B-Trees
(4.7 in Weiss)

M-ary Search Tree

- Maximum branching factor of \( M \)
- Tree with \( N \) values has height =

\[ \text{# disk accesses for find:} \]

Runtime of find:

Solution: B-Trees

- specialized \( M \)-ary search trees
- Each node has (up to) \( M-1 \) keys:
  - subtree between two keys \( x \) and \( y \) contains leaves with values \( v \) such that \( x \leq v < y \)
- Pick branching factor \( M \) such that each node takes one full {page, block} of memory

B-Trees

What makes them disk-friendly?

1. Many keys stored in a node
   - All brought to memory/cache in one access!
2. Internal nodes contain only keys;
   Only leaf nodes contain keys and actual data
   - The tree structure can be loaded into memory irrespective of data object size
   - Data actually resides in disk

B-Tree: Example

B-Tree with \( M = 4 \) (# pointers in internal node)
and \( L = 4 \) (# data items in Leaf)

Data objects, that I’ll ignore in slides

Note: All leaves at the same depth!

B-Tree Properties

- Data is stored at the leaves
- All leaves are at the same depth and contain between \( \lceil L/2 \rfloor \) and \( L \) data items
- Internal nodes store up to \( M-1 \) keys
- Internal nodes have between \( \lceil M/2 \rfloor \) and \( M \) children
- Root (special case) has between 2 and \( M \) children (or root could be a leaf)

†These are technically B*-Trees
Example, Again

B-Tree with $M = 4$
and $L = 4$

(Only showing keys, but leaves also have data!)

B-trees vs. AVL trees

Suppose we have 100 million items (100,000,000):

• Depth of AVL Tree
• Depth of B+ Tree with $M = 128$, $L = 64$

Building a B-Tree

The empty B-Tree
$M = 3 \ L = 2$

Insert(3)

1

Insert(14)

3 14

Now, Insert(1)?

Overflowing leaves

Insert(59)

1 15 14

Insert(26)

1 15 14 26

1 15 14 26

Too many keys in a leaf!

So, split the leaf.

And add a new child

Propagating Splits

Insert(5)

1 13 14 26 59

Add new child

Split the leaf, but no space in parent!

Create a new root

So, split the node.
Insertion Algorithm

1. Insert the key in its leaf
2. If the leaf ends up with L+1 items, overflow!
   - Split the leaf into two nodes:
     • original with \( \lceil (L+1)/2 \rceil \) items
     • new one with \( \lfloor (L+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
3. If an internal node ends up with \( M+1 \) items, overflow!
   - Split the node into two nodes:
     • original with \( \lceil (M+1)/2 \rceil \) items
     • new one with \( \lfloor (M+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
4. Split an overflowed root in two and hang the new nodes under a new root
   This makes the tree deeper!

After More Routine Inserts

Deletion

1. Delete item from leaf
2. Update keys of ancestors if necessary

Deletion and Adoption

1. Delete(59)
2. Delete(3)

Does Adoption Always Work?

- What if the sibling doesn’t have enough for you to borrow from?
  - e.g. you have \( \lfloor L/2 \rfloor - 1 \) and sibling has \( \lceil L/2 \rceil \)?

Deletion and Merging

1. Delete(3)
2. Delete(59)

But now an internal node has too few subtrees!

So, delete the leaf

A leaf has too few keys!

And no sibling with surplus!
Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than \( \lceil \frac{L}{2} \rceil \) items, underflow!
   - Adopt data from a sibling; update the parent
   - If adopting won’t work, delete node and merge with neighbor
     - If the parent ends up with fewer than \( \lceil \frac{M}{2} \rceil \) items, underflow!

Deletion Slide Two

3. If an internal node ends up with fewer than \( \lceil \frac{M}{2} \rceil \) items, underflow!
   - Adopt from a neighbor; update the parent
   - If adoption won’t work, merge with neighbor
     - If the parent ends up with fewer than \( \lceil \frac{M}{2} \rceil \) items, underflow!

4. If the root ends up with only one child, make the child the new root of the tree
   This reduces the height of the tree!
Thinking about B-Trees

- B-Tree **insertion** can cause (expensive) splitting and propagation
- B-Tree **deletion** can cause (cheap) adoption or (expensive) deletion, merging and propagation
- Propagation is rare if \( M \) and \( L \) are large (Why?)
- If \( M = L = 128 \), then a B-Tree of height 4 will store at least 30,000,000 items

Tree Names You Might Encounter

FYI:
- B-Trees with \( M = 3, L = x \) are called **2-3 trees**
  - Nodes can have 2 or 3 pointers
- B-Trees with \( M = 4, L = x \) are called **2-3-4 trees**
  - Nodes can have 2, 3, or 4 pointers

Determining \( M \) and \( L \) for a B-Tree

1 Page on disk = 1 KByte
Key = 8 bytes, Pointer = 4 bytes
Data = 256 bytes per record (includes key)

\[
M = \\
L = \\
\]