programming pearls

by Jon Bentley

THANKS, HEAPS

This column is about "heaps," a data structure that we'll use to solve two problems.

Sorting. Heapsort sorts an N-element array in $O(N \log N)$ time and uses just a few words of extra space.

Priority Queues. Heaps maintain a set of elements under the operations of inserting new elements and extracting the smallest element in the set; each operation requires $O(\log N)$ time.

For both problems, heaps are simple to code and computationally efficient.

Now that the advertising is out of the way, I can tell you the real reason that I'm writing this column: heaps have bothered me for a dozen years. The basic idea is elegant, and the data structure is eminently practical. I've implemented heaps several times in application programs. But I never felt satisfied with my code: it was lengthy and clumsy, fraught with special cases and bugs. Although I didn't know what to do about it, I was deeply upset that a beautiful idea ended up as an ugly program.

Several weeks ago I taught a three-day course on "The Science of Programming," using David Gries's excellent text of that title. As the last exercise in the course, we used the techniques of program verification to write code for heaps. Imagine my delight when the result was a short, clean, elegant program! This column is my attempt to communicate three things: the fundamental idea of heaps, a sweet program that implements the idea, and the powerful programming techniques that lead from the former to the latter.

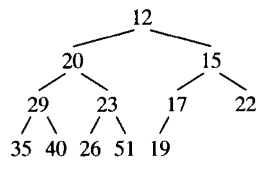
This column has a "bottom-up" organization: we'll start at the details and work up to the big picture. The next two sections describe the heap data structure and two routines to operate on it. The two subsequent sections use those tools to solve the problems mentioned above.

The Data Structure

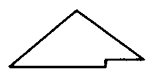
A heap is a data structure for representing a collection of items.[1] Our examples will represent numbers, but the elements in a heap may be of any ordered type.

[1] In other computing contexts, the word "heap" refers to a large segment of memory from which variable-size nodes are allocated; we'll ignore that interpretation in this column.

Here's a heap of 12 integers:

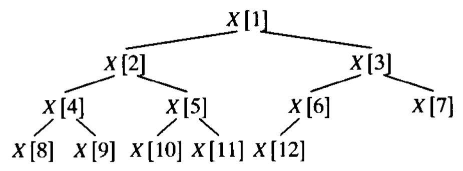


That binary tree is a heap by virtue of two properties. We'll call the first property *Order*: the value at any node is less than or equal to the values of the node's children. This implies that the least element of the set is at the root of the tree (12 in the example), but it doesn't say anything about the relative order of left and right children. The second heap property is *Shape*; the idea is captured by the picture



In words, a binary tree with the *Shape* property has its terminal nodes on at most two levels, with those on the bottom level as far left as possible. There are no "holes" in the tree; if it contains N nodes, no node is of distance more than $\log_2 N$ from the root. We'll soon see how the two properties together are restrictive enough to allow us to find the minimum element in a set, but lax enough so that we can efficiently reorganize the structure after inserting or deleting an element.

Let's turn now from the abstract properties of heaps to their implementation. There are, of course, many possible representations of binary trees, such as records and pointers. We'll use an implementation that is suitable only for binary trees with the *Shape* property, but is quite effective for that special case. A 12-element tree with *Shape* is implemented in the 12-element array X as follows:

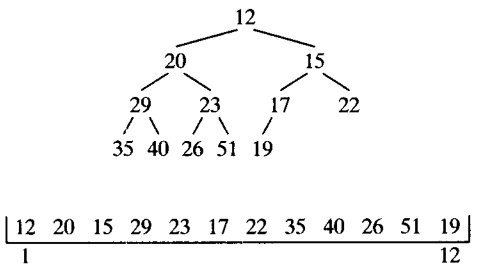


In this *implicit* representation of a binary tree, the root is in X[1], its two children are in X[2] and X[3], and so on. Various functions on the tree are defined as follows.

```
root = 1
value(I) = X[I]
leftchild(I) = 2*I
rightchild(I) = 2*I+1
parent(I) = I div 2
null(I) = (I<1) or (I>N)
```

An N-element implicit tree necessarily has the *Shape* property: there is no provision for missing elements.

