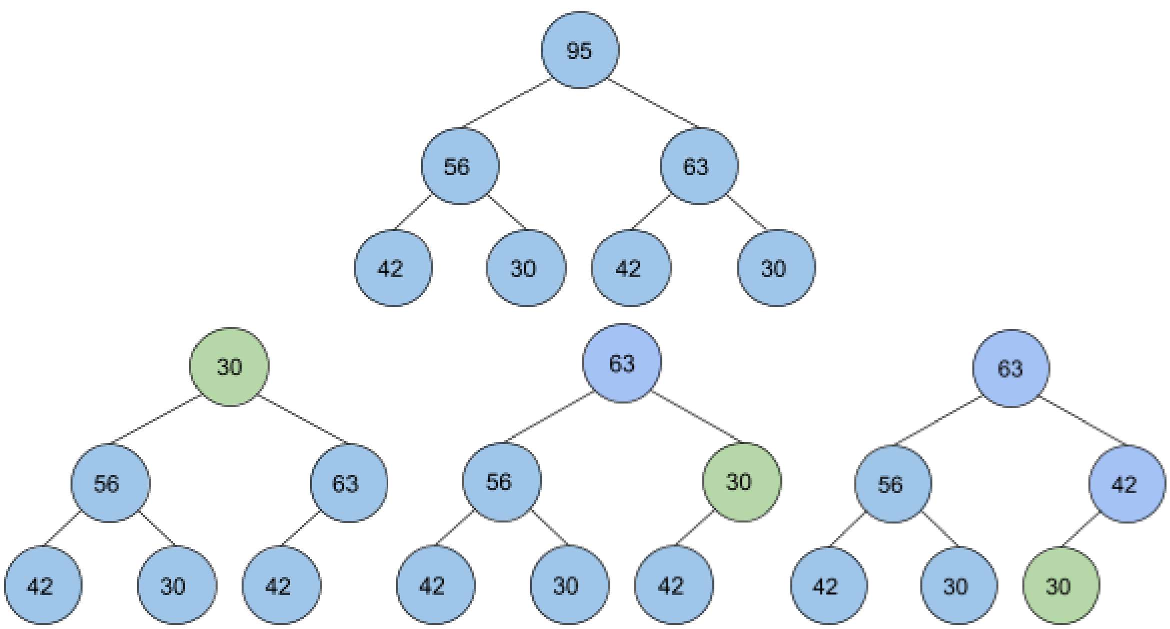
CS 1332 - Week 05

Monday

<u>Topics</u>

I. Heaps review
   A. Heaps are binary trees that are complete; children are either greater than their parents (min-heap) or less than their parents (max-heaps)

II. Removing from a Heap
   A. We'll only ever remove the root of the heap (the min or max element)
   B. Just like adding, we'll maintain the shape before handling the order
   C. Algorithm:
      1. Remove the data from the root (save it somewhere to be returned)
      2. Move the last element in the array (the element at index size) to the root
      3. Heapify-down (down-heap)
         a) Compare the new data at the root to its children
         b) If this data is not in the correct order, swap with its minimum (min-heap)/maximum (max-heap) child
            (1) You also need to handle conditions where the node only has one left child (because the tree is complete, a node will never have just a right child)
         c) Repeat until the data falls back in line with the order property or it has no children/becomes a leaf node



   D. For the purposes of this course, assume we'll never ask you to handle duplicate values in a heap

III. Build Heap
   A. The build heap algorithm takes a collection of data and builds a heap from it

B. How not to: iterate through the collection & call add on every element (O(nlogn))
C. How to:
    1. Drop everything as-is into the backing array (just like before, we're prioritizing the shape property over the order property)
    2. Starting at index n = size/2 (the first node to have children), heapify-down
    3. Decrement n and heapify-down the node at that index
        a) Repeat until n = 0
D. Efficiency: O(n) (proof)

No Content - Wednesday

Friday

Topics

I. Maps
    A. Maps are…
        1. ...collections of <key, value> pairs
        2. ...searchable
        3. ...unordered
    B. Keys are unique (no duplicate keys can exist in one Map) and cannot be changed, i.e. you can't change the key of an entry currently in the map
    C. Maps store and retrieve using the key to identify entries
    D. Values can change, you can change the value of an entry
    E. Values can have duplicates
        1. The entries <A, 3> and <B, 3> can exist in the same map, but the entries <A, 3> and <A, 5> cannot exist in the same map
    F. Hash function
        1. What it returns - an integer value (the hashcode) that represents the key
        2. Why we need it - the hashcode is used to place the map entry into the backing array and then used to search for an entry when the key is known
        3. How we get it - every Object has a .hashcode() method; this method can be customized for any Object you create, but make sure they are relatively unique (i.e. don't return 1 for every instance)
        4. How to use it - compress the hashcode to fit within the bounds of the backing array: index = hashcode % backing array length
    G. Collision strategies → what if 2+ entries hash & compress to the same index?
        1. External chaining - uses a singly linked list at each index so everything that hashes to the same index will be in a linked list at that index
            a) Example - we want to put <phone number, athletic department> key-value pairs into a map with a backing array of length 100

v - value	H(k) - hashcode	H(k) % 100 - index
Men's Basketball	4424	24

Women's Basketball	5406	06
Baseball	2261	61
Public Relations	5445	45
Operations	6668	68
Golf	0961	61

b) We don't want everything to be stored at the same index, so we need to periodically resize when we hit a certain max load factor
   (1) Load factor = size / capacity = (the number of map entries in the map) / (the capacity of the backing array)
   (2) Usually the maximum load factor is around 75%
   (3) Make the capacity of the array prime to reduce collisions

2. Probing Collision Strategies
   a) Linear probing
   b) Quadratic probing
   c) Double hashing

II. HashMaps
   A. HashMaps are subsets of Maps, they inherit all the behaviors and qualities of Maps