5. Priority Queues

Anyone who has ever waited impatiently in line, for example at the ticket counter of a crowded train station, is all too familiar with the notion of a queue. Standard queues exhibit First-In-First-Out (FIFO) behavior, where those entering the queue first are also the first to leave. There are some situations, however, where FIFO behavior is not desired. For instance, in the emergency room of a hospital, patients waiting for treatment are typically considered in order according to the urgency of their condition. This is known as a *priority queue*, where elements enter the queue with associated priorities, and the element with the highest priority is always the first to leave. The priority queue is one of the most fundamental types of data structures, one you are likely to encounter often in practice. In this chapter we discuss several ways to implement a priority queue, focusing in particular on those that are the most popular, the most efficient, and the most illustrative of elegant principles in data structure design.

To be considered a priority queue, a data structure must at the very least support two basic operations:

- *Insert*(e, k). Insert a new element e with key k.
- *Remove-Min*. Remove an element with minimum key from the priority queue, and return this element.

Every element stored in a priority queue has an associated *key*, and by convention elements with lower keys are usually taken to have higher priority, although if desired, it is just as easy to adopt the opposite convention. Several other operations are commonly supported by priority queues:

- *Build*$(e_1 \ldots e_n,\ k_1 \ldots k_n)$. Constructs a priority queue in a batch setting on n elements $e_1 \ldots e_n$ and their corresponding keys $k_1 \ldots k_n$.
- *Find-Min*. Return a pointer to, but do not remove, an element with minimum key.
- *Decrease-Key*$(e, \Delta k)$. Decrease the key of element e by Δk.
- *Increase-Key*$(e, \Delta k)$. Increase the key of element e by Δk.

- *Delete*(e). Remove a specified element e from the priority queue.

Due to redundancies among these operations, we may not need to implement all of them:

- We can implement *remove-min* using *find-min* followed by *delete*,
- We can implement *find-min* by calling *remove-min* followed by *insert* to put the element that was removed back into the priority queue,
- We can implement both *decrease-key* and *increase-key* using *delete* and *insert* by removing an element and re-inserting it with a new key, and
- We can implement *delete* by calling *decrease-key* to lower the key of an element so it becomes the minimum, then by calling *remove-min*.

Often, a priority queue maintains only *pointers* to data records, rather than the records themselves. This can improve efficiency, especially if our data records are large, since we need to shuffle less memory around. If our data records are already stored in some other data structure (as is often the case), we can view our priority queue as a lightweight indexing structure built "on top of" this existing structure, giving it the added functionality of a priority queue. Most important, however, is that not unlike your digestive system, once an element is inserted into a priority queue, it is generally not easy to access until it re-emerges from the other side, in response to a call to *remove-min*. If we need to access elements from outside the priority queue (say, if we need to call *decrease-key* or *delete*), we need to maintain pointers to them from the outside, since *priority queues do not support an efficient means of finding elements* (by contrast, *dictionary* data structures, discussed extensively in the next two chapters, are specifically designed to efficiently find elements based on their keys). Therefore, if e is an element stored inside a priority queue representing data record d stored outside the structure, we typically maintain a pointer from d to e so that we can still access e if needed. Having said all of this, for simplicity we henceforth ignore this issue and pretend that we are storing just a set of keys in our priority queue.

5.1 Priority Queues and Sorting

Priority queues and sorting share much in common. We can easily sort n elements by building a priority queue on them and calling *remove-min* n times, causing the elements to exit the queue in sorted order. We will shortly see a very nice sorting algorithm, *heap sort*, based on this technique.

Recall that we divided sorting algorithms into two main categories: input-insensitive algorithms that work in the comparison-based or real RAM models of computation, and input-sensitive algorithms that sort integers in the RAM model. Priority queues can be similarly classified, and we cover both types in this chapter, starting with comparison-based structures. In this setting, the $\Omega(n \log n)$ worst-case lower bound for comparison-based sorting implies that *insert* or *remove-min* must run in $\Omega(\log n)$ worst-case time, or else we could sort in faster than $O(n \log n)$ time using n calls to *insert* followed by n calls to *remove-min*.

	Unsorted Array	Sorted Array	Binary Heap
Insert	$O(1)$	$O(n)$	$O(\log n)$
Remove-Min	$O(n)$	$O(1)$	$O(\log n)$
Find-Min	$O(n)$	$O(1)$	$O(1)$
Decrease-Key	$O(1)$	$O(n)$	$O(\log n)$
Increase-Key	$O(1)$	$O(n)$	$O(\log n)$
Delete	$O(1)$	$O(n)$	$O(\log n)$
Build	$\Theta(n)$	$O(n \log n)$	$\Theta(n)$

FIGURE 5.1: Running times for the operations of several basic priority queue data structures. If we are using a sorted linked list rather than a sorted array, then *delete* takes $O(1)$ rather than $O(n)$ time.

A priority queue is *stable* if it acts like a FIFO queue for equal elements, causing them to exit in the same order they enter. When used to sort, stable priority queues give us stable sorting algorithms. We can make any priority queue stable by augmenting elements with *sequence numbers* giving the time they were inserted, using these to break ties among equal elements.

5.2 Unsorted and Sorted Arrays

One can implement a simple priority queue using nothing more than an array, possibly maintained in sorted order. The results are not terribly efficient, but they serve as a good baseline for our initial discussion:

The Unsorted Array. By storing elements in an unsorted array, we can easily *insert* in $O(1)$ time by appending to the end of the array. However, *remove-min* takes $\Theta(n)$ worst-case time since we need to scan the array the find the minimum element. Once we have found the minimum element, we can *delete* it (or any other element) in $O(1)$ time by swapping it with the final element in the array and decreasing the array size — this avoids the problem of leaving a "gap" in the array. *Decrease-key* and *increase-key* both obviously run in $O(1)$ time.

The Sorted Array. By maintaining elements in (say, decreasing) sorted order, we achieve the opposite tradeoff. The minimum will always be the final element, so *remove-min* takes only $O(1)$ time. However, now *insert*, *delete*, *decrease-key*, and *increase-key* all take $\Theta(n)$ worst-case time, in order to keep the array sorted.

The results above are summarized in Figure 5.1. Linked lists give us essentially the same performance guarantees, except they make deletion slightly easier in the sorted case. In both cases, however, there is a dramatic tradeoff between the fundamental operations *insert* and *remove-min*, with one always running in $\Theta(n)$ worst-case time. Our next data structure, the *binary heap*, provides some middle ground by supporting all operations in $O(\log n)$ time.

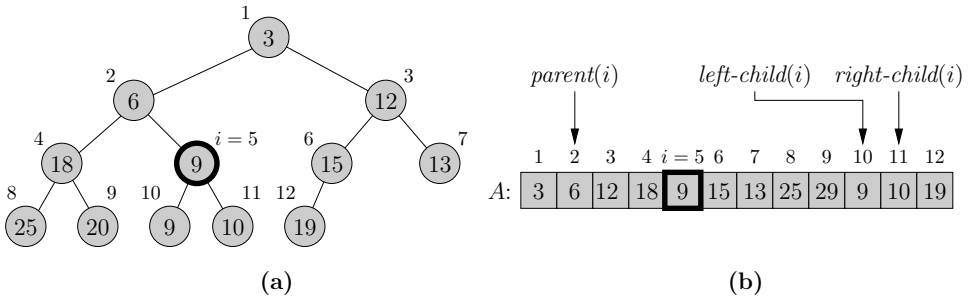

FIGURE 5.2: A binary heap (a) depicted as a tree and (b) stored as an array.

