14

HEAPS

In the last chapter, we worked with binary trees, and we'll continue doing so in this chapter, but with a variant that is stored without the need for dynamic memory: heaps. Heaps allow for easy implementation of a new abstract data type (ADT), a good performance sorting method, and a new structure for an enhanced version of binary search trees. We'll consider implementing heaps (in particular, binary max heaps, but other types too), and we'll look at using heaps for priority queues, sorting arrays with heapsort, and searching with another new structure called treaps.

In the next chapter, we'll follow with another representation for heaps that uses dynamic memory and allows for more freedom and better performance for some new operations.

Binary Heaps

A *binary heap*, usually referred to simply as a *heap*, is a type of binary tree with two particular properties: a *structure property* that determines the shape of the tree and a *heap property* that specifies the relationship between the key of a parent node and those of its children.

The Structure Property

Heaps are a subset of binary trees, and the structure property requires that the tree must be complete and that all leaves on the last level must be located on the left. Consider the trees in Figure 14-1. Only one qualifies as a heap while the other two fail. Can you tell which is which?

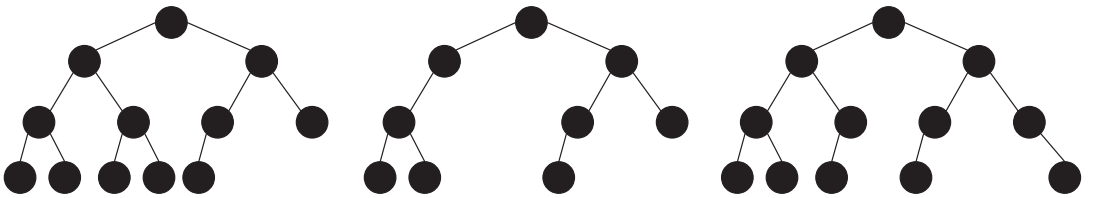

Figure 14-1: Of these three heap candidates, which is the right one?

The tree on the left in Figure 14-1 is the only heap. The tree at the middle has an incomplete middle level, and in the tree on the right, the bottom children are not all on the left.

Given this rule, you can store a heap in a common array without dynamic memory or pointers to simplify implementation. (See question 14.18 to consider an alternative.) Place the root at the first position of the array, followed by the nodes at the second level from left to right, then the nodes at the third level (also from left to right), and so on. Figure 14-2 shows how the array looks for a sample case. The numbers in the nodes correspond to indices in the array.

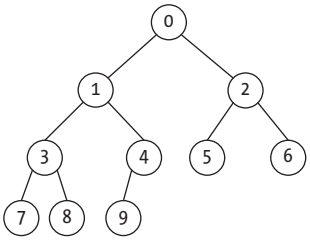

Figure 14-2: Storing a heap's nodes in an array

In this representation, the heap's root is always at position 0. The left and right children of the node at position `p` are placed in consecutive adjacent positions, `2*p+1` and `2*p+2`, respectively—unless they fall beyond the

end of the heap, in which case the node has fewer children. The parent of a non-root node at position `p` is found at the `Math.floor((p-1)/2)` position.

You can verify those rules with a few examples. The children of the root (at position 0) are at positions `2*0+1=1` and `2*0+2=2`. Node 4 has a single child at position `2*4+1=9`, because the other child would be beyond the heap size. The parent of node 9 is at position `Math.floor((9-1)/2)=4`. The parent of node 2 is at position `Math.floor((2-1)/2)=1`.

These rules let you implement algorithms without needing any pointers; a simple array suffices. As with complete trees in Chapter 12, if a heap has up to $n = 2h - 1$ nodes, its height will be h, so its height is bounded by $\log n$, a result that will feature in order calculations.

The Heap Property

The second property of heaps is simple: the key of a node must be greater than or equal to the keys of its children. This is an important contrast from binary search trees: in heaps, there's no difference between a left child and a right child (either could be greater than the other), although they both will be smaller than or equal to their parent. Any tree that follows both the structure property and the heap property is called a *binary max heap* or, more simply, a *heap*.

NOTE *Why do I say the word* heap *means* max heap *by default? The "New York, New York" song lyrics may give us a clue: "I want to wake up in a city that doesn't sleep, and find I'm king of the hill,* top of the heap.*" This suggests the root (top) of the heap should be the greatest value, doesn't it?*

You can reverse the condition and specify that the parent's key be smaller than or equal to those of its children, meaning the root would be the minimum value of the heap. This variant is called a *min heap*, and you can use it (among other scenarios) to merge several linked lists, which requires finding the minimum of many elements repeatedly (see question 14-5). The heap shown in Figure 14-3 satisfies both the structure and heap properties.

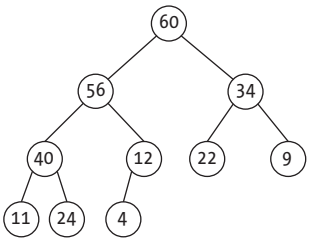

Figure 14-3: A valid heap

As mentioned previously, this tree also could be represented by an array where 60 (the root) is at position 0 of the array, as shown in Figure 14-4.

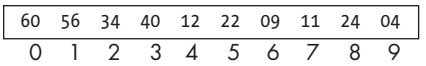

Figure 14-4: The same heap shown in Figure 14-3, as stored in an array

The heap property has an immediate consequence: the highest value in the heap will necessarily be at its root; can you see why? Where in the heap will you find its second highest value? The third highest? The fourth? (See question 14-13.) This result will be key for a sorting algorithm called *heapsort*, which we'll study later in this chapter.

Heap Implementation

To implement a heap, you need only a simple array. A heap is a data structure with a few operations, as shown in Table 14-1.

Table 14-1: Operations for the Heap ADT

Operation	Signature	Description
Create	→ H	Create a new heap.
Empty?	H → boolean	Determine whether the heap is empty.
Top	H → key	Given a heap, produce its top value.
Add	H x key → H	Given a new value, add it to the heap.
Remove	H → H x key	Given a heap, extract its top value and update the structure correspondingly.

