Recitation 5a

Priority Queues

Priority queues provide a general framework for at least three sorting algorithms, which differ only in the data structure used in the implementation.

algorithm	data structure	insertion	extraction	total
Selection Sort	Array	$O(1)$	$O(n)$	$O(n^2)$
Insertion Sort	Sorted Array	$O(n)$	$O(1)$	$O(n^2)$
Heap Sort	Binary Heap	$O(\log n)$	$O(\log n)$	$O(n \log n)$

Let's look at Python code that implements these priority queues. We start with an abstract base class that has the interface of a priority queue but no implementation except for some bounds checking on an internally stored array A. The priority queue inputs an array where the queuing elements will be stored. A priority queue implementation does not *need* to manipulate the input array directly, but doing so leads to *in-place*[1] implementations, which we emphasize here. The priority queue also keeps track of its **size** n, i.e., the number of array elements currently in the queue, where $n \leq |A|$.The `insert` function is not given a value to insert; the function will look to the item stored in `A[n]`, and incorporate it into the now larger queue. Similarly, `delete_max` does not return a value; it merely deposits its output into `A[n]` before decreasing its size.

```python
class PriorityQueue:
    def __init__(self, A):
        self.n, self.A = 0, A

    def insert(self):          # absorb element A[n] into the queue
        if not self.n < len(self.A):
            raise IndexError('insert into full priority queue')
        self.n = self.n + 1

    def delete_max(self):      # remove element A[n - 1] from the queue
        if self.n < 0:         # this implementation is currently NOT correct
            raise IndexError('pop from empty priority queue')
        self.n = self.n - 1

    def sort(A):
        pq = self.__init__(A) # make empty priority queue
        for i in range(len(self.A)):
            pq.insert()        # n x T_i
        for i in range(len(self.A)):
            pq.delete_max()    # n x T_e
```

[1] Recall that an in-place sort only uses $O(1)$ additional space during execution, so only a constant number of array elements can exist outside the array at any given time.

Shared across all of implementations is a method for sorting, given implementations of `insert` and `delete_max`. Sorting simply makes two loops over the array: one to insert all the elements, and another to populate the array with successive maxima.

Implementations

We showed implementations of selection sort and merge sort previously in recitation. Here are implementations from the perspective of priority queues. If you were to unroll the organization of this code, you would have essentially the same code as we presented before. The last implementation based on a binary heap takes advantage of the logarithmic height of a complete binary tree to improve performance. The bulk of the work done by these functions are encapsulated by `max_heapify_up` and `max_heapify_down`, which we discuss in the following section.

```python
class PQ_Array(PriorityQueue):
    def delete_max(self):        # O(n)
        super().delete_max()     # decreases self.n by 1
        n, A, m = self.n, self.A, 0
        for i in range(1, n):
            if A[m].key < A[i].key:
                m = i
        A[m], A[n] = A[n], A[m]
```

```python
class PQ_SortedArray(PriorityQueue):
    def insert(self):            # O(n)
        super().insert()         # increases self.n by 1
        i, A = self.n - 1, self.A
        while 0 < i and A[i + 1].key < A[i].key:
            A[i + 1], A[i] = A[i], A[i + 1]
            i -= 1
```

```python
class PQ_Heap(PriorityQueue):
    def insert(self):            # O(log n)
        super().insert()         # increases self.n by 1
        n, A = self.n, self.A
        max_heapify_up(A, n, n - 1)

    def delete_max(self):        # O(log n)
        super().delete_max()     # decreases self.n by 1
        n, A = self.n, self.A
        A[0], A[n] = A[n], A[0]
        max_heapify_down(A, n, 0)
```

Binary Heaps

The first part of a binary heap implementation is computing parent and child indices given an index representing a node in a tree whose root is the first element of the array. In this implementation, if the computed index lies outside the bounds of the array, we return the input index. Always returning a valid array index instead of throwing an error helps to simplify future code.

```
def parent(i):
    p = (i - 1) // 2
    return p if 0 < i else i

def left(i, n):
    l = 2 * i + 1
    return l if l < n else i

def right(i, n):
    r = 2 * i + 2
    return r if r < n else i
```

Here is the meat of the work done by a max heap. Assuming all nodes in `A[:n]` satisfy the Max-Heap Property except for node `A[i]` makes it easy for these functions to maintain the Node Max-Heap Property locally.

```
def max_heapify_up(A, n, c):              # T(c) = O(log c)
    p = parent(c)                         # O(1) index of parent (or c)
    if A[p].key < A[c].key:               # O(1) compare
        A[c], A[p] = A[p], A[c]           # O(1) swap parent
        max_heapify_up(A, n, p)           # T(p) = T(c/2) recursive call on parent
```

```
def max_heapify_down(A, n, p):            # T(p) = O(log n - log p)
    l, r = left(p, n), right(p, n)        # O(1) indices of children (or p)
    c = l if A[r].key < A[l].key else r # O(1) index of largest child
    if A[p].key < A[c].key:               # O(1) compare
        A[c], A[p] = A[p], A[c]           # O(1) swap child
        max_heapify_down(A, n, c)         # T(c) recursive call on child
```

$O(n)$ Build Heap

Recall that repeated insertion using a max heap priority queue takes time $\sum_{i=0}^{n} \log i = \log n! = O(n \log n)$. We can build a max heap in linear time if the whole array is accessible to you. The idea is to construct the heap in *reverse* level order, from the leaves to the root, all the while maintaining that all nodes processed so far maintain the Max-Heap Property by running `max_heapify_down` at each node. As an optimization, we note that the nodes in the last half of the array are all leaves, so we do not need to run `max_heapify_down` on them.

```
def build_max_heap(A, n):
    for i in range(n // 2, -1, -1): # O(n) loop backward over array
        max_heapify_down(A, n, i)   # O(log n - log i)) fix max heap
```

To see that this procedure takes $O(n)$ instead of $O(n \log n)$ time, we compute an upper bound explicitly using summation. In the derivation, we use Stirling's approximation: $n! = \Theta(\sqrt{n}(n/e)^n)$. Note that using this more efficient procedure to build a max heap will **not** affect the asymptotic efficiency of heap sort because each of n extractions will still take $O(\log n)$ time each. But it **is** a more efficient procedure to initially insert n items into an empty heap.

$$\begin{aligned} T(n) < \sum_{i=0}^{n}(\log n - \log i) = \log\left(\frac{n^n}{n!}\right) &= O\left(\log\left(\frac{n^n}{\sqrt{n}(n/e)^n}\right)\right) \\ &= O(\log(e^n/\sqrt{n})) = O(n \log e - \log\sqrt{n}) = O(n) \end{aligned}$$

We've made a CoffeeScript heap visualizer which you can find them here:
`https://codepen.io/mit6006/pen/KxOpep`

Exercises

1. Draw the complete binary tree associated with the sub-array array `A[:8]`. Turn it into a max heap via linear time bottom-up heap-ification. Run `insert` twice, and then `delete_max` twice.

```
A = [7, 3, 5, 6, 2, 0, 3, 1, 9, 4]
```

2. How would you find the **minimum** element contained in a **max** heap?

   Solution: A max heap has no guarantees on the location of its minimum element, except that it may not have any children. Therefore, one must search over all $n/2$ leaves of the binary tree which takes $\Omega(n)$ time.

3. How long would it take to convert a **max** heap to a **min** heap?

   Solution: Run a modified `build_max_heap` on the original heap, enforcing a **Min**-Heap Property instead of a Max-Heap Property. This takes linear time. The fact that the original heap was a max heap does not improve the running time.