

# **ETH**

**Eidgenössische Technische Hochschule  
Zürich**

**Institut für Informatik**

**Niklaus Wirth**

**A COLLECTION  
OF PASCAL PROGRAMS**

**July 1979**

**33**

E T H

EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE  
ZÜRICH

INSTITUT FÜR INFORMATIK

NIKLAUS WIRTH

A COLLECTION OF PASCAL PROGRAMS

Address of the author:  
Institut für Informatik  
ETH-Zentrum  
CH-8092 Zürich

## Contents

0. Preface
1. Integer arithmetic
  - power, divide, gcd-lcm, binary gcd, i-sqrt
2. Integer arithmetic and arrays
  - palindromes, magic squares, powers of two, fractions, harmonic function, prime numbers, sieve of Eratosthenes
3. Real (floating-point) arithmetic
  - sum10000, complex multiplication, Fibonacci numbers
4. Analytic functions and iteration
  - sqrt, logarithm, reciprocal value, exp, sin, cos, arcsin, arctan, ln
5. Text processing
  - printerplot, edit, count wordlengths, crunch, hitparade
6. Recursion
  - permute, infix-postfix, Hilbert curves, Sierpinski curves
7. Sorting arrays
  - straight insertion, binary insertion, Shellsort, straight selection, heapsort, bubblesort, shakersort, quicksort, mergesort
8. Sequential sorting
  - natural merge, balanced merge, polyphase merge
9. "Problem solving", backtracking
  - eight queens, nonrepeating sequences, sum of cubes, knight's tour, stable marriages, optimal selection
10. List and tree structures, pointers
  - search and insertion in ordered list, search and insertion in reordering list, topological sorting, insertion and deletion in binary tree, insertion and deletion in balanced tree, insertion and deletion in B-tree, optimal search tree
11. Cross-reference generators
  - cross-reference generator using binary tree
  - cross-reference generator using hash table
12. Syntax analysis
  - Syntax analyser and scanner for language PL/0

## 0. Preface

This is a collection of a wide variety of Pascal programs. They range in complexity from simple examples used in introductory courses to illustrate design principles and language features to intricate examples discussed in courses on algorithms and data structures. The programs, however, are grouped according to subject matter rather than complexity. Many are taken from the literature listed below, where they are explained and analyzed in detail.

The main purpose of this booklet is to provide the teacher of programming with a condensed collection of exemplary programs and thereby to exhibit a preferred style of programming using a structured language. At the same time, the booklet may serve as a guide in inventing other, perhaps similar exercises. Lastly, it may be a helpful reference to some widely used, fundamental algorithms, formulated in detail in a widely available language.

### References

N. Wirth, *Systematic programming*. Prentice-Hall, Inc. 1973.

-- *Algorithms + data structures = programs*. Prentice-Hall, Inc. 1975.

## 1. Integer arithmetic

1. Raise integer to a positive power. Repeat reading pairs of integers, until you encounter a 0. Indicate invariant of loop.

```
PROGRAM power(input, output);
  VAR a, b: integer;

  FUNCTION power (x,n: integer): integer;
    VAR w,z,i: integer;
  BEGIN w := x; i := n; z := 1;
    WHILE i # 0 DO
      BEGIN (* z*w+i = x^n *)
        IF odd(i) THEN z := z*w;
          w := sqrt(w); i := i DIV 2
        END;
        power := z
      END (* power *) ;
    END .

  BEGIN read(a);
    WHILE a # 0 DO
      BEGIN read(b); writeln(a, b, power(a,b));
        read(a)
      END
    END .
```

2. Divide an integer by a natural number, using operations of addition, subtraction, doubling and halving only. Repeat reading pairs of integers, until you encounter a 0. For each pair, print dividend, divisor, quotient, and remainder. Indicate invariant of loop.

```
PROGRAM divide(input, output);
  VAR a,b,q,r: integer;

  PROCEDURE divide (x,y: integer; VAR z,a: integer);
    VAR q,r,w: integer;
  BEGIN r := x; w := y; q := 0;
    WHILE w <= r DO w := 2*w;
      WHILE w # y DO
        BEGIN (* x = q*w + r *) w := w DIV 2; q := 2*q;
          IF w <= r THEN
            BEGIN r := r-w; q := q+1
          END
        END ;
        z := q; a := r
      END (*divide*) ;
    END .

  BEGIN read(a);
    WHILE a # 0 DO
      BEGIN read(b); divide(a,b,q,r); writeln(a, b, q, r);
        read(a)
      END
    END .
```

3. Compute the greatest common divisor (gcd) and the lowest common multiple (lcm) of two natural numbers by using addition and subtraction only. Note that  $\text{gcd}(m,n) \cdot \text{lcm}(m,n) = m \cdot n$ . Repeat reading pairs of integers, until you encounter a 0. For each pair, print the arguments, the gcd and the lcm. Indicate the loop invariant.

```
PROGRAM gcdlcm(input, output);
  VAR a,b,c,d: integer;

  PROCEDURE gcd(x,y: integer; VAR u,v: integer);
    VAR a,b,c,d: integer;
  BEGIN a := x; c := x; b := y; d := y;
    WHILE a # b DO
      BEGIN (*gcd(a,b) = gcd(x,y) AND a*d + b*c = 2*x*y*)
        IF a > b THEN
          BEGIN a := a-b; c := c+d
            END
        ELSE
          BEGIN b := b-a; d := d+c
            END
        END ;
      u := a; v := (c+d) DIV 2
    END (*gcdmult*);

  BEGIN read(a);
    WHILE a # 0 DO
      BEGIN read(b); gcd(a,b,c,d); writeln(a, b, c, d);
        read(a)
      END
    END .
```

4. Compute the greatest common divisor (gcd) of two natural numbers. Use addition, subtraction, doubling and halving only.

```
PROGRAM binarygcd(output);
  VAR a,b: integer;

  FUNCTION gcd (x,y: integer): integer;
    VAR u,v,d, a,b,k: integer;
  BEGIN u := x; v := y; a := 0; b := 0;
    WHILE NOT odd(u) DO
      BEGIN u := u DIV 2; a := a+1
        END ;
    WHILE NOT odd(v) DO
      BEGIN v := v DIV 2; b := b+1
        END ;
    IF a<b THEN k := a ELSE k := b;
      d := u - v;
      WHILE d # 0 DO
        BEGIN REPEAT d := d DIV 2 UNTIL odd(d);
          IF d<0 THEN v := -d ELSE u := d;
            d := u - v
          END ;
        WHILE k>0 DO
          BEGIN u := 2*u; k := k-1
            END;
          gcd := u
        END ;
```

```

BEGIN read(a);
  WHILE a # 0 DO
    BEGIN read(b); writeln(a, b, gcd(a,b)); read(a)
    END
  END .

```

5. Compute the largest integer less or equal to the square root of a given integer (due to Hoare).

```

PROGRAM isqrt(input,output);
  VAR n,a2,b2,ab,t: integer;
  BEGIN read(n);
    WHILE n >= 0 DO
      BEGIN a2 := 0; ab := 0; b2 := 1; writeln(" n =", n);
        WHILE b2 <= n DO b2 := 4*b2;
          WHILE b2 # 1 DO
            BEGIN (* a2+2*ab+b2 > n, 0 <= a2 <= n, sqr(ab) = a2*b2 *)
              ab := ab DIV 2; b2 := b2 DIV 4; t := a2 + 2*ab + b2;
              IF t <= n THEN
                BEGIN a2 := t; ab := ab + b2;
                END
            END ;
            writeln(a2,ab,b2); read(n)
          END
        END .

```



## 2. Integer arithmetic and arrays

1. Find all integers between 1 and 1000 whose squares are palindromes. Examples:  
 $11^2 = 121$ ,  $22^2 = 484$ .

```
PROGRAM palindromes(output);
  VAR i,j,l,n,r,s: integer;
      p: boolean;
      d: ARRAY [1..10] OF integer;
BEGIN n := 0;
  REPEAT n := n+1; s := n*n; l := 0;
    REPEAT l := l+1; r := s DIV 10;
      d[l] := s - 10*r; s := r
    UNTIL s = 0;
    i := 1; j := l;
    REPEAT p := d[i]=d[j];
      i := i+1; j := j-1
    UNTIL (i>=j) OR NOT p;
    IF p THEN writeln(n,n*n)
  UNTIL n = 1000
END .
```

2. Compute and print magic squares of order 3, 5, 7, ...

```
PROGRAM magicssquare(output);
  CONST lim = 11;
  VAR i,j,x,nx,nsq,n: integer;
      m: ARRAY [1..lim,1..lim] OF integer;

  PROCEDURE getsquare;
  BEGIN x := 0; nsq := sqr(n);
    i := (n+1) DIV 2; j := n+1;
    REPEAT nx := x + n; j := j-1;
      x := x+1; m[i,j] := x;
      REPEAT i := i+1; IF i > n THEN i := 1;
        j := j+1; IF j > n THEN j := 1;
        x := x+1; m[i,j] := x
      UNTIL x = nx
    UNTIL x = nsq
  END (*getsquare*);

  PROCEDURE printsquare;
  BEGIN
    FOR i := 1 TO n DO
      BEGIN FOR j := 1 TO n DO write(m[i,j]: 6);
        writeln
      END ;
      writeln
    END (*printsquare*);

  BEGIN n := 3;
    REPEAT getsquare; printsquare; n := n+2
    UNTIL n > lim
  END .
```

3. Compute a table of positive and negative powers of 2. Exponents range from 1 to, say, 64. Do not truncate any digits!

```

PROGRAM powersoftwo(output);
CONST m = 31; n = 100; (* m = n*log(2) *)
VAR exp,i,j,l: integer;
    c,r,t: integer;
    d: ARRAY [0..m] OF integer; (*positive powers*)
    f: ARRAY [1..n] OF integer; (*negative powers*)
BEGIN l := 0; r := 1; d[0] := 1;
  FOR exp := 1 TO n DO
    BEGIN (* compute and print 2**exp *) c := 0;
      FOR i := 0 TO l DO
        BEGIN t := 2*d[i] + c;
          IF t >= 10 THEN
            BEGIN d[i] := t-10; c := 1
              END
            ELSE
              BEGIN d[i] := t; c := 0
                END
            END ;
          IF c > 0 THEN
            BEGIN l := l+1; d[l] := 1
              END ;
            FOR i := m DOWNTO l DO write(" ");
            FOR i := l DOWNTO 0 DO write(d[i]:1);
            write(exp:5, " ");
            (*compute and print 2**(-exp) *)
            FOR j := 1 TO exp-1 DO
              BEGIN r := 10*r + f[j];
                f[j] := r DIV 2; r := r - 2*f[j]; write(f[j]:1)
              END ;
              f[exp] := 5; writeln("5"); r := 0
            END
          END .

```

4. Compute a table of exact fractions  $1/2, 1/3, \dots, 1/64$ . If the fraction has a period, print an apostrophe in front of its first digit and truncate after its last digit.

```
PROGRAM fractions(output);
(* fractions to the base b *)
CONST b = 10; max = 64;
VAR i,l,n,q,r: integer;
    a, f: ARRAY [0..max] OF integer;
BEGIN FOR n := 2 TO max DO
  BEGIN l := 0; r := 1;
    FOR i := 0 TO n-1 DO a[i] := 0;
      REPEAT l := l+1; a[r] := l;
        r := b*r; q := r DIV n; r := r - q*n; f[l] := q;
      UNTIL a[r] # 0;
      write(n, " ", ".");
      FOR i := 1 TO a[r]-1 DO write(f[i]:1);
      IF a[r] > 1 THEN write("*****");
      FOR i := a[r] TO l DO write(f[i]:1);
      writeln
    END
  END .
```

5. Compute the harmonic function  $H(n) = 1 + 1/2 + 1/3 + \dots + 1/n$  with m digits accuracy.

```
PROGRAM harmonic(input,output);
CONST lim = 100;
VAR i,k,m,n,c,r,q,sum: integer;
    d,s: ARRAY [0..lim] OF integer;
BEGIN read(m,n);
  IF (m>0) AND (m<lim) THEN
    BEGIN d[0] := 0; s[0] := 1;
      FOR i := 1 TO m DO s[i] := 0;
        FOR k := 2 TO n DO
          BEGIN (*compute 1/k*) r := 1;
            FOR i := 1 TO m DO
              BEGIN r := 10*r; q := r DIV k; r := r-q*k; d[i] := q
                END ;
              IF (10*r DIV k) >= 5 THEN d[m] := d[m]+1; (*round*)
                write(" 0."); (*intermediate output*)
              FOR i := 1 TO m DO write(d[i]:1);
                writeln;
              (*compute s := s + 1/k*) c := 0;
              FOR i := m DOWNT0 0 DO
                BEGIN sum := s[i]+d[i]+c;
                  IF sum >= 10 THEN
                    BEGIN sum := sum-10; c := 1
                      END
                  ELSE c := 0;
                    s[i] := sum
                  END
                END ;
              write(" ", s[0]:1, ".");
              FOR i := 1 TO m DO write(s[i]:1);
                writeln
            END
          END .
```

6. Compute a table of the first  $n$  prime numbers. Print  $m$  numbers per line.

```

PROGRAM primes(output);
CONST n = 1000; n1 = 33; m = 20; (*n1 ~ sqrt(n)*)
VAR i,k,x,inc,lim,square,l: integer;
    prim: boolean;
    p,v: ARRAY [1..n1] OF integer;
BEGIN l := 0;
    x := 1; inc := 4; lim := 1; square := 9;
    FOR i := 3 TO n DO
        BEGIN (*find next prime*)
            REPEAT x := x+inc; inc := 6-inc;
                IF square <= x THEN
                    BEGIN lim := lim+1;
                        v[lim] := square; square := sqr(p[lim+1])
                    END ;
                    k := 2; prim := true;
                    WHILE prim AND (k<lim) DO
                        BEGIN k := k+1;
                            IF v[k] < x THEN v[k] := v[k] + 2*p[k];
                                prim := x # v[k]
                            END
                        UNTIL prim;
                        IF i <= n1 THEN p[i] := x;
                            write(x:6); l := l+1;
                            IF l = m THEN
                                BEGIN writeln; l := 0
                                END
                            END ;
                            writeln
                        END .

```

7. Compute a table of the first  $n$  prime numbers. Print  $m$  numbers per line. Use the method of the sieve of Eratosthenes.

```

PROGRAM primes(output);
CONST m = 100; n = 10000; m = 20; h = 58;
VAR x, inc, i,k, x1,x2, lim, square, a,b,l: integer;
    p,v: ARRAY [1..m] OF integer;
    sieve: SET OF 0..h;
BEGIN l := 0;
  x := 1; inc := 4; lim := 1; square := 9;
  x1 := 0; x2 := 0; sieve := [0..h];
  p[1] := 2; p[2] := 3;
  FOR i := 3 TO n DO
    BEGIN (*find next prime*)
      REPEAT x := x+inc; inc := 6-inc;
        IF x >= square THEN
          BEGIN lim := lim+1; a := square; b := 2*p[lim];
            WHILE a < x2 DO
              BEGIN sieve := sieve - [a-x1]; a := a+b
                END ;
              v[lim] := a; square := sqr(p[lim+1])
            END ;
            IF x >= x2 THEN
              BEGIN (*construct new sieve*)
                x1 := x2; x2 := x2+h; sieve := [0..h];
                FOR k := 3 TO lim DO
                  BEGIN a := v[k]; b := 2*p[k];
                    WHILE a < x2 DO
                      BEGIN sieve := sieve - [a-x1]; a := a+b
                        END ;
                      v[k] := a
                    END
                  END
                UNTIL x-x1 IN sieve;
                IF i <= m THEN p[i] := x;
                  write(x:6); l := l+1;
                IF l = m THEN
                  BEGIN writeln; l := 0
                    END
                END ;
                writeln
            END .

