Lecture Summary

In this lecture, we learned about the ADT Priority Queue. A Priority Queue allows us to insert items and assign them priorities. Then, it allows us to always retrieve (and delete) the highest-priority element. We saw an efficient implementation using a heap data structure, and learned about the heap property and how to maintain it.

Earlier in class, we have seen two simple structures that manage access to shared resources: queues and stacks. Stacks access the data in a LIFO (last-in-first-out) order, in contrast to the FIFO (first-in-first-out) order in queues. In the real world, we often employ queues to achieve some notion of fairness, because we feel it is right to first serve those who arrived first. Examples include:

1. A line at a restaurant or grocery store. When customers arrive, we want to be fair to them, so we serve them in the order in which they arrive.

2. An operating system. Often, when the operating system is running, there are different programs that need access to the same resources (memory, processor, printer, …). Perhaps, they should receive access in the order in which they requested it.

These queues process items in the exact arrival order, and do not naturally include different priorities. For instance, a restaurant may want to give priority to customers who have made a reservation. The operating system should grant higher priority to the code for handling inputs from mouse and keyboard, as opposed to some long-running scientific calculation. Additional examples would include the following:

- Airport landing rights. Planes that are at risk of running out of fuel or are having emergencies should have high priority for landing. Planes that are ahead of their scheduled time might be given lower priority.

- Emergency room access in a hospital. While in general, patients should perhaps be processed in order of arrival, patients in dire need of medical attention (gunshot victim? heart attack?) should be given priority over broken legs or high fever.

So we would like a data structure that supports more flexibility, namely, assigning priorities to items, and processing in order of priority, while adding and removing items.

1 Abstract Data Type: Priority Queue

The priority queue is a data type supporting exactly these types of operations. The key abilities are:

1. Adding an item, while assigning it a priority.

2. Returning the item of highest priority currently stored.

3. Deleting the item of highest priority from the queue.
Notice that there is deliberately no way to return or delete any item other than the one of highest priority. Priority queues serve a particular purpose, which is to process items by priority; for this goal, access to other elements is not necessary.

In order to find the item of highest priority, we need to be able to compare the priorities of two items, both of the type T stored in the priority queue. There are (at least) two natural ways of implementing this comparison.

1. The data type T itself allows comparison of priorities of objects, e.g., by overloading the comparison operators. Thus, for any two objects T a, b, we can directly test if a < b, which returns a comparison of their priorities. Thus, priority is stored inside the object.

2. Priorities are kept separately, for instance as integers. Then, when we add a new item to the priority queue, we also need to tell the priority queue what the item’s priority is. Then, the queue needs to store two things for each item: T data, and int priority. Through this approach, the priority is determined only when the item is inserted.

The first method is perhaps easier to implement, but the second is more flexible. In particular, if you want to have the same items have different priorities based on the context (imagine people who will both visit an emergency room and go to a restaurant, with or without reservation), the priority cannot be an inherent property of the item itself.

For the purpose of these lecture notes, to keep notation simple and focus on the essentials, we will present the abstract data type under the first approach, i.e., we assume that T overloads the comparison operators. In practice, the second approach is perhaps a better choice.\footnote{Another approach, probably even better, is to implement a separate comparison function on objects of type T, and pass that into your data structure at initialization. This can be done by passing either a pointer to your comparison function, or an object (often called Comparator) which contains just the function.} Now that we have discussed how objects are compared, we can formalize the functions that the ADT Priority Queue should support:

```cpp
template <class T>
class PriorityQueue {
    void add (const T & item); // adds an item to the priority queue
    T peek() const; // returns the highest priority item
    void remove(); // removes the highest priority item
    bool isEmpty(); // determines if the priority queue is empty
    void changePriority (T & data, int new priority); // changes the priority of an item.
    // The textbook does not include this, but it can be very useful in many applications
}
```

