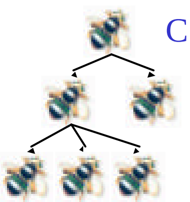


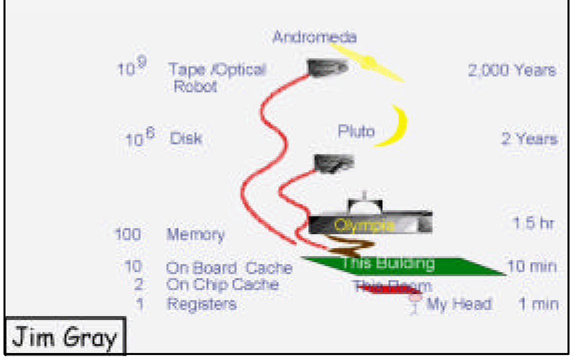
CSE 326: Data Structures
Big, Bad B-Trees

Hannah Tang and Brian Tjaden
Summer Quarter 2002

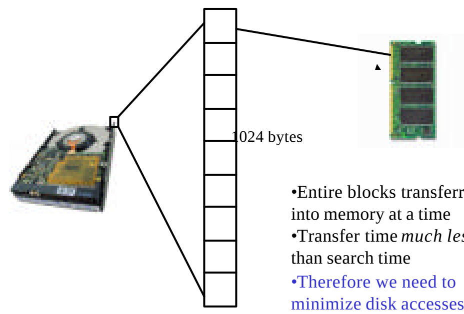
Get out some paper: Charecteristics of some Dictionary ADT Implementations

- (unbalanced) Binary Search Trees
- AVL Trees
- Splay Trees
- Other?

Something We Forgot: Disk Acesses



We Want To Minimize Disk Accesses!

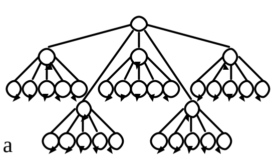


•Entire blocks transferred into memory at a time
•Transfer time *much less* than search time
•Therefore we need to minimize disk accesses!

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

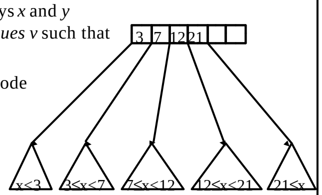
runtime:



Problems with *M*-ary Search Trees

B-Trees

- B-Trees are specialized *M*-ary search trees
- Each node has many keys
  - subtree between two keys *x* and *y* contains leaves with *values* *v* such that $x \leq v < y$
  - binary search within a node to find correct subtree
- Each node takes one full {*page, block, line*} of memory



B-Tree Properties‡

- Properties
  - maximum branching factor of M
  - the root has between 2 and M children *or* at most L keys
  - other internal nodes have between $\lceil M/2 \rceil$ and *M* children
  - internal nodes contain only search keys (no data)
  - smallest datum between search keys *x* and *y* equals *x*
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and *L* keys
  - all leaves are at the same depth

‡These are technically B⁺-Trees

B-Tree Properties

- Properties
  - maximum branching factor of *M*
  - the root has between 2 and *M* children *or* at most *L* keys
  - other internal nodes have between $\lceil M/2 \rceil$ and M children
  - internal nodes contain only *search* keys (no values)
  - All values are stored at the leaves
  - smallest datum between search keys *x* and *y* equals *x*
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and *L* keys
  - all leaves are at the same depth

B-Tree Properties

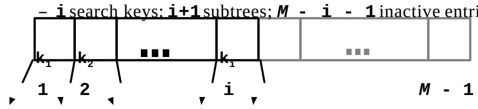
- Properties
  - maximum branching factor of *M*
  - the root has between 2 and *M* children *or* at most *L* keys
  - other internal nodes have between $\lceil M/2 \rceil$ and *M* children
  - internal nodes contain only search keys (no data)
  - smallest datum between search keys *x* and *y* equals *x*
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and L keys
  - all leaves are at the same depth

B-Tree Properties

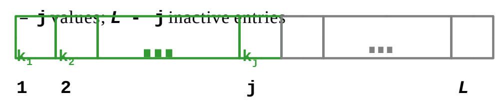
- Properties
  - maximum branching factor of *M*
  - the root has between 2 and *M* children *or* at most *L* keys
  - other internal nodes have between $\lceil M/2 \rceil$ and *M* children
  - internal nodes contain only search keys (no data)
  - smallest datum between search keys *x* and *y* equals *x*
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and *L* keys
  - all leaves are at the same depth
- Result
  - tree is $\Theta(\log_M n)$ deep
  - all operations run in $\Theta(\log_M n)$ time
  - operations pull in about $M/2$ or $L/2$ items at a time