```

### 3. Real (floating point) arithmetic

1. Compute the sum  $1 - 1/2 + 1/3 - 1/4 + \dots - 1/10000$  in four different ways:

1. proceed strictly from left to right,
2. sum positive and negative terms separately,
3. proceed strictly from right to left,
4. as in 2., but from right to left.

Explain the differences in the results.

```

PROGRAM sum10000(output);
  CONST n = 10000;
  VAR i: integer; x, y, s1, s2p, s2n: real;
BEGIN i := 1;
  s1 := 0; s2p := 0; s2n := 0;
  REPEAT x := 1.0/i; y := 1.0/(i+1);
    s1 := s1 + x - y;
    s2p := s2p + x; s2n := s2n + y;
    i := i+2
  UNTIL i > n;
  write (s1, s2p-s2n);
  i := n;
  s1 := 0; s2p := 0; s2n := 0;
  REPEAT x := 1.0/(i-1); y := 1.0/i;
    s1 := s1 + x - y;
    s2p := s2p + x; s2n := s2n + y;
    i := i-2
  UNTIL i = 0;
  writeln(s1, s2p-s2n)
END .

```

2. Multiply the complex number  $z = 5/13 + 12/13i$  50 times with the complex number  $w = (0.6 + 0.8i)$ . Print intermediate products and the square of their absolute value. Note that  $|z| = |w| = 1$ .

```

PROGRAM complexmult(output);
  CONST u = 0.6; v = 0.8;
  VAR i,j: integer; x,x1,y: real;
BEGIN x := 5/13; y := 12/13;
  FOR i := 1 TO 50 DO
    BEGIN FOR j := 1 TO 10 DO
      BEGIN (* (x+iy) := (x+iy) * (u+iv) *)
        x1 := x*u - y*v; y := y*u + x*v; x := x1
      END ;
      writeln(x,y,sqr(x)+sqr(y))
    END
  END
END .

```

3. Compute the Fibonacci numbers  $F(1) \dots F(N)$  in two different ways:
1. By repeated addition according to  $F(n) = F(n-1) + F(n-2)$ ,  $F(0) = F(1) = 1$ ,
  2. Using the formula  $F(n) \sim (\text{phi}^n)/\text{sqrt}(5)$ , where  $\text{phi} = (1+\text{sqrt}(5))/2$ .
- Terminate as soon as the two results differ.

```
PROGRAM fibonacci(output);
  CONST root5 = 2.236068;
  VAR i, fib0, fib1, fib3, t: integer;
      phi, fib2: real;
BEGIN phi := (1.0+root5)/2;
  i := 0; fib0 := 1; fib1 := 0; fib2 := 1.0 / root5;
  REPEAT i := i+1;
    t := fib0+fib1; fib0 := fib1; fib1 := t;
    fib2 := fib2 * phi; fib3 := trunc(fib2 + 0.5);
    writeln(i, fib1, fib3)
  UNTIL fib1 # fib3
END .
```

## 4. Analytic functions and iteration

```

CONST eps = 1E-8;

FUNCTION sqrt (x:real): real;
  VAR a,c: real;      (* 0<x<2 *)
  BEGIN a := x; c := 1.0-x;      (* abs(c)<1 *)
  REPEAT (* a+2 = b*(1-c) => (a*(1+c/2))+2 = b*(1-c)*(1+c/2)+2 =
        b*(1-0.75*(c+2) - 0.25*(c+3)) *)
    a := a*(1.0 + 0.5*c);
      (* a+2 = b*(1-0.75*(c+2) - 0.25*(c+3)) =
        b*(1-(c+2)*(0.75 + 0.25*c) *)
    c := sqrt(c) * (0.75 + 0.25*c);
      (* a+2 = b*(1-c) *)
  UNTIL abs(c) < eps;
  sqrt := a
END ;

FUNCTION log (x: real): real;
  VAR a,b,s: real;      (* 1<=x<2 *)
  BEGIN a := x; b := 1.0; s := 0;
  REPEAT (* log(x) = s + b*log(a), b<=1, 1<=a<2 *)
    a := sqrt(a);      (* log(x) = s + b*log(sqrt(a)), 1<=a<4 *)
    b := 0.5*b;      (* log(x) = s + b*log(a) *)
    IF a >= 2.0 THEN
      BEGIN (* 2<a<4 *) s := s+b (* log(x) = s + (1-b)*log(a) *);
        a := 0.5*a
      END
    UNTIL abs(b) < eps;
  log := s
END ;

FUNCTION recip (x: real): real;
  VAR a,c: real;      (* 0<x<2 *)
  BEGIN a := 1.0; c := 1.0 - x;
  REPEAT (* a*x = 1-c, abs(c)<1 *)
    a := a*(1.0+c);      (* x*a = (1-c)*(1+c) = 1 - c+2 *)
    c := sqrt(c);      (* x*a = 1-c *)
  UNTIL abs(c) < eps;
  recip := a (* recip = 1/x *)
END ;

```



Compute analytic functions as truncated sums. Determine the recurrence relations of their terms.

```

PROGRAM recurrence(input,output);
  VAR i,n: integer; x,y,s,t: real;
BEGIN
  writeln(" exp"); n := 5;
  REPEAT read(x); y := 1.0; i := 0; t := 1.0;
    REPEAT i := i+1; t := t*x/i;
      y := y + t
    UNTIL y+t = y;
    writeln(x,y,i); n := n-1
  UNTIL n = 0;

  writeln(" sin"); n := 5;
  REPEAT read(x); y := x; i := 1; s := sqr(x); t := x;
    REPEAT i := i+2; t := -t*s/((i-1)*i);
      y := y + t
    UNTIL y+t = y;
    writeln(x,y, i DIV 2); n := n-1
  UNTIL n = 0;

  writeln(" cos"); n := 5;
  REPEAT read(x); y := 1.0; i := 0; s := sqr(x); t := 1.0;
    REPEAT i := i+2; t := -t*s/((i-1)*i);
      y := y + t
    UNTIL y+t = y;
    writeln(x,y, i DIV 2); n := n-1
  UNTIL n = 0;

  writeln(" arcsin"); n := 5;
  REPEAT read(x); y := x; i := 1; s := sqr(x); t := x;
    REPEAT i := i+2; t := t*s*sqr(i-2)/((i-1)*i);
      y := y + t
    UNTIL y+t = y;
    writeln(x,y, i DIV 2); n := n-1
  UNTIL n = 0;

  writeln(" arctan"); n := 5;
  REPEAT read(x); y := x; i := 1; s := sqr(x); t := x;
    REPEAT i := i+2; t := -t*s*(i-2)/i;
      y := y + t
    UNTIL y+t = y;
    writeln(x,y, i DIV 2); n := n-1
  UNTIL n = 0;

  writeln(" ln"); n := 5;
  REPEAT read(x); x := x-1.0; y := x; t := x; i := 1;
    REPEAT i := i+1; t := -t*x*(i-1)/i;
      y := y + t
    UNTIL y+t = y;
    writeln(x+1.0, y, i); n := n-1
  UNTIL n = 0;
END .

```

## 5. Text processing

1. Plot the function  $f(x) = \exp(-x) * \cos(2*\pi*x)$  with your line printer in the range  $x = 0 \dots 4$ . Use 32 lines for the unit coordinate.

```
PROGRAM printerplot(output);
  CONST xscale = 32;
  yscale = 50; yshift = 65;
  twopi = 6.2831833071796;
  VAR i,k,n: integer;
  x,y: real;
BEGIN n := 0; (* n = x position *)
  REPEAT x := n / xscale;
    y := exp(-x)*cos(x*twopi); k := round(y*yscale);
    i := 0; write(" "); (* i = no of chars in line *)
    IF k < 0 THEN
      BEGIN write(" ": yshift+k); write(" ");
        k := -k-1; IF k > 0 THEN write(" ":k);
        write("I")
      END ELSE
      BEGIN write(" ": yshift);
        IF k > 0 THEN
          BEGIN write("I"); k := k-1;
            IF k > 0 THEN write(" ":k)
          END ;
          write(" ")
        END ;
        writeln; n := n+1
      UNTIL n > 96
    END .
```

2. Read a text and count the number of words with length 1, 2, ..., 20, and those with length greater than 20. Words are separated by blanks or ends of lines.

```
PROGRAM wordlengths(input,output);
  VAR i,k: integer;
  ch: char;
  count: ARRAY [1..21] OF integer;
BEGIN
  FOR i := 1 TO 21 DO count[i] := 0;
  WHILE NOT eof(input) DO
    BEGIN read(ch);
      IF ("a"<=ch) AND (ch<="z") THEN
        BEGIN (*new word*) k := 0;
          REPEAT k := k+1; read(ch)
            UNTIL (ch<"a") OR ("z"<ch) ;
          IF k > 20 THEN k := 21;
          count[k] := count[k] + 1
        END
      END ;
      writeln;
      writeln(" length count");
      FOR i := 1 TO 21 DO writeln(i,count[i])
    END .
```

3. Read a text and produce a copy with flushed left and right margins. Place a fixed number of characters (say, length = 72) in each line, and distribute blanks as word separators accordingly.

```

PROGRAM edit(input,output);
  CONST length = 72;
  VAR ch: char;
      i,m,k,lim: integer;
      line: ARRAY [1..136] OF char;
      index: ARRAY [1.. 68] OF integer;

  PROCEDURE setline;
    VAR i,j,h,s: integer;
        spaces, q,l,r: integer;
  BEGIN IF m=0 THEN
    BEGIN (*word is longer than line)* m := 1; index[m] := lim
    END ;
    j := 0; write(" "); (*printer control*)
    IF m > 1 THEN
    BEGIN spaces := lim - index[m];
      q := spaces DIV (m-1); r := spaces - (m-1)*q;
      l := (m-r) DIV 2; r := l+r; (*distribute spaces*)
      i := 0;
      REPEAT i := i+1; s := index[i];
        REPEAT j := j+1; write(line[j])
        UNTIL j = s;
        FOR h := 1 TO q DO write(" ");
          IF (l<=i) AND (i<r) THEN write(" ");
        UNTIL i = m-1
      END ;
      s := index[m] - 1;
      REPEAT j := j+1; write(line[j])
      UNTIL j = s;
      j := 0; writeln;
      FOR h := index[m]+1 TO lim DO
        BEGIN j := j+1; line[j] := line[h]
        END ;
      k := j; m := 0
    END (*setline*) ;

  BEGIN lim := length+1;
    k := 0; (*k = no. OF characters IN line*)
    m := 0; (*m = no. OF complete words IN line*)
    WHILE NOT eof(input) DO
    BEGIN read(ch);
      IF ch # " " THEN
      BEGIN (*next word*)
        REPEAT k := k+1; line[k] := ch; read(ch);
          IF k = lim THEN setline
          UNTIL ch = " ";
        k := k+1; line[k] := " ";
        m := m+1; index[m] := k;
        IF k = lim THEN setline
        END
      END ;
      write(" ");
    FOR i := 1 TO k DO write(line[i]);
    writeln
  
```

END .

4. Read a text and replace any sequence of one or more blanks by a single blank.

```
PROGRAM crunch(input,output);
  CONST blank = " ";
  VAR ch: char;
BEGIN
  WHILE NOT eof(input) DO
  BEGIN read(ch); write(blank); (*printer control*)
  WHILE ch = blank DO read(ch);
  WHILE NOT eoln(input) DO
  BEGIN
    REPEAT write(ch); read(ch)
    UNTIL ch = blank;
    write(blank);
    WHILE (ch=blank) AND NOT eoln(input) DO read(ch)
  END ;
  writeln; read(ch)
  END
END .
```

5. A record company conducts a poll to evaluate its products. The most popular hits are to be broadcast in a hit parade. The polled population is divided into four categories according to sex and age (teenager  $\leq 20$ , adult  $> 20$ ). Each person is asked to list five hits, identified by their number between 1 and, say, 50. The result of the poll is presented as a file; each record lists a respondent's name, first name, sex, age, and his choices ordered according to priority. A program is to compute the following data:

1. A list of hits ordered according to popularity. Each entry consists of the hit number and the number of votes it received. Hits not mentioned are omitted.
2. Four separate lists with names and first names of all respondents who had mentioned in first place one of the three hits most popular in their category.

```

PROGRAM hitparade(poll ,output);
CONST n = 50; (* number of hits *)
TYPE sex = ( male, female);
      hitno = 1 .. n;
      query = RECORD
                name, firstname: alfa;
                s: sex;
                age: 0 .. 99;
                choice: PACKED ARRAY [1..5] OF hitno
            END ;
VAR i,k,max: integer;
    b: boolean;
    total: ARRAY [hitno] OF integer;
    count: ARRAY [sex,boolean,hitno] OF integer;
    poll: FILE OF query;

PROCEDURE findnames(x: sex; y: boolean);
    VAR i,j,k: integer;
        selection: SET OF hitno;
BEGIN selection := []; reset(poll);
    writeln(" -----");
    (* find 3 hits most frequently listed in this group *)
    FOR i := 1 TO 3 DO
        BEGIN max := 0;
            FOR j := 1 TO n DO
                IF max < count[x,y,j] THEN
                    BEGIN max := count[x,y,j]; k := j
                END ;
            selection := selection + [k]; count[x,y,k] := 0
        END ;
    (* list persons with one of these hits as first choice *)
    WHILE NOT eof(poll) DO
        BEGIN
            WITH poll↑ DO
                IF s = x THEN
                    IF (age >= 20) = y THEN
                        IF choice[1] IN selection THEN
                            writeln(" ",name," ",firstname);
                            get(poll)
                        END
                    END
                END (*findnames*) ;
    BEGIN reset(poll);
        FOR i := 1 TO n DO
            BEGIN total[i] := 0;

```

```

    count[male,true,i] := 0; count[female,true,i] := 0;
    count[male,false,i] := 0; count[female,false,i] := 0
  END ;
(* collect counts *)
WHILE NOT eof(poll) DO
BEGIN
  WITH poll† DO
  FOR i := 1 TO 5 DO
    BEGIN b := age >= 20; k := choice[i];
      count[s,b,k] := count[s,b,k] + 1
    END ;
    get(poll)
  END ;
(* compute totals *)
FOR i := 1 TO n DO
  total[i] := count[male,true,i] + count[female,true,i]
    + count[male,false,i] + count[female,false,i];
page(output);
writeln(" report on hit popularity poll");
writeln(" list of hits ordered after popularity");
writeln("      hit      frequency");
REPEAT max := 0; k := 0;
  FOR i := 1 TO n DO
    IF max < total[i] THEN
      BEGIN max := total[i]; k := i
      END ;
    IF max > 0 THEN
      BEGIN total[k] := 0; writeln(k, max)
      END ;
  UNTIL max = 0;
writeln(" namelists separate by sex and age");
writeln(" men  "); findnames(male,true);
writeln(" women"); findnames(female,true);
writeln(" boys "); findnames(male,false);
writeln(" girls"); findnames(female,false);
writeln(" end of report")
END .

