Chapter 12

Case study: Heap Sort

Our earlier sorting case study, Insertion Sort in Chapter 10, took time proportional to the square of the number of elements sorted. The code developed in this chapter, Heap Sort, does considerably better: if N elements are to be sorted, then the time taken is proportional to $N \log N$.

That alone is worth the extra trouble needed to develop Heap Sort; but it is also worth seeing how radically different code can be reached from the same initial specification.

12.1 Time complexity of code

The outer iteration of Insertion Sort is executed N times: thus its inner iteration, on average, is executed $k/2$ times for each k from 0 to $N-1$. Overall, therefore, the code performs this many comparisons (one for each inner iteration):

$$\frac{1}{2}(0+1+2+\cdots+N-1) \quad = \quad N(N-1)/4 \; .$$

Thus if the sequence is doubled in length, the time taken to sort it is roughly quadrupled.

In general, the number of operations required to execute code is expressed as some function of the size N of the data given to it. For two such functions f and g, we write $\mathsf{f} \preceq \mathsf{g}$ if there are two numbers $M : \mathbb{N}$ and $c : \mathbb{R}^+$ such that for all $N \geq M$,

$$\mathsf{f}\,N \leq c \times \mathsf{g}\,N \; .$$

Put informally, that reads 'up to a constant factor $(c\times)$, the function f is less than or equal to g for sufficiently large arguments $(\geq M)$'. We write $\mathsf{f} \succeq \mathsf{g}$ if $\mathsf{g} \preceq \mathsf{f}$, and $\mathsf{f} \approx \mathsf{g}$ if both $\mathsf{f} \preceq \mathsf{g}$ and $\mathsf{f} \succeq \mathsf{g}$, and finally $\mathsf{f} \prec \mathsf{g}$ if $\mathsf{f} \preceq \mathsf{g}$ but $\mathsf{f} \not\approx \mathsf{g}$. Thus in this case we have

$$N(N-1)/4 \approx N^2 \; ,$$

and we say that Insertion Sort has N^2, or *quadratic*, time complexity.

Heap Sort has complexity $N \log N$, and is therefore asymptotically more efficient than Insertion Sort, since

$$N \log N \prec N^2 \ .$$

Even better, it can be shown that $N \log N$ is the lowest complexity possible[1] for sorting sequences: in that sense, Heap Sort cannot be improved. Its complexity is *optimal.*

With that background, we now present our abstract program, exactly the same as for Insertion Sort:

$$\begin{array}{l} \textbf{var}\ as : \textbf{seq}_N\, \mathbb{Z};\ \ \textbf{con}\ A; \\ \textbf{and}\ A = \mathsf{bag}\, as\, \bullet \\ as\colon [\mathsf{up}\, as]\ . \end{array}$$

But our development is completely different.

12.2 Heaps

Heap Sort achieves its better-than-quadratic $N \log N$ complexity by sorting a *balanced tree*, rather than a sequence, of values. That is where the $\log N$ comes from in its complexity $N \log N$: a balanced binary tree of N nodes has depth $\lceil \log N \rceil$. (The N factor comes from the fact that it has N nodes.)

Surprisingly, it is possible to lay a balanced tree out entirely within our given sequence as. The arrangement is easily observed in a sequence numbered from 1: in that case, element $as[i]$ is the *parent* of element $as[j]$ exactly when $i = j \div 2$. And element $as[i]$ is an *ancestor* of element $as[j]$ exactly when $\left(\exists\, k \bullet i = j \div 2^k\right)$. *Child* and *descendant* are the complementary terms; note that an element is both an ancestor and a descendant of itself.

Our sequences begin at 0, and so we define a relation $\leadsto$ as follows: for $i, j : \mathbb{N}$,

$$i \leadsto j \ \ \widehat{=}\ \ \left(\exists\, k : \mathbb{N} \bullet i + 1 = (j+1) \div 2^k\right).$$

The relation is illustrated in Figure 12.1.

We say that a sequence as is a *heap* whenever the balanced binary tree within it is *ordered* so that ancestors are no smaller than their descendants: for that we write $\mathsf{hp}\, as$, where

$$\mathsf{hp}\, as \ \ \widehat{=}\ \ (\forall\, i, j : 0 \rightarrow \#as \bullet i \leadsto j \Rightarrow as[i] \geq as[j]) \ .$$

That means incidentally that for any non-empty heap as, its first element is its largest: thus $\mathsf{hd}\, as \geq \mathsf{tl}\, as$.

[1] Better performance is possible, however, if several computers are used at once.