B-Tree Nodes

- Internal node
  - i search keys; $i+1$ subtrees; $M - i - 1$ inactive entries

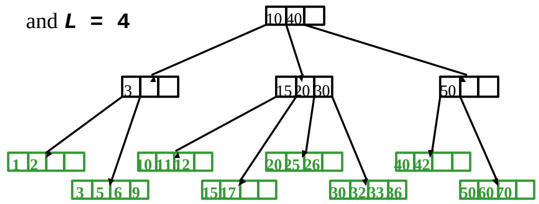


- Leaf
  - j values; $L - j$ inactive entries

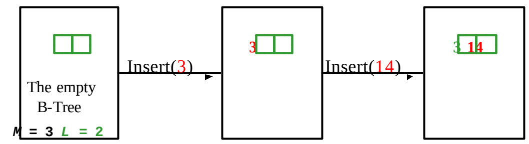


Example

B-Tree with $M = 4$
and $L = 4$

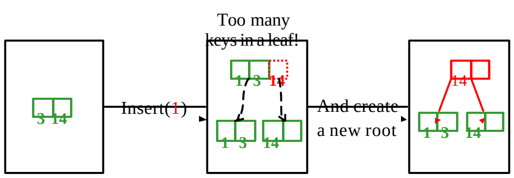


Making a B-Tree



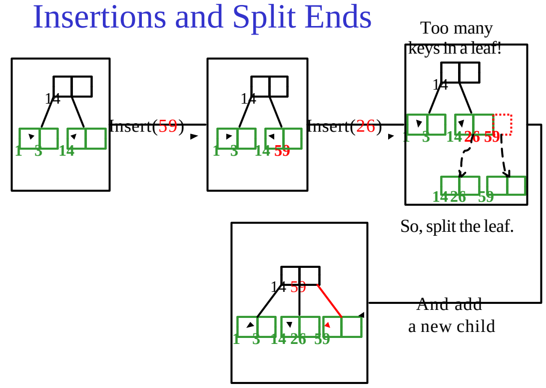
Now, Insert(1)?

Splitting the Root

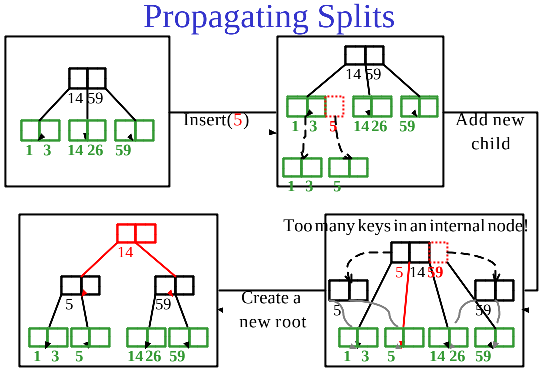


So, split the leaf.

Insertions and Split Ends



Propagating Splits



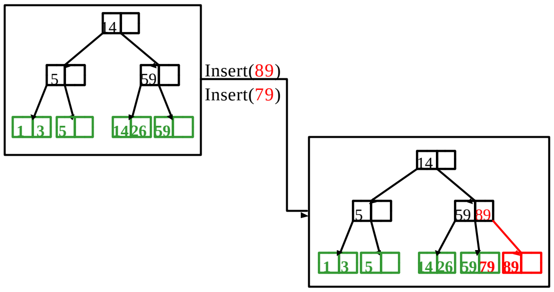
So, split the node.

Insertion in Boring Text

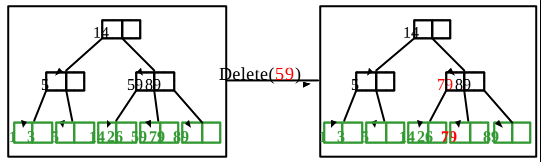
- Insert the key in its leaf
- If the leaf ends up with L+1 items, **overflow**!
  - Split the leaf into two nodes:
    - original with é$(L+1)/2$ù items
    - new one with ë$(L+1)/2$û items
  - Add the new child to the parent
  - If the parent ends up with $M+1$ items, **overflow**!
- If an internal node ends up with M+1 items, **overflow**!
  - Split the node into two nodes:
    - original with é$(M+1)/2$ù items
    - new one with ë$(M+1)/2$û 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!