```

## 6. Recursion

1. Compute all  $n!$  permutations of the integers 1 ... n.

```
PROGRAM permute(output);
  CONST n = 4;
  VAR i: integer;
      a: ARRAY [1..n] OF integer;

  PROCEDURE print;
    VAR i: integer;
  BEGIN FOR i := 1 TO n DO write(a[i]:3);
        writeln
  END (*print*);

  PROCEDURE perm(k: integer);
    VAR i,x: integer;
  BEGIN
    IF k = 1 THEN print ELSE
      BEGIN perm(k-1);
            FOR i := 1 TO k-1 DO
              BEGIN x := a[i]; a[i] := a[k]; a[k] := x;
                    perm(k-1);
                    x := a[i]; a[i] := a[k]; a[k] := x;
              END
            END
          END
    END (*perm*);

  BEGIN
    FOR i := 1 TO n DO a[i] := i;
      perm(n)
    END .
```

2. Convert expressions from infix to postfix form. Each expression is written on a separate line. Expressions have the following syntax:

```
expression = term { ("+"|"=") term }.
term       = factor { "*" factor }.
factor     = letter | "(" expression ")".
```

```
PROGRAM postfix(input,output);
VAR ch: char;
PROCEDURE expression;
  VAR op: char;
  PROCEDURE factor;
  BEGIN IF ch = "(" THEN
    BEGIN read(ch); expression; read(ch) ( * ) *
    END ELSE
    BEGIN write(ch); read(ch)
    END
  END ( * factor * ) ;
  PROCEDURE term;
  BEGIN factor;
    WHILE ch = "*" DO
      BEGIN read(ch); factor; write("*")
      END
    END ( * term * ) ;
  BEGIN term;
    WHILE (ch="+") OR (ch="-") DO
      BEGIN op := ch; read(ch); term; write(op)
      END
    END ( * expression * ) ;
BEGIN
  WHILE NOT eof(input) DO
  BEGIN write(" "); read(ch); expression; writeln; readln
  END
END .
```



3. Plot Hilbert curves of orders 1 ... n. Plot procedure produces output for the Tektronix 4010 terminal. Data are represented as 12-bit bytes: a call of procedure p12 appends a byte to the output file.

```

PROGRAM hilbert(pf,output);
CONST n = 4; h0 = 512;
VAR i,h,x,y,x0,y0: integer;
    cc, wc, buf: integer;
    pf: FILE OF integer;    (*plot file*)

PROCEDURE p12(u: integer);
BEGIN buf := buf * 4096 + u; cc := cc + 1;
  IF cc = 5 THEN
    BEGIN p1t := buf; put(pf);
      wc := wc+1; buf := 0; cc := 0;
      IF wc = 31 THEN
        BEGIN p1t := 0; put(pf); wc := 0
          END
        END
  END
END (*p12*);

PROCEDURE plot;
  VAR u,v: integer;
BEGIN u := x DIV 32; v := y DIV 32;
  p12(40b+v); p12(140b+y-32*v);
  p12(40b+u); p12(100b+x-32*u);
END (*plot*);

PROCEDURE setplot;
BEGIN p12(35b); plot
END ;

PROCEDURE startplot;
BEGIN cc := 0; wc := 0; buf := 0; rewrite(pf)
END ;

PROCEDURE endplot;
BEGIN x := 0; y := 767; setplot; p12(37b);
  REPEAT p12(0) UNTIL cc = 0
END ;

PROCEDURE b(i: integer); FORWARD;
PROCEDURE c(i: integer); FORWARD;
PROCEDURE d(i: integer); FORWARD;

PROCEDURE a(i: integer);
BEGIN IF i > 0 THEN
  BEGIN d(i-1); x := x-h; plot;
    a(i-1); y := y-h; plot;
    a(i-1); x := x+h; plot;
    b(i-1)
  END
END
END ;

PROCEDURE b;
BEGIN IF i > 0 THEN
  BEGIN c(i-1); y := y+h; plot;
    b(i-1); x := x+h; plot;

```

```
        b(i-1); y := y-h; plot;
        a(i-1)
    END
END ;

PROCEDURE c;
BEGIN IF i > 0 THEN
    BEGIN b(i-1); x := x+h; plot;
          c(i-1); y := y+h; plot;
          c(i-1); x := x-h; plot;
          d(i-1)
    END
END ;

PROCEDURE d;
BEGIN IF i > 0 THEN
    BEGIN a(i-1); y := y-h; plot;
          d(i-1); x := x-h; plot;
          d(i-1); y := y+h; plot;
          c(i-1)
    END
END ;

BEGIN startplot;
    i := 0; h := h0; x0 := h DIV 2 + h; y0 := h DIV 2;
    REPEAT (*plot hilbert curve OF order i*)
        i := i+1; h := h DIV 2;
        x0 := x0 + (h DIV 2); y0 := y0 + (h DIV 2);
        x := x0; y := y0; setplot;
        a(i)
    UNTIL i = n;
    endplot
END .
```

4. Plot Sierpinski space-filling curves using their recursive pattern. Plot routine is identical to the one used in the preceding program.

```

PROGRAM sierpinski(pf,output);
CONST n = 4; h0 = 512;
VAR i,h,x,y,x0,y0: integer;
    cc, wc, buf: integer;
    pf: FILE OF integer;    (*plot file*)

PROCEDURE p12(u: integer);
BEGIN buf := buf * 4096 + u; cc := cc + 1;
    IF cc = 5 THEN
        BEGIN pf↑ := buf; put(pf);
            wc := wc+1; buf := 0; cc := 0;
            IF wc = 31 THEN
                BEGIN pf↑ := 0; put(pf); wc := 0
                END
            END
        END
END (*p12*);

PROCEDURE plot;
VAR u,v: integer;
BEGIN u := x DIV 32; v := y DIV 32;
    p12(40b+v); p12(140b+y-32*v);
    p12(40b+u); p12(100b+x-32*u);
END (*plot*);

PROCEDURE setplot;
BEGIN p12(35b); plot
END ;

PROCEDURE startplot;
BEGIN cc := 0; wc := 0; buf := 0; rewrite(pf)
END ;

PROCEDURE endplot;
BEGIN x := 0; y := 767; setplot; p12(37b);
    REPEAT p12(0) UNTIL cc = 0
END ;

PROCEDURE b(i: integer); FORWARD;
PROCEDURE c(i: integer); FORWARD;
PROCEDURE d(i: integer); FORWARD;

PROCEDURE a(i: integer);
BEGIN IF i > 0 THEN
    BEGIN a(i-1); x := x+h; y := y-h; plot;
        b(i-1); x := x + 2*h; plot;
        d(i-1); x := x+h; y := y+h; plot;
        a(i-1)
    END
END
END ;

PROCEDURE b;
BEGIN IF i > 0 THEN
    BEGIN b(i-1); x := x-h; y := y-h; plot;
        c(i-1); y := y - 2*h; plot;
    END
END ;

```

```

        a(i-1); x := x+h; y := y-h; plot;
        b(i-1)
    END
END ;

PROCEDURE c;
BEGIN IF i > 0 THEN
    BEGIN c(i-1); x := x-h; y := y+h; plot;
          d(i-1); x := x - 2*h; plot;
          b(i-1); x := x-h; y := y-h; plot;
          c(i-1)
    END
END ;

PROCEDURE d;
BEGIN IF i > 0 THEN
    BEGIN d(i-1); x := x+h; y := y+h; plot;
          a(i-1); y := y + 2*h; plot;
          c(i-1); x := x-h; y := y+h; plot;
          d(i-1)
    END
END ;

BEGIN startplot;
    i := 0; h := h0 DIV 4; x0 := 2*h; y0 := 3*h;
    REPEAT i := i+1; x0 := x0-h;
        h := h DIV 2; y0 := y0+h;
        x := x0; y := y0; setplot;
        a(i); x := x+h; y := y-h; plot;
        b(i); x := x-h; y := y-h; plot;
        c(i); x := x-h; y := y+h; plot;
        d(i); x := x+h; y := y+h; plot;
    UNTIL i = n;
    endplot
END .

```

## 7. Sorting arrays

Reference: N.Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., 1975

```

PROGRAM sort(output);
  CONST n = 256; nn = 512;
  TYPE index = 0..nn;
       item = RECORD key: integer;
                (*other fields defined here*)
              END ;

  VAR i : index; r: integer;
      a: ARRAY [-15..nn] OF item;
      z: ARRAY [1..n] OF integer;

PROCEDURE test(t: alfa; PROCEDURE sort);
  VAR i,z: integer;
BEGIN write(" ", t);
  FOR i := 1 TO n DO a[i].key := i;
  z := clock; sort; write(clock-z);
  FOR i := 1 TO n DO a[i].key := z[i];
  z := clock; sort; write(clock-z);
  FOR i := 1 TO n DO a[i].key := n-i;
  z := clock; sort; writeln(clock-z);
END (*test*);

PROCEDURE straightinsertion;
  VAR i,j: index; x: item;
BEGIN
  FOR i := 2 TO n DO
  BEGIN x := a[i]; a[0] := x; j := i-1;
    WHILE x.key < a[j].key DO
      BEGIN a[j+1] := a[j]; j := j-1;
        END ;
      a[j+1] := x
    END
  END ;

PROCEDURE binaryinsertion;
  VAR i,j,l,r,m: index; x: item;
BEGIN
  FOR i := 2 TO n DO
  BEGIN x := a[i]; l := 1; r := i-1;
    WHILE l <= r DO
      BEGIN m := (l+r) DIV 2;
        IF x.key < a[m].key THEN r := m-1 ELSE l := m+1;
        END ;
      FOR j := i-1 DOWNTO l DO a[j+1] := a[j];
        a[l] := x;
      END
    END ;
  END ;
END ;

```

```

PROCEDURE shellsort;
  CONST t = 4;
  VAR i,j,k,s: index; x: item; m: 1..t;
      h: ARRAY [1..t] OF integer;
BEGIN h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;
  FOR m := 1 TO t DO
    BEGIN k := h[m]; s := -k; (*sentinel position*)
      FOR i := k+1 TO n DO
        BEGIN x := a[i]; j := i-k;
          IF s=0 THEN s := -k; s := s+1; a[s] := x;
            WHILE x.key < a[j].key DO
              BEGIN a[j+k] := a[j]; j := j-k
                END ;
              a[j+k] := x
            END
          END ;
        END ;
      END ;
    END ;
  END ;

PROCEDURE straightselection;
  VAR i,j,k: index; x: item;
BEGIN FOR i := 1 TO n-1 DO
  BEGIN k := i; x := a[i];
    FOR j := i+1 TO n DO
      IF a[j].key < x.key THEN
        BEGIN k := j; x := a[j]
          END ;
        a[k] := a[i]; a[i] := x;
      END
    END ;
  END ;
END ;

PROCEDURE heapsort;
  VAR l,r: index; x: item;

  PROCEDURE sift;
    LABEL 13;
    VAR i,j: index;
  BEGIN i := l; j := 2*i; x := a[i];
    WHILE j <= r DO
      BEGIN IF j < r THEN
          IF a[j].key < a[j+1].key THEN j := j+1;
            IF x.key >= a[j].key THEN GOTO 13;
              a[i] := a[j]; i := j; j := 2*i
            END ;
          13: a[i] := x
        END ;
      END ;
    END ;

BEGIN l := (n DIV 2) + 1; r := n;
  WHILE l > 1 DO
    BEGIN l := l-1; sift
      END ;
    WHILE r > 1 DO
      BEGIN x := a[l]; a[l] := a[r]; a[r] := x;
        r := r-1; sift
      END
    END (*heapsort*);
END (*heapsort*);

```

```

PROCEDURE bubblesort;
  VAR i,j: index; x: item;
BEGIN FOR i := 2 TO n DO
  BEGIN FOR j := n DOWNT0 i DO
    IF a[j-1].key > a[j].key THEN
      BEGIN x := a[j-1]; a[j-1] := a[j]; a[j] := x;
      END ;
    END
  END
END (*bubblesort*);

```

```

PROCEDURE bubblex;
  VAR j,k,l: index; x: item;
BEGIN l := 2;
  REPEAT k := n;
    FOR j := n DOWNT0 l DO
      IF a[j-1].key > a[j].key THEN
        BEGIN x := a[j-1]; a[j-1] := a[j]; a[j] := x;
        k := j;
        END ;
      l := k+1;
    UNTIL l > n
  END (*bubblex*);

```

```

PROCEDURE shakersort;
  VAR j,k,l,r: index; x:item;
BEGIN l := 2; r := n; k := n;
  REPEAT
    FOR j := r DOWNT0 l DO
      IF a[j-1].key > a[j].key THEN
        BEGIN x := a[j-1]; a[j-1] := a[j]; a[j] := x;
        k := j;
        END ;
      l := k+1;
    FOR j := l TO r DO
      IF a[j-1].key > a[j].key THEN
        BEGIN x := a[j-1]; a[j-1] := a[j]; a[j] := x;
        k := j;
        END ;
      r := k-1;
    UNTIL l > r
  END (*shakersort*);

```

```

PROCEDURE quicksort; (*recursive*)
  PROCEDURE sort(l,r; index);
    VAR i,j; index; x,w; item;
    BEGIN i := l; j := r;
      x := a[(l+r) DIV 2];
      REPEAT
        WHILE a[i].key < x.key DO i := i+1;
        WHILE x.key < a[j].key DO j := j-1;
        IF i <= j THEN
          BEGIN w := a[i]; a[i] := a[j]; a[j] := w;
            i := i+1; j := j-1
          END
        UNTIL i > j;
        IF l < j THEN sort(l,j);
        IF i < r THEN sort(i,r)
      END ;
  END sort(1,n)
END (*quicksort*);

PROCEDURE quicksort1; (*non-recursive*)
  CONST m = 12;
  VAR i,j,l,r; index;
  x,w; item;
  s: 0 .. m;
  stack: ARRAY [1..m] OF
    RECORD l,r; index END ;
  BEGIN s := 1; stack[1].l := 1; stack[1].r := n;
    REPEAT (*take top request from stack*)
      l := stack[s].l; r := stack[s].r; s := s-1;
      REPEAT (*split a[l] ... a[r]*)
        i := l; j := r; x := a[(l+r) DIV 2];
        REPEAT
          WHILE a[i].key < x.key DO i := i+1;
          WHILE x.key < a[j].key DO j := j-1;
          IF i <= j THEN
            BEGIN w := a[i]; a[i] := a[j]; a[j] := w;
              i := i+1; j := j-1
            END
          UNTIL i > j;
          IF i < r THEN
            BEGIN (*stack request to sort right partition*)
              s := s+1; stack[s].l := i; stack[s].r := r
            END ;
            r := j
          UNTIL i >= r
        UNTIL s = 0
      END (*quicksort1*);

```



```

PROCEDURE mergesort;
  VAR i,j,k,l,t: index;
      h,m,p,q,r: integer; up: boolean;
  (*note that a has indices 1..2*n*)
BEGIN up := true; p := 1;
  REPEAT h := 1; m := n;
    IF up THEN
      BEGIN i := 1; j := n; k := n+1; l := 2*n
      END ELSE
      BEGIN k := 1; l := n; i := n+1; j := 2*n
      END ;
    REPEAT (*merge a run from i and j to k*)
      (*q = length of i-run, r = length of j-run*)
      IF m >= p THEN q := p ELSE q := m; m := m-q;
      IF m >= p THEN r := p ELSE r := m; m := m-r;
      WHILE (q#0) AND (r#0) DO
        BEGIN (*merge*)
          IF a[i].key < a[j].key THEN
            BEGIN a[k] := a[i]; k := k+h; i := i+1; q := q-1;
            END ELSE
            BEGIN a[k] := a[j]; k := k+h; j := j-1; r := r-1;
            END
          END ;
        IF q = 0 THEN
          BEGIN (*copy tail of j-run*)
            WHILE r # 0 DO
              BEGIN a[k] := a[j]; k := k+h; j := j-1; r := r-1;
              END
            END ELSE
            BEGIN (*r = 0, copy tail of i-run*)
              WHILE q # 0 DO
                BEGIN a[k] := a[i]; k := k+h; i := i+1; q := q-1;
                END
              END ;
            h := -h; t := k; k := l; l := t
          UNTIL m = 0;
          up := NOT up; p := 2*p
        UNTIL p >= n;
        IF NOT up THEN
          FOR i := 1 TO n DO a[i] := a[i+n]
        END (*mergesort*);
      BEGIN i := 0; r := 54321;
      REPEAT i := i+1;
        r := (131071*r) MOD 2147483647; z[i] := r
      UNTIL i = n;
      test("str insert", straightinsert);
      test("bin insert", binaryinsert);
      test("shell sort", shellsort);
      test("str select", straightselect);
      test("heapsort ", heapsort);
      test("bubblesort", bubblesort);
      test("bubblesort", bubblex);
      test("shakersort", shakersort);
      test("quicksort ", quicksort);
      test("quicksort1", quicksort1);
      test("mergesort ", mergesort);
    END .

