Priority Queue and Binary Heap

Priority Queue

Priority Queue is the same as a normal queue, except that it is always sorted by some measurement of priority, with the item of highest priority being at the front. It still follows the FIFO logic and is stable in that sense, but prioritized items will be pushed forward.

The main operations are:

- Enqueue
- Dequeue
  - Dequeue the item of highest priority in the queue
  - If there are more than one item with the highest priority, return the one that is inserted first (FIFO)

Priority Queue Implementation #1: Circular Array (Quick Find)

We can maintain a priority queue by always inserting in the correct place. The array will thus always be in the correct order. A front and back index is maintained.

- Enqueue(item): O(n) process as we search through the array to find a suitable insertion point. The process here is similar to that of one iteration of insertion sort.
- Dequeue(): O(1) dequeuing from the front of the queue. We do not need to shift all elements forward, and just need to advance the front index. The same circular array logic is maintained as in the case for queue above.

Priority Queue Implementation #2: Circular Array (Quick Insert)

We can achieve the same effect as the above by simply scanning through the queue to find the item of highest priority to dequeue at all times.

- Enqueue(item): O(1), as we simply insert to the back of the queue
- Dequeue(): O(n), as we scan through the entire array and return the first item of highest priority. The gap then needs to be closed, also O(n).

Method	**Enqueue**	**Dequeue**
Quick Find	O(N)	O(1)
Quick Insert	O(1)	O(N)

However, we can do better.

Priority Queue Implementation #3: Binary Heap

Refer to the following section on Binary Heap.

Binary Heap
A Binary Heap is a Binary Tree with the following properties:

- It's a complete tree
    - All levels are completely filled except for the last level
        - Height = O(log n)
    - The last level has all leaves flushed to the left
- It upholds the binary heap property
    - Max Heap
        - A[parent(i)] ≥ A[i]
        - A[i] ≥ A[left(i)] && A[i] ≥ A[right(i)]
    - Min Heap
        - A[parent(i)] ≤ A[i]
        - A[i] ≥ A[left(i)] && A[i] ≤ A[right(i)]

The main operations are:

- Insert
- Extract Max/Min
- Create Heap
- Heap Sort
- Helper Functions
    - Left Child
    - Right Child
    - Parent
    - Shift Up
    - Shift Down

Terminology

- Each node is called a vertex
- The topmost vertex is called the root
- They have a left child and right child
- They have a parent, except for the root
- Vertices without a left child and right child are called leaves
- All non-leaves and non-root are called internal vertices

Binary Heap Implementation: 1-Based Array
To store the values of a binary heap, we can actually use a single array, starting from index 1. The reason for starting at index 1 is so that we can easily calculate the index of the right child, left child and parent from the index of any vertex.

```
int parent(int i) { return i>>1; } // shortcut for i/2, round down
int left(int i) { return i<<1; } // shortcut for 2*i
int right(int i) { return (i<<1) + 1; } // shortcut for 2*i + 1
```

This is done using bit shift operators, which is a lot faster than actual flooring etc.

As for the rest of the operations, here are their steps and analyses:

- Insert(v): O(log n) – equivalent of enqueue

```
heapsize++; // O(1)
A[heapsize] = v; // O(1)
shiftUp(heapsize); // O(log n)
```

- ExtractMax(): O(log n) - equivalent of dequeue

```
maxV = A[1]; // O(1)
A[1] = A[heapsize]; // O(1)
heapsize--; // O(1)
shiftDown(1); // O(log n)
return maxV; // O(1)
```

- ShiftUp(index): O(log n)

```
while i>1 and A[parent(i)] < A[i] // O(log n)
      swap(A[i], A[parent(i)] // O(1)
      i = parent(i) // O(1)
```

- ShiftDown(index): O(log n)

```
while i<=heapsize
      maxV = A[i]
      maxID = i

      if left(i) <= heapsize and maxV < A[left(i)]
            maxV = A[left(i)]
            maxID = left(i)

      if right(i) <= heapsize and maxV < A[right(i)]
            maxV = A[right(i)]
            maxID = right(i)

      if maxID!=i
            swap(A[i], A[maxID])
            i = maxID
      else
            break
```