5.3 Binary Heaps

Binary heaps are probably the most well-known and commonly-used priority queue data structures, owing to their simplicity, speed, and ease of implementation. Since these are such common objects in computer science, you will often hear them simply called "heaps". However, the term *heap* refers more generally to a family of tree data structures that satisfy the *heap property*: $key(parent(e)) \leq key(e)$ for every element e. As shown in Figure 5.2(a), the heap property imposes a "vertical" ordering on a tree, forcing the smallest element to reside at the root. Since the minimum element is in such an obvious place, heap-ordered trees are common in many priority queue implementations.

5.3.1 Storing a Binary Heap in an Array

An n-element binary heap is an array $A[1 \ldots n]$ that we mentally visualize as a heap-ordered binary tree, as shown in Figure 5.2. The tree has a special shape, being *almost complete*, with every level completely filled in except the last, which is filled from left to right up to some point. An almost complete tree can be mapped to our array A in a natural fashion: the root corresponds to $A[1]$, the two children of the root to $A[2]$ and $A[3]$, the four grandchildren of the root to $A[4 \ldots 7]$, and so on. Moreover, we can easily jump around in the array A to mimic movement in the tree. For any element $A[i]$ in the array, its parent in the tree has index $\lfloor i/2 \rfloor$ and its two children are located at indices $2i$ and $2i + 1$ (for those accustomed to zero-based indexing of arrays, you will need to change these formulas slightly).

The fact that we can visualize, navigate, and manipulate the structure like a tree while storing it in a nothing more than a single lightweight array is one of the main advantages of the binary heap. By contrast, most other tree-based data structures are somewhat more cumbersome to implement, since each node is stored in an individually-allocated block of memory, and each node maintains pointers to its parent and children.

Note that the array representing a heap is not sorted or even nearly sorted (e.g., it can have $\Theta(n^2)$ inversions), even though the heap property stipulates that small elements generally precede larger elements, with $A[1]$ being the smallest overall.

5.3.2 Operations on a Binary Heap

It is straightforward to implement all standard priority queue operations on a binary heap after first building two fundamental operations called *sift-up* and *sift-down*. These operations are common to all heaps, not just binary heaps, and are designed to move around elements in the heap so as to enforce or restore the heap property. Their names give us some intuition behind their functionality: *sift-up* causes small "light" elements to drift upward toward the top of the heap, and *sift-down* forces large "heavy" elements to sink down towards the bottom.

Sift-up(i) takes $A[i]$ and pushes it up the tree as far as possible by repeatedly exchanging it with its parent, as long as the parent element has a larger key. Calling *sift-up*(i) will repair a single violation of the heap property between $A[i]$ and its parent. Similarly, *sift-down*(i) will repair a violation of the heap property between $A[i]$ and its children, by pulling $A[i]$ as far down in the heap as possible by repeatedly swapping it with its smallest child (as long as this child has a smaller key). Both *sift-up* and *sift-down* run in $O(\log n)$ time on a binary heap, since they both require at worst time proportional to the height of the tree, and an almost-complete binary tree on n elements has height $O(\log n)$.

The remaining priority queue operations are now very simple to implement using *sift-up* and *sift-down*:

- To *insert* a new element, we increment the size of the heap, n, place the new element in $A[n]$ (corresponding to the next open slot in the bottom row of the heap), and call *sift-up* on it, to fix any potential violation of the heap property with its parent.
- When we perform a *decrease-key* operation on $A[i]$, this might also potentially break the heap property between $A[i]$ and its parent, so we call *sift-up*(i).
- Similarly, *increase-key* calls *sift-down* on an element after increasing its key, since we may need to correct a potential violation of the heap property with its children.
- *Remove-min* swaps $A[1]$ and $A[n]$ and decrements n, so the element we want to remove is deposited at the end of our array, now one position beyond the end of the heap. We are left with an element at the root (formerly $A[n]$) that might now violate the heap property with its children, so we call *sift-down* on the root.
- *Delete* is similar to *remove-min*. We delete $A[i]$ by swapping it with $A[n]$ and decrementing n, then calling both *sift-up*(i) and *sift-down*(i) to correct any potential heap violations created by substituting an arbitrary element (formerly $A[n]$) in place of $A[i]$.

All of these operations call either *sift-up* or *sift-down* or both, so they all run in $O(\log n)$ time. [Animated explanation of heap operations]

Building a Binary Heap in Linear Time. To build a binary heap on n elements, we could start with an empty heap and make n calls to *insert*, although this takes $\Theta(n \log n)$ time in the worst case where we insert elements in decreasing order, with

every element being pulled all the way up to the root by *sift-up*. Surprisingly, there is an even better way to build a binary heap in only $\Theta(n)$ time: starting with our n elements in an unordered array $A[1 \ldots n]$, we simply call *sift-down*(i) on each element i from n down to 1 in sequence. It is not immediately obvious why this approach would build a valid binary heap or that it runs in $\Theta(n)$ time, but we can show both using a bit of careful analysis. [Careful analysis]

Note that the approach above builds a binary heap *in place*, converting an arbitrary array into a heap-ordered array. Also note that the fast $\Theta(n)$ running time for *build* does not violate the $\Omega(n \log n)$ lower bound for comparison-based sorting, since it still takes $O(n \log n)$ time to remove the elements from a binary heap in sorted order.

Problem 78 (Random Insertion in Binary Heaps). Please show that it takes only $\Theta(n)$ time in expectation to insert n elements into a binary heap in *random* order (compared to the $\Theta(n \log n)$ worst-case running time we get from inserting in decreasing order). As a hint, try to apply the result of problem 18(b) to various locations in the structure. [Solution]

Problem 79 (Heaps of Heaps). In this problem, we reduce the running time of *insert* to $O(1)$ amortized time[1]. Suppose we are "lazy" and insert elements in $O(1)$ time into a temporary array T rather than into our priority queue. When *remove-min* is called, we build a new binary heap out of T in linear time. As a result, we end up maintaining a large number of binary heaps, one representing each run of *insert* operations between successive calls to *remove-min*. How can we tie these together into one structure supporting *insert* in $O(1)$ amortized time and *remove-min* in $O(\log n)$ amortized time? [Solution]

Problem 80 (B-Heaps). A natural generalization of the binary heap is the B-heap, which is structured as an almost-complete B-ary tree rather than an almost-complete binary tree, so every node in the tree has B children rather than 2 children. It is easy to generalize the operations of a binary heap to work on a B-heap, as well as the "bottom up" method for building the structure in $\Theta(n)$ time. Please comment on what effect, if any, this generalization will have on the running time of *sift-up*, *sift-down*, and the fundamental priority queue operations. Next, suppose we have an application that is likely to perform k times as many invocations of *decrease-key* as *remove-min* (for example, Dijkstra's shortest path algorithm calls *decrease-key* m times and *remove-min* n times on a graph with n nodes and m edges). What is the best choice for B in this case? [Solution]

5.3.3 Heap Sort

We have already described how to sort using a generic priority queue: first *build* the queue and then perform n consecutive *remove-min* operations. In a binary heap, this leads to a particularly nice result: after building a binary heap out of an unsorted array $A[1 \ldots n]$, the array actually ends up in decreasing sorted order as a "side effect" of calling *remove-min* n times. The first *remove-min* swaps $A[1]$ (the minimum) with $A[n]$ and then decrements the size of the heap, leaving the minimum at the end of the array. The next *remove-min* swaps $A[1]$ (the second-smallest element) with $A[n-1]$, and so on. Of course, if we wanted our array to end up in forward sorted order, we could have used a "max" binary heap, from which we always remove the maximum instead of the minimum.