```

## 8. Sequential sorting

### 1. Natural merge sort with 3 files (tapes) and 2 phases.

```

PROGRAM mergesort(input,output);
TYPE item = RECORD key: integer
              (*other fields defined here*)
END ;
tape = FILE OF item;
VAR c: tape; n: buf: item;

PROCEDURE list(VAR f: tape);
VAR x: item;
BEGIN reset(f);
      WHILE NOT eof(f) DO
          BEGIN read(f,x); write(output, x.key: 4)
              END ;
          writeln
      END (*list*) ;

PROCEDURE naturalmerge;
VAR l: integer; (*no. of runs merged*)
    eor: boolean; (*end-of-run indicator*)
    a,b: tape;

PROCEDURE copy(VAR x,y: tape);
VAR buf: item;
BEGIN read(x, buf); write(y,buf);
      IF eof(x) THEN eor := true ELSE eor := buf.key > x↑.key
      END ;

PROCEDURE copyrun(VAR x,y: tape);
BEGIN (*copy one run from x to y*)
      REPEAT copy(x,y) UNTIL eor
      END ;

PROCEDURE distribute;
BEGIN (*from c to a and b*)
      REPEAT copyrun(c,a);
          IF NOT eof(c) THEN copyrun(c,b)
          UNTIL eof(c)
      END ;

PROCEDURE mergerun;
BEGIN (*from a and b to c*)
      REPEAT
          IF a↑.key < b↑.key THEN
              BEGIN copy(a,c);
                  IF eor THEN copyrun(b,c)
              END ELSE
              BEGIN copy(b,c);
                  IF eor THEN copyrun(a,c)
              END
          UNTIL eor
      END ;

PROCEDURE merge;
BEGIN (*from a and b to c*)

```

```
    REPEAT mergerun; l := l+1
    UNTIL eof(a) OR eof(b);
    WHILE NOT eof(a) DO
    BEGIN copyrun(a,c); l := l+1
    END ;
    WHILE NOT eof(b) DO
    BEGIN copyrun(b,c); l := l+1
    END ;
    list(c)
END ;

BEGIN
    REPEAT rewrite(a); rewrite(b); reset(c);
        distribute;
        reset(a); reset(b); rewrite(c);
        l := 0; merge;
    UNTIL l = 1
END ;

BEGIN (*main program. read input sequence ending with 0*)
    rewrite(c); read(buf.key);
    REPEAT write(c, buf); read(buf.key)
    UNTIL buf.key = 0;
    list(c);
    naturalmerge;
    list(c)
END .
```



```

    IF x < min THEN
    BEGIN min := x; mx := i
    END
END ;
(*ta[mx] has minimal element, move it to t[j]*)
read(f[ta[mx]], buf); eot := eof(f[ta[mx]]);
write(f[t[j]], buf);
IF eot THEN
BEGIN rewrite(f[ta[mx]]); (*eliminate tape*)
    ta[mx] := ta[k2]; ta[k2] := ta[k1];
    k1 := k1-1; k2 := k2-1
END ELSE
IF buf.key > f[ta[mx]].key THEN
BEGIN tx := ta[mx]; ta[mx] := ta[k2]; ta[k2] := tx;
    k2 := k2-1
END
UNTIL k2 = 0;
IF j < n THEN j := j+1 ELSE j := nh+1
UNTIL k1 = 0;
FOR i := 1 TO nh DO
    BEGIN tx := t[i]; t[i] := t[i+nh]; t[i+nh] := tx
    END
UNTIL i = 1;
reset(f[t[1]]); list(f[t[1]], t[1]); (*sorted output is on t[1]*)
END (*tapemergesort*);

BEGIN (*generate random file f0*)
    leng := 200; rand := 7789; rewrite(f0);
    REPEAT rand := (131071*rand) MOD 2147483647;
        buf.key := rand DIV 2147484; write(f0, buf); leng := leng - 1
    UNTIL leng = 0;
    reset(f0); list(f0,1);
    tapemergesort;
END .

```

3. Polyphase sort program. There are  $n-1$  source files for merging and a single output file. The destination of the merged data changes, when a certain number of runs has been distributed. This number is computed according to a Fibonacci distribution.

```

PROGRAM polysort(output);
CONST n = 6;          (*no. of tapes*)
TYPE item = RECORD
    key: integer
END ;
tape = FILE OF item;
tapeno = 1..n;
VAR leng, rand: integer;    (*used to generate file*)
    eof: boolean;
    buf: item;
    f0: tape; (*f0 is the input tape with random numbers*)
    f: ARRAY [1..n] OF tape;

PROCEDURE list(VAR f: tape; n: tapeno);
    VAR z: integer;
BEGIN z := 0;
    writeln(" tape", n:2);
    WHILE NOT eof(f) DO
        BEGIN read(f, buf); write(output, buf.key: 5); z := z+1;
            IF z = 25 THEN
                BEGIN writeln(output); z := 0
                    END ;
            END ;
        IF z # 0 THEN writeln(output); reset(f)
    END (*list*) ;

PROCEDURE polyphasesort;
    VAR i,j,mx,tn: tapeno;
        k, level: integer;
        a, d: ARRAY [tapeno] OF integer;
            (*a[j] = ideal number of runs on tape j*)
            (*d[j] = number of dummy runs on tape j*)
        dn, x, min, z: integer;
        last: ARRAY [tapeno] OF integer;
            (*last[j] = key of tail item on tape j*)
        t,ta: ARRAY [tapeno] OF tapeno;
            (*mappings of tape numbers*)

PROCEDURE selecttape;
    VAR i: tapeno; z: integer;
BEGIN
    IF d[j] < d[j+1] THEN j := j+1 ELSE
        BEGIN IF d[j] = 0 THEN
            BEGIN level := level + 1; z := a[1];
                FOR i := 1 TO n-1 DO
                    BEGIN d[i] := z + a[i+1] - a[i]; a[i] := z + a[i+1]
                        END
                    END ;
                z := 1
            END ;
            d[j] := d[j] - 1
        END ;
END ;

```

```

PROCEDURE copyrun;
BEGIN (*copy one run from f0 to tape j*)
  REPEAT read(f0, buf); write(f[j], buf);
  UNTIL eof(f0) OR (buf.key > f0.key);
  last[j] := buf.key
END ;

BEGIN (*distribute initial runs*)
FOR i := 1 TO n-1 DO
  BEGIN a[i] := 1; d[i] := 1; rewrite(f[i])
  END ;
level := 1; j := 1; a[n] := 0; d[n] := 0;
REPEAT selecttape; copyrun
UNTIL eof(f0) OR (j=n-1);
REPEAT selecttape;
  IF last[j] <= f0.key THEN
  BEGIN (*continue old run*)
    copyrun;
    IF eof(f0) THEN d[j] := d[j] + 1 ELSE copyrun
  END
  ELSE copyrun
UNTIL eof(f0);
FOR i := 1 TO n-1 DO reset(f[i]);

FOR i := 1 TO n DO t[i] := i;
REPEAT (*merge from t[1] ... t[n-1] to t[n]*)
  z := a[n-1]; d[n] := 0; rewrite(f[t[n]]);
  writeln(" level", level:4, " tape", t[n]:4);
  FOR i := 1 TO n DO writeln(t[i], a[i], d[i]);
  REPEAT k := 0; (*merge one run*)
    FOR i := 1 TO n-1 DO
      IF d[i] > 0 THEN d[i] := d[i]-1 ELSE
      BEGIN k := k+1; ta[k] := t[i]
      END ;
    IF k = 0 THEN d[n] := d[n] + 1 ELSE
    BEGIN (*merge one real run from t[1] ... t[k]*)
      REPEAT i := 1; mx := 1;
        min := f[ta[1]].key;
        WHILE i < k DO
          BEGIN i := i+1; x := f[ta[i]].key;
            IF x < min THEN
              BEGIN min := x; mx := i
            END
          END ;
        (*ta[mx] contains minimal element, move it to t[n]*)
        read(f[ta[mx]], buf); eot := eof(f[ta[mx]]);
        write(f[t[n]], buf);
        IF (buf.key > f[ta[mx]].key) OR eot THEN
          BEGIN (*drop this tape*)
            ta[mx] := ta[k]; k := k-1
          END
        UNTIL k = 0
      END ;
      z := z-1;
    UNTIL z = 0;
    reset(f[t[n]]); list(f[t[n]], t[n]); (*rotate tapes*)
    tn := t[n]; dn := d[n]; z := a[n-1];
    FOR i := n DOWNT0 2 DO

```

```
        BEGIN t[i] := t[i-1]; d[i] := d[i-1]; a[i] := a[i-1] - z
        END ;
    t[1] := tn; d[1] := dn; a[1] := z;
    (*sorted output is on t[1]*)
    level := level - 1
    UNTIL level = 0;
END (*polyphasesort*) ;

BEGIN (*generate random file*)
    leng := 200; rand := 7789;
    REPEAT rand := (131071*rand) MOD 2147483647;
        buf.key := rand DIV 2147484; write(f0, buf); leng := leng - 1
    UNTIL leng = 0;
    reset(f0); list(f0,1);
    polyphasesort;
END .
```



## 9. "Problem solving", backtracking.

1. Find all settings of 8 queens on an 8x8 chess board such that no queen checks another queen. [see also, Comm. ACM 14, 4, 221-27 (April 74)].

```

PROGRAM eightqueens(output);
VAR i : integer;
    a : ARRAY [ 1..8 ] OF boolean;
    b : ARRAY [ 2..16 ] OF boolean;
    c : ARRAY [ -7..7 ] OF boolean;
    x : ARRAY [ 1..8 ] OF integer;
    safe : boolean;

PROCEDURE print;
    VAR k : integer;
BEGIN write(" ");
    FOR k := 1 TO 8 DO write(x[k]:2);
        writeln
    END ;

PROCEDURE trycol(j : integer);
    VAR i : integer;

PROCEDURE setqueen;
BEGIN a[i] := false; b[i+j] := false; c[i-j] := false
END ;

PROCEDURE removequeen;
BEGIN a[i] := true; b[i+j] := true; c[i-j] := true
END ;

BEGIN i := 0;
    REPEAT i := i+1; safe := a[i] AND b[i+j] AND c[i-j];
        IF safe THEN
            BEGIN setqueen; x[j] := i;
                IF j < 8 THEN trycol(j+1) ELSE print;
                    removequeen
            END
        UNTIL i = 8
    END;

BEGIN FOR i := 1 TO 8 DO a[i] := true;
    FOR i := 2 TO 16 DO b[i] := true;
    FOR i := -7 TO 7 DO c[i] := true;
        trycol(1);
    END.

```

2. Find sequences of digits 0, 1, 2 and of lengths 1 ... 90, such that they contain no two adjacent subsequences that are equal.

```

PROGRAM sequence012(output);
CONST maxlength = 90;
VAR n: integer;
    good: boolean;
    s: ARRAY [1..maxlength] OF integer;

PROCEDURE printsequence;
VAR k: integer;
BEGIN write(" ");
    FOR k := 1 TO n DO write(s[k]:1);
        writeln
    END (*printsequence*);

PROCEDURE changesequence;
BEGIN IF s[n] = 3 THEN
    BEGIN n := n-1; changesequence
    END ELSE s[n] := succ(s[n])
    END (*changesequence*);

PROCEDURE try;
VAR i,l,nhalf: integer;
BEGIN IF n <= 1 THEN good := true ELSE
    BEGIN l := 0; nhalf := n DIV 2;
        REPEAT l := l+1; i := 0;
            (* compare tails of length l for equality *)
            REPEAT good := s[n-i] # s[n-l-i]; i := i+1
                UNTIL good OR (i=l)
            UNTIL NOT good OR (l>=nhalf);
        END
    END (*try*);

BEGIN n := 0;
    REPEAT n := n+1; s[n] := 1; try;
        WHILE NOT good DO
            BEGIN changesequence; try
            END ;
            printsequence
        UNTIL n = maxlength
    END .

```

3. Find the smallest positive integer that can be represented as the sum of two cubes (integers raised to the third power) in two different ways.

```

PROGRAM sumofcubes(output);
  VAR i, ih, il, min, a, b, k: integer;
      j, sum, pwr: ARRAY [1..200] OF integer;

  (* pwr[k] = power of k, sum[k] = p[k] + p[j[k]],
     j[k] = columnindex of last considered candidate in row k,
     ih = rowindex of highest considered row,
     il = rowindex of least still relevant row *)
BEGIN i := 1; il := 1; ih := 2;
      j[1] := 1; pwr[1] := 1; sum[1] := 2;
      j[2] := 1; pwr[2] := 8; sum[2] := 9;
  REPEAT
    min := sum[i]; a := i; b := j[i];
    (* now get next sum in row i *)
    IF j[i] = i THEN
      BEGIN (* there is none left *) il := il+1;
            END ELSE
      BEGIN IF j[i] = 1 THEN
            BEGIN (* the new min was from the first column, now add
                  a new row before taking the new sum from the old row *)
              ih := ih + 1; pwr[ih] := ih*ih*ih;
              j[ih] := 1; sum[ih] := pwr[ih]+1;
            END ;
            j[i] := j[i]+1; (* next candidate in row i *)
            sum[i] := pwr[i] + pwr[j[i]]
          END ;
          (* now find minimal candidate in rows il .. ih *)
          i := il; k := i+1;
          WHILE k <= ih DO
            BEGIN IF sum[k] < sum[i] THEN i := k; k := k+1
                  END
            UNTIL sum[i] = min;
          writeln(min,a,b,i,j[i])
        END .

