2. SORTING

2.1. Introduction

The primary purpose of this chapter is to provide an extensive set of examples illustrating the use of the data structures introduced in the preceding chapter and to show how the choice of structure for the underlying data profoundly influences the algorithms that perform a given task. Sorting is also a good example to show that such a task may be performed according to many different algorithms, each one having certain advantages and disadvantages that have to be weighed against each other in the light of the particular application.

Sorting is generally understood to be the process of rearranging a given set of objects in a specific order. The purpose of sorting is to facilitate the later search for members of the sorted set. As such it is an almost universally performed, fundamental activity. Objects are sorted in telephone books, in income tax files, in tables of contents, in libraries, in dictionaries, in warehouses, and almost everywhere that stored objects have to be searched and retrieved. Even small children are taught to put their things "in order", and they are confronted with some sort of sorting long before they learn anything about arithmetic.

Hence, sorting is a relevant and essential activity, particularly in data processing. What else would be easier to sort than data! Nevertheless, our primary interest in sorting is devoted to the even more fundamental techniques used in the construction of algorithms. There are not many techniques that do not occur somewhere in connection with sorting algorithms. In particular, sorting is an ideal subject to demonstrate a great diversity of algorithms, all having the same purpose, many of them being optimal in some sense, and most of them having advantages over others. It is therefore an ideal subject to demonstrate the necessity of performance analysis of algorithms. The example of sorting is moreover well suited for showing how a very significant gain in performance may be obtained by the development of sophisticated algorithms when obvious methods are readily available.

The dependence of the choice of an algorithm on the structure of the data to be processed -- an ubiquitous phenomenon -- is so profound in the case of sorting that sorting methods are generally classified into two categories, namely, sorting of arrays and sorting of (sequential) files. The two classes are often called *internal* and *external sorting* because arrays are stored in the fast, high-speed, random-access "internal" store of computers and files are appropriate on the slower, but more spacious "external" stores based on mechanically moving devices (disks and tapes). The importance of this distinction is obvious from the example of sorting numbered cards. Structuring the cards as an array corresponds to laying them out in front of the sorter so that each card is visible and individually accessible (see Fig. 2.1).

Structuring the cards as a file, however, implies that from each pile only the card on the top is visible (see Fig. 2.2). Such a restriction will evidently have serious consequences on the sorting method to be used, but it is unavoidable if the number of cards to be laid out is larger than the available table.

Before proceeding, we introduce some terminology and notation to be used throughout this chapter. If we are given n items

$$a_0, a_1, \ldots, a_{n-1}$$

sorting consists of permuting these items into an array

$$a_{k0}, a_{k1}, \ldots, a_{k[n-1]}$$

such that, given an ordering function f,

$$f(a_{k0}) \leq f(a_{k1}) \leq \ldots \leq f(a_{k[n-1]})$$

Ordinarily, the ordering function is not evaluated according to a specified rule of computation but is stored as an explicit component (field) of each item. Its value is called the *key* of the item. As a consequence, the record structure is particularly well suited to represent items and might for example be declared as follows:

```
TYPE Item =  RECORD key:  INTEGER;
               (*other components declared here*)
             END
```

The other components represent relevant data about the items in the collection; the key merely assumes the purpose of identifying the items. As far as our sorting algorithms are concerned, however, the key is the only relevant component, and there is no need to define any particular remaining components. In the following discussions, we shall therefore discard any associated information and assume that the type *Item* be defined as INTEGER. This choice of the key type is somewhat arbitrary. Evidently, any type on which a total ordering relation is defined could be used just as well.

A sorting method is called *stable* if the relative order if items with equal keys remains unchanged by the sorting process. Stability of sorting is often desirable, if items are already ordered (sorted) according to some secondary keys, i.e., properties not reflected by the (primary) key itself.

This chapter is not to be regarded as a comprehensive survey in sorting techniques. Rather, some selected, specific methods are exemplified in greater detail. For a thorough treatment of sorting, the interested reader is referred to the excellent and comprehensive compendium by D. E. Knuth [2-7] (see also Lorin [2-10]).

