Heaps and priority queues

As mentioned earlier, an array $\underline{t}$ (of objects or pointers to objects) can be seen as a rooted binary tree as follows:

- The root node is $\underline{t}[0]$.
- The child nodes of $\underline{t}[i]$ are the nodes $\underline{t}[2 \cdot i + 1]$ and $\underline{t}[2 \cdot i + 2]$.
- The parent node of $\underline{t}[j]$ is the node $\underline{t}[i]$ where $i = \lfloor (j-1)/2 \rfloor$.

As usual, the actual storage allocated for the array will be greater than the size of the tree so that there is room for the tree to grow. As a result, we have an index $\underline{t}.size$ so that all nodes correspond to indices which are between 0 and $\underline{t}.size - 1$.

A max-heap is a binary tree as above where:

- The nodes can be compared (they lie in an ordered set A).
- The child nodes are $\leq$ the parent node.

In particular, the root node is maximal.

One can similarly talk about a min-heap.

Operations

The common operations on a max-heap are:

- Given an element a of A we have `push`$(\underline{t}, a)$ which inserts a in a new node position so chosen that the resulting tree is *also* a max-heap.
- We can delete the root node of a max-heap with `pop`$(\underline{t})$ to create a max-heap with one less node. This data at this node is the "return value" of this algorithm.

Both these operations are written in terms of other operations as explained below.

Possibly damaged max-heap. The max-heap property at the node p means the following. Let γ denote the parent of p (assuming p is not the root node). Let λ and ρ be its child nodes (assuming p is not a leaf node). The max-heap property at p is the requirement

$$\gamma \geq p \text{ and } p \geq \max\{\lambda, \rho\}$$

We can say that the max-heap property is "possibly up-damaged" at the node p if we don't know the relation between γ and p but $p \geq \max\{\lambda, \rho\}$ (or p is a leaf node).

We can say that the max-heap property is "possibly down-damaged" at the node p if we don't know the relation between p and $\max\{\lambda, \rho\}$ but $\gamma \geq p$ (or p is the root node).

The primary operations will fix possible damage at *exactly* one node.

- The operation siftup($\underline{t}, p$) operates on a tree $\underline{t}$ for which the max-heap property holds except that it is possibly up-damaged at p.
- The operation siftdown ($\underline{t}, p$) operates on a tree $\underline{t}$ for which the max-heap property holds except that it is possibly down-damaged at p.

In both cases, the result of these operations is a tree that satisfies the max-heap property at all nodes.

Given these algorithms, we can make push by adding the node to be inserted in the left-most unoccupied slot available for new leaf modes. This will create a tree which is possibly up-damaged at this leaf node. So after this the siftup operation will produce a max-heap as required.

Similarly, we can make pop by replacing the root node by the right-most leaf node and deleting the right-most leaf node. This will create a tree which is possibly down-damaged at the root node. So after this the siftdown operation will produce a max-heap as required. Note that pop-ping the right-most leaf node is essentially the same as deleting it.

More generally, we can change the data a at a specific node p to data b as follows. We first, compare a with b. If a is greater than b, then replacing a by b creates a tree that is down-damaged at p; so we only need to siftdown to fix it. Similarly, if the value b is greater than a, then replace a by b creates a tree that is up-damaged at p; so we only need to siftup to fix it.

We can also delete the node p as follows. We first replace the p-th node with the right-most leaf node and remove the latter node. We then apply siftdown or siftup at the p-th node depending on the comparison of the current node with the deleted node like we did for change. In other words, this operation can be considered as follows:

- Delete the right-most leaf node after saving its value as b.
- Use the change operation to replace the p-th node by b.

Algorithms

In the following algorithms we assume that a max-heap (or a binary tree) is implemented as an (expandable) array $\underline{t} = [t_0, \ldots, t_{n-1}]$ of elements that can be compared. (In Python, we can use a List data type to represent such an array.)

define siftup($\underline{t}, p$)
    Set γ *as* $\lfloor (p-1)/2 \rfloor$.
    if $\gamma \geq 0$ *and* $t_\gamma < t_p$
        Swap t_p *and* t_γ.
        return siftup($\underline{t}, \gamma$)
    return $\underline{t}$

In this the algorithm p is the index of the node at which the tree may be up-damaged and γ is the index of its parent node. Note that the possible up-damage is propagated from p to γ while removing the damage at p.

The up-damage in the max-heap ascends along branch containing p. As a result the total number of comparisons and swaps is bounded by the depth $\lceil \log_2(n) \rceil$ of the tree.

