\log_M n \log_M n) \) (binary search children)

**Problems with M-ary search trees**

- What should the order property be?
- How would you rebalance (ideally without more disk accesses)?
- Any “useful” data at the internal nodes takes up some disk-block space without being used by finds moving past it

So let’s use the branching-factor idea, but for a different kind of balanced tree
- Not a binary search tree
- But still logarithmic height for any $M > 2$
**B+ Trees (we and the book say “B Trees”)**

- Each internal node has room for up to $M-1$ keys and $M$ children
  - No other data; all data at the leaves!
- **Order property:**
  - Subtree between keys $x$ and $y$ contains only data that is $\geq x$ and $< y$ (notice the $\geq$)
- Leaf nodes have up to $L$ sorted data items

- As usual, we'll ignore the "along for the ride" data in our examples
  - Remember no data at non-leaves

**Find**

- This is a new kind of tree
  - We are used to data at internal nodes
- But **find** is still an easy root-to-leaf recursive algorithm
  - At each internal node do binary search on the $\leq M-1$ keys
  - At the leaf do binary search on the $\leq L$ 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 (very strange case)
  - 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; picked based on disk-block size)

**Example**

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 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...

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

B-Tree vs. AVL Tree

Suppose we have 100,000,000 items

- Maximum height of AVL tree?
  - Recall $S(h) = 1 + S(h-1) + S(h-2)$
  - lecture7.xlsx reports: 47

- Maximum height of B tree with $M=128$ and $L=64$?
  - Recall $2 + \left\lceil \frac{M}{2} \right\rceil^{h-1} \left\lceil \frac{L}{2} \right\rceil$
  - lecture9.xlsx reports: 5 (and 4 is more likely)
  - Also not difficult to compute via algebra

Disk Friendliness

What makes B trees so disk friendly?

- Many keys stored in one node
  - All brought into memory in one disk access
  - Pick $M$ wisely. Example: block=1KB, then $M=128$
  - Makes the binary search over $M-1$ keys totally worth it

- Internal nodes contain only keys
  - Any find wants only one data item
  - 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

The empty B-Tree

\[ M = 3 \quad L = 3 \]

Building a B-Tree (insertions)

\[ \begin{array}{c}
\text{Insert(3)} \\
\text{Insert(18)} \\
\text{Insert(14)}
\end{array} \]

\[ M = 3 \quad L = 3 \]

\[ \begin{array}{c}
\text{Insert(30)} \\
\text{Insert(32)} \\
\text{Insert(36)}
\end{array} \]

\[ M = 3 \quad L = 3 \]

\[ \text{Insert(15)} \]
Insertion Algorithm

1. Insert the data in its leaf in sorted order

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

3. If step (2) caused the parent to have \( M+1 \) children, **overflow!**
   - ...
**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: $O(M \log_M n)$

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

But it's not that bad:
- Splits are not that common (have to fill up nodes)
- Remember disk accesses were the name of the game:
  $O(\log_M n)$