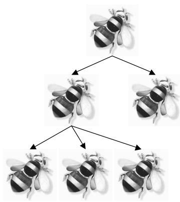
# CSE 326: Data Structures
# Lecture #10
# B-Trees



Henry Kautz
Winter Quarter 2002

## Beyond Binary Trees

- One of the most important applications for search trees is databases
- If the DB is small enough to fit into RAM, almost any scheme for balanced trees is okay

| 1980 | 2002 (WalMart) |
|---|---|
| RAM – 1MB | RAM – 1,000 MB (gigabyte) |
| DB – 100 MB | DB – 1,000,000 MB (terabyte) |

*gap between disk and main memory growing!*

## Time Gap

- For many corporate and scientific databases, the search tree must mostly be on disk
- Accessing disk 200,000 times slower than RAM
- Visiting node = accessing disk
- Even perfectly balance binary trees a disaster!

  $\log_2(10{,}000{,}000) = 24$ disk accesses

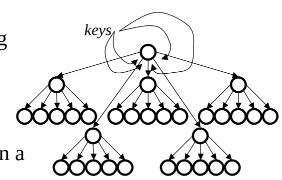
*Goal: Decrease Height of Tree*

## *M*-ary Search Tree

- Maximum branching factor of ***M***
- Complete tree has depth = $\log_M N$
- Each internal node in a complete tree has

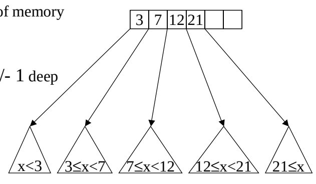
  ***M* - 1** keys

  runtime:




## B-Trees

- B-Trees are balanced *M*-ary search trees
- Each node has many keys
  - internal nodes : between $\lceil M/2 \rceil$ and *M* children (except root), no data – only keys, smallest datum between search keys *x* and *y* equals *x*
  - binary search within a node to find correct subtree
  - each leaf contains between $\lceil L/2 \rceil$ and *L* keys
  - all leaves are at the same depth
  - choose M and L so that each node takes one full {*page*, *block*, *line*} of memory (why?)
- Result:
  - tree is $\log_{\lceil M/2 \rceil} n/(L/2)$ +/- 1 deep




## When Big-O is Not Enough

$\log_{M/2} n/(L/2)$

$= \log_{M/2} n - \log_{M/2} L/2$

$= O(\log_{M/2} n)$ steps per option

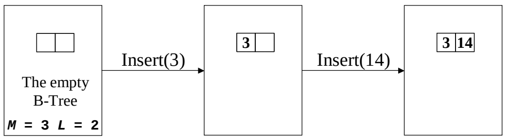
$= O(\log n)$ steps per operation

*Where's the beef?!*

$\lceil \log_2(10{,}000{,}000) \rceil = 24$ disk accesses

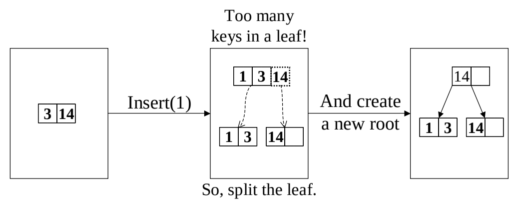
$\lceil \log_{200/2}(10{,}000{,}000/(200/2)) \rceil = \lceil \log_{100}(100{,}000) \rceil = 3$ accesses

# Making a B-Tree

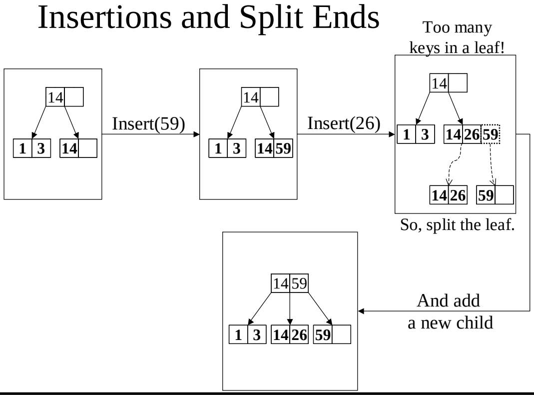



Now, Insert(1)?