[1] Achieving $O(1)$ amortized time for *insert*, by itself, is actually not usually a big win in terms of total running time, since every element is usually the victim of a *remove-min* call down the road, so it still incurs $O(\log n)$ work during its lifetime in the structure.

This algorithm, called *heap sort*, is easy to implement, takes $O(n \log n)$ time, sorts in place, and runs very quickly in practice. Heap sort is not stable since the binary heap is not a stable heap, but if in-place operation and a deterministic $O(n \log n)$ running time are important, heap sort may well be the sorting algorithm of choice. Its closest competitor with these features, the deterministic $O(n \log n)$ variant of quicksort, runs much slower in practice.

5.4 Mergeable Priority Queues

In order to develop more efficient data structures than the binary heap, we need to enter the realm of the mergeable priority queues (sometimes called meldable priority queues). In addition to the standard operations from before, a mergeable priority queue supports the new operation *merge* (sometimes called *meld*), taking two priority queues and merging them into a single priority queue, after which they no longer exist as separate individual structures.

In this section, we discuss two main classes of mergeable priority queues. The first class consists of nearly half a dozen highly-related data structures (randomized mergeable binary heaps, leftist heaps, skew heaps, and several others) that are built from a single heap-ordered tree. The second class contains priority queues that are built from multiple heap-ordered trees of different sizes and shapes, including the binomial heap and its more sophisticated relative, the Fibonacci heap.

5.4.1 Randomized Mergeable Binary Heaps

While the binary heap itself does not support an efficient *merge* operation, there are many close relatives that do. The most elegant of the bunch, in the author's opinion, is the *randomized mergeable binary heap*, which performs all priority queue operations, including *merge*, in $O(\log n)$ time with high probability.

The randomized mergeable binary heap is nothing more than a heap-ordered binary tree. It can be have arbitrary shape and may be quite unbalanced[2], so instead of storing it in a simple array like the binary heap, we store it like most other trees. Each element resides in its own block of memory, and maintains a pointer to its parent, left child, and right child. Since the tree height may be significantly larger than $O(\log n)$, the use of *sift-up* and *sift-down* is no longer prudent. Fortunately, we can now abandon *sift-up* and *sift-down* entirely and implement every priority queue operation in a very simple fashion using *merge*:

- To *insert* a new element into a heap H, we construct a new single-element heap and merge it with H.
- We implement *remove-min* by removing the root element and merging its two former child subtrees (which are themselves valid heap-ordered trees).

[2]In the next chapter, we will spent quite a bit of effort trying to keep binary trees balanced for performance reasons. By contrast, the heap-ordered trees in this section can be quite unbalanced with no negative consequences. Even a degenerate tree in the shape of a long n-element sorted path would be acceptable.

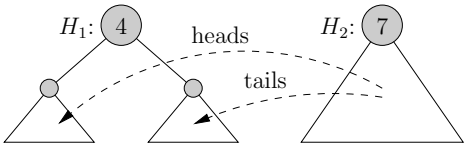

FIGURE 5.3: Recursively merging heap H_2 (larger root) with a random child of heap H_1 (smaller root).

- To decrease the key of an element e, we splice out the subtree rooted at e, decrease e's key, then merge this subtree back into the original tree. Since e is a root element when its key is reduced, there is no danger of violating the heap property.
- Deleting element e is accomplished by replacing e with the result of merging its two child subtrees.
- We can implement *increase-key* in terms of *delete* and *insert*.

Since all of these operations involve a constant number of calls to *merge*, they all run in $O(\log n)$ time with high probability as long as the same is true for *merge*.

Randomization gives us an extremely simple approach for merging two heaps H_1 and H_2. Assuming H_1 has the smaller root, its root becomes the root of the merged tree, and we merge H_2 recursively into one of its child subtrees. As shown in Figure 5.3, if a fair coin toss results in heads, we recursively merge H_2 with the left subtree of H_1. If we see tails, we recursively merge H_2 with the right subtree of H_1. The process ends when we reach the bottom of one of the trees and try to merge some heap H_1 with an empty heap H_2, in which case the result is just H_1. The fact that this runs in $O(\log n)$ time with high probability follows immediately from the randomized reduction lemma: with probability $1/2$, we choose to merge with the smaller of H_1's two subtrees, in which case we effectively reduce the size of H_1 to at most half of its current value.

5.4.2 Leftist Heaps, Skew Heaps, and Other Relatives Based on Null Path Merging

Another natural way to merge two heap-ordered trees H_1 and H_2 is as follows: select a path in H_1 from its root down to an "empty space" at the bottom of the tree (known as a *null path*), select a similar path in H_2, and merge along these paths, as shown in Figure 5.4. Due to the heap property, null paths are sorted from top to bottom, and consequently the process of merging along these paths is completely analogous the familiar process of merging two sorted sequences.

One way to merge two sorted sequences s_1 and s_2 (say, with s_1 having the smaller initial element) is by taking the initial element of s_1 followed by the result of recursively merging the rest of s_1 with s_2. Similarly, we merge heaps H_1 and H_2 along null paths (say, with H_1 having the smaller root), by recursively merging H_2 with either the left or right subtree of H_1, depending on the direction of the null path in

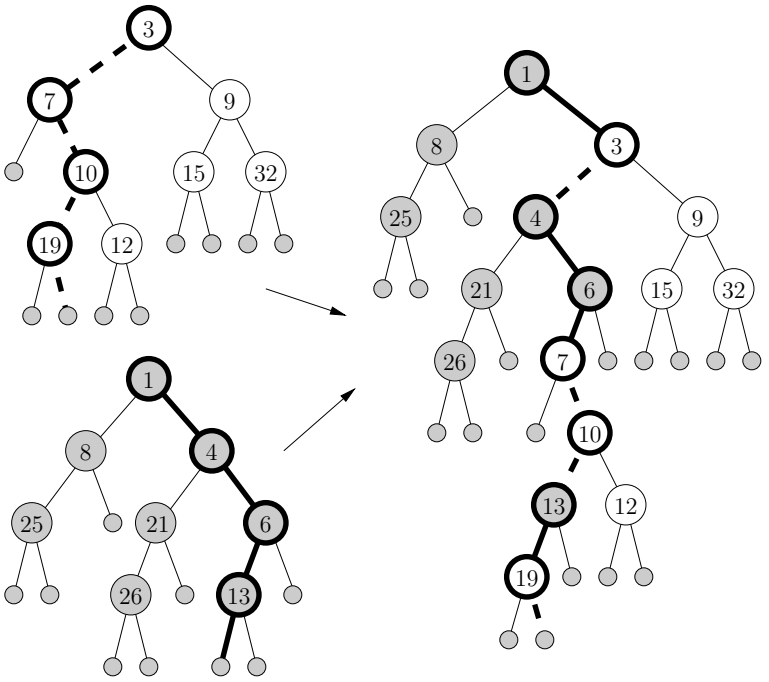

FIGURE 5.4: Merging two heap-ordered trees along a null path in each tree. Rather than using "null" pointers to indicate the lack of a child at the bottom of a tree, we have marked the bottom of each tree using dummy "sentinel" elements.

H_1. If you prefer the iterative outlook on merging two sequences, a similar process works for heaps, where we step two pointers in tandem down our null paths, always taking a step from the pointer to the smaller element, splicing the heaps together as we go. Both the recursive and iterative approaches are completely equivalent. As in the case of merging two sequences, they are just two different ways of looking at the same process.

Since the amount of time required to merge two heap-ordered trees along null paths is proportional to the combined heights of these paths, we clearly want to select two paths of low height in order to merge quickly. The randomized mergeable binary heap does this in perhaps the simplest possible fashion by choosing paths at random — moving left or right at each step according to the result of a fair coin flip. This gives null paths of length $O(\log n)$ with high probability, thanks to our earlier analysis with the randomized reduction lemma. There are several other common alternatives for choosing good null paths, however, which we outline below.

