MA/CSSE 473 Day 21

AVL Tree
Maximum height
2-3 Trees
Heap Review: intro
Student questions?

Review: Representation change: AVL Trees (what you should remember…)

- Named for authors of original paper, **A**delson-**V**elskii and **L**andis (1962).
- An AVL tree is a height-balanced Binary Search Tree.
- A BST T is **height balanced** if T is empty, or if
  - | height(T_L) - height(T_R) | $\leq$ 1, and
  - T_L and T_R are both height-balanced.
- Show: Maximum height of an AVL tree with N nodes is Θ(log N) **Let's review that together**
- How do we maintain balance after insertion?
- **Exercise for later:** Given a pointer to the root of an AVL tree with N nodes, find the height of the tree in log N time
- Details on balance codes and various rotations are in the CSSE 230 slides that are linked from the schedule page.

Representation change: 2-3 trees

- Another approach to balanced trees
- Keeps all leaves on the same level
- Some non-leaf nodes have 2 keys and 3 subtrees
- Others are regular binary nodes.

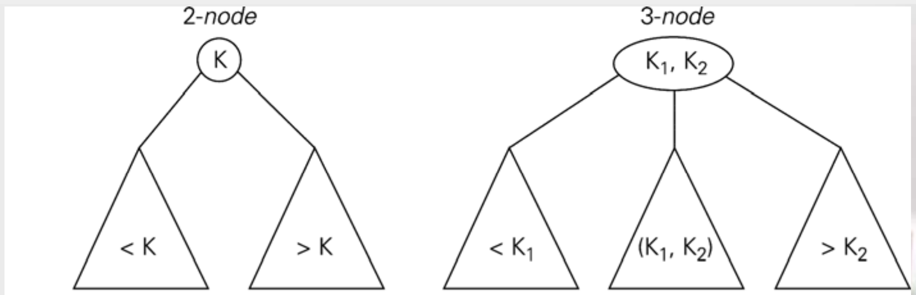


FIGURE 6.7 Two kinds of nodes of a 2-3 tree

2-3 tree insertion example

- More examples of insertion:
  http://www.cs.ucr.edu/cs14/cs14_06win/slides/2-3_trees_covered.pdf
  http://slady.net/java/bt/view.php?w=450&h=300

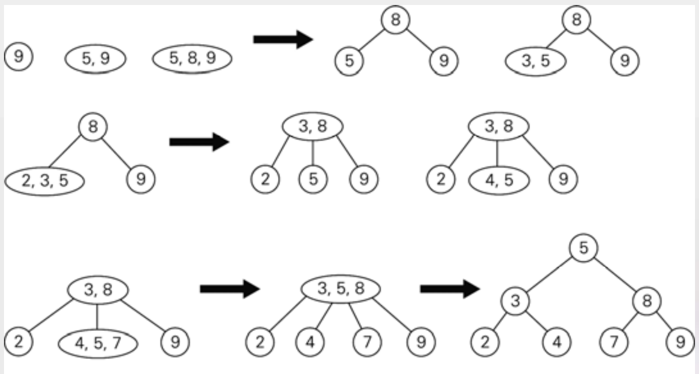


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7

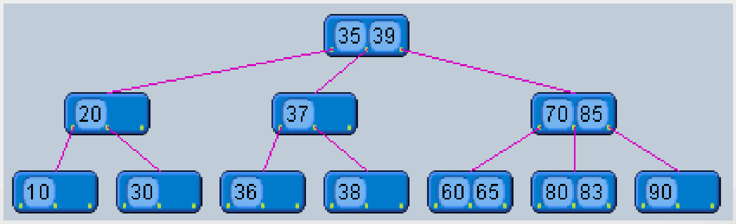
Add 10, 11, 12, … to the last tree

Efficiency of 2-3 tree insertion

- Upper and lower bounds on height of a tree with n elements?
- Worst case insertion and lookup times is proportional to the height of the tree.

