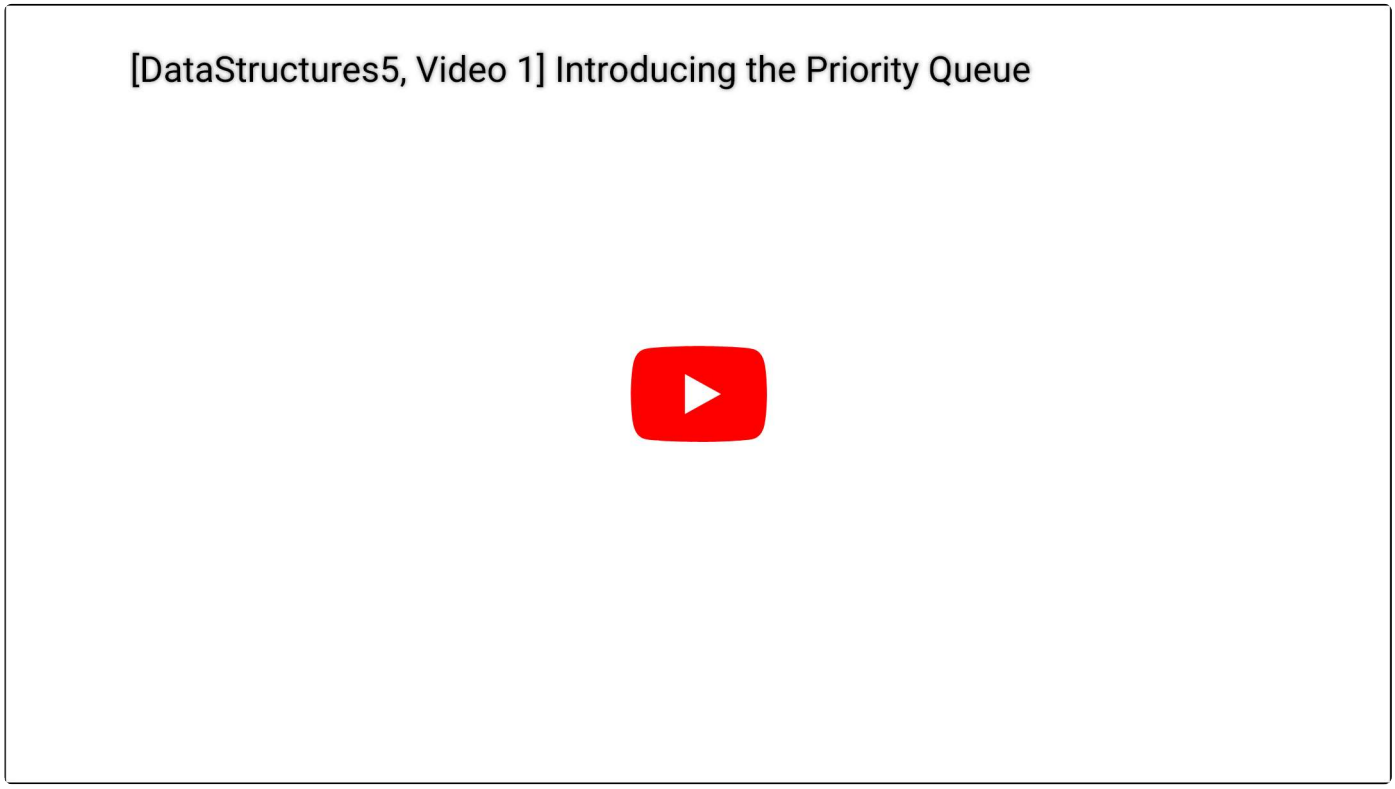
Binary Search Trees allowed us to efficiently search trees, only taking $log(N)$ time. This was because we could eliminate half of the elements of the tree at every step of our search. What if we cared more about quickly finding the *smallest* or *largest* element instead of quickly searching?



Introducing the Priority Queue

Now we come to the Abstract Data Type of a *Priority Queue*. To understand this ADT, consider a bag of items. You can add items to this bag, you can remove items from this bag, etc. The one caveat is that you can only interact with the smallest items of this bag.

```java
/** (Min) Priority Queue: Allowing tracking and removal of
  * the smallest item in a priority queue. */
public interface MinPQ<Item> {
    /** Adds the item to the priority queue. */
    public void add(Item x);
    /** Returns the smallest item in the priority queue. */
    public Item getSmallest();
    /** Removes the smallest item from the priority queue. */
    public Item removeSmallest();
    /** Returns the size of the priority queue. */
    public int size();
}
```

Using Priority

Where would we actually use this structure or need it?

- Consider the scenario where we are monitoring text messages between citizens and want to keep track of unharmonious conversations.
- Each day, you prepare a report of M messages that are the most unharmonious using a `HarmoniousnessComparator`.

Let's take this approach: Collect all of the messages that we receive throughout the day into a single list. Sort this list and return the first M messages.

```java
public List<String> unharmoniousTexts(Sniffer sniffer, int M) {
    ArrayList<String>allMessages = new ArrayList<String>();
    for (Timer timer = new Timer(); timer.hours() < 24; ) {
        allMessages.add(sniffer.getNextMessage());
    }

    Comparator<String> cmptr = new HarmoniousnessComparator();
    Collections.sort(allMessages, cmptr, Collections.reverseOrder());

    return allMessages.sublist(0, M);
```

Potential downsides? This approach will use $\Theta(N)$ space when actuality we only really need to use $\Theta(M)$ space.

Exercise 13.1.1. Complete the method listed above, `unharmoniousTexts` with the same functionality as described using only $\Theta(M)$ space.

Priority Queue Implementation

We solved the same problem using the Priority Queue ADT, making memory more efficient. We can observe that the code is slightly more complicated, but this is not always the case.

Remember that ADT are similar interfaces and the implementation is still to be defined. Let's consider possible implementations using the data structure implementations we already know, analyzing the **worst case** runtimes of our desired operations:

- Ordered Array
  - `add`: $\Theta(N)$
  - `getSmallest`: $\Theta(1)$
  - `removeSmallest`: $\Theta(1)$
- Bushy BST
  - `add`: $\Theta(logN)$
  - `getSmallest`: $\Theta(logN)$
  - `removeSmallest`: $\Theta(logN)$
- HashTable
  - `add`: $\Theta(1)$
  - `getSmallest`: $\Theta(N)$
  - `removeSmallest`: $\Theta(N)$

Exercise 13.1.2. Explain each of the runtimes above. What are the downfalls of each data structure? Describe a way of modifying one of these data structures to improve performance.

Can we do better than these suggested data structures?

Summary

- Priority Queue is an Abstract Data Type that optimizes for handling minimum or maximum elements.
- There can be space/memory benefits to using this specialized data structure.
-

- Implementations for ADTs that we currently know don't give us efficient runtimes for PQ operations.
  - A binary search tree is the most efficient among the other structures

Previous
21. Heaps and Priority Queues

Next
21.2 Heaps

Last updated 1 year ago