A Collection of Research Processes for Genealogy and Proofs

VOLUME SEVEN, SECTION 38

Collection of Hash Functions (Using RCN Input Data Set) (File Dump on September 29, 1991)

by

Dr. Dong-Keun Shin

File Name: RAND PASCAL

A Simulation Program (random) for Fold Shifting, Algebraic Coding, Digit Analysis, Division, Folding, Midsquare, Multiplicative, Radix, Random Hash Functionswhich reads RCN or GCN data set and produces output.

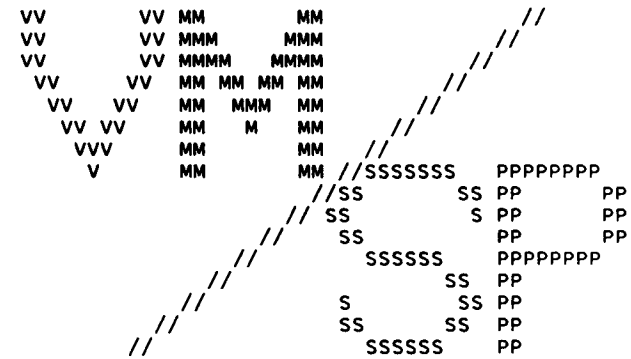
File Name: RANDOM PASCAL

A Simulation Program (random) that includes the above hash functions and Fold Four Shifted Words (RC or formerly FS) hash function which reads either RCN or GCN data set and produces output. This program specially tests RC (FS) hash methods

On June 13, 1996, Submitted to the Chair of
Department of Electrical Engineering and Computer Sciences
College of Engineering
University of California, Berkeley
Berkeley, CA 94720

```
RSC  :C      RSCSREC      USERID     ORIGIN
SHIN         GWUVM        DISTCODE   SYSTEM
ROMMAKE      PASCAL       FILENAME   FILETYPE
09/29/91     21:40:58     FILE CREATION DATE
9671         00003631     SPOOLID    COUNT
09/29/91     21:41:58     FILE PRINT DATE
A            0211         CLASS      DEVICE
P1N          OFF          FORMS      DESTINATION
```

```
333333333   6. 3888         4
333333333  888888888       44
33    33   88     88      444
33    33   88     88     4444
3333       8888888     44 44
333333     8888888     44 44
33    33   88     88  444444444
33    33   88     88  444444444
333333333  888888888       44
 3333333    8888888       44
```




```
TAG DATA: FILE (3163) ORIGIN GWUUNIV  SHIN      9/29/91 21:40:30 E.D.T.
```

```
RRRRRRRR    SSSSSSS    CCCCCCC    SSSSSSS   RRRRRRRR   EEEEEEEEE   CCCCCCC
RRRRRRRRR  SSSSSSSSS  CCCCCCCCC  SSSSSSSSS  RRRRRRRRR  EEEEEEEEE  CCCCCCCCC
RR      RR SS      SS CC      CC SS      SS RR      RR EE         CC      CC
RR      RR SS         CC         SS         RR      RR EE         CC
RRRRRRRRR  SSSSSSSS   CC         SSSSSSSS   RRRRRRRRR  EEEEEE     CC
RRRRRRRR    SSSSSSSS  CC          SSSSSSSS  RRRRRRRR   EEEEEE     CC
RR  RR             SS CC                SS RR  RR     EE         CC
RR    RR    SS      SS CC      CC SS      SS RR    RR   EE         CC      CC
RR     RR   SSSSSSSSS  CCCCCCCCC  SSSSSSSSS RR     RR  EEEEEEEEE  CCCCCCCCC
RR      RR   SSSSSSS    CCCCCCC    SSSSSSS  RR      RR EEEEEEEEE   CCCCCCC
```

```
 SSSSSSS   HH      HH   IIIIII    N       NN
SSSSSSSSS  HH      HH   IIIIII    NN      NN
SS     SS  HH      HH     II      NNN     NN
SS         HH      HH     II      NNNN    NN
SSSSSSSS   HHHHHHHHH     II      NN NN   NN
 SSSSSSSS  HHHHHHHHH     II      NN  NN  NN
       SS  HH      HH     II      NN   NNNN
SS     SS  HH      HH     II      NN    NNN
SSSSSSSSS  HH      HH   IIIIII    NN      NN
 SSSSSSS   HH      HH   IIIIII    NN       N
```

```
program random (input, output);

    t


     DATA_NAME = '1024 RANDOMLY CHOSEN NAMES';

     DEBUG_FLAG = true;


     HASH_NAME = 'FOLD SHIFTING';
   {
     HASH_NAME = 'ALGEBRAIC CODING';
     HASH_NAME = 'DIGIT ANALYSIS';
     HASH_NAME = 'DIVISION';
     HASH_NAME = 'FOLDING';
     HASH_NAME = 'MIDSQUARE';
     HASH_NAME = 'MULTIPLICATIVE';
     HASH_NAME = 'RADIX';
     HASH_NAME = 'RANDOM';
   }

     OFFSET = 5;
     MAX_NUM_KEYS = 1024;
     NUM_IDENT_CHAR = 16;
     MAX_BUCKET_ADDR_BITS = 8;
     MAX_BOOL_DIGIT = 8;
     NUM_ASCII_CHAR = 70;

     NUM_BYTES_IN_WORD = 4;
     NO_BUCKETS = 256;

     NUM_FIRST_EXOR = 8;
     NUM_SECOND_EXOR = 4;
     NUM_THIRD_EXOR = 2;

     BUCKET_SIZE = 255;

     NUM_BITS_IN_WORD = 32;
     BITS_FOR_TWO_BYTES = 16;



type
     KEY_ARRAY_TYPE = array (.1..NUM_IDENT_CHAR.) of char;

     BOOL_TYPE = array (.1..MAX_BOOL_DIGIT.) of boolean;

     BOOL_WORD_TYPE = array (.1..NUM_BITS_IN_WORD.) of boolean;

     CHAR_RECORD_TYPE = record
                          ch : char;
                          ASCII_num : integer;
                          bool_rep : BOOL_TYPE;
                        end;
```

```
     ASCII_TABLE = array (.1..NUM_ASCII_CHAR.) of
                   CHAR_RECORD_TYPE;

     BOOL_KEY_TYPE = array (.1..NUM_IDENT_CHAR.) of
                           record
                                bool_rep : BOOL_TYPE;
                           end;

     TWO_BYTES_TYPE = array (.1..BITS_FOR_TWO_BYTES.) of boolean;

     HASHED_KEY_REG_TYPE = array (.1..MAX_BUCKET_ADDR_BITS.) of boolean;

     COUNT_KEY_TYPE = array (.0..BUCKET_SIZE.) of integer;

     LINK = @KEY_RECORD;
     KEY_RECORD = record
                     Key_Arr : KEY_ARRAY_TYPE;
                     next : LINK;
                  end;

     BUCKET_POINTER_ARRAY = array (.0..BUCKET_SIZE.) of LINK;


     HASH_KINDS = (ALGEBRAIC_CODING,
                   DIGIT_ANALYSIS,
                   DIVISION,
                   FOLDING,
                   FOLD_SHIFTING,
                   MIDSQUARE,
                   MULTIPLICATIVE,
                   RADIX,
                   RANDOM);


var
     Hash_Type : HASH_KINDS;

     Key_Register : KEY_ARRAY_TYPE;

     ASCII_Arr : ASCII_TABLE;

     Key_No, Char_No : integer;

     Key_Char : char;

     Key_Number : integer;

     index : integer;

     Hash_Addr : integer;

     Bool_Key_Arr : BOOL_KEY_TYPE;

     Key_Word_Arr : BOOL_WORD_TYPE;
```

```
     Final_Key_Byte : BOOL_TYPE;

     Temp_Two_Bytes : TWO_BYTES_TYPE;

     Temp_One_Byte : BOOL_TYPE;

     Bucket_Arr : BUCKET_POINTER_ARRAY;

     Count_Arr : COUNT_KEY_TYPE;
```

```
function power (x, y : integer) : integer;

var
     sum : integer;
     i : integer;

begin
     sum := 1;
     for i := 1 to y do
       begin
          sum := sum * x;
       end;
end;
```

```
procedure Int_To_Bool_Convert (number:integer; var Bool_Arr:
                                                    BOOL_TYPE);

var  i : integer;

begin
     for i := 1 to MAX_BOOL_DIGIT do
          Bool_Arr(.i.) := false;

     i := 1;
     while (number >= 2) and (i <= MAX_BOOL_DIGIT) do
       begin

          if (number mod 2) =1 then
               Bool_Arr(.i.) := true
```

```
          else
               Bool_Arr(.i.) := false;

          number := number div 2;
          i := i + 1;
       end;

     if (number = 1) and (i <= MAX_BOOL_DIGIT) then
          Bool_Arr(.i.) := true;

end;




procedure Initialization;

const
