Lecture 11
10/20/2003
B-trees

B-Trees

So far, we have assumed that all of our data resides in memory, so that all data accesses wer Unfortunatly, this is a poor assumption. Large databases cannot store all their records in memory, a store information on disk until it is needed. This destorys our model of run-time analysis, since dis are more expensive than memory accesses by several orders of magnitude.

Example 1: comparing disk and memory accesses

According to the textbook, a 25-MIPS machine can execute 200,000 instructions during the tim to access the disk once. A 25-MIPS machine, assuming one instruction per cycle, only runs at 25 M

If we store our records in a binary search tree, even a balanced one, we have to do O(lg *n*) disk acc a leaf node. Not a pretty picture. Since instructions are so much faster than disk accesses, we are write a much more complicated data structure if it means reducing the height of the tree to (b constant, thus nearly eliminating the penalty of going to disk.

M-ary Search Trees

In an *M-ary search tree*, each node has *M* - 1 keys, and traversing a tree requires us to ma decisions. When searching for element *x*, we compare *x* to each key at the current node. If *x* is less th key, we search the left-most subtree. If *x* is greater than or equal to the first key but less than the seco search the second key, *etc*. If *x* is greater than or equal to the last key, we search the right-most subtre

The depth of a (balanced) *M*-ary tree is only $O(\log_M n)$, which is much better than O(lg *n*). Becau applications we can find a reasonable upper-bound on *n*, we can choose a value of *M* so that $\log_M n$ constant (4 or 5). Since we read a large block of data on each disk read, we can afford to store lots each node.

The B-tree data structure

A *B-tree of order M* is an *M*-ary search tree with the following properties:

- The data items are stored at the leaves only.
- The nonleaf nodes store at most *M*-1 keys; key *i* is the smallest key in subtree *i*+1.
- The root is either a leaf or has between 2 and *M* children.
- All nonleaf nodes have between ceil(*M*/2) and *M* children.
- All leaves are at the same depth and contain between ceil(*L*/2) and *L* elements, for some value o

Our choice for *M* and *L* depend on how large a block is on our disk, and thus on how much data each disk access. Lower-bounding the number of keys allowed in an internal node prevents the E degenerating into a binary tree.

B-tree Operations

We've already discussed how to find an element, when we discussed the structure of an *M*-ary tree and deletions generalize in much the same way. The problems occur when an insertion would put an a leaf that is already full, or when a deletion would leave too few items in a leaf.

Insertion

Suppose we want to insert into a leaf which is already full. We solve the problem by insertir element and spliting the leaf into two equal parts. We then add a new key to the leaf's parent, and child pointers accordinging. If this would place too many keys in the parent, we split the parent into and bubble the middle key up to *its* parent. We continue this as necessary, terminating in the worst c new root with two children.

Alternately, instead of splitting the node, we could push the smallest item into its left sibling, or item into the right sibling, creating a new sibling if the target doesn't exist. Creating a new siblin course add a new key to the parent, which could require the same type of shifting of keys. This pro also percolate up the tree, in the worst case requiring a final split of the root node.

Since splitting a node is expensive (it requires two disk writes), we would like to avoid it a possible. If we can keep new elements distributed across all leaves evenly, we will have to fill the lea a split is required. For this reason, implementing insertion usually involves many cases and computation. We allow this because computation is so much cheaper than disk access.

Deletion

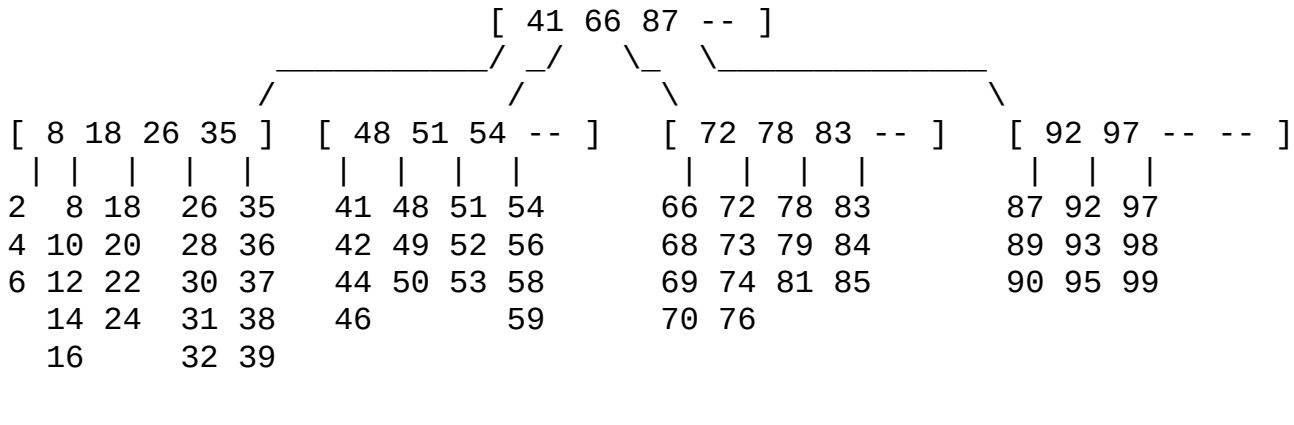
Suppose we delete an item from a leaf, and there are now fewer than *L*/2 items in the leaf. We borrow an item from a sibling, adjusting the keys in the parent as necessary; or we can merge two this results in too few children, the parent can borrow children from its siblings, or merge with pushing a key upto its parent. This process, too, can percoloate up the tree, in the worst case caus root to be replaced by one of its children.

As with insertion, we allow ourselves to check many cases and attempt to borrow sibilings if we merging nodes, since a merge is as expensive as a split.

Examples

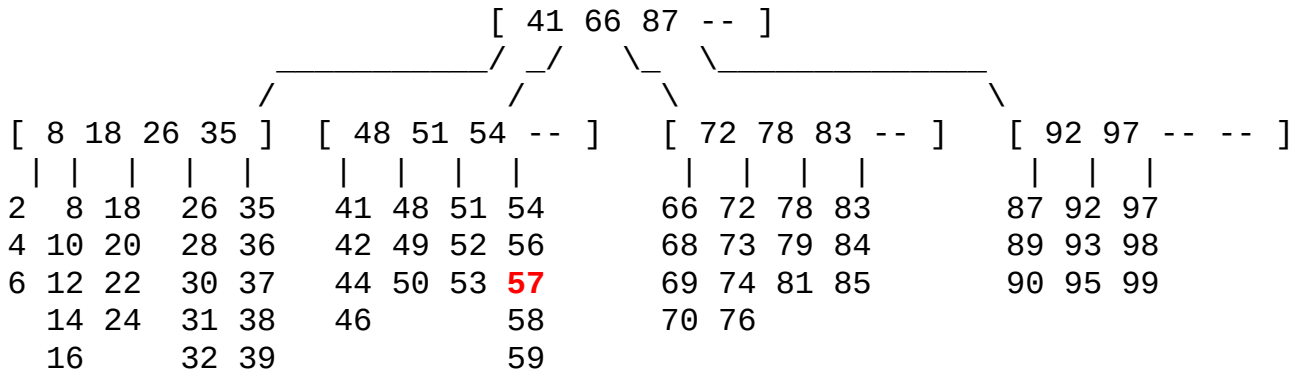
We start with the following B-tree of order 5 (Figure 4.59 on page 141 of the textbook). In this exa 5 and *L* = 5, so each internal node can have between 2 and 4 keys, and the leaves can have betwe elements.

