Chapter 10

Heaps

In this chapter, we discuss two implementations of the extremely useful priority Queue data structure. Both of these structures are a special kind of binary tree called a *heap*, which means "a disorganized pile." This is in contrast to binary search trees that can be thought of as a highly organized pile.

The first heap implementation uses an array to simulate a complete binary tree. This very fast implementation is the basis of one of the fastest known sorting algorithms, namely heapsort (see Section 11.1.3). The second implementation is based on more flexible binary trees. It supports a `meld(h)` operation that allows the priority queue to absorb the elements of a second priority queue `h`.

10.1 BinaryHeap: An Implicit Binary Tree

Our first implementation of a (priority) Queue is based on a technique that is over four hundred years old. *Eytzinger's method* allows us to represent a complete binary tree as an array by laying out the nodes of the tree in breadth-first order (see Section 6.1.2). In this way, the root is stored at position 0, the root's left child is stored at position 1, the root's right child at position 2, the left child of the left child of the root is stored at position 3, and so on. See Figure 10.1.

If we apply Eytzinger's method to a sufficiently large tree, some patterns emerge. The left child of the node at index `i` is at index `left(i)` =

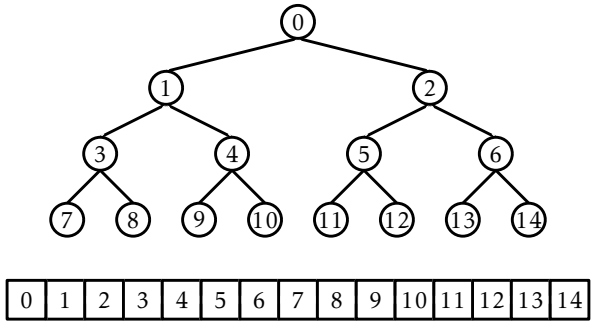

Figure 10.1: Eytzinger's method represents a complete binary tree as an array.

$2i+1$ and the right child of the node at index `i` is at index $\texttt{right}(i) = 2i+2$. The parent of the node at index `i` is at index $\texttt{parent}(i) = (i-1)/2$.

BinaryHeap
```
int left(int i) {
  return 2*i + 1;
}
int right(int i) {
  return 2*i + 2;
}
int parent(int i) {
  return (i-1)/2;
}
```

A `BinaryHeap` uses this technique to implicitly represent a complete binary tree in which the elements are *heap-ordered*: The value stored at any index `i` is not smaller than the value stored at index $\texttt{parent}(i)$, with the exception of the root value, $i = 0$. It follows that the smallest value in the priority `Queue` is therefore stored at position 0 (the root).

In a `BinaryHeap`, the `n` elements are stored in an array `a`:

BinaryHeap
```
T[] a;
int n;
```

Implementing the add(x) operation is fairly straightforward. As with all array-based structures, we first check to see if a is full (by checking if a.length = n) and, if so, we grow a. Next, we place x at location a[n] and increment n. At this point, all that remains is to ensure that we maintain the heap property. We do this by repeatedly swapping x with its parent until x is no longer smaller than its parent. See Figure 10.2.

BinaryHeap
```
boolean add(T x) {
  if (n + 1 > a.length) resize();
  a[n++] = x;
  bubbleUp(n-1);
  return true;
}
void bubbleUp(int i) {
  int p = parent(i);
  while (i > 0 && compare(a[i], a[p]) < 0) {
    swap(i,p);
    i = p;
    p = parent(i);
  }
}
```

Implementing the remove() operation, which removes the smallest value from the heap, is a little trickier. We know where the smallest value is (at the root), but we need to replace it after we remove it and ensure that we maintain the heap property.

The easiest way to do this is to replace the root with the value a[n – 1], delete that value, and decrement n. Unfortunately, the new root element is now probably not the smallest element, so it needs to be moved downwards. We do this by repeatedly comparing this element to its two children. If it is the smallest of the three then we are done. Otherwise, we swap this element with the smallest of its two children and continue.

