ETH

Eidgenössische Technische Hochschule Zürich

Institut für Informatik

N. Wirth

DATA STRUCTURES AND ALGORITHMS

J. Gutknecht

ELEMENTARY PRINCIPLES OF INFORMATICS

August 1984

60

Address of the Authors:

Institut für Informatik
ETH-Zentrum
CH-8092 Zürich / Switzerland

Data Structures and Algorithms

N. Wirth

Abstract

This article attempts to outline the basic concepts of structuring data and of constructing algorithms in a methodical manner. The methods are explained with a few simple and typical examples. The programmer's principal concerns for correctness and efficiency require precise mathematical reasoning. Predicate calculus and probability calculus emerge as the indispensible tools for the modern professional programmer.

This is the original version of the copy-edited and improved article to be published in the September 1984 issue of *Scientific American*.

Computers *are* data structures and algorithms. The *data structures* consist of registers and memory words, and the *algorithm* interprets data stored in memory as a program according to a fixed, never changing pattern of behaviour. But the programs we submit to computers are formulated in terms of numbers, texts, events, and structures like sequences, lists, trees, and they represent highly individual methods for problem solutions. How can so vastly differing problems be solved by one and the same machinery that always acts according to a fixed algorithm?

The explanation is that a computer is a *truly general purpose-device* which, based on the idea of John von Neumann, can process and generate data according to a program A, and in the next moment interpret these data as program B. This allows us to formulate a program in terms of notions that are familiar to us and which represent convenient abstractions to express the problem at hand. Thereafter we let a program A - called a *compiler* - map these notions automatically onto facilities that the computer consists of. It is this very technique which enables us to construct systems of phenomenal complexity, because their specification in terms of hierarchically *structured* abstractions permit us to understand one part at a time, and to abstract - to ignore - other parts and their details for the time being. This contrasts sharply with the situation of a programmer being confronted with a vast, unstructured, homogeneous mass of bits called a computer store. How can he manage to understand their inherent meaning in the absence of any distinguishing features? But no program should leave a programmer's desk unless he has a fully understood it. The fact that this axiom is often violated manifests itself in a computer's behaviour that is - depending on the programmer's viewpoint - sometimes called a bug, sometimes a feature, and sometimes is portrayed as evidence that computers have their own will.

The fact that the words notion and notation are almost the same is not incidental. We understand a notion if we can formulate it; and for this purpose we need a notation. In the realm of computing a *precise* notation is mandatory, and therefore we require a *formal notation* with a precise definition. Unfortunately, such formal notations for programming have become known as *languages*. This terminlogy is misleading; it suggests that we speak to computers. The tendency is amplified by the modern term "interactive language," which reveals an unfortunate but widespread confusion between a formal *notation* to specify behaviour and a *mechanism* which interprets texts specified in the formal notation. I prefer to regard programming as the activity of *designing* a new machine (to be implemented with the aid of an existing, general-purpose device). The design is specified in terms of the facilities provided by the formal notation, just like an electronic circuit is specified in terms of symbols and lines connecting them. If we regard *programming as machine design*, the necessity of precision becomes even more obvious, and thereby also the absurdity of so called natural language programming, i.e. of using imprecise expressions of conventional language instead of precise, formally defined symbolisms.

The notation, then, should express the abstract concepts which constitute the foundation of programming, of data structures and of algorithms. It evolves according to the choice of these concepts - and not vice-versa. Which are these concepts? We shall first take a look at data structures, and then at algorithms, which are the recipes to process the data.

All data stored in a digital computer are stored in the form of *binary digits* (bits). A bit has exactly two possible values, say 0 or 1. As programmers, we prefer to reason in terms of abstractions at a higher level, such as characters, numbers, symbols, sets, etc., because they have useful properties and are governed by laws that we have learnt to master and employ. Each such value can be represented by a (small) set of bits, just like numbers, themselves being abstractions, can be represented by (short) sequences of concrete, visible digits. Even more significantly, that mapping can be performed automatically by the computer itself such that the programmer need not even be aware of the internal concrete representation on the bit-level. The programs which perform this automatic mapping are called *assemblers* or *compilers*.

It is the characteristic of so-called high-level languages that they permit the programmer to attribute meaningful names (*identifiers*) to the variables of his program, and to specify the *type* of a variable. A type indicates, whether a variable's values will be integers, real numbers, characters, or sets of numbers, to cite just a few examples. It holds the clue to the interpretation of the bits representing the variable's value (s. Fig. 1). The compiler can then check automatically whether or not the operations on this variable are consistent with this type declaration throughout the program. Such use of redundancy for consistency checking is an invaluable aid in programming practice. It helps in dectecting mistakes at the design stage instead of "in the field".

The notion of data type has been extended - primarily through the programming language Pascal - to encompass also the description of *structure*. In mathematics, the notion of type has been instrumental in resolving paradoxes, such as Russell's paradox of the set of all sets that are not members of themselves. In computing, the primary motivation for introducing types is different. The association of a type (which never changes during computation) with a variable introduces highly desirable redundancy to be used in consistency checking, it determines the set of components (elements) in the case of a structured variable (and thereby enables a compiler to allocate the necessary amount of storage (bits) to the variable), and it provides information about the intended method of access to elements.

Assume, for instance, that a certain problem postulate the existence of a variable s with N elements s_0, s_1, ... s_{N-1}. In solving a mathematical problem, we gladly ignore questions about representation and how elements are accessible; in computing we cannot. Therefore, we not only specify in the variable's declaration the type(s) of the elements, but also their number and method of storage.

If all elements are of identical type T, the structured type is called *homogeneous*, and we declare it as a so-called *array*

```
        s: ARRAY [0 .. N-1] OF T
```

This permits us to denote and access individual components using a *computed index*, i.e. the identification of the element will in general depend on the preceding computation: s_i, s_{i+j+1}, s_{s_k}. No matter what the computed index is, we know that the element's type is T. And, remember, the more information we obtain without actually executing the program, i.e. by a mere textual scan, the better we are off, because that information holds for *all* possible computations specified by the program, and not only for a test case with specific input values (s. Fig. 2).

In the case of an array, each element requires the same amount of storage. Hence, computation of an element's location in store, its *address*, is straight-forward and efficient:

$$adr(s_i) = adr(s_0) + i * size(T)$$

If, on the other hand, the elements are of different types, this simplicity is lost. However, quite likely the necessity to identify an element through a computed index is now small, because the elements, being of different types, obviously represent different attributes of s. We therefore sacrifice the accessibility via computed index and exchange it for the convenience to denote the elements by individual, characteristic identifiers. We say that the variable is a *record* (or has record structure), and call the elements *fields*.

```
s: RECORD
     name: ARRAY [0 .. N-1] OF CHAR;
     longitude, latitude: REAL;
     altitude, population: CARDINAL
  END
```

Individual elements (which evidently may themselves be structured again) are denoted as, for example, s.name, s.population, s.name[3].

Again, their types can be determined without program execution. Furthermore, a compiler can associate with each field an offset, such that the element's address is computed as

$$adr(s_i) = adr(s_0) + offset(s_i)$$

In addition to the array and the record, the structure called *set* is fundamental. We use it, when not the values of elements, but only their presence (or absence) matters. In this case, we declare a variable s as

```
s: SET OF T
```

and instead of an operation for selecting an element we introduce the membership operation $i \in s$ yielding the logical truth value TRUE, if i is an element of s, and FALSE otherwise. The attractivity of this structure stems from both our familiarity with the mathematical concept of sets, and from the efficiency with which set operations can be implemented. If set values are represented by sequences of bits, where the i'th bit denotes the presence of the i'th element (s. Fig.3), set intersection is easily implemented as a logical AND operation executed on corresponding bits, and set union likewise as a logical OR operation.

Arrays, records, and sets are called *basic* structures (or structuring principles). In many practical applications, however, more complicated structures are required. Instead of postulating additional, more sophisticated structures and inventing standard notations for them, it is appropriate to introduce merely a single, general facility to build arbitrary structures. This is appropriate, because complicated structures, unlike arrays, records, and sets, do not remain constant during execution of the program, but grow, shrink, and alter their topology. In contrast to the basic structures, which are called static, they are *dynamic*. The size and form of such structures cannot be specified by an invariant declaration, but is defined by the algorithmic part of the program. Therefore, the program also specifies the allocation of storage. This appears to be a very dangerous situation, loaded with pitfalls and difficult to master, for nothing seems to be fixed by the program text. Fortunately, there does exist a property of such structures that can be regarded as fixed and can therefore be specified in a declaration, namely the types of components of which the structure ultimately is composed. Their number and the connections between them, however, alter during program execution.

The general constructor facility mentioned above then consists of a way to generate (allocate) basic components, so-called *nodes*, and to establish *connections* between nodes. The latter we represent by explicit values, so-called *pointers*, and we introduce types whose elements are pointers. If, for example, a variable p is declared as

```
        p: POINTER TO T
```

its value is either a pointer (a connection) to an element (a node) of type T, or it is NIL, in which case it points to no element. The beauty of this solution is that it allows us to construct data structures of arbitrary topology, complexity, and size. The elements of such structures are usually records; some of their fields are pointers. Hence, the possibility of recursive structures is included; but also simple lists, queues, rings, and trees are perfectly feasible. Implementation of the concept is straight-forward: pointers are represented as addresses. A vital property is that every pointer type is bound to a fixed referenced type. Unlike addresses in basic machine coding, a pointer variable is known to always point to an element of the same given type, except if its value is NIL. As an example, Fig. 4 shows various structures built with the following record types for their nodes:

```
        TYPE ListNode =
            RECORD data: INTEGER;
                next: POINTER TO ListNode;
            END


        TYPE TreeNode =
            RECORD data: INTEGER;
        left, right: POINTER TO TreeNode
    END
```

Let us now turn our attention to algorithms. An *algorithm* is a recipe for a computation. It is specified in terms of a formal notation, and hence the specification is a formal text called *program*. There exist at least as many algorithms as there are computational problems. The first algorithms known were invented to solve *numeric problems*, such as multiplying numbers, finding the greatest common divisor, computing a trigonometric function, or the logarithm, for integrating an analytic function, for inverting a matrix, finding the solutions of a system of equations, and so on. All these algorithms stem from mathematical problems, and they are used in scientific computations in research and engineering. In contrast to them, we find *non-numerical* algorithms, e.g. for searching a minimal element in a sequence, for searching a given word in a text, for sorting (ordering) sets of data, for converting (translating) texts or programs, for scheduling events, etc. They are called non-numeric because their objects, their data, are not necessarily numbers, and because no deep mathematical concepts are needed to specify, understand, and solve them. This, however, does not imply that mathematical concept would not be essential for finding good solutions, for proving their correctness, and to determine their effectiveness. Quite to the contrary, rigorous, mathematical methods have recently begun to influence programming to a considerable degree. They are providing hope that mastery of complex problems will not remain a craft relying on the principle of trying and testing. It has become obvious that *programming is a highly mathematical discipline*. The roles have interchanged, however: instead of employing computing methods to solve mathematical problems, mathematical methods are applied to solve computing problems.

