Imagine you’re managing a busy airport. Flights are constantly landing and taking off, and you need to quickly find the next most important flight—an emergency landing or a VIP departure. At the same time, new flights must be integrated into the schedule. How do you track all this while finding the highest-priority flight quickly? Without an efficient data structure, you’d have to scan the entire schedule every time a decision is needed, which can be slow and error-prone as the number of flights grows. The time complexity of this inefficient system will be O(n)O(n) for each decision, where nn is the number of flights because it requires scanning the entire schedule to find the highest-priority flight.

The solution is heaps. **Heaps** are a special data structure that helps you efficiently manage priorities. With a min heap, you can always find the flight with the earliest priority, and with a max heap, you can focus on flights that have been waiting for the longest—all while making updates quickly when new flights are added.

A heap is a specialized binary tree that satisfies the heap property:

* **Min heap**: The value of each node is smaller than or equal to the values of its children. The root node holds the minimum value. A min heap always prioritizes the minimum value.
* **Max heap**: The value of each node is greater than or equal to the values of its children. The root node holds the maximum value. A max heap always prioritizes the maximum value.
* **Priority queue:** A priority queue is an abstract data type retrieves elements based on their custom priority. It is often implemented using a heap for efficiency.

A heap is a specific data structure with a fixed ordering (min or max), while a priority queue is an abstract data type that handles custom priority requirements for elements.

A heap is a specific data structure with a fixed ordering (min or max), while a priority queue is an abstract data type that handles custom priority requirements for elements.

Heaps are typically implemented using arrays to efficiently access the parent and child nodes. The major operations performed on heaps are:

* **Add:** This inserts a new element into the heap, which takes O(logn)O(logn)time.
* **Delete:** This removes the root element and rebalances the heap, taking O(logn)O(logn) time.
* **Peek:** This retrieves the smallest or largest element in O(1)O(1).

**Solving problems with a single heap**

Heaps are powerful because they allow us to maintain order (minimum or maximum) without needing full sorting, making operations much faster than other data structures like arrays or linked lists. They are also used when we need to repeatedly access the smallest or largest element in a dataset. For example, let’s look at the *Last Stone Weight* problem. In this problem, we are given an array of stone weights and tasked with repeatedly smashing the two heaviest stones together. If the two heaviest stones have the same weight, both are destroyed. If they have different weights, the smaller stone is destroyed, and the smaller stone’s weight reduces the larger stone’s weight. This process continues until there is at most one stone remaining.

To solve this problem, we need to repeatedly select the heaviest two stones, smash them together, and update the list of stones based on the result of each smash. As we want to efficiently retrieve the two largest stones at each step, a max heap is perfect for this problem. Using a max heap allows us to easily access and remove the heaviest stones while ensuring that the heap structure maintains its property of providing the largest element at the top.

To begin, we first build a max heap using the stone weights. Then, at each turn, we:

* Pop the two largest stones from the heap.
* If the two stones have the same weight, both are destroyed, and no stone is added back to the heap.
* If the two stones have different weights, the stone with the smaller weight is destroyed, and the stone with the larger weight is reduced by the smaller stone’s weight and pushed back into the heap for further processing.

This process continues until either one stone remains or none remains. The weight of the last remaining stone (or 0 if none remains) is returned as the result.

The time complexity of this heap-based solution is O(nlogn)O(nlogn), where nn is the number of stones. This is because building the heap takes O(n)O(n) time, and each of the nn smash operations involves heap insertion and deletion, each taking O(logn)O(logn) time. In contrast, the naive solution, which repeatedly scans the list to find the two largest stones, has a time complexity of O(n²)O(n²) due to the need for nn scans of O(n)O(n) each.

**Using two heaps for problem-solving**

In addition to using a single heap, there are several scenarios where two heaps can be employed in the heaps pattern to optimize the solution. One common use case is when we need to efficiently track a dataset’s smallest and largest elements, such as finding the median or balancing data streams. By maintaining one min heap for the smaller half of the data and one max heap for the larger half, we can quickly access the median or adjust the distribution of elements. Another use case is for problems involving intervals or ranges, where one heap can store one set of values (e.g., start times), and the other tracks the complementary set (e.g., end times) to efficiently identify valid intervals or ranges.

For example, let’s look at the *Find Right Interval* problem. In this problem, we are given an array of intervals and must find the right interval for each. A right interval starts after the current interval ends. If no such interval exists, return -1. The goal is to efficiently find the smallest start time that is greater than or equal to the current interval’s end-time

This problem can be solved using two min heaps**:** one for storing start times and the other for storing interval end times. The key idea is to process intervals based on their end times, ensuring we find the smallest start time from the start heap greater than or equal to its end time for each interval.

* We begin by populating two heaps with the start and end times.
* For each interval, we pop the smallest end time from the end heap and remove any start times smaller than the current end time from the start heap.
* If a valid start time remains in the start heap, it represents the right interval for the current interval.
* If no valid right interval exists, we return -1 for that interval.

The time complexity of this heap-based solution is O(nlogn)O(nlogn), where nn is the number of intervals. This is because we perform O(n)O(n) insertions and deletions for each of the two heaps (one for start times and the other for end times), each heap operation taking O(logn)O(logn) time. In contrast, the naive solution, which involves scanning through all the intervals for each interval to find the right interval, has a time complexity of O(n²)O(n²), as it requires nn comparisons for each of the nn intervals.

**Examples**

The following examples illustrate some problems that can be solved with this approach:

1. **Sliding window median:** Given an array of integers and a window size k, find the median of each sliding window of size k as it moves from left to right through the array.
2. **Find median of a number stream:** Given a continuous stream of numbers, find the median of the numbers seen so far after any insertion in constant time.

**Does your problem match this pattern?**

Yes, if any of these conditions are fulfilled:

* **Linear data:** If the input data is linear, it can be sorted or unsorted. I
  + A heap efficiently finds the maximum or minimum elements if the data is unsorted. Operations like insertion and deletion take O(logn)O(logn) time, ensuring fast access to the top elements.
  + If the data is sorted, a heap can still be useful when frequent insertions and deletions are required, as it allows for efficient updates and retrieval of the highest or lowest elements, with both insertion and deletion operations also taking O(logn)O(logn) time.
* **Stream of data:** The input data continuously arrives in real time, often in an unpredictable order, requiring efficient handling and processing as it flows in. Heaps automatically enforce priority ordering (e.g., largest weight, smallest cost, highest frequency). This saves you from manually resorting to or scanning each time your data changes.
* **Calculation of maxima and minima:** The input data can be categorized into two parts, and we need to repeatedly calculate two maxima, two minima, or one maximum and one minimum from each set.
* **Efficient retrieval of extreme values:** The solution needs to retrieve or update the min or max element repeatedly but cannot afford a full sort each time; a heap-based priority queue offers O(logn)O(logn) insertion/removal and O(1)O(1)retrieval.
* **Custom priority-based selection:** The problem involves selecting the next element based on specific priority at each step, such as processing the largest task or earliest event.

**Real-world problems**

Many problems in the real world use the two heaps pattern. Let’s look at some examples.

* **Video platforms:** As part of a demographic study, we’re interested in the median age of the viewers. We want to implement a functionality whereby the median age can be updated efficiently whenever a new user signs up for video streaming.
* **Gaming matchmaking**: Matching players of similar skill levels is crucial for a balanced and enjoyable gaming experience. By maintaining two heaps (one for minimum skill level and one for maximum skill level), matchmaking algorithms can efficiently pair players based on their skill levels.