CHAPTER

10 Heaps

Using F-heaps we are able to obtain improved running times for several network optimization algorithms.

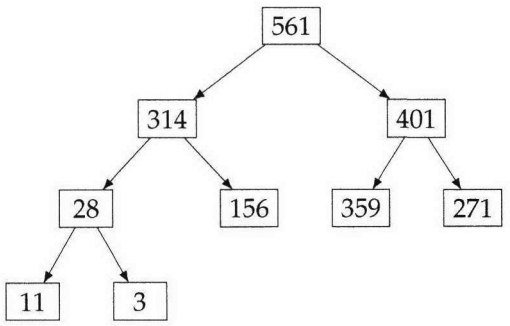
— *"Fibonacci heaps and their uses,"*
M. L. FREDMAN AND R. E. TARJAN, 1987

A *heap* is a specialized binary tree. Specifically, it is a complete binary tree as defined on Page 113. The keys must satisfy the *heap property*—the key at each node is at least as great as the keys stored at its children. See Figure 10.1(a) for an example of a max-heap. A max-heap can be implemented as an array; the children of the node at index i are at indices $2i + 1$ and $2i + 2$. The array representation for the max-heap in Figure 10.1(a) is $\langle 561, 314, 401, 28, 156, 359, 271, 11, 3 \rangle$.

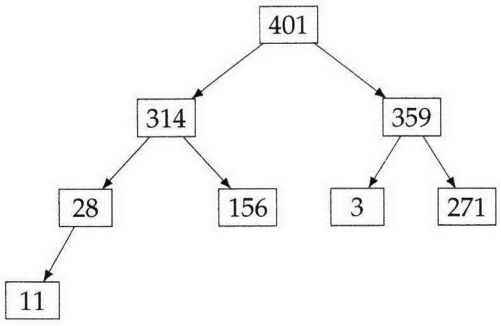
A max-heap supports $O(\log n)$ insertions, $O(1)$ time lookup for the max element, and $O(\log n)$ deletion of the max element. The extract-max operation is defined to delete and return the maximum element. See Figure 10.1(b) for an example of deletion of the max element. Searching for arbitrary keys has $O(n)$ time complexity.

A heap is sometimes referred to as a priority queue because it behaves like a queue, with one difference: each element has a "priority" associated with it, and deletion removes the element with the highest priority.

The *min-heap* is a completely symmetric version of the data structure and supports $O(1)$ time lookups for the minimum element.



(a) A max-heap. Note that the root holds the maximum key, 561.



(b) After the deletion of max of the heap in (a). Deletion is performed by replacing the root's key with the key at the last leaf and then recovering the heap property by repeatedly exchanging keys with children.

Figure 10.1: A max-heap and deletion on that max-heap.

Heaps boot camp

Suppose you were asked to write a program which takes a sequence of strings presented in "streaming" fashion: you cannot back up to read an earlier value. Your program must compute the k longest strings in the sequence. All that is required is the k longest strings—it is not required to order these strings.

As we process the input, we want to track the k longest strings seen so far. Out of these k strings, the string to be evicted when a longer string is to be added is the shortest one. A min-heap (not a max-heap!) is the right data structure for this application, since it supports efficient find-min, remove-min, and insert. In the program below we use a heap with a custom compare function, wherein strings are ordered by length.

```
def top_k(k, stream):
    # Entries are compared by their lengths.
    min_heap = [(len(s), s) for s in itertools.islice(stream, k)]
    heapq.heapify(min_heap)
    for next_string in stream:
        # Push next_string and pop the shortest string in min_heap.
        heapq.heappushpop(min_heap, (len(next_string), next_string))
    return [p[1] for p in heapq.nsmallest(k, min_heap)]
```

Each string is processed in $O(\log k)$ time, which is the time to add and to remove the minimum element from the heap. Therefore, if there are n strings in the input, the time complexity to process all of them is $O(n \log k)$.

We could improve best-case time complexity by first comparing the new string's length with the length of the string at the top of the heap (getting this string takes $O(1)$ time) and skipping the insert if the new string is too short to be in the set.

