Two Heaps: Introduction

Let’s go over the Two Heaps pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following

Overview

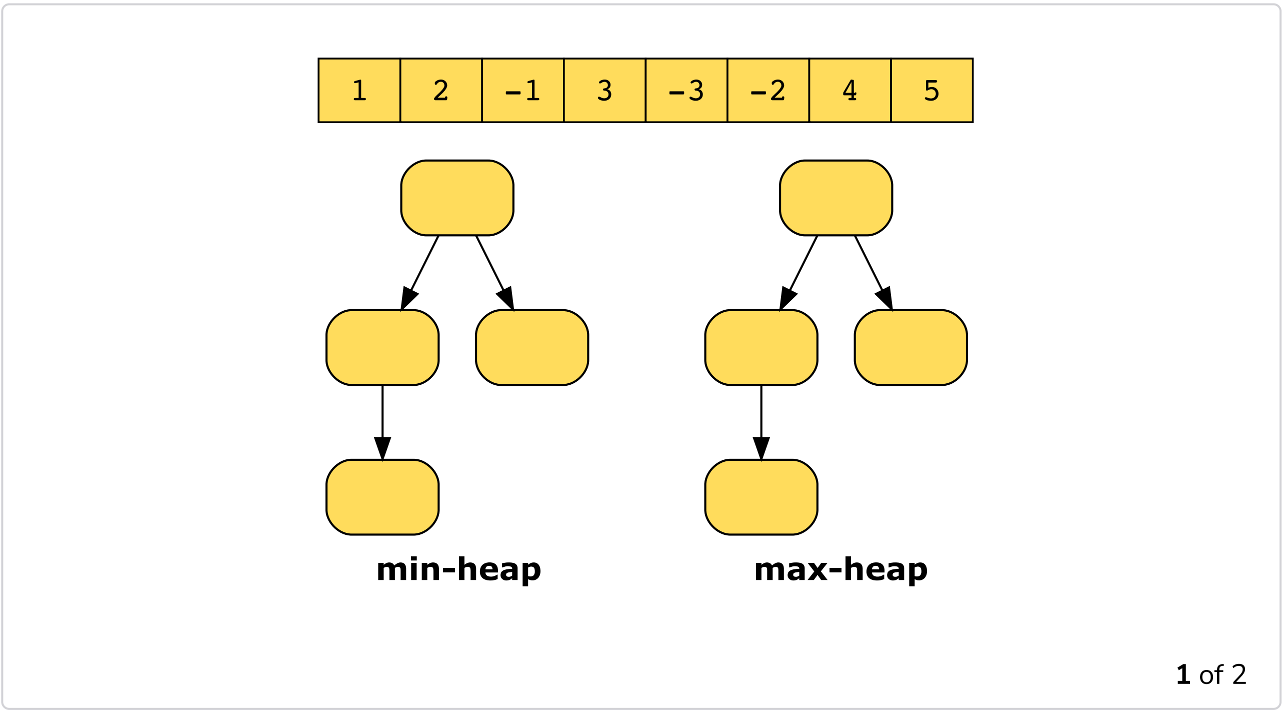
As the name suggests, the **two heaps** pattern uses either two min-heaps, two max-heaps, or a min-heap and a max-heap simultaneously to solve the problem.

Given that there are n elements in a heap, it takes $O(logn)$ time to insert an element in it, $O(logn)$ time to remove an element from it, and $O(1)$ time to access the element at the root of the heap. The root stores the smallest element in the case of a min-heap and the largest element in a max-heap.

In some problems, we’re given a set of data such that it can be divided in two parts. We can either use the first part to find the smallest element using the min-heap and the second part to find the largest element using the max-heap, or we can do the reverse and use the first part to find the largest element using the max-heap and the second part to find the smallest element using the min-heap.

There might be cases where we need to find the two largest numbers from two different data sets. We'll use two max-heaps to store two different data sets in that case. In other cases, we might need to find the two smallest numbers from two different data sets, and then we would use two min-heaps.

