Heaps

Step 1

In the previous chapter, we discussed the **Queue** ADT, which can be thought of like a grocery store line: the first person to enter the line (queue) is the first person to come out of the line (queue), or so we would hope... However, what if, instead of a grocery store, we were talking about an emergency room? Clearly there is some logic regarding the order in which we see patients, but does an emergency room line follow a queue?

Say we have an emergency room line containing 10 individuals, all with pretty minor injuries (e.g. cuts/bruises). Clearly, the order in which these 10 patients are treated doesn't really matter, so perhaps the doctor can see them in the order in which they arrived. Suddenly, someone rushes through the door with a bullet wound that needs to be treated immediately. Should this individual wait until all 10 minor-injury patients are treated before he can be seen?

Clearly, we need an ADT that is similar to a **Queue**, but which can order elements based on some *priority* as opposed to blindly following the "First In First Out" (FIFO) ordering. The ADT that solves this exact problem is called the **Priority Queue**, which is considered a **"Highest Priority In, First Out" (HPIFO)** data type. In other words, the *highest priority* element currently in the **Priority Queue** is the first one to be removed.

Step 2

Recall from when we introduced the previous ADTs that an ADT is defined by a set of functions. The **Priority Queue** ADT can be formally defined by the following set of functions:

- **`insert(element)`:** Add `element` to the Priority Queue
- **`peek()`:** Look at the highest priority element in the Priority Queue
- **`pop()`:** Remove the highest priority element from the Priority Queue

Although we are free to implement a **Priority Queue** any way we want (because it is an ADT), you'll soon realize that the data structures we've learned about so far make it a bit difficult to guarantee a good worst-case time complexity for *both* insertion and removal. We could theoretically use a **sorted Linked List** to back our **Priority Queue**, which would result in O(1) peeking and removing (we would have direct access to the highest priority element), but insertion would be O(*n*) to guarantee that our list remains sorted (we would have to scan through the sorted list to find the correct insertion site). Likewise, we could theoretically use an **unsorted Linked List** to back our **Priority Queue**, which would result in O(1) insertion (we could just insert at the head or tail of the list), but peeking and removing would be O(*n*) (we would have to scan through the unsorted list to find the highest priority element). If you were to implement this using a **sorted** or **unsorted Array**, the worst-case time complexities would be the same as for a **sorted** or **unsorted Linked List**.

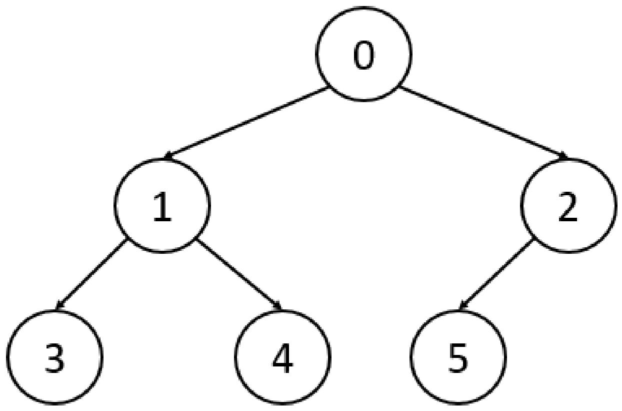
It turns out that there is a specific data structure that developers typically use to implement the **Priority Queue** ADT that guarantees **O(1)** peeking and **O(log *n*)** insertion and removal in the worst case. In this section of the text, we will explore the first of our tree structures: the **Heap**.

Step 3

The **heap** is a tree that satisfies the ***Heap Property***: For all nodes *A* and *B*, if node *A* is the parent of node *B*, then node *A* has *higher priority* (or equal priority) than node *B*. In this text, we will only discuss **binary heaps** (i.e, heaps that are binary trees), but this binary tree constraint is not required of heaps in general. The **binary heap**, has three constraints (two of which should hopefully be obvious):

- ***Binary Tree Property***: All nodes in the tree must have either 0, 1, or 2 children
- ***Heap Property*** (explained above)
- ***Shape Property*:** A heap is a ***complete*** tree. In other words, all levels of the tree, except possibly the bottom one, are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right

