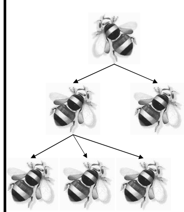
CSE 326: Data Structures

Topic #8: Big, Bad B-Trees



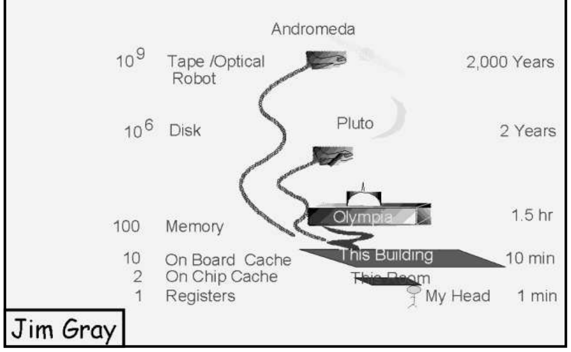
Ashish Sabharwal
Autumn, 2003

Today's Outline

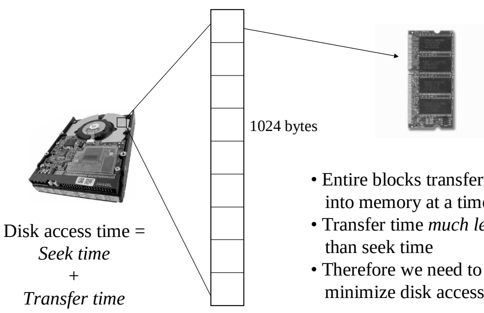
- Admin:
  - Project 2, Phase B code will be ready by **6:00 pm** tonight
  - Due next Monday!
  - "In-progress" checking due this Wed night
  - Remember: README constitutes 30% of project grade
  - Use class email list – ask, answer, share knowledge!
- Finish talking about project 2
- **B-Trees**

2

Something We Forgot: Disk Acesses



We Want To Minimize Disk Accesses!



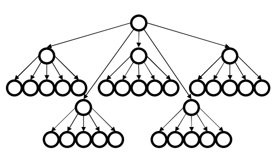
Disk access time =
Seek time
+
Transfer time

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

4

M-ary Search Tree

- Maximum branching factor of ***M***
- Complete tree has height =



\# disk accesses for *find*:

Runtime of *find*:

5

M-ary Search Tree

Subject GRE analogy question:

M-ary Trees are to AVL Trees
as _____________ are to _____________

- Same motivation
- Same idea
- **But** …

6

Problems with *M*-ary Search Trees

1.

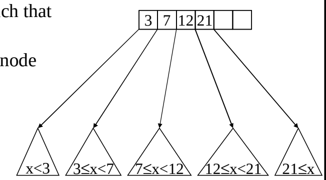
2.

3.

7

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 v* such that $x \le v < y$
  - binary search within a node to find correct subtree
- Each node takes one full {*page*, *block*} of memory



So what's new here??

8

B-Trees

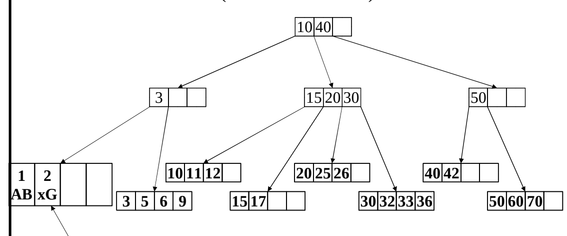
What makes them disk-friendly?

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

9

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

Note: All leaves at the same depth!

10

B-Tree Properties (1) ‡

- maximum branching factor of ***`M`***
- the root has between 2 and ***`M`*** children *or* at most ***`L`*** data items
- other internal nodes have between $\lceil M/2 \rceil$ and *M* children
- internal nodes contain only search keys (no data)
- 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* data items
- all leaves are at the same depth

‡These are technically B^+-Trees

11

B-Tree Properties (2)

- maximum branching factor of *M*
- the root has between 2 and *M* children *or* at most *L* data items
- other internal nodes have between $\lceil M/2 \rceil$ and ***`M`*** children
- internal nodes contain only *search* keys (no data objects)
- All data 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* data items
- all leaves are at the same depth

12

B-Tree Properties (3)

- maximum branching factor of M
- the root has between 2 and M children *or* at most L data items
- other internal nodes have between $\lceil M/2 \rceil$ and M children
- internal nodes contain only search keys (no data)
- 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 data items
- all leaves are at the same depth