The functions you'll implement for those operations are:

`newHeap()` Creates a new heap

`isEmpty( heap)` Determines whether the heap is empty

`top( heap)` Gets the value of the topmost (maximum) element of the heap

`add( heap, value)` Adds a new element to the heap

`remove( heap)` Removes the topmost element of the heap

The first three functions are very short:

```
❶ const newHeap = () => [];

❷ const isEmpty = ( heap) => heap.length === 0;

❸ const top = ( heap) => {
    if (isEmpty( heap)) {
      return undefined;
    } else {
      return heap[0];
    }
  };
```

Creating a new empty heap is the same as returning an empty array ❶. The size of the heap is `heap.length`, so checking it for `0` tells you whether the heap is empty ❷. Also, the top of the heap (unless the heap is empty, in which case this code returns `undefined`) is at the first position of the array ❸, so the implementation details are straightforward.

Adding to a Heap

To add a new value to the heap, follow these steps:

1. Add the new value at the end of the array.
2. If the value is greater than its parent, exchange places with it repeatedly.
3. When the value is smaller than its parent or when it gets to the top of the heap, stop.

Let's see how this works. Start with the heap shown in Figure 14-5.

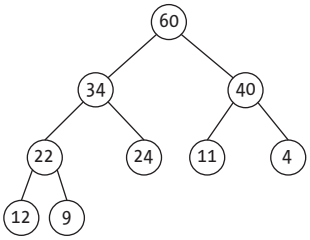

Figure 14-5: An initial heap, before adding a new value

If you want to insert a new value of 56, the first step is to add it at the end of the heap, so you'd get the result shown in Figure 14-6.

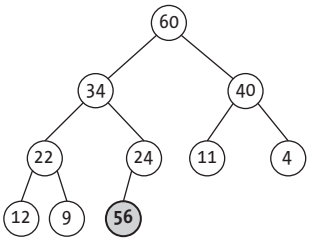

Figure 14-6: The new value (56) starts at the end of the heap.

Let's see if the new value should bubble up. Comparing 56 with its parent (24) shows that they need to be swapped, resulting in a new heap configuration (see Figure 14-7).

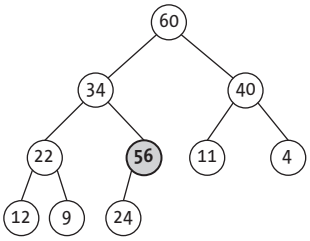

Figure 14-7: If the new value is greater than its parent, it has to "bubble" up.

After bubbling up, keep checking recursively, and a new upward movement is required (see Figure 14-8).

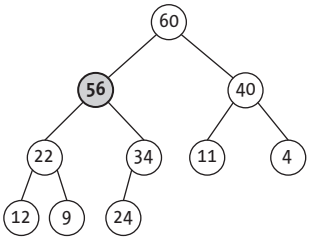

Figure 14-8: Bubbling up continues until the added value is not greater than its parent or at the root of the heap.

The last step led to a situation where the inserted value is not greater than its parent, so the algorithm stops.

Our version of `add()` is short and to the point:

```
const add = ( heap, keyToAdd) => {
❶ heap.push(keyToAdd);
❷ _bubbleUp( heap, heap.length - 1);
  return heap;
};
```

As described, the new value was added at the end of the heap ❶, and it was forced to bubble up to its final position ❷ by using the `_bubbleUp()` auxiliary function.

As before, a recursive implementation is easiest. If the element has moved up, apply `_bubbleUp()` recursively to keep it moving:

```
const _bubbleUp = ( heap, i) => {
❶ if (i > 0) {
  ❷ const p = Math.floor((i - 1) / 2);
  ❸ if ( heap[i] > heap[p]) {
    ❹ [ heap[p], heap[i]] = [ heap[i], heap[p]];
```

```
    ❺ _bubbleUp( heap, p);
    }
  }
};
```

If the element is not already at the top of the heap ❶, use math (as in "The Structure Property" on page XX) to determine the parent `p` of position `i` ❷. If you need to swap elements ❸, destructuring makes it easy ❹, and you can keep bubbling up (if needed) by using recursion ❺.

Removing from a Heap

Next you need the `remove()` method. Remember that the whole heap must become one element smaller, so what happens after removing the top of the heap? If the heap is empty, there's nothing to remove; throw an exception, and you're done. Otherwise, pick the last element of the heap, place it at the top, and then reduce the heap size by one. If the element doesn't have any children, stop. If the element is greater than the greatest of its children, also stop. Otherwise, exchange the element with its greatest child and keep moving it down.

Here is an example of how this works. Start with the heap in Figure 14-9.

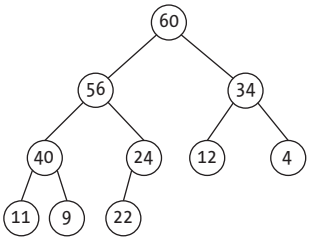

Figure 14-9: An initial heap before removing its top

The first step involves removing the top value (60), replacing it with the last value in the heap (22), and shortening the heap by one, which leads to the situation shown in Figure 14-10. The value that needs to be moved downward to restore the heap is highlighted.

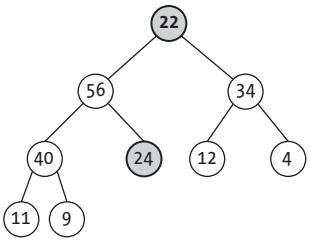

Figure 14-10: After removing the top, replace it with the last element of the heap (22).

Now start sifting down. Comparing 22 with its children, it needs to be swapped with 56, which results in the new situation shown in Figure 14-11.

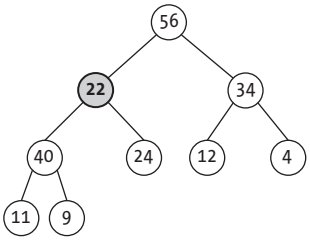

Figure 14-11: If the new top isn't greater than its children, it must "sink" down.

Recursively, compare 22 with its new children, and again, it needs to sink down, as shown in Figure 14-12.

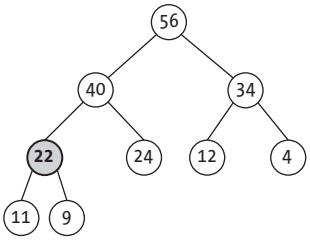

Figure 14-12: Sinking down proceeds until the value is greater than its children or reaches a leaf.

