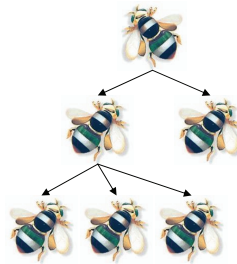


# CSE 326: Data Structures

## B-Trees and B+ Trees

Brian Curless  
Spring 2008



## Announcements (4/30/08)

- Midterm on Friday
- Special office hour: 4:30-5:30 Thursday in Jaech Gallery (6<sup>th</sup> floor of CSE building)
  - This is *instead of* my usual 11am office hour.
- Reading for this lecture: Weiss Sec. 4.7

2

## Traversing very large datasets

Suppose we had very many pieces of data (as in a database), e.g.,  $n = 2^{30} \approx 10^9$ .

How many (worst case) hops through the tree to find a node?

- BST
- AVL
- Splay

3

## Memory considerations

What is in a tree node? In an object?

Node:

Object obj;  
Node left;  
Node right;  
Node parent;

Object:

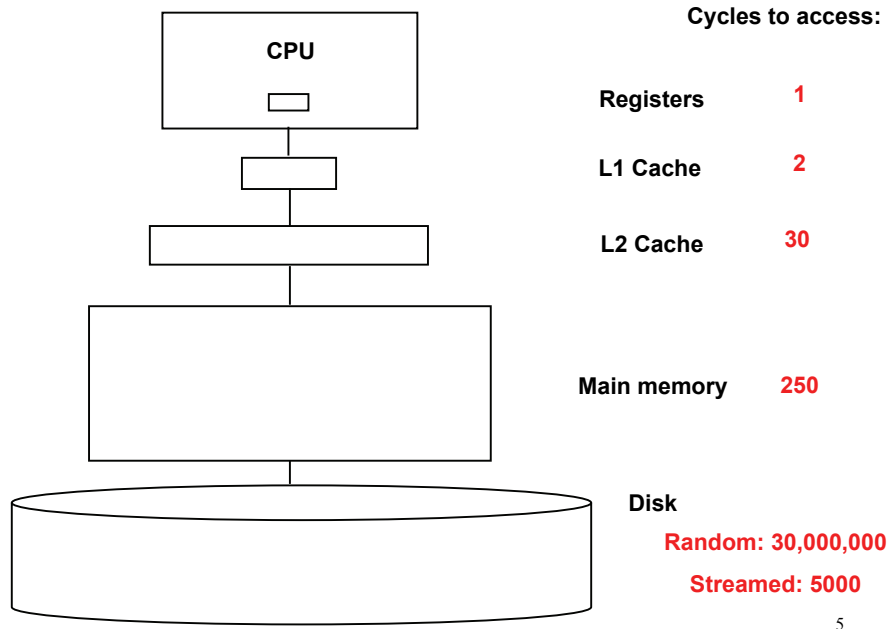
Key key;  
...data...

Suppose the data is 1KB.

How much space does the tree take?

How much of the data can live in 1GB of RAM?

4



5

## Minimizing random disk access

In our example, almost all of our data structure is on disk.

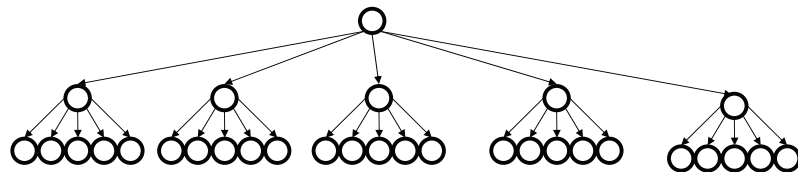
Thus, hopping through a tree amounts to random accesses to disk. Ouch!

How can we address this problem?