B-Tree Properties (4)

- maximum branching factor of M
- the root has between 2 and M children *or* at most L data items
- other internal nodes have between $\lceil M/2 \rceil$ and M children
- internal nodes contain only search keys (no data)
- 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 data items
- all leaves are at the same depth

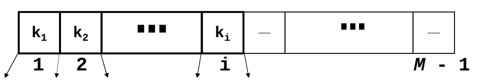
Result

- 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

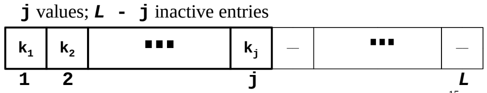
Internal nodes

`i` search keys; `i+1` subtrees; `M - i - 1` inactive entries



Leaf nodes

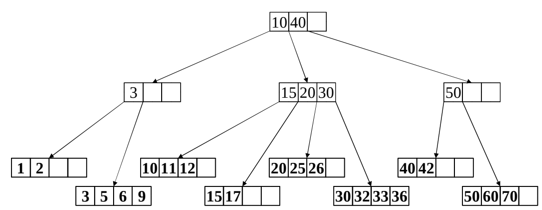
`j` values; `L - j` inactive entries



Example, Again

B-Tree with `M = 4`

and `L = 4`



(Only showing keys, but leaves also have data!)

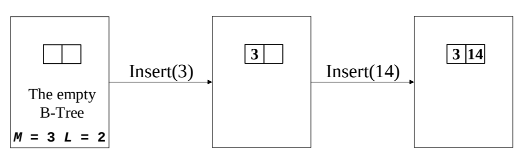
B-trees vs. AVL trees

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

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

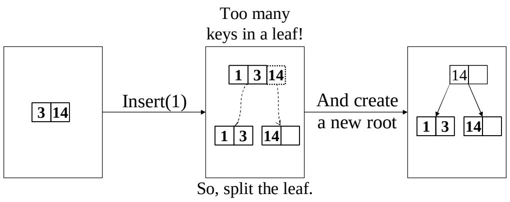
** A very simple type of database, called "Berkeley Database" is basically a B+-tree*

Building a B-Tree

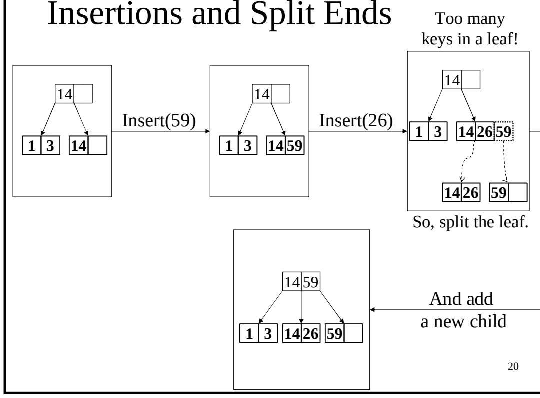


Now, Insert(1)?

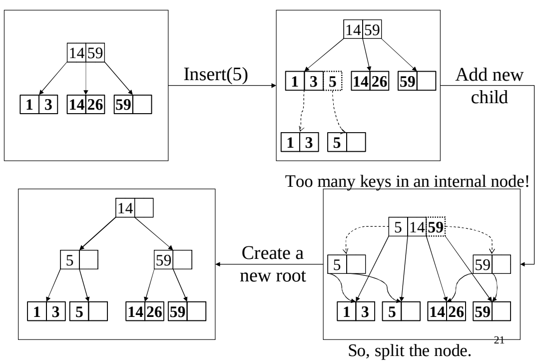
Splitting the Root



Insertions and Split Ends



Propagating Splits

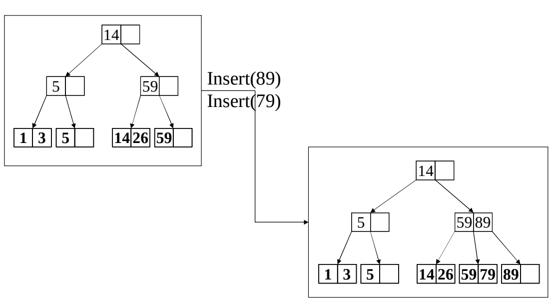


Insertion in Boring Text

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 $\lfloor (L+1)/2 \rfloor$ 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 $\lfloor (M+1)/2 \rfloor$ 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

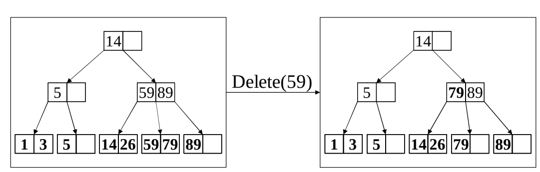
This makes the tree deeper!