BinaryHeap
```
T remove() {
  T x = a[0];
  a[0] = a[--n];
  trickleDown(0);
```

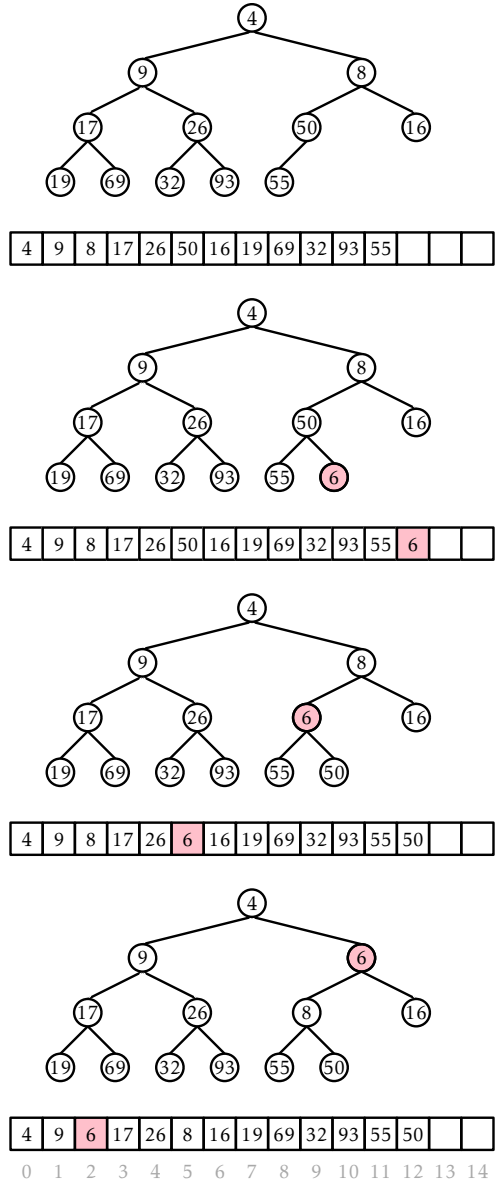

Figure 10.2: Adding the value 6 to a `BinaryHeap`.

```
  if (3*n < a.length) resize();
  return x;
}
void trickleDown(int i) {
  do {
    int j = -1;
    int r = right(i);
    if (r < n && compare(a[r], a[i]) < 0) {
      int l = left(i);
      if (compare(a[l], a[r]) < 0) {
        j = l;
      } else {
        j = r;
      }
    } else {
      int l = left(i);
      if (l < n && compare(a[l], a[i]) < 0) {
        j = l;
      }
    }
    if (j >= 0)  swap(i, j);
    i = j;
  } while (i >= 0);
}
```

As with other array-based structures, we will ignore the time spent in calls to `resize()`, since these can be accounted for using the amortization argument from Lemma 2.1. The running times of both `add(x)` and `remove()` then depend on the height of the (implicit) binary tree. Luckily, this is a *complete* binary tree; every level except the last has the maximum possible number of nodes. Therefore, if the height of this tree is h, then it has at least 2^h nodes. Stated another way

$$n \geq 2^h \ .$$

Taking logarithms on both sides of this equation gives

$$h \leq \log n \ .$$

Therefore, both the `add(x)` and `remove()` operation run in $O(\log n)$ time.