     char_divisor = 20;
     number_divisor = 10;

var  i, j, k: integer;
     number : integer;
     character : char;

begin
     Hash_Type := FOLD_SHIFTING;
   {
     Hash_Type := ALGEBRAIC_CODING;
     Hash_Type := DIGIT_ANALYSIS;
     Hash_Type := DIVISION;
     Hash_Type := FOLDING;
     Hash_Type := MIDSQUARE;
     Hash_Type := MULTIPLICATIVE;
     Hash_Type := RADIX;
     Hash_Type := RANDOM;
   }

     for i := 1 to NUM_ASCII_CHAR do
       begin

          with ASCII_Arr(.i.) do
            begin
               ch := '?';
               ASCII_num := 777;

               for j := 1 to MAX_BOOL_DIGIT do
                    bool_rep(.j.) := false;
            end;
```

```
    end;


  for i := 1 to NUM_ASCII_CHAR do
    begin
       read(character);

       ASCII_Arr(.i.).ch := character;

       if i mod char_divisor = 0 then
            readln;
    end;

  readln;


  for i :=1 to NUM_ASCII_CHAR do
    begin
       read(number);

       ASCII_ARR(.i.).ASCII_num := number;

       if i mod number_divisor = 0 then
            readln;

    end;

 if DEBUG_FLAG then
 begin
 writeln;
 writeln;
 writeln;
 writeln;
 writeln('  <ASCII CHAR> <ORD NO> <BINARY ASCII>');
 end;

  for i := 1 to NUM_ASCII_CHAR do
    begin

       with ASCII_Arr(.i.) do
         begin

            Int_To_Bool_Convert(ASCII_num, bool_rep);

            if DEBUG_FLAG then
              begin

                 write('     ',ch,' :       ');
                 write(ASCII_num:4,'     ');

                 for k := MAX_BOOL_DIGIT downto 1 do
                   begin
                      if bool_rep(.k.) then
                           write(1:2)
                      else
```

```
                                write(0:2);
                        end;

                        writeln;
                        writeln;
                    end;  {if}
                end;  {with}

        end;    {for}

        for i := 0 to BUCKET_SIZE do
          begin
             Bucket_Arr(.i.) := nil;
          end;

end;




procedure Clean_Two_Temp_Bytes;

var
        i : integer;

begin
        for i := 1 to MAX_BOOL_DIGIT*2 do
          begin
             Temp_Two_Bytes(.i.) := false;
          end;
end;




procedure Clean_One_Temp_Byte;

var
        i : integer;

begin
        for i := 1 to MAX_BOOL_DIGIT do
          begin
             Temp_One_Byte(.i.) := false;
          end;
```

```
procedure Clean_Key_Word_Arr;

var
     i : integer;

begin
     for i := 1 to NUM_BITS_IN_WORD do
       begin
          Key_Word_Arr (.i.) := false;
       end;
end;




procedure Init_For_Do_Loop;

var  i: integer;


begin

     Char_No := 1;

     for i := 1 to NUM_IDENT_CHAR do
       begin
          Key_Register(.i.) := ' ';
       end;
end;




function More_Chars_Left_For_Key : boolean;

begin

     if (((Char_No > NUM_IDENT_CHAR) or eoln) or eof) then
       begin
          More_Chars_Left_For_Key := false;
       end
     else
          More_Chars_Left_For_Key := true;
```

```
end;



procedure Read_A_Char;

begin
     read(Key_Char);
     Key_Register(.Char_No.) := Key_Char;
end;



procedure Look_Up_Char_In_ASCII_Num_Table(var idx:integer);

var  found : boolean;
     i : integer;

begin
     i := 1;
     found := false;
     repeat
          if ASCII_Arr(.i.).ch = Key_Char then
            begin
               found := true;
               idx := i;
            end
          else
            begin
               i := i + 1;
            end;
     until found or (i > NUM_ASCII_CHAR);

      if (i > NUM_ASCII_CHAR) and (not found) then
           idx := 64;

end;



procedure Save_Binary_Num(idx: integer);

var  i : integer;

begin
```

```
      with Bool_Key_Arr(.Char_No.) do
      begin

      for i := 1 to MAX_BOOL_DIGIT do
        begin
           bool_rep(.i.) := ASCII_Arr(.idx.).bool_rep(.i.);
        end;

      end;  {with}

end;




function EX_OR (Bit_X, Bit_Y: boolean): boolean;

begin

      if Bit_X and Bit_Y then EX_OR := false

      else if Bit_X and (not Bit_Y) then EX_OR := true

      else if (not Bit_X) and Bit_Y then EX_OR := true

      else if (not Bit_X) and (not Bit_Y) then EX_OR := false;

end;




procedure Bool_To_Int_Convert (Bool_Arr: BOOL_TYPE; var Int_Value:
                               integer);

var   sum : integer;
      i : integer;

begin

      sum := 0;

      for i := MAX_BUCKET_ADDR_BITS downto 1 do
        begin

           if Bool_Arr(.i.) then
```

```
                sum := 2 * sum + 1
            else
                sum := 2 * sum;

        end;

    Int_Value :=  sum;
end;
```

```
{====================<Do_Folding>================================}

procedure Do_Folding;


procedure Store_First_Word_Of_Key;


    i, j : integer;

begin
    for i := 1 to NUM_BYTES_IN_WORD do
      begin
        with Bool_Key_Arr(.i.) do
          begin
            for j := 1 to MAX_BOOL_DIGIT do
              begin
                Key_Word_Arr (.(i-1)*MAX_BOOL_DIGIT + j.)
                              := bool_rep (.j.);
              end;
          end;
      end;
end;



procedure Ex_Or_Words (Word_Num : integer);

var
    i, j : integer;
    Byte_Offset : integer;
    Word_Index : integer;
    Word_Start : integer;
```

```
     Word_End : integer;

    n
     Byte_Offset := NUM_BYTES_IN_WORD - 1;
     Word_End := Word_Num + Byte_Offset;

     Word_Index := 0;
     for i := Word_Num to Word_End do
       begin
          Word_Start := Word_Index * MAX_BOOL_DIGIT;
          with Bool_Key_Arr(.i.) do
            begin
               for j := 1 to MAX_BOOL_DIGIT do
                 begin
                    Key_Word_Arr(.Word_Start+j.)
                         := Ex_Or (Key_Word_Arr(.(Word_Start+j).),
                                                   bool_rep(.j.));
                 end;
            end;
           Word_Index := Word_Index + 1;
       end;
end;




procedure Fold_The_Key_Word;

var
     i : integer;

begin
     for i := 1 to BITS_FOR_TWO_BYTES do
       begin
          Temp_Two_Bytes(.i.) := EX_OR (Key_Word_Arr(.i.),
                    Key_Word_Arr(.BITS_FOR_TWO_BYTES + i.));
       end;
end;



procedure Fold_Two_Bytes_To_One;

var
     i : integer;

begin
     for i := 1 to MAX_BOOL_DIGIT do
       begin
          Temp_One_Byte (.i.) := Ex_Or (Temp_Two_Bytes(.i.),
                    Temp_Two_Bytes(.MAX_BOOL_DIGIT+i.))