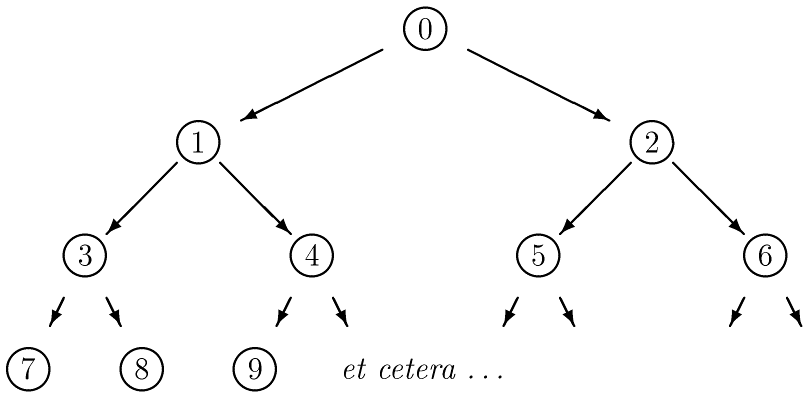

Figure 12.1 Parent–child relation in a heap

12.3 Shrinking a heap

Our strategy is to form the sequence into a heap, and then sort that; thus the first step is

$$\begin{array}{ll} \sqsubseteq & as\colon [\mathsf{hp}\ as]\ ; & \text{(i)} \\ & as\colon [\mathsf{hp}\ as\ ,\ \mathsf{up}\ as]\ . & \text{(ii)} \end{array}$$

Note that we have diverged immediately from the path taken earlier with Insertion Sort. Motivation for it is the suspicion that sorting a heap is faster than sorting a sequence; naturally we hope that our advantage is not eaten up by the cost of (i), the making of the heap in the first place.

Since (ii) has more detail, we work first on it. The operational idea is to pluck successive greatest elements from the root of the heap, at $as[0]$, reforming the heap each time. (There are N elements, and each reforming of the heap takes no more than $\log N$ comparisons.)

Using the form of (ii) as a guide, we bring its pre- and postcondition together by introducing up on the left and hp on the right:

$$\text{(ii)} \quad \sqsubseteq \quad as\colon [\mathsf{hp}\ as{\uparrow}N \wedge \mathsf{up}\ as{\downarrow}N\ ,\ \mathsf{hp}\ as{\uparrow}0 \wedge \mathsf{up}\ as{\downarrow}0]\ .$$

Rushing into an iteration here may be premature, however. The specification above suggests shrinking the heap towards 0, leaving sorted elements in a suffix behind it. That suffix should contain elements which need not be moved again: they should be in their final positions. We add that to the invariant, just to be sure,[2] and now proceed

[2]That sounds a bit imprecise: it means only that if we had not added it, we would have been stuck later on. It does *not* mean that if we had not added it we would have developed incorrect code.

$\sqsubseteq$ $\mathsf{fn}\, i \;\widehat{=}\; as{\uparrow}i \le as{\downarrow}i \wedge \mathsf{up}\, as{\downarrow}i \cdot$

as: $[\mathsf{hp}\, as{\uparrow}N \wedge \mathsf{fn}\, N\ ,\ \mathsf{hp}\, as{\uparrow}0 \wedge \mathsf{fn}\, 0]$

$\sqsubseteq$ **var** $i : \mathbb{N} \cdot$

$i := N;$

as, i: $[i = N\ ,\ \mathsf{hp}\, as{\uparrow}i \wedge \mathsf{fn}\, i\ ,\ i = 0]$ $\triangleleft$

$\sqsubseteq$ "iterate down"

do $i \neq 0 \rightarrow$

  $i := i \ominus 1;$

  as: $[i < N \wedge \mathsf{hp}\, as{\uparrow}(i+1) \wedge \mathsf{fn}(i+1)\ ,\ \mathsf{hp}\, as{\uparrow}i \wedge \mathsf{fn}\, i]$ $\triangleleft$

od .

In the precondition we have $\mathsf{fn}(i+1)$, and in the postcondition we have $\mathsf{fn}\, i$: therefore we need to place the maximum $as[0]$ of $as{\uparrow}(i+1)$ at index i. But $as[i]$ must go somewhere — perhaps to index 0. Unfortunately, that does not establish the remaining conjunct $\mathsf{hp}\, as{\uparrow}i$: the new $as[0]$ is more likely to be the minimum than the maximum of $a{\uparrow}i$!

As for Insertion Sort, we need to say 'as is ... except at 0'. Just $\mathsf{hp}(\mathsf{tl}\, as)$ will not do, since the ancestor relation in $\mathsf{tl}\, as$ is quite different. Thus we must define a *partial heap* as follows: for $s \subseteq 0{\rightarrow}N$,

$$\mathsf{ph}\, s \;\widehat{=}\; (\forall\, i, j : s \cdot i \rightsquigarrow j \Rightarrow as[i] \ge as[j])\,.$$

Remember that $0{\rightarrow}N$ does not contain N (p.79), and note that $\mathsf{ph}\, 0{\rightarrow}N \equiv \mathsf{hp}\, as$.