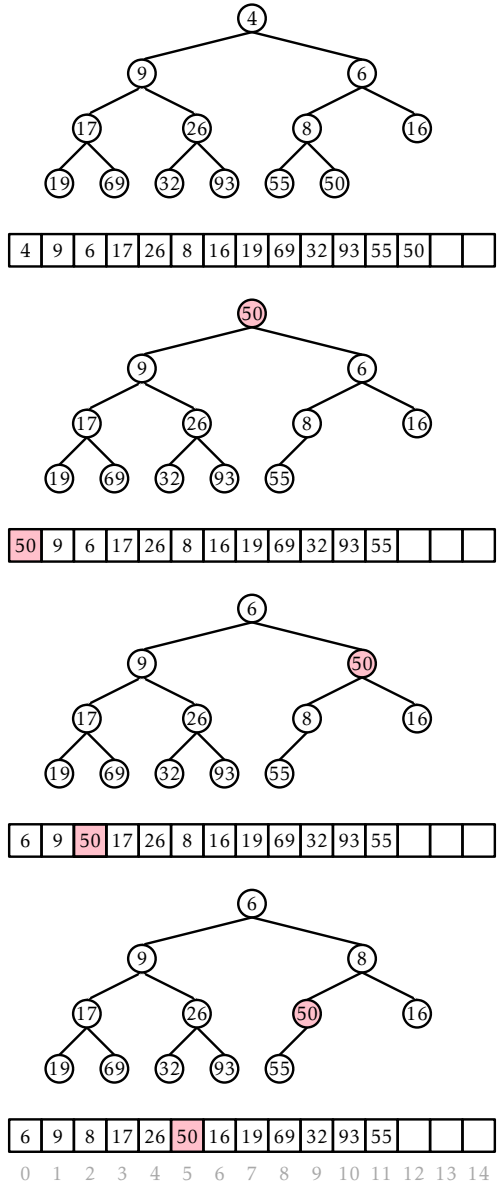

Figure 10.3: Removing the minimum value, 4, from a `BinaryHeap`.

10.1.1 Summary

The following theorem summarizes the performance of a BinaryHeap:

Theorem 10.1. *A BinaryHeap implements the (priority) Queue interface. Ignoring the cost of calls to* resize(), *a BinaryHeap supports the operations* add(x) *and* remove() *in* $O(\log n)$ *time per operation.*

Furthermore, beginning with an empty BinaryHeap, any sequence of m add(x) *and* remove() *operations results in a total of* $O(m)$ *time spent during all calls to* resize().

10.2 MeldableHeap: A Randomized Meldable Heap

In this section, we describe the MeldableHeap, a priority Queue implementation in which the underlying structure is also a heap-ordered binary tree. However, unlike a BinaryHeap in which the underlying binary tree is completely defined by the number of elements, there are no restrictions on the shape of the binary tree that underlies a MeldableHeap; anything goes.

The add(x) and remove() operations in a MeldableHeap are implemented in terms of the merge(h1,h2) operation. This operation takes two heap nodes h1 and h2 and merges them, returning a heap node that is the root of a heap that contains all elements in the subtree rooted at h1 and all elements in the subtree rooted at h2.

The nice thing about a merge(h1,h2) operation is that it can be defined recursively. See Figure 10.4. If either h1 or h2 is nil, then we are merging with an empty set, so we return h2 or h1, respectively. Otherwise, assume $h1.x \leq h2.x$ since, if $h1.x > h2.x$, then we can reverse the roles of h1 and h2. Then we know that the root of the merged heap will contain h1.x, and we can recursively merge h2 with h1.left or h1.right, as we wish. This is where randomization comes in, and we toss a coin to decide whether to merge h2 with h1.left or h1.right:

MeldableHeap

```
Node<T> merge(Node<T> h1, Node<T> h2) {
  if (h1 == nil) return h2;
```