Use a heap when **all you care about** is the **largest** or **smallest** elements, and you **do not need** to support fast lookup, delete, or search operations for arbitrary elements. A heap is a good choice when you need to compute the k **largest or** k **smallest** elements in a collection. For the former, use a min-heap, for the latter, use a max-heap.

Table 10.1: Top Tips for Heaps

Know your heap libraries

Heap functionality in Python is provided by the `heapq` module. The operations and functions we will use are

- `heapq.heapify(L)`, which transforms the elements in `L` into a heap in-place,
- `heapq.nlargest(k, L)` (`heapq.nsmallest(k, L)`) returns the k largest (smallest) elements in `L`,
- `heapq.heappush(h, e)`, which pushes a new element on the heap,
- `heapq.heappop(h)`, which pops the smallest element from the heap,
- `heapq.heappushpop(h, a)`, which pushes `a` on the heap and then pops and returns the smallest element, and
- `e = h[0]`, which returns the smallest element on the heap without popping it.
-

It's very important to remember that `heapq` only provides min-heap functionality. If you need to build a max-heap on integers or floats, insert their negative to get the effect of a max-heap using

heapq. For objects, implement `__lt()__` appropriately. Problem 10.4 on Page 137 illustrates how use a max-heap.

10.1 Merge sorted files

This problem is motivated by the following scenario. You are given 500 files, each containing stock trade information for an S&P 500 company. Each trade is encoded by a line in the following format: `1232111,AAPL,30,456.12`.

The first number is the time of the trade expressed as the number of milliseconds since the start of the day's trading. Lines within each file are sorted in increasing order of time. The remaining values are the stock symbol, number of shares, and price. You are to create a single file containing all the trades from the 500 files, sorted in order of increasing trade times. The individual files are of the order of 5–100 megabytes; the combined file will be of the order of five gigabytes. In the abstract, we are trying to solve the following problem.

Write a program that takes as input a set of sorted sequences and computes the union of these sequences as a sorted sequence. For example, if the input is $\langle 3,5,7\rangle$, $\langle 0,6\rangle$, and $\langle 0,6,28\rangle$, then the output is $\langle 0,0,3,5,6,6,7,28\rangle$.

Hint: Which part of each sequence is significant as the algorithm executes?

Solution: A brute-force approach is to concatenate these sequences into a single array and then sort it. The time complexity is $O(n \log n)$, assuming there are n elements in total.

The brute-force approach does not use the fact that the individual sequences are sorted. We can take advantage of this fact by restricting our attention to the first remaining element in each sequence. Specifically, we repeatedly pick the smallest element amongst the first element of each of the remaining part of each of the sequences.

A min-heap is ideal for maintaining a collection of elements when we need to add arbitrary values and extract the smallest element.

For ease of exposition, we show how to merge sorted arrays, rather than files. As a concrete example, suppose there are three sorted arrays to be merged: $\langle 3,5,7\rangle$, $\langle 0,6\rangle$, and $\langle 0,6,28\rangle$. For simplicity, we show the min-heap as containing entries from these three arrays. In practice, we need additional information for each entry, namely the array it is from, and its index in that array. (In the file case we do not need to explicitly maintain an index for next unprocessed element in each sequence—the file I/O library tracks the first unread entry in the file.)