2.2. Sorting Arrays

The predominant requirement that has to be made for sorting methods on arrays is an economical use of the available store. This implies that the permutation of items which brings the items into order has to be performed in situ, and that methods which transport items from an array a to a result array b are intrinsically of minor interest. Having thus restricted our choice of methods among the many possible solutions by the criterion of economy of storage, we proceed to a first classification according to their efficiency, i.e., their economy of time. A good measure of efficiency is obtained by counting the numbers C of needed key comparisons and M of moves (transpositions) of items. These numbers are functions of the number n of items to be sorted. Whereas good sorting algorithms require in the order of n*log(n) comparisons, we first discuss several simple and obvious sorting techniques, called *straight methods*, all of which require in the order n^2 comparisons of keys. There are three good reasons for presenting straight methods before proceeding to the faster algorithms.

1. Straight methods are particularly well suited for elucidating the characteristics of the major sorting principles.
2. Their programs are easy to understand and are short. Remember that programs occupy storage as well!
3. Although sophisticated methods require fewer operations, these operations are usually more complex in their details; consequently, straight methods are faster for sufficiently small n, although they must not be used for large n.

Sorting methods that sort items in situ can be classified into three principal categories according to their underlying method:

Sorting by insertion
Sorting by selection
Sorting by exchange

These three pinciples will now be examined and compared. The procedures operate on a global variable *a* whose components are to be sorted in situ, i.e. without requiring additional, temporary storage. The components are the keys themselves. We discard other data represented by the record type *Item*, thereby simplifying matters. In all algorithms to be developed in this chapter, we will assume the presence of an array *a* and a constant *n*, the number of elements of *a*:

```
TYPE Item = INTEGER;

VAR a: ARRAY n OF Item
```

2.2.1. Sorting by Straight Insertion

This method is widely used by card players. The items (cards) are conceptually divided into a destination sequence a_1 ... a_{i-1} and a source sequence a_i ... a_n. In each step, starting with i = 2 and incrementing i by unity, the i th element of the source sequence is picked and transferred into the destination sequence by inserting it at the appropriate place.

Initial Keys:	44	55	12	42	94	18	06	67
i=1	44	55	12	42	94	18	06	67
i=2	12	44	55	42	94	18	06	67
i=3	12	42	44	55	94	18	06	67
i=4	12	42	44	55	94	18	06	67
i=5	12	18	42	44	55	94	06	67
i=6	06	12	18	42	44	55	94	67
i=7	06	12	18	42	44	55	67	94

Table 2.1 A Sample Process of Straight Insertion Sorting.

The process of sorting by insertion is shown in an example of eight numbers chosen at random (see Table 2.1). The algorithm of straight insertion is

```
FOR i := 1 TO n-1 DO
  x := a[i];
  insert x at the appropriate place in a0 ... ai
END
```

In the process of actually finding the appropriate place, it is convenient to alternate between comparisons and moves, i.e., to let x sift down by comparing x with the next item a_j, and either inserting x or moving a_j to the right and proceeding to the left. We note that there are two distinct conditions that may cause the termination of the sifting down process:

1. An item a_j is found with a key less than the key of x.
2. The left end of the destination sequence is reached.

```
PROCEDURE StraightInsertion;
  VAR i, j: INTEGER; x: Item;
BEGIN
  FOR i := 1 TO n-1 DO
    x := a[i]; j := i;
    WHILE (j > 0) & (x < a[j-1] DO a[j] := a[j-1]; DEC(j) END ;
    a[j] := x
  END
END StraightInsertion
```

Analysis of straight insertion. The number C_i of key comparisons in the i-th sift is at most i-1, at least 1, and -- assuming that all permutations of the n keys are equally probable -- i/2 in the average. The number Mi of moves (assignments of items) is $C_i + 2$ (including the sentinel). Therefore, the total numbers of comparisons and moves are

$C_{min} = n-1$ $\quad M_{min} = 3*(n-1)$
$C_{ave} = (n^2 + n - 2)/4$ $\quad M_{ave} = (n^2 + 9n - 10)/4$
$C_{max} = (n^2 + n - 4)/4$ $\quad M_{max} = (n^2 + 3n - 4)/2$

The minimal numbers occur if the items are initially in order; the worst case occurs if the items are initially in reverse order. In this sense, sorting by insertion exhibits a truly natural behavior. It is plain that the given algorithm also describes a stable sorting process: it leaves the order of items with equal keys unchanged.

