B-Trees
Section 4.7 in Weiss

CSE 326
Data Structures
Ruth Anderson

Today’s Outline
• Announcements
  – Project 2B due Wednesday, 2/10 at 11pm
  – Midterms returned and discussed in section Thurs
  – Written Homework #4 due Friday 2/12

• Today’s Topics:
  – B-Trees

Trees so far
• BST

• AVL

• Splay

Trees

CPU
(has registers)

SRAM
8KB - 4MB

Cache

Main Memory

DRAM
up to 10GB

Disk

many GB

Time to access:
1 ns per instruction

2-10 ns

40-100 ns

a few milliseconds
(5-10 Million ns)

M-ary Search Tree

• Maximum branching factor of $M$
• Complete tree has height $= \log_M n$

# 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
   - Only retrieve data that we need

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

B-Tree Properties

– Data is stored at the leaves
– All leaves are at the same depth and contains 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)

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)

$3$

Insert(14)

$3 4$

Now, Insert(1)?
Splitting the Root

Too many keys in a leaf!

Insert(1)

And create a new root

So, split the leaf.

Overflowing leaves

Too many keys in a leaf!

Insert(59)

Insert(26)

So, split the leaf

And add a new child

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 \( \lceil (L+1)/2 \rceil \) 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 \( \lceil (M+1)/2 \rceil \) 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!

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(5)

So, borrow from a sibling

Delete(5)

Does Adoption Always Work?

• What if the sibling doesn’t have enough for you to borrow from?
  e.g. you have ⌈L/2⌉-1 and sibling has ⌈L/2⌉?

Deletion and Merging

A leaf has too few keys!

Delete(3)

And no sibling with surplus!

Delete(3)

So, delete the leaf

But now an internal node has too few subtrees!

A Bit More Adoption

Delete(1) (adopt a sibling)

More Adoption

A leaf has too few keys!

Delete(26)

So, delete the leaf; merge

Pulling out the Root

A leaf has too few keys!

Delete(26)

So, delete the leaf; merge

But now the root has just one subtree!

A node has too few subtrees and no neighbor with surplus!
**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 $\lceil L/2 \rceil$ 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 $\lceil M/2 \rceil$ items, **underflow**!

---

**Deletion Slide Two**

3. If an internal node ends up with fewer than $\lceil M/2 \rceil$ 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 $\lceil M/2 \rceil$ items, **underflow**!
   - This reduces the height of the tree!

4. If the root ends up with only one child, make the child the new root 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 keys
- B-Trees with $M = 4$, $L = x$ are called 2-3-4 trees
  - Nodes can have 2, 3, or 4 keys

---

**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)

\[ M = \_ \_ \]

\[ L = \_ \_ \]