In this situation, 22 is now greater than its children, so the sinking-down procedure ends. If the value 40 had been a 20, the 22 would have been exchanged with the 24, and the shifting-down procedure would have finished as well, since 22 would have no children.

Take a look at the following code, which uses a recursive `_sinkDown()` auxiliary function to push a value down the heap:

```
const _sinkDown = ( heap, i, h) => {
❶ const l = 2 * i + 1;
❷ const r = l + 1;
❸ let g = i;
  if (l < h && heap[l] > heap[g]) {
    g = l;
  }
  if (r < h && heap[r] > heap[g]) {
    g = r;
  }
  if (g !== i) {
  ❹ [ heap[g], heap[i]] = [ heap[i], heap[g]];
  ❺ _sinkDown( heap, g, h);
  }
};
```

Calculate `l` and `r`, the children of the parent at `i`; you can use the formula discussed earlier in "The Structure Property" section on page XX ❶ and find `r` just by adding one ❷, since `r` follows `l` in the array. Use `g` to determine the greatest value at positions `i`, `l`, and `r` ❸. If the value at `i` isn't greater than its children, swap it ❹ and keep sinking it down recursively ❺.

At this point, you can finally write the `remove()` function:

```
const remove = ( heap) => {
❶ const topKey = top( heap);
❷ if (!isEmpty( heap)) {
  ❸ heap[0] = heap[ heap.length - 1];
  ❹ heap.length--;
  ❺ _sinkDown( heap, 0, heap.length);
  }
❻ return [ heap, topKey];
};
```

This code closely follows the description in the previous example. When you get the top (which may be undefined, if the heap is empty) ❶, and if the heap isn't empty ❷, place its last value at the top ❸, reduce the heap's length by one ❹, and sink the new top down ❺. Finally, return the top value and the updated heap ❻.

Considering Performance for Heaps

Table 14-2 shows the order of the algorithms just explored.

Table 14-2: Performance for Operations on a Heap

Operation	Performance
Create	*O*(1)
Empty?	*O*(1)
Top	*O*(1)
Add	*O*(log n)
Remove	*O*(log n)

Three operations work in constant time: creating a heap, testing whether it's empty, and getting the top value. The other two operations, adding and removing, are more complex. Adding an element may make it bubble up from the bottom of the heap all the way up to the top. Since you know that the heap's height is log *n*, this operation requires logarithmic time. Similarly, removing an element implies placing a new one at the top and possibly sinking it down to the bottom. It's the same number of operations as in adding a value, but in reverse (also logarithmic time).

Let's move on to consider a new ADT, and compare the performance of heaps versus the other structures already discussed.

Priority Queues and Heaps

Priority queues (PQs) are different from the queues discussed in Chapter 10, because each element has an associated priority, which determines what element is removed first. In a PQ, the first element to be removed is the one with the highest priority, not the first one that was added, as in a first in, first out (FIFO) strategy.

NOTE *There's a problem with the English language! The term "priority one" implies the "highest priority," but 1 is the lowest-priority number. If you order tasks by priority and the lowest numbered task is the one you should tackle first, then lower numbers have a higher priority. However, some tools (like Microsoft Project) assume that 0 is the lowest priority and higher numbers are higher priority, so there's no clear-cut case. Regardless, if you actually need a min heap instead of a max heap, see question 14.4.*

PQs are used in multiple algorithms and many different situations. Operating system schedulers use priorities to select what process will be the next to run. Discrete event simulations decide the next step to apply based on a timestamp (and in this case, lower timestamps equal higher priorities). Dijkstra's shortest path algorithm (which we'll consider in Chapter 17) requires finding the vertex with the minimum distance to another given vertex. Prim's algorithm for finding a minimum spanning tree for a graph also needs to find the vertex with the smallest (cheapest) connection to another vertex. Huffman's coding algorithm builds a tree and repeatedly needs to find the two nodes with the smallest probabilities to replace them with a new node with the sum of those probabilities. All of these things require PQs.

In terms of an ADT, the description for a PQ then requires the following operations shown in Table 14-3 (other versions that add more operations are considered later).

Table 14-3: Operations for the Priority Queue ADT

Operation	Signature	Description
Create	→ PQ	Create a new PQ.
Empty?	PQ → boolean	Determine whether the PQ is empty.
Top	PQ → key	Given a PQ, produce its top value.
Add	PQ x key → PQ	Given a new key, add it to a PQ.
Remove	PQ → PQ x key	Given a PQ, extract its top value and update the PQ correspondingly.

In terms of the provided operations, a heap matches the requirements of a PQ, so implementation is straightforward. For the sake of variety, though, take a look at other simple ways of implementing PQs and compare their performances:

- With an unordered array or list, getting the top value would be $O(n)$. Removing it would also be $O(n)$, because you have to go through all the values to find it, and adding a new element would be $O(1)$.

- With an ordered array (the top value at the last position), getting and removing the top would be $O(1)$, but adding a new element would be $O(n)$; after finding where the element goes, which is $O(\log n)$ with binary search, you have to move the elements physically to make space, and that's $O(n)$.
- With an ordered list (the top value in the first place), the results are the same as with an ordered array.
- With a balanced binary search tree, all three operations would be $O(\log n)$. If you have an extra pointer to the maximum value, getting the top value becomes $O(1)$, but insertions and deletions will be a tad slower because they have to maintain the added pointer.

This list of possible implementations for PQs isn't complete, but it should be enough to show how the heap is one of the best ways of implementing them, with extra points for low complexity. n the following chapter, we'll consider some extra operations you may need, which will lead to other implementations of PQs.

Heapsort

Heaps can be used to create a well-performing sorting method. Given a set of values, with a heap, you can easily find the highest value of the set. After removing it and restoring the heap, you can find the second-highest value of the set, and so on. The basic algorithm structure is as follows:

1. Build a heap out of the values to be sorted.
2. Then, until no more values are left, swap the heap's top element with its last.reduce the heap size by one, and restore the heap condition.

Take a look at how this algorithm works and then consider an enhancement.