Size-Augmented and Null-Path-Length Augmented Heaps. We could "de-randomize" the randomized mergeable binary heap by augmenting each element in our heap with a count of the number of elements in its subtree. If we then choose a path by repeatedly stepping to whichever of our child subtrees has a smaller size, this gives a deterministic approach for finding a path of height at most $\log n$, since every step downward at least halves the size of our current subtree. Similarly, we

could augment each element e with its *null path length*, $npl(e)$, which gives the length of the shortest null path downward from e. By repeatedly stepping to whichever child has a smaller value, this also gives a deterministic approach for finding a null path of height at most $\log n$.

Leftist Heaps. The *leftist heap* is a heap where we augment each element with its null path length and also maintain the invariant that *npl(left-child(e))* $\geq$ *npl(right-child(e))* for every element e. This results in a tree that is somewhat "left heavy" in which the best way to find a path for merging is to follow the *right spine* of the tree, which has height at most $\log n$. We therefore always merge two trees by merging along their right spines, and then walk back up the right spine of the merged tree swapping children anywhere necessary to restore our invariant. This approach is essentially the same as our approach above where we augment each element with its null path length, except we always treat the child with smaller null path length as the right child.

Skew Heaps. The *skew heap* is a relaxed, amortized cousin of the leftist heap that is simpler to implement (perhaps even almost as simple as the randomized mergeable binary heap) but slightly more intriguing to analyze. Like the leftist heap, we merge two heaps by merging their right spines and then adjusting the structure of the merged tree slightly. The readjustment step, however, is now much easier: we just walk up the right spine of the merged heap and swap the two children of every element along the way (except the lowest). As one can prove, this results in the tree being sufficiently "leftist" that *merge* (and hence every other priority queue operation) runs in $O(\log n)$ amortized time. [Detailed analysis of skew heaps]

An alternative view of the skew heap that perhaps illustrates its operation more clearly is the following: let us imagine that every element is augmented with a coin whose state is either heads or tails. To merge two heaps H_1 and H_2 (with H_1 having the smaller root), we look at the coin at the root of H_1; if heads, we merge H_2 recursively with the left subtree of H_1, and if tails, we merge with the right subtree. We then flip over the coin on the root of H_1, toggling its state. Intuitively, this would seem to keep H_1 somewhat "balanced", by alternatively merging into its left and right subtrees. Indeed, if you think carefully about the operation of this structure, you will see that it behaves exactly the same as the skew heap as described above; we have only described it in a top-down recursive manner[3]. This top-down description also highlights the similarity between the randomized mergeable binary heap and the skew heap, with the only difference being that where the randomized mergeable binary heap flips a coin, the skew heap flips the coin *over*.

Skew heaps and their distant relatives splay trees (Section 6.2.7) are known as *self-adjusting* data structures since they can (somewhat miraculously) continually adjust their structure so as to remain efficient despite maintaining no additional augmented information to help them in this task. In fact, among the five different "null path" mergeable heap implementations we have seen above, the randomized mergeable binary heap and skew heap stand out in that they do not require us to keep any augmented state in our heap. The other approaches require augmenting elements with either null path lengths or subtree sizes, and while this extra informa-

[3]To make the correspondence even more direct, we can always recursively merge with the right subtree, and then swap the left and right subtrees. This gives the same mechanics of merging with alternating subtrees, but avoids the need to explicitly store the state of the coin.

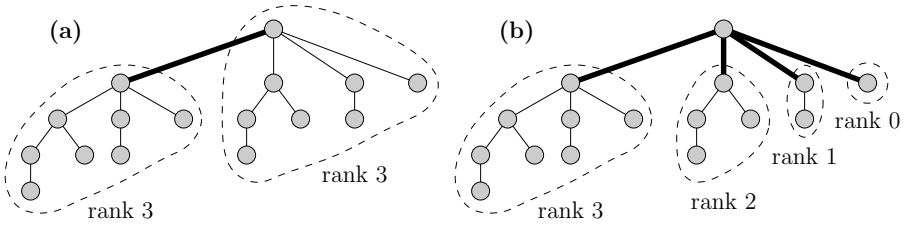

FIGURE 5.5: Recursive construction a rank-4 binomial tree by (a) linking two rank-3 binomial trees, or (b) joining rank-0 through rank-3 trees as siblings under a new common root.

tion can be easily updated with no running time penalty during the merge process, it does cause trouble for the operations *delete*, *decrease-key*, and *increase-key*, as we discover in the following problem.

Problem 81 (Lazy Deletion). Consider the three approaches we discussed above: size-augmented heaps, null-path-augmented heaps, and leftist heaps. In each of these, *delete*, *decrease-key*, and *increase-key* are problematic due to the need to maintain augmented information. Why is this, and how can we fix this issue by being somewhat "lazy" so that all priority queue operations run in $O(\log n)$ amortized time? [Solution]

5.4.3 Binomial Heaps

In this section we describe the *binomial heap*, an elegant priority queue data structure that performs all fundamental operations including *merge* in $O(\log n)$ time in the worst case. Using the binomial heap as a stepping stone, we then describe the *Fibonacci heap*, a more sophisticated relative designed to speed up *decrease-key* to run in $O(1)$ amortized time.

In all of the tree-based priority queues we have seen so far, elements are stored in a single heap-ordered tree. By contrast, the elements in a binomial heap are divided up into a collection of heap-ordered trees. Each element may potentially have many children, stored in a doubly-linked list. Therefore, along with each element we store its *rank* (number of children), a pointer to its parent, a pointer to its first child, and pointers to its previous and next siblings.

The trees in a binomial heap come in specific shapes, known as *binomial trees*, that are built in a recursive fashion, as shown in Figure 5.5. There is a unique shape associated with every rank, where the rank of a tree is the number of children of its root element. A rank-0 tree consists of a single element. More complicated trees are recursively constructed: we either link two trees of rank $j-1$ (one as a child of the other) to obtain a rank j tree, or we join trees of ranks $0 \ldots j-1$ under a new root to obtain a tree of rank j. It is not hard to see from this construction that a tree of rank j has depth j and contains 2^j elements, 2^{j-1} of which are leaves. The name "binomial" tree comes from the fact that number of elements at depth d in a rank-j tree is exactly the binomial coefficient $\binom{j}{d}$.

Binomial trees arise in a number of interesting algorithmic situations. For example,

suppose you broadcast a message from a single source node to all other nodes in a network in a number of rounds, where in each round, every node that has heard the message so far transmits the message to a distinct node that has not yet heard it. The resulting transmission pattern will have the form of a binomial tree. In Chapter 8, we will also study a close relative of the binomial tree known as the binary indexed tree.

As shown in Figure 5.6, a binomial heap is built from a collection of binomial trees, at most one of each rank, whose roots are all connected together in a doubly-linked list. It is clear that there can be at most $\log n$ such trees represented in the list, since a tree of rank larger than $\log n$ would contain more than $2^{\log n} = n$ elements. Since all of the trees in the binomial heap satisfy the heap property, the minimum element must reside at one of their root elements, so we can find it in $O(\log n)$ time by scanning the root list.

Owing to the fact that binomial trees have sizes that are powers of two, there is a only one unique configuration of trees that represents a valid binomial heap on n elements, corresponding precisely to the binary representation of n. As shown in Figure 5.6, if we form a binary number in which the j^{th} bit indicates the presence or absence of a tree of rank j, this gives the binary representation of n.