```

4. Find a path of a knight on a chess board which covers all 64 squares.

```

PROGRAM knightstour(output);
CONST n = 5; nsq = 25;
TYPE index = 1..n;
VAR i,j: index;
    q: boolean;
    s: SET OF index;
    a,b: ARRAY [1..8] OF integer;
    h: ARRAY [index, index] OF integer;

PROCEDURE try(i: integer; x,y: index; VAR q: boolean);
VAR k,u,v: integer; q1: boolean;
BEGIN k := 0;
    REPEAT k := k+1; q1 := false;
        u := x + a[k]; v := y + b[k];
        IF (u IN s) AND (v IN s) THEN
            IF h[u,v] = 0 THEN
                BEGIN h[u,v] := i;
                    IF i < nsq THEN
                        BEGIN try(i+1,u,v,q1);
                            IF NOT q1 THEN h[u,v] := 0
                                END ELSE q1 := true
                        END
                    END
                UNTIL q1 OR (k=8);
                q := q1
            END (*try*) ;

BEGIN s := [1,2,3,4,5];
    a[1] := 2; b[1] := 1;
    a[2] := 1; b[2] := 2;
    a[3] := -1; b[3] := 2;
    a[4] := -2; b[4] := 1;
    a[5] := -2; b[5] := -1;
    a[6] := -1; b[6] := -2;
    a[7] := 1; b[7] := -2;
    a[8] := 2; b[8] := -1;
    FOR i := 1 TO n DO
        FOR j := 1 TO n DO h[i,j] := 0;
            h[1,1] := 1; try(2,1,1,q);
            IF q THEN
                FOR i := 1 TO n DO
                    BEGIN FOR j := 1 TO n DO write(h[i,j]:5);
                        writeln
                    END
                ELSE writeln(" no solution ")
            END .
END .

```

5. Find a solution to the *stable marriage problem*.  $n$  men and  $n$  women state their preferences of partners. Find  $n$  pairs such that no man would prefer to be married to another woman who would also prefer him to her partner. A set of pairs is called *stable*, if no such cases exist [see also Comm. ACM 14, 7, 486-92 (July 71)].

```

PROGRAM marriage(input,output);
CONST n = 8;
TYPE man = 1..n; woman = 1..n; rank = 1..n;
VAR m: man; w: woman; r: rank;
    wmr: ARRAY [man, rank] OF woman;
    mwr: ARRAY [woman, rank] OF man;
    rmw: ARRAY [man, woman] OF rank;
    rwm: ARRAY [woman, man] OF rank;
    x: ARRAY [man] OF woman;
    y: ARRAY [woman] OF man;
    single: ARRAY [woman] OF boolean;

PROCEDURE print;
VAR m: man; rm, rw: integer;
BEGIN rm := 0; rw := 0;
  FOR m := 1 TO n DO
    BEGIN write(x[m]:4);
      rm := rm + rmw[m,x[m]]; rw := rw + rwm[x[m],m]
    END ;
    writeln(rm:8,rw:4);
  END (*print*);

PROCEDURE try(m: man);
VAR r: rank; w: woman;

FUNCTION stable: boolean;
VAR pm: man; pw: woman;
    i, lim: rank; s: boolean;
BEGIN s := true; i := 1;
  WHILE (i<r) AND s DO
    BEGIN pw := wmr[m,i]; i := i+1;
      IF NOT single[pw] THEN s := rwm[pw,m] > rwm[pw,y[pw]]
    END ;
    i := 1; lim := rwm[w,m];
    WHILE (i<lim) AND s DO
      BEGIN pm := mwr[w,i]; i := i+1;
        IF pm < m THEN s := rwm[pm,w] > rwm[pm,x[pm]]
      END ;
    END ;
  stable := s
END (*test*);

BEGIN (*try*)
  FOR r := 1 TO n DO
    BEGIN w := wmr[m,r];
      IF single[w] THEN
        IF stable THEN
          BEGIN x[m] := w; y[w] := m; single[w] := false;
            IF m < n THEN try(succ(m)) ELSE print;
          END ;
          single[w] := true
        END
      END
    END
  END (*try*);

```

```
BEGIN writeln("1");
  FOR m := 1 TO n DO
    FOR r := 1 TO n DO
      BEGIN read(wmr[m,r]); rmw[m,wmr[m,r]] := r
      END ;
    FOR w := 1 TO n DO
      FOR r := 1 TO n DO
        BEGIN read(mwr[w,r]); rwm[w,mwr[w,r]] := r
        END ;
      FOR w := 1 TO n DO single[w] := true;
    try(1)
  END .
```

```
5 7 1 2 6 8 4 3
2 3 7 5 4 1 8 6
8 5 1 4 6 2 3 7
3 2 7 4 1 6 8 5
7 2 5 1 3 6 8 4
1 6 7 5 8 4 2 3
2 5 7 6 3 4 8 1
3 8 4 5 7 2 6 1
5 3 7 6 1 2 8 4
8 6 3 5 7 2 1 4
1 5 6 2 4 8 7 3
8 7 3 2 4 1 5 6
6 4 7 3 8 1 2 5
2 8 5 4 6 3 7 1
7 5 2 1 8 6 4 3
7 4 1 5 2 3 6 8
```

6. Find an optimal selection of objects from a given set of  $n$  objects under a given constraint. Each object is characterised by two properties  $v$  (for value) and  $w$  (for weight). The optimal selection is the one with the largest sum of values of its members. The constraint is that the sum of their weights must not surpass a given limit  $limw$ . The algorithm is called *branch and bound*.

```

PROGRAM selection(input,output);
CONST n = 10;
TYPE index = 1..n;
      object = RECORD v,w: integer END ;
VAR i: index;
    a: ARRAY [index] OF object;
    limw, totv, maxv: integer;
    w1, w2, w3: integer;
    s, opts: SET OF index;
    z: ARRAY [boolean] OF char;

PROCEDURE try(i: index; tw,av: integer);
  VAR av1: integer;
BEGIN (*try inclusion of object i*)
  IF tw + a[i].w <= limw THEN
    BEGIN s := s + [i];
          IF i < n THEN try(i+1, tw+a[i].w, av) ELSE
            IF av > maxv THEN
              BEGIN maxv := av; opts := s
            END ;
            s := s - [i]
          END ;
          (*now try without object i*) av1 := av - a[i].v;
          IF av1 > maxv THEN
            BEGIN IF i < n THEN try(i+1, tw, av1) ELSE
              BEGIN maxv := av1; opts := s
            END
          END
    END
END (*try*) ;

BEGIN totv := 0;
  FOR i := 1 TO n DO
    WITH a[i] DO
      BEGIN read(w,v); totv := totv + v
    END ;
  read(w1,w2,w3);
  z[true] := "***"; z[false] := " ";
  write(" weight ");
  FOR i := 1 TO n DO write(a[i].w:4);
  writeln; write(" value ");
  FOR i := 1 TO n DO write(a[i].v:4);
  writeln;
  REPEAT limw := w1; maxv := 0; s := []; opts := [];
    try(1,0,totv);
    write(limw);
    FOR i := 1 TO n DO write(" ", z[i IN opts]);
    writeln; w1 := w1 + w2
  UNTIL w1 > w3
END .

```

## 10. List and tree structures, pointers.

1. A procedure *search* is to locate records with a given key in an ordered list. If the key is not present, then a new record is to be inserted so that the ordering of keys is maintained. Use a sentinel at the end of the list.

```

PROGRAM list(input,output);
  TYPE ref = ↑word;
    word = RECORD key: integer;
              count: integer;
              next: ref
            END ;
  VAR k: integer; root, sentinel: ref;

  PROCEDURE search(x: integer; VAR root: ref);
    VAR w1,w2,w3: ref;
  BEGIN w2 := root; w1 := w2↑.next; sentinel↑.key := x;
    WHILE w1↑.key < x DO
      BEGIN w2 := w1; w1 := w2↑.next
        END ;
    IF (w1↑.key = x) AND (w1 # sentinel) THEN
      w1↑.count := w1↑.count + 1 ELSE
    BEGIN new(w3); (*insert w3 between w1 AND w2*)
      WITH w3↑ DO
        BEGIN key := x; count := 1; next := w1
          END ;
        w2↑.next := w3
        END
    END (*search*);

  PROCEDURE printlist(w,z: ref);
  BEGIN WHILE w # z DO
    BEGIN writeIn(w↑.key, w↑.count);
      w := w↑.next
    END
  END (*printlist*);

  BEGIN new(root); new(sentinel); root↑.next := sentinel;
    read(k);
    WHILE k # 0 DO
      BEGIN search(k, root); read(k)
        END ;
      printlist(root↑.next,sentinel)
    END .

```



2. Instead of keeping the list ordered according to keys, reorder it as follows: After each search, the accessed record is moved to the top of the list. In this case, repeated accesses to the same element will be very fast. Use a sentinel at the end of the list.

```

PROGRAM list(input,output);
  TYPE ref = ^word;
    word = RECORD key: integer;
              count: integer;
              next: ref
            END ;
  VAR k: integer; root, sentinel: ref;

  PROCEDURE search(x: integer; VAR root: ref);
    VAR w1,w2: ref;
  BEGIN w1 := root; sentinel^.key := x;
    IF w1 = sentinel THEN
      BEGIN (*first element*) new(root);
        WITH root^ DO
          BEGIN key := x; count := 1; next := sentinel
        END
      END ELSE
      IF w1^.key = x THEN w1^.count := w1^.count + 1 ELSE
      BEGIN (*search*)
        REPEAT w2 := w1; w1 := w2^.next
        UNTIL w1^.key = x;
        IF w1 = sentinel THEN
          BEGIN (*insert*)
            w2 := root; new(root);
            WITH root^ DO
              BEGIN key := x; count := 1; next := w2
            END
          END ELSE
          BEGIN (*found, now reorder*)
            w1^.count := w1^.count + 1;
            w2^.next := w1^.next; w1^.next := root; root := w1
          END
        END
      END
    END (*search*) ;

  PROCEDURE printlist(w,z: ref);
  BEGIN WHILE w # z DO
    BEGIN writeln(w^.key, w^.count);
      w := w^.next
    END
  END (*printlist*) ;

  BEGIN new(sentinel); root := sentinel;
    read(k);
    WHILE k # 0 DO
      BEGIN search(k, root); read(k)
      END ;
    printlist(root,sentinel)
  END .

```

3. Read a sequence of relations defining a directed, finite graph. Then establish whether or not a partial ordering is defined. If so, print the elements in a sequence showing the partial ordering. (Topological sorting).

```

PROGRAM topsort(input,output);
TYPE lref = ^leader;
  lref = ^trailer;
  leader = RECORD key: integer;
             count: integer;
             trail: lref;
             next: lref;
           END ;
  trailer = RECORD id: lref;
             next: tref
           END ;
VAR head, tail, p,q: lref;
    l: tref; z: integer;
    x,y: integer;

FUNCTION l(w: integer): lref;
  (*reference to leader with key w*)
  VAR h: lref;
  BEGIN h := head; tail.key := w; (*sentinel*)
        WHILE h.key # w DO h := h.next;
        IF h = tail THEN
          BEGIN (*no element with key w in the list*)
            new(tail); z := z+1;
            h.count := 0; h.trail := NIL; h.next := tail
          END ;
          l := h
        END (*l*) ;

BEGIN (*initialise list of leaders with a dummy*)
  new(head); tail := head; z := 0;

(*input phase*) read(x);
  WHILE x # 0 DO
    BEGIN read(y); writeln(x,y);
           p := l(x); q := l(y);
           new(l); l.id := q; l.next := p.trail;
           p.trail := l; q.count := q.count + 1;
           read(x)
        END ;

(*search for leaders with count = 0*)
  p := head; head := NIL;
  WHILE p # tail DO
    BEGIN q := p; p := p.next;
           IF q.count = 0 THEN
             BEGIN q.next := head; head := q
           END ;
        END ;

(*output phase*) q := head;
  WHILE q # NIL DO
    BEGIN writeln(q.key); z := z-1;
           t := q.trail; q := q.next;
           WHILE t # NIL DO

```

```
BEGIN p := t.t.id; p.t.count := p.t.count - 1;
  IF p.t.count = 0 THEN
    BEGIN (*insert p.t in q-list*)
      p.t.next := q; q := p
    END ;
    t := t.t.next
  END
END ;

IF z # 0 THEN writeln(" this set is not partially ordered")
END .
```

4. Insertion and deletion in a binary tree. Read a sequence of integers. A positive integer signifies that it should be inserted in an ordered binary tree as the key of a node. A negative integer signifies that a node with its absolute value as key should be searched and deleted.

```

PROGRAM tree(input,output);
TYPE ref = ↑word;
   word = RECORD key: integer;
             count: integer;
             left, right: ref;
           END ;
VAR root: ref; k: integer;

PROCEDURE printtree(w: ref; l: integer);
  VAR i: integer;
BEGIN IF w # NIL THEN
  WITH w↑ DO
    BEGIN printtree(left, l+1);
          FOR i := 1 TO l DO write("  ");
          writeln(key);
          printtree(right, l+1)
        END
  END ;

PROCEDURE search(x: integer; VAR p: ref);
BEGIN
  IF p = NIL THEN
    BEGIN (*word is not in tree; insert it*)
      new(p);
      WITH p↑ DO
        BEGIN key := x; count := 1; left := NIL; right := NIL
        END
      END ELSE
    IF x < p↑.key THEN search(x, p↑.left) ELSE
    IF x > p↑.key THEN search(x, p↑.right) ELSE
      p↑.count := p↑.count + 1
    END (*search*);

PROCEDURE delete(x: integer; VAR p: ref);
  VAR q: ref;

  PROCEDURE del(VAR r: ref);
  BEGIN IF r↑.right # NIL THEN del(r↑.right) ELSE
    BEGIN q↑.key := r↑.key; q↑.count := r↑.count;
          q := r; r := r↑.left
        END
  END ;

BEGIN (*delete*)
  IF p = NIL THEN writeln(" word is not in tree") ELSE
  IF x < p↑.key THEN delete(x, p↑.left) ELSE
  IF x > p↑.key THEN delete(x, p↑.right) ELSE
    BEGIN (*deleto p↑*) q := p;
      IF q↑.right = NIL THEN p := q↑.left ELSE
      IF q↑.left = NIL THEN p := q↑.right ELSE del(q↑.left);
      (*dispose(q*)
    END
  END (*delete*);

```

```
BEGIN root := NIL; read(k);
  WHILE k # 0 DO
    BEGIN IF k > 0 THEN
      BEGIN writeln(" insert", k); search(k,root)
        END ELSE
      BEGIN writeln(" delete",-k); delete(-k,root)
        END ;
      printtree(root,0); read(k)
    END ;
  END .
```