Williams' Original Heapsort

First, here is an example implementation of the heapsort algorithm, invented by John W. J. Williams in 1964. You can reuse the `_bubbleUp()` and `_sinkDown()` functions from earlier without any change (so we won't list them here), and what's added is just the following:

```
function heapsort_original(v) {
❶ for (let i = 1; i < v.length; i++) {
    _bubbleUp(v, i);
  }

❷ for (let i = v.length - 1; i > 0; i--) {
  ❸ [v[i], v[0]] = [v[0], v[i]];
  ❹ _sinkDown(v, 0, i);
  }

  return v;
}
```

The first stage of heapsort goes from the beginning to the end of the array, making each element bubble up to its correct place ❶. The second stage ❷ swaps the top element with the last one of the (current) heap ❸ and sinks it down, using the second argument to `_sinkDown()` to limit how far it can sink ❹.

Here is the algorithm in action. The building phase goes through the stages shown in Figure 14-13. The highlighted area corresponds to the heap that's being built, and the rest are the values that haven't yet been added to the heap.

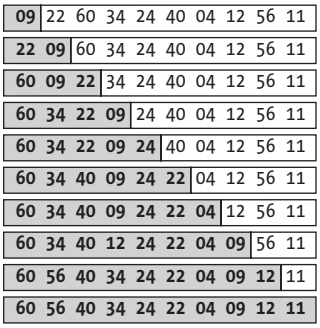

Figure 14-13: The first phase of heapsort is to build up the heap (highlighted area) one value at a time.

In every step of the algorithm, a new value is added to the heap as it is so far, bubbling up as needed, until you get a heap with one more element than before. After the building phase is done, the second part of the algorithm starts. The top element of the heap is swapped with the last value of the heap, which shrinks down in size by one, and the new value at the top sinks down to restore the heap, as shown in Figure 14-14.

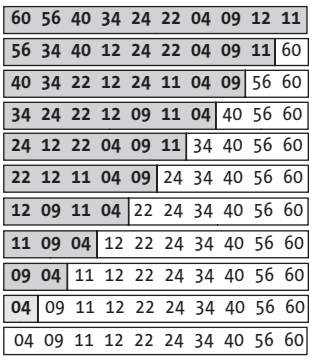

Figure 14-14: In the second phase of heapsort, the top value is repeatedly removed to build up the sorted array.

In the first step, the top value (60) is exchanged with the last value of the heap (11). The 11 sinks down, and 56 moves up to the top. In the next step, 56 is exchanged with the last value (again 11), and 11 sinks down as 40 goes to the top. This continues step by step, and when the heap is at size 1, the array is sorted.

Heapsort Analysis

What's the order of heapsort? Without going too deep into mathematical details, if you are sorting n items, you call the `_bubbleUp()` function n times, and each time, an element may bubble up to the top of the heap, which is log n high, so that's $O(n \log n)$. Similarly, when removing elements from the heap to produce the ordered array, call `_sinkDown()` n times, and elements may sink down to the bottom of the heap, so that's $O(n \log n)$ again; the conclusion is that the algorithm is $O(n \log n)$.

An interesting property is that this behavior is guaranteed. No set of data will lead to a worse case, as in quicksort, which could become $O(n^2)$. Also, since we've established that $O(n \log n)$ is the best possible order for comparison-based sorting algorithms, you can see that heapsort is a solid algorithm with consistent performance, often used in libraries and in other algorithms that may require sorting.

Finally, heapsort isn't a stable sort in the sense shown in Chapter 6 (see question 14-12 for an example).

Floyd's Heap-Building Enhancement

Williams' version of the algorithm is quite efficient, but a possible enhancement, thanks to Robert Floyd, speeds up the heap-building part to O(n), although we won't go into the math here. The reason for this result is that most of the elements are near the bottom, so sinking them down is much faster than bubbling them up. Very few are near the top, where sinking them is slower than bubbling up, all of which is enough to change the order of heap building. Since the second part of the procedure is still $O(n \log n)$, the algorithm's total order won't change, but it will run faster in any case.

Instead of making each element bubble up to its position, the algorithm builds small heaps, which are then made larger by joining them together, until you get the complete heap. Initially, you can consider all the leaves of the tree to be small one-sized heaps. Then, take two leaves and their parent and reorganize them (if needed) so that those three values form a heap. Keep doing this, and eventually, you'll get to the top of the heap and be done.

Take a look at the code first and then at an example. The code for this new heapsort version will depend on the previous `_sinkDown()` code, which you'll use unchanged. The rest of the algorithm is as follows:

```
function heapsort_enhanced(v) {
  for (let i = Math.floor((v.length - 1) / 2); i >= 0; i--) {
    _sinkDown(v, i, v.length);
  }
```

```
  for (let i = v.length - 1; i > 0; i--) {
    [v[i], v[0]] = [v[0], v[i]];
    _sinkDown(v, 0, i);
  }
  return v;
}
```

The code for the second part of the algorithm (exchanging and restructuring) is the same; the only difference is where you build up the heap using `_sinkDown()`. Figure 14-15 shows just the heap-building part of this code in action—specifically, how more parts of the array successively become a heap. At each step, the part of the array that becomes a "mini heap" is highlighted.

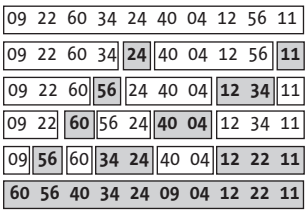

Figure 14-15: The enhanced heap-building algorithm creates the heap out of smaller previously created ones.

To better understand Figure 14-15, look at the heap at different stages. Initially, the array looks like Figure 14-16 and obviously isn't a heap.

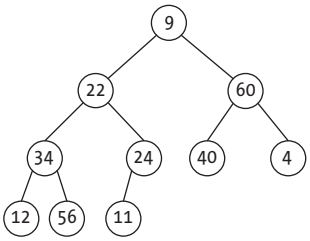

Figure 14-16: An initial array, which doesn't fulfill the heap property

After two steps, subheaps out of 11 and 24 are built (which were left as they were) as well as 12, 34, and 56 (where 34 sifted down and 56 took its place).

Figure 14-17 shows two more steps.

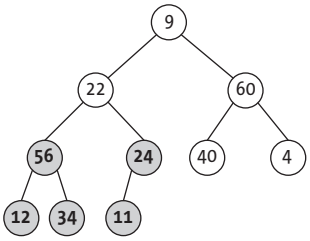

Figure 14-17: After some rotations, a few subheaps are built.