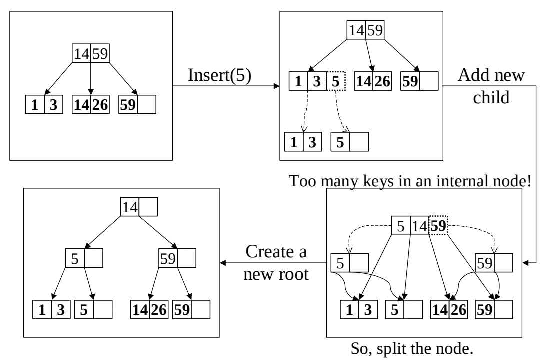
# Splitting the Root




# Insertions and Split Ends




# Propagating Splits

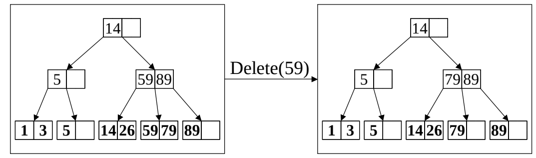



# Insertion in Boring Text

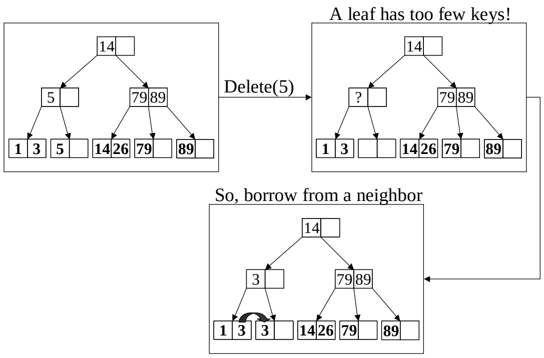
- Insert the key in its leaf
- If the leaf ends up with L+1 items, **overflow**!
  - Split the leaf into two nodes:
    - original with $\lceil (L+1)/2 \rceil$ items
    - new one with $\lfloor (L+1)/2 \rfloor$ items
  - Add the new child to the parent
  - If the parent ends up with $M+1$ items, **overflow**!
- 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**!
- Split an overflowed root in two and hang the new nodes under a new root

This makes the tree deeper!

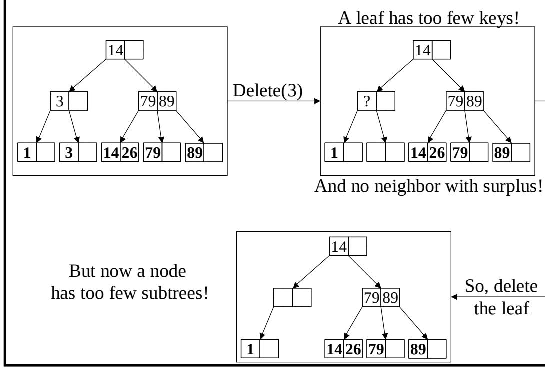
# Deletion



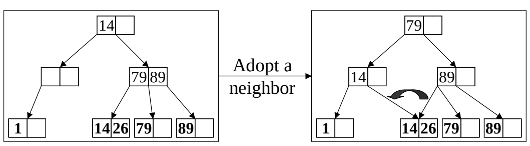
## Deletion and Adoption




## Deletion with Propagation




## Finishing the Propagation (More Adoption)




## Deletion in *Two* Boring Slides of Text

- Remove the key from its leaf
- If the leaf ends up with fewer than $\lceil L/2 \rceil$ items, **underflow**!
  - Adopt data from a neighbor; update the parent
  - If borrowing won't work, delete node and divide keys between neighbors
  - If the parent ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!

Why will dumping keys always work if borrowing doesn't?

## Deletion Slide Two

- If an internal node ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!
  - Adopt subtrees from a neighbor; update the parent
  - If borrowing won't work, delete node and divide subtrees between neighbors
  - If the parent ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!
- 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!

## Run Time Analysis of B-Tree Operations

- For a B-Tree of order M:
  - Depth is $\log_{\lceil M/2 \rceil} n/(L/2)$ +/- 1
- Find: run time in terms of *both* n and M=L is:
  - O(log M) for binary search of each internal node
  - O(log L) = O(log M) for binary search of the leaf node
  - Total is $\leq$ $O((\log_{M/2} n/(M/2))(\log M) + \log M)$

    $= O((\log n/(M/2))/(\log M/2))(\log M))$

    $= O(\log n + \log M)$

## Run Time Analysis of B-Tree Operations

- Insert and Delete: run time in terms of *both* n and M=L is:
  - O(M) for search and split/combine of internal nodes
  - O(L) = O(M) for search and split/combine of leaf nodes
  - Total is $\leq$ $O((\log_{M/2} n/(M/2))M + M)$
    $= O((M/\log M)\log n)$

## A Tree with Any Other Name

FYI:
- B-Trees with **`M = 3`**, **`L = x`** are called 2-3 trees
- B-Trees with **`M = 4`**, **`L = x`** are called 2-3-4 trees

Why would we ever use these?

## Summary

- BST: fast finds, inserts, and deletes O(log *n*) on average (*if* data is random!)
- AVL trees: guaranteed O(log *n*) operations
- B-Trees: also guaranteed O(log *n*), but shallower depth makes them better for disk-based databases
- What would be even better?
  - *How about: O(1) finds and inserts?*

## Coming Up

- Hash Tables
- Another assignment ?!
- Midterm
- Another project?!