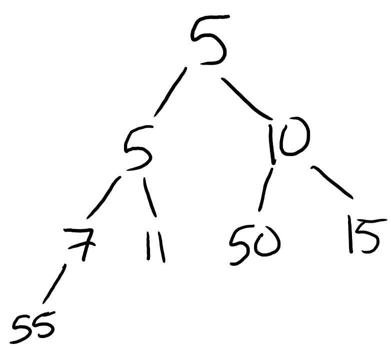
2019-04-02 Priority Queues

Tuesday, April 2, 2019 8:57 AM

- Unlike a normal queue, items in a priority queue don't come out based on when they are inserted into the queue.
  - Rather, they come out based on some algorithmic priority
- Thus, it is possible for an item to enter a PQ and never come out of the PQ
- General idea: the most important thing always comes out of a PQ
  - [max queue] - The biggest thing comes out first
  - [min queues; class default] - The smallest thing comes out first
- There are lots of versions of priority queues, all with different strengths and weaknesses
- The first, and most common, type of PQ is the **binary heap**
- A binary heap is a binary tree with two rules:
  - The tree must be complete
  - (recursive) A node's parent is more "important" than the node

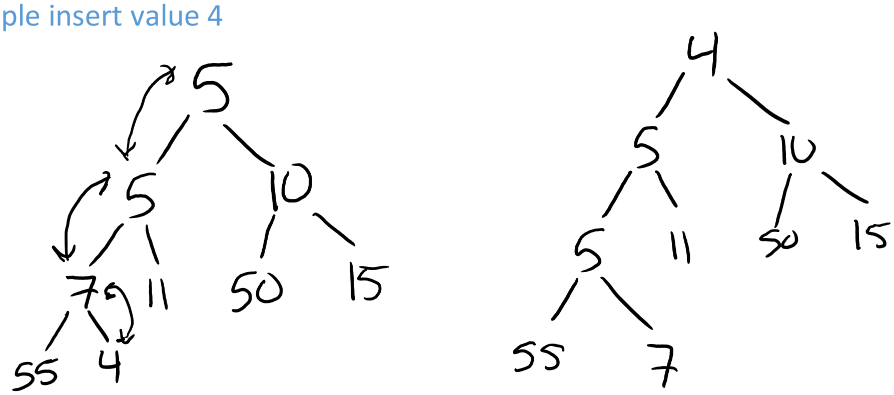
Example Min-Heap



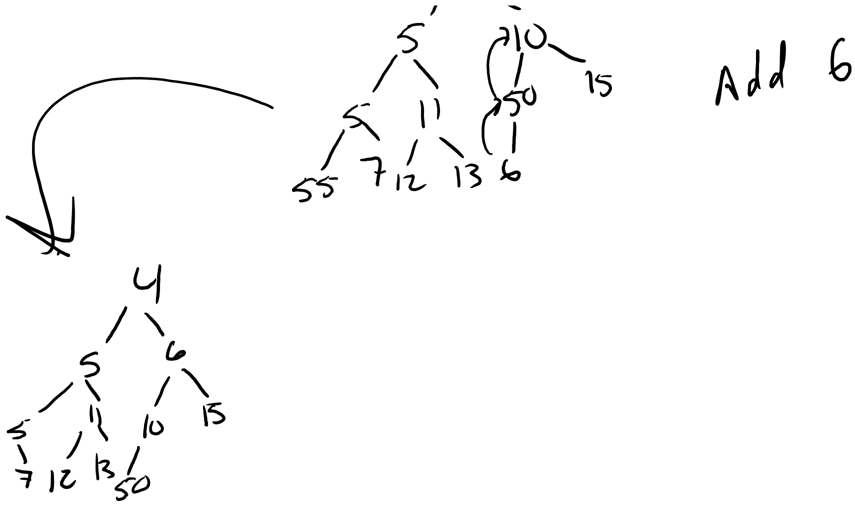
Inserting an item into a min-heap

1. Insert the new item at the bottom of the tree such that completeness is maintained.
2. While the new value is more important than its parent, swap value with parent
   a. Continue until parent is more important