The algorithm of straight insertion is easily improved by noting that the destination sequence a_0 ... a_{i-1}, in which the new item has to be inserted, is already ordered. Therefore, a faster method of determining the insertion point can be used. The obvious choice is a binary search that samples the destination sequence in the middle and continues bisecting until the insertion point is found. The modified sorting algorithm is called *binary insertion*.

```
PROCEDURE BinaryInsertion(VAR a: ARRAY OF Item; n: INTEGER);
  VAR i, j, m, L, R: INTEGER; x: Item;
BEGIN
  FOR i := 1 TO n-1 DO
```

```
    x := a[i]; L := 1; R := i;
    WHILE L < R DO
      m := (L+R) DIV 2;
      IF a[m] <= x THEN L := m+1 ELSE R := m END
    END ;
    FOR j := i TO R+1 BY -1 DO a[j] := a[j-1] END ;
    a[R] := x
  END
END BinaryInsertion
```

Analysis of binary insertion. The insertion position is found if L = R. Thus, the search interval must in the end be of length 1; and this involves halving the interval of length i log(i) times. Thus,

$$C = \mathbf{S}i: 0 \le I \le n: \lceil \log(i) \rceil$$

We approximate this sum by the integral

$$\text{Int}(0:n-1) \log(x)\, dx = n*(\log n - c) + c$$

where c = log e = 1/ln 2 = 1.44269... .

The number of comparisons is essentially independent of the initial order of the items. However, because of the truncating character of the division involved in bisecting the search interval, the true number of comparisons needed with i items may be up to 1 higher than expected. The nature of this bias is such that insertion positions at the low end are on the average located slightly faster than those at the high end, thereby favoring those cases in which the items are originally highly out of order. In fact, the minimum number of comparisons is needed if the items are initially in reverse order and the maximum if they are already in order. Hence, this is a case of unnatural behavior of a sorting algorithm. The number of comparisons is then approximately

$$C \approx n*(\log n - \log e \pm 0.5)$$

Unfortunately, the improvement obtained by using a binary search method applies only to the number of comparisons but not to the number of necessary moves. In fact, since moving items, i.e., keys and associated information, is in general considerably more time-consuming than comparing two keys, the improvement is by no means drastic: the important term M is still of the order n^2. And, in fact, sorting the already sorted array takes more time than does straight insertion with sequential search.

This example demonstrates that an "obvious improvement" often has much less drastic consequences than one is first inclined to estimate and that in some cases (that do occur) the "improvement" may actually turn out to be a deterioration. After all, sorting by insertion does not appear to be a very suitable method for digital computers: insertion of an item with the subsequent shifting of an entire row of items by a single position is uneconomical. One should expect better results from a method in which moves of items are only performed upon single items and over longer distances. This idea leads to sorting by selection.

2.2.2 Sorting by Straight Selection

This method is based on the following principle:

1. Select the item with the least key.
2. Exchange it with the first item a_0.
3. Then repeat these operations with the remaining n-1 items, then with n-2 items, until only one item -- the largest -- is left.

This method is shown on the same eight keys as in Table 2.1.

Initial keys	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67

06	12	18	42	44	55	94	67
06	12	18	42	44	55	94	67
06	12	18	42	44	55	67	94

Table 2.2 A Sample Process of Straight Selection Sorting.

The algorithm is formulated as follows:

```
FOR i := 0 TO n-1 DO
  assign the index of the least item of a_i ... a_n-1 to k;
  exchange a_i with a_k
END
```

This method, called *straight selection*, is in some sense the opposite of straight insertion: Straight insertion considers in each step only the one next item of the source sequence and all items of the destination array to find the insertion point; straight selection considers all items of the source array to find the one with the least key and to be deposited as the one next item of the destination sequence..

```
PROCEDURE StraightSelection;
  VAR i, j, k: INTEGER; x: Item;
BEGIN
  FOR i := 0 TO n-2 DO
    k := i; x := a[i];
    FOR j := i+1 TO n-1 DO
      IF a[j] < x THEN k := j; x := a[k] END
    END ;
    a[k] := a[i]; a[i] := x
  END
END StraightSelection
```