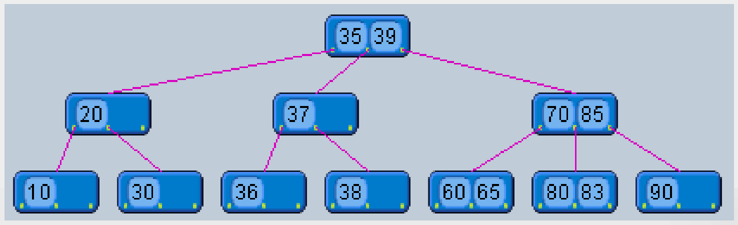
2-3 Tree insertion practice

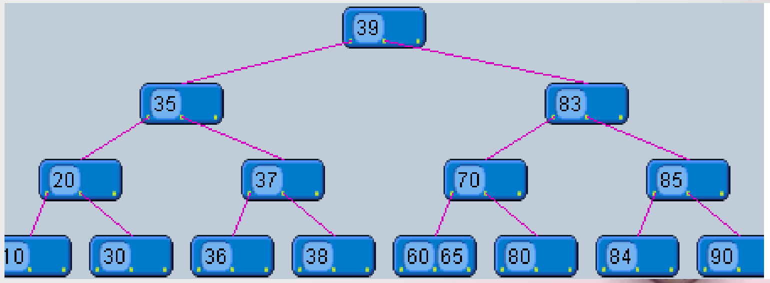
- Insert 84 into this tree and show the resulting tree



2-3 Tree insertion practice

- Insert 84 into this tree and show the resulting tree



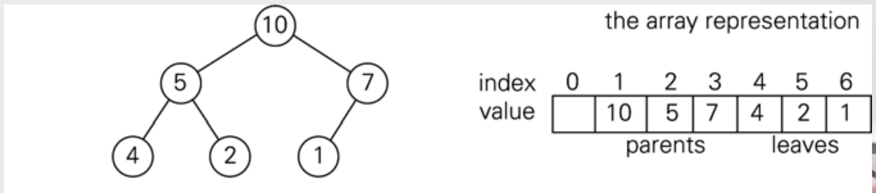


Binary (max) Heap Quick Review

Representation change example

See also Weiss, Chapter 21 (Weiss does min heaps)

- An almost-complete Binary Tree
  - All levels, except possibly the last, are full
  - On the last level all nodes are as far left as possible
  - No parent is smaller than either of its children
  - A great way to represent a Priority Queue
- Representing a binary heap as an array:



index	0	1	2	3	4	5	6
value		10	5	7	4	2	1

FIGURE 6.10 Heap and its array representation

Insertion and RemoveMax

- Insertion:
  - Insert at the next position (end of the array) to maintain an almost-complete tree, then "percolate up" within the tree to restore heap property.
- RemoveMax:
  - Move last element of the heap to replace the root, then "percolate down" to restore heap property.
- Both operations are Θ(log n).
- Many more details (done for min-heaps):
  - http://www.rose-hulman.edu/class/csse/csse230/201230/Slides/18-Heaps.pdf

Heap utilitiy functions

```
def percolateDown(a,i, n):
    """Within the n elements of A to be "re-heapified", the two subtrees of A[i]
       are already maxheaps. Repeatedly exchange the element currently in A[i] with
       the largest of its children until the tree whose root is a[i] is a max heap. """
    current = i # root position for subtree we are heapifying
    lastNodeWithChild = n//2  # if a node number is higher than this, it is a leaf.
    while current <= lastNodeWithChild:
        max = current
        if a[max] < a[2*current]:  # if it is larger than its left child.
            max = 2*current
        if 2*current < n and a[max] < a[2*current+1]:  # But if there is a right child,
            max = 2*current + 1                         # right child may be larger than either
        if max == current:
            break  # larger than its children, so we are done.
        swap(a, current, max) # otherwise, exchange, move down tree, and check again.
        current = max

def percolateUp(a,n):
    'Assume that elements 1 through n-1 are a heap; add element n and "re-heapify"'
    # compare to parent and swap until not larger than parent.
    current = n
    while current > 1:  # or until this value is in the root.
        if a[current//2] >= a[current]:
            break
        swap(a, current, current//2)
        current //= 2
```