The heap is practically done with root 56 (the 22 value sank down and 56 took its place) and another with root 60 (where no changes were needed). You are just one step away from finishing, as shown in Figure 14-18.

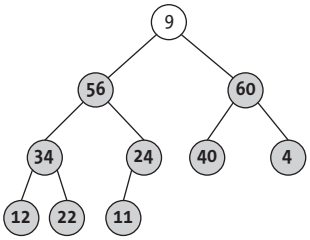

Figure 14-18: More and more subheaps are built, reaching to the top of the heap.

The last step completes the heap; the 9 value sinks down to its place, being replaced by 60. Figure 14-19 shows the finished heap.

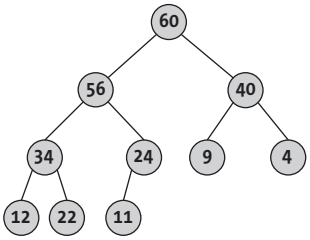

Figure 14-19: Upon reaching the top, the array has become a heap.

Floyd's enhanced algorithm has two advantages: a faster first phase (the second phase, which generates the sorted result, is the same) and shorter code. You can take advantage of this method for your heap logic and modify the `newHeap()` function (see question 14-8).

Treaps

In Chapter 12, we discussed binary search trees and several ways to try to keep them balanced in order to avoid slow searches. In 1989, a new structure was invented that mixes the characteristics of trees and heaps: treaps. These trees are balanced, although their heights are not guaranteed to be $O(\log n)$; rather, randomization and the heap property are used to maintain balance with high probability.

The (invented) term *treap* is a portmanteau of the words *tree* and *heap*. How does this mix come about? Basically, every key is associated with a random priority, and when you build the binary search tree, take care to satisfy the heap property, so a parent node always will have a priority greater than those of its children. (The structure property will not be satisfied; nodes are linked by pointers, not an array.) Note that instead of using a random number generator, you can apply a hash function to the key, and thus produce its "random" priority. Mathematical analysis of treaps depends on truly random numbers, but the form of randomness generated by hashing also works. And in terms of testing algorithms, hashing has the advantage of determinacy.

Let's think about this a bit more. If you happened to order the keys by priority and insert them in the tree in decreasing order of priority, the resulting tree would satisfy the heap property. (Can you see why?) This means that assigning random priorities is equivalent to taking a random permutation of the keys before inserting them in the tree, which will probabilistically provide a good shape for the resulting tree, with an expected height that is $O(\log n)$, as with balanced trees.

Given a set of distinct keys and their corresponding (also distinct) priorities, the resulting treap is unique, and we can construct a recursive proof for this. First, the root of the treap must be the key with the highest priority. Then, all the smaller keys will go into the root's left subtree and the greater keys will go into the right subtree, and we can apply the same reasoning recursively to prove that those two subtrees will also be unique.

You can also modify the binary search tree algorithms from Chapter 12 to create treaps. The code is simpler than code for AVL or red-black trees, yet it provides competitive performance, often better than that of its more complex alternatives.

Creating and Searching a Treap

Given that treaps are just binary search trees at heart, most of the code discussed in Chapter 12 will still work. Start coding treaps as follows:

```
const {
  find,
  inOrder,
  isEmpty,
  maxKey,
  minKey,
  postOrder,
```

```
  preOrder
❶ } = require("../binary_trees/binary_search_tree.func.js");

const newTreap = () => null;

const newNode = (key) => ({
  key,
  left: null,
  right: null,
❷ priority: Math.random()
});
```

The treap is based on the previous binary search tree ❶, and many of the functions written there are still valid. When creating a new node, add a random priority ❷, but that's all that will change here.

NOTE *If you want to test your code and need deterministic results, you could compute priority as a hash of the key; as long as the results are random enough, it will do.*

Let's move on to adding a new key, which requires some extra coding.

Adding a Key to a Treap

Insertion into a treap requires basically the same logic as for binary search trees, except that after having added the node in its place, you may need to do some rotations to maintain the heap property. Adding new keys to a treap is possible using the `_rotate()` method introduced in previous chapters. The basic idea is first to place the new node in the tree according to its value and then rotate it, if needed, in a way that the binary search tree condition is kept but that also satisfies the heap condition. Figure 14-20 shows the basic rotations.

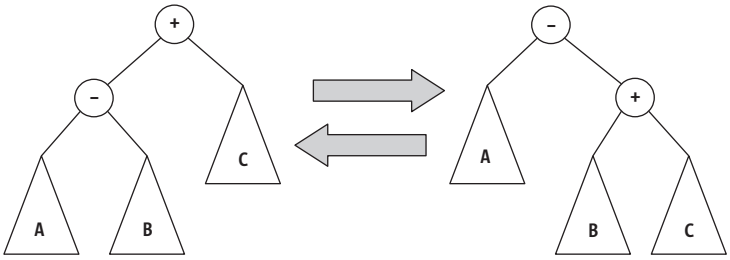

Figure 14-20: Rotations also make the binary search tree become a heap.

The minus sign represents a smaller key value than the plus sign. If the node placed lower (the minus) has a higher priority than its parent (the plus), which is shown in the left side of the figure, you can apply a *right rotation* and produce the situation on the right, which would satisfy the heap property. Conversely, if you have the situation on the right and the lower node (plus) has a higher priority than its parent (minus), you can apply a *left rotation* to get the situation on the left. In both cases, the resulting tree is

still a binary search tree, but the nodes are shifted around so that the final parent node has a greater priority than its children.

Here is an example of how the `add()` algorithm should work. Start with the treap shown in Figure 14-21, where priorities are shown to the right of nodes.

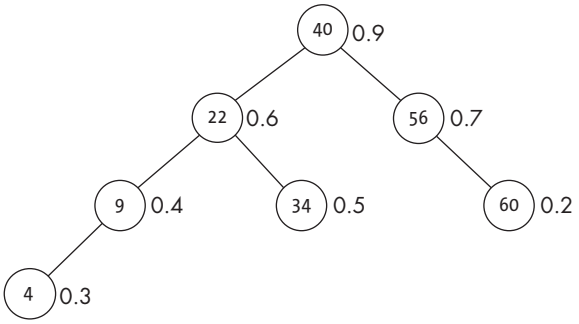

Figure 14-21: In a treap, keys form a binary search tree and priorities form a heap.

