```
        }
    }
}
```

You could, in principle, rearrange any number of additional arrays along with brr, but this becomes wasteful as the number of such arrays becomes large. The preferred technique is to make use of an index table, as described in §8.4.

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1978, *Communications of the ACM*, vol. 21, pp. 847–857. [1]

8.3 Heapsort

While usually not quite as fast as Quicksort, Heapsort is one of our favorite sorting routines. It is a true "in-place" sort, requiring no auxiliary storage. It is an $N \log_2 N$ process, not only on average, but also for the worst-case order of input data. In fact, its worst case is only 20 percent or so worse than its average running time.

It is beyond our scope to give a complete exposition on the theory of Heapsort. We will mention the general principles, then let you refer to the references [1,2], or analyze the program yourself, if you want to understand the details.

A set of N numbers $a_i,\ i = 1, \ldots, N$, is said to form a "heap" if it satisfies the relation

$$a_{j/2} \geq a_j \quad \text{for} \quad 1 \leq j/2 < j \leq N \tag{8.3.1}$$

Here the division in $j/2$ means "integer divide," i.e., is an exact integer or else is rounded down to the closest integer. Definition (8.3.1) will make sense if you think of the numbers a_i as being arranged in a binary tree, with the top, "boss," node being a_1, the two "underling" nodes being a_2 and a_3, *their* four underling nodes being a_4 through a_7, etc. (See Figure 8.3.1.) In this form, a heap has every "supervisor" greater than or equal to its two "supervisees," down through the levels of the hierarchy.

If you have managed to rearrange your array into an order that forms a heap, then sorting it is very easy: You pull off the "top of the heap," which will be the largest element yet unsorted. Then you "promote" to the top of the heap its largest underling. Then you promote *its* largest underling, and so on. The process is like what happens (or is supposed to happen) in a large corporation when the chairman of the board retires. You then repeat the whole process by retiring the new chairman of the board. Evidently the whole thing is an $N \log_2 N$ process, since each retiring chairman leads to $\log_2 N$ promotions of underlings.

Well, how do you arrange the array into a heap in the first place? The answer is again a "sift-up" process like corporate promotion. Imagine that the corporation starts out with $N/2$ employees on the production line, but with no supervisors. Now a supervisor is hired to supervise two workers. If he is less capable than one of his workers, that one is promoted in his place, and he joins the production line. After supervisors are hired, then supervisors of supervisors are hired, and so on up

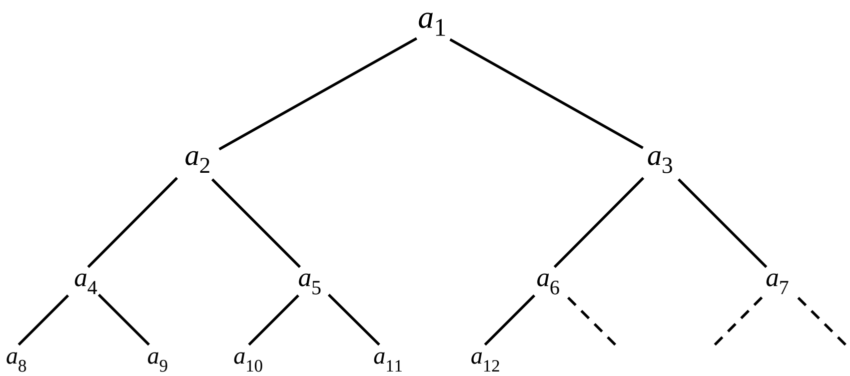

Figure 8.3.1. Ordering implied by a "heap," here of 12 elements. Elements connected by an upward path are sorted with respect to one another, but there is not necessarily any ordering among elements related only "laterally."

the corporate ladder. Each employee is brought in at the top of the tree, but then immediately sifted down, with more capable workers promoted until their proper corporate level has been reached.

In the Heapsort implementation, the same "sift-up" code can be used for the initial creation of the heap and for the subsequent retirement-and-promotion phase. One execution of the Heapsort function represents the entire life-cycle of a giant corporation: $N/2$ workers are hired; $N/2$ potential supervisors are hired; there is a sifting up in the ranks, a sort of super Peter Principle: in due course, each of the original employees gets promoted to chairman of the board.

```
void hpsort(unsigned long n, float ra[])
Sorts an array ra[1..n] into ascending numerical order using the Heapsort algorithm. n is
input; ra is replaced on output by its sorted rearrangement.
{
    unsigned long i,ir,j,l;
    float rra;

    if (n < 2) return;
    l=(n >> 1)+1;
    ir=n;
    The index l will be decremented from its initial value down to 1 during the "hiring" (heap
    creation) phase. Once it reaches 1, the index ir will be decremented from its initial value
    down to 1 during the "retirement-and-promotion" (heap selection) phase.
    for (;;) {
        if (l > 1) {                    Still in hiring phase.
            rra=ra[--l];
        } else {                        In retirement-and-promotion phase.
            rra=ra[ir];                 Clear a space at end of array.
            ra[ir]=ra[1];               Retire the top of the heap into it.
            if (--ir == 1) {            Done with the last promotion.
                ra[1]=rra;              The least competent worker of all!
                break;
            }
        }
        i=l;                            Whether in the hiring phase or promotion phase, we
        j=l+l;                              here set up to sift down element rra to its proper
        while (j <= ir) {                   level.
            if (j < ir && ra[j] < ra[j+1]) j++;    Compare to the better underling.
            if (rra < ra[j]) {          Demote rra.
                ra[i]=ra[j];
```

```
            i=j;
            j <<= 1;
        } else break;             Found rra's level. Terminate the sift-down.
    }
    ra[i]=rra;                    Put rra into its slot.
  }
}
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.3. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 11. [2]

8.4 Indexing and Ranking

The concept of *keys* plays a prominent role in the management of data files. A data *record* in such a file may contain several items, or *fields*. For example, a record in a file of weather observations may have fields recording time, temperature, and wind velocity. When we sort the records, we must decide which of these fields we want to be brought into sorted order. The other fields in a record just come along for the ride, and will not, in general, end up in any particular order. The field on which the sort is performed is called the *key* field.

For a data file with many records and many fields, the actual movement of N records into the sorted order of their keys $K_i,\ i = 1, \ldots, N$, can be a daunting task. Instead, one can construct an *index table* $I_j,\ j = 1, \ldots, N$, such that the smallest K_i has $i = I_1$, the second smallest has $i = I_2$, and so on up to the largest K_i with $i = I_N$. In other words, the array

$$K_{I_j} \quad j = 1, 2, \ldots, N \tag{8.4.1}$$

is in sorted order when indexed by j. When an index table is available, one need not move records from their original order. Further, different index tables can be made from the same set of records, indexing them to different keys.

The algorithm for constructing an index table is straightforward: Initialize the index array with the integers from 1 to N, then perform the Quicksort algorithm, moving the elements around *as if* one were sorting the keys. The integer that initially numbered the smallest key thus ends up in the number one position, and so on.

```
#include "nrutil.h"
#define SWAP(a,b) itemp=(a);(a)=(b);(b)=itemp;
#define M 7
#define NSTACK 50

void indexx(unsigned long n, float arr[], unsigned long indx[])
```

Indexes an array `arr[1..n]`, i.e., outputs the array `indx[1..n]` such that `arr[indx[j]]` is in ascending order for $j = 1, 2, \ldots, N$. The input quantities `n` and `arr` are not changed.

```
{
    unsigned long i,indxt,ir=n,itemp,j,k,l=1;
    int jstack=0,*istack;
    float a;

    istack=ivector(1,NSTACK);
```