Below is an example of a **heap** (higher priority is given to smaller elements):

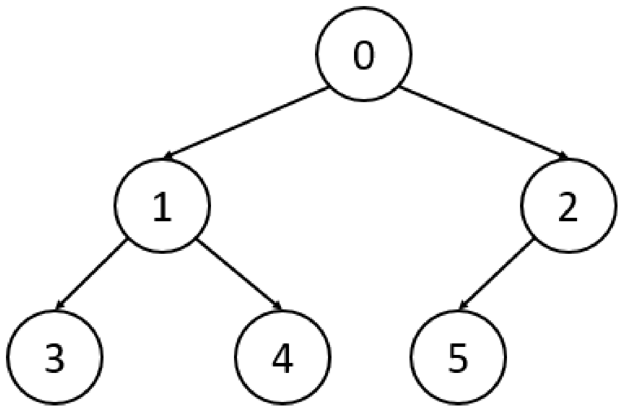


We will further clarify the notion of "priority" in a **heap** in the next step.

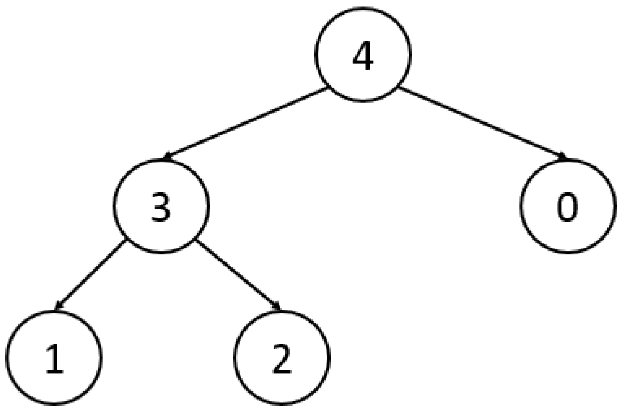
Step 4

We mentioned the term "priority", but what does it formally mean in the context of elements in a tree? The concept of *priority* can be more easily understood by looking at the two types of **heaps**: the **min-heap** and the **max-heap**.

A **min-heap** is a heap where every node is ***smaller*** than (or equal to) all of its children (or has no children). In other words, a node *A* is said to have *higher priority* than a node *B* if *A* < *B* (i.e., when comparing two nodes, the *smaller* of the two has higher priority). Below is an example of a **min-heap** (which happens to be the same example as the previous step):



A **max-heap** is a heap where every node is ***larger*** than (or equal to) all of its children (or has no children). In other words, a node *A* is said to have *higher priority* than a node *B* if *A* > *B* (i.e., when comparing two nodes, the *larger* of the two has higher priority). Below is an example of a **max-heap**:

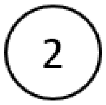


Note: We hinted at this above by sneaking in "or equal to" when describing **min-heaps** and **max-heaps**, but to be explicit, note that elements of a **heap** are interpreted as *priorities*, and as a result, it makes sense for us to **allow duplicate priorities** in our **heap**. Going back to the initial hospital example that motivated this discussion in the first place, what if we had *two* victims with bullet wounds, both of whom entered the hospital at the same time? The choice of which patient to see first is arbitrary: both patients have equally high priorities. As a result, in a **heap**, if you encounter duplicate priorities, simply break ties arbitrarily.

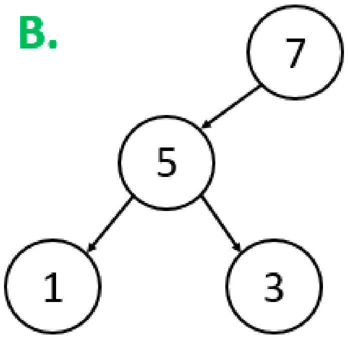
Step 5

EXERCISE BREAK: Which of the following are valid binary heaps? (Select all that apply)

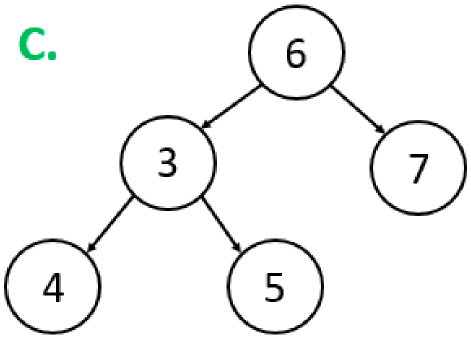
A.