(Note that in Python, if $\underline{t}$ is implemented as a list, it will be modified in-place. So the **return** is not required.)

define `siftdown`$(\underline{t}, p)$
  Set n *to be size of* $\underline{t}$.
  Set q *as* p.
  Set λ *to be* $2 \cdot p + 1$.
  Set ρ *to be* $2 \cdot p + 2$.
  if $\lambda < n$ *and* $t_\lambda > t_q$
    Set q *to be* λ.
  if $\rho < n$ *and* $t_\rho > t_q$
    Set q *to be* ρ.
  if p *is different from* q
    Swap a_p *and* a_q.
    return `siftdown`$(\underline{t}, q)$
 return $\underline{t}$

Here p represents the node where the possible down-damage is. It's child nodes are λ and ρ. Next, note that only *one* of the nodes λ or ρ is (possibly) swapped with p. The possible down-damage is propagated from p to the node with which it is swapped while removing the damage at p.

There are two comparisons of nodes of t and at most one swap of notes of t in each call. The possible down-damage in the max-heap descends along *only* one branch. As a result the total number of comparisons is bounded by $2\lceil log_2(n) \rceil$ and the number of swaps is bounded by $\lceil log_2(n) \rceil$.

Exercise: Modify the pseudo-code above to use **while** constructions instead of **if** and recursion.

As mentioned above, the `push` and `pop` operations can be made using these two algorithms.

define `push`$(\underline{t}, a)$
  Set n *to be size of* $\underline{t}$.
  Append a *to* $\underline{t}$.
  Call `siftup`$(\underline{t}, n)$.

define `pop`$(\underline{t})$
  Set n *to be size of* $\underline{t}$.
  Save the return value a *which is set as the root element* t_0.
  Set t_0 *to be the last leaf node* t_{n-1}.
  Delete the last leaf node from $\underline{t}$.
  Call `siftdown`$(\underline{t}, 0)$.
  return a

Note that `pop` only works if the tree has at least one node!

More generally, we can `change` the value at the node at index p in the tree $\underline{t}$, or `extract` it.

define `change`$(\underline{t}, p, b)$
    Save the current value a *of* t_p.
    Set t_p *to be* b.
    if $t_p > a$
        Call `siftup`$(\underline{t}, p)$.
    else
        Call `siftdown`$(\underline{t}, p)$.

define `extract`$(\underline{t}, p)$
    Set n *to be size of* $\underline{t}$.
    Save the return value a *which is set as* t_p.
    Set t_p *to be the last leaf node* t_{n-1}.
    Delete the last leaf node from $\underline{t}$.
    if p *is* $n-1$
        return a
    if $t_p > a$
        Call `siftup`$(\underline{t}, 0)$.
    else
        Call `siftdown`$(\underline{t}, p)$.
    return a

Build

Naively, one may build a heap out of an array of data incrementally, using `push`. This will take $2\sum_{k=1}^{n}\lceil\log_2(k)\rceil$ or $O(n\log(n))$ comparisons. However, there is a better approach to `heapify` an array as follows.

Given an array $\underline{t}$ of elements of an ordered set A, we can convert it into a max-heap as follows. Starting at the right-most and deepest non-leaf node, we work our way to the left and upwards in the tree "fixing" the tree that has the given node as root using `siftdown`.

define `heapify`$(\underline{t})$
    Set n *as the size of* $\underline{t}$.
    for i *from* $\lfloor n/2 \rfloor$ *decreasing to* 0
        Call `siftdown`$(\underline{t}, i)$.

We note that the call to `siftdown` takes $2h_i$ steps where h_i is the depth of the sub-tree rooted at i. When i in one of the 2^k indices the in range $[2^k-1, 2^{k+1}-2]$ we have $h_i = h = \lceil\log_2(n)\rceil - k$. In other words, 2^k is roughly $n/2^h$. Thus, we see that the number of steps is (roughly)

$$\sum_{h=1}^{n/2} \frac{n}{2^h} 2h = 2n \sum_{h=1}^{n/2} \frac{h}{2^h} \leq 2n \sum_{h=1}^{\infty} \frac{h}{2^h} = 4m$$

Exercise: Make the above calculation more precise and calculate the precise number of steps in the worst case.

Priority Queues

One can use a max-heap to implement queues. Each node is a pair-record consisting of the queued object and a "priority". Comparison for the max-heap property is based only on the priority.

- `enqueue` is implemented by using `push` to insert a new node whose priority is 1 less than the priority of the last object queued.
- `dequeue` is implemented easily using `pop`.

In fact, since we can choose the priority at the time of insertion *and* change the priority, what we have is a "priority queue" data structure. This has the operations:

- `insert` which inserts an object into the queue with a given priority.
- `maximum` which finds the element with the highest priority.
- `extract` which extracts the element with the highest priority and removes it from the queue.
- `change` which changes the priority of a chosen element of the queue.

From the above discussion it is clear how such a data structure can be implemented using max-heap.