Now we can proceed by swapping elements i and 0:

$\sqsubseteq$ *Swap* $(as, 0, i)$;

as: $[\mathsf{ph}\, 1{\rightarrow}i\ ,\ \mathsf{fn}\, i\ ,\ \mathsf{ph}\, 0{\rightarrow}i]$. (iii)

Specification (iii) requires us to 'repair the heapiness' of $as[0{\rightarrow}i]$, without disturbing the elements at i and beyond.

(It is that last step that needed $\mathsf{fn}\, i$ in the invariant rather than the weaker $\mathsf{up}\, as{\downarrow}i$: having only $\mathsf{up}\, as{\downarrow}(i+1)$ is not sufficient for a *Swap* $(as, 0, i)$ to establish $\mathsf{up}\, as{\downarrow}i$ even when $as[0] \ge as{\uparrow}(i+1)$.)

12.4 Establishing the heap

Now we return to (i): in fact specification (iii) above suggests an approach, since it extends a heap towards lower indices. We are further encouraged by the fact that $\mathsf{ph}(N \div 2){\rightarrow}N$ is identically true (since there are no ancestors to consider): any sequence is 'half a heap' already. So we proceed

(i) $\sqsubseteq$ **var** $j : \mathbb{N} \cdot$

$j := N \div 2;$

as, j: $[j = N \div 2\ ,\ \mathsf{ph}\, j{\rightarrow}N\ ,\ j = 0]$ $\triangleleft$

$$
\begin{array}{l}
\sqsubseteq \text{ “iterate down”} \\
\quad \textbf{do } j \neq 0 \rightarrow \\
\qquad j := j \ominus 1; \\
\qquad as\colon [j < N \div 2 \wedge \mathsf{ph}(j+1){\rightarrow}N \ , \ \mathsf{ph}\, j{\rightarrow}N] \qquad\qquad \text{(iv)} \\
\quad \textbf{od} \ .
\end{array}
$$

Specification (iv) looks very much like (iii); and since $\mathsf{fn}\, N$ is identically true, we can bring the two closer still, as follows:

$$
\text{(iv)} \ \sqsubseteq \ as\colon [\mathsf{ph}(j+1){\rightarrow}N \ , \ \mathsf{fn}\, N \ , \ \mathsf{ph}\, j{\rightarrow}N] \ . \qquad\qquad \text{(v)}
$$

We are now ripe for a procedure.

12.5 Procedure *Sift*

The similarity of Specifications (iii) and (v) suggests a procedure defined as follows:

$$
\begin{array}{l}
\textbf{procedure } \mathit{Sift} \ (\textbf{value } l, h : \mathbb{N}) \ \widehat{=} \\
\quad as\colon [\mathsf{ph}(l+1){\rightarrow}h \ , \ \mathsf{fn}\, h \ , \ \mathsf{ph}\, l{\rightarrow}h] \ ,
\end{array}
$$

and that gives the following refinements immediately:

$$
\begin{array}{rl}
\text{(iii)} & \sqsubseteq \ \mathit{Sift} \ (0, i) \\
\text{(v)} & \sqsubseteq \ \mathit{Sift} \ (j, N) \ .
\end{array}
$$

We have used value substitution for both parameters.

Now of course we still must develop *Sift* itself; but at least we need do that only once, and not twice as would have been necessary had we treated (iii) and (v) separately.

The invariant $\mathsf{fn}\, h$ of *Sift* suggests that we confine our attention to the prefix $0{\rightarrow}h$ of as, just leaving the suffix from h on alone. The pre- and postcondition suggest that we need to say “as is a partial heap $l{\rightarrow}h$ at some k as well”, and so we define

$$
\begin{array}{lcl}
\mathsf{lo}\, k & \widehat{=} & (\forall\, i : l{\rightarrow}h \cdot i \leadsto k \Rightarrow as[i] \geq as[k]) \\
\mathsf{hi}\, k & \widehat{=} & (\forall\, j : l{\rightarrow}h \cdot k \leadsto j \Rightarrow as[k] \geq as[j]) \ ,
\end{array}
$$

so that $\mathsf{lo}\, k$ means that element k is correctly ordered with respect to elements below it, and $\mathsf{hi}\, k$ means the same for elements above it.