Rather than providing a general survey over various categories of algorithms, which necessarily would have to remain woefully incomplete, we shall concentrate on a few examples to demonstrate the new approach in reasoning about algorithms. We shall not emphasize a particular methodology for constructing new algorithms, but rather present a given solution, and show how mathematical reasoning is employed to understand an algorithm and to gain confidence in its correctness without resorting to a computer to "test a few cases". In this, we shall confine our discussion to purely sequential algorithms, thereby avoiding the additional and interesting problems that arise when several concurrent activities have to be coordinated. We may therefore confine our arsenal of notations to facilities describing *sequencing* (1), *conditional execution* (2), and *repetition* (3).

```
(1)  S0; S1
(2)  IF B THEN S END
(3)  WHILE B DO S END
```

Here B is a condition, also called *Boolean expression*. In (2), the statement S is to be executed, if the preceding evaluation of B yields the value TRUE; in (3) it is to be repeated, as long as the (renewed) evaluation yields TRUE. Evidently, S is executed only if B is satisfied; hence B is a protecting condition for S, called a *guard*.

Our first, simple algorithm shall describe the multiplication of two natural numbers x and y in terms of repeated addition. We introduce variables a, b, and c to represent multiplicand, mutiplier, and partial sum, which ultimately becomes the product. Initially, we set a to x, b to y, and c to zero, and our first solution is

```
a := x; b := y; c := 0;
WHILE b > 0 DO
    b := b-1; c := c+a
END
```

Here we use b as a counter; in each repetition, b is decremented by 1 (observe that the guard $b > 0$ allows us to do so), and a is added to c. We have full confidence in this straight-forward solution, because we know that, starting with $c = 0$, adding a b times yields $c = a*b$. We are also guaranteed to reach the goal, because each decrementing of b brings us closer to the negation of $b > 0$. The reader whose confidence in the proposed solution is still shaky will probably rush to his computer to try a few cases. But he will quickly become aware of the unsatisfactory nature of this approach to gaining confidence, for the number of potential test cases is (at least in theory) infinite. We should like to have a method at our disposal which caters for *all* cases, without "executing" even a single one.

The principal idea is that we assert certain conditions which hold at specific points of the program independent on whatever path the computation had previously taken. If such an *assertion* is placed at the beginning of a repetition (a so-called loop), then it is called a (loop-) *invariant*, because it invariably holds no matter how often the repetition had already been executed. The specification of assertions and invariants is shown below for the multiplication algorithm, first for the specific case with initial values $a = 7$, $b = 3$, where the loop is unfolded and specific numeric values can be specified after each repetition, then for the general case, where an invariant assertion is indicated that holds after each repetition. Assertions are enclosed in curly brackets and may be regarded as comments.

```
a := 7; b := 3; c := 0;
{a = 7 ∧ b = 3 ∧ c = 0}
b := b-1; c := c+a;
{a = 7 ∧ b = 2 ∧ c = 7}
b := b-1; c := c+a;
{a = 7 ∧ b = 1 ∧ c = 14}
b := b-1; c := c+a;
{a = 7 ∧ b = 0 ∧ c = 21}

{x, y ≥ 0}  a := x; b := y; c := 0;
WHILE {a*b + c = x*y ∧ b ≥ 0}  b > 0 DO
    b := b-1; c := c+a
END
```

```
{ab + c = x*y ∧ b = 0 yields c = x*y}
```

We now recognize this formulation to be a specific case of the more general statement

```
a := x; b := y; c := 0;
WHILE b > 0 DO
      decrement b under invariance of P
END
```

where P is the invariant equation $ab + c = xy$ and decrementing is understood not to lead to negative values. Unfortunately, the goal $b = 0$ is approached at minimal speed; an efficient algorithm would require fewer steps. Multiplication, as we had learnt it in primary school, indeed functions according to the very same principle, but it reduces b in larger steps, namely by *dividing* the multiplier b by 10, an operation that is notoriously easy when using the decimal number system: the quotient is obtained by removing the last digit, and this digit is the remainder according to the equation

$$10 * (x \text{ DIV } 10) + (x \text{ MOD } 10) = x$$

Before using division, we must remember that decrementing b must be done while maintaining the invariant P. This is easily achieved by adjusting the other variables a and c appropriately. When we divide b by 10, we also multiply a by 10, thus leaving the product a*b unchanged; if the remainder of division is not zero, we must additionally make a correction to c, namely adding to it a times the remainder.

```
a := x; b := y; c := 0;
WHILE {a*b + c = x*y ∧ b ≥ 0}  b > 0 DO
      c := c + (b MOD 10) * a;
      a := 10*a;  b := b DIV 10
END
{ab + c = x*y  ∧  b = 0  yields  c = x*y}
```

The very same algorithm is used in all computers. Instead of base 10, computers use the binary number system, i.e. base 2. Although our algorithm functions just as well if we exchange 10 by 2, a version which takes advantage of obvious simplicfications is shown below. Note that x MOD 2 can only result in 0 or 1.

```
a := x; b := y; c := 0;
WHILE {a*b + c = x*y ∧ b ≥ 0}  b > 0 DO
      IF ODD(b) THEN c := c+a END ;
      a := 2*a; b := b DIV 2
END
```

The improvement obtained by using division instead of subtraction is very substantial. Instead of executing the repetiton b times, it suffices to execute it log b times. The effort required for multiplying and dividing by 10 (or 2) is negligible, because it is achieved by shifting by one digit position. Without this improvement, computers would spend most of their time multiplying!

In this first example we have seen that a program is a formal text expressing an algorithm, and that this text can be subjected to mathematical manipulations. These are needed to convincingly demonstrate the program's correctness under all specified circumstances. They are much more powerful than empirical testing with the aid of even the fastest computer.

Our second example is from the non-numerical variety, but it is equally fundamental as the first. Assume a text t, given as a sequence of characters, and a word w, being a (shorter) sequence of characters. Let us find the first occurrence of w in t. (We could just as well use other elements in the sequences; characters are used because searching a text is such an intuitive example). First, we wish to specify the desired result in a precise, formal manner. We begin by declaring the involved variables:

```
t: ARRAY [0 .. M-1] OF CHARACTER
w: ARRAY [0 .. N-1] OF CHARACTER
```

We shall assume $0 \leq N \leq M$ (usually $N \ll M$), and will make use of some additional index variables i, j, and k. The result is to be the position (index) i of the first matching subsequence, i.e. the result R shall be

$$R: \forall k: (0 \leq k < N): w_k = t_{i+k}$$

(Note: we use the quantified predicate $\forall k: (x \leq k < y): P_k$ to mean that for all values of k between x and y the predicate P_k holds). Remember, however, that we are supposed to find the *first* occurrence of w in t. This implies that all subsequences of t to the left, i.e. with lower index i, must *not* match. A mismatch at position i is established, if there exists at least one unequal pair of characters in t and w. We denote a *mismatch at position j* by

$$P_j: \exists k: (0 \leq k < N): w_k \neq t_{j+k}$$

(Note that this results immediately from de Morgan's theorem: $\exists k$: NOT P_k = NOT $\forall k$: P_k). We introduce the predicate Q_i to denote *mismatches at all positions to the left of i:*

$$Q_i: \forall j: (0 \leq j < i): P_j$$

and can now specify the complete result $R' = \text{NOT } P_i \wedge Q_i$. But now we have forgotten to specify the result in case that no match exists at all. Such an oversight in specification is indeed a frequent source of program failures. We correct this by specifying that in this case the index value i be greater than the largest possible value for a match:

$$R'': (i \geq M\text{-}N \vee \text{NOT } P_i) \wedge Q_i$$

The obvious approach to establish this result is by a repetition. Each step consists of checking for a match at the next position i, maintaining the loop-invariant Q_i.

```
i := 0;
WHILE {$Q_i$} $(i < M-N) \wedge P_i$ DO  i := i+1  END
```

Since the terminating state is Q_i combined with the negation of the continuation condition between the symbols WHILE and DO, the state after termination is indeed the requested R". Here, P_i suddenly occurs not only as an assertion, but as a condition to be evaluated during execution of the algorithm. It is therefore our duty to decompose it further until the expression consists of individual character comparisons. Again, the approach is stepwise; we repeat comparing individual characters until a mismatch is established. As invariant we use EQ_j: $\forall k: (0 \leq k < j): \; w_k = t_{i+k}$.

```
j := 0;
WHILE {$EQ_j$} $(j < N) \wedge (w_j = t_{i+j})$ DO  j := j+1  END
{$EQ_j \;\wedge\; ((j \geq N) \vee (w_j \neq t_{i+j}))$ }
```

Notice that $j = N$ implies EQ_N, i.e. a match at i, and that $j < N$ implies termination due to an unequal pair, i.e. a mismatch at i. Hence, our complete algorithm to find a word w in a text t is

```
i := 0; j := 0;
WHILE {$Q_i$} $(j < N) \wedge (i < M-N)$ DO
    j := 0;
    WHILE {$EQ_j$} $(j < N) \wedge (w_j = t_{i+j})$ DO  j := j+1  END ;
    IF j < N THEN  i := i+1  END
END
{R":  $Q_i \wedge$ (NOT $P_i \vee i = M-N$)}
```

This algorithm is straight-forward, but relatively inefficient. One begins to wonder, whether a faster solution could be devised. The focus of our search for an improvement must be on the incrementation of i. But how could we increment i by more than 1 without checking for mismatches in between? At first sight this seems impossible indeed, but in 1976 R.S. Boyer and J.S. Moore have nevertheless found a faster way. We shall here present a simplified version of their technique with the main purpose to demonstrate the reasoning employed to establish the correctness of the solution. In passing we note how seemingly well-known problems are still amenable to the search for better solutions.

