**Lesson 22 – Tables II: Balanced Search Trees**

**Learning Objectives:**

* Identify implementations of trees that maintain balance.
* Describe the benefits of a balanced tree.
* Describe how various trees maintain balance.

**Reading Assignment: Chapter 13, Section 13.1**

**2-3 Trees:**

* A 2-3 tree is a tree in which each internal node (nonleaf) has either two or three children, and all leaves are at the same level.
* More formally, *T* is a 2-3 tree of height *h* if:
  + *T* is empty (a 2-3 tree of height 0).

- OR -

* + *T* is of the form

*r*

/ \

*TL* *TR*

where *r* is a node that contains one data item and *TL* and *TR* are both 2-3 trees, each of height *h* – 1 (2-node).

* + - In this case, the search key in *r* must be greater than each search key in the left subtree *TL* and smaller than each search key in the right subtree *TR*.

- OR -

* + *T* is of the form

*r*

/ | \

*TL* *TM* *TR*

where *r* is a node that contains two data item and *TL* , *TM* and *TR* are 2-3 trees, each of height *h* – 1 (3-node).

* + - In this case, the smaller search key in *r* must be greater than each search key in the leftsubtree *TL* and smaller than each search key in the middle subtree *TM*.
    - The larger search key in *r* must be greater than each search key in the middle subtree *TM* and smaller than each search key in the right subtree *TR*.

**Searching a 2-3 Tree:**

* The ordering of items in a 2-3 tree is analogous to the ordering for a binary search tree and allows you to search efficiently for a particular item – O(log n).
  + A BST with *n* nodes cannot be shorter than .
  + A 2-3 tree with *n* nodes cannot be taller than .
* Searching a 2-3 tree is not ***more*** efficient than searching a balanced binary search tree, even though the 2-3 tree may be shorter.
* The benefit of using a 2-3 tree is that it is simpler to maintain balance.

**Inserting into a 2-3 Tree:**

* Consider the following tree:

50

/ \

30 70-90

/ \ / | \

10-20 40 60 80 100

* Insert 39:
  + Search the tree for location of 39 (if it existed), terminating at a leaf <40>.
  + Since this node only contains a single item, simply insert in existing node.

50

/ \

30 70-90

/ \ / | \

10-20 39-40 60 80 100

* Insert 38:
  + Search to leaf <39-40>, however we can’t simply add another value to this node (it already contains two).
  + But, the parent only contains a single value. Thus, the middle value <39> moves to the parent, and the other two values are split into two nodes as middle child and right child.

50

/ \

30-39 70-90

/ | \ / | \

10-20 38 40 60 80 100

* Insert 37:
  + Search takes us to node <38> and the new item can be simply stored in this node.

50

/ \

30-39 70-90

/ | \ / | \

10-20 37-38 40 60 80 100

* Insert 36:
  + Search take us to node <37-38> which is full (two items).
  + Between items 36, 37, and 38, the middle value (37) floats to the parent and the other two become individual nodes.

50

/ \

30-37-39 70-90

/ | | \ / | \

10-20 36 38 40 60 80 100

* + However, the parent is also full. So we again move the middle item to the next higher level and split <30-39> into two 2-nodes <30> and <39>. This provides two parent nodes for the four children.

37-50

/ | \

30 39 70-90

/ \ / \ / | \

10-20 36 38 40 60 80 100

* Insert 35, 34, and 33:
  + First, inserting 35.

37-50

/ | \

30 39 70-90

/ \ / \ / | \

10-20 35-36 38 40 60 80 100

* + Then, inserting 34.

37-50

/ | \

30-35 39 70-90

/ | \ / \ / | \

10-20 34 36 38 40 60 80 100

* + Finally, inserting 33:

37-50

/ | \

30-35 39 70-90

/ | \ / \ / | \

10-20 33-34 36 38 40 60 80 100

* Insert 32:

37-50

/ | \

30-35 39 70-90

/ | \ / \ / | \

10-20 32-33-34 36 38 40 60 80 100

37-50

/ | \

30-33-35 39 70-90

/ | | \ / \ / | \

10-20 32 34 36 38 40 60 80 100

33-37-50

/ / \ \

30 35 39 70-90

/ \ / \ / \ / | \

10-20 32 34 36 38 40 60 80 100

* + We now increase the height of the tree by 1:

37

/ \

33 50

/ \ / \

30 35 39 70-90

/ \ / \ / \ / | \

10-20 32 34 36 38 40 60 80 100

**Deleting from a 2-3 Tree:**

* The deletion strategy for a 2-3 tree is the inverse of its insertion strategy.
* Consider the following tree:

50

/ \

30 70-90

/ \ / | \

10-20 40 60 80 100

* Delete 70:
  + We see that the item is in the <70-90> node.
  + We start the process in a leaf. Swap 70 with its in-order successor (80). The in-order successor will always be a leaf. And, delete the leaf.

50

/ \

30 80-90

/ \ / \

10-20 40 60 100

* + But, this situation is not allowed; a 3-Node only has two children. So, 80 is merged with 60 as so:

50

/ \

30 90

/ \ / \

10-20 40 60-80 100

* Delete 100:
  + Notice that 100 is a 2-Node leaf. Deleting it means that we need a new right child. Since the left child contains 2 items, they are split and the 3 items are rearranged.

50

/ \

30 80

/ \ / \

10-20 40 60 90

* Delete 80:
  + Node <80> is on an internal node. So, 80 is swapped with its in-order successor and then the leaf is deleted.

50

/ \

30 90

/ \ / .

10-20 40 60 .

* + With the child deleted, we merge 60 and 90 into <60-90>. However, we do not have a second value in the parent to split.

50

/ \

30 -

/ \ / .

10-20 40 60-90 .

* + With the internal empty node, we merge the root with its left child to become <30-50> with 3 children.

30-50

/ | \

10-20 40 60-90

* In summary, to delete an item:
  + locate the node that contains it
  + if not a leaf, swap with in-order successor
  + if leaf contains two items, delete the appropriate item (done)
  + if deleting item also deletes leaf, then check siblings.

P L

/ \ => / \

S-L - S P

Redistribute

L ?

/ \ => / .

S - S-L .

Merge

P L

/ \ => / \

S-L - S P

/ | \ | / \ / \

a b c d a b c d

Redistribute

P ?

/ \ => / .

S - S-P .

/ \ | / | \ .

a b c a b c .

Merge

- .

| => .

S-L S-L

/ | \ / | \

a b c a b c

Delete Root