We will describe how to merge two binomial heaps in $O(\log n)$ time in a moment, but let us first see how easy it is to write the remaining fundamental operations in terms of *merge*. To *insert* an element in $O(\log n)$ time, we build a new 1-element binomial heap and merge it with our existing heap. *Remove-min* is also simple to express in $O(\log n)$ time in terms of *merge*. Observe from Figure 5.6 that if we remove the root corresponding to the minimum element, the linked list of child subtrees of this root is itself a valid binomial heap. Therefore, to remove the minimum element, we splice out the root containing the minimum element and merge the binomial heap consisting of its children back into the main heap. *Decrease-key* works the same as in a regular binary heap, by using *sift-up*. Notice that *sift-up* doesn't change the shape of any of our trees, and it runs in $O(\log n)$ time since every element belongs to a tree of height $O(\log n)$. We can write *delete* in terms of *decrease-key* followed by *remove-min*, and we can write *increase-key* by deleting an element and then re-inserting it with a new key. Note that we could implement *increase-key* using *sift-down*, but this would be somewhat slower ($O(\log^2 n)$ time) since each node has up to $\log n$ children; recall from problem 80 that as opposed to *sift-up*, *sift-down* becomes slower when nodes have more children.

All that remains is to show how to merge two binomial heaps H_1 and H_2 in $O(\log n)$ time. If H_1 and H_2 have n_1 and n_2 elements, the merge process corresponds exactly to the binary addition of n_1 and n_2. This makes intuitive sense, because the root lists in H_1 and H_2 reflect the binary composition of n_1 and n_2, and we are producing a merged heap whose root list needs to reflect the binary composition of $n_1 + n_2$. When adding two binary numbers, we add each bit position starting from the least significant; when we find several 1 bits in position j, we add two of them to form a carry bit that is added to position $j + 1$. Translated to our merge operation, we splice together the two root lists, starting from the smallest rank, and whenever we find more than one tree of rank j, we link them together in $O(1)$ time to form a "carry" tree of rank $j + 1$. [Detailed explanation of the merge operation]

If we insert successive elements into a binomial heap, the resulting merge operations

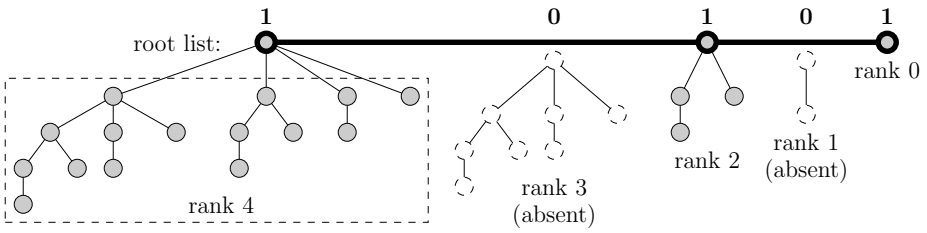

FIGURE 5.6: Illustration of a binomial heap: a linked list of binomial trees, at most one of each rank. Here we are storing $n = 21 = 10101_2$ elements, and since $21 = 2^4 + 2^2 + 2^0$, we need to use a tree of rank 4, a tree of rank 2, and a tree of rank 0. The dashed rectangle illustrates that the children of a root node form a valid binomial heap by themselves.

are identical in structure to the process of incrementing a binary counter, which takes only $O(1)$ amortized time (problem 75). Therefore, successive insertions into a binomial heap take only $O(1)$ amortized time each, so we can build a binomial heap in $\Theta(n)$ time using n calls to *insert*.

5.4.4 Fibonacci Heaps

The *Fibonacci heap* cleverly extends the binomial heap to support *decrease-key* in $O(1)$ amortized time, which yields improvements in efficiency for several important algorithms, such as Dijkstra's shortest path algorithm and Jarník's (Prim's) minimum spanning tree algorithm. However, these gains tend to be more theoretical than practical, as they rarely outweigh the added overhead of the Fibonacci heap (for example with Dijsktra's shortest path algorithm, the author has not yet found even a single real-world scenario where Fibonacci heaps are advantageous).

Starting from a binomial heap, two key enhancements appear in the Fibonacci heap:

- **"Lazy" Maintenance of the Root List.** In the binomial heap we are careful to maintain at most one tree of each rank. By contrast, the Fibonacci heap allows multiple trees of the same rank in the root list. This actually simplifies many operations; for example, we can *insert* a new element in $O(1)$ time by linking a new rank-0 tree into the root list, and we can *merge* two heaps in $O(1)$ time by linking their root lists together. The list of roots can grow quite long as a result. However, whenever we perform a *remove-min* operation, we do a bit of housekeeping and *consolidate* the list of root elements, linking together equal-rank trees until there are only $O(\log n)$ root elements, at most one of each rank. This causes *remove-min* to run in $O(R+\log n)$ time, where R is initial length of the root list. To amortize this properly, we charge only $O(\log n)$ running time to the *remove-min* operation, and we charge $O(1)$ additional time to any operation (e.g., *insert*) that creates a new root, so that each root element always maintains a credit of $O(1)$ units that can be spent to pay for future consolidations. *Merge* only takes $O(1)$ worst-case time, since it simply links two root lists together without drawing any credit from them. [Further details of the consolidation process].

	"Null Path" Mergeable Binary Heaps	Binomial Heaps	Fibonacci Heaps (amortized)
Insert	$O(\log n)$	$O(\log n)$	$O(1)$
Remove-Min	$O(\log n)$	$O(\log n)$	$O(\log n)$
Find-Min	$O(1)$	$O(1)$	$O(1)$
Decrease-Key	$O(\log n)$	$O(\log n)$	$O(1)$
Increase-Key	$O(\log n)$	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$
Merge	$O(\log n)$	$O(\log n)$	$O(1)$
Build	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

FIGURE 5.7: Running times for the operations of several mergeable heaps. The class of "null path" mergeable binary heaps from Sections 5.4.1 and 5.4.2 includes randomized mergeable heaps (whose running time bounds hold with high probability), skew heaps (whose running time bounds are amortized), and also leftist heaps, "size-augmented" heaps, and "null-path-length-augmented" heaps. The running time bounds of these last three are deterministic as long as they don't need to support *delete*, *decrease-key*, or *increase-key*, and otherwise they are amortized.

- **"Lazy" Maintenance of Tree Shapes.** The trees in a binomial heap are maintained in very strict shapes (binomial trees). For the Fibonacci heap, we relax this constraint slightly and allow up to one child to be deleted from every non-root element. Whenever we delete the child of a non-root element, we remember this fact by *marking* the element. If we attempt to delete a child from a marked element e (i.e., if we attempt to delete a second child from e), we first detach e from its parent and then insert e into the list of roots. This forces us to mark e's former parent, unless it was also previously marked, in which case we also detach it from its parent, and so on. The resulting series of cuts may potentially cascade up the tree and is known as a *cascading cut*. Root elements are somewhat special: they may lose more than one child to deletion, and they are never marked. Any marked element that is cut from its parent and made a root becomes unmarked in the process.