The clever idea is the following: First, instead of starting the comparisons at the beginning of word w, start at its end proceeding backwards; second, if a mismatching pair is found (and mismatches are highly likely), immediately advance the word up to the last character in the word which matches.

For reasons of simplicity, we shall let i point to the end of the word rather than to its beginning:

```
i := N-1; j := N-1;
WHILE {$Q_{i-(N-1)}$} $(i < M) \wedge (j \geq 0)$ DO
    j := N-1;
    WHILE $(j \geq 0) \wedge (w_j = t_i)$ DO j := j-1 END ;
    i := i + ?
END
```

Of course, there is always a price for every advantage gained. Here it is the need for additional information about the character positions in the word. However, this information can be established once only, namely before starting the actual search. If the text is long enough, it will pay off. Let the information be represented in the form of a table of positions

d: ARRAY CHARACTER OF INTEGER

For each character x in the alphabet (character set), let d_x be the distance from the end of the last occurrence of x in the word. If x does not occur at all, let it be N, the length of w. We express these facts formally as

$$\forall x: NE(x) \wedge ((w_{N-d_x-1} = x) \vee (d_x = N))$$
$$NE(x) \;=\; \forall k: (N-d_x \leq k < N-1): w_k = x$$

Fig. 5 shows an example with N = 5 and a word w = "DEFAB". Note that the last character forms an exception; it does not obtain the value $d_B = 0$ but, since there is no reoccurrence, $d_B = N$.

In the straight-forward search algorithm we had cautiously incremented i by merely 1, and therefore, in order to guarantee invariance of Q_i, had to establish the single term P_i only. Now we intend to increment i in the larger step d_x, where $x = t_i$. Hence we must, in order to maintain Q_i, demonstrate that P_j holds for $j = i-(N-1), i-(N-1)+1, \ldots, i-(N-1)+(d_x-1)$. To this end, we use the fact that $w_k \neq t_i$ implies P_{i-k}. Then the specification of d tells us that

$$\forall k: (N-d_{t_i} \leq k < N-1):\; P_{i-k}$$

and a simple substitution of the bound variable k yields

$$\forall j: (i-(N-1) < j \leq i-(N-d_{t_i})): P_j$$

This is almost the required result with the exception of the term $P_{i-(N-1)}$. Notice, however, that if none of the P_j were established, we would reach $j < 0$, i.e. we find a match and terminate the search.

Correctness and efficiency are the principal concerns of programmers. We have shown that an analytic treatment can and must be used to establish correctness, because an exhaustive empirical test would take far too long for even simple problems.

Whereas the tool for demonstrating correctness is the first-order predicate calculus, the tool for analyzing efficiency and performance is the probability calculus. These two calculi indeed form the mathematical basis on which programming rests. Why should probability come into play? An example may help to find an answer.

Assume an array a of n numbers. Let us find the maximum among them. The obvious way is to scan the array sequentially, in each step comparing the next element with the previously found maximum:

```
max := $a_0$; i := 1;
WHILE $\{\forall j: (0 \leq j < i): a_j \leq max \wedge i \leq n\}$ i < n DO
    IF $a_i$ < max THEN max := $a_i$ END
    $\{\forall j: (0 \leq j \leq i): a_j \leq max\}$
    i := i+1
END
$\{\forall j: (0 \leq j < n): a_j \leq max\}$
```

Quite evidently, the step is repeated n times. But how often is a value assigned to *max*? If y chance the maximum is a_0, the assignment is done once only. If, on the other hand, a is a strictly increasing sequence, it is executed n times. Hence, 1 and n are the extremal values; what is the *average*? Let it be clear that this average cannot be determined experimentally, because even for small n examining all n! cases would take too much effort. Assuming that a single test take 1 us, examining all permutations of 16 elements would require more than half a year's time!

The analytic method to determine the average is, on the other hand, quite simple: The initial assignment to the variable *max* is always executed once, setting our count to 1. The chance that the second element is larger than the first is 1/2, always assuming that all possible permutations are equally likely. *max* is replaced a third time, if a_2 is larger than both a_0 and a_1; the chance for this is 1/3. Continuing this argument, our weighted count becomes the sum

$$ave = 1 + 1/2 + 1/3 + \dots + 1/n$$

which is known as the harmonic series, for which the mathematician L. Euler (1707-1783) had computed the approximation $H(n) = \ln(n) + g$, where g is Euler's constant 0.577... . Evidently, the effort required for the replacement of max is entirely negligible compared with the effort for the comparison and repetition count. It is therefore fair to say that the effort to find the maximum among n numbers is proportional to n.

Most practical problems do not yield to an analytic treatment so readily. Therefore, the *analysis of algorithms* is a field of active research. Often it suffices to know the order of magnitude of required effort, for instance whether it grows proportionally with some parameter, with the square, or even exponentially. In the latter case, the algorithm is useless for all practical purposes. This field of research is called *complexity analysis*. We shall now demonstrate such analysis on hand of a simple, yet eminently practical example, namely the construction of binary trees.

Binary trees characterize an organization which is frequently used to store sets of data for quick retrieval. For example, Fig. 6 shows a binary tree with twelve elements, each being identified by a so-calley *key*. The characteristic property of an ordered binary tree is that, for each node, elements having a smaller key are to be found in its left, those having a larger key in its right subtree. This makes a search very effective, for every comparison halves the set of potential candidates. The expected number of required key comparisons is equal to the average *path length*, the path length to a node being the number of nodes encountered to reach. The sum of all path lengths in Fig. 8 is 40, the average therefore 40/12. In the optimal case of a perfectly balanced tree with n nodes, the average is about $\log_2 n$; in the worst case of a tree degenerated to a list it is n/2.

When constructing the tree, we therefore might take care to keep it balanced. But this in itself takes a considerable effort, and the question must be asked, whether this extra effort will pay off in faster searches later on. Surprisingly, in most cases it does not.

Assume that n values will be read sequentially and, starting with an empty tree, inserted in the tree. We postulate that insertions will be such that the tree always remains ordered. Hence searches will always be possible in an efficient manner. Typically in such applications, only a single operation is made available: a search which, if unsuccessful, inserts the new element at the place where it belongs. Such an algorithm is conveniently formulated in recursive fashion. The tree consists of nodes as declared in the examples of Fig. 4 and starts out with the value t = NIL.

```
PROCEDURE search(VAR t: TreeNode);
(* search key x in t, insert it if not found *)
BEGIN
    IF t = NIL THEN insert new node with key x
    ELSIF x < t↑.key THEN search(t↑left)
    ELSIF x > t↑.key THEN search(t↑right)
    ELSE (* found: t↑.key = x *)
    END
END search
```

Since we do not bother to balance the tree, but insert elements just as they come, this method is also called *random insertion*. The question then arises, what the average path length of such trees will be. We know that it will at best be $\log_2 n$ and at worst n/2. Once again, empirical measurements are out of the question, because the number of possible trees is beyond imagination even for moderate numbers of nodes.

In the following we shall, without impeding on generality, assume that the keys will be the integers 1, 2, ... , n, and that all n! permutations will be equally probable. We denote the path lengths of the tree with n nodes by p_n. Assume, then, that the key i arrives first and therefore will be at the root of the tree. The chance for this to happen is 1/n, and the tree can be depicted as in Fig. 9. The left subtree will have i-1 nodes (remember that all nodes with keys less than i must be there). Its path length is by definition p_{i-1}. The right subtree has n-i elements and therefore path length p_{n-i}. This yields the formula for the total path length as the sum of the contributions of the left subtree, the root, and the right subtree.

$$p_n(i) = ((i-1)*(p_{i-1}+1) + 1 + (n-i)*(p_{n-i}+1)) / n$$

with $p_1 = 1$, $p_0 = 0$. The recursiveness of this definition should not surprise, because both the algorithm and its underlying data structure are recursive. We now must average this value over all i from 1 to n. With the aid of some formula manipulation we obtain a simple recursive definition

$$p_n = ((n^2-1)\, p_{n-1} + 2n - 1)/n^2$$

and this can be expressed explicitly as

$$p_n = 2(n+1)\, H_n/n - 3$$

where H_n is again the harmonic series. Using Euler's formula, we approximate $p_n = 2 \ln n - 1.845...$

The beauty of this result is that the path length of the average tree constructed without regard to balancing is also logarithmic and differs from the optimal case only by a constant factor of $2*\ln n / \log_2 n = 1.386...$. This fortunately tells us that the average case is much closer to the optimum than to the worst case; in fact it is "only" 38% off.

The two principal concerns of programming, correctness and performance, were exemplified by a few simple, elementary cases. The programs discussed, although non-trivial, were very short. This contrasts sharply with programs in common use which are complicated, involved, and grow as fast as new memory devices shrink. Can the reasoning thus demonstrated be applied profitably to these "real-world" monsters? It is my conviction that they *must*, for complex systems require exact reasoning most urgently. Of course, they may not be verified at the same level of detail; the proofs will be built on higher-level abstractions; and if a large system includes 100 search procedures, we need not conduct the same verification 100 times. In fact, most complicated systems contain astonishingly few "deep" algorithms; they all rely on multiplication, searches, and a few others, interacting in all sorts of variations and combinations. The same method can be applied to understand these variations and their interactions: formal, precise reasoning.

