```
            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 $K_i,\ i = 1,\dots,N$, can be a daunting task. Instead, one can construct an *index table* $I_j,\ j = 1,\dots,N$, 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

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

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 $j = 1, 2, \dots, N$. 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;
```

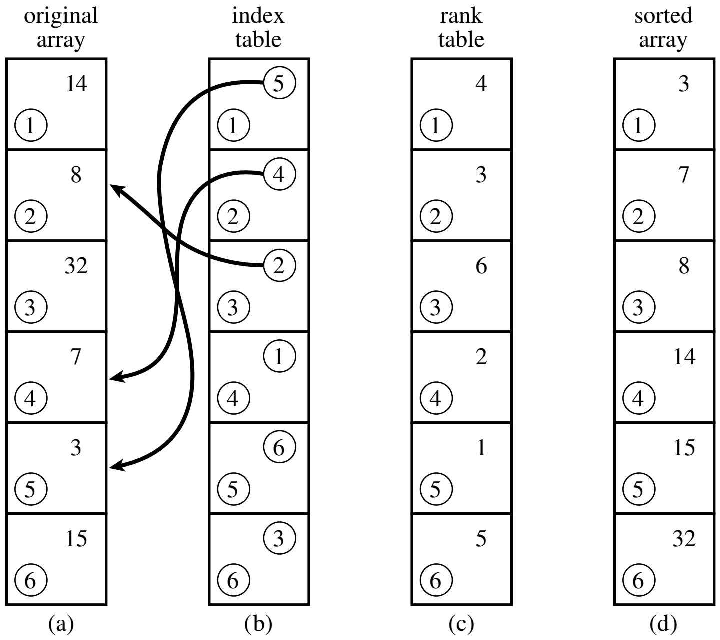

Figure 8.4.1. (a) An unsorted array of six numbers. (b) Index table, whose entries are pointers to the elements of (a) in ascending order. (c) Rank table, whose entries are the ranks of the corresponding elements of (a). (d) Sorted array of the elements in (a).

```
istack=ivector(1,NSTACK);
for (j=1;j<=n;j++) indx[j]=j;
for (;;) {
    if (ir-l < M) {
        for (j=l+1;j<=ir;j++) {
            indxt=indx[j];
            a=arr[indxt];
            for (i=j-1;i>=l;i--) {
                if (arr[indx[i]] <= a) break;
                indx[i+1]=indx[i];
            }
            indx[i+1]=indxt;
        }
        if (jstack == 0) break;
        ir=istack[jstack--];
        l=istack[jstack--];
    } else {
        k=(l+ir) >> 1;
        SWAP(indx[k],indx[l+1]);
        if (arr[indx[l]] > arr[indx[ir]]) {
            SWAP(indx[l],indx[ir])
        }
        if (arr[indx[l+1]] > arr[indx[ir]]) {
            SWAP(indx[l+1],indx[ir])
        }
        if (arr[indx[l]] > arr[indx[l+1]]) {
            SWAP(indx[l],indx[l+1])
        }
        i=l+1;
        j=ir;
        indxt=indx[l+1];
```

```
            a=arr[indxt];
            for (;;) {
                do i++; while (arr[indx[i]] < a);
                do j--; while (arr[indx[j]] > a);
                if (j < i) break;
                SWAP(indx[i],indx[j])
            }
            indx[l+1]=indx[j];
            indx[j]=indxt;
            jstack += 2;
            if (jstack > NSTACK) nrerror("NSTACK too small in indexx.");
            if (ir-i+1 >= j-l) {
                istack[jstack]=ir;
                istack[jstack-1]=i;
                ir=j-1;
            } else {
                istack[jstack]=j-1;
                istack[jstack-1]=l;
                l=i;
            }
        }
    }
    free_ivector(istack,1,NSTACK);
}
```