6

## M-ary Search Tree

Suppose, *somehow*, we devised a search tree with maximum branching factor  $M$ :



Complete tree has height:

# hops for *find*:

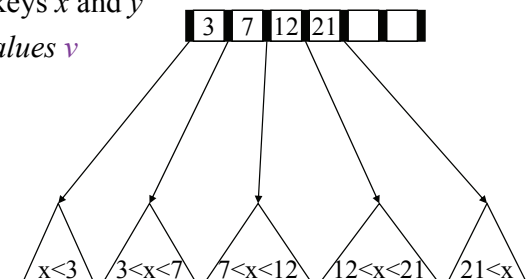
Runtime of *find*:

7

## B-Trees

How do we make an  $M$ -ary search tree work?

- Each **node** has (up to)  $M-1$  keys.
- Order property:
  - subtree between two keys  $x$  and  $y$  contain leaves with *values*  $v$  such that  $x < v < y$



8

## B-Tree Structure Properties

### Root (special case)

- has between 2 and  $M$  children (or root could be a leaf)

### Internal nodes

- store up to  $M-1$  keys
- have between  $\lceil M/2 \rceil$  and  $M$  children

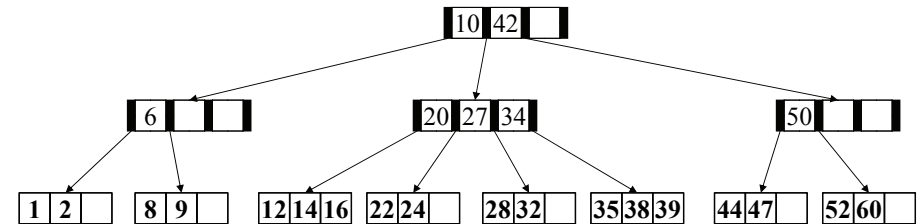
### Leaf nodes

- store between  $\lceil (M-1)/2 \rceil$  and  $M-1$  sorted keys
- all at the same depth

9

## B-Tree: Example

B-Tree with  $M = 4$

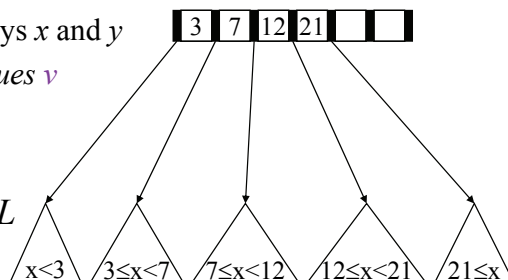


10

## B+ Trees

In a B+ tree, the internal nodes have no data – only the leaves do!

- Each internal node still has (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 “ $\leq$ ”
- Leaf nodes have up to  $L$  sorted keys.



11

## B+ Tree Structure Properties

### Root (special case)

- has between 2 and  $M$  children (or root could be a leaf)

### Internal nodes

- store up to  $M-1$  keys
- have between  $\lceil M/2 \rceil$  and  $M$  children

### Leaf nodes

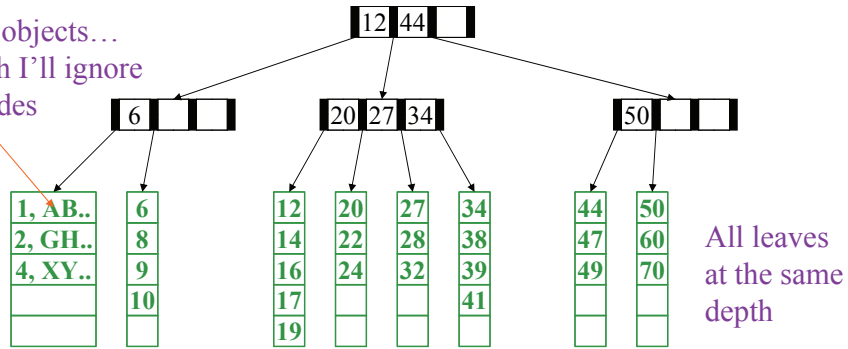
- where data is stored
- all at the same depth
- contain between  $\lceil L/2 \rceil$  and  $L$  data items

12

## B+ Tree: Example

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

Data objects...  
which I'll ignore  
in slides



Definition for later: “neighbor” is the next sibling to the left or right.<sup>13</sup>

## Disk Friendliness

What makes B+ trees disk-friendly?