Analysis of straight selection. Evidently, the number C of key comparisons is independent of the initial order of keys. In this sense, this method may be said to behave less naturally than straight insertion. We obtain

$$C = (n^2 - n)/2$$

The number M of moves is at least

$$M_{min} = 3*(n-1)$$

in the case of initially ordered keys and at most

$$M_{max} = n^2/4 + 3*(n-1)$$

if initially the keys are in reverse order. In order to determine Mavg we make the following deliberations: The algorithm scans the array, comparing each element with the minimal value so far detected and, if smaller than that minimum, performs an assignment. The probability that the second element is less than the first, is 1/2; this is also the probability for a new assignment to the minimum. The chance for the third element to be less than the first two is 1/3, and the chance of the fourth to be the smallest is 1/4, and so on. Therefore the total expected number of moves is H_{n-1}, where H_n is the n th harmonic number

$$H_n = 1 + 1/2 + 1/3 + ... + 1/n$$

H_n can be expressed as

$$H_n = \ln(n) + g + 1/2n - 1/12n^2 + ...$$

where g = 0.577216... is Euler's constant. For sufficiently large n, we may ignore the fractional terms and therefore approximate the average number of assignments in the i th pass as

$$F_i = \ln(i) + g + 1$$

The average number of moves M_{avg} in a selection sort is then the sum of F_i with i ranging from 1 to n.

$$M_{avg} = n*(g+1) + (\mathbf{S}i: 1 \le i \le n: \ln(i))$$

By further approximating the sum of discrete terms by the integral

Integral (1:n) ln(x) dx = n * ln(n) - n + 1

we obtain an approximate value

$M_{avg} = n * (ln(n) + g)$

We may conclude that in general the algorithm of straight selection is to be preferred over straight insertion, although in the cases in which keys are initially sorted or almost sorted, straight insertion is still somewhat faster.

2.2.3 Sorting by Straight Exchange

The classification of a sorting method is seldom entirely clear-cut. Both previously discussed methods can also be viewed as exchange sorts. In this section, however, we present a method in which the exchange of two items is the dominant characteristic of the process. The subsequent algorithm of straight exchanging is based on the principle of comparing and exchanging pairs of adjacent items until all items are sorted.

As in the previous methods of straight selection, we make repeated passes over the array, each time sifting the least item of the remaining set to the left end of the array. If, for a change, we view the array to be in a vertical instead of a horizontal position, and -- with the help of some imagination -- the items as bubbles in a water tank with weights according to their keys, then each pass over the array results in the ascension of a bubble to its appropriate level of weight (see Table 2.3). This method is widely known as the *Bubblesort*.

I = 1	2	3	4	5	6	7	8
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

Table 2.3 A Sample of Bubblesorting.

```
PROCEDURE BubbleSort;
  VAR i, j: INTEGER; x: Item;
BEGIN
  FOR i := 1 TO n-1 DO
    FOR j := n-1 TO i BY -1 DO
      IF a[j-1] > a[j] THEN
        x := a[j-1]; a[j-1] := a[j]; a[j] := x
      END
    END
  END
END BubbleSort
```

This algorithm easily lends itself to some improvements. The example in Table 2.3 shows that the last three passes have no effect on the order of the items because the items are already sorted. An obvious technique for improving this algorithm is to remember whether or not any exchange had taken place during a pass. A last pass without further exchange operations is therefore necessary to determine that the algorithm may be terminated. However, this improvement may itself be improved by remembering not merely the fact that an exchange took place, but rather the position (index) of the last exchange. For example, it is plain that all pairs of adjacent items below this index k are in the desired order. Subsequent scans may therefore be terminated at this index instead of having to proceed to the predetermined lower limit i. The careful programmer notices, however, a peculiar asymmetry: A single misplaced bubble in the heavy end of an otherwise sorted array will sift into order in a single pass, but a misplaced item in the light end will sink towards its correct position only one step in each pass. For example, the array

12 18 42 44 55 67 94 06

is sorted by the improved Bubblesort in a single pass, but the array

94 06 12 18 42 44 55 67

requires seven passes for sorting. This unnatural asymmetry suggests a third improvement: alternating the direction of consecutive passes. We appropriately call the resulting algorithm *Shakersort*. Its behavior is illustrated in Table 2.4 by applying it to the same eight keys that were used in Table 2.3.

```
PROCEDURE ShakerSort;
  VAR j, k, L, R: INTEGER;  x: Item;
BEGIN L := 1; R := n-1; k := R;
  REPEAT
    FOR j := R TO L BY -1 DO
      IF a[j-1] > a[j] THEN
        x := a[j-1]; a[j-1] := a[j]; a[j] := x; k := j
      END
    END ;
    L := k+1;
    FOR j := L TO R BY +1 DO
      IF a[j-1] > a[j] THEN
        x := a[j-1]; a[j-1] := a[j]; a[j] := x; k := j
      END
    END ;
    R := k-1
  UNTIL L > R
END ShakerSort
```

