Recitation 9

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. Recall that a priority queue maintains an ordering of priorities such that an element with higher priority is dequeued before an element with lower priority.

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, maintains an internal array A of items, and trivially implements `insert(x)` and `delete_max()` (the latter being incorrect on its own, but useful for subclasses).

```
class PriorityQueue:
    def __init__(self):
        self.A = []

    def insert(self, x):
        self.A.append(x)

    def delete_max(self):
        if len(self.A) < 1:
            raise IndexError('pop from empty priority queue')
        return self.A.pop()                  # NOT correct on its own!

    @classmethod
    def sort(Queue, A):
        pq = Queue()                         # make empty priority queue
        for x in A:                          # n x T_insert
            pq.insert(x)
        out = [pq.delete_max() for _ in A]   # n x T_delete_max
        out.reverse()
        return out
```

Shared across all 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 output array with successive maxima in reverse order.

Array Heaps

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.

```
class PQ_Array(PriorityQueue):
    # PriorityQueue.insert already correct: appends to end of self.A
    def delete_max(self):           # O(n)
        n, A, m = len(self.A), 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]     # swap max with end of array
        return super().delete_max() # pop from end of array
```

```
class PQ_SortedArray(PriorityQueue):
    # PriorityQueue.delete_max already correct: pop from end of self.A
    def insert(self, *args):            # O(n)
        super().insert(*args)           # append to end of array
        i, A = len(self.A) - 1, self.A  # restore array ordering
        while 0 < i and A[i + 1].key < A[i].key:
            A[i + 1], A[i] = A[i], A[i + 1]
            i -= 1
```

We use `*args` to allow `insert` to take one argument (as makes sense now) or zero arguments; we will need the latter functionality when making the priority queues in-place.

Binary Heaps

The next implementation is based on a binary heap, which 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` below.

```
class PQ_Heap(PriorityQueue):
    def insert(self, *args):         # O(log n)
        super().insert(*args)        # append to end of array
        n, A = self.n, self.A
        max_heapify_up(A, n, n - 1)

    def delete_max(self):            # O(log n)
        n, A = self.n, self.A
        A[0], A[n] = A[n], A[0]
        max_heapify_down(A, n, 0)
        return super().delete_max() # pop from end of array
```

Before we define `max_heapify` operations, we need functions to compute 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. Note that the methods below assume the array is 0-indexed whereas the methods presented in class assume 1-indexing.

```
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 = len(A)
    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: $n! = \Theta(\sqrt{n}(n/e)^n)$.

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

Note that using this linear-time procedure to build a max heap does not affect the **asymptotic** efficiency of heap sort, because each of n `delete_max` still takes $O(\log n)$ time each. But it is **practically** more efficient procedure to initially insert n items into an empty heap.

In-Place Heaps

To make heap sort **in place**[1] (as well as restoring the in-place property of selection sort and insertion sort), we can modify the base class `PriorityQueue` to take an entire array `A` of elements, and maintain the queue itself in the prefix of the first `n` elements of `A` (where `n <= len(A)`). We can think of the first `n` elements as what is kept inside the heap, and the last `len(A) - n` elements as extra elements that we do not wish to put in the queue for now.

When we insert now, the total length of the array `A` doesn't change, only the size of the heap we maintain using the first `n` elements of array `A`. So, the `insert` function is no longer given a value to insert; instead, it inserts the item already stored in `A[n]`, and incorporates it into the now-larger queue. To maintain the heap, we insert `A[n]` as usual and call `max_heapify_up` to maintain the ordering of the heap.

Similarly, `delete_max` does not return a value; but rather dequeues the `n`-th element out of the heap and into the auxiliary space. To delete, we swap `A[n-1]` with `A[0]` and perform `max_heapify_down` to maintain the ordering of the heap. Then, we decrement `n` by 1.

In priority queue sort, we will see that all `n insert` operations come before all `n delete_max` operations, as in priority queue sort.