Code is on-line, linked from the schedule page

HeapSort

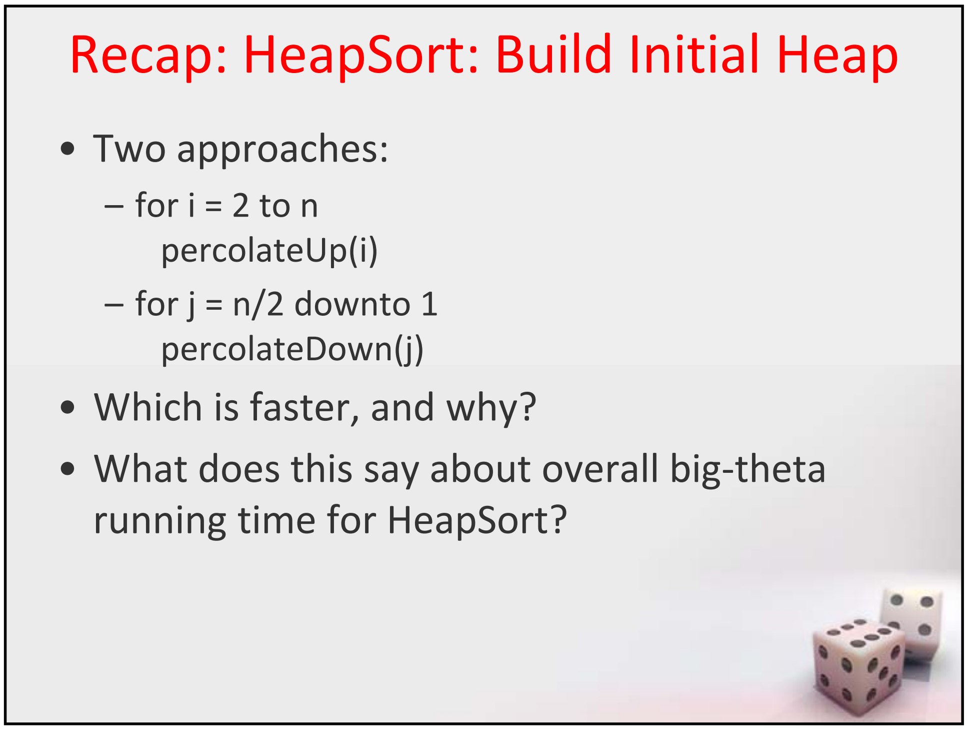
- Arrange array into a heap. (details next slide)
- for i = n downto 2:
  a[1]↔a[i], then "reheapify" a[1]..a[i-1]

HeapSort Code

```
# The next two functions tdo the same thing; take an unordered
# array and turn it into a max-heap.  In HW 10, you will show
# that the secondis much more efficient than the first.
# So this first one is not actually called in this code.
def heapifyByInsert(a, n):
    """ Repeatedly insert elements into the heap.
        Worst case number of element exchanges:
            sum of depths of nodes."""
    for i in range(2, n+1):
        percolateUp(a, i)

def buildHeap(a, n):
    """ Each time through the loop, each of node i's two
        subtreees is already a heap.
        Find the correct position to move the root down to
        in order to "reheapify."
        Worst case number of element exchanges:
            sum of heights of nodes."""
    for i in range (n//2, 0, -1):
        percolateDown(a, i, n)

def heapSort(a, n):
    buildHeap(a, n)
    for i in range(n, 1, -1):
        swap(a, 1, i)
        percolateDown(a, 1, i-1)
```



Recap: HeapSort: Build Initial Heap

- Two approaches:
  - for i = 2 to n
    percolateUp(i)
  - for j = n/2 downto 1
    percolateDown(j)
- Which is faster, and why?
- What does this say about overall big-theta running time for HeapSort?