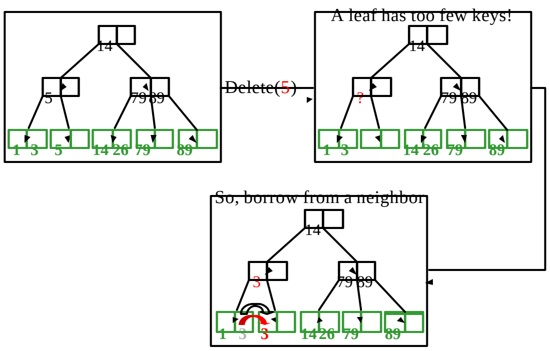
After More Routine Inserts



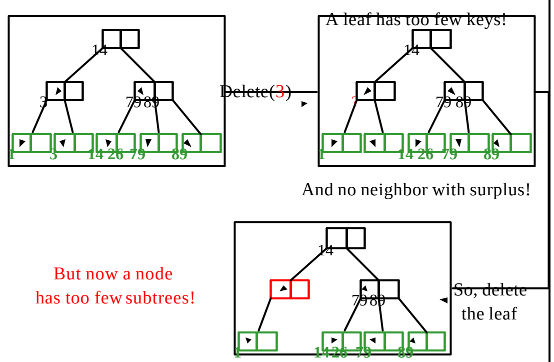
Deletion



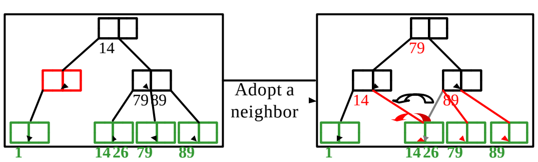
Deletion and Adoption



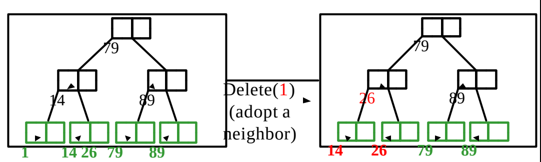
Deletion with Propagation



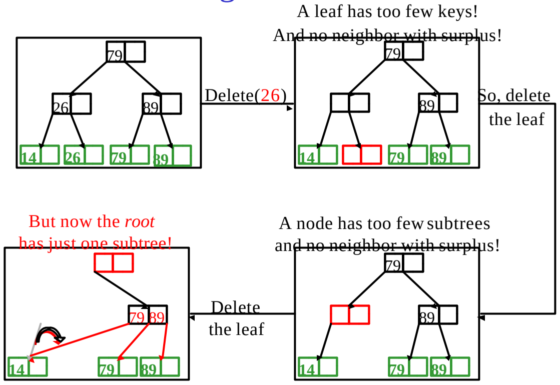
Finishing the Propagation (More Adoption)



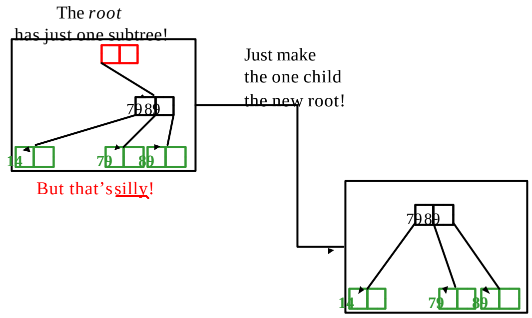
A Bit More Adoption



Pulling out the Root



Pulling out the Root (continued)



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 a 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!

B-trees vs AVL trees

We have a database* with 100 million items (100,000,000):

- Depth of AVL Tree
- Depth of B+ Tree with B = 128, L = 64

** A very simple type of database, called "Berkeley Database" is basically a B+-tree*

Thinking about B-Trees

- B-Tree insertion can cause (expensive) splitting and propagation
- B-Tree deletion can cause (cheap) borrowing or (expensive) deletion and propagation
- Propagation is rare if M and L are large *(Why?)*
- Repeated insertions and deletion can cause thrashing
- If $M = L = 128$, then a B-Tree of height 4 will store at least 30,000,000 items

A Tree with Any Other Name

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

Why would we ever use these?

Another Way

- B-Trees are *hairy* in practice

hairy - Slang. Fraught with difficulties; hazardous: a hairy escape; hairy problems.