If you want to sort an array while making the corresponding rearrangement of several or many other arrays, you should first make an index table, then use it to rearrange each array in turn. This requires two arrays of working space: one to hold the index, and another into which an array is temporarily moved, and from which it is redeposited back on itself in the rearranged order. For 3 arrays, the procedure looks like this:

```
#include "nrutil.h"

void sort3(unsigned long n, float ra[], float rb[], float rc[])
Sorts an array ra[1..n] into ascending numerical order while making the corresponding re-
arrangements of the arrays rb[1..n] and rc[1..n]. An index table is constructed via the
routine indexx.
{
    void indexx(unsigned long n, float arr[], unsigned long indx[]);
    unsigned long j,*iwksp;
    float *wksp;

    iwksp=lvector(1,n);
    wksp=vector(1,n);
    indexx(n,ra,iwksp);                          Make the index table.
    for (j=1;j<=n;j++) wksp[j]=ra[j];            Save the array ra.
    for (j=1;j<=n;j++) ra[j]=wksp[iwksp[j]];     Copy it back in rearranged order.
    for (j=1;j<=n;j++) wksp[j]=rb[j];            Ditto rb.
    for (j=1;j<=n;j++) rb[j]=wksp[iwksp[j]];
    for (j=1;j<=n;j++) wksp[j]=rc[j];            Ditto rc.
    for (j=1;j<=n;j++) rc[j]=wksp[iwksp[j]];
    free_vector(wksp,1,n);
    free_lvector(iwksp,1,n);
}
```

The generalization to any other number of arrays is obviously straightforward.

A *rank table* is different from an index table. A rank table's jth entry gives the rank of the jth element of the original array of keys, ranging from 1 (if that element was the smallest) to N (if that element was the largest). One can easily construct a rank table from an index table, however:

```
void rank(unsigned long n, unsigned long indx[], unsigned long irank[])
Given indx[1..n] as output from the routine indexx, returns an array irank[1..n], the
corresponding table of ranks.
{
    unsigned long j;

    for (j=1;j<=n;j++) irank[indx[j]]=j;
}
```

Figure 8.4.1 summarizes the concepts discussed in this section.

8.5 Selecting the Mth Largest

Selection is sorting's austere sister. (Say *that* five times quickly!) Where sorting demands the rearrangement of an entire data array, selection politely asks for a single returned value: What is the kth smallest (or, equivalently, the $m = N+1-k$th largest) element out of N elements? The fastest methods for selection do, unfortunately, rearrange the array for their own computational purposes, typically putting all smaller elements to the left of the kth, all larger elements to the right, and scrambling the order within each subset. This side effect is at best innocuous, at worst downright inconvenient. When the array is very long, so that making a scratch copy of it is taxing on memory, or when the computational burden of the selection is a negligible part of a larger calculation, one turns to selection algorithms without side effects, which leave the original array undisturbed. Such *in place* selection is slower than the faster selection methods by a factor of about 10. We give routines of both types, below.

The most common use of selection is in the statistical characterization of a set of data. One often wants to know the median element in an array, or the top and bottom quartile elements. When N is odd, the median is the kth element, with $k = (N+1)/2$. When N is even, statistics books define the median as the arithmetic mean of the elements $k = N/2$ and $k = N/2 + 1$ (that is, $N/2$ from the bottom and $N/2$ from the top). If you accept such pedantry, you must perform two separate selections to find these elements. For $N > 100$ we usually define $k = N/2$ to be the median element, pedants be damned.

The fastest general method for selection, allowing rearrangement, is *partitioning*, exactly as was done in the Quicksort algorithm (§8.2). Selecting a "random" partition element, one marches through the array, forcing smaller elements to the left, larger elements to the right. As in Quicksort, it is important to optimize the inner loop, using "sentinels" (§8.2) to minimize the number of comparisons. For sorting, one would then proceed to further partition both subsets. For selection, we can ignore one subset and attend only to the one that contains our desired kth element. Selection by partitioning thus does not need a stack of pending operations, and its operations count scales as N rather than as $N \log N$ (see [1]). Comparison with `sort` in §8.2 should make the following routine obvious: