```
                    CS 61B: Lecture 27
                  Friday, October 29, 2010

2-3-4 TREES
===========
A 2-3-4 tree is a perfectly balanced tree.  It has a big advantage over regular
binary search trees:  because the tree is perfectly balanced, find, insert, and
remove operations take O(log n) time, even in the worst case.


2-3-4 trees are thus named because every node has 2, 3, or 4 children, except
leaves, which are all at the bottom level of the tree.  Each node stores 1, 2,
or 3 entries, which determine how other entries are distributed among its
children's subtrees.


Each internal (non-leaf) node has one more child than keys.  For example, a
node with keys [20, 40, 50] has four children.  Eack key k in the subtree
rooted at the first child satisfies k <= 20; at the second child,
20 <= k <= 40; at the third child, 40 <= k <= 50; and at the fourth child,
k >= 50.


WARNING:  The algorithms for insertion and deletion I'll discuss today are
different from those discussed by Goodrich and Tamassia.  The text presents
"bottom-up" 2-3-4 trees, so named because the effects of node splits at the
bottom of the tree can work their way back up toward the root.  I'll discuss
"top-down" 2-3-4 trees, in which insertion and deletion finish at the leaves.
Top-down 2-3-4 trees are usually faster than bottom-up ones.  Goodrich and
Tamassia call 2-3-4 trees "(2, 4) trees".


2-3-4 trees are a type of B-tree, which you may learn about someday in
connection with fast disk access for database systems.


[1]  Object find(Object k);

Finding an entry is straightforward.        ==========
Start at the root.  At each node,           +20 40 50+
check for the key k; if it's not         /--===============--\
present, move down to the           /---/       /  \         \-----\
appropriate child chosen by      ----        ----      ----         ======
comparing k against the keys.    |14|        |32|      |43|         +70 79+
Continue until k is found,       ----        ----      ----         ======
or k is not found at a           /  \        /  \      /  \         /  |  \
leaf node.  For example,      ---- ---- ---- ---- ---- ---- ---------- ==== ----
find(74) visits the           |10| |18| |25| |33| |42| |47| |57 62 66| +74+ |81|
double-lined boxes at right.  ---- ---- ---- ---- ---- ---- ---------- ==== ----


Incidentally, you can define an inorder traversal on 2-3-4 trees analogous to
that on binary trees, and it visits the keys in sorted order.
```

```
[2]  void insert(Object k, Object e);

insert(), like find(), walks down the tree in search of the key k.  If it finds
an entry with key k, it proceeds to that entry's "left child" and continues.

Unlike find(), insert() sometimes modifies         ----        -------
nodes of the tree as it walks down.                |20|        |11 20|
Specifically, whenever insert() encounters         ----        -------
a 3-key node, the middle key is ejected,           / \   =>    / | \
and is placed in the parent node instead.     ========= ----   ---- ---- ----
Since the parent was previously treated the   +10 11 12+ |30|  |10| |12| |30|
same way, the parent has at most two keys,    ========= ----   ---- ---- ----
and always has room for a third.  The other
two keys in the 3-key node are split into two separate 1-key nodes, which are
divided underneath the old middle key (as the figure illustrates).


For example, suppose we                    ----
insert 60 into the tree                    |40|
depicted in [1].  The                    /------\
first node visited is              /---/        \----\
the root, which has three        ----                 ----
children; so we kick the         |20|                 |50|
middle key (40) upstairs.        ----               /------\
Since the root node has         /    \             /        \
no parent, a new node        ----    ----       ----         ----------
is created to hold 40        |14|    |32|       |43|         |62 70 79|
and becomes the root.        ----    ----       ----         ----------
Similarly, 62 is kicked      /  \    /  \       /  \         /  |  |  \
upstairs when insert()    ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
finds the node containing |10| |18| |25| |33| |42| |47| |57 60| |66| |74| |81|
it.  This ensures us that ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
when we arrive at the leaf
(labeled 57 in this case), there's room to add the new key 60.


Observe that along the way, we created a new 3-key node "62 70 79".  We do not
kick its middle key upstairs until the next time it is visited.


Again, the reasons why we split every 3-key node we encounter (and move its
middle key up one level) are (1) to make sure there's room for the new key in
the leaf node, and (2) to make sure that above the leaves, there's room for any
key that gets kicked upstairs.  Sometimes, an insertion operation increases the
height of the tree by one by creating a new root.
```

[3] void remove(Object k);

2-3-4 tree remove() is similar to remove() on binary trees: you find the entry
you want to remove (having key k). If it's in a leaf, you remove it. If it's
in an internal node, you replace it with the entry with the next higher key.
That entry is always in a leaf. In either case, you remove an entry from a
leaf in the end.

Like insert(), remove() changes nodes of the tree as it walks down. Whereas
insert() eliminates 3-key nodes (moving nodes up the tree) to make room for new
keys, remove() eliminates 1-key nodes (pulling keys down the tree) so that a
key can be removed from a leaf without leaving it empty. There are three ways
1-key nodes (except the root) are eliminated.

(1) When remove() encounters a 1-key
node (except the root), it tries
to steal a key from an adjacent
sibling. But we can't just steal
the sibling's key without
violating the search tree
invariant. This figure shows
remove's action, called a
"rotation", when it reaches "30".
We move a key from the sibling to
the parent, and we move a key from the parent to the 1-key node. We also move
a subtree S from the sibling to the 1-key node (now a 2-key node).

```
      -------                -------
      |20 40|                |20 50|
      -------                -------
      /  |  \        =>      /   |   \
  ---- ==== ----------   ---- ------- -------
  |10| +30+ |50 51 52|   |10| |30 40| |51 52|
  ---- ==== ----------   ---- ------- -------
   /\   /\   / | | \      /\   / | \  / | \
         S                        S
```

Goodrich & Tamassia call rotations "transfer" operations. Note that we can't
steal a key from a non-adjacent sibling.