       end;
end;
```

```
{---------------------< Do_Folding Starts Here >-----------------------}
begin  {Do_Folding}
     Store_First_Word_of_Key;
     Ex_Or_Words (NUM_BYTES_IN_WORD * 1 + 1);
     Ex_Or_Words (NUM_BYTES_IN_WORD * 2 + 1);
     Ex_Or_Words (NUM_BYTES_IN_WORD * 3 + 1);
     Fold_The_Key_Word;
     Fold_Two_Bytes_To_One;  {for some hash functions}
end;  {Do_Folding}
```

```
procedure Get_Number_From_Folded_Key (var Key_Num : integer);
var
     Long_Key_Num, Jumbo_Key_Num : integer;
     sum : integer;
     i : integer;
begin
     sum := 0;
     for i := NUM_BITS_IN_WORD downto 1 do
       begin
          if Key_Word_Arr(.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;
     Jumbo_Key_Num := sum;

     sum := 0;
     for i := MAX_BOOL_DIGIT * 2 downto 1 do
       begin
```

```
          if Temp_Two_Bytes(.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;

     Long_Key_Num := sum;

     case Hash_Type of
          ALGEBRAIC_CODING : Key_Num := Jumbo_Key_Num;
          DIGIT_ANALYSIS   : Key_Num := 0;  { Get from other source }
          DIVISION         : Key_Num := Jumbo_Key_Num;
          FOLDING          : Key_Num := 0;  { Get from other source }
          FOLD_SHIFTING    : Key_Num := 0;  { Get from other source }
          MIDSQUARE        : Key_Num := Long_Key_Num;
          MULTIPLICATIVE   : Key_Num := Long_Key_Num;
          RADIX            : Key_Num := Long_Key_Num;
          RANDOM           : Key_Num := Long_Key_Num;
       end;  {case}
end;  { Get_Number_From_Folded_Key }



function Hash_Algebraic_Coding (Key_Num : integer) : integer;
const
     DIVISOR = 1021; {509,9901,Try 263, 401, 1021, 4409}

     temp : integer;
```

```
beqin

     temp := Key_Num MOD DIVISOR;

     Hash_Algebraic_Coding := temp MOD NO_BUCKETS;

end;




function Hash_Digit_Analysis : integer;

{ The bits from 1 to 4 and from 9 to 12 in two bytes folded key are
  selected to produce a hash address.}

const
     DA4_FLAG = false;   {If this flag is false, two lowest bits from
                          4 bytes.}

     U_UPPER_BIT = 12;
     U_LOWER_BIT = 9;

     L_UPPER_BIT = 4;
     L_LOWER_BIT = 1;
var
     i : integer;
     sum : integer;


begin   {Digit_Analysis_Method}

   sum := 0;

   if DA4_FLAG then
    begin
     for i := U_UPPER_BIT downto U_LOWER_BIT do
       begin
          if Temp_Two_Bytes(.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;

     for i := L_UPPER_BIT downto L_LOWER_BIT do
       begin
          if Temp_Two_Bytes(.i.) then
```

```
                sum := sum * 2 + 1
           else
                sum := sum * 2;
        end;
     end
    else
     begin
      Temp_One_Byte(.1.) := Key_Word_Arr(.1.);
      Temp_One_Byte(.2.) := Key_Word_Arr(.2.);
      Temp_One_Byte(.3.) := Key_Word_Arr(.9.);
      Temp_One_Byte(.4.) := Key_Word_Arr(.10.);
      Temp_One_Byte(.5.) := Key_Word_Arr(.17.);
      Temp_One_Byte(.6.) := Key_Word_Arr(.18.);
      Temp_One_Byte(.7.) := Key_Word_Arr(.25.);
      Temp_One_Byte(.8.) := Key_Word_Arr(.26.);

      Bool_To_Int_Convert(Temp_One_Byte, sum);
     end;

   Hash_Digit_Analysis := sum;

end;    {Digit_Analysis_Method}
```

```
function Hash_Division (Key_Value : integer) : integer;

const
     DIVISOR = 259;

var
     Temp_Value : integer;

begin

     Temp_Value := 0;

     Temp_Value := Key_Value MOD DIVISOR;

     if Temp_Value >= No_Buckets then
          Hash_Division := Temp_Value - No_Buckets
     else
          Hash_Division := Temp_Value;

end;
```

```
function Hash_Folding : integer;

{ The 32 bits word has been divided into 3 groups such as the leftmost
  11 bits, middle 11 bits, and rightmost 10 bits.  The rightmost 10
  bits are folded into the corresponding middle bits }


procedure Fold_Right_Side;

var
     i, j : integer;

begin

     i := 1;
     j := 20;

     while i <= 10 do
       begin
          Key_Word_Arr (.j.) := Ex_Or (Key_Word_Arr (.i.),
                                       Key_Word_Arr (.j.));

          i := i + 1;
          j := j - 1;
       end;
end;




procedure Fold_Left_Side;

var
     i, j : integer;

begin

     i := 11;
     j := 32;

     while i <= 21 do
       begin
          Key_Word_Arr (.i.) := Ex_Or (Key_Word_Arr (.j.),
                                       Key_Word_Arr (.i.));

          i := i + 1;
```

```
          j := j - 1;
        end;



function Get_Folding_Value : integer;

var
     i : integer;
     sum : integer;

begin

     sum := 0;
     { get 11-18, 12-19, and 13-20 bits }
     for i := 18 downto 11 do
       begin
          if Key_Word_Arr (.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;

     Get_Folding_Value := sum;

end;



{-----------------< Folding_Method >-------------------------------}

begin

     Fold_Right_Side;

     Fold_Left_Side;

     Hash_Folding := Get_Folding_Value;

end;
```

```
    tion Hash_Shift_Folding : integer;

{ The 32 bits word has been divided into 3 groups such as the leftmost
  11 bits, middle 11 bits, and rightmost 10 bits.  The rightmost 10
  bits are folded into the corresponding middle bits }


procedure Fold_Right_Side;

var
     i, j : integer;

begin

     i := 1;
     j := 11;

     while i <= 10 do
       begin
          Key_Word_Arr (.j.) := Ex_Or (Key_Word_Arr (.i.),
                                       Key_Word_Arr (.j.));

          i := i + 1;
          j := j + 1;
       end;
end;




procedure Fold_Left_Side;

var
     i, j : integer;

begin

     i := 11;
     j := 22;

     while i <= 21 do
       begin
          Key_Word_Arr (.i.) := Ex_Or (Key_Word_Arr (.j.),
                                       Key_Word_Arr (.i.));

          i := i + 1;
          j := j + 1;
       end;
end;
```

```
function Get_Folding_Value : integer;

var
     i : integer;
     sum : integer;

begin

     sum := 0;
     for i := 18 downto 11 do
       begin
          if Key_Word_Arr (.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;

     Get_Folding_Value := sum;

end;



{-----------------< Shift_Folding_Method >-----------------------}

begin

     Fold_Right_Side;

     Fold_Left_Side;

     Hash_Shift_Folding := Get_Folding_Value;

end;




function Hash_Midsquare (Key_Value : integer) : integer;

var
     Square_Key : integer;
     Temp_Key_Word : BOOL_WORD_TYPE;


   cedure Store_Temp_Two_Bytes (number : integer);
```

```
var
     i, bit : integer;

begin

     for i := 1 to NUM_BITS_IN_WORD do
       begin

          bit := number MOD 2;

          number := number DIV 2;

          if bit = 1 then
               Temp_Key_Word (.i.) := true

          else if bit = 0 then
               Temp_Key_Word (.i.) := false;

       end;
end;    { Store-Temp_Two_Bytes }



function Get_Midsquare_Hash_Value : integer;

const
     START_BIT = 13;  {5 for Short_Key_Num}
     END_BIT = 20;    {12 for Short_Key_Num}

var
     i : integer;
     sum : integer;

begin

     sum := 0;

     for i := END_BIT downto START_BIT do
       begin
          if Temp_Key_Word(.i.) then
               sum := sum * 2 + 1

          else
               sum := sum * 2;
       end;

     Get_Midsquare_Hash_Value := sum;

end;


{ ----------------< Midsquare_Method starts from here >------------}
```

```
begin
     Square_Key := sqr (Key_Value);
     Clean_Two_Temp_Bytes;
     Store_Temp_Two_Bytes (Square_Key);
     Hash_Midsquare := Get_Midsquare_Hash_Value;
end;  { Midsquare_Method }
```

```
   ction Hash_Multiplicative (Key_Value : integer) : integer;

const
    {C_Value = 0.6180339887 or 0.3819660113 from Tanenbaum }
     C_Value = 0.3819660113;

var
     fraction, temp : real;

begin
     temp := C_Value * Key_Value;
     fraction := temp - trunc(temp);
     Hash_Multiplicative := trunc (No_Buckets * fraction);
end;   { Multiplicative_Method }
```

```
unction Hash_Radix (Key_Value : integer) : integer;
```

```
{ Convert assumed Key_Value in base 11 to a number in base 10 }

const
     NEW_BASE = 11;

var
     Hash_Value : integer;
     quotient : integer;
     i : integer;
     sum : integer;

begin

     sum := 0;

     for i := 5 downto 0 do
       begin
         {quotient := (Key_Value div (10 ** i))}
          quotient := (Key_Value div (power(10,i)));
          sum := sum * NEW_BASE + quotient;
         {Key_Value := Key_Value - quotient * (10 ** i)}
          Key_Value := Key_Value - quotient * (power(10,i));
       end;

     Hash_Radix := sum MOD No_Buckets;

end;




function Hash_Random (Key_Value : integer) : integer;

const
     RANDOM_DEBUG = true;
var
     Temp_Num : integer;


function Random_Number (seed : integer) : integer;

const
   { MULTIPLIER = 25173
     INCREMENT = 13849
     MODULUS = 65536 }

     MULTIPLIER = 23209;
     INCREMENT = 131071;
     MODULUS = 44497;
```

```
begin

     Random_Number := (MULTIPLIER * seed + INCREMENT) MOD MODULUS;

end;


{-----------------< Random_Method >-----------------------------}

begin
     Temp_Num := Random_Number(Key_Value);

     if RANDOM_DEBUG then
          writeln(' KEY VALUE : ',Key_Value:5,
                  '    RANDOM NUMBER : ', Temp_Num:5,
                  '    BUCKET NUMBER : ',Temp_Num MOD No_Buckets:3);

     Hash_Random := Temp_Num MOD No_Buckets;

end;   {Random_Method}









procedure Store_Key_In_Addressed_Bucket (address : integer);

var  Key_Pt : LINK;
     i : integer;

begin

     new(Key_Pt);

