Priority Queues

A priority queue

- is a collection of records with keys
- allows the operations:

Insert: a record is added to the collection

FindMax: a record with maximum key is returned

DeleteMax: a record with maximum key is deleted

If implemented by unsorted array:

- Insert $\in \Theta(1)$
- Find/DeleteMax $\in \Theta(n)$

If implemented by sorted array:

- Find/DeleteMax $\in \Theta(1)$
- Insert $\in \Theta(n)$ (as need to push)

We shall aim at all operations sublinear

The Heap Property

A heap is a rooted tree (for now of arbitrary shape) that satisfies the heap property:

if q is a child of p then $key(p) \geq key(q)$

Thus parents have keys at least as great as their children.

- sometimes we instead want the dual property

Heap property may be violated if the key of a node n

- decreases: then we have to "sift down" n:

  while n has child with greater key
  swap n with the child with greatest key

- increases: then we have to "percolate" n:

  while n has parent with smaller key
  swap n and its parent

Worst case running time: in $\Theta(h)$ with h the tree height.

Representation of Heaps

We use rooted trees that are binary and

- complete: balanced, except that some rightmost leaves may be missing

We can use an array A to represent a complete binary tree, with

- root $= A[1]$
- parent$(A[i]) = A[\lfloor i/2 \rfloor]$ (if $i > 1$)
- left_child$(A[i]) = A[2i]$ (if exists)
- right_child$(A[i]) = A[2i+1]$ (if exists)

The size n and its height h are related:

- $n \leq 2^{h+1} - 1$, and thus
- $h = \lfloor \lg(n) \rfloor$

Priority Queues by Heaps

With heap implementation:

- FindMax: just take the root; this is in $\Theta(1)$.
- Insert:
  1. insert the new node as the first free array position
  2. percolate that leaf up

  This is in $\Theta(h) = \Theta(\lg n)$.
- DeleteMax:
  1. move the rightmost bottom leaf to the root
  2. sift down the root

  This is in $\Theta(h) = \Theta(\lg n)$.

Converting Tree Into Heap

Given a complete binary tree, we want to

- convert it into a heap
- by node swapping only

Tentative approach: grow heap incrementally

- for each element, percolate it up to proper spot
- running time: $\sum_{i=1}^{n} \lg(i)$ in $\Theta(n \lg(n))$

Better approach (top-down): for each node,

1. recursively convert its child(ren) into heaps
2. then sift down the node.

Iterative implementation:

$$\begin{array}{l} \textbf{for } \mathrm{i} \leftarrow \lfloor n/2 \rfloor \textbf{ downto } 1 \\ \quad \textsc{SiftDown}(i) \end{array}$$

Running time recurrence: $T(n) = 2T(n/2) + \lg(n)$ (at least when n is power of 2) and thus $T(n) \in \Theta(n)$

Heap Sort

Given array $A[1.n]$ to be sorted, we

1. convert it into heap
2. incrementally extract solution from heap.

For part 2, we keep decrementing i while maintaining the invariant that

1. $A[i+1..n]$ consists of the $n-i$ largest elements, in non-decreasing order;
2. $A[1..i]$ has the heap property

which is

- established with $i = n$: (1) vacuously; (2) by Phase 1
- sufficient for correctness when $i = 1$

To maintain invariant:

1. Exchange $A[1]$ and $A[i]$
2. Sift down $A[1]$ in tree $A[1..i-1]$

Complexity of Heap Sort

Recall that to sort an array of n elements, we

1. convert it into a heap, in time $\Theta(n)$
2. for i from n down to 2:
   2.1 Exchange $A[1]$ and $A[i]$, in time $\Theta(1)$
   2.2 Sift down $A[1]$ in heap $A[1..i]$, in time $\Theta(\lg(i))$.

   This contributes

$$\sum_{i=1}^{n} \lg(i)$$

   which we know is in $\Theta(n \lg(n))$.

Thus heap sort has

- Time Complexity in $\Theta(n \lg(n))$
  which improves insertion sort
- Space Complexity is in-place which improves merge-sort