Whereas complex programs, such as operating systems, compilers, data base systems, text editors, process control systems, usually contain few sophisticated algorithms, they characteristically employ complex data structures, whose management is often complicated by the fact that the data are stored on media with widely differing characteristics. Much attention is therefore devoted to the organization of data under these aggravating circumstances, to their allocation and access methods. Very frequently, the right choice of data representation is the key to successful programming, and it may be much more influential on a system's performance than the details of the employed algorithm. But we would be ill advised to expect a General Theory for choosing data structures. It is therefore mandatory to understand the elements of structuring, the basic building blocks and their properties, just as we learn about the fundamental algorithms and methods of effective reasoning about them. The ability to successfully apply this knowledge in constructing large systems is above all a matter of individual engineering skill and experience. In gaining this skill, the programmer must constantly fight complexity, never be satisfied with a solution that he does not fully understand, abhor shoddy workmanship, and never give up the search for simpler, more elegant solutions.

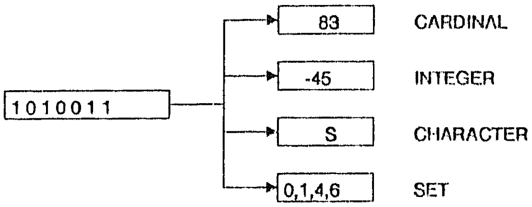


Fig. 1: Interpretation of 7 bits as values of different types

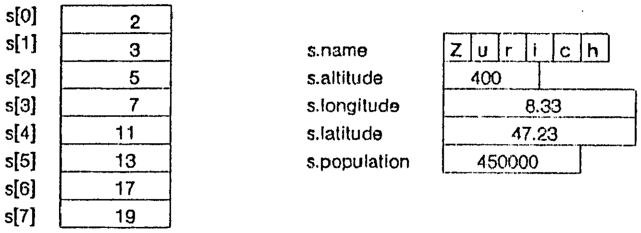
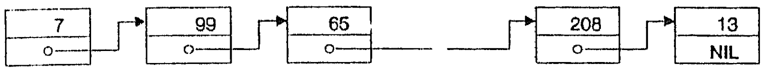


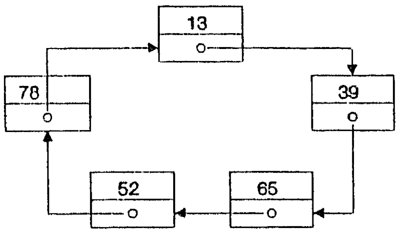
Fig.2: Representation of array and record structures

x = {0, 1, 4, 6}	1010011
y = {3, 4, 6}	1011000
x•y = {4, 6}	1010000
x+y = {0, 1, 3, 4, 6}	1011011

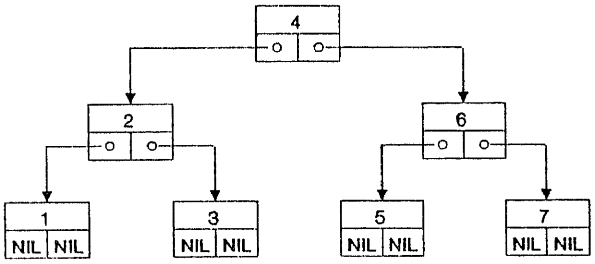
Fig. 3: Set intersection and union computed by AND and OR



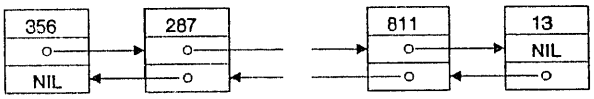
list



ring

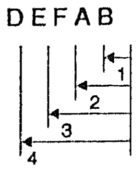


tree



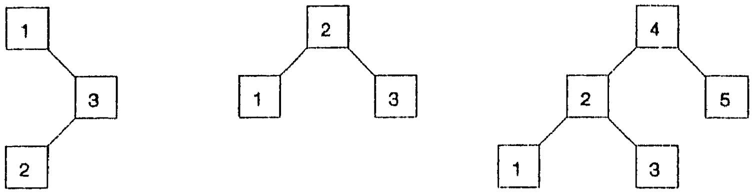
doubly linked list

Fig.4: Dynamic data structures



x	dx
A	1
B	5
C	5
D	4
E	3
F	2
G	5
H	5
...	...

Fig. 5: Table denoting distance of letter from the end



s = 1, 3, 2

s = 2, 3, 1

s = 4, 2, 3, 5, 1

Fig. 8: Trees constructed by "random insertion"

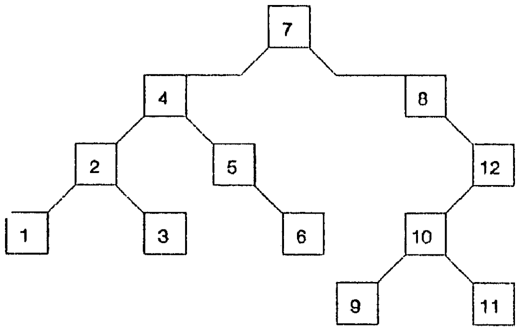


Fig. 6: Binary tree with 12 nodes

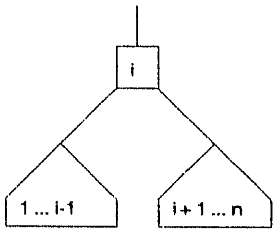


Fig. 9: Tree with root node i has i-1 and n-i nodes in subtrees

Elementary Principles of Informatics

J. Gutknecht

Abstract

This article takes a reader who is familiar with the mathematical way of thinking for a kind of expedition through the realm of elementary informatics. Mainly, the logical principles of computers and the essence of rigorous programming are explained by numerous examples, partly being lighted from a new side.

State

Informatics is in a curious state at the present time: while its fundamental significance is largely undisputed, the opposite is true with respect to its actual matter. This situation is particularly pronounced in the field of education. While the discipline of informatics is being introduced throughout in school curricula, the opinions about the subject matter and standards of achievment radically diverge.

After all, two basic tendencies have crystallized out: the first fights for the position of informatics as an instrument of mathematics and the second establishes a close connection between informatics and commercial and economical applications. It is the purpose of this article to show that neither of these views is essentially appropriate.

Undoubtedly, informatics has grown out of mathematics. The notion of *computer* - denoting the most important instrument of informatics - is a witness for that. Computers were originally meant to relieve mathematiciens of performing calculations which are costly in terms of time but in principle uninteresting.

On the other hand, today, the non-numerical applications of informatics are by far predominant. What made such a drastic widening of the spectrum of applications possible? Foremost, it is the brilliant simple and universal concept of the "computing machines". Actually, computers are *universal devices* rather than machines in the customary sense. Ordinary machines are a priori geared to a specific application. In contrast, computers were inherently designed to process potentially all kind of data in potentially any sense.

A second characteristic of computers is of no less importance for the opening up of new horizons. It is their capability to *store* data. Indeed, for numerous applications, processing is equivalent to storing and retrieving of (typically enormous amounts of) information.

Computers as universal devices

To explain in more detail let us now turn to the major components of a computer. There are essentially two: the so called *processor* and the *storage*. The processor masters a well-defined repertoire of elementary *instructions*. The storage is subdivided into an array of elementary *cells* and serves as a memory of the computer. Both, processor and storage are actually realized in the form of electronic components and circuits, i.e. in the form of *hardware*.

By executing instructions, the contents of the storage cells can be inspected and modified. So, in a sense, the storage cells can take the role of *variables*. It is crucial that the data within a certain cell can be interpreted in very different ways. More precisely, data is coded as a sequence of *binary digits (bits)* and must be decoded according to the current context. For example, arithmetic instructions interpret the contents of involved storage cells as *numbers*.

Our second example of how data can be interpreted is known as principle of *John von Neumann*. It is just this principle which helped the combination of processor and storage on the road to success. A program *P* is a sequence of instructions operating on a set of data. Now, if a code is assigned to each instruction, the program itself, i.e. the sequence of instruction codes can be stored as *data* within the memory.

From this point of view, *executing* a program *P* is a *universal* process. The corresponding universal program *U* can be formulated as a continual repetition of the sequence

Interpret next instruction code; Execute instruction

The hardware of a computer clearly depends on the set of elementary instructions and hence on the universal program *U*. The execution of *U* is usually controlled by hardware. It is noteworthy, however, that hardware depends on nothing else, in particular not on the program *P*. *P* plays the role of a parameter for *U* and can thus be substituted by an arbitrary program *P'* without any need for adjusting the hardware. Programs as *P* and *P'* are therefore said to belong to the category of *software*.

Obviously, the universality of computers is much a consequence of the software concept. Let us take up a seemingly detail in the universal program *U*. What means *next* instruction? Normally, this is the instruction in the next storage cell. But it is perfectly possible and in fact inevitable that this rule is occasionally broken.

Assume, for example that the next instruction depends on the current state of the program. Then, the processor has possibly to "jump" to a new location and there to resume the linear interpretation process. Or, look at the repetitive universal program itself. Having executed an instruction, the next code must be interpreted, i.e. the processor has to jump back to the beginning of the loop.

As we shall see, *alternatives* and *repetitions* are essential components of a program. Hence, jumps are indispensable members in the set of elementary instructions. It is the pecularity of jump instructions that they provide a means for the processed program to direct its processor.

External devices complete a computer to a *computer system*. We can distinguish two classes: devices facilitating *communication* (typically a keyboard to accept input data and a display to write messages and results) and devices serving as *mass storage* (e.g. magnetizable disks). External devices are usually connected with the computer storage. Hence, data can directly be read into and written from storage cells.