CreateHeap(Arr)

Naïve Version – O(N log N)

```
createHeapSlow(arr) // naïve version
      N = size(arr)
      A[0] = 0 // dummy entry
      for i = 1 to N // O(N)
            insert(arr[i-1]) // O(log N)
```

Stirling's approximation states that:
log 1 + log 2 + log 3 +…+ log N = log N! = N log N

Better Version – O(N)

```
createHeap(arr)
      heapsize = size(arr)
      A[0] = 0 // dummy entry
      for i = 1 to heapsize // copy the content O(N)
            A[i] = arr[i-1]
      for i = parent(heapsize) down to 1 // O(N/2)
            ShiftDown(i) // O(log N)
```

Mathematical Proof
To solve for the complexity, we first need to be aware of the following:

- Height of complete binary tree of size N = floor(log N) = h
- Cost to run shiftDown(index) = h
- Number of nodes at height h (bottommost layer is 0) = ceil(N/(2^(h+1)))

The cost to perform createHeap is thus:

$$\sum_{h=0}^{\lfloor \lg(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

The Big-O can be taken to the outside along with n, and h can be moved on top of the fraction. We can also take out ½ as it is a constant.

$$O(n \sum_{h=0}^{\lfloor \lg(n) \rfloor} \left\lceil \frac{h}{2^{h}} \right\rceil)$$

The summation can be further summarised and broadened to sum till infinity (as it is an upper bound), resulting in:

$$O(n \sum_{k=0}^{\infty} kx^k)$$

The right side can be evaluated to a case where x=1/2, which forms a geometric series. You can very simply sum it up:

$$\frac{x}{(1-x)^2} = 2$$

Thus the time complexity is O(2N) = O(N).

<u>HeapSort(Arr)</u>
The logic is very simple. Convert an array into a binary heap and pull out the numbers/items in value of priority.

```
HeapSort(array)
      CreateHeap(array) // O(N)
      N = size(array)
      for i from 1 to N // O(N)
            A[N-i+1] = ExtractMax() // ~O(log N)
      return A
```

Overall Time Complexity: O(N log N)

This sorting method is in-place. However, it is not cache friendly, as the index is always changing and the jumps can be huge due to the doubling of indices, despite the usage of an array for the heap.

Extra Proving from Tutorial
<u>Claim: The second largest element in a max heap with more than two elements (all elements are unique) is always one of the children of the root.</u>

Proof:
Yes it is true. This can be proven easily by proof of contradiction. Suppose the second largest element is not one of the children of the root. Then, it could be the root, but this cannot be since root is the largest element by definition, thus contradiction. Or it has a parent that is not the root. There will be a violation to max heap property (there is nothing between largest and 2nd largest element), which is a contradiction. So, the second largest element must always be one of the children of the root.

<u>Claim: Finding the maximum value in a Binary Min Heap will always take O(lg n) time.</u>

Proof:
False, as it will take O(n) time. As a binary min heap is not strictly sorted, but is only held to a certain heap property, there is a need to traverse through all of the nodes in a binary min heap to find all of the leaves and compare them against each other to identify the largest element.

Put succinctly, any leaf can be a max, and there are n/2 leaves.

<u>Claim: The min and max number of nodes in a complete binary tree with height h are 2^h - 1 and 2^(h+1) - 1 respectively.</u>

Proof:
False, as the minimum number of nodes in a complete binary tree with height h is 2^h. This is because the maximum number of "empty slots" for nodes to fill for a perfect binary tree is 2^(h+1)-1, and the minimum would naturally be 2^(h+1)-1-((2^h)-1).

<u>Claim: It is possible to write an O(lg n) algorithm to check a 1-based array represents a Min Heap.</u>

Proof:
False, as you need to traverse and check every edge to ensure that the array represents a Min Heap/you need to check every internal vertex + root against its children, and there are n-1 edges in a heap.