CSE 373: Data Structures and Algorithms

Lecture 25: B-Trees
Cycles to access:

- Registers: 1
- Cache: tens
- Main memory: hundreds
- Disk: millions
Hard Disks

• Large amount of storage but slow access

• Identifying a page takes a long time
  – Pays to read or write data in pages (i.e. blocks) of 0.5 – 8 KB in size
Algorithm Analysis

• Running time of disk-based data structures measured in terms of
  – computing time (CPU)
  – number of disk accesses
    • sequential reads
    • random reads

• Regular main-memory algorithms that work one data element at a time can not be "ported" to secondary storage in a straight forward way
Principles

• Almost all of our data structure is on disk.

• Every time we access a node in the tree it amounts to a random disk access.

• How can we address this problem?
$M$-ary Search Tree

- Suppose we devised a search tree with branching factor $M$:

- $M - 1$ keys needed to decide branch to take

- Complete tree has height: $\Theta(\log_M n)$

- # Nodes accessed for search: $\Theta(\log_M n)$
B-Trees

- Internal nodes store (up to) $M - 1$ keys

- Order property:
  - subtree between two keys $x$ and $y$ contain leaves with values $v$ such that $x \leq v < y$
  - Note the “≤”

- Leaf nodes contain up to $L$ sorted values/records.

$M = 7$
Disk Friendliness

What makes B-trees disk-friendly?

1. Many keys stored in a node
   • Each node is one disk page/block.
   • All brought to memory/cache in one disk access.

2. Internal nodes contain only keys;
   Only leaf nodes contain keys and actual data
   • Much of tree structure can be loaded into memory
     irrespective of data object size
   • Data actually resides in disk

What is limiting you from increasing the number of keys stored in each node?

Exercise: If disk block is 4000 bytes, key size is 20 bytes, pointer size is 4 bytes, and
data/value size is 200 bytes, what should M and L be for our B-Tree?
B-Tree Structure Properties

• Root (special case)
  – has between 2 and $M$ children (or could be a leaf)

• Internal nodes
  – store up to $M-1$ keys
  – have between $\lfloor M/2 \rfloor$ and $M$ children

• Leaf nodes
  – where data is stored
  – contain between $\lfloor L/2 \rfloor$ and $L$ data items

The tree is **perfectly balanced**!
B-Tree: Example

B-Tree with $M = 4$ (# pointers in internal node)
and $L = 5$  (# data items in leaf)

Data objects...
which we’ll ignore in slides

Definition for later: “neighbor” is the next sibling to the left or right.
B-trees vs. AVL trees

Suppose we have $n = 10^9$ data items:

- Depth of AVL Tree: $\log_2 10^9 = 30$

- Depth of B-Tree with $M = 256$, $L = 256$: $\log_{128} 10^9 = 4.3$
Building a B-Tree with Insertions

The empty B-Tree

$M = 3 \quad L = 3$
$M = 3 \quad L = 3$
$M = 3 \quad L = 3$
Insert(16)
\[ M = 3 \quad L = 3 \]

Insert(12, 40, 45, 38)
Insertion Algorithm: The Overflow Step

\[ M = 5 \]
Insertion Algorithm

1. Insert the key in its leaf in sorted order
2. If the leaf ends up with $L+1$ items, **overflow**!
   - Split the leaf into two nodes:
     - $\lceil(L+1)/2\rceil$ smaller keys
     - $\lfloor(L+1)/2\rfloor$ larger keys
   - Add the new child to the parent
3. If an internal node ends up with $M+1$ children, **overflow**!
   - Split the node into two nodes:
     - $\lceil(M+1)/2\rceil$ children with smaller keys
     - $\lfloor(M+1)/2\rfloor$ children with larger keys
   - Add the new child to the parent
   - If the parent ends up with $M+1$ items, **overflow**!
4. If the root ends up with $M+1$ children, split it in two, and create new root with two children

This makes the tree deeper!
And Now for Deletion...

$M = 3 \quad L = 3$
Are we okay?

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

Dang, not half full

Are you using that 14?
Can I borrow it?
$M = 3 \quad L = 3$
$M = 3$  \hspace{1cm} $L = 3$

Are you using that 12?
$M = 3 \quad L = 3$

Are you using the node18/30?
$M = 3 \quad L = 3$
$M = 3 \quad L = 3$
Delete(18)

\[ M = 3 \quad L = 3 \]
$M = 3 \quad L = 3$
\[ M = 3 \quad L = 3 \]
$M = 3 \quad L = 3$
Deletion Algorithm: Rotation Step

This is left rotation. Similarly, right rotation

$M = 5$
Deletion Algorithm: Merging Step

M = 5

Too small

Can’t get smaller

Too small?

K1  K2

K1  K2  K3  K4

K1  K2  K3  K4
Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than \( \lfloor L/2 \rfloor \) items, **underflow**!
   - Try a left rotation
   - If not, try a right rotation
   - If not, merge, then check the parent node for underflow
3. If an internal node ends up with fewer than \(\lceil M/2 \rceil\) children, **underflow**!
   - Try a left rotation
   - If not, try a right rotation
   - If not, merge, then check the parent node for 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!