1 Priority Queue ADT

Priority Queues are a special type of ADT that stores maps of key-value items where we can remove the smallest or the largest item in a min or max PQ respectively.

- insert(k,v): Inerts itme with key k and value v.
- remove_ming()/remove_max(): Removes & returns the item with smallest/largest key.
- min()/max(): Returns item with the smallest/largest key.
- size():Returns how many items are stored.
- is_empty(): Tests of queue is empty.

1.1 Sequence based Priority Queue

Unsorted list implementation

- insert runs in O(1) time since we can insert the item at the beginning or end of the sequence.
- remove_min and min and their equivalents run in O(n) time since we have to traverse the entire list to find the smallest key.

Sorted list implementation

- insert runs in O(n) time since we have to find the correct place in the order to insert the item.
- remove_min and min and their equivalents run in O(1) time since the smallest key is at the beginning.

2 Priority Queue Sorting

We can use a priority queue to sort a list of keys. To do so, first iteratively insert keys into an empty PQ. Then iteratively remove_min to get the keys in sorted order. Either sequeunce based implementation takes O(n^2).

```
def priority_queue_sorting(A):
    pq ← new priority queue
    n ← size(A)
    for i in [0:n] do do
        pq.insert(A[i])
            pq.insert(A[i])
        for i in [0:n] do
            A[i] = pq.remove_min()
```

2.1 Selection Sort

Selection sort is a variant of pq sort that uses unsorted sequence implementation. The algorithm works by first inserting elements with n insert operations which takes O(n) time. It them removes elements with n remove_min operations which takes O(n^2).

```
def selection_sort(A):
    n ← size(A)
    for i in [0:n] do                          # find s >= i minimizing A[s]
        s ← i
        for j in [i:n] do
            if A[j] < A[s] then
                s ← j
        A[i], A[s] ← A[s], A[i]                # swap A[i] and A[s]
```

2.2 Insertion Sort

Variant of pq-sort using sorted sequence implementation that first inserts eleemnts with n insert operations which takes O(n^2) time before removing elements with n remove_min operations which takes O(n) time.

```
def insertion_sort(A):
    n ← size(A)
    for i in [1:n] do
        x ← A[i]                               # move forward entries > x
        j ← i
        while j > 0 and x < A[j-1] do
            A[j] ← A[j-1]
            j ← j -1                           # if x>0, x >= A[j-1]
        A[j] ← x                               # if j<i, x < A[j+1]
```

3 Heap data structure (min-heap)

A heap is a binary tree stroing (key. value) items at its nodes. It satisifies the properties:

- Heap-order: for every node m = root, key(m) >= key(parent(m))
- Complete Binary Tree: let h be the height. Eevery level i < h is full (i.e., there are 2i nodes). Remaining nodes take leftmost positions of level h.

RTP: The root always holds the smallest key in the heap

- Suppose the minimum key is at some internal node x.
- Because of the heap property, as we move up the tree, the keys can only get smaller (assuming repeats, otherwise contradiction)
- If x is not the root, then its parent must hold a smaller key.
- Keep going until we reach the root.

RTP: A heap storing n keys has height log n

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth i = 0, . . . , h - 1 and at least one key at depth h, we have n >= 1 + 2 + 4 + . . . + 2h-1 + 1
- Thus, n >= 2h , applying log2 on both sides, log2 n >= h

3.1 Insertion into a Heap

Firstly, create a new node with the given key. Fine location for new node. Restore the heap-order property.