     for i := 1 to NUM_IDENT_CHAR do
       begin
          Key_Pt@.Key_Arr(.i.) := Key_Register(.i.);
       end;

     if DEBUG_FLAG then
       begin
          write('   KEY ATTRIBUTE : ');
          for i := 1 to NUM_IDENT_CHAR do
            begin
               write(Key_Register(.i.));
            end;

          writeln('   BUCKET ADDRESS : ',address:3);
          writeln;
```

```
        end;

      readln;
      Key_Pt@.next := Bucket_Arr(.address.);

      Bucket_Arr(.address.) := Key_Pt;

end;




procedure Print_Keys_In_Each_Bucket;

var   i, j : integer;
      pt : LINK;
      count : integer;


begin

      for i := 0 to BUCKET_SIZE do
        begin
           count := 0;
           pt := Bucket_Arr(.i.);

           if DEBUG_FLAG then
             begin
                writeln;
                writeln('----- Bucket Number : ', i : 3,'----');
                writeln;
             end;

           while pt <> nil do
             begin

                if DEBUG_FLAG then
                  begin
                     write('      ');

                     for j := 1 to NUM_IDENT_CHAR do
                       begin
                          write(pt@.Key_Arr(.j.));
                       end;
                     writeln;
                  end;

                pt := pt@.next;
                count := count + 1;
```

```
            end;  {while}

          if DEBUG_FLAG then writeln;

          Count_Arr(.i.) := count;

       end;  {for}

end;




procedure Print_Statistics;

const
     MAX_KEYS = 40;
     EMPTY_STRING =  '                              ';
     LINE_LIMIT = 1;

var  i : integer;
     idx : integer;
     max, sum : integer;

     Count_Keys : array (.0..MAX_KEYS.) of integer;

     variance, Var_Sum : real;
     mean : integer;

     Keys_Over_40 : integer;

begin

     max := 0;
     sum := 0;
     Keys_Over_40 := 0;
     variance := 0;
     Var_Sum := 0;
     mean := MAX_NUM_KEYS div (BUCKET_SIZE + 1);

     for i := 0 to MAX_KEYS do
       begin
          Count_Keys(.i.) := 0;
       end;

     for i := 0 to BUCKET_SIZE do
       begin

          if DEBUG_FLAG then
            begin
```

```
            write(' Bucket Number : ',i:3);
            writeln(' contains ', Count_Arr(.i.):3, ' keys.');
            writeln;
        end;

    sum := sum + Count_Arr(.i.);

    if Count_Arr(.i.) > max then
        max := Count_Arr(.i.);

    if Count_Arr(.i.) = 0 then
        Count_Keys(.0.) := Count_Keys(.0.) + 1

    else if Count_Arr(.i.) = 1 then
        Count_Keys(.1.) := Count_Keys(.1.) + 1

    else if Count_Arr(.i.) = 2 then
        Count_Keys(.2.) := Count_Keys(.2.) + 1

    else if Count_Arr(.i.) = 3 then
        Count_Keys(.3.) := Count_Keys(.3.) + 1

    else if Count_Arr(.i.) = 4 then
        Count_Keys(.4.) := Count_Keys(.4.) + 1

    else if Count_Arr(.i.) = 5 then
        Count_Keys(.5.) := Count_Keys(.5.) + 1

    else if Count_Arr(.i.) = 6 then
        Count_Keys(.6.) := Count_Keys(.6.) + 1

    else if Count_Arr(.i.) = 7 then
        Count_Keys(.7.) := Count_Keys(.7.) + 1

    else if Count_Arr(.i.) = 8 then
        Count_Keys(.8.) := Count_Keys(.8.) + 1

    else if Count_Arr(.i.) = 9 then
        Count_Keys(.9.) := Count_Keys(.9.) + 1

    else if Count_Arr(.i.) = 10 then
        Count_Keys(.10.) := Count_Keys(.10.) + 1

    else if Count_Arr(.i.) = 11 then
        Count_Keys(.11.) := Count_Keys(.11.) + 1

    else if Count_Arr(.i.) = 12 then
        Count_Keys(.12.) := Count_Keys(.12.) + 1

    else if Count_Arr(.i.) = 13 then
        Count_Keys(.13.) := Count_Keys(.13.) + 1

    else if Count_Arr(.i.) = 14 then
        Count_Keys(.14.) := Count_Keys(.14.) + 1
```

```
        else if Count_Arr(.i.) = 15 then
             Count_Keys(.15.) := Count_Keys(.15.) + 1

        else if Count_Arr(.i.) = 16 then
             Count_Keys(.16.) := Count_Keys(.16.) + 1

        else if Count_Arr(.i.) = 17 then
             Count_Keys(.17.) := Count_Keys(.17.) + 1

        else if Count_Arr(.i.) = 18 then
             Count_Keys(.18.) := Count_Keys(.18.) + 1

        else if Count_Arr(.i.) = 19 then
             Count_Keys(.19.) := Count_Keys(.19.) + 1

        else if Count_Arr(.i.) = 20 then
             Count_Keys(.20.) := Count_Keys(.20.) + 1

        else if Count_Arr(.i.) = 21 then
             Count_Keys(.21.) := Count_Keys(.21.) + 1

        else if Count_Arr(.i.) = 22 then
             Count_Keys(.22.) := Count_Keys(.22.) + 1

        else if Count_Arr(.i.) = 23 then
             Count_Keys(.23.) := Count_Keys(.23.) + 1

        else if Count_Arr(.i.) = 24 then
             Count_Keys(.24.) := Count_Keys(.24.) + 1

        else if Count_Arr(.i.) = 25 then
             Count_Keys(.25.) := Count_Keys(.25.) + 1

        else if Count_Arr(.i.) = 26 then
             Count_Keys(.26.) := Count_Keys(.26.) + 1

        else if Count_Arr(.i.) = 27 then
             Count_Keys(.27.) := Count_Keys(.27.) + 1

        else if Count_Arr(.i.) = 28 then
             Count_Keys(.28.) := Count_Keys(.28.) + 1

        else if Count_Arr(.i.) = 29 then
             Count_Keys(.29.) := Count_Keys(.29.) + 1

        else if Count_Arr(.i.) = 30 then
             Count_Keys(.30.) := Count_Keys(.30.) + 1

        else if Count_Arr(.i.) = 31 then
             Count_Keys(.31.) := Count_Keys(.31.) + 1

        else if Count_Arr(.i.) = 32 then
             Count_Keys(.32.) := Count_Keys(.32.) + 1

        else if Count_Arr(.i.) = 33 then
```

```
            Count_Keys(.33.) := Count_Keys(.33.) + 1

        else if Count_Arr(.i.) = 34 then
            Count_Keys(.34.) := Count_Keys(.34.) + 1

        else if Count_Arr(.i.) = 35 then
            Count_Keys(.35.) := Count_Keys(.35.) + 1

        else if Count_Arr(.i.) = 36 then
            Count_Keys(.36.) := Count_Keys(.36.) + 1

        else if Count_Arr(.i.) = 37 then
            Count_Keys(.37.) := Count_Keys(.37.) + 1

        else if Count_Arr(.i.) = 38 then
            Count_Keys(.38.) := Count_Keys(.38.) + 1

        else if Count_Arr(.i.) = 39 then
            Count_Keys(.39.) := Count_Keys(.39.) + 1

        else if Count_Arr(.i.) = 40 then
            Count_Keys(.40.) := Count_Keys(.40.) + 1;


        if (Count_Arr(.i.) > 40) then
            Keys_Over_40 := Keys_Over_40 + 1;

        Var_Sum := Var_Sum + sqr(Count_Arr(.i.) - mean);

    end;  {for}

  variance := Var_Sum / (BUCKET_SIZE + 1);

  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln('--------------------< STATISTICS >---------------------');
  writeln;
  writeln(' THE ',HASH_NAME,' HASH METHOD IS APPLIED TO ');
  writeln;
  writeln(' THE ',DATA_NAME,' DATA.');
  writeln;
```

```
      writeln;
      writeln(' MEAN SQUARE DEVIATION :   ',
               variance:8:2);
      writeln;
      writeln;
      writeln;

      for i := 0 to MAX_KEYS do
        begin
           write(' ',i:2,' KEYS BUCKET =',Count_Keys(.i.):3,' : ');

           for idx := 1 to Count_Keys (.i.) do
             begin
               {if idx mod LINE_LIMIT = 0 then}
                     write ('*');
             end;
           writeln;
        end;


      writeln;
      writeln(' OVER 40 KEYS BUCKET ');
      write('                =', Keys_Over_40:3,' : ');
      for idx := 1 to Keys_Over_40 do
        begin
           if idx mod LINE_LIMIT = 0 then
                write ('*');
        end;
      writeln;
end;



{*************** MAIN PROGRAM STARTS HERE ******************}

begin

      Initialization;

      {Read Keys one by one converting them to hashed addresses}

      for Key_No := 1 to MAX_NUM_KEYS do

        begin

          Init_For_Do_Loop;

          {Read character by character for a key looking up the
           corresponding ASCII number and convert it to binary number}

          while More_Chars_Left_For_Key do
            begin
               Read_A_Char;
               Look_Up_Char_In_ASCII_Num_Table(index);
```

```
            Save_Binary_Num(index);
            Char_No := Char_No + 1;
          end;

        Do_Folding;

        Get_Number_From_Folded_Key (Key_Number);

        case Hash_Type of
             ALGEBRAIC_CODING : Hash_Addr := Hash_Algebraic_Coding
                                             (Key_Number);
             DIGIT_ANALYSIS   : Hash_Addr := Hash_Digit_Analysis;

             DIVISION         : Hash_Addr := Hash_Division
                                             (Key_Number);
             FOLDING          : Hash_Addr := Hash_Folding;

             FOLD_SHIFTING    : Hash_Addr := Hash_Shift_Folding;

             MIDSQUARE        : Hash_Addr := Hash_Midsquare
                                             (Key_Number);
             MULTIPLICATIVE   : Hash_Addr := Hash_Multiplicative
                                             (Key_Number);
             RADIX            : Hash_Addr := Hash_Radix
                                             (Key_Number);
             RANDOM           : Hash_Addr := Hash_Random
                                             (Key_Number);
          end;  {case}


        {Using the resulting hashed address, store the input key in
         the corresponding bucket.}

        Store_Key_In_Addressed_Bucket (Hash_Addr);

      end; {outer for}

    {Print out all the keys in every hashed buckets.}

    Print_Keys_In_Each_Bucket;

    {Print out all the necessary statistics for an analysis.}

    Print_Statistics;

end.