L =	2	3	3	4	4
R =	8	8	7	7	4
dir =	↑	↓	↑	↓	↑
	44	06	06	06	06
	55	44	44	12	12
	12	55	12	44	18
	42	12	42	18	42
	94	42	55	42	44
	18	94	18	55	55
	06	18	67	67	67
	67	67	94	94	94

Table 2.4 An Example of Shakersort.

Analysis of Bubblesort and Shakersort. The number of comparisons in the straight exchange algorithm is

$$C = (n^2 - n)/2$$

and the minimum, average, and maximum numbers of moves (assignments of items) are

$$M_{min} = 0, \qquad M_{avg} = 3*(n^2 - n)/2, \qquad M_{max} = 3*(n^2 - n)/4$$

The analysis of the improved methods, particularly that of Shakersort, is intricate. The least number of comparisons is $C_{min} = n-1$. For the improved Bubblesort, Knuth arrives at an average number of passes proportional to $n - k_1*n^{½}|$, and an average number of comparisons proportional to $(n^2 – n*(k_2 + \ln(n)))/2$. But we note that all improvements mentioned above do in no way affect the number of exchanges; they only reduce the number of redundant double checks. Unfortunately, an exchange of two items is generally a more costly operation than a comparison of keys; our clever improvements therefore have a much less profound effect than one would intuitively expect.

This analysis shows that the exchange sort and its minor improvements are inferior to both the insertion and the selection sorts; and in fact, the Bubblesort has hardly anything to recommend it except its catchy name.

The Shakersort algorithm is used with advantage in those cases in which it is known that the items are already almost in order -- a rare case in practice.

It can be shown that the average distance that each of the n items has to travel during a sort is n/3 places. This figure provides a clue in the search for improved, i.e. more effective sorting methods. All straight sorting methods essentially move each item by one position in each elementary step. Therefore, they are bound to require in the order n^2 such steps. Any improvement must be based on the principle of moving items over greater distances in single leaps.

Subsequently, three improved methods will be discussed, namely, one for each basic sorting method: insertion, selection, and exchange.

2.3. Advanced Sorting Methods

2.3.1 Insertion Sort by Diminishing Increment

A refinement of the straight insertion sort was proposed by D. L. Shell in l959. The method is explained and demonstrated on our standard example of eight items (see Table 2.5). First, all items that are four positions apart are grouped and sorted separately. This process is called a 4-sort. In this example of eight items, each group contains exactly two items. After this first pass, the items are regrouped into groups with items two positions apart and then sorted anew. This process is called a 2-sort. Finally, in a third pass, all items are sorted in an ordinary sort or 1-sort.

One may at first wonder if the necessity of several sorting passes, each of which involves all items, does not introduce more work than it saves. However, each sorting step over a chain either involves relatively few items or the items are already quite well ordered and comparatively few rearrangements are required.
It is obvious that the method results in an ordered array, and it is fairly obvious that each pass profits from previous passes (since each i-sort combines two groups sorted in the preceding 2i-sort). It is also obvious that any sequence of increments is acceptable, as long as the last one is unity, because in the worst case the last pass does all the work. It is, however, much less obvious that the method of diminishing increments yields even better results with increments other than powers of 2.

	44	55	12	42	94	18	06	67
4-sort yields	44	18	06	42	94	55	12	67
2-sort yield	06	18	12	42	44	55	94	67
1-sort yields	06	12	18	42	44	55	67	94

Table 2.5 An Insertion Sort with Diminishing Increments.