If you want to add a 12 node with a (random) priority of 0.8, the first step is to insert the new node, without worrying about priorities and the heap property, which will be sorted out later. This insertion (done in the standard way for binary search trees, as described in Chapter 12) leads to the tree shown in Figure 14-22.

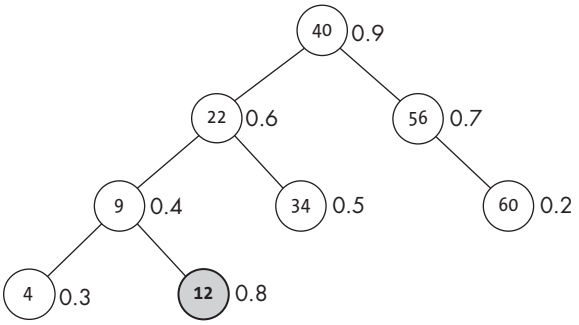

Figure 14-22: After standard insertion, the resulting binary search tree may no longer be a heap.

The treap works for searches, but the heap property isn't satisfied, because the priority for the 12 node is higher than the priority of its parent. You can fix that problem by doing a left rotation, leading to the tree shown in Figure 14-23.

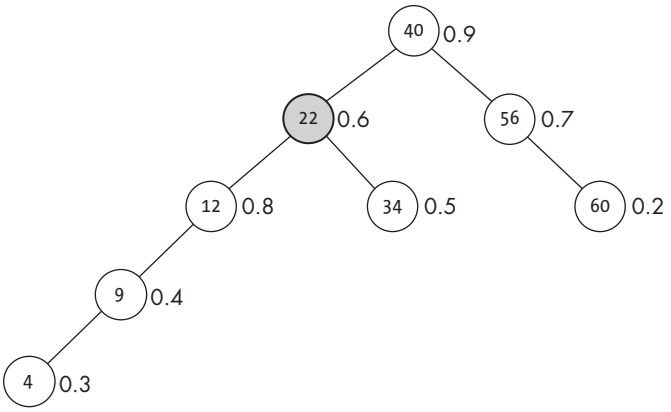

Figure 14-23: Rotations are applied until the heap property is satisfied, but this tree is still wrong.

The rotation still leaves a valid binary search tree, but the process hasn't ended, because the heap property isn't yet fully satisfied. The 12 node has a higher priority than its parent; perform a right rotation to solve this, which leads to the tree shown in Figure 14-24.

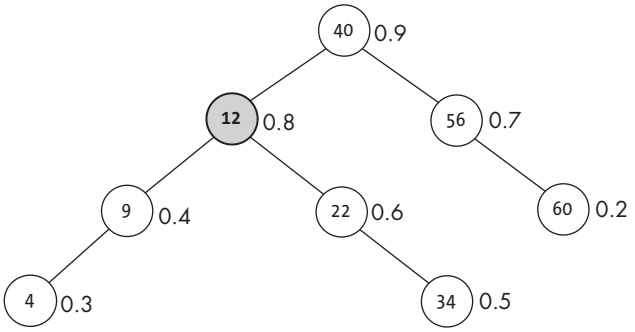

Figure 14-24: Now the heap property is satisfied.

After this second rotation, you can check that the heap property is satisfied, thus the addition to the treap was done correctly.

The final code follows, and there's a single line that's different from common binary search trees:

```
const add = (tree, keyToAdd) => {
  if (isEmpty(tree)) {
    return newNode(keyToAdd);
  } else {
    const side = keyToAdd <= tree.key ? "left" : "right";
    tree[side] = add(tree[side], keyToAdd);
    return tree[side].priority <= tree.priority ? tree : _rotate(tree, side);
  }
};
```

The line in bold makes sure that the heap property is satisfied. After adding the new key on `tree[side]`, if the priority of that subtree is not greater than the priority of the root, you're done; otherwise, apply a rotation on that side to bring up the greater priority.

The last method needed is to remove a key from a treap.

Removing a Key from a Treap

The algorithm for removing a key from a binary search tree involves finding it first, possibly finding its successor, and putting it in the removed node's place. With treaps, given that you must maintain the heap property, it's a tad more complex, but just as with insertions, you can use rotations to make things come out right. To remove a node, use a different logic from earlier with binary search trees:

- If you search for a key to remove in an empty tree, there's nothing to be done.
- If the key to remove is lower than the key at the root, delete the key from the tree at the root's left child.
- Otherwise, if the key to remove is greater than the key at the root, delete the key from the tree at the root's right child.
- Otherwise, if the key has neither a left child nor a right child, just delete it.
- Otherwise, if it has a right child but no left child, set it to the right child.
- Otherwise, if it has a left child, but no right child, set it to the left child.
- Finally, if it has both left and right children, apply a rotation to move the key lower in the tree and attempt to delete it again.

The last step is the one that may be surprising, and it's certainly different from binary search trees. With the rotations shown earlier when inserting in a treap, it's possible to rotate a node with one of its children, and the rotated node will be lower in the treap. If you carefully choose what rotation to use, you can keep satisfying the heap property; so if you had a valid treap before the rotation, you'll still have a valid one after it. And finally, as the key to delete moves lower and lower, it can't always keep having two children. At some point, it will have one or none, and then you can finish the job quickly.

Consider a more complex case. Start with the treap shown in Figure 14-25 and remove the 9 node (as in the section "Adding a Key to a Treap" on page XX, the priorities are shown to the right of the nodes).

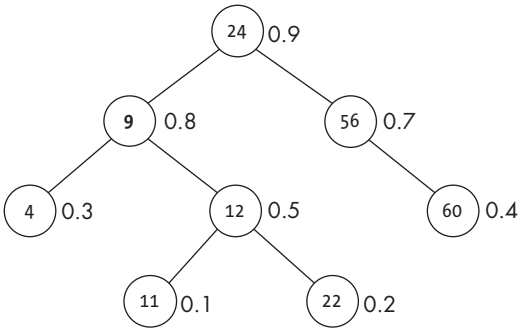

Figure 14-25: An initial treap, with a node to be removed

After finding the node, it happens that it has two children, so you need to do a rotation. The right child of 9 has greater priority, so apply a left rotation, leaving the intermediate situation shown in Figure 14-26 (notice that the heap property is not satisfied, but it will be after you remove the 9 node).

