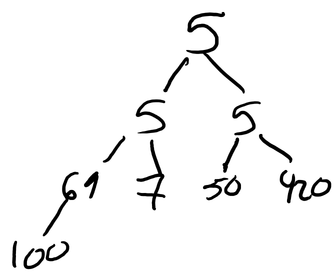
2019-10-31 Priority Queues

Thursday, October 31, 2019 8:56 AM

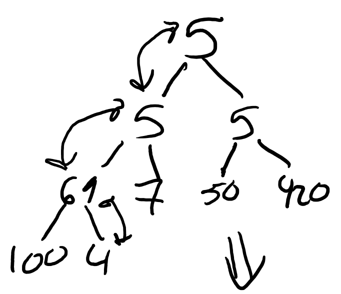
- Unlike a normal queue, items in a priority queue don't come out in the order in which they were inserted (TOTALLY NOT FAIR!)
    - Instead some algorithm programmed by "the man" determine when items come out of the queue
- It is possible for an item to go into the queue and never come out (WHOA!)
- In other words, the "most important" thing always comes out first
    - [Max queue] - The largest thing comes out first (C++ default)
    - [Min queue] - The smallest thing comes out first (textbook default)
- Lots of versions of these things, but they all adhere to this principle
- Binary Min Heap is a binary tree with two rules
    - The tree must be complete
    - (recursive) A node's parent is more "important" than the node

Example Min Heap



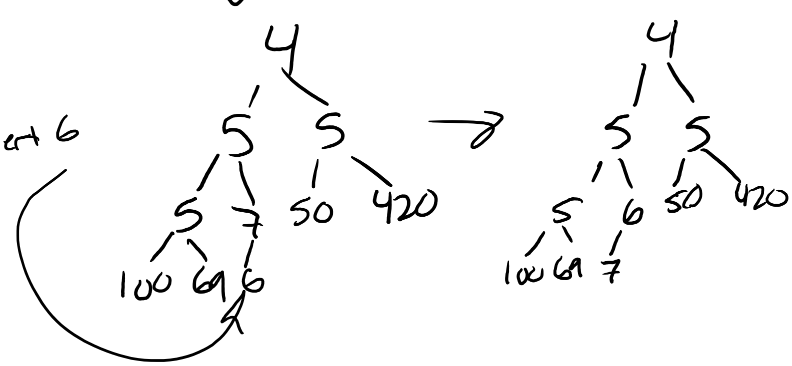
Binary Heap Inserts

1. Insert new item in the bottom of heap such that completeness is maintained.
2. That might have thrown the heap out of wack (NOT COOL)
    a. Working from that location, swap item with parent. Continue this process until heap properties are satisfied



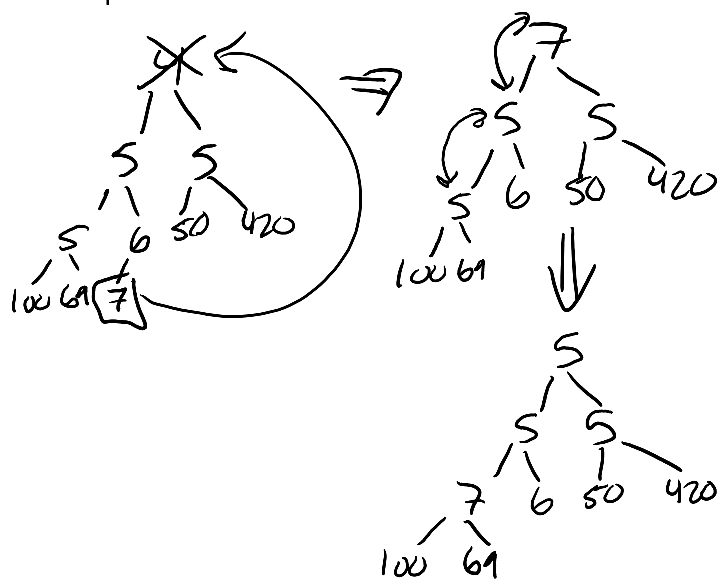
insert 4

insert 6



Removing an item from a min-heap

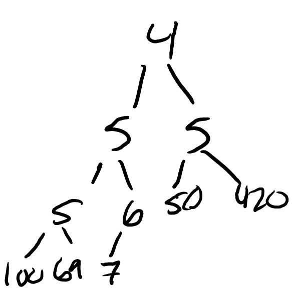
1. We always dequeue the root
2. Conceptually, we now have a "hole" at the top of the tree
3. Select as a temporary root the bottom-right most node in the tree. This maintains completeness rule
4. Usually, this breaks rule #2 (most important on top)
    a. While this value is less important than at least one child, swap with most important child



Implementing a priority queue (PQ) using different data structures

- Standard vector
    - FindMostImportant: O(N) first time, O(1) thereafter
    - Dequeue: O(N)
    - Enqueue: O(1)
- AVL Tree
    - FindMostImportant: O(LogN)
    - Enqueue: O(LogN)
    - Dequeue: O(LogN)
- Min Heap
    - FindMostImportant: O(1)
    - Enqueue: O(LogN)
    - Dequeue: O(LogN)

How does one program a binary heap?