This picture shows a 12-element heap and its implementation as an implicit tree in a 12-element array.



Because the *Shape* property is guaranteed by the representation, from now on we'll use the name *Heap* to mean that the value in any node is greater than or equal to the value in its parent. Phrased precisely, the array X[1..N] has the *Heap* property if

$$\forall_{2 \le i \le N}\ X[i \text{ div } 2] \le X[i]$$

In the next section we'll want to talk about X[L..U] having the heap property; we'll define *Heap*(L, U) as

$$\forall_{2L \le i \le U}\ X[i \text{ div } 2] \le X[i]$$

Two Critical Routines

In this section we'll study two routines for fixing an array whose *Heap* property has been broken at one end or the other. Both routines are efficient: they require roughly log N time to reorganize a heap of N elements. In the bottom-up spirit of this column, we'll define the routines here and then use them in the next sections.

Placing an arbitrary element in X[N] when X[1..$N-1$] is a heap will probably not yield *Heap*(1, N); establishing the property is the job of procedure *SiftUp*. Its name describes its strategy: we sift the new element up the tree as far as it should go, swapping it with its parent along the way. (Take a second to contemplate which way is up: the root of the heap is at the top of the tree, and therefore X[N] is at the bottom of the array.) The process is illustrated in Figure 1, which shows the new element 13 being sifted up the heap until it is at its proper position as the right child of the root. The process continues until the circled node is greater than or equal to its parent (as in Figure 1) or it is at the root of the tree. If the process starts with *Heap*(1, $N-1$) true, it leaves *Heap*(1, N) true.

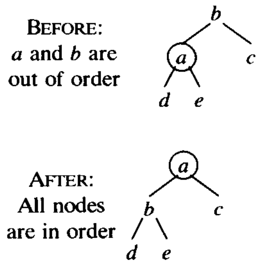
With that intuitive background, let's write the code. The sifting process calls for a loop, so we'll start with the loop invariant. In Figure 1, the heap property holds everywhere in the tree except between the circled node and its parent. If we let I be the index of the circled node, then we can use the invariant

```
loop
    /* Inv: Heap(1,N) except (perhaps)
       between I and its parent */
```

Because we originally have *Heap*(1, $N-1$), we initialize the loop by the assignment `I:=N`.

The job of the loop is to check whether we have finished yet (either by the circled node being at the top of the heap or greater than or equal to its parent) and, if not, to make progress towards termination. The invariant says that the *Heap* property holds everywhere except (perhaps) between I and its parent. If the test $I = 1$ is true, then I has no parent and the *Heap* property thus holds everywhere; the loop may therefore terminate. When I does have a parent, we'll let P be the parent's index by assigning `P := I div 2`. If X[P] ≤ X[I] then the heap property holds everywhere, and the loop may terminate.

If, on the other hand, I is out of order with its parent, then we swap X[I] and X[P]. This step is illustrated in the following picture, in which node I is circled.



After the swap, all five elements are in the proper order: $b < d$ and $b < e$ because b was originally higher in the heap, $a < b$ because the test X[P] ≤ X[I] failed, and $a < c$ by combining $a < b$ and $b < c$. This gives the heap property everywhere in the array except (possibly) be-

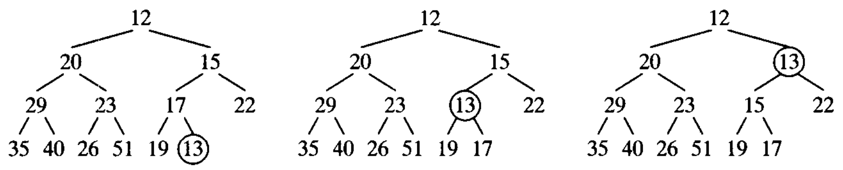


FIGURE 1. Element 13 is Sifted Up the Heap (left to right)