Example 2: The initial B-tree



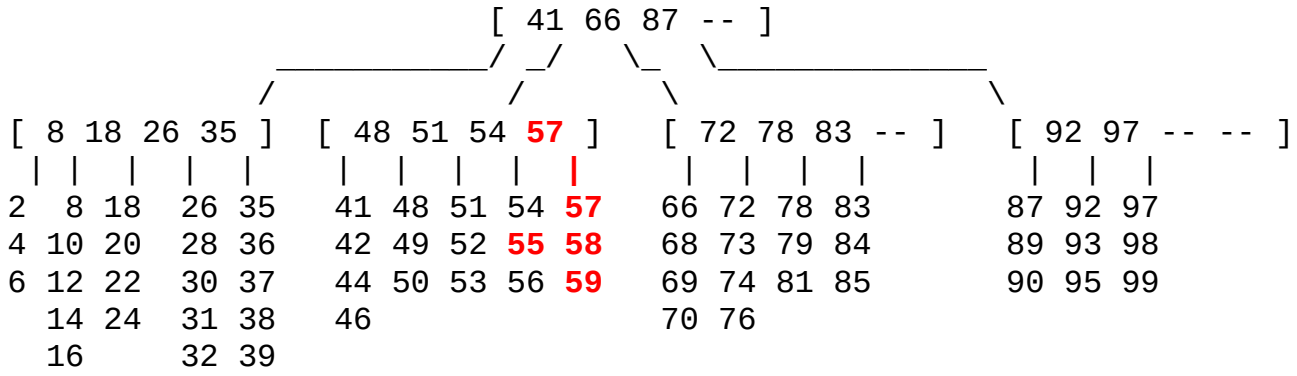
Example 3: Insert(57)

Since the leaf 57 belongs in has 4 elements, we can just put 57 in the proper leaf.



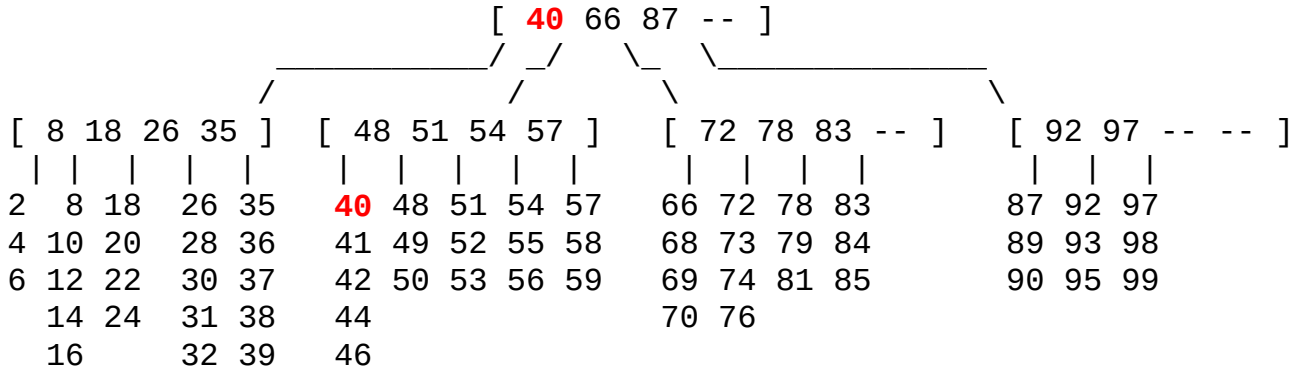
Example 4: Insert(55)

Since 55 goes in the same leaf we just insert 57 into, we either split the leaf, or push some d sibling. Here, we'll split the leaf, promoting one element of the leaf (57) to the parent as a key.