- Using BinaryNode class before presents problems:
    - How do you track the "bottom right" in the tree
    - Can't track who our parents are
- Instead of using a LL-based implementation, we use a vector-based implementation
- Vector-based implementations turn out to be super sweet when the tree is complete

4	5	5	5	6	50	420	100	69	7
0	1	2	3	4	5	6	7	8	9

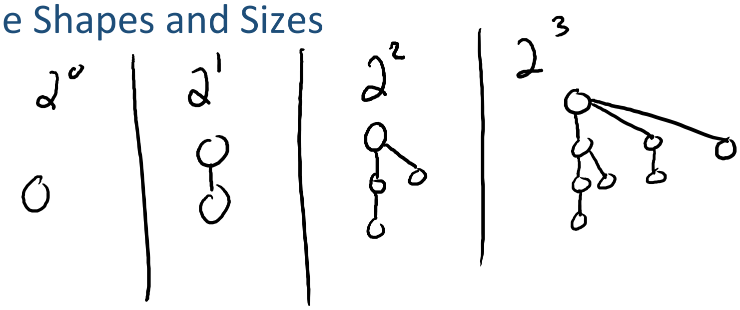
With vector-based trees we use math rather than pointers to navigate the tree
Left Child = 2 * index + 1
Right Child = 2 * index + 2
Parent: floor([index - 1] / 2)

- Extra cool factor: When complete, vector-based trees take less space (green!)
    - Vectors: one box per value
    - LL-based: one box for value, 2 boxes for pointers

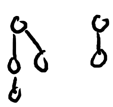
Binomial Heap

- A LL-based implementation of a priority queue
- Binomial heaps are comprised of multiple trees
    - We call this collection of trees a forest
- Each tree in a binomial forest has a unique shape
    - Each tree size is of some power of 2
    - Tree size patterns repeat recursively

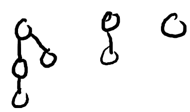
Tree Shapes and Sizes



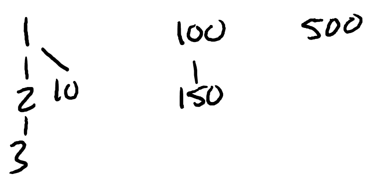
Binomial forest of size 6



Binomial forest of size 7



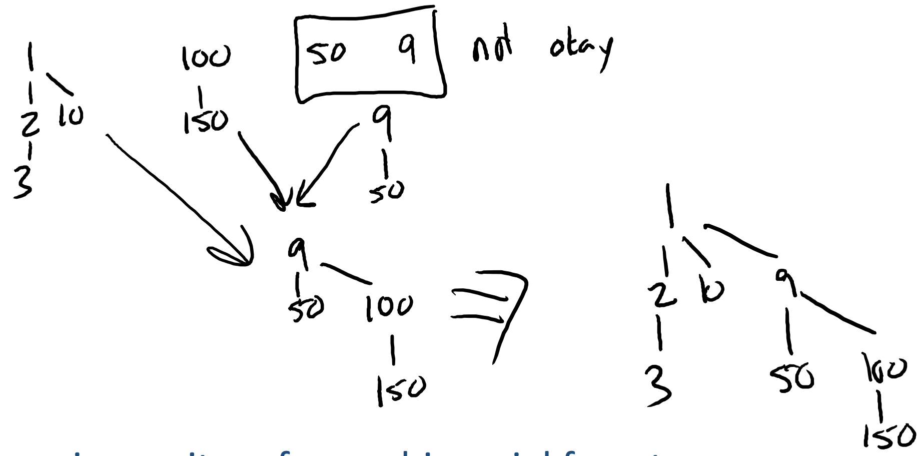
- Each tree in the forest follows heap rules (everything below is less important)



Adding element to binomial heap

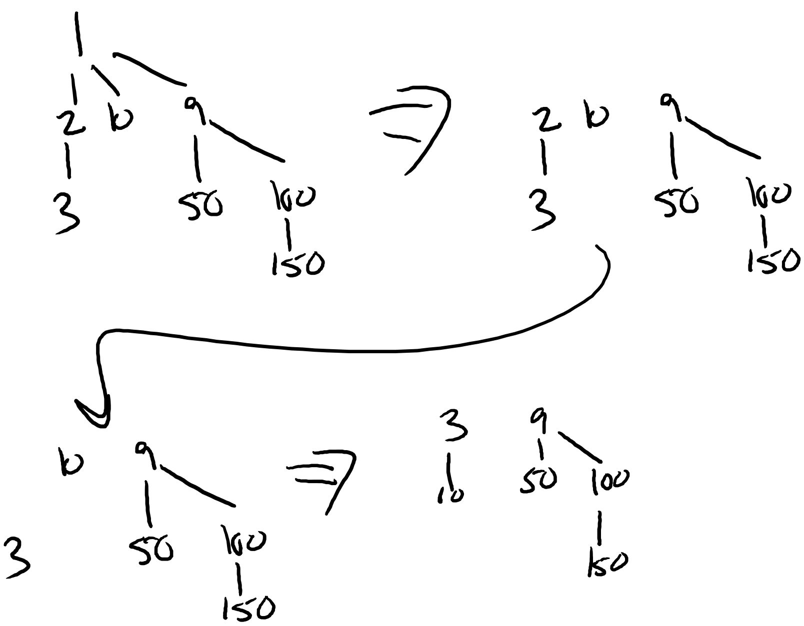
1. Add a new tree to binomial forest of size 1
2. If this violates the uniqueness rule (each tree must be unique size), merge trees until each tree is unique

What if we add value "9" to above tree?



Removing an item from a binomial forest

1. Pop off whatever tree root node is the smallest
2. If this causes a violation in uniqueness rule, merge trees



- Binomial heaps also have LogN Inserts and Removes with O(1) findMostImportant
- Unlike binary heaps, which have O(N), binomial heaps can be merged in LogN