### 1. Many keys stored in a node

- 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

14

## B+ trees vs. AVL trees

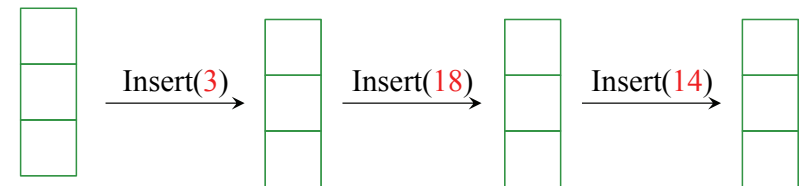
Suppose again we have  $n = 2^{30} \approx 10^9$  items:

- Depth of AVL Tree
- Depth of B+ Tree with  $M = 256$ ,  $L = 256$

Great, but how to we actually make a B+ tree and keep it balanced...?

15

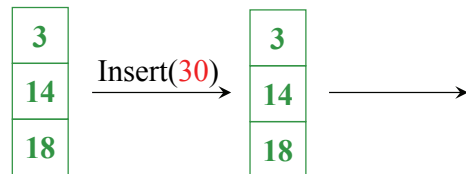
## Building a B+ Tree with Insertions



The empty  
B-Tree

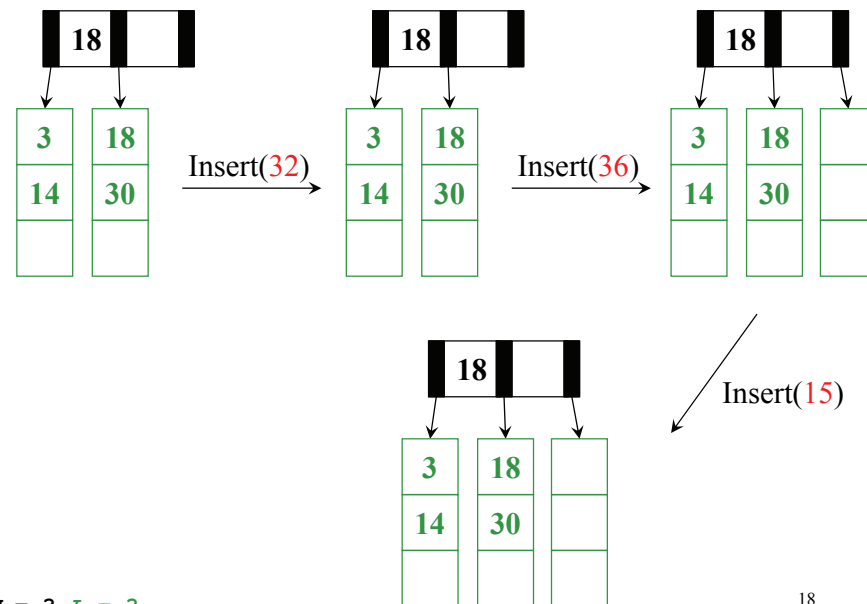
$M = 3$   $L = 3$

16



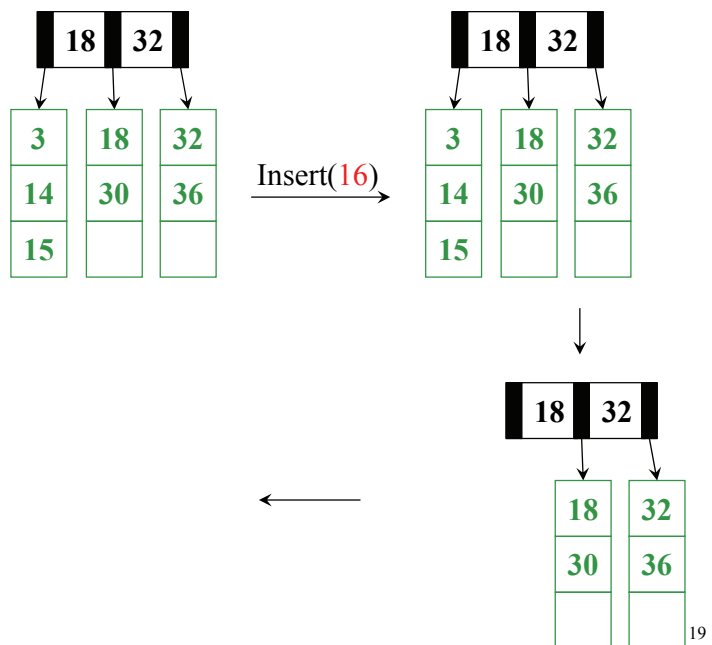
$M = 3$   $L = 3$

17



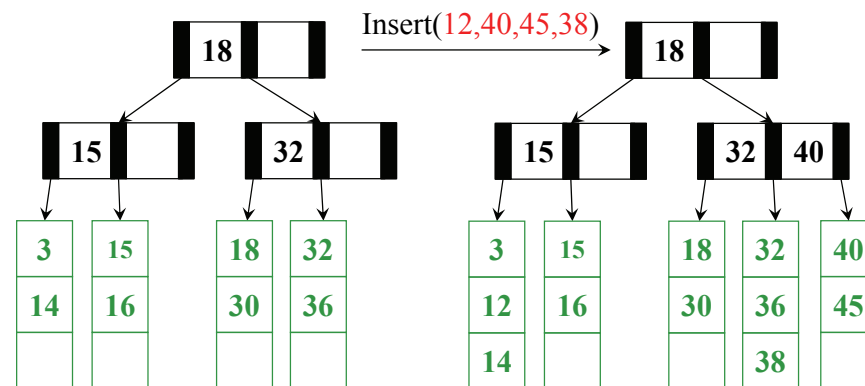
