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)` =

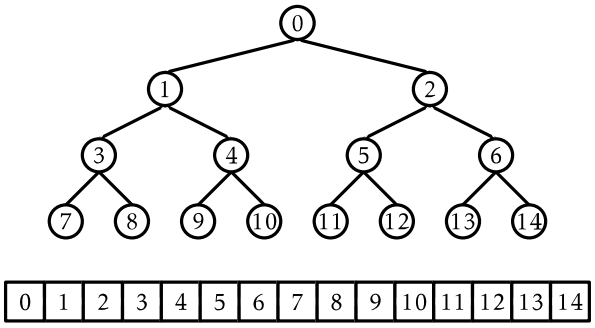

Figure 10.1: Eytzinger's method represents a complete binary tree as an array.

$2i+1$ 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);
```

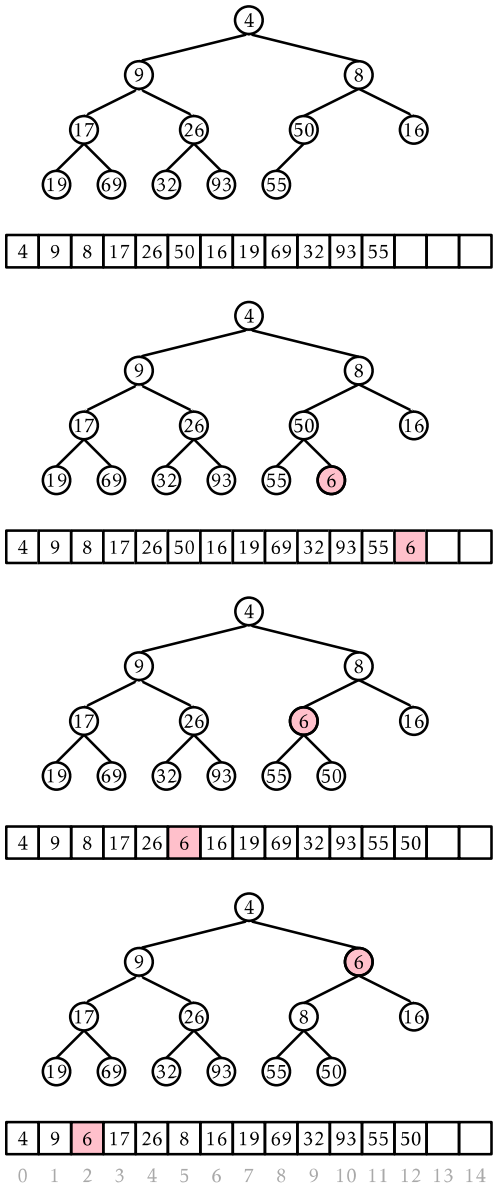

Figure 10.2: Adding the value 6 to a `BinaryHeap`.

```
  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.

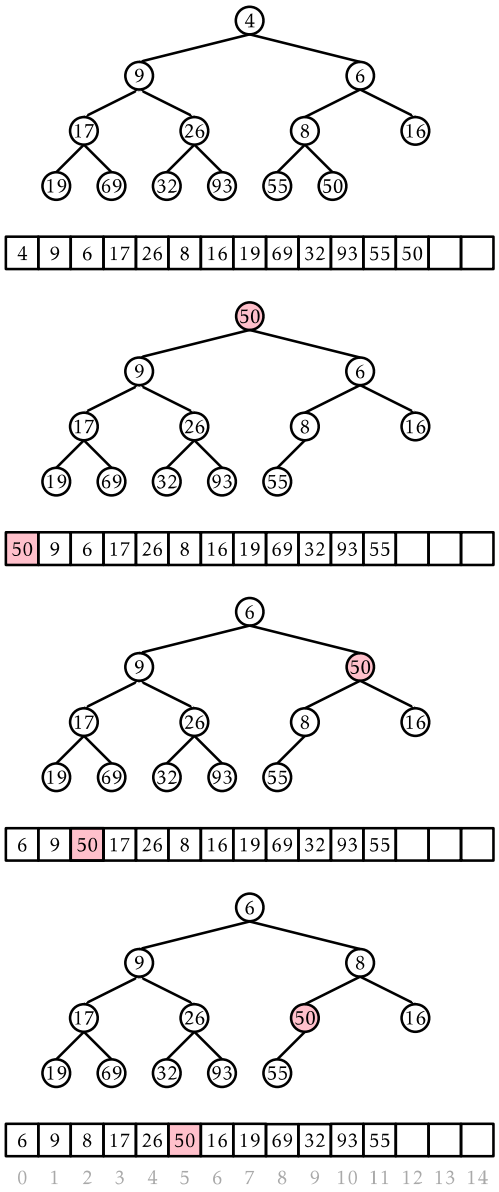

Figure 10.3: Removing the minimum value, 4, from a `BinaryHeap`.

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;
```