Above, we discussed landing rights, hospital room emergencies, and operating systems as some application areas of priority queues. Truth be told, computer scientists are rarely called upon to implement hospital admissions systems; and even in operating systems, it is not particularly important to have the best data structures for dealing with the priorities of processes — there just aren’t that many of them. The true reason to study priority queues is for their role in search algorithms, which are the key part of much of the work in Artificial Intelligence and such applications as route planning. Building an AI (for many games) often involves exploring a state space (for a game, this could be the positions where all pieces or characters are located, their individual status, etc.). Some states are good, some are bad, and we would like to find a way to get from the current state to a good one. Since there is usually a huge number of states, exploring them in an intelligent order is crucial — this is what algorithms like A* search and Dijkstra’s Algorithm do. They assign numerical values to states, and always next explore the most promising one. From that state, several new states may be discovered, and they need to be considered for exploration, depending on their priority.
This application also suggests why a function `changePriority` could be useful. For instance, suppose that during the exploration so far, you’ve found a way to get your game character to a certain place, at the cost of losing almost all energy. If that place is surrounded by monsters, it may not be worth exploring for now. But suppose that by exploring something else, we now discover a way to get to the same place with almost all energy still in place. Now, exploring from that state should have higher priority, as it looks more promising. In other words, we’d like to now change the priority of this state.

As we discussed above, depending on how we think about our application, we can think of priority as inherent in an item, or determined only when the item is added to the data structure. In the former case, the priority is assigned at creation of the item, while in the latter case, it is assigned when `add` is called.

One natural question is whether we should allow items with the same priority in the priority queue at the same time. If we do, then there is no guarantee which of them will be returned by `peek` (and removed by `remove`, though usually, you would want to guarantee that it’s the same one). After all, if they have exactly the same priority, by definition, each is equally important. If that’s not what you want, then you should change your definition of priority to break ties in the way you want them to be broken. For instance, if you want to break ties in FIFO order, then the priority should be a pair `(originalPriority, additionTime)`, and item `a` has higher priority than `b` if it has higher `originalPriority`, or the same `originalPriority` and earlier `additionTime`.

2 Implementation of a Priority Queue

Notice that even though “Priority Queue” has the word “Queue” in it (because there are some conceptual similarities), it is not a good idea to implement it as a queue; in fact, it’s not clear how you would even do that.

The two naïve implementations would be using a (unsorted or sorted) linked list or array. For a linked list or unsorted array (or `List` type), the implementation would be as follows:

**Add**: Add the item at the tail or the head of the linked list, or at the end of the array. This runs in Θ(1).

**Peek**: To return the highest-priority element, we now have to search the entire array or list, so it takes Θ(n).

**Remove**: To remove the highest-priority element, we first have to find it (linear search). In an array, we also have to shift array contents to fill the gap. So this takes Θ(n).

So a linked list or unsorted array are great for inserting elements, but not good at finding high-priority elements (or removing them). An alternative is to sort the array (or `List` or linked list) by priority, with the highest priority at the end of the array. Then, the implementations would work as follows:

**Add**: Search for the correct position (O(log n) using binary search in an array or `List`, linear search in a linked list). Once the correct position is found, insert the element there. In an array, this involves shifting everything to the right of the inserted element over by one position. So in either case, the cost is Θ(n).

**Peek**: Returns the last element, so it runs in Θ(1).

**Remove**: Removes the last element, e.g., by decreasing the `size` counter, so it takes Θ(1).

Thus, a sorted array or linked list is fast at looking up and removing high-priority elements, but pays with linear insertion cost.

2.1 An Implementation using a Heap

The “sorted array” implementation of a Priority Queue is going in the right direction: it makes sure we can access the highest-priority element easily, by putting it in a designated spot. But beyond that, it is somewhat
overkill. Just to access the highest-priority element, it is not necessary to keep the array entirely sorted — it would be enough for it to be “kind of” sorted, so that after removing the highest-priority element, the next one can be found from among only few alternatives.