```

FILE: RANDOM PASCAL A THE GEORGE WASHINGTON UNIVERSITY COMPUTER CENTER

```
program random (input, output);

    ;t

      DATA_NAME = '1024 RANDOMLY CHOSEN NAMES';

      DEBUG_FLAG = true;


      HASH_NAME = 'FOLDING';
    {
      HASH_NAME = 'ALGEBRAIC CODING';
      HASH_NAME = 'DIGIT ANALYSIS';
      HASH_NAME = 'DIVISION';
      HASH_NAME = 'FOLD FOUR SHIFTED WORDS';
      HASH_NAME = 'FOLD SHIFTING';
      HASH_NAME = 'MIDSQUARE';
      HASH_NAME = 'MULTIPLICATIVE';
      HASH_NAME = 'RADIX';
      HASH_NAME = 'RANDOM';
    }

      FOUR_WORDS = 4;
      OFFSET = 5;
      MAX_NUM_KEYS = 1024;
      NUM_IDENT_CHAR = 16;
      MAX_BUCKET_ADDR_BITS = 8;
      MAX_BOOL_DIGIT = 8;
      NUM_ASCII_CHAR = 70;

      NUM_BYTES_IN_WORD = 4;
      NO_BUCKETS = 256;

      NUM_FIRST_EXOR = 8;
      NUM_SECOND_EXOR = 4;
      NUM_THIRD_EXOR = 2;

      BUCKET_SIZE = 255;

      NUM_BITS_IN_WORD = 32;
      BITS_FOR_TWO_BYTES = 16;


type
      KEY_ARRAY_TYPE = array (.1..NUM_IDENT_CHAR.) of char;

      BOOL_TYPE = array (.1..MAX_BOOL_DIGIT.) of boolean;

      BOOL_WORD_TYPE = array (.1..NUM_BITS_IN_WORD.) of boolean;

      CHAR_RECORD_TYPE = record
                           ch : char;
                           ASCII_num : integer;
```

```
                        bool_rep : BOOL_TYPE;
                      end;

    ASCII_TABLE = array (.1..NUM_ASCII_CHAR.) of
                  CHAR_RECORD_TYPE;

    BOOL_KEY_TYPE = array (.1..NUM_IDENT_CHAR.) of
                        record
                            bool_rep : BOOL_TYPE;
                        end;

    TWO_BYTES_TYPE = array (.1..BITS_FOR_TWO_BYTES.) of boolean;

    HASHED_KEY_REG_TYPE = array (.1..MAX_BUCKET_ADDR_BITS.) of boolean;

    COUNT_KEY_TYPE = array (.0..BUCKET_SIZE.) of integer;

    WORD_RECORD = record
                    Word_Array : BOOL_WORD_TYPE;
                  end;

    WORD_TYPE_ARRAY = array (.1..FOUR_WORDS.) of WORD_RECORD;


    LINK = @KEY_RECORD;
    KEY_RECORD = record
                    Key_Arr : KEY_ARRAY_TYPE;
                    next : LINK;
                 end;

    BUCKET_POINTER_ARRAY = array (.0..BUCKET_SIZE.) of LINK;


    HASH_KINDS = (ALGEBRAIC_CODING,
                  DIGIT_ANALYSIS,
                  DIVISION,
                  FOLDING,
                  FOLD_FOUR_SHIFTED_WORDS,
                  FOLD_SHIFTING,
                  MIDSQUARE,
                  MULTIPLICATIVE,
                  RADIX,
                  RANDOM);


var
    Hash_Type : HASH_KINDS;

    Key_Register : KEY_ARRAY_TYPE;

    ASCII_Arr : ASCII_TABLE;

    Key_No, Char_No : integer;

    Key_Char : char;
```

```
     Key_Number : integer;
     index : integer;
     Hash_Addr : integer;
     Bool_Key_Arr : BOOL_KEY_TYPE;
     Key_Word_Arr : BOOL_WORD_TYPE;
     Final_Key_Byte : BOOL_TYPE;
     Temp_Two_Bytes : TWO_BYTES_TYPE;
     Temp_One_Byte : BOOL_TYPE;
     Temp_Word : WORD_TYPE_ARRAY;
     Bucket_Arr : BUCKET_POINTER_ARRAY;
     Count_Arr : COUNT_KEY_TYPE;
     FS_ADDR_RANGE : integer; {1, 2, 3, 4}
```

```
function power (x, y : integer) : integer;

var
     sum : integer;
     i : integer;

begin
     sum := 1;
     for i := 1 to y do
       begin
          sum := sum * x;
       end;
end;
```

```
   cedure Int_To_Bool_Convert (number:integer; var Bool_Arr:
                                                   BOOL_TYPE);
```

```
var  i : integer;

begin
     for i := 1 to MAX_BOOL_DIGIT do
          Bool_Arr(.i.) := false;

     i := 1;
     while (number >= 2) and (i <= MAX_BOOL_DIGIT) do
       begin

          if (number mod 2) =1 then
               Bool_Arr(.i.) := true
          else
               Bool_Arr(.i.) := false;

          number := number div 2;
          i := i + 1;
       end;

     if (number = 1) and (i <= MAX_BOOL_DIGIT) then
          Bool_Arr(.i.) := true;

end;
```

```
procedure Initialization;

const
     char_divisor = 20;
     number_divisor = 10;

var  i, j, k: integer;
     number : integer;
     character : char;

begin
     FS_ADDR_RANGE := 4;    {1, 2, 3, 4}

     Hash_Type := FOLDING;
   {
     Hash_Type := ALGEBRAIC_CODING;
     Hash_Type := DIGIT_ANALYSIS;
     Hash_Type := DIVISION;
     Hash_Type := FOLD_FOUR_SHIFTED_WORDS;
     Hash_Type := FOLD_SHIFTING;
     Hash_Type := MIDSQUARE;
     Hash_Type := MULTIPLICATIVE;
```

```
    Hash_Type := RADIX;
    Hash_Type := RANDOM;
  }

    for i := 1 to NUM_ASCII_CHAR do
      begin

         with ASCII_Arr(.i.) do
           begin
              ch := '?';
              ASCII_num := 777;

              for j := 1 to MAX_BOOL_DIGIT do
                   bool_rep(.j.) := false;
           end;

      end;


    for i := 1 to NUM_ASCII_CHAR do
      begin
         read(character);

         ASCII_Arr(.i.).ch := character;

         if i mod char_divisor = 0 then
              readln;
      end;

    readln;


    for i :=1 to NUM_ASCII_CHAR do
      begin
         read(number);

         ASCII_ARR(.i.).ASCII_num := number;

         if i mod number_divisor = 0 then
              readln;

      end;

   if DEBUG_FLAG then
   begin
   writeln;
   writeln;
   writeln;
   writeln;
   writeln('  <ASCII CHAR> <ORD NO> <BINARY ASCII>');
   end;

    for i := 1 to NUM_ASCII_CHAR do
      begin
```

```
          with ASCII_Arr(.i.) do
            begin

               Int_To_Bool_Convert(ASCII_num, bool_rep);

               if DEBUG_FLAG then
                 begin

                    write('     ',ch,' :       ');
                    write(ASCII_num:4,'     ');

                    for k := MAX_BOOL_DIGIT downto 1 do
                      begin
                         if bool_rep(.k.) then
                              write(1:2)
                         else
                              write(0:2);
                      end;

                      writeln;
                      writeln;
                 end;  {if}
            end;  {with}

     end;    {for}

     for i := 0 to BUCKET_SIZE do
       begin
          Bucket_Arr(.i.) := nil;
       end;

end;




procedure Clean_Two_Temp_Bytes;

var
     i : integer;

begin
     for i := 1 to MAX_BOOL_DIGIT*2 do
       begin
          Temp_Two_Bytes(.i.) := false;
       end;
end;
```

```
procedure Clean_One_Temp_Byte;

var
     i : integer;

begin
     for i := 1 to MAX_BOOL_DIGIT do
       begin
          Temp_One_Byte(.i.) := false;
       end;
end;




procedure Clean_Key_Word_Arr;

var
     i : integer;

   in
     for i := 1 to NUM_BITS_IN_WORD do
       begin
          Key_Word_Arr (.i.) := false;
       end;
end;




procedure Init_For_Do_Loop;

var  i: integer;


begin

     Char_No := 1;

     for i := 1 to NUM_IDENT_CHAR do
       begin
          Key_Register(.i.) := ' ';
       end;
end;
```

```
function More_Chars_Left_For_Key : boolean;

begin

     if (((Char_No > NUM_IDENT_CHAR) or eoln) or eof) then
       begin
          More_Chars_Left_For_Key := false;
       end
     else
          More_Chars_Left_For_Key := true;
end;




procedure Read_A_Char;