Then we continue as we did for Insertion Sort:

$$
\begin{array}{l}
\quad \mathit{Sift} \\
\sqsubseteq \left\{ \begin{array}{l} \textbf{var } k : \mathbb{N} \\ J \ \widehat{=} \ \mathsf{ph}(l{\rightarrow}h - \{k\}) \wedge \mathsf{lo}\, k \wedge \mathsf{fn}\, h \cdot \end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
k := l; \\
\textbf{do}\ \neg\,\mathsf{hi}\,k \rightarrow \\
\qquad as, k\colon [\neg\,\mathsf{hi}\,k\ ,\ J\ ,\ k_0 < k \le h] \qquad\qquad \triangleleft \\
\textbf{od}\ .
\end{array}
$$

Now if $\neg\,\mathsf{hi}\,k$ holds, we know two things:

1. $as[k]$ has a child in $l \rightarrow h$ (otherwise, $\mathsf{hi}\,k$ would be vacuously true); hence $2k + 1 < h$.
2. $as[k]$ is less than one of its children; certainly it is less than the greater of its children.

Accordingly, we make our final refinement by swapping $as[k]$ with the greater of its children, say $as[m]$. Since that establishes $\mathsf{ph}(l \rightarrow h - \{m\})$, we also assign m to k (thus increasing k, but not beyond h).

$$
\begin{array}{ll}
\sqsubseteq & \textbf{var}\ m : \mathbb{N}\bullet \\
& \textbf{if}\ 2k + 2 = h \rightarrow m := 2k + 1 \\
& [\!]\ \ 2k + 2 < h \wedge as[2k + 1] \ge as[2k + 2] \rightarrow m := 2k + 1 \\
& [\!]\ \ 2k + 2 < h \wedge as[2k + 1] \le as[2k + 2] \rightarrow m := 2k + 2 \\
& \textbf{fi}; \\
& \mathit{Swap}\ (as, k, m); \\
& k := m\ .
\end{array}
$$

That concludes the development of *Sift*, and thus of Heap Sort itself.

12.6 Exercises

Ex. 12.1 Prove that $\mathsf{hp}\,as \Rrightarrow \mathsf{hd}\,as \ge \mathsf{tl}\,as$, if as is not empty.

Ex. 12.2 ♡ Use the invariant J to code the guard $\neg\,\mathsf{hi}\,k$. *Hint*: Recall Exercise 10.4.

Ex. 12.3 Make *Sift* more efficient, as we did for Insertion Sort.

Ex. 12.4 ♡ Show that if

$$\lim_{N \rightarrow \infty} (\mathsf{f}\,N / \mathsf{g}\,N) = 0,$$

then $\mathsf{f} \prec \mathsf{g}$.

Ex. 12.5 Show that $N \log N \prec N^2$.

Ex. 12.6 Show that $\log_a N \approx \log_b N$ for any logarithmic bases $a, b > 1$.

Ex. 12.7 Prove that *polynomial* complexity is always better than *exponential* complexity by showing that, for all $e \geq 0$ (no matter how large) and all $b > 1$ (no matter how small),

$$N^e \prec b^N \ .$$

Hint: Use l'Hôpital's Rule.

Ex. 12.8 ♡ The conventional notation for $\mathsf{f} \preceq \mathsf{g}$ is $\mathsf{f} = O(\mathsf{g})$, and for $\mathsf{f} \succeq \mathsf{g}$ it is $\mathsf{f} = \Omega(g)$. Finally, $\mathsf{f} \approx \mathsf{g}$ is written $\mathsf{f} = \Theta(\mathsf{g})$. What makes those notations unusual? *Hint*: Consider the properties of equality.

How could the conventional notations be improved? *Hint*: Suppose $O(\mathsf{f})$ were a *set* of functions; how would $\mathsf{f} \preceq \mathsf{g}$ be written then?

Ex. 12.9 Give two functions f and g such that neither $\mathsf{f} \preceq \mathsf{g}$ nor $\mathsf{g} \preceq \mathsf{f}$.

Ex. 12.10 ♡ In spite of its optimal complexity, Heap Sort is not the fastest sorting code. (Quick Sort, for example, is considerably faster in most cases.) How can anything be faster than optimal?

Ex. 12.11 *Bubble Sort* Using the definitions of this chapter, complete the following development:

$$
\begin{array}{ll}
 & as\colon [\mathsf{up}\ as] \\
\sqsubseteq & \mathbf{var}\ i : \mathbb{N} \\
 & i := N; \\
 & as, i\colon [i = N\ ,\ \mathsf{fn}\, i\ ,\ i = 0]\ .
\end{array}
$$

Hint: Imagine Heap Sort with no heap: in the inner iteration, you must nevertheless establish $as{\uparrow}i \leq as[i]$.