The machinery above for marking and cutting may seem a bit mysterious at first, but it gives us two important properties. First, it allows us to perform *decrease-key* in $O(1)$ amortized time. To decrease e's key, we detach e from its parent, decrease e's key, and re-insert e's subtree it into the list of roots. The act of detaching e from its parent could initiate a cascading cut, but it turns out that large cascading cuts happen infrequently. This is another excellent example of the recursive slowdown principle (Section 4.5), since it takes two deletions from an element to cause a cut to propagate to its parent, thereby slowing the rate of a cascading cut by 1/2 every step up the tree. Amortized analysis of *decrease-key* is actually quite simple. We associate one unit of credit with every marked node, allowing it to pay for its part

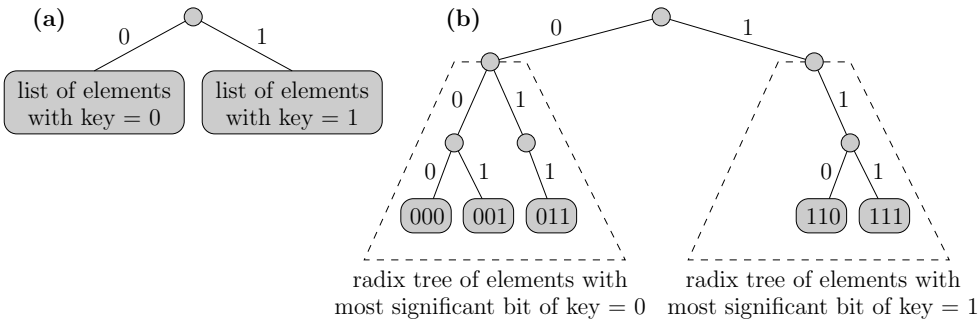

FIGURE 5.8: Illustrations of (a) a simple two-list priority queue data structure of elements whose keys only take the values zero and one, and (b) the generalization of this structure to a radix tree. Each leaf node in the radix tree contains a single element, whose key is written above in binary.

in any cascading cut. The only place *decrease-key* needs to actually pay is at the very end of the cut, where it finally reaches an unmarked element e. Here, it pays 1 unit to delete a child from e and mark e, and then 1 more unit to install the necessary credit on e.

The second important property afforded by the cutting machinery above is the following: in a binomial heap, a tree of rank j contains exactly 2^j elements. This fact was important to guarantee that there were $O(\log n)$ subtrees in the heap. For the case of the Fibonacci heap, it is no longer the case that a rank j tree contains exactly 2^j elements, since *decrease-key* can detach subtrees within a tree. This is slightly worrisome, since the amortized $O(\log n)$ running time of *remove-min* depends on the fact that only $O(\log n)$ trees remain after consolidating the root list. Fortunately, since we are careful to delete at most one child from every non-root element, we can show that a tree of rank j must contain at least F_{j+2} elements, where F_j denotes the j^{th} Fibonacci number [Proof]. As one might guess, this fact explains how Fibonacci heaps get their name. Since F_j is exponentially large in j (one can show that $F_{j+2} \geq \phi^j$, where $\phi \approx 1.618$ is the so-called "golden ratio"), a tree of rank larger than $\log_\phi n$ cannot exist in an n-element Fibonacci heap, so our root list will still contain only $O(\log n)$ elements after consolidation, as desired.

5.5 Integer Priority Queues

Having described quite a few comparison-based priority queues, we now switch gears and focus on "input-sensitive" data structures that operate in the RAM model of computation and assume keys are integers in the range $0 \ldots C-1$. If C is suitably small with respect to n, these data structures can potentially outperform their comparison-based counterparts. To motivate this, consider the extremely simple case where $C = 2$, in which keys have values of zero or one. In this case, we can easily implement every priority queue operation in $O(1)$ time as shown in Figure 5.8(a) by maintaining a pair of doubly-linked lists: one for the elements whose keys have value zero, and the other for elements whose keys have value one.

5.5.1 The Radix Tree

The two-list data structure for $C = 2$ shown in Figure 5.8(a) can be generalized via recursion to a much more versatile and powerful structure called a *radix tree*, shown in Figure 5.8(b). A radix tree is a binary tree whose left subtree is a radix tree recursively constructed from all keys having a most significant bit of zero, and whose right subtree is a radix tree constructed from keys having most significant bit of one. At the next level, the tree branches on the 2^{nd} most significant bit, and so on down the tree. Elements are stored in leaf nodes, and the root-to-leaf path for each element corresponds precisely to its binary representation — zero for a step to the left, one for a step to the right. Every key is represented by $\log C$ bits[4], so the tree has height $\log C$. Since we do not store empty subtrees, a radix tree on n elements requires only $O(n \log C)$ space, because we store $\log C$ nodes along the root-to-leaf path for each element.

The radix tree is actually a very general data structure that has many uses beyond serving as a good priority queue, which we discuss in Chapter 7. Used as a priority queue, it supports all fundamental operations in $O(\log C)$ time, all relatively easy to implement. For example, to *insert* a new element with key k, we walk downward from the root according to the binary representation of k, adding nodes when necessary until we finally deposit the new element as a leaf. To remove the minimum, we first locate the leaf containing this element by walking down from the root (always moving left when possible), and after removing it we walk back up the tree cleaning up any subtrees that become empty as a result.

5.5.2 Multilevel Buckets: Another Route to the Radix Tree

Another way to generalize our trivial two-list structure for $C = 2$ to higher values of C is to build it out "horizontally", as shown in Figure 5.9(a), giving an array $A[0 \ldots C-1]$ of C "buckets", where $A[k]$ points to a list of all elements with key k. Here, all standard priority queue operations take $O(1)$ time except *remove-min*, which takes $\Theta(C)$ worst-case time since it requires scanning through A to find the first non-empty bucket.

For large values of C, either the $\Theta(C)$ worst-case running time of *remove-min* or the $\Theta(C)$ space required to store the structure may be prohibitively large. To remedy this, we again generalize our structure in a hierarchical fashion, arriving at what is sometimes called a *multilevel bucket* data structure. For example, by taking $C = 100$ with $k = 2$ levels of hierarchy, we get the structure shown in Figure 5.9(b). The top level contains $\sqrt{C}$ "super buckets", each corresponding to a range of $\sqrt{C}$ different key values. A non-empty top-level bucket points to a second-level array of $\sqrt{C}$ buckets, each containing a list of elements with a specific key.

We can easily extend such a data structure from a tree of height 2 to one of height k, where each tree node is an array of buckets of length $B = C^{1/k}$. If we take $k = \log C$, we get a structure with $\log C$ levels where every bucket array has length $B = C^{1/\log C} = 2$. This structure should look familiar, as it is precisely the radix tree! In fact, the multilevel bucket data structure is nothing more than a B-ary radix tree, where we write our keys in base B, and at every node we branch in one

[4]We usually assume here that C is a power of two, making $\log C$ an integer.

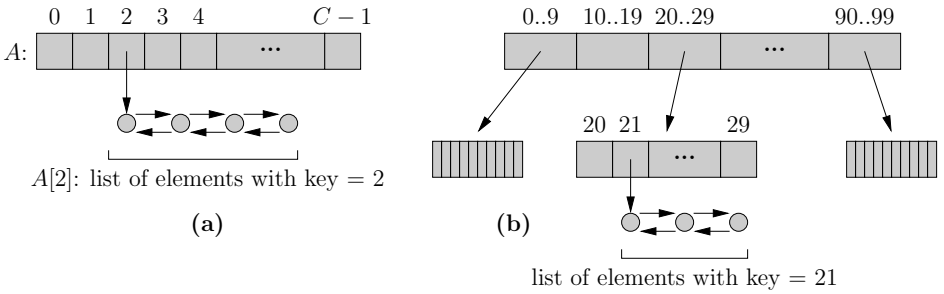

FIGURE 5.9: The multi-level bucket data structure: (a) a one-level array of C "buckets", and (b) a two-level structure, which one can interpret as a two-level tree where every node contains an array of length $\sqrt{C}$.

of B ways based on the value of a particular digit. For the tree in Figure 5.9(b), we write our keys in base 10.