$M = 3$   $L = 3$

18



$M = 3$   $L = 3$

19



$M = 3$   $L = 3$

20

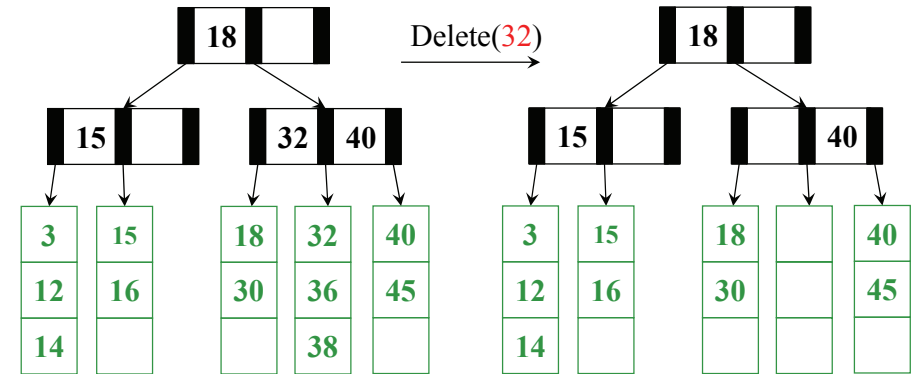
# Insertion Algorithm

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:
    - 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  $M+1$  children, **overflow!**
3. If an internal node ends up with  $M+1$  children, **overflow!**
  - Split the node into two nodes:
    - original with  $\lceil (M+1)/2 \rceil$  children
    - new one with  $\lfloor (M+1)/2 \rfloor$  children
  - 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
5. Propagate keys up tree.

This makes the tree deeper!