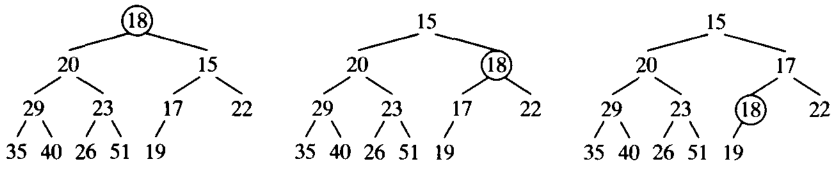


FIGURE 2. Element 18 is Sifted Down the Heap (left to right)

tween *P* and its parent; we therefore reestablish the invariant by assigning `I:=P`.

The above pieces are assembled in Program 1, which runs in time proportional to log *N* because the heap has that many levels. The "pre" and "post" lines characterize the procedure: if the pre-condition is true before the routine is called then the post-condition will be true afterwards.

```
proc SiftUp(N)
        pre   Heap(1,N-1) and N>0
        post  Heap(1,N)
I := N
loop
        /* Inv: Heap(1,N) except (perhaps)
                between I and its parent */
        if I=1 then break
        P := I div 2
        if X[P] <= X[I] then break
        Swap(X[P], X[I])
        I := P
    endloop
```

PROGRAM 1. Procedure *SiftUp*

Assigning a new value to X[1] when X[1..*N*] is a heap leaves *Heap*(2, *N*); Procedure *SiftDown's* job is to make *Heap*(1, *N*) true. It does so by sifting X[1] down the array until either it has no children or it is less than or equal to the children it does have. Figure 2 shows 18 being sifted down the heap until it is finally less than its single child, 19. When an element is sifted up, it always goes toward the root. Sifting down is more complicated: an out-of-order element is swapped with its lesser child.

The pictures illustrate the invariant of the *SiftDown* loop: the heap property holds everywhere except, possibly, between the circled node and its children.

```
loop
    /* Inv: Heap(1,N) except
       (perhaps)
       between I and its
       (0, 1 or 2) children */
```

The loop is similar to *SiftUp*'s. We first check whether *I* has any children, and terminate the loop if it has none. Now comes the subtle part: if *I* does have (one or two) children, then we set the variable *C* to index the least child of *I*. Finally, we either terminate the loop if X[*I*] ≤ X[C], or progress towards the bottom by swapping X[*I*] and X[C] and assigning `I:=C`.

The complete *SiftDown* routine is presented as Program 2. A case analysis like that done for *SiftUp* shows that the swap operation leaves the heap property true everywhere except (possibly) between *C* and its children. Like *SiftUp*, this procedure takes time proportional to log *N*, because it does a fixed amount of work at each level of the heap.

```
proc SiftDown(N)
        pre   Heap(2,N) and N>=0
        post  Heap(1,N)
I := 1
loop
        /* Inv: Heap(1,N) except (perhaps) between
                I and its (0, 1 or 2) children */
        C := 2*I
        if C>N then break
        /* C is the left child of I */
        if C+1 <= N then
                /* C+1 is the right child of I */
                if X[C+1] < X[C] then
                        C := C+1
        /* C is the least child of I */
        if X[I] <= X[C] then break
        Swap(X[C], X[I])
        I := C
    endloop
```

PROGRAM 2. Procedure *SiftDown*

Priority Queues

There are two sides to any data structure: its *abstraction* tells what it does (a queue maintains a sequence of elements under the operations of insert and extract), while its *implementation* tells how it does it (with an array or linked list, perhaps). We'll start our study of priority queues by specifying their abstract properties, and then turn to implementations.

