Priority Queues, Heaps, Heapsort

Introduce PQ ADT:

* A priority queue is a new abstract data type that is similar to a regular stack or queue, but each item in the structure has a numerical **priority** associated with it. This priority controls the order in which the items come off of the queue.
* Recall that with a stack or a (regular) queue, the order that things come off the stack or the queue is 100% determined by the order than they go in.
  + Stack = last in, first out.
  + Queue = first in, first out.
* With a PQ, items either come out of the PQ in order of their priorities highest to lowest, or lowest to highest, depending on how you set up the PQ. For consistency with the book, we will assume our priority queues remove items with LOW priorities before items with HIGHER priorities.
* This is how waiting lines work when you take a number—lower numbers are for earlier people.

Define PQ ADT:

* Structure: A collection of items, each one associated with a numerical priority.
* Operations: insert(item, priority): adds a new item with the associated priority to the PQ.
* deleteMin(): find, remove, and return the item that currently has the HIGHEST priority in the PQ.
* Other common operations: change the priority of a specified item, return size of the PQ, find and delete a specified item, join 2 PQs together.
  + sometimes these are split into retrieveMin() and deleteMin().
* Notice that like a stack and a queue, there's exactly one way to put things in (insert) and one way to get things out. These operations are called different things in different books and programming languages. Java has a built-in priority queue where insert is called offer, and deleteMin is called poll.

When would a PQ be useful:

* Metaphorically, it’s useful in situations where you have a bunch of things waiting for a limited set of resources, but you need to prioritize them. So in that sense, it’s more related to the QUEUE adt than the STACK adt, b/c we normally think of the regular QUEUE adt as a bunch of people in line waiting to be helped or served or given a resource.
* So for a real world example, recall that for the Queue ADT, we often think of the metaphor as people waiting in line at a Bank or a Store. The order that they walk through the door is the order that they are served, no cuts, no buts, no coconuts, no exceptions.
* But say the situation is now changed to a hospital emergency room. Clearly now the order that people should be helped cannot be dependent just on the time they walk through the door, it should be in order of how badly they are hurt.
  + So people can metaphorically cut in line in front of someone in a PQ. Like if everyone in the room just has a hangnail or a cough, but then someone comes in with a life-threatening situation, they’re going to the front of the line immediately.
* In CS, one common situation where this arises is how does a computer run multiple applications at once if there’s only one processor (one CPU)? What happens behind the scenes is that the CPU rapidly switches between running operations for each application. If the CPU switches between programs fast enough, you don’t notice that they’re not all technically running simultaneously. It’s the exact same phenomenon when you watch a movie or a TV show. Each image is actually a still picture, but if you play them fast enough, your brain interprets all the pictures merging together as movement.
* Anyway, each application on your computer has a priority associated with it, and the CPU tries to prioritize applications with higher priorities. These priorities don't always correspond to what **we** think are important things, but rather things that must appear to be operating in “real-time”. For instance, streaming a movie usually has a higher priority than doing intense mathematical calculations, because you can’t slow down the movie without the person noticing, but you can slow down math and not really notice.

How a PQ works, in the abstract:

* Insert(A, 20)
* Insert(B, 15)
* Insert(C, 18)
* DeleteMin() [returns B-15]
* Insert(D, 25)
* Insert(E, 10)
* DeleteMin () [returns E-10]
* DeleteMin () [returns C-18]
* DeleteMin () [returns A-20]
* Insert(F, 23)
* DeleteMin () [returns F-23]
* DeleteMin () [returns D-25]

How to implement a PQ – simplistic ideas.

* PQ backed by an Unordered array(list) of items.
  + ArrayList of item/priority pairs.
  + Time to insert? O(n) [might have to grow the array]
  + Time to deletemin? O(n) [time to search, delete, and shift]
* PQ backed by a sorted linked list, sorted by priority.
  + insert: O(n) [insert in right location]
  + delete [O(1)]: O(1) to find (always head or tail) and delete.
* PQ backed by a BST.
  + insert: O(log n) avg, O(n) worst
  + delete max: O(log n) avg, O(n) worst.
  + Ask: BST is OK for a priority queue, but not great, because it can easily cause the tree to become unbalanced because you’re always deleting the largest thing from the tree, which means typically you’ll end up with a lot more stuff in the left half of the tree than the right.

New data structure that we commonly use to implement a PQ: HEAP (or a binary heap to distinguish from other kinds of heaps).