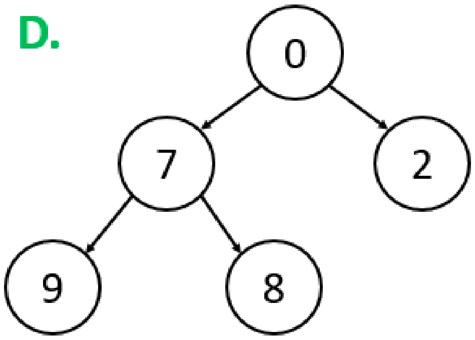
B.



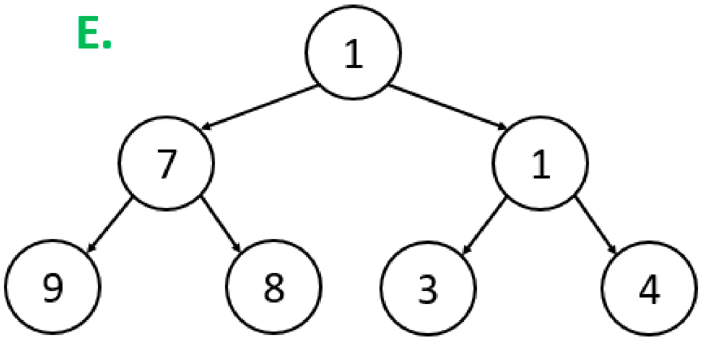
C.



D.



E.



To solve this problem please visit https://stepik.org/lesson/28863/step/5

Step 6

Given the structure of a heap (specifically, given the *Heap Property*), we are guaranteed that any given element has a higher priority than all of its children. As an extension, it should hopefully be clear that any given element must also have a higher priority than all of its *descendants*, not just its direct children. Thus, it should be intuitive that the root, which is the ancestor of all other elements in the tree (i.e., all other elements of the tree are descendants of the root) must be the highest-priority element in the heap.

Thus, the "algorithm" to peek at the highest-priority element in a heap is very trivial (so trivial, that we won't even consider it an algorithm):

```
peek():
    return the root element
```

Since we can assume that our implementation of a heap will have direct access to the root element, the **peek** operation is **O(1)** in the worst case.

Step 7

The first heap algorithm we will discuss is the insertion operation.

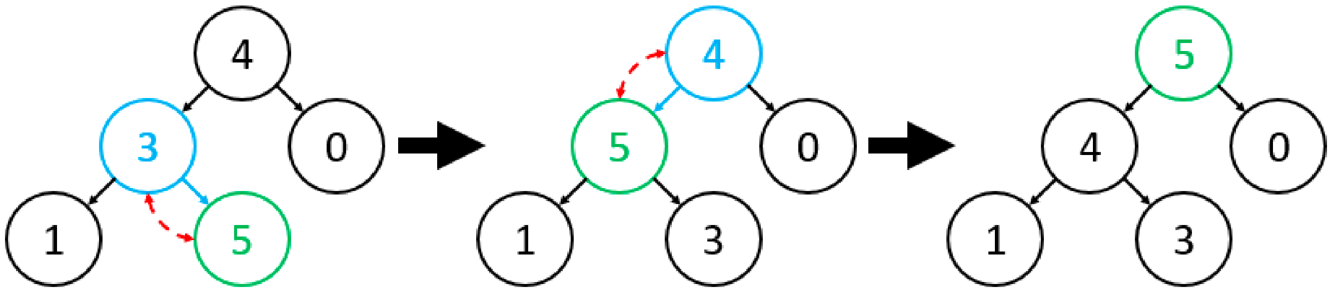
First, recall that one of the constraints of a heap is that it must be a full tree. Given this constraint, the first step of the heap insertion algorithm is quite intuitive: simply insert the new element in the next open slot of the tree (the new node will be a leaf in the bottom level of the tree, by definition of "next open slot"). In doing so, we have maintained the *Shape Property*, so the only constraint that might be violated is the *Heap Property*: the new element might potentially have higher priority than its parent, which would make the heap invalid.

