Archived Content

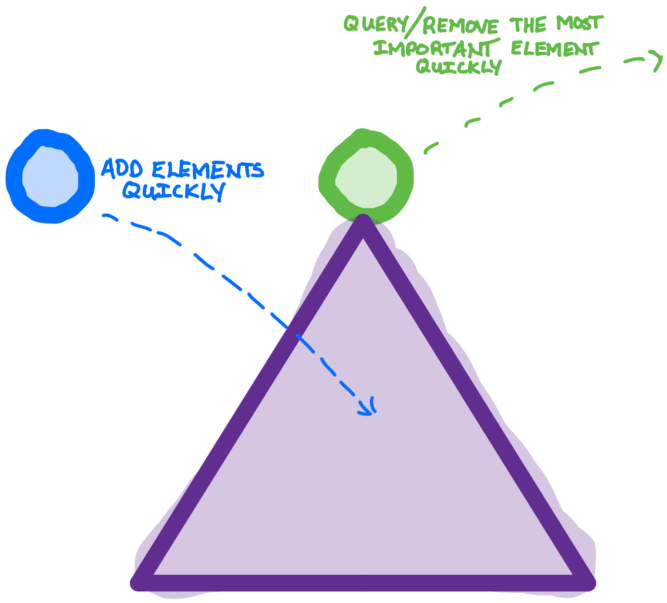
This website is an archive of the Spring 2019 semester of CS 225.

➔ **Click here to view the current semester.**

Back to Notes

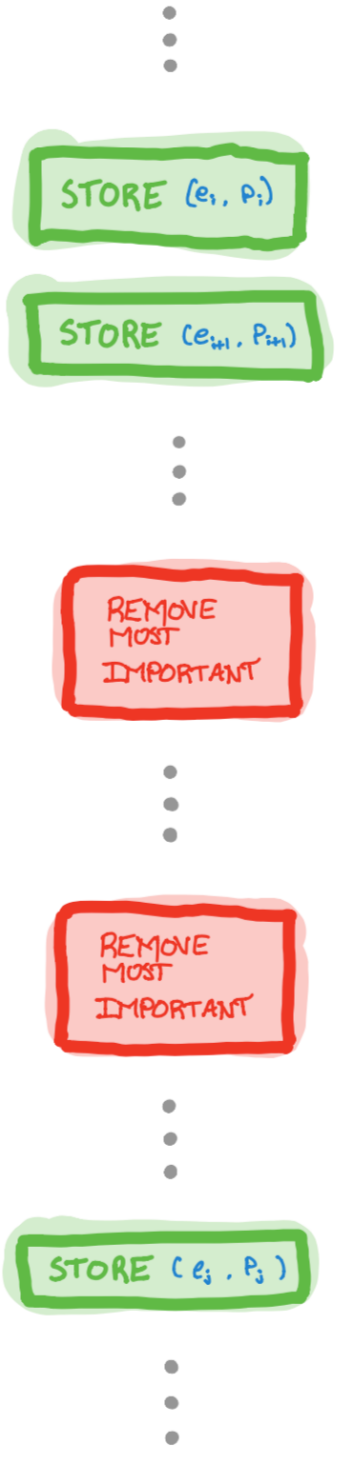
Heaps

by Eddie Huang



A heap

Motivation



A sequence of stores and removal of elements

Let's say a set of at most `n` elements, each with an associated priority p_i, is given to you one element at a time. At any point in time, you may be asked to remove the most important (highest priority) element that you currently hold so far. How would you do it?

A naive solution may be to maintain an array of size `n` and simply append new elements to the end of the array. Whenever there is a query, you simply scan through the array to find the most important element. The query would take linear (`O(n)`) time and if there were `k` queries in total, then the combined query time would be `O(kn)`. The total time would therefore be `O(n+kn)`.

We can do better with heaps

Features of Heaps

A heap allows you to store elements and query the most important elements quickly, which is usually logarithmic time.

Heaps are represented as trees where each node is an element with an associated `priority` or `weight`. There is only one property that the tree must satisfy:

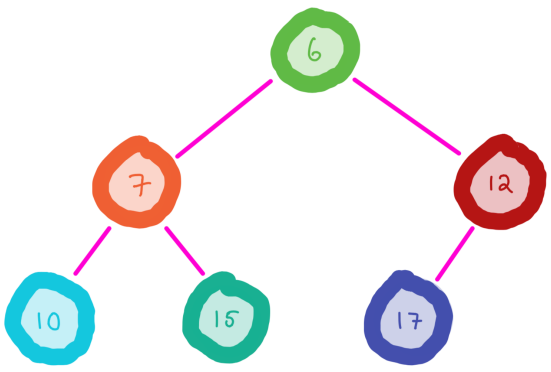
The parent has higher priority than any of its children

Therefore, the root should always have the highest priority.

In the example above where we are storing elements and querying the most important elements at random times, the total runtime would be `O(n log(n) + k log(n))`, where storing elements and removing the most important elements both take `O(log n)` time. This runtime is faster if the number of queries `k` is larger than `O(log n)`

Binary Heaps

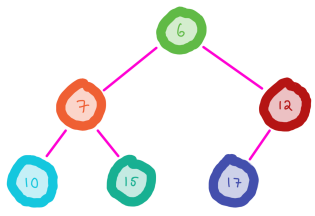
A heap is an abstract data structure and there are many data structure versions of heaps such as Fibonacci heaps or Binomial heaps that have their own advantages and disadvantages. Binary heaps are one of the most common, simplest heap data structures, and they are also the ones that we cover in this course. Binary heaps are complete binary trees that satisfy the heap property. Since they are complete trees, their heights are always logarithmic to their number of nodes.