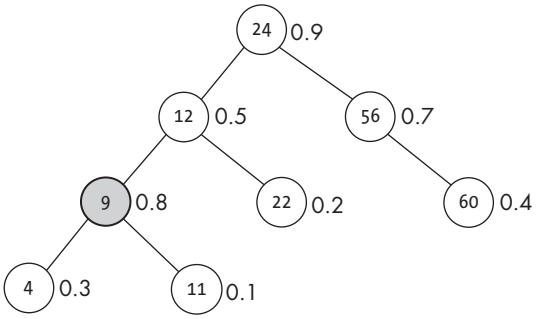

Figure 14-26: A left rotation brings down the node to be removed.

The 9 node again has two children, so do a new rotation. This time, the child with the greater priority is the left one, so you can do a right rotation, resulting in a new situation (Figure 14-27).

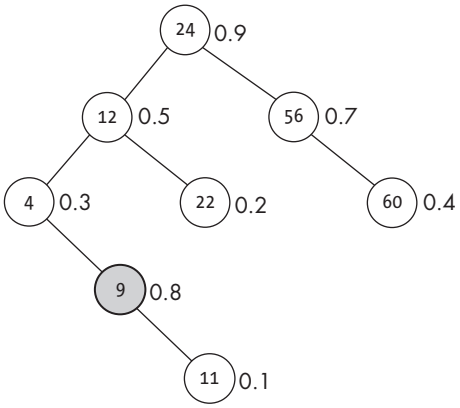

Figure 14-27: A new rotation moves the node to be deleted further down the treap.

Now you've reached a simple case, since the 9 node has only one child, which allows you to remove it, resulting in the final treap (Figure 14-28).

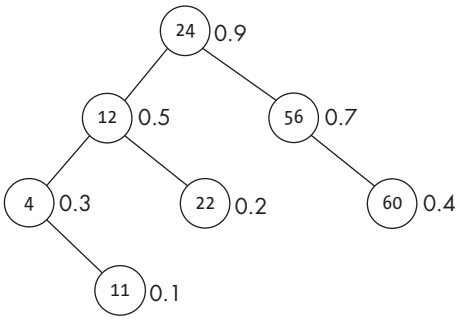

Figure 14-28: The final treap after removing the node you want to delete

The code for removing a key is as follows (notice that the implementation closely follows the previous bulleted list):

```
const remove = (tree, keyToRemove) => {
  if (isEmpty(tree)) {
    // nothing to do
  } else if (keyToRemove < tree.key) {
    tree.left = remove(tree.left, keyToRemove);
  } else if (keyToRemove > tree.key) {
    tree.right = remove(tree.right, keyToRemove);
  } else if (isEmpty(tree.left) && isEmpty(tree.right)) {
    tree = null;
  } else if (isEmpty(tree.left)) {
    tree = tree.right;
  } else if (isEmpty(tree.right)) {
    tree = tree.left;
❶ } else {
```

```
  ❷ const [side, other] =
      tree.left.priority < tree.right.priority
        ? ["right", "left"]
        : ["left", "right"];
  ❸ tree = _rotate(tree, side);
  ❹ tree[other] = remove(tree[other], keyToRemove);
  }
  return tree;
};
```

The code is the same as for common binary search trees, except that when a key is found that has two children ❶, you decide which rotation you need ❷, apply it ❸, and go down recursively to attempt deleting the key ❹. If you had done a left rotation, the original root (with the key you wanted to delete) would be displaced to the left subtree, so the removal process continues there. With a right rotation, the process would continue at the right subtree.

You've now used heaps to extend binary search trees. Let's look at the results of this change.

Considering the Performance of Treaps

As mentioned earlier, the expected height of treaps is $O(\log n)$, which means that adding, removing, and finding keys all have that same expected order. The randomization implied by the assignment of priorities does not ensure, however, that there won't be bad cases, and in fact, the worst-case scenario is the same as with binary search trees: trees having $O(n)$ depth and an effect on the algorithms' performance.

A main difference with common binary search trees is that in practice, getting a "bad" sequence of data may not be unexpected, always leading to bad trees. However, in a treap, because of the random priorities, the probability of building a "bad" treap is very low, no matter the order of the original data. In fact, to get a badly balanced treap, priorities would have to be correlated with the key values, which is very unlikely to happen with random numbers. Thus, the average performance of algorithms will be independent of the sequence of key insertions (see Table 14-4).

Table 14-4: Performance of Operations for Treaps

Operation	Average performance	Worst case
Create	$O(1)$	$O(1)$
Add	$O(\log n)$	$O(n)$
Remove	$O(\log n)$	$O(n)$
Find	$O(\log n)$	$O(n)$
Traverse	$O(n)$	$O(n)$

The key to treaps is that randomization makes it highly probable that some balance will be achieved, thus providing high performance. (This was

the same argument for randomized binary search trees.) In addition, treaps allow you to implement other methods, like partitioning a treap into two or rejoining two treaps to make one. We won't discuss those methods directly here, but see questions 14.15 and 14.16 at the end of the chapter.

Ternary and D-ary Heaps

If binary heaps are a good structure for priority queues, the logical generalization is that as with B-trees, having more children at each level makes for a shorter tree and faster algorithms, such as with *ternary* (also known as *trinary*) heaps, in which each node has three children, or *quaternary* heaps with four children, and in general *d-ary* heaps with *d* children for each node.

Basically, all the differences are in the `_bubbleUp()` and `_sinkDown()` methods, which now have to deal with more than two children, as shown:

```
const { newHeap, isEmpty, top } = require("./ heap.func.js");

❶ const ORDER = 3; // with ORDER===2, we get classic heaps

const _bubbleUp = ( heap, i) => {
  if (i > 0) {
  ❷ const p = Math.floor((i - 1) / ORDER);
    if ( heap[i] > heap[p]) {
      [ heap[p], heap[i]] = [ heap[i], heap[p]];
      _bubbleUp( heap, p);
    }
  }
};

const _sinkDown = ( heap, i, h) => {
❸ const first = ORDER * i + 1;
❹ const last = first + ORDER;
  let g = i;
❺ for (let j = first; j < last && j < h; j++) {
    if ( heap[j] > heap[g]) {
      g = j;
    }
  }
  if (g !== i) {
    [ heap[g], heap[i]] = [ heap[i], heap[g]];
    _sinkDown( heap, g, h);
  }
};

const add = ( heap, keyToAdd) => { ...exactly as before... }

const remove = ( heap) => { ...exactly as before... }
```

Take a look at the changes in the code. We added an `ORDER` variable (here set to `3`) to store the order of the new heap ❶. Calculating the *parent*

of a node requires using a corrected formula; instead of dividing by 2 as in binary heaps, divide by the heap order ❷. Then make the same change (substituting the heap order for the 2) to find the children of an element (❸ ❹). Since you may have any number of children for a node, loop over them to find the greatest ❺.

