Data structure possibilities for PQs

Monday, November 14, 2022 1:04 PM

Back to PriorityQueues -- how to implement a PQ?
Need 3 operations: a way to insert elts
a way to get the max elt (without deleting it)
a way to delete the max elt
(for the purposes of this lecture, we are concerned with max only; this still works if we talk about the min)

Data structures we've seen so far: Linked Lists, Arrays/ArrayLists, Trees, HashMaps/Dicts, HashSets/sets, own class out of these (e.g. different ways to represent graphs)

Want to write class PriorityQueue w/ 3 operations above

Set -- unique elements good, but how to associate priority?

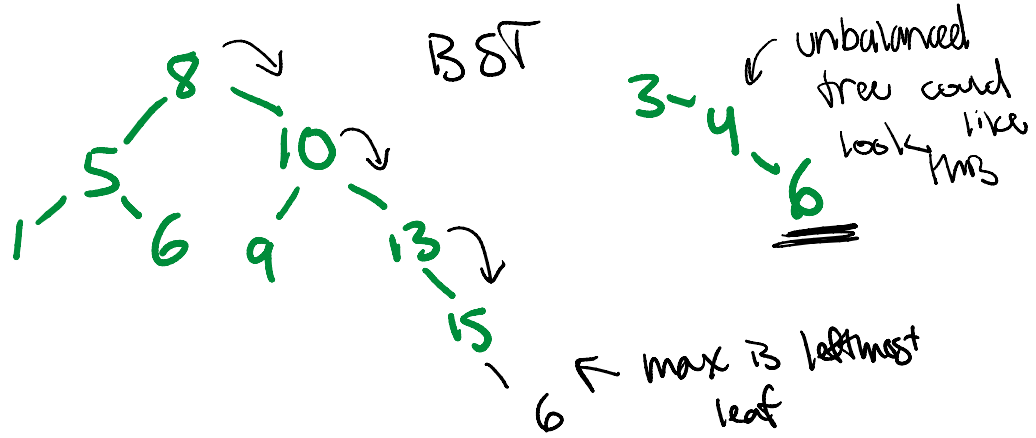
HashMap/Dict -- map an elt. to its priority (keys = elts, values = priorities)
let us get an item's priority quickly, but not get/delete max elt quickly

have "priority levels" to map to elts (keys = priorities, values = elts)
if we know max priority level, can quickly get the max elt
BUT what about the next max? and the next?

LinkedList -- sorted by priority
insert - O(N) time
getting max - O(1)
deleting max - O(1)

Trees?? How to manage priority?
insert - O(logN) for balanced tree
O(N) for unbalanced tree

getting/deleting max --
O(logN) for balanced,
O(N) for unbalanced



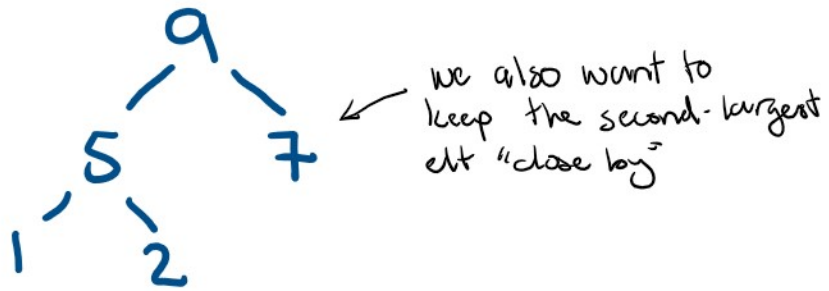
Note on terminology:
Trees are just hierarchical data structures (each node has subtrees -- two subtrees for a binary tree)
BSTs (binary search trees) are binary trees with the ordering property (everything in left subtree is smaller than the root and everything in the right subtree is bigger than the root)