The min-heap is initialized to the first entry of each array, i.e., it is $\{3,0,0\}$. We extract the smallest entry, 0, and add it to the output which is $\langle 0\rangle$. Then we add 6 to the min-heap which is $\{3,0,6\}$ now. (We chose the 0 entry corresponding to the third array arbitrarily, it would be perfectly acceptable to choose from the second array.) Next, extract 0, and add it to the output which is $\langle 0,0\rangle$; then add 6 to the min-heap which is $\{3,6,6\}$. Next, extract 3, and add it to the output which is $\langle 0,0,3\rangle$; then add 5 to the min-heap which is $\{5,6,6\}$. Next, extract 5, and add it to the output which is $\langle 0,0,3,5\rangle$; then add 7 to the min-heap which is $\{7,6,6\}$. Next, extract 6, and add it to the output which is $\langle 0,0,3,5,6\rangle$; assuming 6 is selected from the second array, which has no remaining elements, the min-heap is $\{7,6\}$. Next, extract 6, and add it to the output which is $\langle 0,0,3,5,6,6\rangle$; then add 28 to the min-heap which is $\{7,28\}$. Next, extract 7, and add it to the output which is $\langle 0,0,3,5,6,6,7\rangle$; the min-heap is $\{28\}$. Next, extract 28, and add it to the output which is $\langle 0,0,3,5,6,6,7,28\rangle$; now, all elements are processed and the output stores the sorted elements.

```
def merge_sorted_arrays(sorted_arrays):
```

```
    min_heap = []
    # Builds a list of iterators for each array in sorted_arrays.
    sorted_arrays_iters = [iter(x) for x in sorted_arrays]

    # Puts first element from each iterator in min_heap.
    for i, it in enumerate(sorted_arrays_iters):
        first_element = next(it, None)
        if first_element is not None:
            heapq.heappush(min_heap, (first_element, i))

    result = []
    while min_heap:
        smallest_entry, smallest_array_i = heapq.heappop(min_heap)
        smallest_array_iter = sorted_arrays_iters[smallest_array_i]
        result.append(smallest_entry)
        next_element = next(smallest_array_iter, None)
        if next_element is not None:
            heapq.heappush(min_heap, (next_element, smallest_array_i))
    return result


# Pythonic solution, uses the heapq.merge() method which takes multiple inputs.
def merge_sorted_arrays_pythonic(sorted_arrays):
    return list(heapq.merge(*sorted_arrays))
```

Let k be the number of input sequences. Then there are no more than k elements in the min-heap. Both extract-min and insert take $O(\log k)$ time. Hence, we can do the merge in $O(n \log k)$ time. The space complexity is $O(k)$ beyond the space needed to write the final result. In particular, if the data comes from files and is written to a file, instead of arrays, we would need only $O(k)$ additional storage.

Alternatively, we could recursively merge the k files, two at a time using the merge step from merge sort. We would go from k to $k/2$ then $k/4$, etc. files. There would be $\log k$ stages, and each has time complexity $O(n)$, so the time complexity is the same as that of the heap-based approach, i.e., $O(n \log k)$. The space complexity of any reasonable implementation of merge sort would end up being $O(n)$, which is considerably worse than the heap based approach when $k \ll n$.

10.2 Sort an increasing-decreasing array

An array is said to be k-increasing-decreasing if elements repeatedly increase up to a certain index after which they decrease, then again increase, a total of k times. This is illustrated in Figure 10.2.

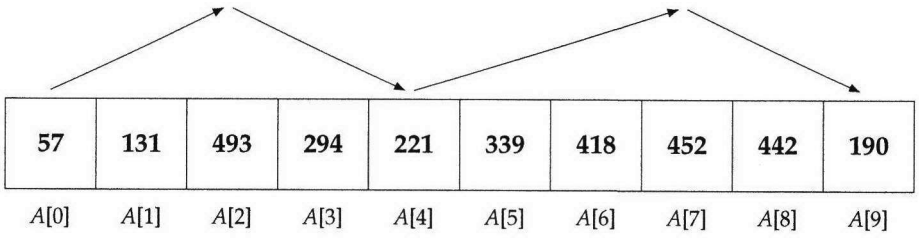


Figure 10.2: A 4-increasing-decreasing array.

Design an efficient algorithm for sorting a k-increasing-decreasing array.

Hint: Can you cast this in terms of combining k sorted arrays?

Solution: The brute-force approach is to sort the array, without taking advantage of the k-increasing-decreasing property. Sorting algorithms run in time $O(n \log n)$, where n is the length of the array.

If k is significantly smaller than n we can do better. For example, if $k = 2$, the input array consists of two subarrays, one increasing, the other decreasing. Reversing the second subarray yields two sorted arrays, and the result is their merge. It is fairly easy to merge two sorted arrays in $O(n)$ time.