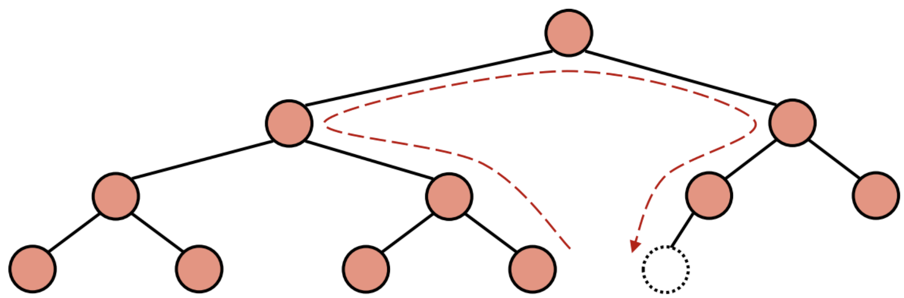
3.2 Upheap

Restore heap-order property by swapping keys along the upward path.

```
def up_heap(z):                                      # O(log(n))
    while z != root and key(parent(z)) > key(z) do
        swap key of z and parent(z)
        x ← parent(z)
```

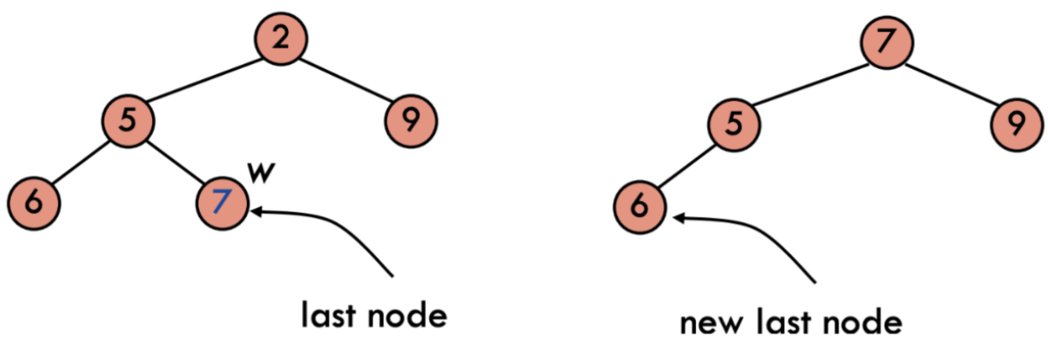
3.3 Finding the position for insertion: O(log n)

- Start from the last node
- Go up until a left child or the root is reached
- If we reach the root then need to open a new level
- Otherwise, go to the sibling (right child of parent)
- Go down left until a leaf is reached



3.4 Removal from a heap

Replace the root key with the key of the last node w. Delete w. Restore the heap-order property



3.5 Downheap

Restore heap-order property by swapping keys along downward path from the root.

```
def down_heap(z):                                          # O(log(n))
    while z has child with key(child) < key(z) do
        x ← child of z with the smallest key
        x ← parent(z)
        z ← x                                              # swap keys of z and x
```

4 Heap-Sort

Consider a priority queue with n items implemented with a heap: the space used is O(n) methods insert and remove_min take O(log n). Heap-sort is the version of priority-queue sorting that implements the priority queue with a heap. It runs in O(n log n) time.

4.1 Heap-in-array implementation

We can represent a heap with n keys by means of an array of length n.

- The root is at 0.
- The last node is at n-1.
- The left child of i is at index 2i+1.
- The right child of i is at index 2i+2.
- The parent of i is at index floor((i-1)/2).

4.2 Summary of Heap-Sort

Heap-sort can be arranged to work in place using part of the array for the output and part for the priority queue A heap on n keys can be constructed in O(n) time. But the n remove_min still take O(n log n) time. Some other operations include:

- remove(e): Remove item e from the priority queue.
- replace_key(e, k): Update key of item e with k.
- replace_value(e, v): Update value of item e with v.

Method	Unsorted List	Sorted List	Heap
size, isEmpty	O(1)	O(1)	O(1)
insert	O(1)	O(n)	O(logn)
min	O(n)	O(1)	O(1)
removeMin	O(n)	O(1)	O(logn)
remove	O(1)	O(1)	O(logn)
replaceKey	O(1)	O(n)	O(logn)
replaceValue	O(1)	O(1)	O(1)