Chapter 13

Priority Queues

Imagine that you are writing a system to manage tickets—service requests—for an IT department. A natural choice of data structure to hold these tickets is a *queue* which provides *first-in-last-out* behavior. Recall that queues provide two important operations:

- `enqueue(T v)`: adds the value `v` onto the end of the queue.
- `dequeue()`: removes and returns the oldest value (the first value in line) from the queue.

This interface is suitable for dealing with the line-like-behavior that our system needs to manage.

However, a queue is insufficient if the tickets also carry along *priorities* with them. Some tickets are higher priority than others—an electrical fire should take precedence over an accountant's email being down—and our system needs to be able handle this. In particular, we ought to require that dequeue return the *highest priority item* in the queue, regardless of how long it has been in the structure. If there are multiple items with the same highest priority, then we can service any of them first.

13.1 The Priority Queue ADT

A priority queue is an abstract data type that represents a queue-like structure that respects priorities. A priority is defined to be an *ordering* on the elements contained in the queue. For example, the tickets may have a numeric value associated with them denoting their priority, and we could order the tickets based off this priority.

A priority queue provides the following operations:

- `void add(T v)`: adds the value `v` into the queue, respecting priority order (*i.e.*, enqueue).
- `T poll()`: removes and returns the value with the highest priority from the queue (*i.e.*, dequeue).
- `T peek()`: returns, but does not remove, the value with the highest priority in the queue.

If multiple values have the same priority, then the order in which they are dequeued is unspecified. For example, consider a priority queue for tickets as described above where a ticket is represented by an integer that is its priority. For simplicity's sake, we'll represent a ticket by an integer representing its priority with higher priority tickets needing to be serviced first. For example, if we start with an empty priority queue and add in tickets 10, 5, and 2, the queue has the following shape:

$$\text{(front)} \quad [10, 5, 2].$$

Peeking at the top element of the queue will result in 10 because it is the highest priority ticket. After adding tickets 15 and 7, the queue changes to:

$$[15, 10, 7, 5, 2].$$

Polling the queue at this point dequeues 15 from the front of the queue, leaving:

$$[10, 7, 5, 2].$$

Finally, if we add another ticket with priority 5, the queue becomes:

$$[10, 7, 5_a, 5_b, 2].$$

The newest ticket of priority 5, denoted 5_b appears after the older ticket, 5_a, in the queue.

Note that the higher priority ticket goes to the front of the queue. For simplicity's sake, we show the queue as an ordered list although the data structure we use to represent this priority queue may not maintain this ordering. The only thing it needs to do is ensure that the highest priority element is easily removable when the queue is polled.

13.2 Heaps

Our example suggests a simple implementation: an ordered array list where elements are ordered by their priority.

- `void add(T v)`: because the list is ordered, we can use binary insertion to insert `v` into the list in amortized $\mathcal{O}(\log n)$ time.
- `T peek()`: because the list is ordered, the head of the list is the highest priority. We can access this element in $\mathcal{O}(1)$ time.
- `T poll()`: poll requires that we remove the head of the list which takes $\mathcal{O}(n)$.

Can we do better than this? Recall that a *binary search tree* allows us to maintain an ordered list but with $\mathcal{O}(\log n)$ time complexity for `add` and `remove`. This sounds good on paper, but the problem is that the $\mathcal{O}(\log n)$ time complexity is dependent on the BST being *balanced*. If it is balanced, then we obtain the desired $\mathcal{O}(\log n)$ complexity. However, if the tree is degenerate, *i.e.*, a linked list, then we have $\mathcal{O}(n)$ time complexity instead.

There are general schemes for maintaining a balanced tree that allow us to get the $\mathcal{O}(\log n)$ complexity we want but at the cost of higher constant factors in the runtime. However, can we obtain similar performance in this restricted domain of supporting a priority queue without the complexities of a general balancing strategy?

It turns out that we can do this by relaxing the invariant on our binary search tree. Recall that the *binary search tree invariant* says that for any node in a binary search tree:

(a) The left branch contains elements that are less than the element at this node.

(b) The right branch contains elements that are greater than the element at this node.

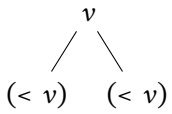
Because of this invariant, we are forced to unbalance the tree in certain situations, *e.g.*, inserting elements in ascending order. If we relax the invariant, we can hit a sweet spot between enforcing the ordering that we to support a priority queue while allowing us to easily balance the tree.