5. Insertion and deletion in a AVL-balanced tree. In the previous program, the binary tree may grow in all sorts of shapes -- if the inserted keys are ordered upon arrival, the "tree" even degenerates into a linear list. In the following program, a balance is maintained, such that at each node the heights of its two subtrees differ by at most 1.

```

PROGRAM baltree(input,output);
TYPE ref = ^word;
   word = RECORD key: integer;
             count: integer;
             left, right: ref;
             bal: -1..+1
           END ;
VAR root: ref; h: boolean; k: integer;

PROCEDURE printtree(w: ref; l: integer);
  VAR i: integer;
BEGIN IF w # NIL THEN
  WITH w↑ DO
  BEGIN printtree(left, l+1);
        FOR i := 1 TO l DO write("  ");
        writeln(key:5, bal:3);
        printtree(right, l+1)
  END
END ;

PROCEDURE search(x: integer; VAR p: ref; VAR h: boolean);
  VAR p1,p2: ref; (*h = false*)
BEGIN
  IF p = NIL THEN
  BEGIN (*word is not in tree; insert it*)
    new(p); h := true;
    WITH p↑ DO
      BEGIN key := x; count := 1;
            left := NIL; right := NIL; bal := 0
      END
  END ELSE
  IF x < p↑.key THEN
  BEGIN search(x, p↑.left, h);
    IF h THEN (*left branch has grown higher*)
      CASE p↑.bal OF
      1: BEGIN p↑.bal := 0; h := false
          END ;
      0: p↑.bal := -1;
      -1: BEGIN (*rebalance*) p1 := p↑.left;
            IF p1↑.bal = -1 THEN
              BEGIN (*single LL rotation*)
                p↑.left := p1↑.right; p1↑.right := p;
                p↑.bal := 0; p := p1
            END ELSE
              BEGIN (*double LR rotation*) p2 := p1↑.right;
                p1↑.right := p2↑.left; p2↑.left := p1;
                p↑.left := p2↑.right; p2↑.right := p;
                IF p2↑.bal = -1 THEN p↑.bal := +1 ELSE p↑.bal := 0;
                IF p2↑.bal = +1 THEN p1↑.bal := -1 ELSE p1↑.bal := 0;
                p := p2
            END ;
            p↑.bal := 0; h := false
          END
  END
END

```

```

    END
  END ELSE
  IF x > p↑.key THEN
  BEGIN search(x, p↑.right, h);
    IF h THEN (*right branch has grown higher*)
    CASE p↑.bal OF
    -1: BEGIN p↑.bal := 0; h := false
        END ;
    0: p↑.bal := +1;
    1: BEGIN (*rebalance*) p1 := p↑.right;
        IF p1↑.bal = +1 THEN
        BEGIN (*single RR rotation*)
          p↑.right := p1↑.left; p1↑.left := p1;
          p↑.bal := 0; p := p1
        END ELSE
        BEGIN (*double RL rotation*) p2 := p1↑.left;
          p1↑.left := p2↑.right; p2↑.right := p1;
          p↑.right := p2↑.left; p2↑.left := p;
          IF p2↑.bal = +1 THEN p↑.bal := -1 ELSE p↑.bal := 0;
          IF p2↑.bal = -1 THEN p1↑.bal := +1 ELSE p1↑.bal := 0;
          p := p2
        END ;
        p↑.bal := 0; h := false
      END
    END
  END
  END
  ELSE
  BEGIN p↑.count := p↑.count + 1; h := false
  END
  END (*search*);

PROCEDURE delete(x: integer; VAR p: ref; VAR h: boolean);
  VAR q: ref; (*h = false*)

PROCEDURE balance1(VAR p: ref; VAR h: boolean);
  VAR p1,p2: ref; b1,b2: -1..+1;
  BEGIN (*h = true, left branch has become less high*)
  CASE p↑.bal OF
  -1: p↑.bal := 0;
  0: BEGIN p↑.bal := +1; h := false
      END ;
  1: BEGIN (*rebalance*) p1 := p↑.right; b1 := p1↑.bal;
      IF b1 >= 0 THEN
      BEGIN (*single RR rotation*)
        p↑.right := p1↑.left; p1↑.left := p;
        IF b1 = 0 THEN
        BEGIN p↑.bal := +1; p1↑.bal := -1; h := false
        END ELSE
        BEGIN p↑.bal := 0; p1↑.bal := 0
        END ;
        p := p1
      END ELSE
      BEGIN (*double RL rotation*)
        p2 := p1↑.left; b2 := p2↑.bal;
        p1↑.left := p2↑.right; p2↑.right := p1;
        p↑.right := p2↑.left; p2↑.left := p;
        IF b2 = +1 THEN p↑.bal := -1 ELSE p↑.bal := 0;
        IF b2 = -1 THEN p1↑.bal := +1 ELSE p1↑.bal := 0;

```

```

        p := p2; p2↑.bal := 0
    END
END
END
END (*balance1*);

PROCEDURE balance2(VAR p: ref; VAR h: boolean);
    VAR p1,p2: ref; b1,b2: -1..+1;
BEGIN (*h = true, right branch has become less high*)
    CASE p↑.bal OF
        1: p↑.bal := 0;
        0: BEGIN p↑.bal := -1; h := false
            END ;
        -1: BEGIN (*rebalance*) p1 := p↑.left; b1 := p1↑.bal;
            IF b1 <= 0 THEN
                BEGIN (*single LL rotation*)
                    p↑.left := p1↑.right; p1↑.right := p;
                    IF b1 = 0 THEN
                        BEGIN p↑.bal := -1; p1↑.bal := +1; h := false
                            END ELSE
                        BEGIN p↑.bal := 0; p1↑.bal := 0
                            END ;
                    p := p1
                END ELSE
                BEGIN (*double LR rotation*)
                    p2 := p1↑.right; b2 := p2↑.bal;
                    p1↑.right := p2↑.left; p2↑.left := p1;
                    p↑.left := p2↑.right; p2↑.right := p;
                    IF b2 = -1 THEN p↑.bal := +1 ELSE p↑.bal := 0;
                    IF b2 = +1 THEN p1↑.bal := -1 ELSE p1↑.bal := 0;
                    p := p2; p2↑.bal := 0
                END
            END
        END
    END
END
END (*balance2*);

PROCEDURE del(VAR r: ref; VAR h: boolean);
BEGIN (*h = false*)
    IF r↑.right ≠ NIL THEN
        BEGIN del(r↑.right,h); IF h THEN balance2(r,h)
            END ELSE
        BEGIN q↑.key := r↑.key; q↑.count := r↑.count;
            r := r↑.left; h := true
        END
    END
END ;

BEGIN (*delete*)
    IF p = NIL THEN
        BEGIN writeln(" key is not in tree"); h := false
            END ELSE
        IF x < p↑.key THEN
            BEGIN delete(x,p↑.left,h); IF h THEN balance1(p,h)
                END ELSE
            IF x > p↑.key THEN
                BEGIN delete(x,p↑.right,h); IF h THEN balance2(p,h)
                    END ELSE
                BEGIN (*delete p*) q := p;
                    IF q↑.right = NIL THEN

```



```
        BEGIN p := q↑.left; h := true
        END ELSE
    IF q↑.left = NIL THEN
        BEGIN p := q↑.right; h := true
        END ELSE
    BEGIN del(q↑.left,h);
        IF h THEN balance1(p,h)
        END ;
        (*dispose(q)*)
    END
END (*delete*) ;

BEGIN read(k); root := NIL;
    WHILE k # 0 DO
    BEGIN IF k >= 0 THEN
        BEGIN writeln(" insert", k); search( k,root,h)
        END ELSE
        BEGIN writeln(" delete",-k); delete(-k,root,h)
        END ;
        printtree(root,0); read(k)
    END ;
END .
```

6. Insert and delete elements in a B-tree of page size  $2n$ . Read a sequence of keys; positive values denote insertion, negative ones deletion. Print the resulting B-tree after each operation.

```

PROGRAM Btree(input,output);
CONST n = 2; nn = 4; (*page size*)
TYPE ref = ↑page;
      item = RECORD key: integer;
                p: ref;
                count: integer;
            END ;
page = RECORD m: 0..nn; (*no. of items*)
        p0: ref;
        e : ARRAY [1..nn] OF item;
    END ;
VAR root, q: ref; x: integer;
    h: boolean; u: item;

PROCEDURE printtree(p: ref; l: integer);
    VAR i: integer;
BEGIN IF p ≠ NIL THEN
    WITH p↑ DO
        BEGIN FOR i := 1 TO l DO write(" ");
            FOR i := 1 TO m DO write(e[i].key: 4);
                writeln;
                printtree(p0,l+1);
            FOR i := 1 TO m DO printtree(e[i].p, l+1)
        END
    END ;
END ;

PROCEDURE search(x: integer; a:ref;
                VAR h: boolean; VAR v: item);
(*search key x on B-tree with root a; if found, increment counter. Otherwise
insert an item with key x and count 1 in tree. If an item emerges to be passed
to a lower level, then assign it to v; h := "tree a has become higher"*)

    VAR k,l,r: integer; q: ref; u: item;

PROCEDURE insert;
    VAR i: integer; b: ref;
BEGIN (*insert u to the right of a↑.e[r]↑*)
    WITH a↑ DO
        BEGIN IF m < nn THEN
            BEGIN m := m+1; h := false;
                FOR i := m DOWNTO r+2 DO e[i] := e[i-1];
                    e[r+1] := u
            END ELSE
            BEGIN (*page a↑ is full; split it and assign the emerging
                item to v*) new(b);
                IF r <= n THEN
                    BEGIN IF r = n THEN v := u ELSE
                        BEGIN v := e[n];
                            FOR i := n DOWNTO r+2 DO e[i] := e[i-1];
                                e[r+1] := u
                            END ;
                            FOR i := 1 TO n DO b↑.e[i] := a↑.e[i+n];
                                END ELSE

```

```

    BEGIN (*insert u in right page*) r := r-n; v := e[n+1];
      FOR i := 1 TO r-1 DO bt.e[i] := at.e[i+n+1];
        bt.e[r] := u;
        FOR i := r+1 TO n DO bt.e[i] := at.e[i+n]
      END ;
      m := n; bt.m := n; bt.p0 := v.p; v.p := b;
    END
  END (*WITH*)
END (*insert*);

BEGIN (*search key x on page at; h = false*)
  IF a = NIL THEN
    BEGIN (*item with key x is not in tree*) h := true;
      WITH v DO
        BEGIN key := x; count := 1; p := NIL
        END
      END ELSE
      WITH at DO
        BEGIN l := 1; r := m; (*binary array search*)
          REPEAT k := (l+r) DIV 2;
            IF x <= e[k].key THEN r := k-1;
              IF x >= e[k].key THEN l := k+1;
            UNTIL r < l;
            IF l-r > 1 THEN
              BEGIN (*found*) e[k].count := e[k].count + 1; h := false
            END ELSE
              BEGIN (*item is not on this page*)
                IF r = 0 THEN q := p0 ELSE q := e[r].p;
                search(x,q,h,u); IF h THEN insert
              END
            END
          END (*search*);

PROCEDURE delete(x: integer; a: ref; VAR h: boolean);
(*search and delete key x in B-tree a; if a page underflow is necessary,
balance with adjacent page if possible, otherwise merge; h := "page a is
undersize"*)

  VAR i,k,l,r: integer; q: ref;

PROCEDURE underflow(c,a: ref; s: integer; VAR h: boolean);
(*a = underflow page, c = ancestor page*)
  VAR b: ref; i,k,mb,mc: integer;
  BEGIN mc := ct.m; (*h = true, at.m = n-1*)
    IF s < mc THEN
      BEGIN (*b := page to the right of a*) s := s+1;
        b := ct.e[s].p; mb := bt.m; k := (mb-n+1) DIV 2;
        (*k = no. of items available on adjacent page b*)
        at.e[n] := ct.e[s]; at.e[n].p := bt.p0;
        IF k > 0 THEN
          BEGIN (*move k items from b to a*)
            FOR i := 1 TO k-1 DO at.e[i+n] := bt.e[i];
              ct.e[s] := bt.e[k]; ct.e[s].p := b;
              bt.p0 := bt.e[k].p; mb := mb-k;
              FOR i := 1 TO mb DO bt.e[i] := bt.e[i+k];
                bt.m := mb; at.m := n-l+k; h := false
            END ELSE
              BEGIN (*merge pages a and b*)

```

```

FOR i := 1 TO n DO at.e[i+n] := bt.e[i];
FOR i := s TO mc-1 DO ct.e[i] := ct.e[i+1];
at.m := nn; ct.m := mc-1; (*dispose(b)*)
END
END ELSE
BEGIN (*b := page to the left of a*)
IF s = 1 THEN b := ct.p0 ELSE b := ct.e[s-1].p;
mb := bt.m + 1; k := (mb-n) DIV 2;
IF k > 0 THEN
BEGIN (*move k items from page b to a*)
FOR i := n-1 DOWNT0 1 DO at.e[i+k] := at.e[i];
at.e[k] := ct.e[s]; at.e[k].p := at.p0; mb := mb-k;
FOR i := k-1 DOWNT0 1 DO at.e[i] := bt.e[i+mb];
at.p0 := bt.e[mb].p;
ct.e[s] := bt.e[mb]; ct.e[s].p := a;
bt.m := mb-1; at.m := n-1+k; h := false
END ELSE
BEGIN (*merge pages a and b*)
bt.e[mb] := ct.e[s]; bt.e[mb].p := at.p0;
FOR i := 1 TO n-1 DO bt.e[i+mb] := at.e[i];
bt.m := nn; ct.m := mc-1; (*dispose(a)*)
END
END
END (*underflow*);

PROCEDURE del(p: ref; VAR h: boolean);
VAR q: ref; (*global a,k*)
BEGIN
WITH pt DO
BEGIN q := e[m].p;
IF q # NIL THEN
BEGIN del(q,h); IF h THEN underflow(p,q,m,h)
END ELSE
BEGIN pt.e[m].p := at.e[k].p; at.e[k] := pt.e[m];
m := m-1; h := m<n
END
END
END (*del*);

BEGIN (*delete*)
IF a = NIL THEN
BEGIN writeln(" key is not in tree"); h := false
END ELSE
WITH at DO
BEGIN l := 1; r := m; (*binary array search*)
REPEAT k := (l+r) DIV 2;
IF x <= e[k].key THEN r := k-1;
IF x >= e[k].key THEN l := k+1;
UNTIL l > r;
IF r=0 THEN q := p0 ELSE q := e[r].p;
IF l-r > 1 THEN
BEGIN (*found, now delete e[k]*)
IF q = NIL THEN
BEGIN (*a is a terminal page*) m := m-1; h := m<n;
FOR i := k TO m DO e[i] := e[i+1];
END ELSE
BEGIN del(q,h); IF h THEN underflow(a,q,r,h)
END
END

```

```

        END ELSE
        BEGIN delete(x,q,h); IF h THEN underflow(a,q,r,h)
        END
    END
END (*delete*);

BEGIN root := NIL; read(x);
    WHILE x # 0 DO
        BEGIN writeln(" search key", x);
            search(x,root,h,u);
            IF h THEN
                BEGIN (*insert new base page*) q := root; new(root);
                    WITH root↑ DO
                        BEGIN m := 1; p0 := q; e[1] := u
                        END
                    END ;
                printtree(root,1); read(x)
            END ;
            read(x);
            WHILE x # 0 DO
                BEGIN writeln(" delete key", x);
                    delete(x,root,h);
                    IF h THEN
                        BEGIN (*base page size was reduced*)
                            IF root↑.m = 0 THEN
                                BEGIN q := root; root := q↑.p0; (*dispose(q)*)
                                END ;
                            END ;
                        END ;
                        printtree(root,1); read(x)
                    END
                END
            END .

