Notes 9-25

Cameron Chalk

1 Inserting/deleting with a heap

Recall from last lecture:

Claim 1. *If starting with a valid heap, increasing (resp., decreasing) any node i and running Heapify-Down(i) (resp., Heapify-Up(i)), the result is a valid heap. Running Heapify-Down(i) (resp., Heapify-Up(i)) takes $\Theta(\log n)$ time.*

We can design the insert operation using Claim 1. To insert, we are adding a new node at the next empty position in the heap; e.g., in the array implementation, the new element being inserted will be inserted in the first non-occupied array position.

Before inserting, imagine adding a node with value ∞ in this position; because this node is a leaf on the heap, the result is still a heap because all possible parents (or grandparents, etc.) will be less than ∞. Then we can call decrease on the node with value ∞, and decrease it to our value to insert. By Claim 1, this takes $\Theta(\log n)$ time and results in a valid heap.

To delete the value in node i, we must play a trick. To maintain the structure property, it is not simple to just delete a node in the middle of the tree/array and "fix" it afterwards. The easiest node to delete is the last node in the heap; afterwards, no restructuring is necessary. Let node j be the last node in the heap (the rightmost leaf). Store the value of j in a temporary variable, then delete node j. The result is a valid heap; then we increase or decrease the value in node i to j. By Claim 1, the result is a heap and this takes $\Theta(\log n)$ time.

Example problem: Design a data structure such that you can find the min and max in $\mathcal{O}(1)$ time and insert/delete in $\mathcal{O}(\log n)$ time. Try this on your own!

1.1 Building a heap

To build a heap, note that we can simply insert n elements each in $\mathcal{O}(\log n)$ time for a total of $\mathcal{O}(n \log n)$ time. However, there is a trick to construct a heap in $\mathcal{O}(n)$ time:

For simplicity, assume n is $2^k - 1$ for some $k \geq 1$; i.e., assume the elements will construct a full binary tree. Our input is an array A of length n in arbitrary order. Do the following:

- for $i = \left\lfloor \frac{n}{2} \right\rfloor$ to 1, run Heapify-Down(i).

To prove correctness, consider two heaps H_1 and H_2. Make a new node with some value, and make that node's children the roots of H_1 and H_2. When we call Heapify-Down on the new node, we get a valid heap. In the algorithm, we start at the second-to-last row of the heap, and work backwards; each case of Heapify-Down satisfies the above property that we are running Heapify-Down on a node which is the parent of two subtrees which are heaps.

Intuitively, this runs Heapify-Down n times, and thus should still take $\mathcal{O}(n \log n)$ time. However, we can do some better analysis. We will analyze how much work is done at each level of the tree. (Recall that the root is level 0, its children are level 1, their children are level 2, etc.) Let the last level (the level with leaves) be level $n-1$, so there are n levels in total. In level $n-1$, we do zero work; we only call Heapify-Down starting at level $n-2$. In level $n-2$, there are 2^{n-2} nodes; when we call Heapify-Down at this level, however, we must swap at most 1 times, since after 1 swap we reach the leaves. In level $n-3$, we must swap at most 2 times, etc. In general, in layer ℓ, per node, we must swap at most $h-\ell$ where h is the height of the tree. Since there are 2^ℓ nodes, the total swaps is $(h-\ell)2^\ell$. Thus the total work is:

$$\sum_{\ell=0}^{h}(h-\ell)2^\ell$$

If we let $j = h - \ell$ we have[1]:

$$= \sum_{j=h}^{0}(j)2^{h-j} = 2^h \sum_{j=h}^{0} \frac{j}{2^j} \leq 2^h \cdot 2 = 2^{h+1}$$

And since the number of nodes in is n, in a full binary tree we have:

$$n = \sum_{\ell=0}^{h} 2^\ell = 2^{h+1} - 1$$

And so the runtime we showed above is $\mathcal{O}(n)$.

2 Graph algorithms

There are several ways to represent graphs in data such that algorithms can operate on them. Adjacency matrices! Adjacency lists! Adjacency matrices allow queries of "is i connected to j" in constant time, but if the graph is "sparse", i.e., if there are one million nodes but each has degree at most 10, most entries are 0 in the one million $\times$ one million matrix. Adjacency lists are better on sparse graphs, but if some node's degree is large, then asking the query "is i connected to j" will take a long time. Adjacency lists are also more useful when you want to "explore" or "walk" around the graph; we will see why soon.

[1] Using $\sum_{j=0}^{\infty} \frac{j}{2^j} = 2$ to obtain the inequality.