Programming

It follows from the foregoing section that a computer *U* can and in fact must be tailored to a specific application by providing an appropriate program *P*. It turned out that the activity of programming obeys certain thought processes which are law-governed and greatly independent of the specific application aimed at.

As a consequence, laws, rules and methods concerning the discipline of programming have been formulated. Programming in its most general sense has developed into the core of information processing or *informatics*. So, what is programming?

We have already declared a program as a sequence of instructions operating on a set of data. Of course, one thinks of a program to be *executed*. Therefore, programming amounts to formulate a *dynamic process* as a *static text*. But things get even more complicated. In general, a program accepts input data on which its execution may depend (remember the universal program *U* accepting program *P* as input).

Hence, a program text describes in general a vast class of processes rather than a single process. To say the program is *correct* obviously means that each of these processes runs correctly, i.e. produces the correct output data (in a finite amount of time). Writing a program correctly is not only a noble goal but necessity, if the program is intended to control an aircraft or an atomic power station.

In most cases "unfolding" a program into all processes which it describes is hopeless. A more promising way to prove its correctness is safeguarding the (static) text itself. Programming in this rigorous sense is a strongly mathematical activity. Notice the remarkable fact that the roles of mathematics and informatics have been reversed in the course of time: mathematics has become an instrument of informatics!

Before we can illustrate these concepts by examples, we have to make a few remarks on programming notations or *programming languages*. Although the program *P* will finally be stored in the computer's memory as a sequence of binary digits, this representation is certainly inappropriate for humans.

Assume that each elementary instruction had assigned a *mnemonic code*, i.e. an identifier which suitably abbreviates the name of the instruction. Assume further that a name could be assigned to each memory cell. Then, the program could be written as a text in readable form.

But how would this text be translated into the sequence of binary digits? Of course, this translation process obeys well-defined rules. Hence, we could provide a program C that translates any program *P*. We shall call such a translation program a *compiler*. The idea of compilers has proven to be extremely productive.

It has initiated a development which is not yet terminated. Probably the most obvious step in this development was the introduction of programming notations which are *independent* of the underlying machine. These programming notations, or programming languages as they are conventionally called, opened a door for the so called *application oriented programming*. In contrast, the compiler *C* for such a programming language, being inherently machine dependent, is counted to the class of *system software*.

The next steps in the development of programming languages can best be characterized by "increasing the level of abstraction". The essence of abstraction is - in this context - the usage of self-defined entities ("molecules") of possibly different type and complexity. Let us point out here an interesting duality between the data operated on by instructions and the instructions operating on the data. The former represents the *data structures* of a program and the latter the so called *algorithms*.

Elementary storage cells are the "atoms" of data structures, elementary instructions the "atoms" of algorithms. While the process of abstraction for the notation of algorithms has started at the very beginning in the development of programming languages (by necessity, for there is no general set of elementary instructions), the inclusion of data structures is greatly the merit of *Pascal* [1]. Notably, Pascal has introduced the notion of *type* of a data entity. A type is a rule according to that the data entity is interpreted. Numbers and characters are examples of basic types.

Pascal has also popularized the method of *structured programming*, i.e. the method of successively refining the individual actions of an algorithm. This brought a new aspect of programming languages into play: the aspect of the programming language as a tool which supports the programmer in the *design* (not only formulation) of his programs.

Modern languages like *Modula-2* [2] or Ada [3] go even one step further. The key idea behind is a clear separation between the *definition* of the functions of a program part and the methods and techniques used to *implement* them. So, in a large software project where several programmers are involved, the interfaces between the various program parts can be laid out in detail centrally and a priori.

A first set of examples

We have previously stated that the discipline of programming is governed by fundamental rules and laws. We can best explain these and the kind of arising difficulties and methods to surmount them by simple but typical examples.

We shall now introduce a programming notation due to *E.W. Dijkstra* [4]. Let us first define its "instructions", in the sequel called *statements*. A *statement list* is a sequence of statements, separated by ";". Variables (for the time being of an arbitrary type) are denoted as a1, a2, ..., expressions as E1, E2, ..., conditions as C1, C2, ... and statement lists as S1, S2, ...

The skip statement

```
skip                                No operation, skip
```

The assignment statement

```
a1, a2, ..., an := E1, E2, ..., En  Assign (simultaneously) the value
                                    of E1 to a1, ..., of En to an
```

The alternative statement

```
IF C1 -> S1                         If no condition is valid at all, then halt with error
 | C2 -> S2                         else select any valid Ci and execute statement list Si
...
 | Cn -> Sn
FI
```

The repetitive statement

```
DO C1 -> S1                         While any of the conditions is valid,
  | C2 -> S2                        select any valid Ci and execute statement list Si
...
  | Cn -> Sn
OD
```

Notice that this notation, although looking harmless, it is not! The fact that statements or even sequences of statements occur within an IF- and DO-statement has far reaching consequences. Complex hierarchies of nested alternatives and repetitions can thus be constructed.

The conditions Ci somehow serve as *guards.* Si can be executed only if the guard Ci has been passed. That's why the list of Ci's and Si's is sometimes called a *guarded statement.*

In our first programming example we shall deal with the problem of laying out a rectangular area of width *a* and height *b* (*a* and *b* natural numbers) with squares of a maximal extent *w.*

Assume that one of the sides of the rectangle is greater than the other, say a>b. The original problem is obviously equivalent to laying out the reduced rectangle with sides a-b and *b* (see. Fig. 1). This step can be repeated to get a still smaller rectangle.

Does the process of reduction come to a natural end? At the most, if no side of the rectangle is greater than the other, i.e. the rectangle has become quadratic. But, if initially a>0 and b>0, this constellation must always come after a finite number of reduction steps. At each repetition step, *a* and *b* are namely reduced by a positive integer under invariance of the condition a>0 and b>0. (What happens, if the initial value of *a* or *b* is 0?)

What have we gained by this successive reduction? In fact, the result! To see that, apply the problem to the final rectangle which is a square. In this case, the maximal square coincides with the rectangle.

We can now formulate our algorithm in the notation of Dijkstra. We add (in curled brackets) the so called *assertions* in the form of logical predicates.

```
{a>0 and b>0}
DO a > b -> a := a-b {a>0 and b>0}
  | b > a -> b := b-a {a>0 and b>0}
OD
{a=b=extent of desired square}
```

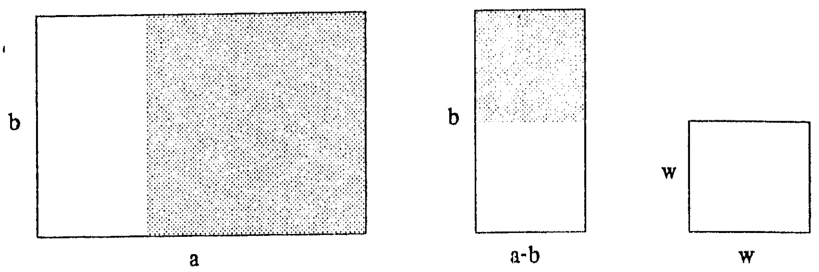


Figure 1. Laying out a rectangular area with squares

We emphasize that the success of this procedure is crucially based on the fact that the extent of the desired square is not changed during a reduction step. Thus, the predicate "a>0 and b>0 and w=extent of desired square" serves as an *invariant* of the repetition. If its validity is guaranted as a *precondition* at the entry point, then the *postcondition* (implying the result) is the conjunction of the invariant and the negations of all guards.

Let us extend our problem to the more complicated region in Fig. 2. If we assign numbers *a*, *b*, *c*, *d* as indicated in Fig. 2, we can see that the idea of reduction to find the maximal square can be carried over.

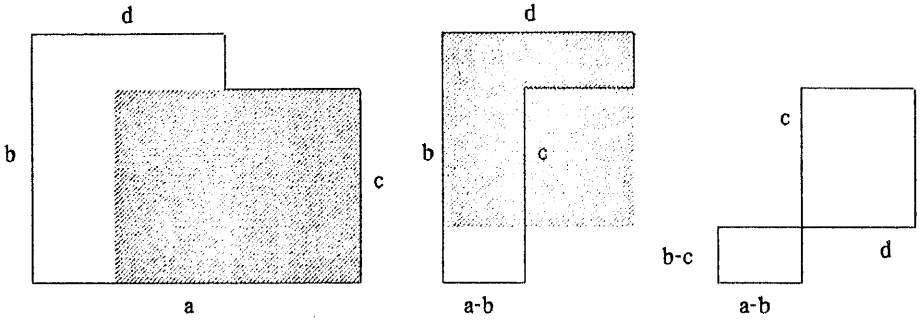


Figure 2. Laying out a horizontally and vertically bounded area with squares

The reduction process now continues as long as at least two of the four numbers are different. We must arrange the guards so that the validity of all their negations implies that all numbers are equal. The following program shows a simple solution. *I* stands for the invariant

```
I: a>0 and b>0 and c>0 and d>0 and w=extent of desired square
{I}
DO a > b -> a := a-b  {I}
  | b > c -> b := b-c {I}
  | c > d -> c := c-d  {I}
  | d > a -> d := d-a  {I}
OD
{a=b=c=d=extent of desired square}
```

It is clear that the same procedure will work for even more complicated areas and also for three-dimensional objects to be filled out by maximal cubes. The reader who is familiar with elementary mathematics certainly has recognized that the mathematical solution of our problem is just the *greatest common divisor (gcd)* of the numbers involved.

It is this mathematical interpretation which underlies the next extension of our algorithm. It is a well-known but noteworthy fact that gcd(A,B) can always be represented as an integral linear combination of *A* and *B*:

$$\gcd(A,B) = xA + yB$$

The key idea in the development of an algorithm which determines integral coefficients *x* and *y* is to extend the invariant by two equations which, finally, coincide with $\gcd(A,B) = xA + yB$.