Generalizing, we could first reverse the order of each of the decreasing subarrays. For the example in Figure 10.2 on the preceding page, we would decompose A into four sorted arrays—$\langle 57, 131, 493 \rangle$, $\langle 221, 294 \rangle$, $\langle 339, 418, 452 \rangle$, and $\langle 190, 442 \rangle$. Now we can use the techniques in Solution 10.1 on Page 134 to merge these.

```
def sort_k_increasing_decreasing_array(A):
    # Decomposes A into a set of sorted arrays.
    sorted_subarrays = []
    INCREASING, DECREASING = range(2)
    subarray_type = INCREASING
    start_idx = 0
    for i in range(1, len(A) + 1):
        if (i == len(A) or  # A is ended. Adds the last subarray.
            (A[i - 1] < A[i] and subarray_type == DECREASING) or
            (A[i - 1] >= A[i] and subarray_type == INCREASING)):
            sorted_subarrays.append(A[start_idx:i] if subarray_type ==
                                    INCREASING else A[i - 1:start_idx - 1:-1])
            start_idx = i
            subarray_type = (DECREASING
                             if subarray_type == INCREASING else INCREASING)
    return merge_sorted_arrays(sorted_subarrays)


# Pythonic solution, uses a stateful object to trace the monotonic subarrays.
def sort_k_increasing_decreasing_array_pythonic(A):
    class Monotonic:
        def __init__(self):
            self._last = float('-inf')

        def __call__(self, curr):
            res = curr < self._last
            self._last = curr
            return res

    return merge_sorted_arrays([
        list(group)[::-1 if is_decreasing else 1]
        for is_decreasing, group in itertools.groupby(A, Monotonic())
    ])
```

Just as in Solution 10.1 on Page 134, the time complexity is $O(n \log k)$ time.

10.3 Sort an almost-sorted array

Often data is almost-sorted—for example, a server receives timestamped stock quotes and earlier quotes may arrive slightly after later quotes because of differences in server loads and network routes. In this problem we address efficient ways to sort such data.

Write a program which takes as input a very long sequence of numbers and prints the numbers in sorted order. Each number is at most k away from its correctly sorted position. (Such an array is sometimes referred to as being k-sorted.) For example, no number in the sequence $\langle 3, -1, 2, 6, 4, 5, 8 \rangle$ is more than 2 away from its final sorted position.

Hint: How many numbers must you read after reading the ith number to be sure you can place it in the correct location?

Solution: The brute-force approach is to put the sequence in an array, sort it, and then print it. The time complexity is $O(n \log n)$, where n is the length of the input sequence. The space complexity is $O(n)$.

We can do better by taking advantage of the almost-sorted property. Specifically, after we have read $k+1$ numbers, the smallest number in that group must be smaller than all following numbers. For the given example, after we have read the first 3 numbers, $3, -1, 2$, the smallest, -1, must be globally the smallest. This is because the sequence was specified to have the property that every number is at most 2 away from its final sorted location and the smallest number is at index 0 in sorted order. After we read in the 4, the second smallest number must be the minimum of $3, 2, 4$, i.e., 2.

To solve this problem in the general setting, we need to store $k+1$ numbers and want to be able to efficiently extract the minimum number and add a new number. A min-heap is exactly what we need. We add the first k numbers to a min-heap. Now, we add additional numbers to the min-heap and extract the minimum from the heap. (When the numbers run out, we just perform the extraction.)

```
def sort_approximately_sorted_array(sequence, k):
    result = []
    min_heap = []
    # Adds the first k elements into min_heap. Stop if there are fewer than k
    # elements.
    for x in itertools.islice(sequence, k):
        heapq.heappush(min_heap, x)

    # For every new element, add it to min_heap and extract the smallest.
    for x in sequence:
        smallest = heapq.heappushpop(min_heap, x)
        result.append(smallest)

    # sequence is exhausted, iteratively extracts the remaining elements.
    while min_heap:
        smallest = heapq.heappop(min_heap)
        result.append(smallest)

    return result
```

The time complexity is $O(n \log k)$. The space complexity is $O(k)$.

10.4 Compute the k closest stars

Consider a coordinate system for the Milky Way, in which Earth is at $(0, 0, 0)$. Model stars as points, and assume distances are in light years. The Milky Way consists of approximately 10^{12} stars, and their coordinates are stored in a file.

How would you compute the k stars which are closest to Earth?

