Day 30 – Priority Queue ADT and Heaps

Chapter 09, 9.1 – The Priority Queue Interface

Chapter 09, 9.2 – The Priority Queue Implementation

Chapter 09, 9.3 – The Heap, Review Problem(s): ***10, 16***

Chapter 09, 9.4 – The Heap Implementation, Review Problem(s): ***15***

Think about the last time you participated in a bid competition, like eBay. Or when you had multiple processes running on your computer, how the OS kept track of what to run.

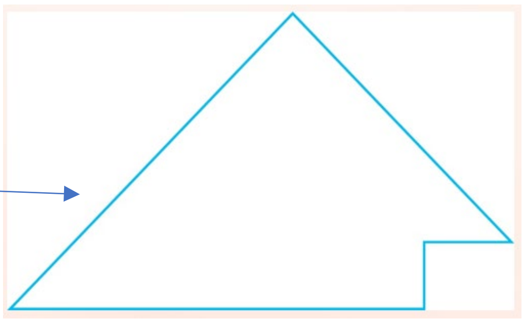
And did you happen to wonder about the underlying data structure used in these systems?

First, we present the Priority Queue ADT. Simply, only the ***highest-priority*** element can be accessed. There are many different ways to implement a Priority Queue: arrays, lists (both sorted and unsorted), BST, and a ***Heap***.

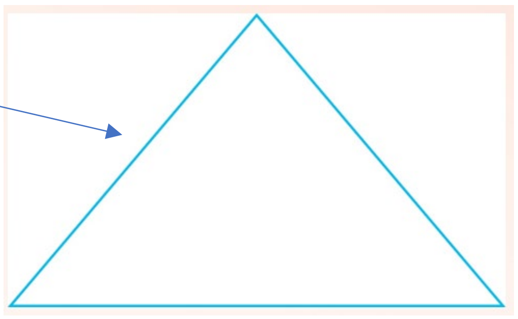
Chapter 09 PPT, pg 01-05.

We'll focus on the Heap.

First, we define a ***Complete*** binary tree → filled top-to-bottom, left-to-right.

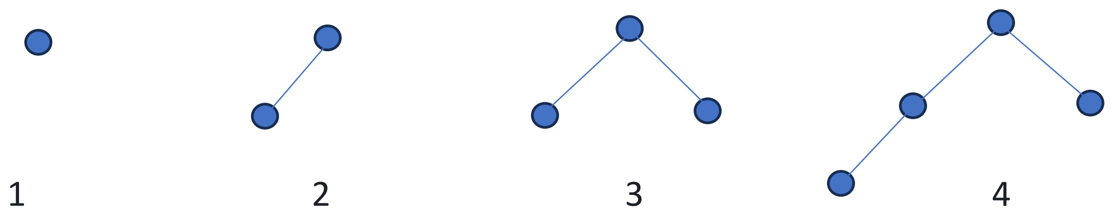


We note that a ***Full*** Binary Tree is a Complete binary tree with the last row having the max number of elements.