These invariant equations are $a = xA + yB$ and $b = uA + vB$, where *a* and *b* are auxiliary variables corresponding to the constants *A* and *B*. Our program now read as

```
a, b, x, y, u, v := A, B, 1, 0, 0, 1;
DO a > b -> a := a-b; x := x-u; y := y-v
  | b > a -> b := b-a; u := u-x; v := v-y
OD
```

The guarded statement preserves the predicate

$$a>0 \text{ and } b>0 \text{ and } a = xA + yB \text{ and } b = uA + vB$$

which is therefore an invariant. This invariant and the final condition $a=b=\gcd(A,B)$ imply that (x,y) and (u,v) are solutions. Having already seen that the algorithm terminates after a finite number of repetition steps, we have actually provided a *mathematical proof* for our initial claim.

With the next example we will enter a field which plays a dominant role primarily in non-numerical informatics, namely that of *searching* and *retrieving* data.

Let us start with an extremly simple problem. Given a list a[0], a[1], ..., a[n-1] and a value *x*. Write a program to find an element a[i] whose value equals *x*. The most obvious way to search for such an a[i] is to scan the list, say from left to right, until the value of a[i] equals *x*. We shall call this procedure *linear searching*:

```
i := 0;
DO a[i]#x -> i := i+1 OD
```

Notice that "#" abbreviates "not equal" and that i := i+1 states that *i* is to be increased by 1. The postcondition is just the negation of the guard, i.e. a[i]=x. Thus, if our program properly terminates, it has certainly found a desired element. Is the program correct?

Suppose that the value *x* is not contained in the list at all. Then, the guard of the repetition is never false. But it is undefined if i>=n. Hence, the process will halt with error.

We can remedy our program by strengthening the guard:

```
i := 0;
DO i#n and a[i]#x -> i := i+1 OD
{i=n or a[i]=x}
```

According to de Morgan's law, the postcondition is now a disjunction. If i=n, then no element with value *x* is contained in the list, otherwise *i* is the index of a desired element. But a new difficulty has arisen. If i=n, then the second part of the guard is undefined. Thus, from the point of view of ordinary logic, the whole conjunction is undefined in this case.

Modern programming languages (like Modula-2) circumvent this type of difficulty by interpreting the *and-* and *or-*operator as *conditional and* and *conditional or.* The *conditional and* assigns the value *false*, if the first argument has the value *false*, independent of the second argument. Dually, the *conditional or* assigns the value *true*, if the value of the first argument is *true*, independent of the second argument. Of course, the "conditional"-operators are not commutative!

The reader may find a correct guard for the conventional logic. We shall follow now a new avenue. Correctness is an essential but not the only quality a program must have. *Efficiency* is equally important. It is somehow unsatisfactory that the check if i#n must be done at each repetition step. An elegant and typical way out is to suitably modify the *data structure* instead of the algorithm. If we add a list element a[n] containing the value *x*, then we exclude definitely the case which caused all our problems.

Hence, the first variant of our algorithm, being extended by the statement a[n] := x proves to be optimal. An element with value *x* is not contained in the original list if and only if the repetition terminates with i=n. The element a[n] acts also as a kind of guard; it is occasionally called a *sentinel.*

In most cases it is desirable that a searching algorithm finds *all* occurrences of a value within a list. So, we extend our program to respect this.

```
a[n] := x; i := 0;
DO i#n+1 -> DO a[i]#x -> i := i+1 OD;
            {Process a[i]} i := i+1
OD
```

Assume now that the elements of our list a[0], a[1], ..., a[n-1] represent *measurements.* We want to develop a program that finds the lowest and the highest measurement. Let us first concentrate on the lowest value. We shall maintain a variable *low* representing the current minimum. Thus, the invariant of the repetition is

```
I: low=min(a[0], ..., a[i-1])
```

Together with i=n, *I* yields the result.

```
low := a[0]; i := 1; {I}
DO i#n -> IF a[i]<low -> low := a[i] | a[i]>=low -> skip FI;
          i := i+1 {I}
OD
{I and i=n}
```

If we include analogous actions for a variable *high*, we get the following solution of our problem:

```
I: low=min(a[0], ..., a[i-1]) and high=max(a[0], ..., a[i-1])
```

```
low := a[0]; high := a[0]; i := 1; {I}
DO i#n -> IF a[i]<low -> low := a[i] | a[i]>=low -> skip FI;
          IF a[i]>high -> high := a[i] | a[i]<=high -> skip FI;
          i := i+1 {I}
OD
{I and i=n}
```

Two comparisons per list element are performed by this algorithm. Hence, altogether, the process requires 2n comparisons. One might wonder if this is a minimum. Clearly, if a[i] is smaller than the current value of *low*, a[i] cannot be greater than *high* at the same time. But optimizing this rarely occurring event does not much improve.

If we knew the relation between two elements a[i] and a[j] of our list, then it would be sufficient to test the lower of them against *low* and the higher against *high*. For we can find the relation between a[i] and a[j] by one comparison, we need three comparisons for each pair by this method, i.e. totally 3(n/2) comparisons, if *n* is even. We shall therefore consider the pairs a[i], a[n1-i], where n1 denotes the constant n-1. The invariant is now

```
I: low=min(a[0], ..., a[i-1], a[n-i], ..., a[n-1]) and
   high=max(a[0], ..., a[i-1], a[n-i], ..., a[n-1])
```

Together with *I*, the result is established as soon as i>=n-i, i.e. when i = (n+1) DIV 2, where *DIV* denotes the integer division (non-integral part truncated). We shall denote by n2 the constant (n+1) DIV 2. The gain in efficiency has obviously to be paid with a loss of simplicity. Notice, however, that the following program does not depend on the parity of *n*:

```
IF a[0]<=a[n1] -> low := a[0]; high := a[n1]
 | a[0]>=a[n1] -> low := a[n1]; high := a[0]
FI;
i := 1; {I}
DO i#n2 -> IF a[i]<=a[n1-i] ->
                IF a[i]<low -> low := a[i] | a[i]>=low -> skip FI;
                IF a[n1-i]>high -> high := a[n1-i] | a[n1-i]<=high -> skip FI
            | a[i]>=a[n1-i] ->
                IF a[n1-i]<low -> low := a[n1-i] | a[n1-i]>=low -> skip FI;
                IF a[i]>high -> high := a[i] | a[i]<=high -> skip FI
           FI;
           i := i+1 {I}
OD
```

Let us turn now to a variant of the problem of finding an element a[i] with a given value *x*. Interpret the a[i] as *departure times* of railway trains. The problem is to develop a program which returns the time of the next departure. Here, the reference value *x* is the current time. Try to solve this problem without further assumptions about the list a[0], a[1], ..., a[n-1]. The problem becomes much simpler, if we require the list to be *ordered*, i.e. a[0]<=a[1]<=... <=a[n-1].

The desired result is a[i-1]<x<=a[i], where we have added an element a[-1] of value -infinity (in thought) and an element a[n] of value infinity. We try to keep I: a[i-1]<x invariant. Then, if we declare the guard to be a[i]<x, the postcondition implies the result:

```
a[n] := infinity; i := 0; {I}
DO a[i]<x -> i := i+1 {I} OD
{I and x<=a[i]}
```

The solution turns out to be just a variant of the linear search algorithm presented above. It solves as well our original problem of finding an element a[i] of value *x*.

However, under the assumption of an *ordered* list a[0], a[1], ..., a[n-1], there exists a method which significantly outdoes the linear search algorithm. It is called *binary searching.* The key idea is to succesively halve the number of elements which possibly coincide with *x*.

Thus, we may take as invariant a relation of the form a[i]<x<=a[j]. But, unfortunately, we cannot anchor this relation, for *x* is not necessarily contained within the interval]a[0], a[n-1]]. We therefore extend the list (in thought, as they are not really accessed) by two elements a[-1] and a[n] which are initialized as -infinity and infinity. So, if we set i := 0 and j := n-1, we get as a precondition I: a[i-1]<x<=a[j+1]. This predicate implies the result, if j-i<0. (Notice that I and j-i<0 implies that j+1=i). So, we try to successively reduce the difference j-i by keeping the relation *I* invariant:

```
i := 0; j := n-1; {I}
DO j-i>=0 -> m:=(i+j) DIV 2;
             IF a[m]<x -> i:=m+1 {I}| a[m]>=x -> j:=m-1 {I} FI
OD
{I and j-i<0}
```

The postcondition states that a[i-1]<x<=a[i]. Hence, a final test distinguishes the two outcomes: if x=a[i], then an element with value *x* was found, if not, *x* is not contained in our list and the list elements a[i-1] and a[i] are closest to *x*.

Investigating now termination of our algorithm, we state that, if j>=i, then m>=i and m<=j, i.e. j-(m+1)<j-i and (m-1)-i<j-i Hence, j-i is strongly decreased at each repetition step.

What is the gain in efficiency when using the binary search in place of the linear search algorithm? Linearly searching an element in an ordered list of *n* elements requires n/2 comparisons in average. If our variant of binary searching is used, then log n comparisons are always required (log is the logarithm with respect to base 2). Here, we have assumed *n* to be a power of 2. To visualize the gain, remember that if n=1024, then n/2=512 and log n=10.

With the examples of linear and binary searching we have illustrated that programming is a *goal oriented* activity. It was the result which suggested the invariant and the guard and it was the invariant which practically fixed the repetition step, except the definition of *m*. Indeed, the binary search algorithm is correct and terminates after a finite number of repetition steps whenever it is assured that, if j>=i, then m>=i and m<=j.

Goal oriented proceeding will pay also in our next example which is due to D. Gries [5] and is called the *welfare crook.* In contrast to the searching problems discussed so far, the searching argument is here not explicitly known. Three ordered lists a[0], a[1], ..., a[n-1], b[0], b[1], ..., b[m-1] and c[0], c[1], ..., c[l-1] are now involved. The first contains the names of all students of the New York university, the second the names of all employees of IBM New York, and the third the names of all welfare recipients in New York. The problem is to develop a program which searches for a person who is registered in all three lists.