(2) If no adjacent sibling has more than one
key, a rotation can't be used. In this case,
the 1-key node steals a key from its parent.
Since the parent was previously treated the
same way (unless it's the root), it has at
least two keys, and can spare one. The
sibling is also absorbed, and the 1-key node
becomes a 3-key node. The figure illustrates
remove's action when it reaches "10". This is called a "fusion" operation.

```
      -------               ----
      |20 40|               |40|
      -------               ----
      /  |  \       =>       /  \
  ==== ---- ----   ---------- ----
  +10+ |30| |50|   |10 20 30| |50|
  ==== ---- ----   ---------- ----
```

(3) If the parent is the root and contains only one key, and the sibling
contains only one key, then the current 1-key node, its 1-key sibling, and the
1-key root are fused into one 3-key node that serves as the new root. The
height of the tree decreases by one.

Eventually we reach a leaf. After processing the leaf, it has at least two
keys (if there are at least two keys in the tree), so we can delete the key
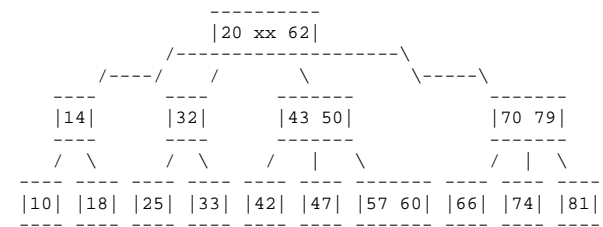and still have one key in the leaf.

For example, suppose we
remove 40 from the large
tree depicted in [2]. The
root node contains 40,
which we mark "xx" to
remind us that we plan to
replace it with the
smallest key in the root
node's right subtree. To
find that key, we move on
to the 1-key node labeled
50. Following our rules
for 1-key nodes, we fuse 50 with its sibling and parent to create a new 3-key
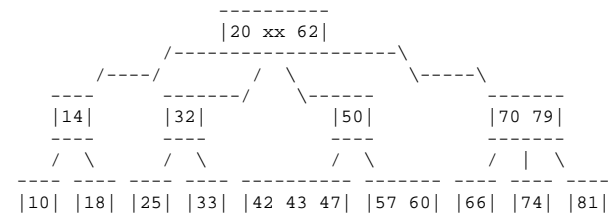root labeled "20 xx 50".

```
                  ----------
                  |20 xx 50|
                  /-----------------\
         /--/     /    \      \-----\
      ----    ----    ----     ----------
      |14|    |32|    |43|     |62 70 79|
      ----    ----    ----     ----------
      /  \    /  \    /  \      / | | \
  ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
  |10| |18| |25| |33| |42| |47| |57 60| |66| |74| |81|
  ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
```

Next, we visit the node
labeled 43. Again
following our rules for
1-key nodes, we rotate
62 from a sibling to the
root, and move 50 from
the root to the node
containing 43.

```
                  ----------
                  |20 xx 62|
                  /--------------------\
         /----/     /        \        \-----\
      ----    ----         -------       -------
      |14|    |32|         |43 50|       |70 79|
      ----    ----         -------       -------
      /  \    /  \         / | | \       / | \
  ---- ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
  |10| |18| |25| |33| |42| |47| |57 60| |66| |74| |81|
  ---- ---- ---- ---- ---- ---- ------- ---- ---- ----
```

Finally, we move down to
the node labeled 42. A
different rule for 1-key
nodes requires us to
fuse the nodes labeled
42 and 47 into a 3-key
node, stealing 43 from
the parent node.

```
                  ----------
                  |20 xx 62|
                  /--------------------\
         /----/        /  \        \-----\
      ----    -------/    \------     -------
      |14|    |32|         |50|       |70 79|
      ----    ----         ----       -------
      /  \    /  \         / \        / | \
  ---- ---- ---- ---- ---- ---------- ------- ---- ---- ----
  |10| |18| |25| |33| |42 43 47| |57 60| |66| |74| |81|
  ---- ---- ---- ---- ---------- ------- ---- ---- ----
```

The last step is to remove 42 from the leaf and replace "xx" with 42.

Running Times

A 2-3-4 tree with height h has between 2^h and 4^h leaves. If n is the total
number of nodes (including internal nodes), then n >= 2^(h+1) - 1. By taking
the logarithm of both sides, we find that h is in O(log n).

The time spent visiting a 2-3-4 node is typically longer than in a binary
search tree (because the nodes and the rotation and fusion operations are
complicated), but the time per node is still in O(1).

The number of nodes visited is proportional to the height of the tree. Hence,
the running times of the find(), insert(), and remove() operations are in O(h)
and hence in O(log n), even in the worst case.

Compare this with the Theta(n) worst-case time of ordinary binary search trees.

Another Approach to Duplicate Keys

Rather than have a separate node for each entry, we might wish to collect all
the entries that share a common key in one node. In this case, each node's
entry becomes a list of entries. This simplifies the implementation of
findAll(), which finds all the entries with a specified key. It also speeds up
other operations by leaving fewer entries in the main data structure.
Obviously, this is a change in the implementation, but not a change in the
dictionary ADT.

This idea can be used with hash tables, binary search trees, and 2-3-4 trees.