begin
     read(Key_Char);
     Key_Register(.Char_No.) := Key_Char;
   ;




procedure Look_Up_Char_In_ASCII_Num_Table(var idx:integer);

var  found : boolean;
     i : integer;

begin
     i := 1;
     found := false;
     repeat
          if ASCII_Arr(.i.).ch = Key_Char then
            begin
               found := true;
               idx := i;
            end
          else
            begin
               i := i + 1;
            end;
     until found or (i > NUM_ASCII_CHAR);
```

```
      if (i > NUM_ASCII_CHAR) and (not found) then
           idx := 64;

end;



procedure Save_Binary_Num(idx: integer);

var  i : integer;

begin

     with Bool_Key_Arr(.Char_No.) do
     begin

     for i := 1 to MAX_BOOL_DIGIT do
       begin
          bool_rep(.i.) := ASCII_Arr(.idx.).bool_rep(.i.);
       end;

     end;  {with}

end;



function EX_OR (Bit_X, Bit_Y: boolean): boolean;

begin

     if Bit_X and Bit_Y then EX_OR := false

     else if Bit_X and (not Bit_Y) then EX_OR := true

     else if (not Bit_X) and Bit_Y then EX_OR := true

     else if (not Bit_X) and (not Bit_Y) then EX_OR := false;

end;
```

```
    edure Bool_To_Int_Convert (Bool_Arr: BOOL_TYPE; var Int_Value:
                               integer);

var  sum : integer;
     i : integer;

begin

     sum := 0;

     for i := MAX_BUCKET_ADDR_BITS downto 1 do
       begin

          if Bool_Arr(.i.) then
               sum := 2 * sum + 1
          else
               sum := 2 * sum;

       end;

     Int_Value :=  sum;
end;









{====================<Do_Folding>================================}

procedure Do_Folding;


procedure Store_First_Word_Of_Key;

var
     i, j : integer;

begin
     for i := 1 to NUM_BYTES_IN_WORD do
       begin
          with Bool_Key_Arr(.i.) do
            begin
               for j := 1 to MAX_BOOL_DIGIT do
                 begin
                   Key_Word_Arr (.(i-1)*MAX_BOOL_DIGIT + j.)
```

```
                                    := bool_rep (.j.);
                end;
            end;
        end;
end;



procedure Ex_Or_Words (Word_Num : integer);

var
     i, j : integer;
     Byte_Offset : integer;
     Word_Index : integer;
     Word_Start : integer;
     Word_End : integer;

begin
     Byte_Offset := NUM_BYTES_IN_WORD - 1;
     Word_End := Word_Num + Byte_Offset;

     Word_Index := 0;
     for i := Word_Num to Word_End do
       begin
          Word_Start := Word_Index * MAX_BOOL_DIGIT;
          with Bool_Key_Arr(.i.) do
            begin
               for j := 1 to MAX_BOOL_DIGIT do
                 begin
                    Key_Word_Arr(.Word_Start+j.)
                         := Ex_Or (Key_Word_Arr(.(Word_Start+j).),
                                                   bool_rep(.j.));
                 end;
            end;
           Word_Index := Word_Index + 1;
       end;
end;



procedure Fold_The_Key_Word;

var
     i : integer;

begin
     for i := 1 to BITS_FOR_TWO_BYTES do
       begin
          Temp_Two_Bytes(.i.) := EX_OR (Key_Word_Arr(.i.),
                    Key_Word_Arr(.BITS_FOR_TWO_BYTES + i.));
       end;
end;
```

```
procedure Fold_Two_Bytes_To_One;

var
     i : integer;

begin
     for i := 1 to MAX_BOOL_DIGIT do
       begin
          Temp_One_Byte (.i.) := Ex_Or (Temp_Two_Bytes(.i.),
                    Temp_Two_Bytes(.MAX_BOOL_DIGIT+i.))
       end;
end;



{----------------------< Do_Folding Starts Here >-----------------------}

begin  {Do_Folding}

     Store_First_Word_of_Key;

     Ex_Or_Words (NUM_BYTES_IN_WORD * 1 + 1);

     Ex_Or_Words (NUM_BYTES_IN_WORD * 2 + 1);

     Ex_Or_Words (NUM_BYTES_IN_WORD * 3 + 1);

     Fold_The_Key_Word;

     Fold_Two_Bytes_To_One;  {for some hash functions}

end;  {Do_Folding}









procedure Get_Number_From_Folded_Key (var Key_Num : integer);

var
     Long_Key_Num, Jumbo_Key_Num : integer;
     sum : integer;
     i : integer;
```

```
begin
     sum := 0;
     for i := NUM_BITS_IN_WORD downto 1 do
       begin
          if Key_Word_Arr(.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;

     Jumbo_Key_Num := sum;


     sum := 0;
     for i := MAX_BOOL_DIGIT * 2 downto 1 do
       begin
          if Temp_Two_Bytes(.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;

     Long_Key_Num := sum;


     case Hash_Type of

          ALGEBRAIC_CODING : Key_Num := Jumbo_Key_Num;

          DIGIT_ANALYSIS   : Key_Num := 0;  { Get from other source }

          DIVISION         : Key_Num := Jumbo_Key_Num;

          FOLDING          : Key_Num := 0;  { Get from other source }

          FOLD_FOUR_SHIFTED_WORDS :
                             Key_Num := 0;  { Ger from other source }

          FOLD_SHIFTING    : Key_Num := 0;  { Get from other source }

          MIDSQUARE        : Key_Num := Long_Key_Num;

          MULTIPLICATIVE   : Key_Num := Long_Key_Num;

          RADIX            : Key_Num := Long_Key_Num;

          RANDOM           : Key_Num := Long_Key_Num;

       end;  {case}

end;  { Get_Number_From_Folded_Key }
```

```
function Hash_Algebraic_Coding (Key_Num : integer) : integer;

const
     DIVISOR = 1021; {509,9901,Try 263, 401, 1021, 4409}

var
     temp : integer;

begin

     temp := Key_Num MOD DIVISOR;

     Hash_Algebraic_Coding := temp MOD NO_BUCKETS;

end;
```

```
function Hash_Digit_Analysis : integer;

{ The bits from 1 to 4 and from 9 to 12 in two bytes folded key are
  selected to produce a hash address.}

const
     DA4_FLAG = false;   {If this flag is false, two lowest bits from
                          4 bytes.}

     U_UPPER_BIT = 12;
     U_LOWER_BIT = 9;

     L_UPPER_BIT = 4;
     L_LOWER_BIT = 1;
var
     i : integer;
     sum : integer;
```

```
    n   {Digit_Analysis_Method}

   sum := 0;

   if DA4_FLAG then
    begin
     for i := U_UPPER_BIT downto U_LOWER_BIT do
       begin
          if Temp_Two_Bytes(.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;

     for i := L_UPPER_BIT downto L_LOWER_BIT do
       begin
          if Temp_Two_Bytes(.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;
    end
   else
    begin
     Temp_One_Byte(.1.) := Key_Word_Arr(.1.);
     Temp_One_Byte(.2.) := Key_Word_Arr(.2.);
     Temp_One_Byte(.3.) := Key_Word_Arr(.9.);
     Temp_One_Byte(.4.) := Key_Word_Arr(.10.);
     Temp_One_Byte(.5.) := Key_Word_Arr(.17.);
     Temp_One_Byte(.6.) := Key_Word_Arr(.18.);
     Temp_One_Byte(.7.) := Key_Word_Arr(.25.);
     Temp_One_Byte(.8.) := Key_Word_Arr(.26.);

     Bool_To_Int_Convert(Temp_One_Byte, sum);
    end;

   Hash_Digit_Analysis := sum;

end;    {Digit_Analysis_Method}




function Hash_Division (Key_Value : integer) : integer;

const
```

```
     DIVISOR = 259;


     Temp_Value : integer;

begin

     Temp_Value := 0;

     Temp_Value := Key_Value MOD DIVISOR;

     if Temp_Value >= No_Buckets then
          Hash_Division := Temp_Value - No_Buckets
     else
          Hash_Division := Temp_Value;

end;
```

```
function Hash_Folding : integer;

{ The 32 bits word has been divided into 3 groups such as the leftmost