13.3 The Heap Invariants

The data structure we'll use to efficiently implement a priority queue is a called a *heap* (which has no relation to the heap in memory). A (binary) heap is a tree which maintains two invariants:

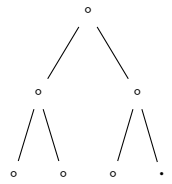
- The *semantic binary heap invariant* says that for any node in the tree, the sub-trees of that node only contain elements *less than* the element at this node.
- The *structural binary heap invariant* says that the heap is always *complete*. That is, all the levels but the last of the heap are completely filled in, and the last level is filled from left-to-right.

The semantic invariant is represented graphically as follows:



This may seem like a useless property, but by doing this, we implicitly require that all elements greater than v appear above it in the tree. By applying this reasoning recursively at each level of the tree, we know that the *maximum priority element* must be placed at the root of the tree.

The structural invariant is represented graphically as follows:



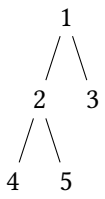
Note that with such a tree, the length of any path from the root to a leaf is upper-bounded by $\log n$ which is critical in ensuring that the runtime of our operations will be $\mathcal{O}(\log n)$.

A final note on the semantic invariant of our heaps: by choosing to push smaller elements further down the tree, the maximum element sits at the root. We call such a heap a *max heap*. In contrast, we could instead have our semantic invariant require that the children of a node only contain elements *greater than* the value at the node. By doing this, the minimum element sits at the root of tree. Such a heap is a *min heap*. For the purposes of simplifying our discussion, we will consider a max heap in our discussions below. However, keep in mind that a min heap is obtainable by simply flipping the ordering in our invariant.

13.4 Array-based Trees

Because our heaps are complete trees, we are able to use an *array* to represent the tree rather than a linked structure (compare array lists versus linked lists). The array will contain the contents of the nodes of our tree, and rather than containing references to its children, we will use explicit formulae to find the indices of children and parent nodes given the index of a particular node in the tree.

To arrive at these formulae, we note that a natural way to lay out the elements of a complete tree in an array is to proceed in a top-down, left-to-right fashion. For example, if we have the following tree:



We could represent it with the following array:

$$[1, 2, 3, 4, 5, \ldots]$$

keeping in mind that like an array list, only part of the overall array is in use at any given time. Because the tree is complete, this layout strategy ensures we fill the array from left to right. This fact is why we did not previously consider using an array to represent a tree. Most trees will not be complete like a heap and so, there will be many indices of the array that are unused.

If we look at each node in the array:

- 1 is at index 0 with children 2 (index 1) and 3 (index 2)
- 2 is at index 1 with children 4 (index 3) and 5 (index 4)
- 3 is at index 2 with no children (indices 5 and 6).

From this, we can derive the following formulae:

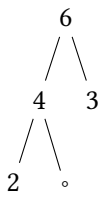
- The *left child* of the node at index i is $i \times 2 + 1$.
- The *right child* of the node at index i is $i \times 2 + 2 = (i + 1) \times 2$.
- The *parent* of the node at index i is $\lfloor \frac{i-1}{2} \rfloor$ for nodes that are not the root of the overall tree.

13.5 Heap Operations

Now we discuss implementing each of our heap operations in terms of our array-based tree data structure:

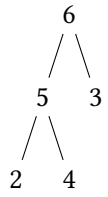
Peek Noting that the root of the tree is the first index of the array, we simply return that element. This takes $\mathcal{O}(1)$ time to do so.

Add Adding an element requires a bit more thought than `peek`. We must add an element in such a way that the maximum element is the root of the tree. When we add a new smallest element into the heap, we can simply insert the element in such a way as to maintain a complete tree, *i.e.*, top-down and left-to-right. For example, if we insert 1 into the following heap:



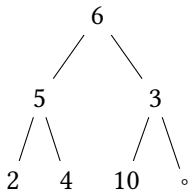
Then we can make 1 the right-child of 4 which preserves both the semantic and structural heap invariants.

However, what happens if we need to add an element that won't be a leaf in the tree? For example, if we insert 5 into the tree, it must be in either the left or right subtree of 6, but it must also be the parent of 4 and 3. Because our semantic invariant is relaxed, we have a choice about which branch of the tree we should modify. To preserve the structural invariant, it makes sense to make 5 the direct left child of 6 and make 4 a right-child of it.