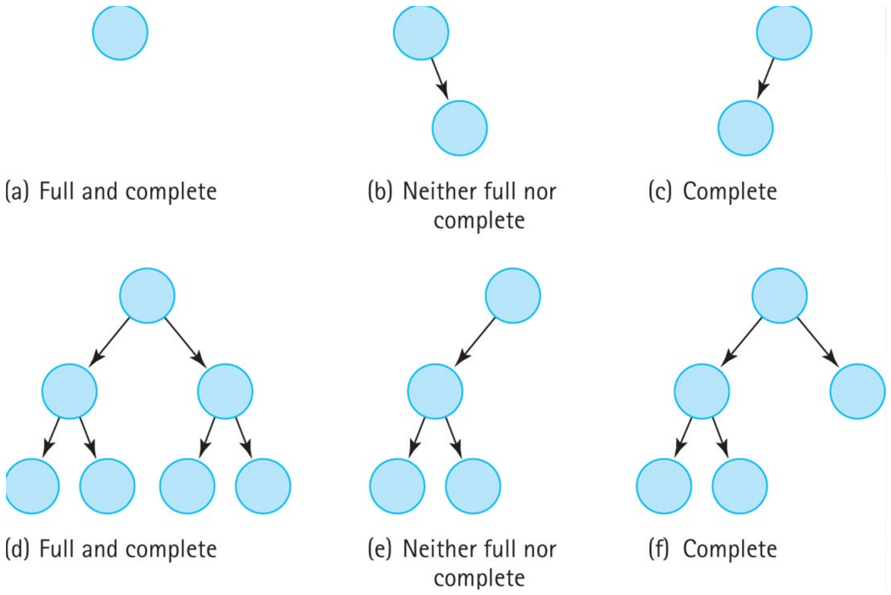


Note, we are talking about the structure or shape of the tree, but we will ***NOT*** be discussing Binary ***Search*** Trees. A Heap will not satisfy the Binary Search Tree property.

Without concerning ourselves with the values of the elements, simply the shape, what do Heaps of size 1, 2, 3, and 4 look like?



What about the following shapes? Give whether they have shapes which would be valid for a heap. If not, give a reason why.

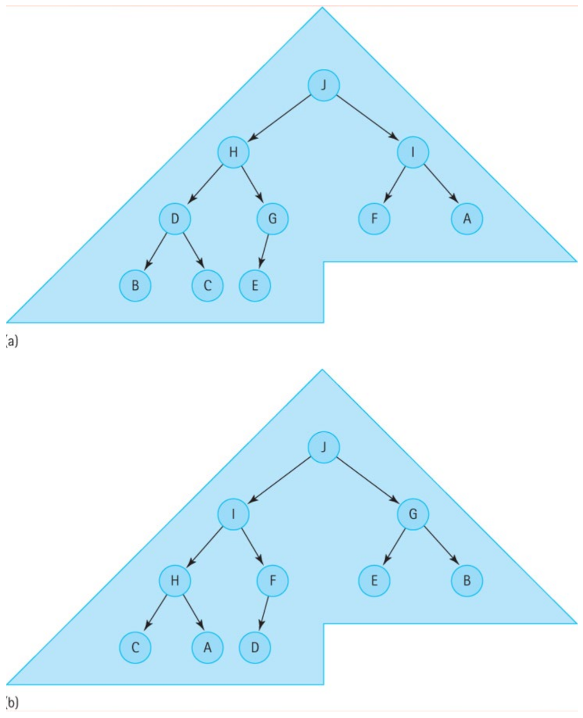


Now that we have established the shape of Heaps, we will explore the ordering property for Heaps.

A binary tree satisfying the Heap Property has:

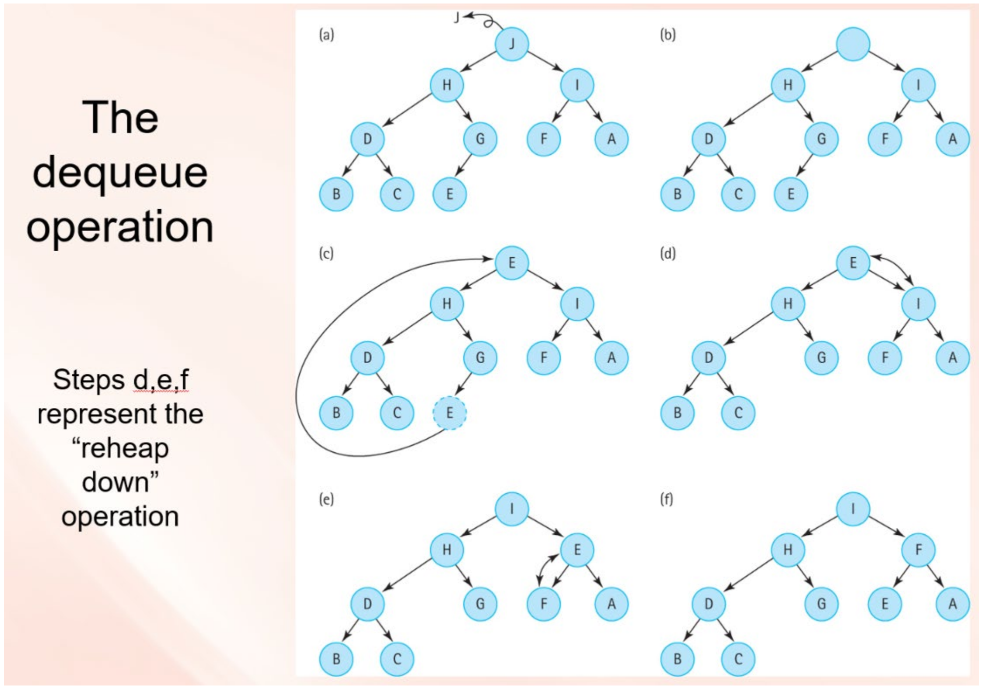
- The ***shape property***: the tree must be a complete binary tree.
- The ***order property***: for every node in the tree, the value stored in that node is greater than or equal to the value in each of its children nodes.