Example insert value 4

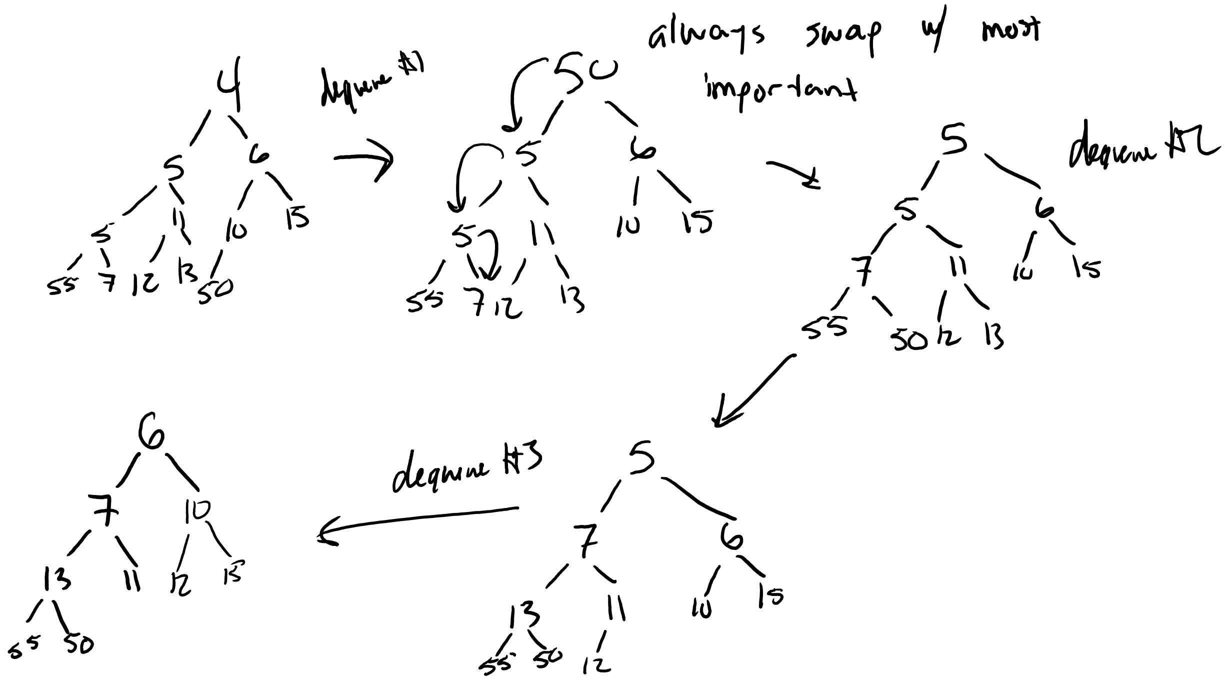


Add 6



Removing an item from a binary min-heap

1. The item to be dequeued is the root
2. Conceptually, we now have a "hole" at the top of our tree
3. Put as new root the only item in the tree that could be removed such that completeness property is maintained.
4. Doing step #3 likely results in an invalid min-heap. This value must now percolate down to its correct location.



Performance of a priority queue using different structures

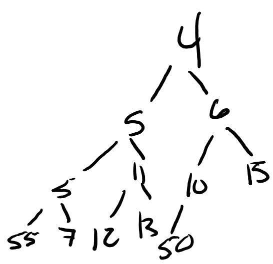
- PQs allow us to efficiently find the most important element
- Sorted Vector
  - Enqueue: Linear
  - Dequeue: Linear
  - FindMostImportant: Constant
- AVL Tree

    - Enqueue: Logarithmic
    - Dequeue: Logarithmic
    - FindMostImportant: Logarithmic
- Binary Heap
    - Enqueue: Logarithmic
    - Dequeue: Logarithmic
    - FindMostImportant: Constant

How does one program a binary heap?

- Recall, all data structures can be built using either modified vectors or linked lists
- Thus far, all of our tree structures have been built using linked lists
- It turns out that if a tree is complete, it can efficiently be represented using a vector

  Representing as a vector:



4	5	6	5	11	10	15	55	7	12	13	50
0	1	2	3	4	5	6	7	8	9	10	11

Rather than traversing the tree w/ pointers, we use math instead
Left Child: 2*index + 1
Right Child: 2*index + 2
Parent: floor[(index - 1) / 2]

Why use a vector

- Vector-based implementations allow us to quickly find bottom-right most element in tree for enqueue (last element in array)
- Vector-based implementations allow us to easily find our parent (allows for easier bubble up swapping)
- When complete, vector-based implementations take up less memory
    - Vector-based: one memory slot for item value
    - LL-based: one memory slot for item value, 2 memory slots for left/right pointer