```
                k=j << 1;
                if (k > m) break;
                if (k != m && heap[k] > heap[k+1]) k++;
                if (heap[j] <= heap[k]) break;
                swap=heap[k];
                heap[k]=heap[j];
                heap[j]=swap;
                j=k;
            }
        }
    }
}
```

CITED REFERENCES AND FURTHER READING:

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), pp. 126ff. [1]

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

8.6 Determination of Equivalence Classes

A number of techniques for sorting and searching relate to data structures whose details are beyond the scope of this book, for example, trees, linked lists, etc. These structures and their manipulations are the bread and butter of computer science, as distinct from numerical analysis, and there is no shortage of books on the subject.

In working with experimental data, we have found that one particular such manipulation, namely the determination of equivalence classes, arises sufficiently often to justify inclusion here.

The problem is this: There are N "elements" (or "data points" or whatever), numbered $1, \ldots, N$. You are given pairwise information about whether elements are in the same *equivalence class* of "sameness," by whatever criterion happens to be of interest. For example, you may have a list of facts like: "Element 3 and element 7 are in the same class; element 19 and element 4 are in the same class; element 7 and element 12 are in the same class, ..." Alternatively, you may have a procedure, given the numbers of two elements j and k, for deciding whether they are in the same class or different classes. (Recall that an equivalence relation can be anything satisfying the *RST properties*: reflexive, symmetric, transitive. This is compatible with any intuitive definition of "sameness.")

The desired output is an assignment to each of the N elements of an equivalence class number, such that two elements are in the same class if and only if they are assigned the same class number.

Efficient algorithms work like this: Let $F(j)$ be the class or "family" number of element j. Start off with each element in its own family, so that $F(j) = j$. The array $F(j)$ can be interpreted as a tree structure, where $F(j)$ denotes the parent of j. If we arrange for each family to be its own tree, disjoint from all the other "family trees," then we can label each family (equivalence class) by its most senior great-great-...grandparent. The detailed topology of the tree doesn't matter at all, as long as we graft each related element onto it *somewhere*.

Therefore, we process each elemental datum "j is equivalent to k" by (i) tracking j up to its highest ancestor, (ii) tracking k up to its highest ancestor, (iii) giving j to k as a new parent, or vice versa (it makes no difference). After processing all the relations, we go through all the elements j and reset their $F(j)$'s to their highest possible ancestors, which then label the equivalence classes.

The following routine, based on Knuth [1], assumes that there are `m` elemental pieces of information, stored in two arrays of length `m`, `lista,listb`, the interpretation being that `lista[j]` and `listb[j]`, `j=1...m`, are the numbers of two elements which (we are thus told) are related.

```
void eclass(int nf[], int n, int lista[], int listb[], int m)
```
Given m equivalences between pairs of n individual elements in the form of the input arrays lista[1..m] and listb[1..m], this routine returns in nf[1..n] the number of the equivalence class of each of the n elements, integers between 1 and n (not all such integers used).
```
{
    int l,k,j;

    for (k=1;k<=n;k++) nf[k]=k;              Initialize each element its own class.
    for (l=1;l<=m;l++) {                     For each piece of input information...
        j=lista[l];
        while (nf[j] != j) j=nf[j];          Track first element up to its ancestor.
        k=listb[l];
        while (nf[k] != k) k=nf[k];          Track second element up to its ancestor.
        if (j != k) nf[j]=k;                 If they are not already related, make them
    }                                            so.
    for (j=1;j<=n;j++)                       Final sweep up to highest ancestors.
        while (nf[j] != nf[nf[j]]) nf[j]=nf[nf[j]];
}
```

Alternatively, we may be able to construct a function equiv(j,k) that returns a nonzero (true) value if elements j and k are related, or a zero (false) value if they are not. Then we want to loop over all pairs of elements to get the complete picture. D. Eardley has devised a clever way of doing this while simultaneously sweeping the tree up to high ancestors in a manner that keeps it current and obviates most of the final sweep phase:

```
void eclazz(int nf[], int n, int (*equiv)(int, int))
```
Given a user-supplied boolean function equiv which tells whether a pair of elements, each in the range 1...n, are related, return in nf[1..n] equivalence class numbers for each element.
```
{
    int kk,jj;

    nf[1]=1;
    for (jj=2;jj<=n;jj++) {                  Loop over first element of all pairs.
        nf[jj]=jj;
        for (kk=1;kk<=(jj-1);kk++) {         Loop over second element of all pairs.
            nf[kk]=nf[nf[kk]];               Sweep it up this much.
            if ((*equiv)(jj,kk)) nf[nf[nf[kk]]]=jj;
            Good exercise for the reader to figure out why this much ancestry is necessary!
        }
    }
    for (jj=1;jj<=n;jj++) nf[jj]=nf[nf[jj]];     Only this much sweeping is needed
}                                                    finally.
```

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.3.3. [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 30.