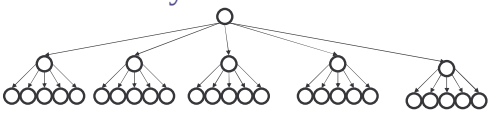
B-Trees

Section 4.7 in Weiss

M-ary Search Tree



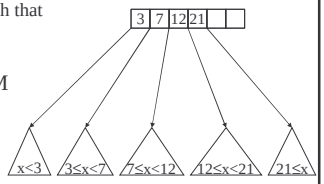
- Maximum branching factor of `M`
- Complete tree has height =

disk accesses for *find*:

Runtime of *find*:

Solution: B-Trees

- specialized *M*-ary search trees
- Each **node** has (up to) M-1 keys:
  - subtree between two keys *x* and *y* contains leaves with *values v* such that $x \le v < y$
- Pick branching factor M such that each node takes one full {*page*, *block*} of memory



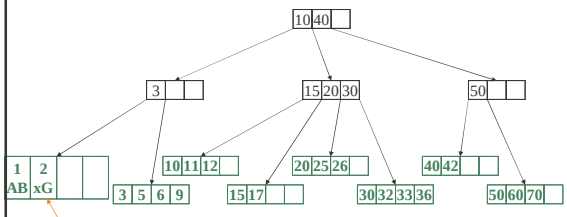
B-Trees

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

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!

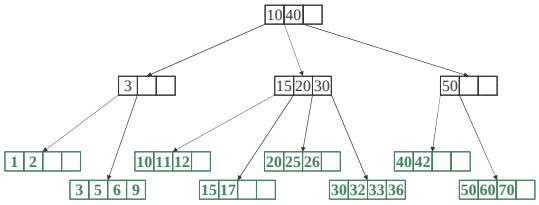
B-Tree Properties ‡

- Data is stored at the leaves
- All leaves are at the same depth and contain between $\lceil L/2 \rceil$ and L data items
- Internal nodes store up to M-1 keys
- Internal nodes have between $\lceil M/2 \rceil$ and M children
- Root (special case) has between 2 and `M` children (or root could be a leaf)

‡These are technically B⁺-Trees

Example, Again

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



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

7

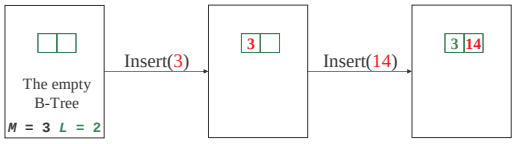
B-trees vs. AVL trees

Suppose we have 100 million items (100,000,000):

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

8

Building a B-Tree

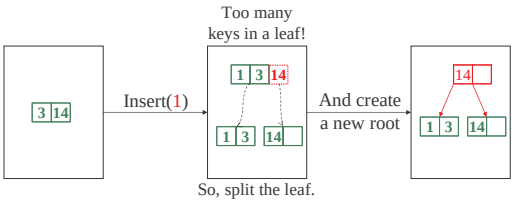


Now, Insert(1)?

9

Splitting the Root

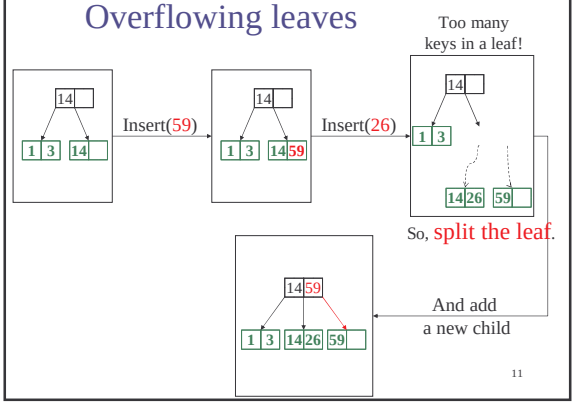
M = 3 L = 2



10

M = 3 L = 2

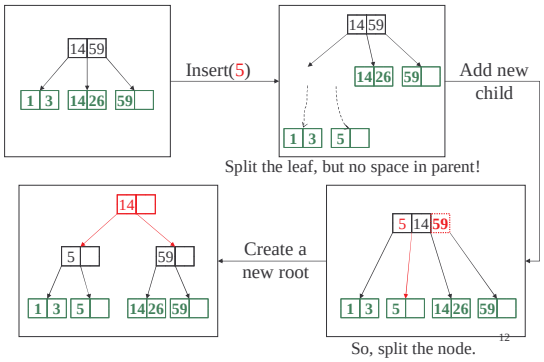
Overflowing leaves



11

M = 3 L = 2

Propagating Splits



12

Insertion Algorithm

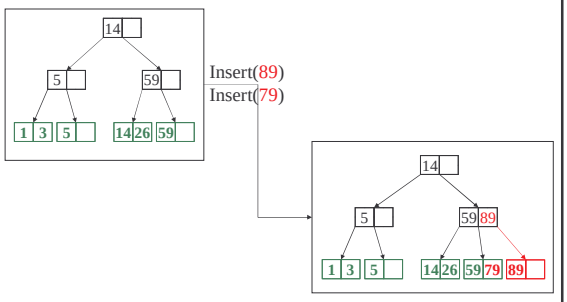
1. Insert the key in its leaf
2. If the leaf ends up with L+1 items, **overflow**!
   - Split the leaf into two nodes:
     - original with $\lceil (L+1)/2 \rceil$ items
     - new one with $\lfloor (L+1)/2 \rfloor$ items
   - Add the new child to the parent
   - If the parent ends up with $M+1$ items,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