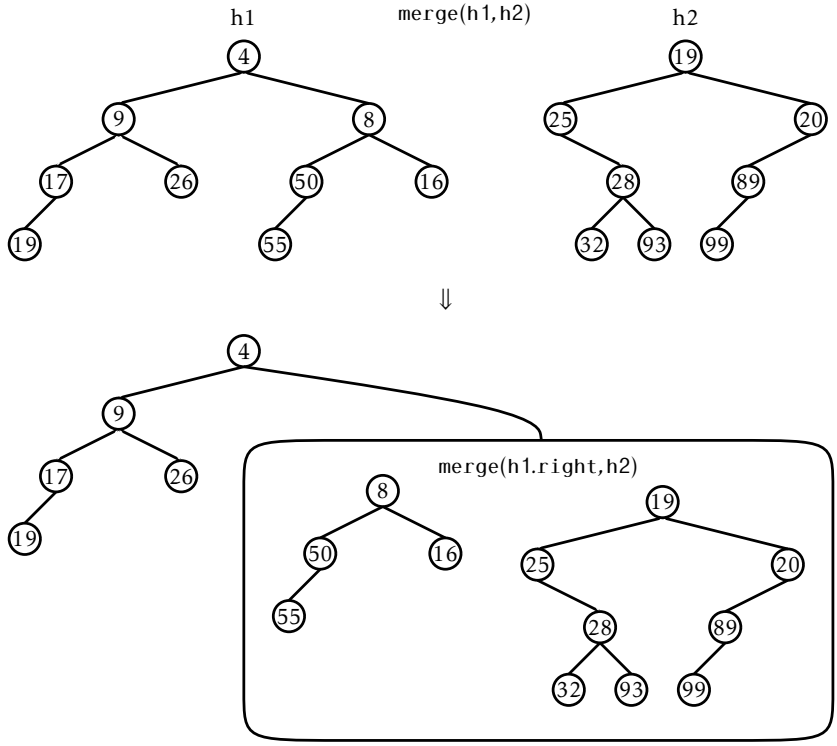

Figure 10.4: Merging `h1` and `h2` is done by merging `h2` with one of `h1.left` or `h1.right`.

```
  if (h2 == nil) return h1;
  if (compare(h2.x, h1.x) < 0) return merge(h2, h1);
  // now we know h1.x <= h2.x
  if (rand.nextBoolean()) {
    h1.left = merge(h1.left, h2);
    h1.left.parent = h1;
  } else {
    h1.right = merge(h1.right, h2);
    h1.right.parent = h1;
  }
  return h1;
}
```

In the next section, we show that `merge(h1,h2)` runs in $O(\log n)$ ex-

pected time, where n is the total number of elements in h1 and h2.

With access to a merge(h1,h2) operation, the add(x) operation is easy. We create a new node u containing x and then merge u with the root of our heap:

MeldableHeap
```
boolean add(T x) {
  Node<T> u = newNode();
  u.x = x;
  r = merge(u, r);
  r.parent = nil;
  n++;
  return true;
}
```

This takes $O(\log(n+1)) = O(\log n)$ expected time.

The remove() operation is similarly easy. The node we want to remove is the root, so we just merge its two children and make the result the root:

MeldableHeap
```
T remove() {
  T x = r.x;
  r = merge(r.left, r.right);
  if (r != nil) r.parent = nil;
  n--;
  return x;
}
```

Again, this takes $O(\log n)$ expected time.

Additionally, a MeldableHeap can implement many other operations in $O(\log n)$ expected time, including:

- remove(u): remove the node u (and its key u.x) from the heap.
- absorb(h): add all the elements of the MeldableHeap h to this heap, emptying h in the process.

Each of these operations can be implemented using a constant number of merge(h1,h2) operations that each take $O(\log n)$ expected time.

10.2.1 Analysis of merge(h1,h2)

The analysis of merge(h1,h2) is based on the analysis of a random walk in a binary tree. A *random walk* in a binary tree starts at the root of the tree. At each step in the random walk, a coin is tossed and, depending on the result of this coin toss, the walk proceeds to the left or to the right child of the current node. The walk ends when it falls off the tree (the current node becomes nil).

The following lemma is somewhat remarkable because it does not depend at all on the shape of the binary tree:

Lemma 10.1. *The expected length of a random walk in a binary tree with* n *nodes is at most* $\log(n+1)$.

Proof. The proof is by induction on n. In the base case, $n = 0$ and the walk has length $0 = \log(n+1)$. Suppose now that the result is true for all non-negative integers $n' < n$.

Let n_1 denote the size of the root's left subtree, so that $n_2 = n - n_1 - 1$ is the size of the root's right subtree. Starting at the root, the walk takes one step and then continues in a subtree of size n_1 or n_2. By our inductive hypothesis, the expected length of the walk is then

$$\mathrm{E}[W] = 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \ ,$$

since each of n_1 and n_2 are less than n. Since log is a concave function, $\mathrm{E}[W]$ is maximized when $n_1 = n_2 = (n-1)/2$. Therefore, the expected number of steps taken by the random walk is

$$\begin{aligned}\mathrm{E}[W] &= 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \\ &\le 1 + \log((n-1)/2+1) \\ &= 1 + \log((n+1)/2) \\ &= \log(n+1) \ .\end{aligned}$$

□

We make a quick digression to note that, for readers who know a little about information theory, the proof of Lemma 10.1 can be stated in terms of entropy.

Information Theoretic Proof of Lemma 10.1. Let d_i denote the depth of the ith external node and recall that a binary tree with n nodes has $n+1$ external nodes. The probability of the random walk reaching the ith external node is exactly $p_i = 1/2^{d_i}$, so the expected length of the random walk is given by

$$H = \sum_{i=0}^{n} p_i d_i = \sum_{i=0}^{n} p_i \log\left(2^{d_i}\right) = \sum_{i=0}^{n} p_i \log(1/p_i)$$

The right hand side of this equation is easily recognizable as the entropy of a probability distribution over $n+1$ elements. A basic fact about the entropy of a distribution over $n+1$ elements is that it does not exceed $\log(n+1)$, which proves the lemma. □

With this result on random walks, we can now easily prove that the running time of the `merge(h1,h2)` operation is $O(\log n)$.