* Abstractly, heaps are another kind of binary tree. Just like a BST is a binary tree with particular rules about how the nodes must be organized, a HEAP has a different set of rules.
* So here are the rules for HEAPS:
  + A heap is a Binary Tree with the following restrictions:
  + It is complete---each level of the tree is completely filled, except possibly the bottom level, and if bottom level isn't full, all the nodes are in the left most positions.
  + It satisfies the heap-order property: the data item stored in each node is <= the data items stored in its two children. 🡨 for min heap. For max-heap, each node is >= children.
* This type of heap is called a MIN-HEAP. There is also a variety called MAX-HEAP, where the parent node must be >= both its children. The reason to use one over the other is whether you want the heap to prioritize LOWER numbers or HIGHER numbers. a max heap prioritizes higher numbers, a min heap prioritizes lower numbers.
* Write up a heap on the board: root 1, children 2 & 6. Next level 4,8,7.
* Because heaps are always complete Binary trees, there’s an optimization we can do in the way we represent a heap internally. We can certainly do it with pointers, like a normal binary tree, but we normally represent heaps in vectors/arrays instead. The reason is because all levels in a HEAP are full, there’s a nice mathematical property we can take advantage of to store all the nodes in an array with no gaps.
* What we do is number the nodes starting from 1, 2, 3, ....n. These correspond to the indices in the array where these data items are stored.

Implementing a PQ with a heap:

Sample heap to use: 34, 40, 59, 85, 45, 65

* How to write retrieveMin: ROOT NODE! O(1)! Super fast!
* deleteMin: algorithm
  + Replace data at root node with the last node in the tree.
  + This is no longer a heap!
  + We do a step called heap down (sometimes called percolate down, sift down, sink).
    - Interchange the root with the SMALLER of its two children.
  + This guarantees that the root is < its children and one of its subtrees is also a heap. The other subtree may or may not be a heap.
  + Repeat heapdown until we have a parent that is already smaller than both its children.
  + Big oh? O(log n)
* insert:
  + example: 40, 45, 59, 85, 65
  + INSERT 50
  + Add the new item into the last position in the tree.
  + We heapup!
    - Interchange with parent as long as the child > its parent.
  + big-oh? O(log n)

HEAPSORT

* Similarly how for a BST, we can get an algorithm called TREESORT, we can use a heap in a similar manner to get an algorithm called HEAPSORT.
* However, unlike treesort, we can do better than the naïve way of implementing HEAPSORT.
  + Naïve way: start off with an unsorted array.
  + Make an empty heap.
  + Insert() all elements from array into heap.
  + RemoveMax() elements from heap, one by one, and put them at the end of the array.
* BETTER WAY:
* Use a procedure that we call heapify!
* Heapify:
  + start at last node in the tree that is not a leaf.
  + Apply percolate down to convert that subtree to a heap.
  + Move to preceding node and apply percolate down there.
  + Work backwards until we reach the root.
  + Guaranteed to be a heap now.
* Demonstrate with “fake starting heap” of 35, 15, 77, 60, 22, 41
  + Start at 77. Leave alone.
  + Move to 15. Swap with child 60.
  + Move to root 35. Swap with 77. Now this is a heap.
* this is only the first half of the HEAPSORT alg. Because as an array, this still isn’t sorted.
* What we will do is notice that the largest element is at the root of the heap.
* Swap this element with the last element in the heap --- it moves to the end of the array.
  + Prune this element from the heap (metaphorically).
  + Now this is no longer a heap.
  + But this is the exact same procedure we did for deleteMax.
  + What did we do at deleteMax?
  + Percolate down to turn this thing back into a heap.
* Whole alg:
  + assume we want to sort A[1] through A[n]
  + Consider A as a binary heap and use heapify to convert to a heap.
  + for i = n down to 2
    - Interchange a[1] and a[i], putting largest element in heap in correct spot.
    - Prune that leaf
    - Percolate down from root
* Big-oh:
  + What is time of heapsort?
    - percdown – log n
    - heapify executes percdown n/2 times, so heapify is n log n.
    - Heapsort executes heapify once (n log n) and percolate down n-1 times, so n log n in total.

Slightly too much material for one day. Probably needed another 15 minutes.

Either expand to 2 days or cut something out.

Probably expand to 2 days, and add in some other sorting algorithms to flesh it out. Also can take more time by handing out source code and doing more in-depth examples on board. I probably hurried this lecture in S18.