Splay Tree Summary

- All operations are in amortized $O(\log n)$ time
- Splaying can be done top-down; this may be better because:
  - only one pass
  - no recursion or parent pointers necessary
  - we didn’t cover top-down in class
- Splay trees are very effective search trees
  - Relatively simple
  - No extra fields required
  - Excellent locality properties: frequently accessed keys are cheap to find

Disk vs. Memory

- Disks many times slower than memory:
  - Processor measured in GH = $10^9$ cycles per second
  - Main memory measured in microsec. = $10^6$ per second
  - Disk seek measured in milliseconds = $10^3$ per second
- i.e. ~ 1 million instructions per disk lookup
- Measuring runtime by pointer lookups meaningless if data can’t fit in main memory

Trees on disk

- Each pointer lookup means seeking the disk
- Want as shallow a tree as possible
- Balanced binary tree with N nodes has height ________?
- Balanced M-ary tree with N nodes has height ________?

$M$-ary Search Tree

- Maximum branching factor of $M$
- Complete tree has height $= \log_M N$

# disk accesses for find:

Runtime of find:

Problems with $M$-ary Search Trees

1.

2.

3.
Solution: B-Trees

- B-Trees are specialized M-ary search trees
- Each node has many keys (max M-1)
  - subtree between two keys x and y contains leaves with values $x < v < y$
  - binary search within a node to find correct subtree
- 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

Example

- 1k byte page
- Key 8 bytes, pointer 4 bytes
- $(M-1)8 + 4M = 1024$
  - $12M = 1032$
  - $M = \lceil 1032/12 \rceil = 86$

B-Tree: Example

B-Tree with $M = 4$ (# pointers in internal node)

B-Tree Details

Each (non-leaf) internal node of a B-tree has:
- Between $\lceil M/2 \rceil$ and M children.
- Up to M-1 keys $k_1 < k_2 < \ldots < k_{M-1}$

Keys are ordered so that:
$k_1 < k_2 < \ldots < k_{M-1}$
B-Tree Details

Each leaf node of a B-tree has:
- Between \([M/2]\) and \(M\) keys and pointers.

Keys are ordered so that:
\[ k_1 < k_2 < \ldots < k_{M-1} \]

Keys point to data on other pages.

Properties of B-Trees

Children of each internal node are "between" the items in that node.

Suppose subtree \(T_i\) is the \(i\)-th child of the node:
- all keys in \(T_i\) must be between keys \(k_{i-1}\) and \(k_i\)
  - i.e. \(k_{i-1} \leq T_i < k_i\)
  - \(k_{i-1}\) is the smallest key in \(T_i\)

All keys in first subtree \(T_1 < k_i\)
All keys in last subtree \(T_M \geq k_M\)

Inserting into B-Trees

- Insert \(X\): Do a Find on \(X\) and find appropriate leaf node
  - If leaf node is not full, fill in empty slot with \(X\)
  - E.g. Insert 5
  - If leaf node is full, split leaf node and adjust parents up to root node
    - E.g. Insert 9

Insert Example

- Insert Example

- Insert Example
Deleting From B-Trees

• Delete X: Do a find and remove from leaf
  › Leaf underflows – borrow from a neighbor
  • E.g. 11
  › Leaf underflows and can’t borrow – merge nodes, delete parent
    • E.g. 17

Delete Example

Run Time Analysis of B-Tree Operations

• For a B-Tree of order M
  › Each internal node has up to M-1 keys to search
  › Each internal node has between \([M/2]\) and M children
  › Depth of B-Tree storing N items is \(O(\log_{[M/2]} N)\)
• Example: M = 86
  › \(\log_{43} N = \log_2 N / \log_2 43 = 1.84 \log_2 N\)
  › \(\log_{43} 1,000,000,000 = 5.51\)

Summary of Search Trees

• Problem with Search Trees: Must keep tree balanced to allow fast access to stored items
• AVL trees: Insert/Delete operations keep tree balanced
• Splay trees: Repeated Find operations produce balanced trees on average
• Multi-way search trees (e.g. B-Trees): More than two children
  › per node allows shallow trees; all leaves are at the same depth
  › keeping tree balanced at all times