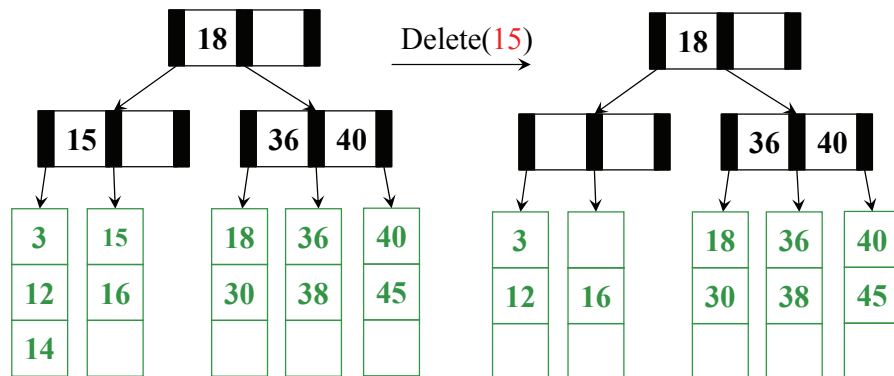
21

# And Now for Deletion...



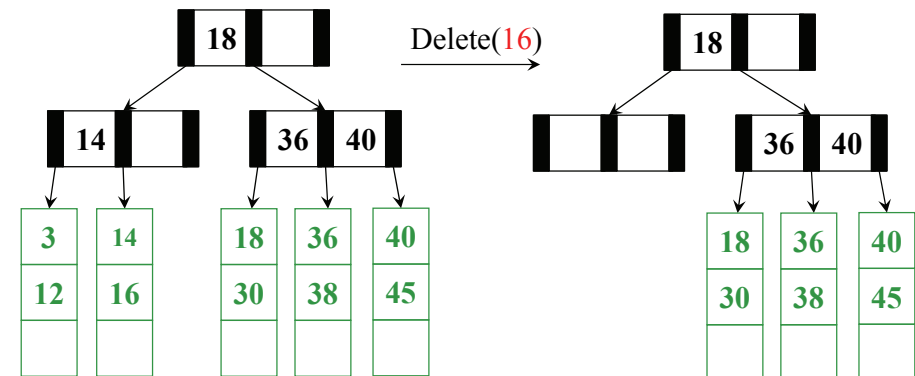
$M = 3 \quad L = 3$

22



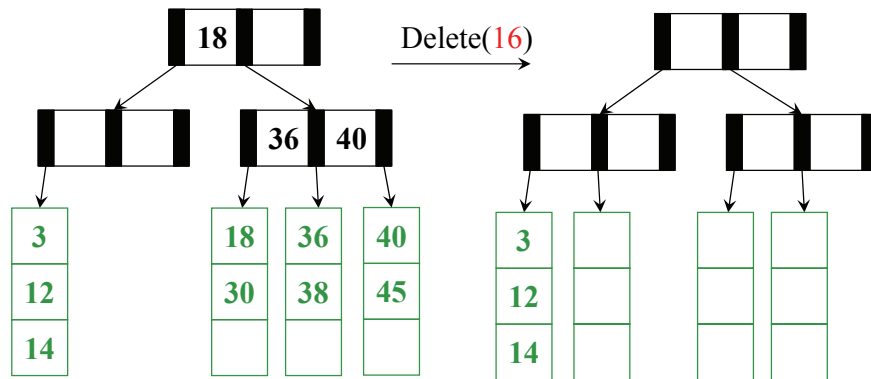
$M = 3 \quad L = 3$

23



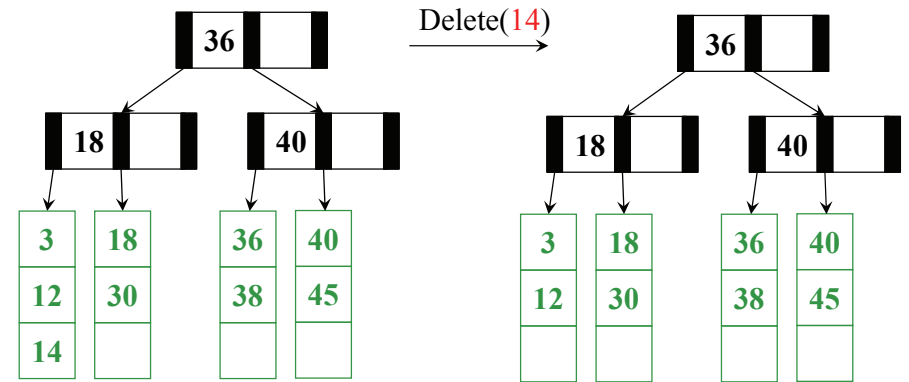
$M = 3 \quad L = 3$

24



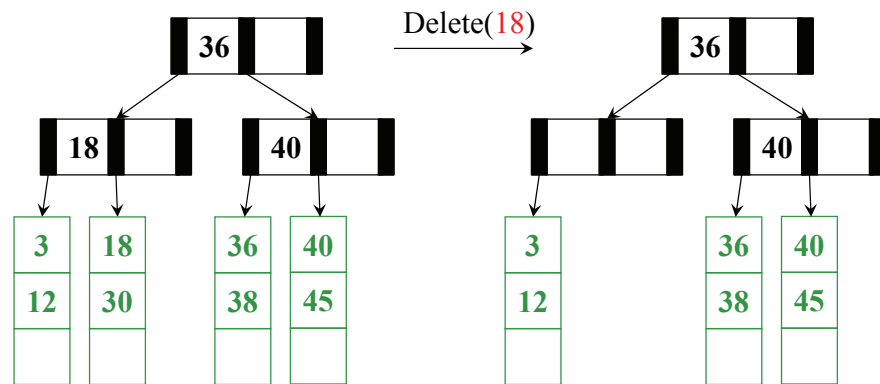
$M = 3 \quad L = 3$

25



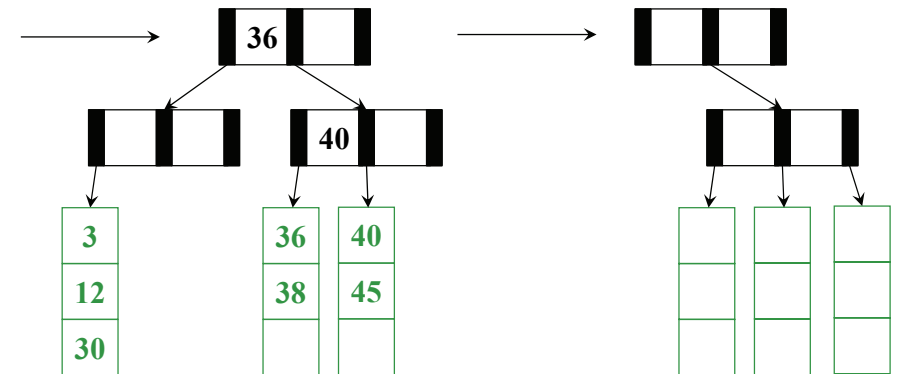
$M = 3 \quad L = 3$

26



$M = 3 \quad L = 3$

27



$M = 3 \quad L = 3$

28

## Deletion Algorithm

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  $\lceil M/2 \rceil$  children, **underflow!**

29

## Deletion Slide Two

3. If an internal node ends up with fewer than  $\lceil M/2 \rceil$  children, **underflow!**
    - Adopt from a neighbor; update the parent
    - If adoption won't work, merge with neighbor
    - If the parent ends up with fewer than  $\lceil M/2 \rceil$  children, **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!
5. Propagate keys up through tree.

30

## 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?)
- Pick branching factor  $M$  and data items/leaf  $L$  such that each node takes one full page/block of memory/disk.

31

## 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

32