Suppose our the tree has height k, so each node has a "width" of $B = C^{1/k}$ buckets. Since *insert*, *delete*, *decrease-key*, and *increase-key* only require walking vertically through the tree, they take $O(k)$ time. This is constant for the 2-level structure in Figure 5.9(b), but $O(\log C)$ in a radix tree. By contrast, *remove-min* requires scanning horizontally through each node (to locate the first non-empty bucket) as it walks down the tree, so its running time is $O(kB) = O(kC^{1/k})$. For the 2-level structure in Figure 5.9(b), this is $O(\sqrt{C})$; for the radix tree, it is $O(\log C)$. As you can see, the radix tree balances the cost of all operations at $O(\log C)$.

5.5.3 Monotone Integer Priority Queues and the Radix Heap

We can often gain efficiency in the special case of a *monotone* integer priority queue, where it is guaranteed that the sequence of minimum elements removed from the queue will be monotonically nondecreasing. That is, if k denotes the key of the most-recently-removed minimum element, then we promise never to insert an element with key less than k, and we also promise never to decrease the key of an existing element to a value less than k. Such monotonic behavior arises in many important applications; for example, Dijkstra's shortest path algorithm uses a monotone priority queue to processes the nodes of a graph in nondecreasing order of distance from a given source node.

For many applications (again, Dijkstra's algorithm is a good example), the number of calls to *decrease-key* is expected to be significantly higher than the number of calls to *insert* and *remove-min*. This was one of the main motivations for developing the Fibonacci heap, since it allows for *decrease-key* to run in $O(1)$ amortized time. It turns out that monotonicity allows the radix tree / multilevel bucket data structure to perform *decrease-key* also in $O(1)$ amortized time, giving a data structure often called a *radix heap*.

The radix heap is a "lazy" radix tree. Monotonicity tells us that elements will be removed from the left subtree of the root for a while, but if *remove-min* ever hap-

	Radix Tree	k-Level Buckets	Monotone Radix Heap (amortized)	Monotone Radix Heap, Improved (amortized)
Insert	$O(\log C)$	$O(k)$	$O(\log C)$	$O(\sqrt{\log C})$
Remove-Min	$O(\log C)$	$O(kC^{1/k})$	$O(\log C)$	$O(\sqrt{\log C})$
Find-Min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Decrease-Key	$O(\log C)$	$O(k)$	$O(1)$	$O(1)$
Increase-Key	$O(\log C)$	$O(k)$	$O(\log C)$	$O(\sqrt{\log C})$
Delete	$O(\log C)$	$O(k)$	$O(\log C)$	$O(\sqrt{\log C})$

FIGURE 5.10: Running times for the operations of integer priority queues. The improved monotone radix heap data structure refers to the result of problem 82.

pens to extract an element from the right subtree, we know that the left subtree will never again be used. Elements in the right subtree are in some sense not "important" until we reach this crossover point, so instead of maintaining them in a proper radix tree, we store them in a simple doubly-linked list, just as in Figure 5.8(a). This allows elements in the right subtree to be inserted, deleted, and have their keys decreased efficiently. When we reach the crossover point in time when *remove-min* needs to extract from the right subtree, only then do we build and henceforth maintain a tree on these elements. Moreover, we apply the same technique to this tree, leaving its right subtree "unbuilt" until absolutely necessary, and so on. It is easy to show that this approach reduces the running time of *decrease-key* to $O(1)$ amortized. [Details]

Problem 82 (Improving the Radix Heap with Fibonacci Heaps). Consider building a radix heap with branching factor $B = 2^{\sqrt{\log C}}$. By using a Fibonacci heap to store the elements in the length-B arrays at each level, show how to reduce the amortized time of *insert* and *remove-min* from $O(\log C)$ to $O(\sqrt{\log C})$. You may first need to argue how to make a Fibonacci heap on n elements of magnitude at most M take only $O(\log \min(n, M))$ amortized time for *remove-min*. [Solution]

5.6 Additional Problems

We have seen quite a few priority queue data structures throughout this chapter, and in the next two chapters we will see even more data structures that can function as priority queues. In particular, all of the comparison-based search structures in the next chapter can serve as a priority queue (with $O(\log n)$ performance per operation), and in Chapter 7 we will revisit the radix tree and also develop highly-efficient integer search structures such as the Y-fast tree and the van Emde Boas structure that can perform priority queue operations in only $O(\log \log C)$ time (depending on implementation, this bound may be amortized or in expectation).

Problem 83 (Incremental Priority Queues). An *incremental* priority queue supports three operations: *insert*(e), *remove-max*, and *increase-priority*(e). Elements entering the structure start out with priority zero, and this is incremented every time *increase-priority* is called. *Remove-max* removes and returns an element having maximum priority. This structure is useful for several applications, such as computing maximum adjacency orderings in graphs (Chapter **??**). Please describe how to build an incremental priority queue in which all operations take $O(1)$ time (amortized is acceptable, but worst-case is preferred). [Solution]

Problem 84 (Range-Bounded Priority Queues). An integer priority queue is *range-bounded* if the total range of key values stored within the queue never exceeds some small number R (presumably much smaller than C). Give a universal technique that allows us to take any integer priority queue whose running time depends on C and adapt it for the range-bounded case, replacing all instances of C in the running time with R in the process. [Solution]

Problem 85 (Min-Max Heaps). If we want to maintain a set of n elements so that both the minimum and the maximum can be located and removed efficiently, one possibility is to maintain two separate binary heaps, one of them an "min" heap and the other a "max" heap. However, another clever solution is to use only a single binary heap that satisfies a modified version of the heap property: at even-depth elements, we satisfy a "min" heap property (i.e., every element is the smallest in its subtree) while at odd-depth elements, we satisfy a "max" heap property (i.e., every element is the largest in its subtree). Show how to modify the operations of a binary heap so they work in this extended setting and still require only $O(\log n)$ time. [Solution]

Problem 86 (The Offline Priority Queue Problem). Due to the comparison-based sorting lower bound, either *insert* or *remove-min* must run in $\Omega(\log n)$ worst-case time in the comparison model. This is true even in the "offline" case where we are told the sequence of *insert* and *remove-min* operations in advance. However, suppose we have a set of keys whose sorted ordering is already known. For example, let us take the set of values $S = \{1, 2, 3, \ldots, n\}$. Consider now the offline priority queue problem where we are told in advance a sequence of *insert* and *remove-min* operations, where the elements being inserted come from S. Each value in S is inserted at most once, so there are no more than $2n$ total operations. For example, if we are given the input "I4 I1 I5 R R I3 R I2 R R" (where Ix means we insert x, and R means we call *remove-min*, then the correct output should be "1 4 3 2 5". Design an algorithm that computes the answers to all the *remove-min* queries in only $O(n\alpha(n))$ time. As a hint, the target running time of this algorithm suggests a particular data structure you should use. [Solution]

Problem 87 (Multi-Dimensional Monotonic Array Priority Queues). In problem 56, we investigated the time required to find an element in a multi-dimensional array that is "monotonic" in the sense that its elements are sorted along each dimension. In the two-dimensional case, this looks like a $\sqrt{n} \times \sqrt{n}$ array whose rows and columns are in sorted order. In the three-dimensional case, it becomes an $n^{1/3} \times n^{1/3} \times n^{1/3}$ array whose elements are monotonically increasing if we move forward along any single dimension (e.g., $A[i, j+1, k] \geq A[i, j, k]$). Using the observation that one can "sift" effectively (in a similar fashion to the sift operations in a binary heap) in one of these structures if an element is modified, please discuss how to build a priority queue whose underlying implementation is a monotonic multi-dimensional array. Please describe the running time of all major priority queue operations (*insert*, *remove-min*, *decrease-key*, *increase-key*, and *delete*) in terms of n as well as the dimensionality, d, and comment on what should be the optimal choice for d. [Solution]