If you create a new heap (ternary, in this case) and add values 22, 9, 60, 34, 24, 40, 11, 12, 56, 4, and 58, in that order, you'll get the heap in Figure 14-20 (shown as both a tree and an array).

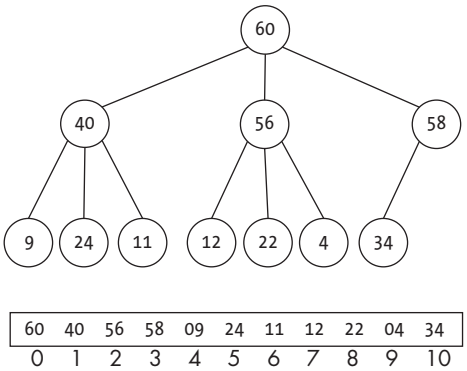

Figure 14-29: In a ternary heap, the implementation is similar to binary heaps.

What about the order of *d*-ary heaps in general? Since the height of the tree is always $O(\log n)$, the order for all operations is the same. However, some operations may have better or worse performance. For instance, bubbling up becomes faster (because the tree is flatter), but sinking down is slower (because you have to find the largest out of more values).

Summary

This chapter introduced a new data structure, a heap in several variants: binary and *d*-ary, as well as min and max heaps. We showed how heaps could be used to implement a new ADT: priority queues. Another usage of heaps was a sorting algorithm with good constant performance. Finally, we applied the heap concept to create a randomized binary search tree: a treap. In the next chapter, we'll continue exploring related concepts and consider some variants of heaps that allow for new operations.

Questions

14.1 Is It a Heap?

Given an array, write a function that returns whether the array is a max heap. You don't need to build a heap; just answer whether it already is one.

14.2 Making Do with Queues

Suppose you could work only with priority queues but not with stacks or queues. How could you emulate those two ADTs with a priority queue? (Hint: since stacks and queues do not use a priority field, you can assign them whichever values you want.)

14.3 Max to Min

Suppose you have a max heap; can you convert it into a min heap in linear $O(n)$ time?

14.4 Max or Min

What changes would you have to apply to your max heaps in order to get min heaps instead?

14.5 Merge Away!

Suppose you are given several ordered lists and want to merge them into a single list. Implement this algorithm using a min heap to decide what node to choose at each step.

14.6 Searching a Heap

Even though it makes little sense (because the heap isn't structured for it), how could you implement a `find()` function to search for a value in a heap?

14.7 Removing from the Middle of a Heap

In a heap, you always remove the value at the top, and if you want to remove the last value of a heap, it's trivial, but can you write an algorithm that allows you to remove any element whatsoever from a heap?

14.8 Faster Build

Floyd's enhancement builds a heap in $O(n)$ time. Modify `newHeap()` so if given an array of values, it will use Floyd's method to initialize the heap.

14.9 Another Way of Looping

In the `heapsort_original` function, you could have easily used `forEach()` to build up the heap; can you see how?

14.10 Extra Looping?

In the `heapsort_enhanced` function, what would have happened if you had done a complete loop when building up the heap? More specifically, what if that code had been written as follows:

```
for (let i = v.length - 1; i >= 0; i--) {
  sinkDown(i, v.length);
}
```

14.11 Maximum Equality

What's the order of heapsort if you use it to sort an array filled with the same value?

14.12 Unstable Heap?

Heapsort isn't stable, and trying to sort a short array is enough to verify this. Can you produce such an example and show the lack of stability? Tip: you won't need a very large array.

14.13 Trimmed Selection

You can use a heap to select the k greatest values out of n by removing the top of the heap k times. However, if $k \ll n$, you may speed up things a bit. Show that the k greatest values must be found in level 1 (the root) up to level k (but not beyond), and use this finding to prune the heap before doing the selection.

14.14 Is It a Treap?

Given a binary tree whose nodes have `key` and `priority` fields, can you write a function that will check whether that tree is actually a treap?

14.15 Splitting a Treap

Given a treap and a limit value, partition it into two separate treaps: one with all the keys smaller than the limit value and the other with all the keys larger than the limit. Assume that the limit value isn't in the treap.

14.16 Rejoining Two Treaps

Consider the inverse to question 14.15: assume that you have two separate treaps, such that all keys in the first are smaller than all keys in the second. Can you find a way to join those two treaps to form a single one?

14.17 Removing from a Treap

If in the `remove()` method for a treap you changed the line

```
tree[other] = remove(tree[other], keyToRemove);
```

to

```
tree = remove(tree, keyToRemove);
```

would it still work? For reference, this is how the code would look (the changed line is in bold):

```
const remove = (tree, keyToRemove) => {
  if (isEmpty(tree)) {
    // nothing to do
  } else if (keyToRemove < tree.key) {
    tree.left = remove(tree.left, keyToRemove);
  } else if (keyToRemove > tree.key) {
    tree.right = remove(tree.right, keyToRemove);
  } else if (isEmpty(tree.left) && isEmpty(tree.right)) {
    tree = null;
```

```
  } else if (isEmpty(tree.left)) {
    tree = tree.right;
  } else if (isEmpty(tree.right)) {
    tree = tree.left;
  } else {
    const [side, other] =
      tree.left.priority < tree.right.priority
        ? ["right", "left"]
        : ["left", "right"];
    tree = _rotate(tree, side);
    tree = remove(tree, keyToRemove);
  }

  return tree;
};
```

14.18 Trees as Heaps

What would happen if you used binary search trees to represent heaps? What would the performance be of the three basic operations: `add()`, `remove()`, and `top()`? Can you think of ways to make `top()` faster?