A priority queue manipulates an initially empty set[2] of elements, which we'll call *S*. The *Insert* procedure inserts a new element into the set; we can define that more precisely in terms of its pre- and post-conditions.

```
proc Insert(T)
    pre   |S| < MaxSize
    post Current S = Original SU{T}
```

Procedure *ExtractMin* deletes the smallest element in the set and returns that value in its single parameter *T*.

```
proc ExtractMin(T)
    pre   |S| > 0
    post Original S = Current SU{T}
              and T=min(Original S)
```

This procedure could, of course, be modified to yield the maximum element, or any extreme element under a total ordering.

Priority queues are useful in many applications. An operating system may use such a structure to represent a set of tasks; they are inserted in an arbitrary order, and the task with highest priority is extracted to be executed. In discrete event simulation, the elements are times of events: the simulation loop extracts the events with the least (next) time, and possibly adds more events to the queue. In both applications the basic priority queue must be augmented to contain additional information beyond the elements in the set; we'll ignore that "implementation detail" in our discussion.

Sequential structures such as arrays or linked lists are obvious candidates for implementing priority queues. If the sequence is sorted it is easy to extract the minimum but hard to insert a new element; the situation is reversed for unsorted structures. Table I shows the performance of the structures on an *N*-element set.

TABLE I. Priority Queue Implementation

Data Structure	Run Times		
	1 Insert	1 ExtractMin	*N* of Each
Sorted Sequence	$O(N)$	$O(1)$	$O(N^2)$
Heaps	$O(\log N)$	$O(\log N)$	$O(N \log N)$
Unsorted Sequence	$O(1)$	$O(N)$	$O(N^2)$

The heap implementation of priority queues provides a middle ground between the two extremes. It represents an *N*-element set in the array $X[1..N]$ with the heap property, where *X* is declared as $X[1..MaxSize]$.

[2] Because the set can contain multiple copies of the same element, we would be more precise to call it a "multiset" or a "bag." The union operator is defined so that {2,3} ∪ {2} = {2,2,3}.

We initialize the set to be empty by assignment `N:=0`. To insert a new element we increment *N* and place the new element in *X*[*N*]. That gives the situation that *SiftUp* was designed to fix: *Heap*(1, *N* − 1). The insertion code is therefore

```
proc Insert(T)
        if N>=MaxSize then error
        N:=N+1; X[N]:=T
        /* Heap(1,N-1) */
        SiftUp(N)
        /* Heap(1,N) */
```

Procedure *ExtractMin* finds the minimum element in the set, deletes it, and restructures the array to have the heap property. Because the array is a heap, the minimum element is in X[1]. The *N* − 1 elements remaining in the set are now in X[2..*N*], which has the heap property; we regain *Heap*(1, *N*) in two steps. We first move X[*N*] to X[1] and decrement *N*; the elements of the set are now in X[1..*N*], and *Heap*(2, *N*) is true. The second step calls *SiftDown*. The code is straightforward.

```
proc ExtractMin(T)
        if N<1 then error
        T := X[1]
        X[1]:=X[N]; N:=N-1
        /* Heap(2,N) */
        SiftDown(N)
        /* Heap(1,N) */
```

Both *Insert* and *ExtractMin* require *O*(log *N*) time when applied to *N*-element heaps.

A Sorting Algorithm

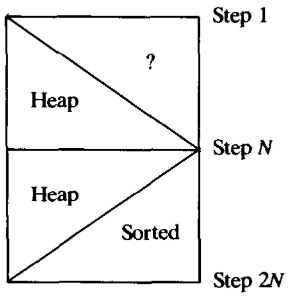
Priority queues provide a simple algorithm for sorting *A*[1..*N*]:

```
for I := 1 to N do
        Insert(A[I])
for I := 1 to N do
        ExtractMin(A[I])
```

The *N Insert* and *ExtractMin* operations have a total cost of *O*(*N* log *N*), while the array X[1..*N*] used for heaps requires *N* additional words of storage.