Hint: Suppose you know the k closest stars in the first n stars. If the $(n+1)$th star is to be added to the set of k closest stars, which element in that set should be evicted?

Solution: If RAM was not a limitation, we could read the data into an array, and compute the k smallest elements using sorting. Alternatively, we could use Solution 11.8 on Page 153 to find the kth smallest element, after which it is easy to find the k smallest elements. For both, the space complexity is $O(n)$, which, for the given dataset, cannot be stored in RAM.

Intuitively, we only care about stars close to Earth. Therefore, we can keep a set of candidates, and iteratively update the candidate set. The candidates are the k closest stars we have seen so far. When we examine a new star, we want to see if it should be added to the candidates. This entails comparing the candidate that is furthest from Earth with the new star. To find this candidate efficiently, we should store the candidates in a container that supports efficiently extracting the maximum and adding a new member.

A max-heap is perfect for this application. Conceptually, we start by adding the first k stars to the max-heap. As we process the stars, each time we encounter a new star that is closer to Earth than the star which is the furthest from Earth among the stars in the max-heap, we delete from the max-heap, and add the new one. Otherwise, we discard the new star and continue. We can simplify the code somewhat by simply adding each star to the max-heap, and discarding the maximum element from the max-heap once it contains $k+1$ elements.

```
class Star:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    @property
    def distance(self):
        return math.sqrt(self.x**2 + self.y**2 + self.z**2)

    def __lt__(self, rhs):
        return self.distance < rhs.distance



def find_closest_k_stars(stars, k):
    # max_heap to store the closest k stars seen so far.
    max_heap = []
    for star in stars:
        # Add each star to the max-heap. If the max-heap size exceeds k, remove
        # the maximum element from the max-heap.
        # As python has only min-heap, insert tuple (negative of distance, star)
        # to sort in reversed distance order.
        heapq.heappush(max_heap, (-star.distance, star))
        if len(max_heap) == k + 1:
            heapq.heappop(max_heap)

    # Iteratively extract from the max-heap, which yields the stars sorted
    # according from furthest to closest.
    return [s[1] for s in heapq.nlargest(k, max_heap)]
```

The time complexity is $O(n \log k)$ and the space complexity is $O(k)$.

Variant: Design an $O(n \log k)$ time algorithm that reads a sequence of n elements and for each element, starting from the kth element, prints the kth largest element read up to that point. The

length of the sequence is not known in advance. Your algorithm cannot use more than $O(k)$ additional storage. What are the worst-case inputs for your algorithm?

10.5 Compute the median of online data

You want to compute the running median of a sequence of numbers. The sequence is presented to you in a streaming fashion—you cannot back up to read an earlier value, and you need to output the median after reading in each new element. For example, if the input is $1, 0, 3, 5, 2, 0, 1$ the output is $1, 0.5, 1, 2, 2, 1.5, 1$.

Design an algorithm for computing the running median of a sequence.

Hint: Avoid looking at all values each time you read a new value.

Solution: The brute-force approach is to store all the elements seen so far in an array and compute the median using, for example Solution 11.8 on Page 153 for finding the kth smallest entry in an array. This has time complexity $O(n^2)$ for computing the running median for the first n elements.

The shortcoming of the brute-force approach is that it is not incremental, i.e., it does not take advantage of the previous computation. Note that the median of a collection divides the collection into two equal parts. When a new element is added to the collection, the parts can change by at most one element, and the element to be moved is the largest of the smaller half or the smallest of the larger half.

We can use two heaps, a max-heap for the smaller half and a min-heap for the larger half. We will keep these heaps balanced in size. The max-heap has the property that we can efficiently extract the largest element in the smaller part; the min-heap is similar.