```
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 += 1
        max_heapify_up(self.A, self.n, self.n-1)

    def delete_max(self):       # remove element A[n - 1] from the queue
        if self.n < 1:
            raise IndexError('pop from empty priority queue')
        self.A[0], self.A[n] = self.A[n], self.A[0]
        max_heapify_down(self.A, self.n, 0)
        self.n -= 1

    @classmethod
    def sort(Queue, A):
        pq = Queue(A)           # make empty priority queue
        for i in range(len(A)): # n x T_insert
            pq.insert()
        for i in range(len(A)): # n x T_delete_max
            pq.delete_max()
        return pq.A
```

[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.

This new base class works for sorting via any of the subclasses: `PQ_Array`, `PQ_SortedArray`, `PQ_Heap`. The first two sorting algorithms are even closer to the original selection sort and insertion sort, and the final algorithm is what is normally referred to as **heap sort**.

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

Exercises

1. Draw the complete binary tree associated with the sub-array array `A[0:7]` (first 7 elements). 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.

4. **Proximate Sorting**: An array of **distinct** integers is ***k-proximate*** if every integer of the array is at most k places away from its place in the array after being sorted, i.e., if the ith integer of the unsorted input array is the jth largest integer contained in the array, then $|i - j| \leq k$. In this problem, we will show how to sort a k-proximate array faster than $\Theta(n \log n)$.

   (a) Prove that insertion sort (as presented in this class, without any changes) will sort a k-proximate array in $O(nk)$ time. (Hint: think about the pizza problem from Problem Set 3 that sorts based on the time and duration of the pizza cooking)

   Solution: To prove $O(nk)$, we show that each of the n insertion sort rounds swap an item left by at most $O(k)$. In the original ordering, entries that are $\geq 2k$ slots apart must already be ordered correctly: indeed, if $A[s] > A[t]$ but $t - s \geq 2k$, there is no way to reverse the order of these two items while moving each at most k slots. This means that for each entry $A[i]$ in the original order, fewer than $2k$ of the items $A[0], \ldots, A[i-1]$ are greater than $A[i]$. Thus, on round i of insertion sort when $A[i]$ is swapped into place, fewer than $2k$ swaps are required, so round i requires $O(k)$ time.

   It's possible to prove a stronger bound: that $a_i = A[i]$ is swapped at most k times in round i (instead of $2k$). This is a bit subtle: the final sorted index of a_i is at most k slots away from i by the k-proximate assumption, but a_i might not move to its final position immediately, but may move **past** its final sorted position and then be bumped to the right in future rounds. Suppose for contradiction a loop swaps the pth largest item $A[i]$ to the left by more than k to position $p' < i - k$, past at least k items larger than $A[i]$. Since A is k-proximate, $i - p \leq k$, i.e. $i - k \leq p$, so $p' < p$. Thus at least one item less than $A[i]$ must exist to the right of $A[i]$. Let $A[j]$ be the smallest such item, the qth largest item in sorted order. $A[j]$ is smaller than $k+1$ items to the left of $A[j]$, and no item to the right of $A[j]$ is smaller than $A[j]$, so $q \leq j - (k+1)$, i.e. $j - q \geq k + 1$. But A is k-proximate, so $j - q \leq k$, a contradiction.

   (b) $\Theta(nk)$ is asymptotically faster than $\Theta(n^2)$ when $k = o(n)$, but is not asymptotically faster than $\Theta(n \log n)$ when $k = \omega(\log n)$. Describe an algorithm to sort a k-proximate array in $O(n \log k)$ time, which can be faster (but no slower) than $\Theta(n \log n)$.

   Solution: We perform a variant of heap sort, where the heap only stores $k+1$ items at a time. Build a min-heap H out of $A[0], \ldots, A[k-1]$. Then, repeatedly, insert the next item from A into H, and then store `H.delete_min()` as the next entry in sorted order. So we first call `H.insert(A[k])` followed by `B[0] = H.delete_min()`; the next iteration calls `H.insert(A[k+1])` and `B[1] = H.delete_min()`; and so on. (When there are no more entries to insert into H, do only the `delete_min` step.) B is the sorted answer. This algorithm works because the ith smallest entry in array A must be one of $A[0], A[1], \ldots, A[i+k]$ by the k-proximate assumption, and by the time we're about to write $B[i]$, all of these entries have already been inserted into H (and some also deleted). Assuming entries $B[0], \ldots, B[i-1]$ are correct (by induction), this means the ith smallest value is still in H while all smaller values have already been removed, so this ith smallest value is in fact `H.delete_min()`, and $B[i]$ gets filled

correctly. Each heap operation takes time $O(\log k)$ because there are at most $k + 1$ items in the heap, so the n insertions and n deletions take $O(n \log k)$ total.