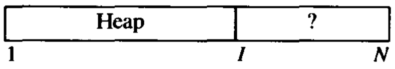
In this section we'll study the Heapsort algorithm, which has several advantages over the obvious approach: it uses less code, it uses less space because it doesn't require the auxiliary array, and it may use less time (see Problem 2). For purposes of this algorithm we'll assume that *SiftUp* and *SiftDown* have been modified to operate on heaps in which the *largest* element is at the top; that is easy to accomplish by swapping "<" and ">" signs.

The simple algorithm uses two arrays, one for the priority queue and one for the elements to be sorted; Heapsort saves space by using just one. The single implementation array *X* represents two abstract structures: a heap at the left end and at the right end the sequence of elements, originally in arbitrary order and finally sorted. This picture shows the evolution of the array X; the array is drawn horizontally, while time marches down the vertical axis.



The Heapsort algorithm is a two-stage process: the first *N* steps build the array into a heap, and the next *N* steps extract the elements in decreasing order and build the final sorted sequence, right to left.

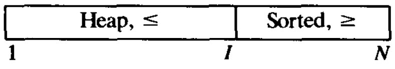
The job of the first stage is to build the heap. Its invariant can be drawn as



This code establishes *Heap*(1, *N*).

```
for I := 2 to N do
        /* Inv: Heap(1,I-1) */
        SiftUp(I)
        /* Heap(1,I) */
```

The second stage uses the heap to build the sorted sequence. Its invariant is



The loop body maintains the invariant in two operations: because X[1] is the largest among the first *I* elements, swapping it with X[*I*] extends the sorted sequence by one element. That swap compromises the heap property, which we regain by sifting down the new top element. The code for the second stage is

```
for I := N downto 2 do
        /* Heap(1,I) and Sorted(I+1,N)
           and X[1..I] <= X[I+1..N] */
        Swap(X[1], X[I])
        /* Heap(2,I-1) and Sorted(I,N)
           and X[1..I-1] <= X[I..N] */
        SiftDown(I-1)
        /* Heap(1,I-1) and Sorted(I,N)
           and X[1..I-1] <= X[I..N] */
```

The complete Heapsort algorithm requires just five lines of code.

```
for I := 2 to N do
        SiftUp(I)
for I := N downto 2 do
        Swap(X[1], X[I])
        SiftDown(I-1)
```

Because the algorithm uses $N - 1$ *SiftUp* and *SiftDown* operations, each of cost $O(\log N)$, it runs in time $O(N \log N)$.

Principles

Efficiency. The *Shape* property guarantees that all nodes in a heap are within $\log_2 N$ of the root; procedures *SiftUp* and *SiftDown* have efficient run times precisely because the trees are balanced. Heapsort avoids using extra space by overlaying two abstract structures (a heap and a sequence) in one implementation array.

Correctness. To write code for a loop we first state its invariant precisely; the loop then makes progress towards termination while preserving its invariant. The *Shape* and *Order* properties represent a different kind of invariant: they are invariant properties of the heap data structure. A routine that operates on a heap may assume that the properties are true when it starts to work on the structure, and it must in turn make sure that they remain true when it finishes.

Abstraction. Good engineers distinguish between *what* a component does (the abstraction seen by the user) and *how* it does it (the implementation inside the black box). In this column we've seen two ways to package black boxes: procedural abstraction and abstract data types.

Procedural Abstraction. We can use a sort procedure to sort an array without knowing its implementation: we view the sort as a single operation. Procedures *SiftUp* and *SiftDown* provide us with a similar level of abstraction: as we built priority queues and heapsort, we didn't care *how* the procedures worked, but we knew *what* they did (fixing an array with the *Heap* property broken at one end or the other). Good engineering allowed us to define these black-box components once, and then use them to assemble two different kinds of systems.