The procedure is therefore developed without relying on a specific sequence of increments. The T increments are denoted by $h_0, h_1, \ldots, h_{T-1}$ with the conditions

$$h_{t-1} = 1, \quad h_{i+1} < h_i$$

The algorithm is described by the procedure *Shellsort* [2.11] for t = 4:

```
PROCEDURE ShellSort;
  CONST T = 4;
  VAR i, j, k, m, s: INTEGER;
    x: Item;
    h: ARRAY T OF INTEGER;
BEGIN h[0] := 9; h[1] := 5; h[2] := 3; h[3] := 1;
  FOR m := 0 TO T-1 DO
    k := h[m];
    FOR i := k+1 TO n-1 DO
      x := a[i]; j := i-k;
      WHILE (j >= k) & (x < a[j]) DO a[j+k] := a[j]; j := j-k END ;
      a[j+k] := x
    END
```

```
    END
  END ShellSort
```

Analysis of Shellsort. The analysis of this algorithm poses some very difficult mathematical problems, many of which have not yet been solved. In particular, it is not known which choice of increments yields the best results. One surprising fact, however, is that they should not be multiples of each other. This will avoid the phenomenon evident from the example given above in which each sorting pass combines two chains that before had no interaction whatsoever. It is indeed desirable that interaction between various chains takes place as often as possible, and the following theorem holds: If a k-sorted sequence is i-sorted, then it remains k-sorted. Knuth [2.8] indicates evidence that a reasonable choice of increments is the sequence (written in reverse order)

1, 4, 13, 40, 121, ...

where $h_{k-1} = 3h_{k+1}$, $h_t = 1$, and $t = k \times \log_3(n) - 1$. He also recommends the sequence

1, 3, 7, 15, 31, ...

where $h_{k-1} = 2h_{k+1}$, $h_t = 1$, and $t = k \times \log_2(n) - 1$. For the latter choice, mathematical analysis yields an effort proportional to n^2 required for sorting n items with the Shellsort algorithm. Although this is a significant improvement over n^2, we will not expound further on this method, since even better algorithms are known.

2.3.2 Tree Sort

The method of sorting by straight selection is based on the repeated selection of the least key among n items, then among the remaining n-1 items, etc. Clearly, finding the least key among n items requires n-1 comparisons, finding it among n-1 items needs n-2 comparisons, etc., and the sum of the first n-1 integers is $(n^2-n)/2$. So how can this selection sort possibly be improved? It can be improved only by retaining from each scan more information than just the identification of the single least item. For instance, with n/2 comparisons it is possible to determine the smaller key of each pair of items, with another n/4 comparisons the smaller of each pair of such smaller keys can be selected, and so on. With only n-1 comparisons, we can construct a selection tree as shown in Fig. 2.3. and identify the root as the desired least key [2.2].

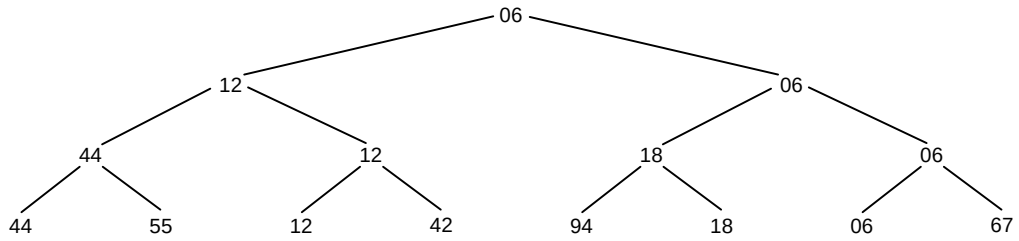

Fig. 2.3. Repeated selection among two keys

The second step now consists of descending down along the path marked by the least key and eliminating it by successively replacing it by either an empty hole at the bottom, or by the item at the alternative branch at intermediate nodes (see Figs. 2.4 and 2.5). Again, the item emerging at the root of the tree has the (now second) smallest key and can be eliminated. After n such selection steps, the tree is empty (i.e., full of holes), and the sorting process is terminated. It should be noted that each of the n selection steps requires only log n comparisons. Therefore, the total selection process requires only on the order of n*log n elementary operations in addition to the n steps required by the construction of the tree. This is a very significant improvement over the straight methods requiring n^2 steps, and even over Shellsort that requires $n^{1.2}$ steps. Naturally, the task of bookkeeping has become more elaborate, and therefore the complexity of individual steps is greater in the tree sort method; after all, in order to retain the increased amount of information gained from the initial pass, some sort of tree structure has to be created. Our next task is to find methods of organizing this information efficiently.