For example, let the input values be $1, 0, 3, 5, 2, 0, 1$. Let L and H be the contents of the min-heap and the max-heap, respectively. Here is how they progress:

(1.) Read in 1: $L = [1], H = []$, median is 1.
(2.) Read in 0: $L = [1], H = [0]$, median is $(1 + 0)/2 = 0.5$.
(3.) Read in 3: $L = [1, 3], H = [0]$, median is 1.
(4.) Read in 5: $L = [3, 5], H = [1, 0]$, median is $(3 + 1)/2 = 2$.
(5.) Read in 2: $L = [2, 3, 5], H = [1, 0]$, median is 2.
(6.) Read in 0: $L = [2, 3, 5], H = [1, 0, 0]$, median is $(2 + 1)/2 = 1.5$.
(7.) Read in 1: $L = [1, 2, 3, 5], H = [1, 0, 0]$, median is 1.

```
def online_median(sequence):
    # min_heap stores the larger half seen so far.
    min_heap = []
    # max_heap stores the smaller half seen so far.
    # values in max_heap are negative
    max_heap = []
    result = []

    for x in sequence:
        heapq.heappush(max_heap, -heapq.heappushpop(min_heap, x))
        # Ensure min_heap and max_heap have equal number of elements if an even
        # number of elements is read; otherwise, min_heap must have one more
        # element than max_heap.
        if len(max_heap) > len(min_heap):
            heapq.heappush(min_heap, -heapq.heappop(max_heap))

        result.append(0.5 * (min_heap[0] + (-max_heap[0])))
```

```
                    if len(min_heap) == len(max_heap) else min_heap[0])
    return result
```

The time complexity per entry is $O(\log n)$, corresponding to insertion and extraction from a heap.

10.6 Compute the k largest elements in a max-heap

A heap contains limited information about the ordering of elements, so unlike a sorted array or a balanced BST, naive algorithms for computing the k largest elements have a time complexity that depends linearly on the number of elements in the collection.

Given a max-heap, represented as an array A, design an algorithm that computes the k largest elements stored in the max-heap. You cannot modify the heap. For example, if the heap is the one shown in Figure 10.1(a) on Page 132, then the array representation is $\langle 561, 314, 401, 28, 156, 359, 271, 11, 3\rangle$, the four largest elements are 561, 314, 401, and 359.

Solution: The brute-force algorithm is to perform k extract-max operations. The time complexity is $O(k \log n)$, where n is the number of elements in the heap. Note that this algorithm entails modifying the heap.

Another approach is to use an algorithm for finding the kth smallest element in an array, such as the one described in Solution 11.8 on Page 153. That has time complexity almost certain $O(n)$, and it too modifies the heap.

The following algorithm is based on the insight that the heap has partial order information, specifically, a parent node always stores value greater than or equal to the values stored at its children. Therefore, the root, which is stored in $A[0]$, must be one of the k largest elements—in fact, it is the largest element. The second largest element must be the larger of the root's children, which are $A[1]$ and $A[2]$—this is the index we continue processing from.

The ideal data structure for tracking the index to process next is a data structure which support fast insertions, and fast extract-max, i.e., in a max-heap. So our algorithm is to create a max-heap of candidates, initialized to hold the index 0, which serves as a reference to $A[0]$. The indices in the max-heap are ordered according to corresponding value in A. We then iteratively perform k extract-max operations from the max-heap. Each extraction of an index i is followed by inserting the indices of i's left child, $2i + 1$, and right child, $2i + 2$, to the max-heap, assuming these children exist.

```
def k_largest_in_binary_heap(A, k):
    if k <= 0:
        return []

    # Stores the (-value, index)-pair in candidate_max_heap. This heap is
    # ordered by value field. Uses the negative of value to get the effect of
    # a max heap.
    candidate_max_heap = []
    # The largest element in A is at index 0.
    candidate_max_heap.append((-A[0], 0))
    result = []
    for _ in range(k):
        candidate_idx = candidate_max_heap[0][1]
        result.append(-heapq.heappop(candidate_max_heap)[0])

        left_child_idx = 2 * candidate_idx + 1
        if left_child_idx < len(A):
```

```
            heapq.heappush(candidate_max_heap, (-A[left_child_idx],
                                                left_child_idx))
        right_child_idx = 2 * candidate_idx + 2
        if right_child_idx < len(A):
            heapq.heappush(candidate_max_heap, (-A[right_child_idx],
                                                right_child_idx))
    return result
```

The total number of insertion and extract-max operations is $O(k)$, yielding an $O(k \log k)$ time complexity, and an $O(k)$ additional space complexity. This algorithm does not modify the original heap.