  11 bits, middle 11 bits, and rightmost 10 bits.  The rightmost 10
  bits are folded into the corresponding middle bits }


procedure Fold_Right_Side;

var
     i, j : integer;

begin

     i := 1;
     j := 20;

     while i <= 10 do
       begin
          Key_Word_Arr (.j.) := Ex_Or (Key_Word_Arr (.i.),
                                       Key_Word_Arr (.j.));

          i := i + 1;
          j := j - 1;
       end;
end;
```

```
procedure Fold_Left_Side;

var
     i, j : integer;

begin

     i := 11;
     j := 32;

     while i <= 21 do
       begin
          Key_Word_Arr (.i.) := Ex_Or (Key_Word_Arr (.j.),
                                       Key_Word_Arr (.i.));

          i := i + 1;
          j := j - 1;
       end;
end;



function Get_Folding_Value : integer;


     i : integer;
     sum : integer;

begin

     sum := 0;
     { get 11-18, 12-19, and 13-20 bits }
     for i := 20 downto 13 do
       begin
          if Key_Word_Arr (.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;

     Get_Folding_Value := sum;

end;



{-----------------< Folding_Method >-------------------------------}

   in
```

```
     Fold_Right_Side;

     Fold_Left_Side;

     Hash_Folding := Get_Folding_Value;

end;
```

```
function Hash_FS_4 : integer;


{Shift the key word in 4 different ways based on the input parameters
 and fold them together with exclusive-OR operation.}

procedure shift(No_Shifts, No_Word : integer);

var
     i, j : integer;
     Num_Processes : integer;

begin

     Num_Processes := NUM_BITS_IN_WORD - No_Shifts;

     for i := 1 to Num_Processes do
       begin
          Temp_Word(.No_Word.).Word_Array(.i.) :=
                                Key_Word_Arr (.i + No_Shifts.);
       end;

     j := 1;

     for i := Num_Processes + 1 to NUM_BITS_IN_WORD do
       begin
          Temp_Word(.No_Word.).Word_Array(.i.) :=
                                Key_Word_Arr(.j.);
          j := j + 1;
       end;
end;
```

```
   edure Fold_First_Two_Words;

var
     i : integer;

begin

     for i := 1 to NUM_BITS_IN_WORD do
       begin

          Temp_Word(.1.).Word_Array(.i.) :=
               EX_OR(Temp_Word(.1.).Word_Array(.i.),
                    Temp_Word(.2.).Word_Array(.i.));

       end;
end;
```

```
rocedure Fold_Second_Two_Words;

var
     i : integer;

begin

     for i := 1 to NUM_BITS_IN_WORD do
       begin

          Temp_Word(.3.).Word_Array(.i.) :=
               EX_OR(Temp_Word(.3.).Word_Array(.i.),
                    Temp_Word(.4.).Word_Array(.i.));

       end;
end;
```

```
   cedure Fold_Third_Two_Words;
```

```
var
     i : integer;

begin

     for i := 1 to NUM_BITS_IN_WORD do
       begin

          Temp_Word(.1.).Word_Array(.i.) :=
               EX_OR(Temp_Word(.1.).Word_Array(.i.),
                     Temp_Word(.3.).Word_Array(.i.));

       end;
end;




function Addr_Range (First_Bit, Last_Bit : integer): integer;

var
     i : integer;
     sum : integer;

begin

     sum := 0;
     for i := First_Bit to Last_Bit do
       begin
          if Temp_Word(.1.).Word_Array(.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;

       end;

     Addr_Range := sum;

end;




function Get_Address : integer;

var
     i : integer;

begin
     case FS_ADDR_RANGE of
```

```
          1 : Get_Address := Addr_Range(1,8);

          2 : Get_Address := Addr_Range(9,16);

          3 : Get_Address := Addr_Range(17,24);

          4 : Get_Address := Addr_Range(25,32);
     end;
end;
```

```
{-----------------< Fold_Four_Shifted_Words >-----------------------}

{(0,2,4,6) => (0,10,20,30)}
{(0,3,6,1) => (0,11,22,25)}
{(0,4,1,5) => (0,12,17,29)}
{(0,5,2,7) => (0,13,18,31)}
{(0,7,6,5) => (0,15,22,29)}

   in

     Shift( 0,1);

     Shift(15,2);

     Fold_First_Two_Words;

     Shift(22,3);

     Shift(29,4);

     Fold_Second_Two_Words;

     Fold_Third_Two_Words;

     Hash_FS_4 := Get_Address;

end;
```

```
..ction Hash_Shift_Folding : integer;
```

```
{ The 32 bits word has been divided into 3 groups such as the leftmost
     bits, middle 11 bits, and rightmost 10 bits.  The rightmost 10
  bits are folded into the corresponding middle bits }


procedure Fold_Right_Side;

var
     i, j : integer;

begin

     i := 1;
     j := 11;

     while i <= 10 do
       begin
          Key_Word_Arr (.j.) := Ex_Or (Key_Word_Arr (.i.),
                                       Key_Word_Arr (.j.));

          i := i + 1;
          j := j + 1;
       end;
end;




procedure Fold_Left_Side;

var
     i, j : integer;

begin

     i := 11;
     j := 22;

     while i <= 21 do
       begin
          Key_Word_Arr (.i.) := Ex_Or (Key_Word_Arr (.j.),
                                       Key_Word_Arr (.i.));

          i := i + 1;
          j := j + 1;
       end;
end;



tion Get_Folding_Value : integer;
```

```
var
     i : integer;
     sum : integer;

begin

     sum := 0;
     for i := 18 downto 11 do
       begin
          if Key_Word_Arr (.i.) then
               sum := sum * 2 + 1
          else
               sum := sum * 2;
       end;

     Get_Folding_Value := sum;

end;



{-----------------< Shift_Folding_Method >-------------------------}

begin

     Fold_Right_Side;

     Fold_Left_Side;

     Hash_Shift_Folding := Get_Folding_Value;

end;




function Hash_Midsquare (Key_Value : integer) : integer;

var
     Square_Key : integer;
     Temp_Key_Word : BOOL_WORD_TYPE;


procedure Store_Temp_Two_Bytes (number : integer);

var
     i, bit : integer;
```

```
begin

     for i := 1 to NUM_BITS_IN_WORD do
       begin

          bit := number MOD 2;

          number := number DIV 2;

          if bit = 1 then
               Temp_Key_Word (.i.) := true

          else if bit = 0 then
               Temp_Key_Word (.i.) := false;

       end;
end;    { Store-Temp_Two_Bytes }



function Get_Midsquare_Hash_Value : integer;

const
     START_BIT = 11;  {5 for Short_Key_Num}
     END_BIT = 18;    {12 for Short_Key_Num}


     i : integer;
     sum : integer;

begin

     sum := 0;

     for i := END_BIT downto START_BIT do
       begin
          if Temp_Key_Word(.i.) then
               sum := sum * 2 + 1

          else
               sum := sum * 2;
       end;

     Get_Midsquare_Hash_Value := sum;

end;


{-----------------< Midsquare_Method starts from here >------------}

   in
```

```
      Square_Key := sqr (Key_Value);

      Clean_Two_Temp_Bytes;

      Store_Temp_Two_Bytes (Square_Key);

      Hash_Midsquare := Get_Midsquare_Hash_Value;

end;  { Midsquare_Method }
```

```
function Hash_Multiplicative (Key_Value : integer) : integer;

const
    {C_Value = 0.6180339887 or 0.3819660113 from Tanenbaum }
     C_Value = 0.3819660113;

var
     fraction, temp : real;

begin

     temp := C_Value * Key_Value;
     fraction := temp - trunc(temp);

     Hash_Multiplicative := trunc (No_Buckets * fraction);

end;   { Multiplicative_Method }
```

```
function Hash_Radix (Key_Value : integer) : integer;