Example 5: Insert(40)

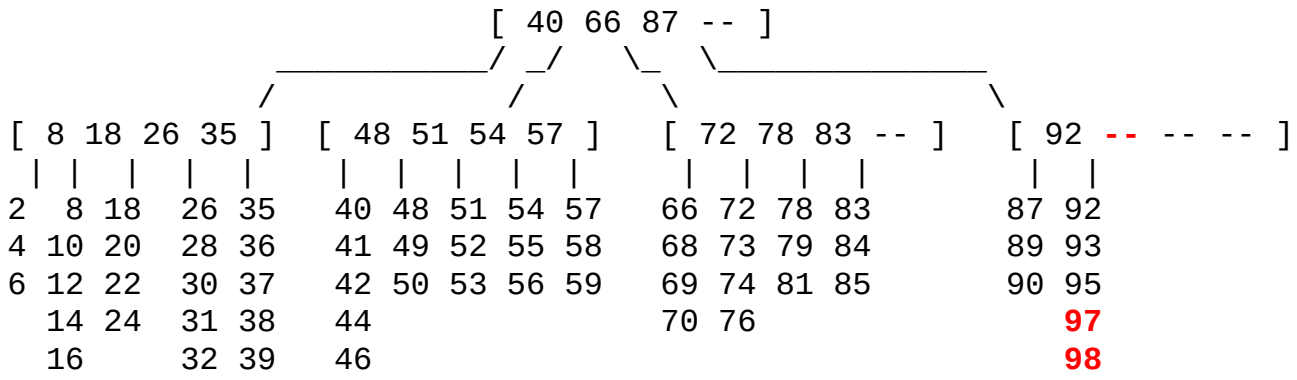
This time, not only is the leaf which takes 40 full, but it's parent is full as well. Instead of splitting leaf and the parent (which would promote 38 as a key, and promote 26 as a key to the root), we'll 40 over to the leaf's right sibling. This also makes 40 the key in the parent, instead of 41.



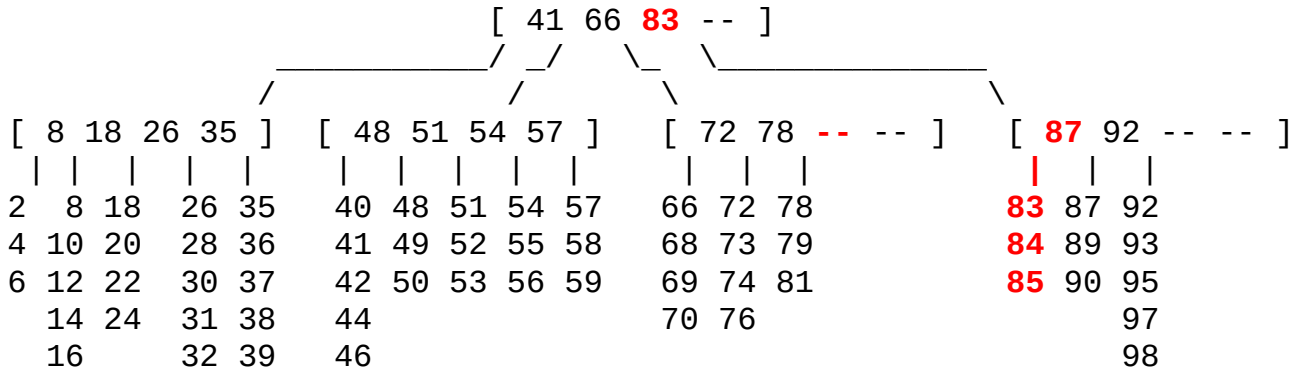
Example 6: Delete(99)

In order to delete 99, we see that we deleting 99 causes its leaf to hold too few elements. We

this by merging two leaves.



But now, the parent node has too few keys. Instead of merging the two left nodes, we will borr from the left sibling. This promotes 83 as a key into the root, and demotes 87 as a key from the root.



A special case

A B-tree of order 4 is also known as a 2,3,4-tree (since each internal node has 2, 3 or 4 children) implementation of a 2,3,4-tree, known as a red-black tree, is another type of balanced binary tree. V talk about red-black trees in this class; you will see them in CS25. Suffice to say, a red-black alternative to AVL trees for maintaining balance in a binary tree.

The `TreeMap` class, part of the standard Java™ packages, uses a red-black tree to store its items.