With that in mind, and using the expected alphabetical ordering, give two different Heaps containing the letters ***A*** through ***J***.



Chapter 09 PPT, pg 06-09.

Finally, we consider how to add and remove elements from a Heap, while maintaining the heap's shape and order properties.

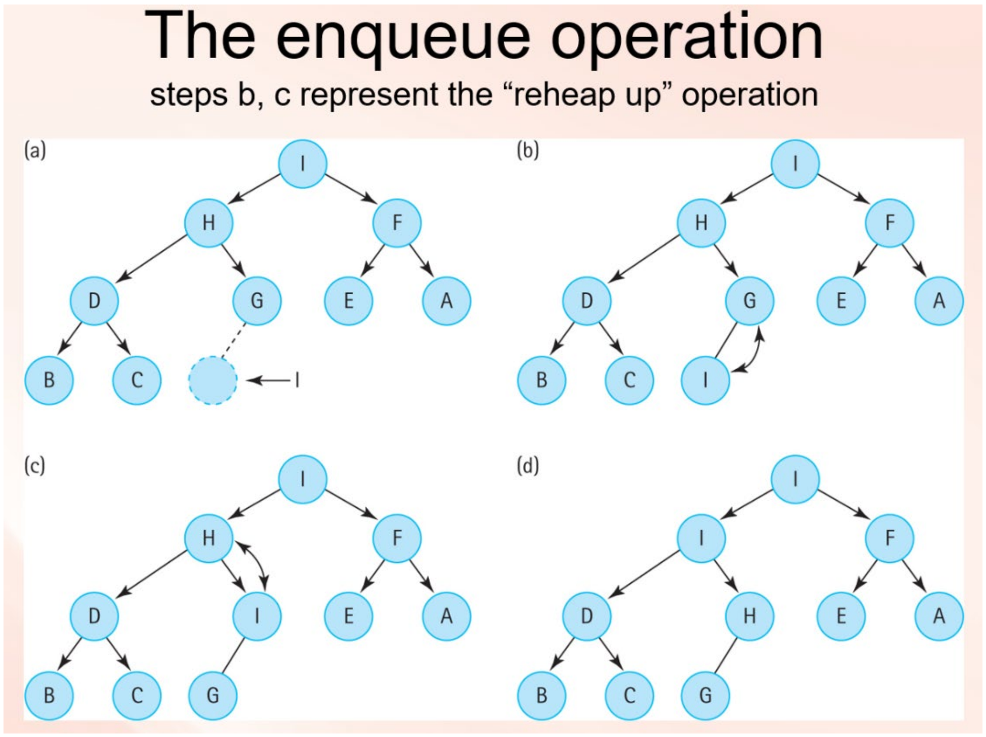

The dequeue operation

Steps d,e,f represent the "reheap down" operation

The name of the method to remove is `dequeue()`.