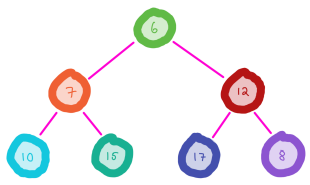
A binary min-heap

Priority is often expressed as a number in which the order of priority can either be the minimum or the maximum between two numbers. Thus, we use the terms `min-heap` or `max-heap` to name numerical binary heaps.

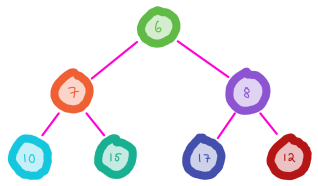
Insertion



A binary min-heap



Inserting a new element

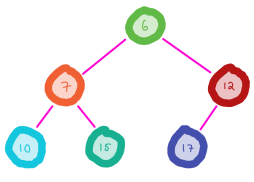


`Heapify up`: Repeatedly swap with parent until heap property is satisfied

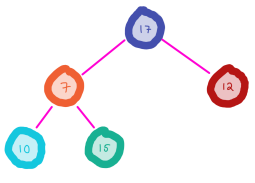
When inserting a node in a binary heap, we add it to the next leaf position such that the heap remains a complete tree. Until the new node has lower priority than its parent, we repeatedly swap this node with its parent, which will then restore the tree's heap property. This repeated swapping is called `heapify up`.

The number of swaps is at most the height of the tree, so insertion takes logarithmic time.

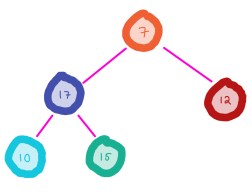
Removal



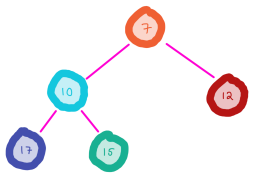
A binary min-heap



Removing the most important element and replacing it with the rightmost leaf



`Heapify down`: Repeatedly swap with higher priority child until heap property is satisfied

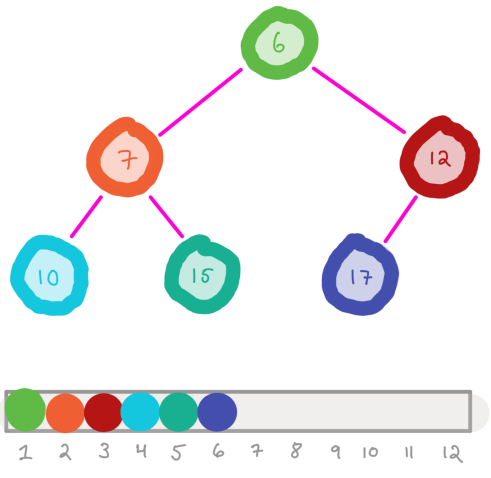


`Heapify down`: Repeatedly swap with higher priority child until heap property is satisfied

To remove the highest priority element, we simply remove the root and move the rightmost leaf in its place. Until this former leaf node has higher priority than both its children, we repeatedly swap this node with the child of higher priority, which will then restore the tree's heap property. This repeated swapping is called `heapify down`.

The number of swaps is at most the height of the tree and so removal takes logarithmic time.

Implementing complete trees with arrays



Representing a complete tree with a 1-indexed array

A complete tree can be implemented with an 1-indexed array where the $[a = 2^i+j,\ j < 2^i]$'th index of the array, represents the j'th node of the i'th level of the tree.

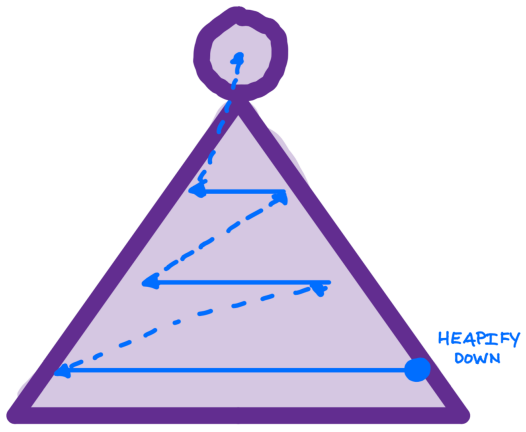
- The parent is at the $\lfloor a/2 \rfloor$'th index.

- The left and right child are at the `2a`'th index and `2a+1`'th index respectively.

Since heaps are complete trees, you can implement them with arrays as well.

> You can also model complete trees with a 0-indexed array, but it's arguably not as elegant

Turning an unordered complete tree into a binary heap



Turning an unordered complete tree into a heap

The most efficient way of turning an unordered complete tree into a binary heap is to `heapify down` all the nodes, starting with the lowest level of the tree and working your way up.

This takes *linear* time

Applications

Priorty Queues

Heaps can be used to implement priority queues, which are abstract data types that always dequeue the element with the highest priority.

> Abstract data types are more abstract than data structures in that simply describe the behavior of what they do and give no indication of the conceptual model

Heap Sort

Heaps can be used to sort elements as well. Simply add all elements to the heap and then dequeue them. The result will be the elements coming out in order.

Let `n` be the number of elements. With a binary heap the runtime of each insertion is logarithmic and each removal is also logarithmic, so the overall runtime is `O(n log(n) + n log(n)) = O(n log(n))`.