Let's look at the following illustration to understand how we can use heaps to find the smallest or largest number:

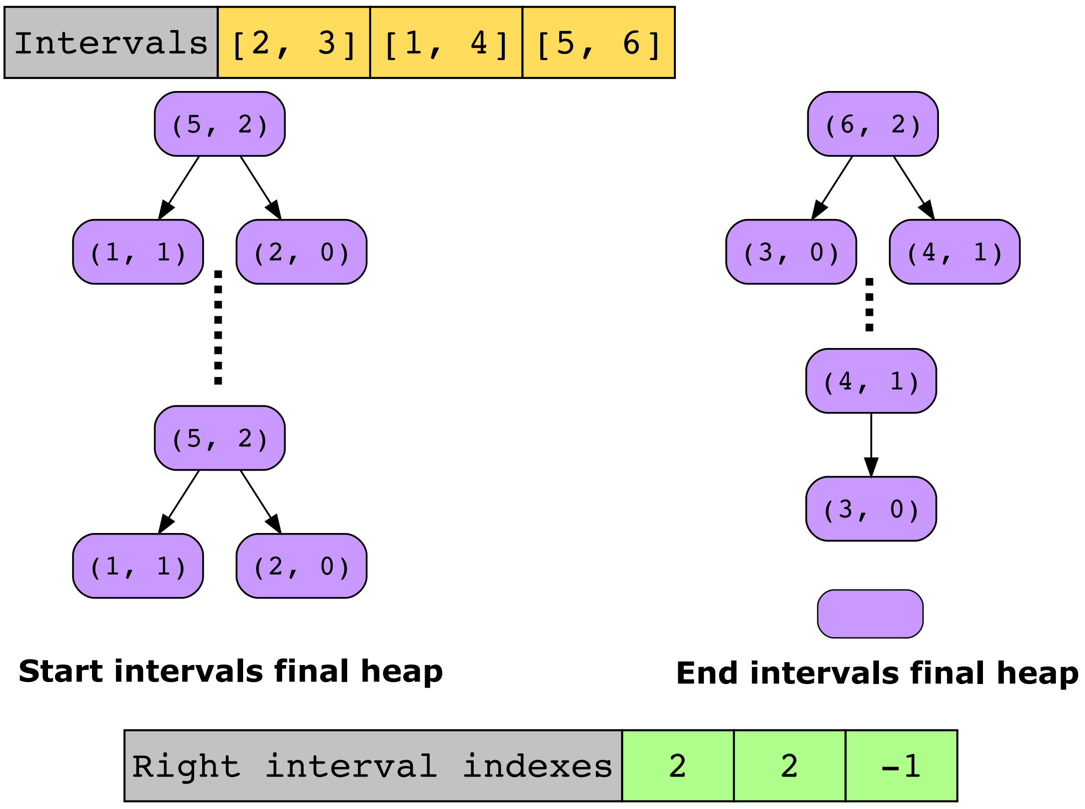


Examples

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

Find right interval

For each **interval[i] = [start[i], end[i]]** in the list of intervals, find the right interval. The right interval of an interval **i** is the earliest interval **j**, where **start[j] >= end[i]**. We'll maintain two max-heaps, one to store the start of the interval in the form:**(start[i], i)** and the second to store the end of the interval: **(end[i], i)**. Start popping the 2nd heap **(end[i], i)** and check for the right interval by comparing the top of 1st heap, **start[j]**. If there is no interval, **-1** will be added, else **j** will be added in the output array at index **i**.



1 of 2

Does my problem match this pattern?

- Yes, if both of these conditions are fulfilled:
  - We need to repeatedly calculate two maxima, two minima, or one maximum and one minimum, based on a changing set of data.
  - The input data is not sorted.
- No, if any of these conditions are fulfilled:

- We don’t need to track two extreme values (minima or maxima), but only one.
- When we don’t need to repeatedly calculate the extreme values (minima or maxima), but only need to calculate it a fixed number of times.
- The input data is already sorted—in which case, there is no benefit to using heaps.

Real-world problems

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

- **Video streaming:** During a user session, there is often a possibility that packet drops and buffering might occur. We want to record the

session, which could then be used to improve the user experience.
- **Netflix:** As part of a demographic study, we’re interested in the median age of our viewers. We want to implement a functionality whereby the median age can be updated efficiently whenever a new user signs up for video streaming.

Strategy time!

Match the problems that can be solved using the two heaps pattern.

> **Note:** Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.

Match The Answer

Select an option from the left-hand side

For a given set of intervals, find the earliest interval that starts after the end of a specific interval.

Two Heaps

Sort the characters of the given string by frequency.

Some other pattern

Find the medians of subarrays of size 3 in the array [1, 3, -1, -2, 5, 3].

Given the string “leetcodegrindnomore”, find the longest substring with 5 distinct characters.

Reset Show Solution Submit