Lecture 5: Priority Queues, Binary Heaps

Priority Queues

- Keep track of many items, quickly access/remove the most important
  - E.g., router with limited bandwidth, must prioritize certain kinds of messages
  - Fun example: `https://beta.observablehq.com/@mbostock/quadtree-art` by Mike Bostock
- Set API, not Sequence API
- keys, not indices
- Optimized for a particular subset of Set operations:

`len()`	n
`insert(v)`	Add new element
`find_max()`, `remove_max()`	Find or remove most important (biggest)
`find_min()`, `remove_min()`	Find or remove most important (smallest)

- (Usually optimized for max or min, not both)
- Really a **multi**-set interface: allow duplicate keys without overwriting
- `find_max` means find **any** max, don't care which

Priority Queues are great for sorting

- Insert in any order, extract in sorted order
- All the hard work happens inside the data structure

```
def max_pq_sort(A):
  n = len(A)
  Q = <empty priority queue>
  for v in A:
    insert(Q, v)
  for i in range(n):
    A[n-1-i] = remove_max(Q)

def min_pq_sort(A):
  # ...
  for i in range(n):
    A[i] = remove_min(Q)
```

Easy Implementation of Priority Queue: Array

- Store elements in a dynamic array Q, in any order
- Insertion is quick, but `remove_max` is slow

```
def insert(Q, v):          # Yay, O(1)
  Q.append(v)

def remove_max(Q):         # Aww, O(n)
  best, n = 0, len(Q)
  for i in range(1, n):
    if Q[i] > Q[best]:
      best = i
  <swap Q[best] and Q[n-1]>
  return Q.pop()
```

- `max_pq_sort` is selection sort! (plus some copying)

Easy Implementation of Priority Queue: Sorted Array

- Store elements in a (dynamic) array Q, in sorted order

```
def remove_max(Q):         # Yay, O(1)
  return Q.pop()

def insert(Q, v):          # Aww, O(n)
  Q.append(v)
  n = len(Q)
  vi = n-1
  while vi > 0 and Q[vi] > Q[vi - 1]:
    <swap Q[vi - 1] and Q[vi]>
```

- `max_pq_sort` is insertion sort! (plus some copying)
- Can we find a compromise between these two extreme priority queues?
- Put your hand down, it was rhetorical

Detour: Arrays as dense binary trees

- Clever idea: interpret an array as a dense binary tree
- Fill densely in reading order: root to leaves, left to right
  - As many full rows as possible
  - Last row left-aligned

  - Call such a tree "packed"

- Why is this recipe useful?
  - `parent(i) = floor(i-1)`
  - `left(i) = 2*i + 1`
  - `right(i) = 2*i + 2`
  - `depth(i) = floor(log(i+1))` (roughly $\log i$, but beware $\log 0$)

- This is a bijection between arrays and packed binary trees

- Tree structure is implicit:
  - No pointers, no extra space overhead
  - Only the array lives in memory
  - Just another way of interpreting an array of data

- This only works because we know the shape beforehand. If you want different tree shapes, you'll have to store the pointers explicitly. (See next lecture!)

Binary Heaps

- Clever idea: keep larger elements higher in tree, but only locally
  - Node max-heap property at i: $A[i] \geq A[\text{left}(i)], A[\text{right}(i)]$
  - Tree max-heap property at i: $A[i] \geq$ every node in S(i)
    * $S(i)$ is the subtree rooted at i

- A **max-heap** is an array where every node satisfies the node max-heap property

- Claim: this implies every node satisfies the tree max-heap property

- Intuitive, but let's review induction:
  - IH: If j is in $S(i)$ with $\text{depth}(j) - \text{depth}(i) = d$, claim $A[i] \geq A[j]$.
  - Proof by induction on d.
  - Base case: $d = 0$ implies $i = j$ implies $A[i] \geq A[i]$, yay
  - Inductive step: $d \geq 1$
    * $\text{depth}(\text{parent}(j)) - \text{depth}(i) < d$, so $A[i] \geq A[\text{parent}(j)]$ by IH
    * $A[\text{parent}(j)] \geq A[j]$ by node max-heap property at $\text{parent}(j)$
    * yay

- In particular, if max heap property everywhere, max elt is at root

```
def find_max(Q):    # O(1)
  return Q[0]
```

Maintaining heap property

- Given array satisfying max-heap property (everywhere), how to insert an element?

```
def insert(Q, v):
  Q.append(v)
  max_heapify_up(Q, len(Q) - 1)

def max_heapify_up(Q, i):
  if i > 0 and Q[i] > Q[parent(i)]:
    <swap Q[i] and Q[parent(i)]>
    max_heapify_up(Q, parent(i))
```

- Correctness:
  - `max_heapify_up` assumes all nodes are $\geq$ their descendants, except that $Q[i]$ might be greater than some of its ancestors
  - If swap is necessary, same assumption is true with i replaced by $\text{parent}(i)$
- How to remove max?

```
def remove_max(Q):
  <swap Q[0] and Q[len(Q) - 1]>
  result = Q.pop()
  max_heapify_down(Q, 0)
  return result

def max_heapify_down(Q, i):
  best = <index of largest node among i, left(i), right(i)>
  if best != i:
    <swap Q[i] and Q[best]>
    max_heapify_down(Q, best)
```

- Correctness:
  - `max_heapify_down` assumes all nodes are $\geq$ their descendants, except that $Q[i]$ might be less than some of its descendants
  - if swap is necessary, same property holds with i replaced by “best”

Optimization: in-place operations

- Max-heap lives in larger array A, remembers how many elts belong to the heap
  - n is now different from len(A)
  - initially full array, empty heap
- insert just gobbles next element in array
- `remove_max` moves max elt to end then abandons it with `n -= 1`
- `pq_sort` with Array is exactly selection sort (without the copying)
- `pq_sort` with Sorted Array is exactly insertion
- heapsort is fully in-place

Optimization: build heap in linear time

```
def build_max_heap(A):
  for i in range(len(A) // 2, -1, -1):
    max_heapify_down(A, i)
```

- will be analyzed in recitation
- Note: In the wild, the term ”heapsort” usually includes the in-place optimization and this linear-time `build_heap` optimization.