Lemma 10.2. *If* `h1` *and* `h2` *are the roots of two heaps containing* n_1 *and* n_2 *nodes, respectively, then the expected running time of* `merge(h1,h2)` *is at most* $O(\log n)$*, where* $n = n_1 + n_2$.

Proof. Each step of the merge algorithm takes one step of a random walk, either in the heap rooted at `h1` or the heap rooted at `h2`. The algorithm terminates when either of these two random walks fall out of its corresponding tree (when `h1 = null` or `h2 = null`). Therefore, the expected number of steps performed by the merge algorithm is at most

$$\log(n_1+1) + \log(n_2+1) \le 2\log n \ .$$ □

10.2.2 Summary

The following theorem summarizes the performance of a `MeldableHeap`:

Theorem 10.2. *A* `MeldableHeap` *implements the (priority)* `Queue` *interface. A* `MeldableHeap` *supports the operations* `add(x)` *and* `remove()` *in* $O(\log n)$ *expected time per operation.*

10.3 Discussion and Exercises

The implicit representation of a complete binary tree as an array, or list, seems to have been first proposed by Eytzinger [27]. He used this representation in books containing pedigree family trees of noble families. The `BinaryHeap` data structure described here was first introduced by Williams [78].

The randomized `MeldableHeap` data structure described here appears to have first been proposed by Gambin and Malinowski [34]. Other meldable heap implementations exist, including leftist heaps [16, 48, Section 5.3.2], binomial heaps [75], Fibonacci heaps [30], pairing heaps [29], and skew heaps [72], although none of these are as simple as the `MeldableHeap` structure.

Some of the above structures also support a `decreaseKey(u,y)` operation in which the value stored at node `u` is decreased to `y`. (It is a precondition that $y \leq u.x$.) In most of the preceding structures, this operation can be supported in $O(\log n)$ time by removing node `u` and adding `y`. However, some of these structures can implement `decreaseKey(u,y)` more efficiently. In particular, `decreaseKey(u,y)` takes $O(1)$ amortized time in Fibonacci heaps and $O(\log\log n)$ amortized time in a special version of pairing heaps [25]. This more efficient `decreaseKey(u,y)` operation has applications in speeding up several graph algorithms, including Dijkstra's shortest path algorithm [30].

Exercise 10.1. Illustrate the addition of the values 7 and then 3 to the `BinaryHeap` shown at the end of Figure 10.2.

Exercise 10.2. Illustrate the removal of the next two values (6 and 8) on the `BinaryHeap` shown at the end of Figure 10.3.

Exercise 10.3. Implement the `remove(i)` method, that removes the value stored in `a[i]` in a `BinaryHeap`. This method should run in $O(\log n)$ time. Next, explain why this method is not likely to be useful.

Exercise 10.4. A d-ary tree is a generalization of a binary tree in which each internal node has d children. Using Eytzinger's method it is also possible to represent complete d-ary trees using arrays. Work out the

equations that, given an index i, determine the index of i's parent and each of i's d children in this representation.

Exercise 10.5. Using what you learned in Exercise 10.4, design and implement a *DaryHeap*, the d-ary generalization of a BinaryHeap. Analyze the running times of operations on a DaryHeap and test the performance of your DaryHeap implementation against that of the BinaryHeap implementation given here.

Exercise 10.6. Illustrate the addition of the values 17 and then 82 in the MeldableHeap h1 shown in Figure 10.4. Use a coin to simulate a random bit when needed.

Exercise 10.7. Illustrate the removal of the next two values (4 and 8) in the MeldableHeap h1 shown in Figure 10.4. Use a coin to simulate a random bit when needed.

Exercise 10.8. Implement the remove(u) method, that removes the node u from a MeldableHeap. This method should run in $O(\log n)$ expected time.

Exercise 10.9. Show how to find the second smallest value in a BinaryHeap or MeldableHeap in constant time.

Exercise 10.10. Show how to find the kth smallest value in a BinaryHeap or MeldableHeap in $O(k \log k)$ time. (Hint: Using another heap might help.)

Exercise 10.11. Suppose you are given k sorted lists, of total length n. Using a heap, show how to merge these into a single sorted list in $O(n \log k)$ time. (Hint: Starting with the case $k = 2$ can be instructive.)