Priority Queues and Heaps

A queue is an abstract data type that supports addition and removal of items, with the following restrictions:
- new elements can only be added to the end of the queue
- only the item at the beginning of the queue can be accessed and/or removed

A queue can be implemented with an array:
- addition and removal take O(1) time
- size is limited
- need to keep track of where the beginning and end of the queue are in the array
- the queue may eventually "wrap around" the end of the array - not a problem but just a special case that we need to deal with

An alternative implementation is with a linked list:
- size is not limited
- addition and removal take O(1) time
- no special cases to worry about
- linked lists take more space than arrays, and are a bit slower

A priority queue is a queue in which each item has a priority attached. We will assume that 1 is the lowest priority. Priority queues have the following operations:
- add new elements
- find (and/or remove) the item with the highest priority
- in practice, we want items with equal priority be sequenced in such a way that items added earlier reach the head of the queue before items added later (for example, if item A and item B both have the same priority and A is added to the priority queue before B is added, then item A should come to the head of the priority queue before B does)

Note that searching the set is not a required operation. Some implementations of priority queues do support search operations – we will discuss this later.

Priority queues have many applications including hospital emergency wards, airport plane landing sequencing, and operating system task scheduling.

In every application, the priority of an item is just one of its attributes and a full implementation will have to store all of the data for each item. However in our discussion we
will focus only on the priority values of the items. We will assume that the rest of the data “goes along for the ride”. In our diagrams we will only show the priority values of the items unless we need to distinguish between items that happen to have the same priority.

**Implementation of a Priority Queue:**

We can certainly implement a priority queue with an array or linked list:
- a new item with priority \( p \) is inserted after all items with priority \( p \geq \) and before all items with priority \( < p \)
- adding an item is in \( O(n) \) where \( n \) is the number of items in the queue
- accessing (and/or removing) the item with maximum priority is in \( O(1) \)

Example of the problem when using an array:

```
7 4 4 3 3 3 3 2 1 1 1
```

To insert a new item with priority 7, almost all the values need to be moved one space to the right to create an empty space for the new item – this is \( O(n) \)

Example of the problem when using a linked list:

```
7 -> 4 -> 4 -> 3 -> 2 -> 2 -> 1
```

To insert a new item with priority 2, we need to walk through most of the list to find the insertion point – this is \( O(n) \)

Note that if the set of possible priorities is limited to the set \{1,2, \ldots \} \( k \) for some fixed integer \( k \), then we can implement the priority queue with a set of separate queues, one for each priority class. Adding a new item takes \( O(1) \) time, and accessing (and/or removing) the item with maximum priority takes \( O(k) \) time, and since \( k \) is a fixed integer, this is \( O(1) \).

That’s a nice result (\( O(1) \) complexity is always good) but not very interesting.
However if the set of possible priorities is not limited, we need to be smart to improve on the O(n) time to add new items. We will use a structure called a **max-heap**.

**Max-Heaps**

A max-heap is a binary tree such that

- the value stored at each vertex is $\geq$ the values stored at its children. Note that this results in the largest value being at the root of the tree
  - all levels are full, except possibly the last one
  - if the bottom level is not full, all of its vacancies are at the right side

So we have one **organizational** rule, and two **structural** rules. Because of the structural rules, the “shape” of a max-heap containing a set of k values is completely fixed. For example if the set contains 6 values, a max-heap containing these values must look like this.

![Max-heap diagram]
However, it is important to see that the organizational rule can be satisfied by different arrangements of the values. For example if the set of values is \{1,3,3,7,9,10\}, the heap could be arranged as

```
10
 /   \
9     7
 / \   / \
1   3  3 1
```

or as

```
10
 /   \
3     9
 / \   /
3 1 7
```

or as any of several other arrangements. Note that there is no “left child \( \leq \) right child” rule. However there are two consistent features of all legal max-heap arrangements:

- the largest value is at the root of the tree

- the second largest value (which may be a duplicate of the largest value) is in one of the root’s children.
Accessing the maximum value is $O(1)$, since it is at the root of the tree. But what is the complexity of removing that value?

**Removing the largest value from a max-heap**

The problem of course is that we can’t simply delete the first vertex (as we could if the priority queue were stored in a linked list) – we need to choose a new root. One idea that occurs to many people is to “promote” the larger of the root’s children by moving that value up into the root vertex ... then filling the newly empty vertex by promoting the larger of its children, and so on down the tree until we end up with a leaf with nothing in it. Then delete that leaf.

The problem with this is that it spoils the structural rules of the max-heap. Well, so what? Rules are made to be broken, right? But the danger here is that by allowing this, our tree may end up looking very sparse ... in fact it may end up looking like a linked list. And in that (extreme) case, subsequent “remove largest” operations will be in $O(n)$ because we would go all the way down the tree, promoting the value at each level. This is bad because our goal is to avoid $O(n)$ operations.

So here’s a better approach: Consider the first max-heap example shown above. We know that after we remove the largest value, the set will have exactly 5 elements, and therefore we can see exactly what the max-heap will look like after the deletion:

![Max-Heap Diagram](image)

So when we take out the largest value (10) we have an empty vertex at the root. But we also have to get rid of the last vertex on the bottom level, which contains 3. Hmm, a vertex that needs a new value, and a value that needs a new vertex ... what shall we do?
The answer of course is that we put the 3 in the empty vertex at the root, giving this

But now the organizational rule is violated. We need to fix that without changing the structure of the max-heap. Fortunately this is really easy: we compare the value we just moved up and the values of its children. If it is smaller than either of them, we “swap” it with the larger of its children (in this example, we swap the 3 in the root with the 9). Now we compare the moving 3 with the values of its new children, and if necessary we swap it with the larger of its children ... and we continue pushing it down the tree until it reaches a valid location. (In this example the 3 does not need to move down any more because it is $\geq 1$ and $\geq 3$.) It is important to understand that this process is guaranteed to produce a properly arranged max-heap: whenever we swap a value “up”, it is in a valid location because it is $\geq$ all the values below it.

So what is the complexity of this process? Moving the value from the bottom to the top takes constant time, then each time we swap it with one of its children, that is also a constant time operation ... and because we have kept the tree as compact as possible, we know there are $\leq \log n$ levels in the tree! So this entire process is in $O(\log n)$ Win!

Note: we can improve the practical efficiency of this by not actually re-inserting the value from the bottom into any vertex until we actually find its new home. This means that all the points where we used the term “swap” are really just “promote” operations – it saves a bit of time.
Here is pseudo-code for the entire “remove largest” operation:

```python
def remove_max():
    max_value = root.value
    mover = value from the right-most filled position in the bottom level of
    the tree
    delete the vertex that contained mover
    temp_pos = root
    while mover < either of the children of temp_pos:
        new_pos = temp_pos.left or temp_pos.right, whichever has the larger
        value
        temp_pos.value = new_pos.value
        temp_pos = new_pos
    temp_pos.value = mover
    return max_value
```

For implementation, we have to handle a special case: temp_pos may have only a left child
(note that there is no way a vertex can have a right child but no left child) so the logic is
something like this:

```python
if temp_pos.left != None:
    max_child = temp_pos.left
    if temp_pos.right != None and
        temp_pos.right.value > temp_pos.left.value:
        max_child = temp_pos.right
    # now max_child points to the child with the largest value
    if mover < max_child.value:
        ...
```

There is also a significant step in the algorithm left unspecified ... how do we find the right-
most occupied vertex on the bottom level of the tree? We will come back to that.