All BSTs are binary trees, not all binary trees are BSTs

Heaps

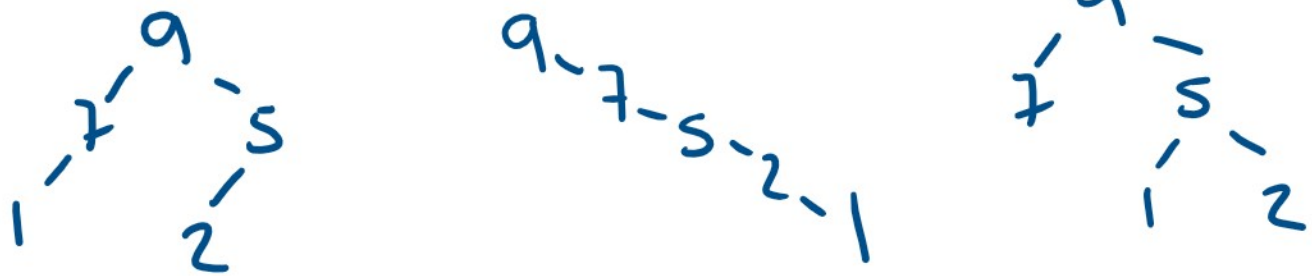
Thursday, November 17, 2022 11:03 AM

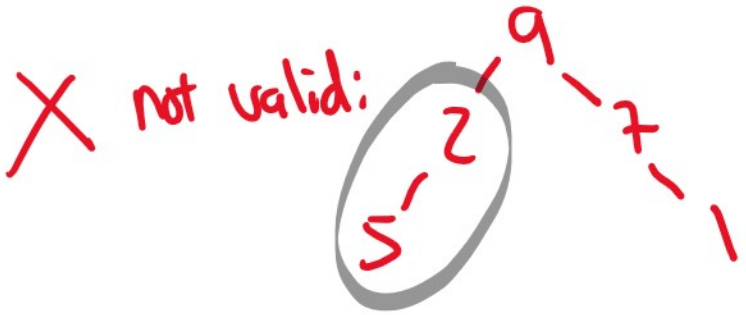
What if relax the rules a bit?
What if we keep the max item at the top?
Let's do this with the elts [1, 2, 5, 7, 9]



A (binary max) HEAP is a binary tree with the max elt at the top, and which obeys the property that each subtree is also a heap

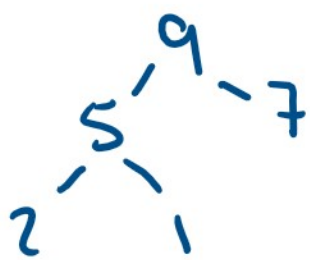
For a given set of elts, there are multiple valid heaps

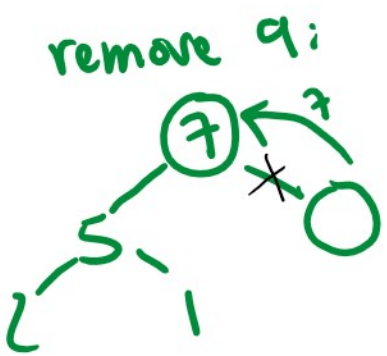


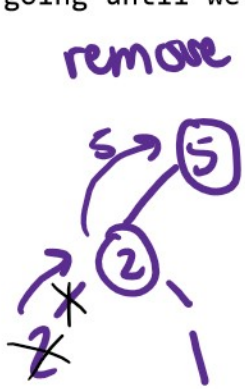


Get max elt - constant time
Remove max elt - ????
Insert - ???

When we remove an elt, we create a "hole" where that elt used to be. We want to swap in the second-largest elt, and keep going until we reach a leaf





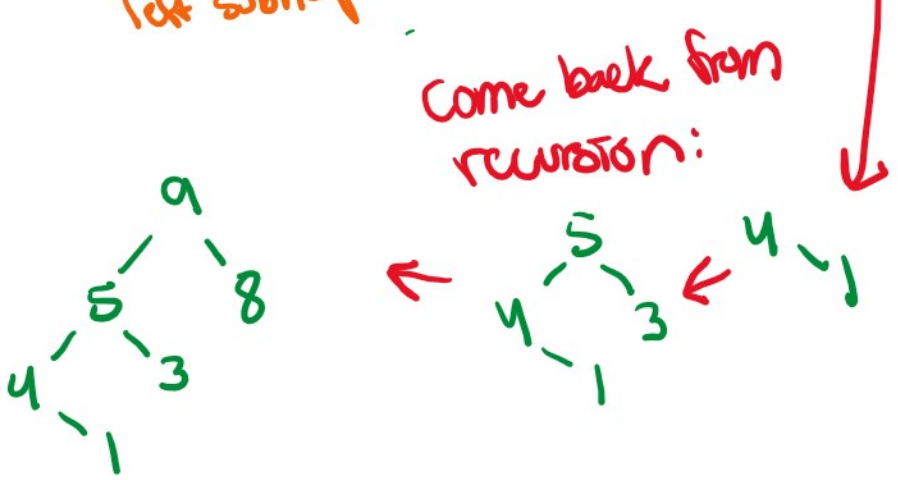


Pseudocode:

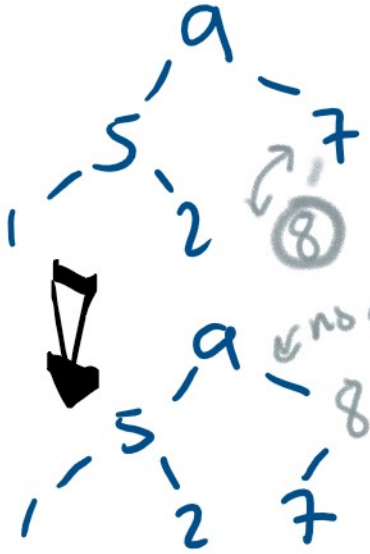
```
get_max() -> int:
   return the root of the heap

remove_max():
   child1 = left_subheap.get_max()
   child2 = right_subheap.get_max()
   if child1 > child2: (child1 goes in the top spot)
      root of this heap = child1
      left_subheap.remove_max() (or remove if leaf)
   else:
      root of this heap = child2
      right_subheap.remove_max() (or remove if leaf)
```

Runtime: O(logN) for a balanced heap
O(N) for an unbalanced heap

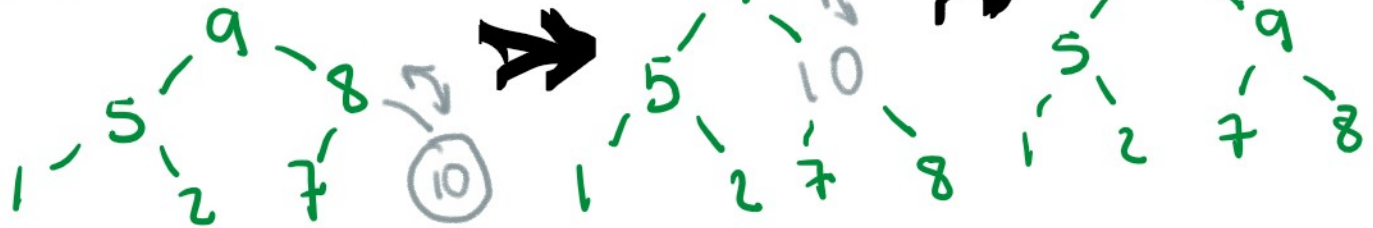


What about insertion?

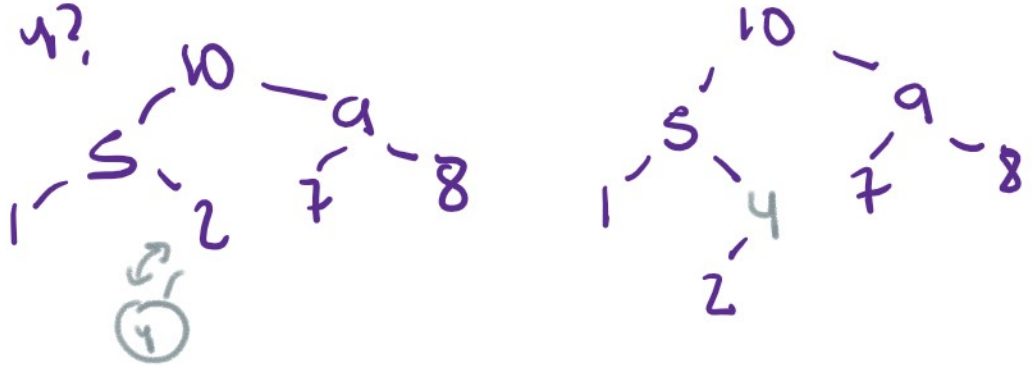


insert 8?
- find an empty spot (leaf)
- move it up until: parent is bigger OR reach the top

now insert 10?



now insert 4?



For a BALANCED HEAP:
get_max: O(1)
delete_max: O(logN)
insert: O(logN)

Unbalanced:
get_max: O(1)
delete_max: O(N)
insert: O(N)

Closing questions:
- how to find empty spot for insertion?
- how to keep heap balanced so that we can guarantee logN runtime?