13

M = 3 L = 2

After More Routine Inserts

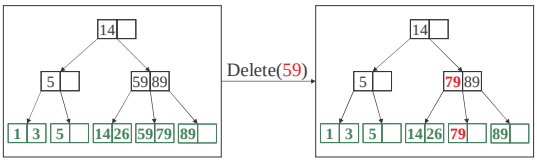


14

M = 3 L = 2

Deletion

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

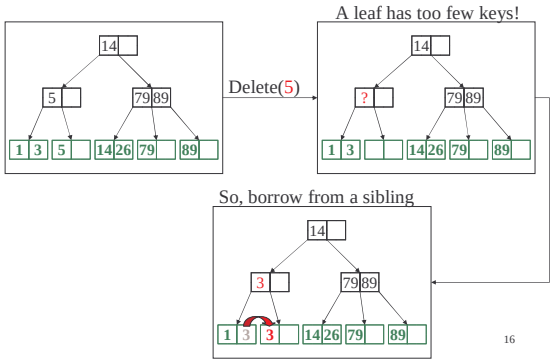


What could go wrong?

15

M = 3 L = 2

Deletion and Adoption



16

Does Adoption Always Work?

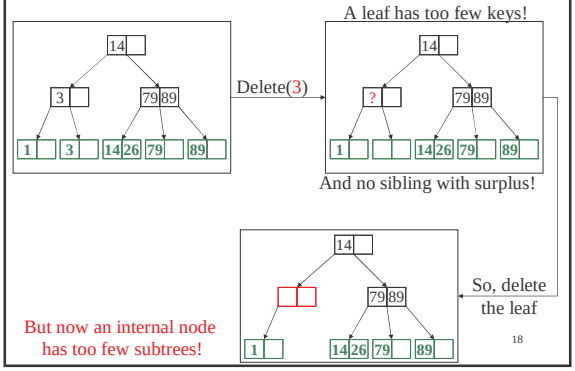
- What if the sibling doesn't have enough for you to borrow from?

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

17

M = 3 L = 2

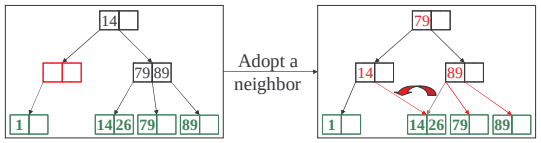
Deletion and Merging



18

M = 3 L = 2

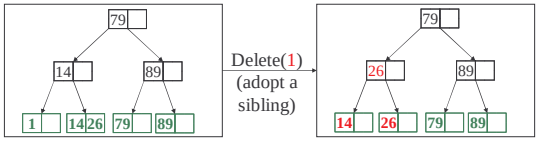
Deletion with Propagation (More Adoption)



19

M = 3 L = 2

A Bit More Adoption

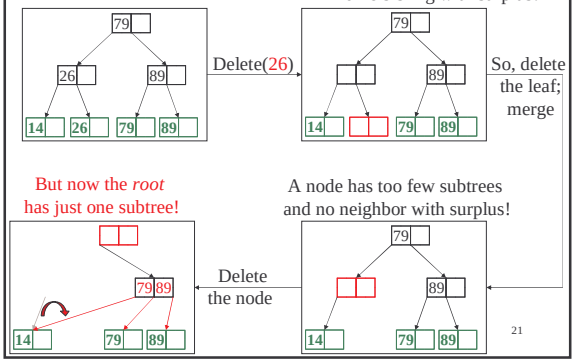


20

M = 3 L = 2

Pulling out the Root

A leaf has too few keys! And no sibling with surplus!

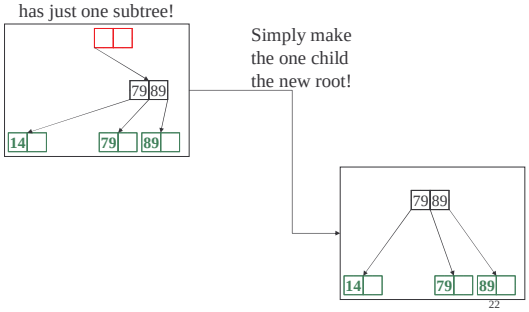


21

M = 3 L = 2

Pulling out the Root (continued)

The *root* has just one subtree!



22

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 sibling; 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**!

23

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!

24

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?)*
- If `M = L = 128`, then a B-Tree of height 4 will store at least 30,000,000 items

25

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

26