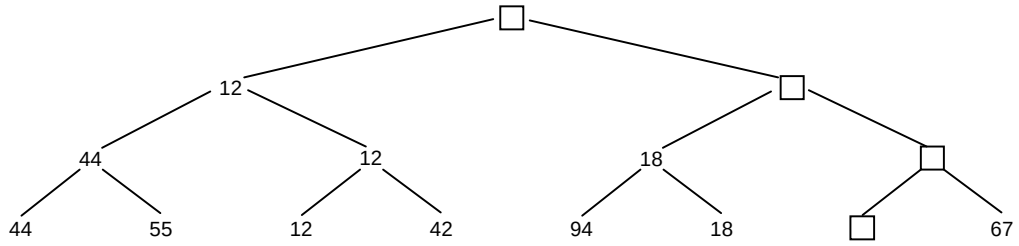

Fig. 2.4. Selecting the least key

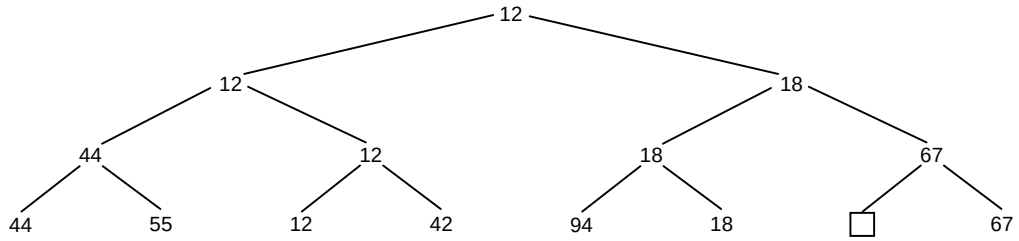

Fig. 2.5. Refilling the holes

Of course, it would seem particularly desirable to eliminate the need for the holes that in the end populate the entire tree and are the source of many unnecessary comparisons. Moreover, a way should be found to represent the tree of n items in n units of storage, instead of in 2n - 1 units as shown above. These goals are indeed achieved by a method called *Heapsort* by its inventor J. Williams [2-14]; it is plain that this method represents a drastic improvement over more conventional tree sorting approaches. A *heap* is defined as a sequence of keys $h_L, h_{L+1}, \ldots, h_R$ $(L \geq 0)$ such that

$$h_i < h_{2i+1} \quad \text{and} \quad h_i < h_{2i+2} \qquad \text{for } i = L \ldots R/2-1$$

If a binary tree is represented as an array as shown in Fig. 2.6, then it follows that the sort trees in Figs. 2.7 and 2.8 are heaps, and in particular that the element h_0 of a heap is its least element:

$$h_0 = \min(h_0, h_1, \ldots, h_{n-1})$$

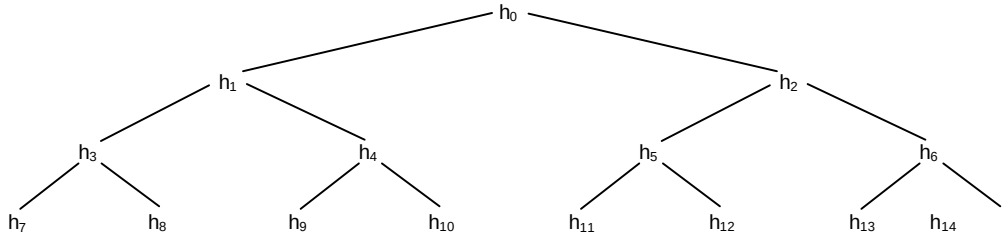

Fig. 2.6. Array viewed as a binary tree

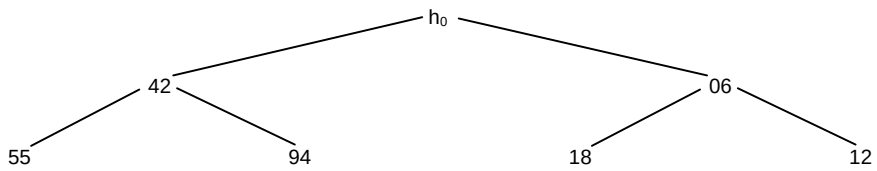

Fig. 2.7. Heap with 7 elements