What happens if we add a element that displaces the root of the tree, *e.g.*, 10? Where should 6 go, and how should its children be shifted to maintain both invariants?

Rather than worrying about preserving both invariants at once, we'll place a new element in the tree in such a way as to preserve one of the invariants immediately and then fix its position in the heap to fix the second. It is relatively easy to place a new element in the heap to obey the structural invariant: just respect the top-down, left-to-right order we've discussed.

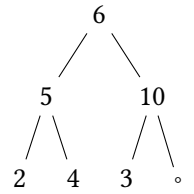


Note that this is particularly elegant with our array-based implementation of a heap. Insertion that preserves completeness is simply adding onto the end of the array:

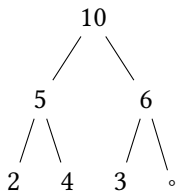
$$[6, 5, 3, 2, 4, 10, \ldots]$$

One downside is that like an array list, we will need to grow our backing array, *e.g.*, double its size, when it is full. We also need to make an amortized analysis arguing that this growth operation is irrelevant when considering sequences of add operations. All of these details are identical to those discussed early with array lists, so we won't discuss them here in favor of focusing on details specific to heaps.

With our element inserted into the heap, we must fix up its position to maintain the semantic invariant. To do this, note that in our heap diagram above, 10 is out of position with respect to its parent, 3. We can fix this by simply swapping 10 with 3.



Now 10 is out of position with respect to its new parent, 6. To fix this, we swap them, leaving 10 in its final position at the root of the tree.



This fix-up operation is called *sifting* or *percolation*, specifically *sift up* and *percolate up*. We repeatedly push the inserted element up the heap until it is in a position where the semantic invariant is preserved. Note that because the element that we swap with its parent is always greater than its parent, the semantic invariant over the other branch is preserved, ensuring correctness of the operation.

Thus, addition into the priority queue with a heap has the following steps:

(a) Add the element onto the end of the heap.

(b) Sift/percolate up the element into its proper position in the array.

The first operation takes (amortized) $\mathcal{O}(1)$ time as it is equivalent to array list addition (to end of the list). In the worst case, the sifting operation takes $\mathcal{O}(\log n)$ time because the inserted element may be sifted from leaf to root in the tree which requires a number of swaps equal to the height of the tree. Thus, overall the runtime of add is (amortized) $\mathcal{O}(\log n)$.

Poll Analogously to `add`, we have the issue that by removing the root of the tree, we end up potentially disrupting both invariants. To get around this, we preserve the structural invariant by moving the last element of the priority queue to the root (preserving structure) and then *sift down*/*percolate down* the element into its proper position in the priority queue.

Sift down operates analogously to the sift up operation described for add. The only wrinkle is that we have a choice of whether we sift down the left-hand branch or the right-hand branch of the node. Either leads to a correct implementation, however, to minimize the number of potential swaps we need to perform, we should favor the branch with the larger element, increasing the likelihood that we find an element that is less than this one.

Summarizing the poll operation:

(a) Replace the root with the last element in the heap.

(b) Sift/percolate down the new root into its proper position in the array.

(c) Return the old root value.

Like add, the first step takes constant time and the sifting takes $\mathcal{O}(\log n)$ time for an overall time complexity of $\mathcal{O}(\log n)$.

13.6 Heap Sort

With our heap structure defined, we can readily implement a sorting algorithm using it:

(a) Insert the elements into a min-heap.

(b) Repeatedly poll the heap until it is empty, reading off the element in ascending order in the process.

Insertion takes $\mathcal{O}(n \log n)$ time because for each of the n elements, we perform an $\mathcal{O}(\log n)$ add. Likewise, we require n calls to poll which takes $\mathcal{O}(n \log n)$ time overall. Thus, the overall run time of *heap sort* is $\mathcal{O}(n \log n)$, same as merge sort and quicksort. The constant factors associated with heap sort are higher than quicksort, in particular, the need for an auxiliary array to hold the heap. However, unlike quicksort which has worst case $\mathcal{O}(n^2)$ time for bad pivots, heap sort has consistent $\mathcal{O}(n \log n)$ performance which may be desirable depending on the problem at hand.