After More Routine Inserts



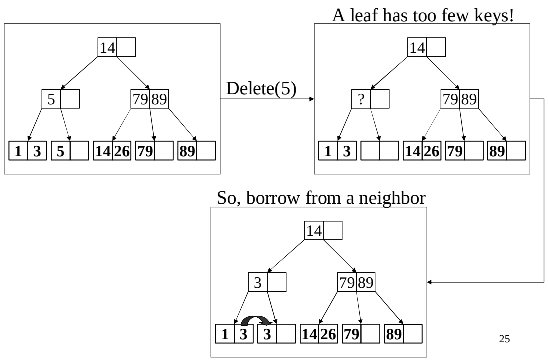
Deletion

1. Delete item from leaf
2. Update keys of ancestors if necessary



What could go wrong?

Deletion and Adoption

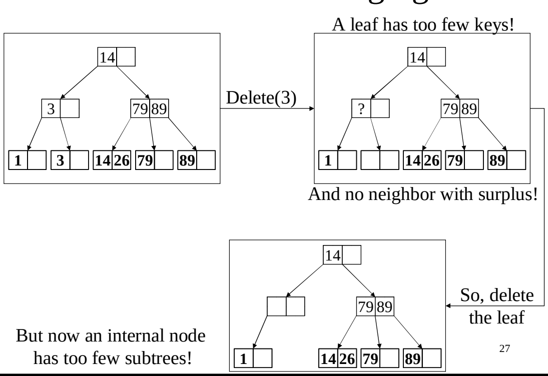


Deletion and Merging

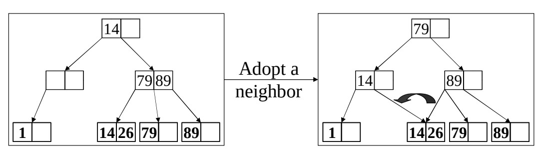
- What if the neighbor doesn't have enough for you to borrow from?

  e.g. you have $\lceil M/2 \rceil$-1 and he has $\lceil M/2 \rceil$?

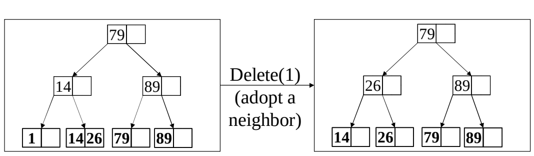
Deletion and Merging



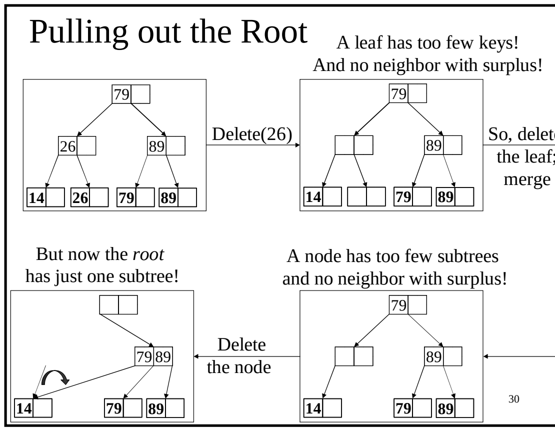
Deletion with Propagation (More Adoption)



A Bit More Adoption

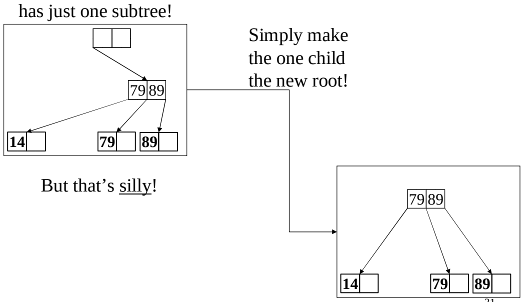


Pulling out the Root



Pulling out the Root (continued)

The *root*
has just one subtree!



31

Deletion in *Two* Boring Slides of Text

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 neighbor; 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**!

Why will merging always work if adopting doesn't?

32

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**!

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!

33

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?)*
- 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

34

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

Why would we ever use these?

35

To Do

- Work on Project #2

- Finish reading Chapter 4
- Start reading Chapter 5

36