Notice that the list elements are now strings of characters instead of numbers. We assume them to be *lexicographically* ordered. The postcondition must be of the form a[i]=b[j]=c[k] for some indices *i, j, k*. Hence, to establish the guards we follow the well tried scheme of an earlier example. Furthermore, we include three sentinels a[n], b[m] and c[l]:

```
a[n], b[m], c[l], i, j, k := infinity, infinity, infinity, 0, 0, 0;
DO a[i]>b[j] -> j := j+1
 | b[j]>c[k] -> k := k+1
 | c[k]>a[i] -> i := i+1
OD
```

The postcondition a[i]<=b[j]<=c[k]<=a[i] implies the result. The invariant is here empty, if not the technical condition 0<=i<=n and 0<=j<=m and 0<=k<=l is interpreted as invariant. Clearly, the algorithm terminates after at most n+m+l repetition steps, for one index is increased at each step.

We have seen that ordering of lists is an appropriate means to accelerate and simplify searching processes. We shall now study ordering algorithms. Ordering algorithms belong to the class of the most interesting and most subtle algorithms at all.

Let us start with an easy question. Write a program to order two elements *a* and *b*. If a>b, then we must swap the two variables. A first try to swap *a* and *b* is

```
b := a; a := b;
```

This, of course, fails. The postcondition is namely a=b. Swapping necessitates an auxiliary variable *u*:

```
u := a; a := b; b := u;
```

If we abbreviate this action as swap(a,b), then our program is

```
DO a>b -> swap(a,b) OD
```

The repetition obviously terminates after zero or one step. We now increase the number of variables to be ordered, say to six: *a, b, c, d, e, f.* We want to order them *in situ*, i.e. no other variable (except *u*) should be involved. The ordering repetition must certainly not stop as long as any two neighbouring variables are in the wrong relation:

```
DO a > b -> swap(a,b)
  | b > c -> swap(b,c)
  | c > d -> swap(c,d)
  | d > e -> swap(d,e)
  | e > f -> swap(e,f)
OD
```

We claim that this is already the correct program. First, the termination condition, i.e. the conjunction of the negations of all guards is a<=b<=c<=d<=e<=f. Notice that the sequence of swappings is not determined by this program; remember the definition of the repetition statement!

Four szenarios of how the process could progress with a given setup:

a b c d e f	a b c d e f	a b c d e f	a b c d e f
6 5 2 1 4 3	6 5 2 1 4 3	6 5 2 1 4 3	6 5 2 1 4 3
5 6 2 1 4 3	6 2 5 1 4 3	6 5 1 2 4 3	6 5 2 1 3 4
5 2 6 1 4 3	6 2 1 5 4 3	6 5 1 2 3 4	6 5 1 2 3 4
5 2 1 6 4 3	6 2 1 4 5 3	5 6 1 2 3 4	6 1 5 2 3 4
2 5 1 6 4 3	6 2 1 4 3 5	5 1 6 2 3 4	6 1 2 5 3 4
2 5 1 6 3 4	6 1 2 4 3 5	1 5 6 2 3 4	6 1 2 3 5 4
2 5 1 3 6 4	1 6 2 4 3 5	1 5 2 6 3 4	1 6 2 3 5 4
2 1 5 3 6 4	1 6 2 3 4 5	1 5 2 3 6 4	1 2 6 3 5 4
2 1 3 5 6 4	1 2 6 3 4 5	1 5 2 3 4 6	1 2 3 6 5 4
1 2 3 5 6 4	1 2 3 6 4 5	1 2 5 3 4 6	1 2 3 5 6 4
1 2 3 5 4 6	1 2 3 4 6 5	1 2 3 5 4 6	1 2 3 5 4 6
1 2 3 4 5 6	1 2 3 4 5 6	1 2 3 4 5 6	1 2 3 4 5 6

Does the process always terminate after a finite amount of time? It is sufficient to find an integer valued function which is bounded from above and strongly increased at each repetition step. The ordinary sum of all elements being constant, consider the following *weighted sum*: s := 0a+1b+2c+3d+4e+5f. The upper bound is *s*, being evaluated for the ordered list.

The number of repetition steps clearly depends on the degree of the initial ordering of the list. The reader may try to find out, if it is pure chance that in our example all szenarios result in the same number of steps.

Similar to the situation with searching, algorithms have been developed which are an order of magnitude more efficient than straightforward solutions. One of the most successful is *Quicksort* [6] which we will now roughly discuss.

We can best explain the method of working of Quicksort, if we suppose that our processor wants to delegate two colleagues to do the ordering of the list a[0], a[1], ..., a[n-1]. For that, the processor has to prepare the list, i.e. to divide it up into two "independent" parts. Independent means that each of the colleagues is able to do his part without involving the other. This is effectively the case, if the processor divides up the list into two sublists a[0], ..., a[i-1] and a[i], ..., a[n-1], such that each element of the first sublist is not greater than each element of the second.

This dividing process is not much involved. It is best done by succesively exchanging unmatching elements as we shall subsequently demonstrate. It is surely desirable that both sublists have about the same extent. So, the key problem is to find an appropriate reference element *x*, according to that the splitting up can be done. It is a solution of embarrassment to take for *x* the value of an element around the middle of the list.

```
                         x
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
 22   56   31   75   49   19   87   38   17   54
      ↑                                  ↑
 22   17   31   75   49   19   87   38   56   54
                ↑                   ↑
 22   17   31   38   49   19   87   75   56   54
                     ↑    ↑
 22   17   31   38   19 | 49   87   75   56   54
a[0]              a[i-1] | a[i]              a[n-1]
```

As in real life, each of our two colleagues will have two own assistants who he delegates to do the job. Hence, the original situation recurs. If a sublist consists of one element only, then, clearly, the job is already done and nobody has to be delegated. It is a characteristic of such *recursive algorithms* that the problem is never actually "solved" but merely reduced until it has disappeared.

Ordering is thus carried out by several autonomous *processes*. In fact, delegating an assistant corresponds to creating a new and independent process. In principle, all existing processes could run simultaneously. This would obviously result in a saving of time for the ordering of the whole list. But that property of the algorithm can be fully exploited only if the underlying computer is equipped with an appropriate number of hardware processors. Considering the progress in hardware technology, especially in *VLSI* (very large scale integration (of electronic components)), such computers can be expected in the near future.

But the Quicksort algorithm is attractive even if only one hardware processor is used. If we are lucky with our reference elements, then with each delegation, we can halve the extent of the list to be ordered. And it turns out that ordering two lists of the half size is more efficient than ordering the original list (even if one includes the preparation).

A second set of examples

Having emphasized the algorithmic aspect of computer programs in the previous section, data structures will be in the foreground in this section. Normally, a *data bank* which is processed by a computer program is not a static entity. New items may have to be included or existing items may have to be deleted.

Suppose that we are confronted with the problem of developing a program to maintain a *table of results* for a downhill race. Let each participant be characterized by his *data record* containing as components his name and the time he has achieved in the race.

Then, accepting a maximum of n participants, we declare the *array* p[1], p[2], ..., p[n] of records as base of our data bank. Let us denote the name and the time of participant i by p[i].name and p[i].time respectively. We shall design our program to accept two commands: "insert new participant" and "display current results table". Putting ourselves in a position to insert a new item, we can make out the principal difficulty. Either we insert the items into our array in the sequence of their arrival or in the sequence of the achieved times.

In the former case, we simply fill up the array successively. But then, the array must be timely ordered whenever the "display"-command is given. In the latter case, however, we must rearrange the whole array to correctly insert a new item. Rearranging means moving present items within the array to make room for the new item. In the worst case (if the new participant has achieved the lowest time), *each* entry would have to be moved to the back.

This is obviously not an effective solution (although, in our present example, the damage is limited). As a way out we shall introduce the concept of *chaining*. For that, we add a component *next* to the data record p[i] which contains the index of the (timely) successor of p[i]. We shall interpret *next* as a *pointer* to the next element in the chain. We agree that 0 indicates the end of the chain. Further, we need a variable *head* which points to its beginning.

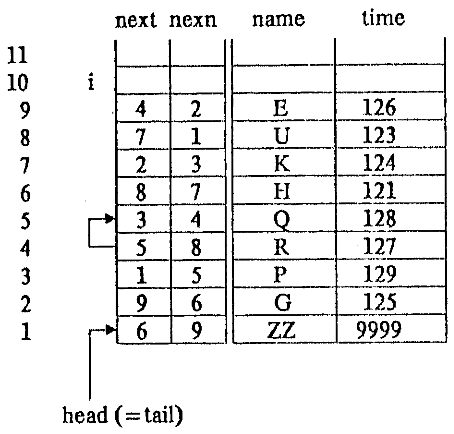
Physically, the items can now successively be inserted into the array. An index *i* is maintained to indicate the next free location. Additionally, however, each new item must be properly linked into the chain. Suppose the data of the new participant to be registered in variables *newname* and *newtime*. Our program has to scan through the chain until an item *x* is found of which the time is greater than *newtime*. Let the scan process look one entry ahead and call *cur* the currently scanned entry. Then, finally, p[cur].next is *x*. The new element p[i] must afterwards immediately precede this item. The correct linkage is thus established by adjusting the *next*-components as follows: p[i] points to *x* and p[cur] points to p[i].

So, our insertion algorithm becomes

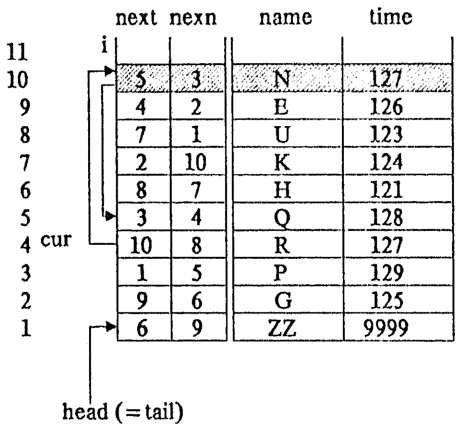
```
{read newname and newtime from keyboard}
p[i].name := newname; p[i].time := newtime;
cur := head;
DO p[p[cur].next].time <= newtime -> cur := p[cur].next OD;
p[i].next := p[cur].next; p[cur].next := i;
i := i+1
```

Is the program correct? It is in general recommandable to consider extreme cases. Here, there are two. They correspond to the situations where the new item becomes the head, respectively the tail of the chain. The former case is certainly incorrectly handled for the element *head* is not inspected at all. But the program also fails in the latter case. The *next*-component of the last item being 0, the guard becomes undefined.