```

7. Find the optimally structured binary search tree for  $n$  keys. Known are the search frequencies of the keys,  $b[i]$  for  $key[i]$ , and the frequencies of searches with arguments that are not keys (represented in the tree).  $a[i]$  is the frequency of an argument lying between  $key[i-1]$  and  $key[i]$ . Use Knuth's algorithm, *Acta Informatica* 1, 1, 14-25 (1971). The following example uses Pascal keywords as keys.

```

PROGRAM optimaltree(input,output);
CONST n = 31; (*no. of keys*)
      kln = 10; (*max keylength*)
TYPE index = 0..n;
      alfa = PACKED ARRAY [1..kln] OF char;
VAR ch: char;
      k1, k2: integer;
      id: alfa; (*identifier or key*)
      buf: ARRAY [1..kln] OF char; (*character buffer*)
      key: ARRAY [1..n] OF alfa;
      i,j,k: integer;
      a: ARRAY [1..n] OF integer;
      b: ARRAY [index] OF integer;
      p,w: ARRAY [index,index] OF integer;
      r: ARRAY [index,index] OF index;
      suma, sumb: integer;

FUNCTION baltree(i,j: index): integer;
  VAR k: integer;
BEGIN k := (i+j+1) DIV 2; r[i,j] := k;
      IF i >= j THEN baltree := b[k] ELSE
        baltree := baltree(i,k-1) + baltree(k,j) + w[i,j]
      END (*baltree*);

PROCEDURE opttree;
  VAR x, min: integer;
      i,j,k,h,m: index;
BEGIN (*argument: w, result: p,r*)
  FOR i := 0 TO n DO p[i,i] := w[i,i]; (*width of tree h = 0*)
  FOR i := 0 TO n-1 DO (*width of tree h = 1*)
    BEGIN j := i+1;
      p[i,j] := p[i,i] + p[j,i]; r[i,j] := j
    END ;
    FOR h := 2 TO n DO (* h = width of considered tree *)
      FOR i := 0 TO n-h DO (* i = left index of considered tree *)
        BEGIN j := i+h; (* j = right index of considered tree *)
          m := r[i,j-1]; min := p[i,m-1] + p[m,j];
          FOR k := m+1 TO r[i+1,j] DO
            BEGIN x := p[i,k-1] + p[k,j];
              IF x < min THEN
                BEGIN m := k; min := x
              END
            END ;
          p[i,j] := min + w[i,j]; r[i,j] := m
        END ;
      END (*opttree*);

PROCEDURE printtree;
CONST lw = 120; (*line width of printer*)
TYPE ref = 1node;
      lineposition = 0..lw;
      node = RECORD key: alfa;

```

```

                pos: lineposition;
                left, right, link: ref;
        END ;
VAR root, current, next: ref;
    q,q1,q2: ref;
    i: integer;
    k: integer;
    u, u1, u2, u3, u4: lineposition;

FUNCTION tree(i,j: index): ref;
    VAR p: ref;
    BEGIN IF i = j THEN p := NIL ELSE
        BEGIN new(p);
            p+.left := tree(i, r[i,j]-1);
            p+.pos := trunc((lw-kl*n)*k/(n-1)) + (kl*n DIV 2); k := k+1;
            p+.key := key[r[i,j]];
            p+.right := tree(r[i,j], j)
        END ;
        tree := p
    END ;

BEGIN k := 0; root := tree(0,n);
    current := root; root+.link := NIL;
    next := NIL;
    WHILE current # NIL DO
    BEGIN (*proceed down; first write vertical lines*)
        FOR i := 1 TO 3 DO
        BEGIN u := 0; q := current;
            REPEAT u1 := q+.pos;
                REPEAT write(" "); u := u+1
                UNTIL u = u1;
                write(":"); u := u+1; q := q+.link
            UNTIL q = NIL;
            writeln
        END ;
        (*now print master line; descending from nodes on current list
        collect their descendants and form next list*)
        q := current; u := 0;
        REPEAT unpack(q+.key, buf, 1);
            (*center key about pos*) i := kln;
            WHILE buf[i] = " " DO i := i-1;
                u2 := q+.pos - ((i-1) DIV 2); u3 := u2+i;
                q1 := q+.left; q2 := q+.right;
                IF q1 = NIL THEN u1 := u2 ELSE
                    BEGIN u1 := q1+.pos; q1+.link := next; next := q1
                    END ;
                IF q2 = NIL THEN u4 := u3 ELSE
                    BEGIN u4 := q2+.pos+1; q2+.link := next; next := q2
                    END ;
                i := 0;
                WHILE u < u1 DO BEGIN write(" "); u := u+1 END ;
                WHILE u < u2 DO BEGIN write("-"); u := u+1 END ;
                WHILE u < u3 DO BEGIN i := i+1; write(buf[i]); u := u+1 END ;
                WHILE u < u4 DO BEGIN write("-"); u := u+1 END ;
                q := q+.link
            UNTIL q = NIL;
            writeln;
        (*now invert next list AND make it current list*)
    
```

```

current := NIL;
WHILE next # NIL DO
  BEGIN q := next; next := q↑.link;
        q↑.link := current; current := q
  END
END
END (*printtree*);

BEGIN (*initialize table of keys and counters*)
  key[ 1] := "ARRAY "; key[ 2] := "BEGIN "; key[ 3] := "CASE ";
  key[ 4] := "CONST "; key[ 5] := "DIV "; key[ 6] := "DOWNTO ";
  key[ 7] := "DO "; key[ 8] := "ELSE "; key[ 9] := "END ";
  key[10] := "FILE "; key[11] := "FOR "; key[12] := "FUNCTION ";
  key[13] := "GOTO "; key[14] := "IF "; key[15] := "IN ";
  key[16] := "LABEL "; key[17] := "MOD "; key[18] := "NIL ";
  key[19] := "OF "; key[20] := "PROCEDURE "; key[21] := "PROGRAM ";
  key[22] := "RECORD "; key[23] := "REPEAT "; key[24] := "SET ";
  key[25] := "THEN "; key[26] := "TO "; key[27] := "TYPE ";
  key[28] := "UNTIL "; key[29] := "VAR "; key[30] := "WHILE ";
  key[31] := "WITH ";
  FOR i := 1 TO n DO
    BEGIN a[i] := 0; b[i] := 0
    END ;
  b[0] := 0; k2 := kln;
  (*scan input text and determine a and b*)
  WHILE NOT eof(input) DO
    BEGIN read(ch);
      IF ch IN ["a".."z"] THEN
        BEGIN (*identifier or key*) k1 := 0;
          REPEAT IF k1 < kln THEN
            BEGIN k1 := k1+1; buf[k1] := ch
            END ;
            read(ch)
          UNTIL NOT (ch IN ["a".."z", "0".."9"]);
          IF k1 >= k2 THEN k2 := k1 ELSE
            REPEAT buf[k2] := " "; k2 := k2-1
            UNTIL k2 = k1;
            pack(buf,1,id);
            i := 1; j := n;
            REPEAT k := (i+j) DIV 2;
              IF key[k] <= id THEN i := k+1;
              IF key[k] >= id THEN j := k-1;
            UNTIL i > j;
            IF key[k] = id THEN a[k] := a[k] + 1 ELSE
              BEGIN k := (i+j) DIV 2; b[k] := b[k]+1
              END ;
            END ELSE
          END ELSE
        IF ch = "'''''" THEN
          REPEAT read(ch) UNTIL ch = "'''''" ELSE
        IF ch = "(" THEN
          REPEAT read(ch) UNTIL ch = ")"
        END ;
        writeln(" keys and frequencies of occurrence:");
        suma := 0; sumb := b[0];
        FOR i := 1 TO n DO
          BEGIN suma := suma+a[i]; sumb := sumb+b[i];
            writeln(b[i-1], a[i], " ", key[i])
          END ;

```



```

writeln(b[n]);
writeln(" -----  -----");
writeln(sumb, suma);

(*compute w from a and b*)
FOR i := 0 TO n DO
BEGIN w[i,i] := b[i];
      FOR j := i+1 TO n DO w[i,j] := w[i,j-1] + a[j] + b[j]
END ;
writeln;
write(" average path length of balanced tree =");
writeln(balltree(0,n)/w[0,n]:6:3); printtree;

opttree;
writeln;
write(" average path length of optimal tree =");
writeln(p[0,n]/w[0,n]:6:3); printtree;

(*now consider keys only, setting b = 0*)
FOR i := 0 TO n DO
BEGIN w[i,i] := 0;
      FOR j := i+1 TO n DO w[i,j] := w[i,j-1] + a[j]
END ;
opttree;
writeln;
writeln(" optimal tree considering keys only");
printtree
END .

```

## 11. Cross reference generators

1. Read a text and generate a cross reference table of all words, i.e. sequences of characters that begin with a letter and consist of letters and digits only. Blanks, ends of lines, and special characters are considered to be separators. Use a binary tree to store the words encountered.

```

PROGRAM crossref(f,output);
CONST c1 = 10; (*length of words*)
      c2 = 8; (*numbers per line*)
      c3 = 6; (*digits per number*)
      c4 = 9999; (*max line number*)
TYPE alfa = PACKED ARRAY [1..c1] OF char;
      wordref = ↑word;
      itemref = ↑item;
      word = RECORD key: alfa;
                  first, last: itemref;
                  left, right: wordref
      END ;
      item = PACKED RECORD
              lno: 0..9999;
              next: itemref
      END ;
VAR root: wordref;
    k,k1: integer;
    n: integer; (*current line number*)
    id: alfa;
    f: text;
    a: ARRAY [1..c1] OF char;

PROCEDURE search(VAR w1: wordref);
  VAR w: wordref; x: itemref;
  BEGIN w := w1;
        IF w = NIL THEN
          BEGIN new(w); new(x);
                WITH w↑ DO
                  BEGIN key := id; left := NIL; right := NIL;
                        first := x; last := x
                  END ;
                  x↑.lno := n; x↑.next := NIL; w1 := w
                END ELSE
                  IF id < w↑.key THEN search(w↑.left) ELSE
                    IF id > w↑.key THEN search(w↑.right) ELSE
                      BEGIN new(x); x↑.lno := n; x↑.next := NIL;
                            w↑.last↑.next := x; w↑.last := x
                      END
                    END
                  END (*search*) ;

PROCEDURE printtree(w: wordref);

  PROCEDURE printword(w: word);
    VAR l: integer; x: itemref;
  BEGIN write(" ", w.key);
        x := w.first; l := 0;
        REPEAT IF l = c2 THEN
          BEGIN writeln;
                l := 0; write(" ";c1+1)
          END ;

```

```

    l := l+1; write(xt.lno:c3); x := xt.next
  UNTIL x = NIL;
  writeln
END (*printword*);

BEGIN IF w # NIL THEN
  BEGIN printtree(wt.left);
        printword(wt); printtree(wt.right)
  END
END (*printtree*);

BEGIN root := NIL; n := 0; k1 := c1;
  page(output); reset(f);
  WHILE NOT eof(f) DO
  BEGIN IF n = c4 THEN n := 0;
    n := n+1; write(n:c3);    (*next line*)
    write(" ");
    WHILE NOT eoln(f) DO
    BEGIN (*scan non-empty line*)
      IF ft IN ["a".."z"] THEN
      BEGIN k := 0;
        REPEAT IF k < c1 THEN
          BEGIN k := k+1; a[k] := ft;
            END ;
          write(ft); get(f)
        UNTIL NOT (ft IN ["a".."z","0".."9"]);
        IF k >= k1 THEN k1 := k ELSE
          REPEAT a[k1] := " "; k1 := k1-1
            UNTIL k1 = k;
          pack(a,l,id); search(root)
        END ELSE
        BEGIN (*check for quote or comment*)
          IF ft = "\"" THEN
            REPEAT write(ft); get(f)
              UNTIL ft = "\"\" ELSE
            IF ft = "{" THEN
              REPEAT write(ft); get(f)
                UNTIL ft = "}";
              write(ft); get(f)
            END
          END ;
          writeln; get(f)
        END ;
    page(output); printtree(root);
  END .

```

2. Cross reference generator as above, but using a hash table instead of a binary tree to store the words encountered.

```

PROGRAM crossref(f,output);
LABEL 13;
CONST c1 = 10; (*length of words*)
      c2 = 8;  (*numbers per line*)
      c3 = 6;  (*digits per number*)
      c4 = 9999; (*max line number*)
      p = 997; (*prime number*)
      free = " ";
TYPE index = 0..p;
      itemref = titem;
      word = RECORD key: alfa;
                   first, last: itemref;
                   fol: index
            END ;
      item = PACKED RECORD
            lno: 0..9999;
            next: itemref
            END ;
VAR i, top: index;
    k,k1: integer;
    n: integer; (*current line number*)
    id: alfa;
    f: text;
    a: ARRAY [1..c1] OF char;
    letters, letdigs: SET OF char;
    t: ARRAY [0..p] OF word; (*hash table*)

PROCEDURE search;
  VAR h,d,i: index;
      x: itemref; f: boolean;
  (*global variables: t, id, top*)
  BEGIN h := ord(id) DIV 4096 MOD p;
  (*Pascal-6000 defines ord on packed character array of length 10.
  Division is needed because division operates on 48 bits only! *)
  f := false; d := 1;
  new(x); x↑.lno := n; x↑.next := NIL;
  REPEAT
    IF t[h].key = id THEN
      BEGIN (*found*) f := true;
        t[h].last↑.next := x; t[h].last := x
      END ELSE
    IF t[h].key = free THEN
      BEGIN (*new entry*) f := true;
        WITH t[h] DO
          BEGIN key := id; first := x; last := x; fol := top
          END ;
          top := h
        END ELSE
      BEGIN (*collision*) h := h+d; d := d+2;
        IF h >= p THEN h := h-p;
        IF d = p THEN
          BEGIN writeln(" table overflow"); GOTO 13
          END
        END
      END
  UNTIL f

```

```

END (*search*);

PROCEDURE printable;
  VAR i,j,m: index;

  PROCEDURE printword(w: word);
    VAR l: integer; x: itemref;
  BEGIN write(" ", w.key);
    x := w.first; l := 0;
    REPEAT IF l = c2 THEN
      BEGIN writeln;
        l := 0; write(" ":c1+1)
      END;
      l := l+1; write(x↑.lno:c3); x := x↑.next
    UNTIL x = NIL;
    writeln
  END (*printword*);

BEGIN i := top;
  WHILE i # p DO
    BEGIN (*scan linked list and find minimal key*)
      m := i; j := t[i].fol;
      WHILE j # p DO
        BEGIN IF t[j].key < t[m].key THEN m := j;
          j := t[j].fol
        END;
        printword(t[m]);
        IF m # i THEN
          BEGIN t[m].key := t[i].key;
            t[m].first := t[i].first; t[m].last := t[i].last
          END;
          i := t[i].fol
        END
      END (*printable*);

BEGIN n := 0; k1 := c1; top := p; reset(f);
  FOR i := 0 TO p DO t[i].key := free;
  letters := ["a".."z"]; letdigs := letters + ["0".."9"];
  WHILE NOT eof(f) DO
    BEGIN IF n = c4 THEN n := 0;
      n := n+1; write(n:c3); (*next line*)
      write(" ");
      WHILE NOT eoln(f) DO
        BEGIN (*scan non-empty line*)
          IF ft IN letters THEN
            BEGIN k := 0;
              REPEAT IF k < c1 THEN
                BEGIN k := k+1; a[k] := ft;
                  END;
                  write(ft); get(f)
              UNTIL NOT (ft IN letdigs);
              IF k >= k1 THEN k1 := k ELSE
                REPEAT a[k1] := " "; k1 := k1-1
                  UNTIL k1 = k;
                pack(a,l,d); search;
            END ELSE
              BEGIN (*check for quote or comment*)
                IF ft = "" THEN

```

```
        REPEAT write(f↑); get(f)
        UNTIL f↑ = "" ELSE
    IF f↑ = "{" THEN
        REPEAT write(f↑); get(f)
        UNTIL f↑ = "}" ;
        write(f↑); get(f)
    END
END ;
writeln; get(f)
END ;
```

13: page; printable  
END .