Thus, the next (and last) step of the insertion algorithm should be intuitive as well: we must fix the (potentially) disrupted *Heap Property* of our tree. To do so, we must **bubble up** the newly-inserted element: if the new element has a higher priority than its parent, swap it with its parent; now, if the new element has higher priority than the new parent, swap; repeat until it has reached its correct place (either it has lower priority than its parent, or it is the root of the tree).

Formally, the pseudocode of heap insertion is as follows:

```
insert(element):
    place element in next open slot (as defined by a "full tree")
    while element has a parent and element.priority > element.parent.priority:
        swap element and element.parent
```

Below is a visualization of the heap insertion algorithm, where we insert 5 into the existing max-heap:



Because of the *Shape Property* of a heap, we are guaranteed to have a perfectly balanced binary tree, which means that we are guaranteed to have O(log *n*) levels in our tree. If we have *L* levels in our tree, in the worst case, we do *L*-1 comparisons for the bubble up process. As a result, if *L* = O(log *n*) and we do O(*L*) comparisons for the insertion algorithm, the overall **insertion** algorithm is **O(log *n*)**.

Step 8

The second heap algorithm we will discuss is the removal operation, "pop." Just like the insertion algorithm, it is actually quite simple when we use the constraints of a heap to derive it.

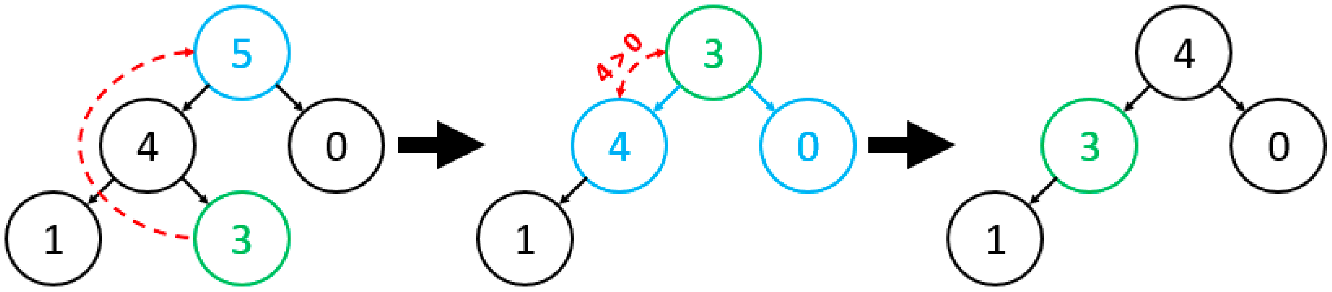
First, recall that the root element is by definition the highest-priority element in the heap. Thus, since we always remove the highest-priority element of a heap, the first step is to simply remove the root. However, since we need a valid tree structure, we need some other node to become the new root. In order to maintain the *Shape Property* of the heap, we can simply place the "last" element of the heap (i.e., the rightmost element of the bottom row of the heap) as the new root.

By making the "last" element of the heap the new root, we most likely have violated the *Heap Property*: the new root might have lower priority than its children, which would make the heap invalid. Thus, the next (and last) step of the "pop" algorithm should be intuitive as well: we must fix the (potentially) disrupted *Heap Property* of our tree. To do so, we must **trickle down** the newly-placed root: if the root has lower priority than its children, swap it with its highest-priority child; now, if it still has lower priority than its children in this new position, swap with its highest-priority child again; repeat until it has reached its correct place (either it has higher priority than all of its children, or it is a leaf of the tree).