We could try to correct the algorithm, but, like in our very first example of the last



	next	nexn	name	time
11				
10				
9	4	2	E	126
8	7	1	U	123
7	2	3	K	124
6	8	7	H	121
5	3	4	Q	128
4	5	8	R	127
3	1	5	P	129
2	9	6	G	125
1	6	9	ZZ	9999



	next	nexn	name	time
11				
10	5	3	N	127
9	4	2	E	126
8	7	1	U	123
7	2	10	K	124
6	8	7	H	121
5	3	4	Q	128
4	10	8	R	127
3	1	5	P	129
2	9	6	G	125
1	6	9	ZZ	9999

Figure 3. Inserting a new item into time- and name-ring-chain

section, it is preferable to adjust the data structure. We will introduce a definitely last "pseudo-participant". Furthermore, we will link this pseudo-participant to the currently first in the chain, closing in this way the chain to a *ring*.

Thus, the initialization of our program is

```
p[1].next := 1; p[1].time := infinity; head := 1
```

Let us now turn to the handling of the "display results table" command. It is a running through the chain:

```
cur := p[head].next;
DO p[cur].time # infinity -> {write p[cur].name onto display} cur := p[cur].next OD
```

The method of chaining has proven to be extremly helpful in disentangling the conflict between the succession of inserting items and the succession of achieved times. It is practically indispensable if a data bank has to be ordered according to several different criteria at the same time.

In our case, an additional ordering according to names comes in question. For that, we have simply to add a second component to p[i], say *nexn*. The insertion algorithm must be extended by a corresponding statement sequence (*next* by *nexn*) and p[1].name has to be initialized with an "infinitly large name".

The display procedure for the name chain is the same as that for the time chain, where again *next* is replaced by *nexn*. The reader may try to develop a procedure to remove a (disqualified) participant from the results table (but not from the name chain).

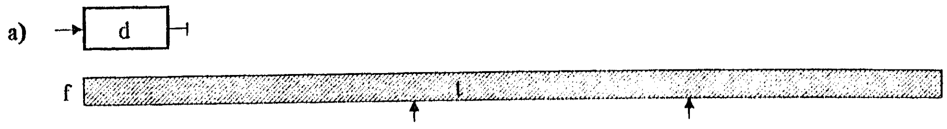
Figure 3 shows the *dynamic* data structure that we have just discussed. The two ring chains vary in size and ordering structure during the process. The next illustration which concludes our collection of examples goes even a step further. *Text editing* has become an important application in non-numerical computing. A logically connected sequence of data elements is called a *file*. So, a *text* is simply a file of characters. A text-editor is a program that shows a text on the display and interprets commands to edit this text.

Let us restrict to the commands "insert a piece of text" and "delete a piece of text". The source of the text to be inserted might be the keyboard or an existing text file. Suppose first that these commands operate *directly* on the text, i.e. on the respective sequence of characters. Then, the previous dilemma stemming from the discrepancy between the logical sequence and the sequence of entering data recurs. Indeed, the insertion or deletion of a text piece necessitates in general moving a possibly large part of the original text.

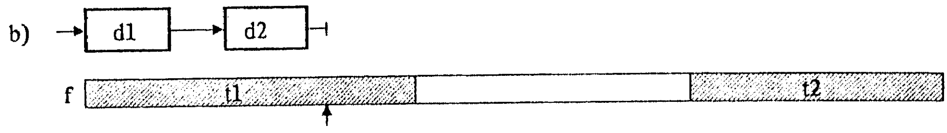
The concept of chaining turns again out to be an appropriate solution. Our data structure will reflect the interpretation of the text to be edited as a sequence of *pieces*. A piece is a sequence of logically successive characters which are stored on the same text file. We assign to each piece its *descriptor*. The descriptor indicates the respective text file, the starting position within the file and the length (number of characters). As the text to be edited will be represented as a chain of piece descriptors, we also include a pointer to the next descriptor.

Thus, our data base is an array d[1], d[2], ..., d[n] of records d[i] with components d[i].file, d[i].pos, d[i].len and d[i].next. Initially, the text to be edited consists of a single piece. Figure 4 shows the development of the chain during the editing process. Note that, in contrast to our previous example (apart from removing disqualified participants), the chain here can grow *and shrink* depending on the operations.

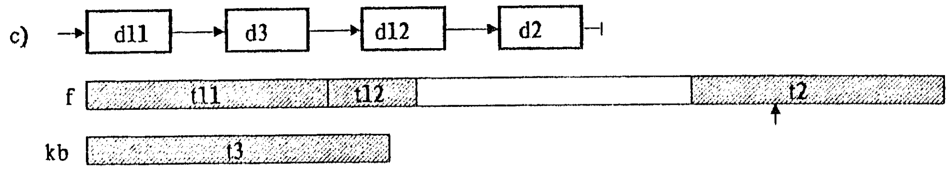
Hence, we will introduce a *pool* of currently free descriptor records. Most simply, this pool is also organized as a chain. The variable *free* might point to the head of this chain. The chain of free records is initialized as follows:



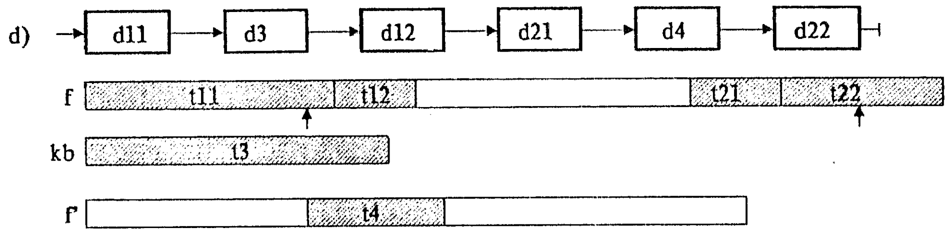
delete text part between marks



Insert text from keyboard kb at marked position



copy text piece from a second text file f' to marked position



delete text part between marks

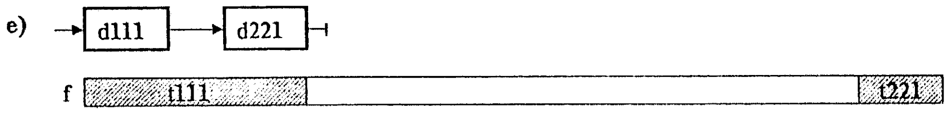


Figure 4. Sample editing sequence on text file f
and development of the descriptor chain

```
free := 1; i := 1;
DO i#n -> d[i].next := i+1 OD;
next[n] := 0
```

The operations get(i) and return(i) to get a descriptor (with index i) from and return a descriptor (with index i) to the pool are inverse to each other:

```
get(i):     IF free#0 -> i := free; free := d[i].next FI
return(i):  d[i].next := free; free := i
```

As a final example let us program out the algorithm to delete a part of the edited text. Let this part be specified by its beginning position *A* and its length *L*. Assume further that A>0 and A+L<position of last character in the text. This is practically guaranteed, if the first and the last character of the text are declared as sentinels (start of text and end of text). Let the variable *first* point to the first piece of the text.

```
B := A+L; sum := 0; a := first; {sum<A}
DO sum+d[a].len < A -> sum := sum+d[a].len; a := d[a].next {sum<A} OD;
{sum<A<=sum+d[a].len}
get(b); d[b].next := d[a].next; d[a].next := b;
d[b].len := sum+d[a].len-A; d[a].len := A-sum {>0};
d[b].file := d[a].file; d[b].pos := d[a].pos+d[a].len;
{sum<=B and b=d[a].next}
DO sum+d[b].len <= B -> sum := sum+d[b].len;
   d[a].next := d[b].next; return(b); b := d[a].next {sum<=B and b=d[a].next}
OD;
{sum<=B<sum+d[b].len and b=d[a].next}
d[b].len := sum+d[b].len-B {>0};
d[b].pos := d[b].pos+B-sum;
```

Notice that this algorithm creates no pieces of length 0. Also, the first piece p[a] which is involved in the delete operation is explicitly split up (between the two repetition statements) into two pieces p[a] and p[b]. Hence, the algorithm is not optimally efficient, if the deleted part is not contained in one piece.

Summary

We have seen that *computers* are universal devices to process information of very different kind. Their universality is founded on the concept of software or *programs*. The development of correct programs is a highly mathematical activity, even if the application is non-numeric. Programming-methods and -techniques have been developed which are of a very general nature. Programming languages facilitate the abstract formulation of a dynamic process as a static text. Modern representatives even support the programmer in the design of a possibly large program system.

Although *programming* is a central discipline of informatics, it is not alone. The design of *data banks* and of *hardware architectures* for computer systems are further cornerstones. Both of these branches have recently gained in significance: the former due to the opening up of new applications (e.g. data bases of geometric objects) and the latter due to the terrific progress of the electronic technology (VLSI).

It is a noteworthy fact that computers are not only objects but also tools in all of these branches. So, things have come full circle.

References

[1] K.Jensen and N.Wirth, Pascal User Manual and Report, Springer Verlag, 1975
[2] N. Wirth, Programming in Modula-2, Springer Verlag, 1982
[3] The Ada Programming Language, ANSI/MIL-STD-1815A, Amer. Nat. Std. Inst., 1983
[4] E.W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976
[5] D. Gries, The Science of Programming, Springer Verlag, 1981
[6] C.A.R. Hoare, Quicksort, Comp. Journal 5, No. 1, 1962, 10-15