B-Trees
(4.7 in Weiss)

CSE 373
Data Structures & Algorithms
Ruth Anderson

Trees so far

• BST
• AVL

CPU
(has registers)

Cache

SRAM

8KB - 4MB

Main Memory

DRAM

up to 10GB

Disk

many GB

Time to access:
1 ns per instruction

Cache

2-10 ns

Main Memory

40-100 ns

Disk

a few milliseconds
(5-10 Million ns)

M-ary Search Tree

• Maximum branching factor of M
• Complete tree has height =

# 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

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 \( \lceil L/2 \rceil \) and \( L \) data items
- Internal nodes store up to \( M-1 \) keys
- Internal nodes have between \( \lceil M/2 \rceil \) 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

Now, Insert(1)?

---

**Splitting the Root**

Too many keys in a leaf!

And create a new root

So, split the leaf.
Overflowing leaves

Too many keys in a leaf!

Insert(59)

And add a new child

Insert(26)

So, split the leaf.

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 \( \left\lceil \frac{(L+1)}{2} \right\rceil \) items
     - new one with \( \left\lfloor \frac{(L+1)}{2} \right\rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!

This makes the tree deeper!

After More Routine Inserts

Insert(89)

Insert(79)

Deletion

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

What could go wrong?

Deletion and Adoption

A leaf has too few keys!

Delete(59)

So, borrow from a sibling
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

A leaf has too few keys!

And no sibling with surplus!

But now an internal node has too few subtrees!

So, delete the leaf

A leaf has too few keys!

Delete(3)

M = 3 \ L = 2

Deletion with Propagation (More Adoption)

A leaf has too few keys!

And no sibling with surplus!

So, delete the leaf

Deletion with Propagation (More Adoption)

A Bit More Adoption

A leaf has too few keys!

Delete(1)

(adopt a sibling)

M = 3 \ L = 2

Pulling out the Root

A leaf has too few keys!

And no sibling with surplus!

But now the root has just one subtree!

A node has too few subtrees and no neighbor with surplus!

Delete the node

M = 3 \ L = 2

Pulling out the Root (continued)

The root has just one subtree!

Simply make the one child the new root!
Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than \( \left\lceil \frac{L}{2} \right\rceil \) 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 \( \left\lceil \frac{M}{2} \right\rceil \) items, underflow!

Deletion Slide Two

3. If an internal node ends up with fewer than \( \left\lceil \frac{M}{2} \right\rceil \) 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 \( \left\lceil \frac{M}{2} \right\rceil \) 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)

\( M = \)

\( L = \)