Formally, the pseudocode of heap removal is as follows:

```
pop():
    replace the root with the rightmost element of the bottom level of the tree (call it "curr")
    while curr is not a leaf and curr has lower priority than any of its children:
        swap curr and its highest-priority child
```

Below is a visualization of the heap "pop" algorithm, where we pop from the existing max-heap:



Because of the *Shape Property* of a heap, we are guaranteed to have a perfectly balanced binary tree, which means that we are guaranteed to have O(log *n*) levels in our tree. If we have *L* levels in our tree, in the worst case, we do *L*-1 comparisons for the trickle down process. As a result, if *L* = O(log *n*) and we do O(*L*) comparisons for the insertion algorithm, the overall **pop** algorithm is **O(log *n*)**.

STOP and Think: When trickling down, why do we have to swap with the largest child specifically?

Step 9

EXERCISE BREAK: Which property (or properties) of a **binary heap** give rise to the **O(log *n*)** worst-case time complexity of the **insertion** algorithm? (Select all that apply)

To solve this problem please visit https://stepik.org/lesson/28863/step/9

Step 10

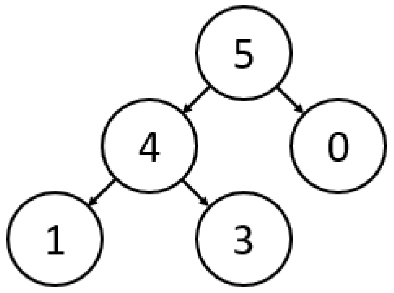
EXERCISE BREAK: Which property (or properties) of a **binary heap** give rise to the **O(log *n*)** worst-case time complexity of the **pop** algorithm? (Select all that apply)

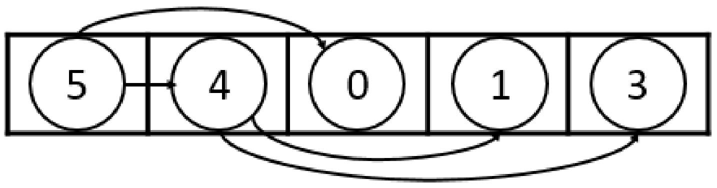
To solve this problem please visit https://stepik.org/lesson/28863/step/10

Step 11

Although we like visualizing a binary heap as a tree structure with nodes and edges, when it comes to implementation, by being a little clever, it turns out that we can store a binary heap as an array. Technically, any binary tree can be stored as an array, and by storing a binary tree as an array, we can avoid the overhead of wasted memory from parent and child pointers. Thus, given an index in the array, we can find the element's parent and children using basic arithmetic. An added bonus with binary heaps over unbalanced binary trees is that, since a binary heap is by definition a complete tree, there is no wasted space in the array: all elements will be stored perfectly contiguously.

Below is a diagram showing how we can represent an example binary max-heap as an array, where the tree representation is shown to the left and the array representation (with the edges maintained for the sake of clarity) is shown to the right:



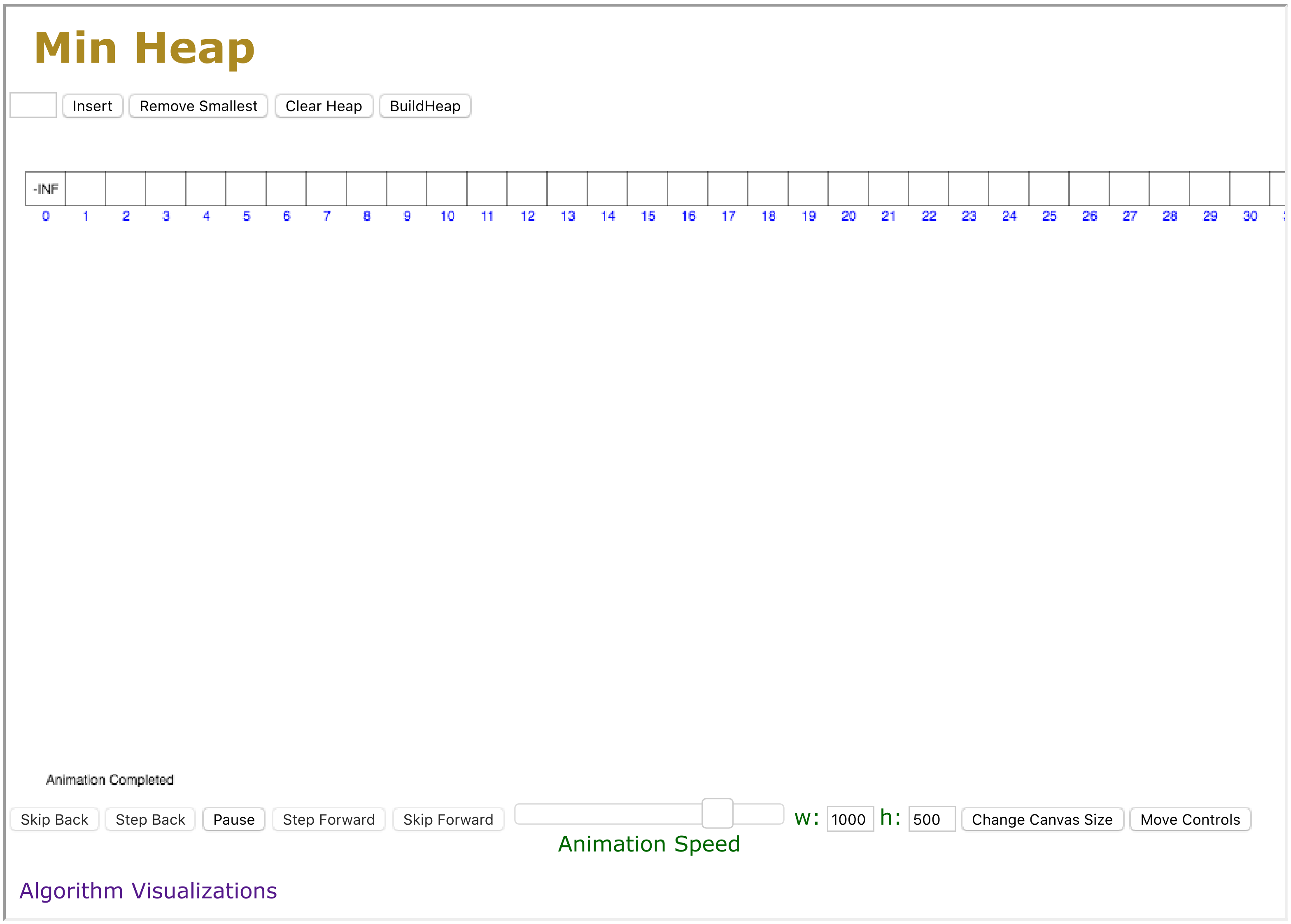


Notice that, in the array representation, we can access the parent, left child, or right child of any given element in constant time using simple arithmetic. For an element at index *i* of the array, using 0-based indexing,

- its **parent** is at index $\lfloor \frac{i-1}{2} \rfloor$
- its **left child** is at index $2i + 1$
- its **right child** is at index $2i + 2$

Step 12

Below is a visualization of the heap algorithms created by David Galles at the University of San Francisco. Notice how his visualization uses both the tree representation and the array representation simultaneously. Also, note that the heap he uses is a **min-heap**.



Step 13

EXERCISE BREAK: Say we have a heap represented as an array, using 0-based indexing, and we are looking at the element at index 101. At what index would we find its parent?

To solve this problem please visit https://stepik.org/lesson/28863/step/13

Step 14

EXERCISE BREAK: Which of the following statements are true? (Select all that apply)

To solve this problem please visit https://stepik.org/lesson/28863/step/14

Step 15

We have now very efficiently solved our initial problem of implementing a queue-like data type that can take into account a sense of ordering (based on "priority") of the elements it contains. We first described the **Priority Queue** ADT, which defined the functions we needed in such a data type.

Then, we dove into implementation specifics by discussing the **Heap** data structure, which is almost always used to implement a **Priority Queue**. Because of the constraints of the **Heap** data structure, we are able to guarantee a **worst-case** time complexity of **O(log *n*)** for both **inserting** and **popping** elements, as well as a **O(1)** worst-case time complexity for **peeking** at the highest-priority element.

In the next sections of this chapter, we will continue to explore other types of binary trees, and it will hopefully become clear why we would want to have so many different data structures and ADTs in our arsenal of tools.