Problem 88 (Median Filtering with Binary Heaps). An n-element array of noisy data can be smoothed using a *median filter*, by sliding a length-k window over

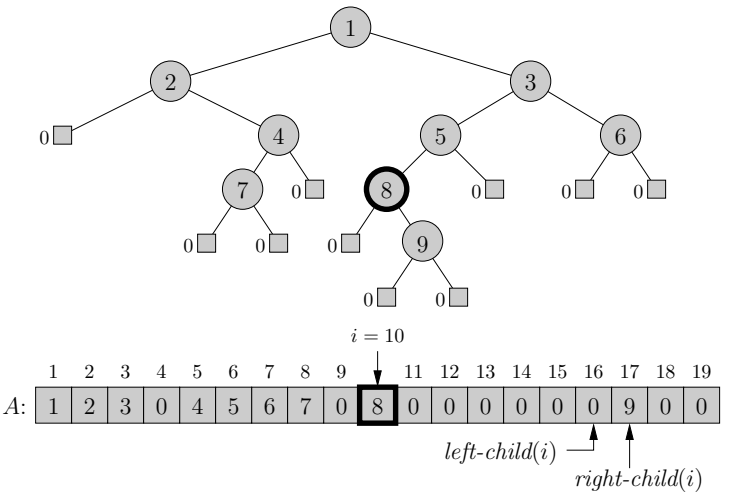

FIGURE 5.11: Level-order encoding of an arbitrary n-node binary tree using a single array of length $2n+1$. Nodes are numbered $1 \ldots n$ in a level-ordered fashion, with zeros representing external nodes. Both internal and external nodes are then written in a level-ordered fashion into the array A.

the sequence, outputting at each location the median element within the window. There are several nice ways to implement a median filter in $O(n \log k)$ time (the best possible in the comparison model). For example, we can store the contents of the window in a balanced binary search tree (discussed in the next chapter), which supports the operations *insert*, *delete*, and *select* (for finding the median) all in $O(\log k)$ time. Since a balanced binary search tree can be complicated to implement, however, please discuss how you might implement a median filter in $O(n \log k)$ time using a pair of binary heaps instead. In general, show how to build a data structure out of two binary heaps capable of tracking the rth order statistic of a dynamic data set (with r known in advance and unchanging over the lifetime of the structure) such that insertion or deletion takes only $O(\log n)$ time. [Solution]

Problem 89 (Level-Order Encoding of an Arbitrary Binary Tree). We have emphasized earlier that one of the benefits of a binary heap — being shaped like an almost-complete binary tree — is that one can represent the heap in memory using nothing more than a simple array. Moreover, we can step around within the array as if we were moving in the tree, since the left and right children of the node at index i are located at indices $2i$ and $2i + 1$. In this problem, we show that a generalization of this mapping allows us to represent any static binary tree within an array, also allowing for easy tree-based movement. As shown in Figure 5.11, we map an n-node tree into an array $A[1 \ldots 2n + 1]$ in a level-by-level fashion, storing the nodes of the tree as well as a set of dummy "external" nodes representing null spaces at the bottom of the tree (these are stored as zeros in the array). Note that the almost-complete binary heap is just a special case of this mapping, where all the external nodes lie at the end of the array. Movement from parent to child generalizes in a pleasantly simple way: please prove that the children of the node at index i live at indices $2A[i]$ and $2A[i] + 1$. We will revisit this mapping later in problem 136, when we use it to develop a "succinct" data structure for encoding a static, rooted tree. [Solution]

Problem 90 (Priority Queues Based on Braun Trees). A *Braun* tree (named after one of the first researchers to investigate the structure) is a binary tree that is "perfectly balanced" in the sense that for every node, its left subtree is either the same size or one element larger than its right subtree (and as a consequence, the tree has $O(\log n)$ height — see also problem 98 and Section 6.2.6 to see examples of these types of structures used in the context of balancing a binary search tree). To store a Braun tree in memory, each node resides in its own block of memory and maintains pointers to its parent, left child, and right child. In our present application, however, each node does not need to maintain information about the size of its subtree. In this problem, we show how Braun trees give us an elegant means of building priority queues, which happen to be particularly well-suited for functional programming environments.

(a) Given a pointer to the root of a Braun tree satisfying the heap property, show how to insert a new element in $O(\log n)$ time while maintaining the heap property. As a hint, you may want to draw inspiration from the way nodes in a skew heap swap their children. [Solution]

(b) Given a pointer to the root of a Braun tree satisfying the heap property, show how to remove the root (the minimum element) in $O(\log^2 n)$ time. Similarly, given a pointer to an arbitrary element, show how to delete that element in $O(\log^2 n)$ time, thereby enabling support for operations like *decrease-key* and *increase-key*. All of your operations should preserve the heap property. [Solution]

(c) For a challenge, show how to compute the size of a Braun tree in $O(\log^2 n)$ time, given only a pointer to its root. [Solution]

Problem 91 (A "Binomial Heap" Comprised of Sorted Arrays). At the beginning of this chapter, we briefly discussed the use of a single sorted array as a simple but inefficient means of implementing a priority queue. In this problem, we consider a generalization of this idea that uses several sorted arrays (or linked lists, if you think this is preferable) of different lengths. Each priority queue element will be stored in one of these arrays. Let us require that for each array in our collection (say it has length L), the next-largest array must have length at least $2L$. Since this property ensures that we will have at most $O(\log n)$ different sorted arrays in our data structure, *find-min* should take $O(\log n)$ time. To ensure that this property is maintained, we augment each array with its length, and we store these arrays in a linked list in order of their lengths. To insert an element e into the structure, we insert a new length-1 array containing only e. If this happens to violate the property above (i.e., if there already exists a length-1 array), we repeatedly merge the array containing the new element with the next-largest array until the property finally becomes satisfied (using the same linear-time merge operation as in merge sort). To perform *remove-min*, we first locate the minimum element at the end of one of our arrays and then remove it by shortening the array. Again, this might violate the property above if our array shrinks to less than twice the size of the next-smallest array, so in this case we again perform successive merges until the property is restored. Both *insert* and *remove-min* have $\Theta(n)$ worst-case running times. However, for a challenge, can you prove that both *insert* and *remove-min* run in only $O(\log n)$ amortized time? Is it possible to implement the remaining priority queue operations *decrease-key*, *increase-key*, and *delete* in $O(\log n)$ amortized time as well? [Solution]

Problem 92 (Combining Disjoint Sets and Mergeable Priority Queues). Mergeable priority queues maintain a collection of priority queues on disjoint sets of elements, so that pairs of priority queues can be merged together efficiently. This should sound very much like the disjoint set problem (Section 4.6), where we maintain disjoint sets of elements that can be quickly unioned together. The only disjoint set operation that is not commonly offered by mergeable priority queues is the *find* operation, which reports the identifier of the set (in this case, the priority queue) to which a specific element belongs.

In this problem, we show how to take any fast priority queue (not even one supporting *merge*) and make it mergeable, while also supporting the disjoint set *find* operation. Suppose we take the highly efficient tree-based data structure for disjoint sets from Section 4.6 (supporting *union* and *find* in $O(\alpha(n))$ amortized time) and modify it slightly. In each tree, we will store elements only at leaves, and each non-leaf (internal) node will maintain a priority queue. Please determine what we should store in these priority queues so that if our original priority queue operations run in $O(T)$ amortized time, the operations *insert*, *remove-min*, *find*, and *merge/union* all now take $O(T\alpha(n))$ amortized time in this hybrid data structure. [Solution]