ReHeap Down (btw, a great band name…) is when the last element is promoted to the empty root position, and swaps are made if the largest child is larger.

Similarly, inserting uses `enqueue()`, and ***ReHeap Up*** is when the inserted element is put in the last position of the Heap and then promoted up until it is smaller than the parent of the current subtree.


The enqueue operation

steps b, c represent the "reheap up" operation

Chapter 09 PPT, pg 10-12.

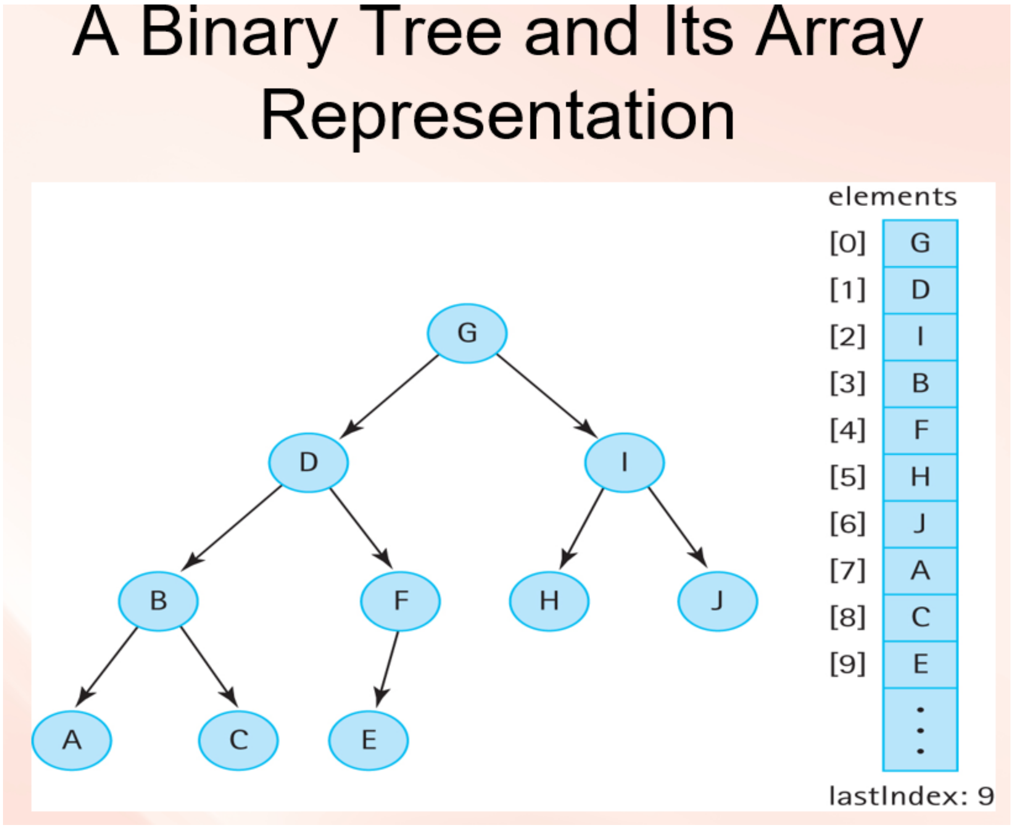
Heaps are most often implemented with arrays.

A binary tree can be stored in an array in such a way that the relationships in the tree are ***not physically represented*** by link members, but are ***implicit*** in the algorithms that manipulate the tree stored in the array.

We store the tree elements in the array, level by level, left-to-right. We call the array `elements` and store the index of the last tree element in a variable `lastIndex`.

The tree elements are stored with the root in `elements[0]` and the last node in `elements[lastIndex]`.

We follow this approach in the Heap implementation of a Priority Queue ADT.



The text provides an in-depth exploration of the Heap implementation.

Chapter 09 PPT, pg 13-26.

In particular, we compare algorithmic efficiency for various implementations:

	enqueue	dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(1)$
Binary Search Tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$