Abstract Data Types. Built-in data types in programming languages are abstractly defined by means of a mathematical object and operations on the object, together with certain limitations; users needn't know about their implementation. Priority queues can be viewed in the same way, as shown in Table II. Some modern programming languages allow programmers to define their own data types, such as priority queues. Subsequent code may declare a variable to be of type "Priority Queue"; the code sees only the abstraction,

TABLE II. Integers and Priority Queues as Abstract Data Types

Property	Integers	Priority Queues
Mathematical Model	Integer	Set of Integers
Operations	Assignment, Addition, etc.	Initialize to empty, *Insert* *ExtractMin*
Limitations	Maximum and minimum size	Maximum set size, Size of elements
Implementations	Two's complement, Signed decimal	Sorted array, Heap

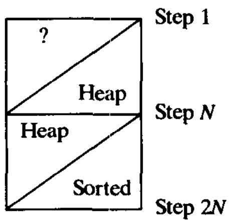
and may not know about its implementation. This discipline increases the probability of reusing software.

Problems

1. Modify *SiftDown* to have the following specification.

```
proc SiftDown(L,U)
          pre   Heap(L+1,U)
          post  Heap(L,U)
```

   What is the run time of the code? Show how it can be used to construct an N-element heap in $O(N)$ time, and thereby a faster Heapsort with the following structure.



   (Not only is this Heapsort faster than the one in the text, it also uses less code.)

2. Implement Heapsort to run as quickly as possible (see Problem 1, and also consider moving code out of loops). How does it compare to other $O(N \log N)$ sorting algorithms, such as Quicksort and Mergesort?

3. How might the heap implementation of priority queues be used to solve the following problems? How do your answers change when the inputs are sorted?
   a. Construct a Huffman code (such codes are discussed in most books on information theory and many books on data structures).
   b. Compute the sum of a large set of floating point numbers.
   c. Find the 1000 largest of ten million numbers stored on a magnetic tape.
   d. Merge many small sorted files into one large sorted file (this problem arises in implementing a disk-based Mergesort program).

4. [D.S. Johnson] The bin packing problem calls for assigning a set of N weights (each between zero and one) to a minimal number of unit-capacity bins. The first-fit heuristic for this problem considers the weights in the sequence in which they are presented, and places each weight in the first bin in which it fits, scanning the bins in increasing order. Show how a heap-like structure can implement this heuristic in $O(N \log N)$ time. (This problem is distantly related to efficient algorithms for first-fit storage allocation, such as those discussed in Exercise 6.2.4.30 of Knuth's *Sorting and Searching.*)

5. [E. McCreight] A common implementation of sequential files on disk has each block point to its successor, which may be any block on the disk. This method requires a constant amount of time to write a block (as the file is originally written), to read the first block in the file, and to read the I^{th} block, once you have read the $I - 1^{st}$ block. Reading the I^{th} block therefore requires time proportional to I. Show how by adding just one additional pointer per node, you can keep all the other properties, but allow the I^{th} block to be read in time proportional to $\log I$. [Hint: heaps have implicit pointers from node I to what other nodes?] Explain what the algorithm for reading the I^{th} block has in common with an algorithm for raising a number to the I^{th} power in time proportional to $\log I$.

6. On many computers the most expensive part of a binary search program is the division by 2 to find the center of the current range. Show how the implicit trees used for heaps can replace that division with a multiplication (assuming that the table has been constructed properly). Give algorithms for building and searching such a table.

7. What are appropriate implementations for a priority queue that represents integers in the range $1..K$, when the average size of the set is much larger than K?

Further Reading

J.W.J. Williams's original paper on heaps appeared in *Communications of the ACM* in June 1964, and is still fascinating reading today. Even though it is miniature by today's standards (less than a page long), it is large by the standards of the time (Floyd's *TREESORT* algorithm in the August 1962 *Communications* was less than a quarter of a page!).

For a more up-to-date view of heaps, see Tarjan's *Data Structures and Network Algorithms*, published in 1983 by the Society for Industrial and Applied Mathematics. Chapter 3 is devoted to heaps. This monograph is a fine introduction to the field described in its title, and at only $14.50, you can't afford not to buy it.

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Avenue, Murray Hill, NJ 07974.