## 12. Syntax analysis

Skeleton compiler which checks the syntax of its input text according to the following grammar. Principle is top-down, recursive descent with one symbol lookahead. (see also N.Wirth, *Algorithms + Data Structures = Programs*, Ch. 5, Prentice-Hall, Inc. 1975)

```

program = block "." .
block = [ "CONST" ident "=" number {"," ident "=" number} ";";
        [ "VAR" ident {"," ident} ";";
        { "PROCEDURE" ident ";" block ";"; } statement .
statement = [ ident "!=" expression | "CALL" ident |
            "BEGIN" statement {"," statement} "END" |
            "IF" condition "THEN" statement |
            "WHILE" condition "DO" statement ] .
condition = "ODD" expression |
            expression ("="|"#"<|"<"|"<="|">"|">=") expression .
expression = ["+"|"-" ] term {("+"|"-" ) term}.
term = factor {("*"|"/" ) factor}.
factor = ident | number | "(" expression ")" .

```

```

PROGRAM PLO(input,output);
LABEL 99;

```

```

CONST norw = 11;      (*no. of reserved words*)
      txmax = 100;    (*length of identifier table*)
      nmax = 14;      (*max. no of digits in numbers*)
      al = 10;        (*length of identifiers*)
      chsetsize = 128; (*for ASCII character set*)

```

```

TYPE symbol =
(nul,ident,number,plus,minus,times,slash,oddsym,
 eql,neq,lss,leq,gtr,geq,lparen,rparen,comma,semicolon,
 period,becomes,beginsym,endsym,ifsym,thensym,
 whitesym,dosym,callsym,constsym,varsym,procsym);
alfa = PACKED ARRAY [1..al] OF char;
object = (constant,variable,prozedure);

```

```

VAR ch: char;          (*last character read*)
    sym: symbol;       (*last symbol read*)
    id: alfa;          (*last identifier read*)
    num: integer;      (*last number read*)
    cc: integer;       (*character count*)
    ll: integer;       (*line length*)
    kk: integer;
    line: ARRAY [1..81] OF char;
    a: alfa;
    word: ARRAY [1..norw] OF alfa;
    wsym: ARRAY [1..norw] OF symbol;
    ssym: ARRAY [char] OF symbol;
    table: ARRAY [0..txmax] OF
        RECORD name: alfa;
              kind: object
        END ;

```

```

PROCEDURE error(n: integer);
BEGIN writeln(" :cc, "+n); GOTO 99
END (*error*) ;

PROCEDURE getsym;
  VAR i,j,k: integer;

  PROCEDURE getch;
  BEGIN IF cc = 0 THEN
    BEGIN IF eof(input) THEN
      BEGIN write(" program incomplete"); GOTO 99
      END ;
      ll := 0; cc := 0; write(" ");
      WHILE NOT eoln(input) DO
        BEGIN ll := ll+1; read(ch); write(ch); line[ll] := ch
        END ;
        writeln; ll := ll+1; read(line[ll])
      END ;
      cc := cc+1; ch := line[cc]
    END (*getch*) ;

  BEGIN (*getsym*)
    WHILE ch = " " DO getch;
    IF ch IN ["a".."z"] THEN
      BEGIN (*identifier or reserved word*) k := 0;
        REPEAT IF k < al THEN
          BEGIN k := k+1; a[k] := ch
          END ;
          getch
        UNTIL NOT (ch IN ["a".."z","0".."9"]);
        IF k >= kk THEN kk := k ELSE
          REPEAT a[kk] := " "; kk := kk-1
          UNTIL kk = k;
        id := a; i := 1; j := norw;
        REPEAT k := (i+j) DIV 2;
          IF id <= word[k] THEN j := k-1;
          IF id >= word[k] THEN i := k+1
        UNTIL i > j;
        IF i-1 > j THEN sym := wsym[k] ELSE sym := ident
      END ELSE
        IF ch IN ["0".."9"] THEN
          BEGIN (*number*) k := 0; num := 0; sym := number;
            REPEAT num := 10*num + (ord(ch)-ord("0"));
              k := k+1; getch
            UNTIL NOT (ch IN ["0".."9"]);
            IF k > nmax THEN error(30)
          END ELSE
            IF ch = ":" THEN
              BEGIN getch;
                IF ch = "=" THEN
                  BEGIN sym := becomes; getch
                END ELSE sym := nul;
            END ELSE
            IF ch = "<" THEN
              BEGIN getch;

```



```

    IF ch = "=" THEN
    BEGIN sym := leq; getch
    END ELSE sym := lss
    END ELSE
    IF ch = ">" THEN
    BEGIN getch;
    IF ch = "=" THEN
    BEGIN sym := geq; getch
    END ELSE sym := gr
    END ELSE
    BEGIN sym := ssym[ch]; getch
    END
    END (*getsym*);

PROCEDURE block(tx: integer);

    PROCEDURE enter(k: object);
    BEGIN (*enter object into table*)
    tx := tx + 1;
    WITH table[tx] DO
    BEGIN name := id; kind := k;
    END
    END (*enter*);

    FUNCTION position(id: alfa): integer;
    VAR i: integer;
    BEGIN (*find identifier id in table*)
    table[0].name := id; i := tx;
    WHILE table[i].name # id DO i := i-1;
    position := i
    END (*position*);

    PROCEDURE constdeclaration;
    BEGIN IF sym = ident THEN
    BEGIN getsym;
    IF sym = eql THEN
    BEGIN getsym;
    IF sym = number THEN
    BEGIN enter(constant); getsym
    END
    ELSE error(2)
    END ELSE error(3)
    END ELSE error(4)
    END (*constdeclaration*);

    PROCEDURE vardeclaration;
    BEGIN IF sym = ident THEN
    BEGIN enter(variable); getsym
    END ELSE error(4)
    END (*vardeclaration*);

    PROCEDURE statement;
    VAR i: integer;

```

```

PROCEDURE expression;
  PROCEDURE term;
    PROCEDURE factor;
      VAR i: integer;
    BEGIN
      IF sym = ident THEN
        BEGIN i := position(id);
          IF i = 0 THEN error(11) ELSE
            IF table[i].kind = prozedure THEN error(21);
              getsym
            END ELSE
              IF sym = number THEN
                BEGIN getsym
                  END ELSE
                    IF sym = lparen THEN
                      BEGIN getsym; expression;
                        IF sym = rparen THEN getsym ELSE error(22)
                          END
                        ELSE error(23)
                      END (*factor*);
                    BEGIN (*term*) factor;
                      WHILE sym IN [times,slash] DO
                        BEGIN getsym; factor
                          END
                        END (*term*);
                    BEGIN (*expression*)
                      IF sym IN [plus,minus] THEN
                        BEGIN getsym; term
                          END ELSE term;
                      WHILE sym IN [plus,minus] DO
                        BEGIN getsym; term
                          END
                        END (*expression*);
                    PROCEDURE condition;
                      BEGIN
                        IF sym = oddsym THEN
                          BEGIN getsym; expression
                            END ELSE
                              BEGIN expression;
                                IF NOT (sym IN [eq,neq,iss,leq,glr,geq]) THEN
                                  error(20) ELSE
                                    BEGIN getsym; expression
                                      END
                                    END
                                  END
                                END (*condition*);
                              BEGIN (*statement*)
                                IF sym = ident THEN
                                  BEGIN i := position(id);
                                    IF i = 0 THEN error(11) ELSE
                                      IF table[i].kind # variable THEN error(12);

```

```

    getsym; IF sym = becomes THEN getsym ELSE error(13);
    expression
  END ELSE
  IF sym = callsym THEN
  BEGIN getsym;
    IF sym # ident THEN error(14) ELSE
      BEGIN i := position(id);
        IF i = 0 THEN error(11) ELSE
          IF table[i].kind # prozedure THEN error(15);
          getsym
        END
      END ELSE
    IF sym = ifsym THEN
    BEGIN getsym; condition;
      IF sym = thensym THEN getsym ELSE error(16);
      statement;
    END ELSE
    IF sym = beginsym THEN
    BEGIN getsym; statement;
      WHILE sym = semicolon DO
        BEGIN getsym; statement
        END ;
      IF sym = endsym THEN getsym ELSE error(17)
    END ELSE
    IF sym = whitesym THEN
    BEGIN getsym; condition;
      IF sym = dosym THEN getsym ELSE error(18);
      statement
    END
  END (*statement*);

```

```

BEGIN (*block*)
  IF sym = constsym THEN
  BEGIN getsym; constdeclaration;
    WHILE sym = comma DO
      BEGIN getsym; constdeclaration
      END ;
    IF sym = semicolon THEN getsym ELSE error(5)
  END ;
  IF sym = varsym THEN
  BEGIN getsym; vardeclaration;
    WHILE sym = comma DO
      BEGIN getsym; vardeclaration
      END ;
    IF sym = semicolon THEN getsym ELSE error(5)
  END ;
  WHILE sym = procsym DO
  BEGIN getsym;
    IF sym = ident THEN
      BEGIN enter(prozedure); getsym
      END
    ELSE error(4);
    IF sym = semicolon THEN getsym ELSE error(5);
    block(tx);

```

```

    IF sym = semicolon THEN getsym ELSE error(5);
  END ;
  statement
END (*block*) ;

BEGIN (*main program*)
  FOR ch := chr(0) TO chr(chsetsize-1) DO ssym[ch] := nul;
  word[ 1 ] := "BEGIN   "; word[ 2 ] := "CALL   ";
  word[ 3 ] := "CONST   "; word[ 4 ] := "DO     ";
  word[ 5 ] := "END     "; word[ 6 ] := "IF     ";
  word[ 7 ] := "ODD     "; word[ 8 ] := "PROCEDURE ";
  word[ 9 ] := "THEN    "; word[10] := "VAR    ";
  word[11] := "WHILE   ";
  wsym[ 1 ] := beginsym; wsym[ 2 ] := callsym;
  wsym[ 3 ] := constsym; wsym[ 4 ] := dosym;
  wsym[ 5 ] := endsym;   wsym[ 6 ] := ifsym;
  wsym[ 7 ] := oddsym;   wsym[ 8 ] := procsym;
  wsym[ 9 ] := thensym;  wsym[10] := varsym;
  wsym[11] := whilesym;
  ssym["+"] := plus;    ssym["-"] := minus;
  ssym["*"] := times;   ssym["/"] := slash;
  ssym["("] := lparen;  ssym[")"] := rparen;
  ssym["="] := eql;     ssym[","] := comma;
  ssym["."] := period;  ssym["#"] := neq;
  ssym["<"] := lss;     ssym[ "> " ] := gtr;
  ssym[";"] := semicolon;
  page(output);
  cc := 0; ll := 0; ch := " "; kk := at; getsym;
  block(0);
  IF sym # period THEN error(9);
99: writeln
END .

```

Berichte des Instituts für Informatik

- \* Nr. 1 N. Wirth The Programming Language PASCAL
- \* Nr. 2 N. Wirth Program development by step-wise refinement
- Nr. 3 P. Läuchli Reduktion elektrischer Netzwerke und Gauss'sche Elimination
- Nr. 4 W. Gander,  
A. Mazzario Numerische Prozeduren I
- \* Nr. 5 N. Wirth The Programming Language PASCAL (Revised Report)
- \* Nr. 6 C.A.R.Hoare,  
N. Wirth An Axiomatic Definition of the Language PASCAL
- Nr. 7 W. Gander,  
A. Mazzario Numerische Prozeduren II
- Nr. 8 E. Engeler,  
E. Wiedmer,  
E. Zachos Ein Einblick in die Theorie der Berechnungen
- \* Nr. 9 H.P. Frei Computer Aided Instruction: The Author Language and  
the System THALES
- \* Nr.10 K.V. Nori,  
U. Ammann,  
K. Jensen, The Pascal 'P' Compiler: Implementation Notes  
H.H. Nägeli, (Revised Edition)  
Ch. Jacobi
- Nr.11 G.I. Ugron,  
F.R. Lüthi Das Informations-System ELSBETH
- \* Nr.12 N. Wirth PASCAL-S: A subset and its Implementation
- \* Nr.13 U. Ammann Code Generation in a PASCAL Compiler
- Nr.14 K.Lieberherr Toward Feasible Solutions of NP-Complete Problems
- \* Nr.15 E. Engeler Structural Relations between Programs and Problems
- Nr.16 W. Bucher A contribution to solving large linear problems
- Nr.17 N. Wirth Programming languages: what to demand and how to  
access them and  
Professor Cleverbyte's visit to heaven
- \* Nr.18 N. Wirth MODULA: A language for modular multiprogramming
- \* Nr.19 N. Wirth The use of MODULA and  
Design and Implementation of MODULA
- Nr.20 E. Wiedmer Exaktes Rechnen mit reellen Zahlen
- \* Nr.21 J.Nievergelt,  
H.P. Frei,  
et al. XS-0, a Self-explanatory School Computer
- Nr.22 P. Läuchli Ein Problem der ganzzahligen Approximation
- Nr.23 K. Bucher Automatisches Zeichnen von Diagrammen
- Nr.24 E. Engeler Generalized Galois Theory and its Application to Complexity

- Nr.25 U. Ammann Error Recovery in Recursive Descent Parsers and Run-time Storage Organization
- Nr.26 E. Zachos Kombinatorische Logik und S-Terme
- Nr.27 N. Wirth MODULA-2
- Nr.28 J.Nievergelt, Sites, Modes and Trails: Telling the User of an Interactive  
J. Weydert System where he is, what he can do, and how to get to Places
- Nr.29 A.C. Shaw On the Specification of Graphic Command Languages and their Processors
- Nr.30 B.Thurnherr, Global Data Base Aspects, Consequences for the Relational  
C.A.Zehnder Model and a Conceptual Scheme Language
- Nr.31 A.C. Shaw Software Specification Languages based on regular Expressions
- Nr.32 E. Engeler Algebras and Combinators
- Nr.33 N. Wirth A Collection of PASCAL Programs