This is exactly the clever idea underlying a heap. A heap is a complete binary tree with the following additional key property on the contents of its nodes:

**Heap Property:** For each node $u$, the priority of the content at $u$ is at least as high as for all of $u$’s children.

![Figure 1: An illustration of the heap property, for a Max-Heap. The given tree is a complete binary tree. It satisfies the heap property at all nodes except one: the node with priority 3 has a child (its left child) with priority 5, which is not supposed to happen. Notice that for any node, it does not matter which of its children has higher priority, so long as both have no higher priority than the node itself.](image)

Notice that when we are implementing a Priority Queue using a heap, it is our responsibility to ensure that the tree remains a complete binary tree, and the Heap Property is maintained.

We mentioned above that a heap should be a complete binary tree. This means that all leaves are within at most one level of each other, and all leaves at the lowest level are as far left as possible. As we discussed in previous lectures, this implies that the heap can easily be stored in an array. For any node $i$, the parent is then located at array index $\lfloor i/2 \rfloor$, while the children are at $2i$ and $2i+1$.

We can now proceed to implementing the functions for a priority queue. We will assume that the binary tree is stored in an array $a$, and that the root is in $a[1]$. $a[0]$ is unused, to keep our math simpler. The variable `size` captures the number of elements currently stored in the array, so it starts at 0. We will not worry about exceeding the array size — this can be taken care of by using our `List<T>` data type instead of an actual array.

**Peek:** This is particularly easy, since the highest-priority element will always be at the root node.

```cpp
T peek() const { return a[1]; }
```

**Add:** When we add an element, we first wonder where to add it. Since we want to maintain the “complete binary tree” property, there is really only one place: as a new leaf as far to the left as possible. (In array positions, this translates to inserting it at $a[\text{size}+1]$.

But inserting it there may violate the heap property, if the newly added element has higher priority than its parent. So we need to fix the heap property after adding the element. We do so by swapping

---

2 The data structure `heap` shares only the name with the heap memory used to allocate objects; don’t confuse the two.

3 If we build a heap using integer priorities, is is up to us to decide whether larger or smaller integers encode higher priority. A heap that puts the smallest number at the top is called a Min Heap, while one that puts the largest number at the top is called a Max Heap. Obviously, it is easy to change from one to the other by switching comparison operators, or multiplying all numbers with -1.
it with its parent, if necessary, and then continuing at that node, having the element “trickle up” until it actually has lower priority than its parent node, or reaches the root. This gives us the following:

```c++
void add (const T & data)
{
    size++;
    a[size] = data;
    trickleUp(size);
}

void trickleUp (int pos)
{
    if (pos > 1 && a[pos] > a[pos/2])
    {
        swap (a, pos, pos/2);
        trickleUp(pos/2);
    }
}
```

Remove: In trying to remove the highest-priority element, our first thought would be to just delete that element from the array. But we can’t have a tree without a root. This would perhaps suggest filling the root with the one of its children that has larger priority, then recursing on the child. The problem with that approach is that by the time we reach the lowest level of the tree, we may end up swapping up a leaf from the “middle”, i.e., violating the part of the “complete binary tree” property requiring that the leaves are as far left as possible.

A slightly improved way is to instead move the very last leaf’s content into the root first. That way, we know which leaf is being moved. The rest of the way, we just do swap operations, trickling down this new root to where it actually belongs. This gives us the following solution.

```c++
void remove ()
{
    swap (a, 1, size); // swaps the highest priority element with the rightmost leaf
    size --; // we want to consider the former root "deleted"
    trickleDown (1); // trickles down, so the former leaf will go where it needs
}
```

void trickleDown (int pos)
// we only covered this in the next lecture, but it kind of belongs here.
{
    if (pos is not a leaf) // implement test as 2*pos <= size
    {
        i = child of pos of highest priority;
        // If you only have one child, return that child.
        // Else, use comparisons to see which has highest priority
        if (a[i] > a[pos])
        {
            swap (a, i, pos);
            trickleDown (i);
        }
    }
}