   onvert assumed Key_Value in base 11 to a number in base 10 }
```

```
const
     NEW_BASE = 11;

var
     Hash_Value : integer;
     quotient : integer;
     i : integer;
     sum : integer;

begin

     sum := 0;

     for i := 5 downto 0 do
       begin
         {quotient := (Key_Value div (10 ** i))}
          quotient := (Key_Value div (power(10,i)));
          sum := sum * NEW_BASE + quotient;
         {Key_Value := Key_Value - quotient * (10 ** i)}
          Key_Value := Key_Value - quotient * (power(10,i));
       end;

     Hash_Radix := sum MOD No_Buckets;

end;
```

```
function Hash_Random (Key_Value : integer) : integer;

const
     RANDOM_DEBUG = true;
var
     Temp_Num : integer;


function Random_Number (seed : integer) : integer;

const
   { MULTIPLIER = 25173
     INCREMENT = 13849
     MODULUS = 65536 }

     MULTIPLIER = 23209;
     INCREMENT = 131071;
     MODULUS = 44497;

   n
```

```
     Random_Number := (MULTIPLIER * seed + INCREMENT) MOD MODULUS;



{-----------------< Random_Method >-----------------------------}

begin
     Temp_Num := Random_Number(Key_Value);

     if RANDOM_DEBUG then
          writeln(' KEY VALUE : ',Key_Value:5,
                  '    RANDOM NUMBER : ', Temp_Num:5,
                  '    BUCKET NUMBER : ',Temp_Num MOD No_Buckets:3);

     Hash_Random := Temp_Num MOD No_Buckets;

end;   {Random_Method}
```

```
   cedure Store_Key_In_Addressed_Bucket (address : integer);

var  Key_Pt : LINK;
     i : integer;

begin

     new(Key_Pt);

     for i := 1 to NUM_IDENT_CHAR do
       begin
          Key_Pt@.Key_Arr(.i.) := Key_Register(.i.);
       end;

     if DEBUG_FLAG then
       begin
          write('   KEY ATTRIBUTE : ');
          for i := 1 to NUM_IDENT_CHAR do
            begin
               write(Key_Register(.i.));
            end;

          writeln('   BUCKET ADDRESS : ',address:3);
          writeln;
       end;

     readln;
```

```
     Key_Pt@.next := Bucket_Arr(.address.);

     Bucket_Arr(.address.) := Key_Pt;

end;




procedure Print_Keys_In_Each_Bucket;

var  i, j : integer;
     pt : LINK;
     count : integer;


begin

     for i := 0 to BUCKET_SIZE do
       begin
          count := 0;
          pt := Bucket_Arr(.i.);

          if DEBUG_FLAG then
            begin
               writeln;
               writeln('----- Bucket Number : ', i : 3,'----');
               writeln;
            end;

          while pt <> nil do
            begin

               if DEBUG_FLAG then
                 begin
                    write('      ');

                    for j := 1 to NUM_IDENT_CHAR do
                      begin
                         write(pt@.Key_Arr(.j.));
                      end;
                    writeln;
                 end;

               pt := pt@.next;
               count := count + 1;

            end;  {while}
```

```
          if DEBUG_FLAG then writeln;

          Count_Arr(.i.) := count;

       end;  {for}

end;




procedure Print_Statistics;

const
     MAX_KEYS = 40;
     EMPTY_STRING =  '                              ';
     LINE_LIMIT = 1;

var  i : integer;
     idx : integer;
     max, sum : integer;

     Count_Keys : array (.0..MAX_KEYS.) of integer;

     variance, Var_Sum : real;
     mean : integer;

     Keys_Over_40 : integer;

begin

     max := 0;
     sum := 0;
     Keys_Over_40 := 0;
     variance := 0;
     Var_Sum := 0;
     mean := MAX_NUM_KEYS div (BUCKET_SIZE + 1);

     for i := 0 to MAX_KEYS do
       begin
          Count_Keys(.i.) := 0;
       end;

     for i := 0 to BUCKET_SIZE do
       begin

          if DEBUG_FLAG then
            begin
               write(' Bucket Number : ',i:3);
               writeln(' contains ', Count_Arr(.i.):3, ' keys.');
               writeln;
```

```
    end;

  sum := sum + Count_Arr(.i.);

  if Count_Arr(.i.) > max then
       max := Count_Arr(.i.);

  if Count_Arr(.i.) = 0 then
       Count_Keys(.0.) := Count_Keys(.0.) + 1

  else if Count_Arr(.i.) = 1 then
       Count_Keys(.1.) := Count_Keys(.1.) + 1

  else if Count_Arr(.i.) = 2 then
       Count_Keys(.2.) := Count_Keys(.2.) + 1

  else if Count_Arr(.i.) = 3 then
       Count_Keys(.3.) := Count_Keys(.3.) + 1

  else if Count_Arr(.i.) = 4 then
       Count_Keys(.4.) := Count_Keys(.4.) + 1

  else if Count_Arr(.i.) = 5 then
       Count_Keys(.5.) := Count_Keys(.5.) + 1

  else if Count_Arr(.i.) = 6 then
       Count_Keys(.6.) := Count_Keys(.6.) + 1

  else if Count_Arr(.i.) = 7 then
       Count_Keys(.7.) := Count_Keys(.7.) + 1

  else if Count_Arr(.i.) = 8 then
       Count_Keys(.8.) := Count_Keys(.8.) + 1

  else if Count_Arr(.i.) = 9 then
       Count_Keys(.9.) := Count_Keys(.9.) + 1

  else if Count_Arr(.i.) = 10 then
       Count_Keys(.10.) := Count_Keys(.10.) + 1

  else if Count_Arr(.i.) = 11 then
       Count_Keys(.11.) := Count_Keys(.11.) + 1

  else if Count_Arr(.i.) = 12 then
       Count_Keys(.12.) := Count_Keys(.12.) + 1

  else if Count_Arr(.i.) = 13 then
       Count_Keys(.13.) := Count_Keys(.13.) + 1

  else if Count_Arr(.i.) = 14 then
       Count_Keys(.14.) := Count_Keys(.14.) + 1

  else if Count_Arr(.i.) = 15 then
       Count_Keys(.15.) := Count_Keys(.15.) + 1
```

```
        else if Count_Arr(.i.) = 16 then
             Count_Keys(.16.) := Count_Keys(.16.) + 1

        else if Count_Arr(.i.) = 17 then
             Count_Keys(.17.) := Count_Keys(.17.) + 1

        else if Count_Arr(.i.) = 18 then
             Count_Keys(.18.) := Count_Keys(.18.) + 1

        else if Count_Arr(.i.) = 19 then
             Count_Keys(.19.) := Count_Keys(.19.) + 1

        else if Count_Arr(.i.) = 20 then
             Count_Keys(.20.) := Count_Keys(.20.) + 1

        else if Count_Arr(.i.) = 21 then
             Count_Keys(.21.) := Count_Keys(.21.) + 1

        else if Count_Arr(.i.) = 22 then
             Count_Keys(.22.) := Count_Keys(.22.) + 1

        else if Count_Arr(.i.) = 23 then
             Count_Keys(.23.) := Count_Keys(.23.) + 1

        else if Count_Arr(.i.) = 24 then
             Count_Keys(.24.) := Count_Keys(.24.) + 1

        else if Count_Arr(.i.) = 25 then
             Count_Keys(.25.) := Count_Keys(.25.) + 1

        else if Count_Arr(.i.) = 26 then
             Count_Keys(.26.) := Count_Keys(.26.) + 1

        else if Count_Arr(.i.) = 27 then
             Count_Keys(.27.) := Count_Keys(.27.) + 1

        else if Count_Arr(.i.) = 28 then
             Count_Keys(.28.) := Count_Keys(.28.) + 1

        else if Count_Arr(.i.) = 29 then
             Count_Keys(.29.) := Count_Keys(.29.) + 1

        else if Count_Arr(.i.) = 30 then
             Count_Keys(.30.) := Count_Keys(.30.) + 1

        else if Count_Arr(.i.) = 31 then
             Count_Keys(.31.) := Count_Keys(.31.) + 1

        else if Count_Arr(.i.) = 32 then
             Count_Keys(.32.) := Count_Keys(.32.) + 1

        else if Count_Arr(.i.) = 33 then
             Count_Keys(.33.) := Count_Keys(.33.) + 1

        else if Count_Arr(.i.) = 34 then
```

```
            Count_Keys(.34.) := Count_Keys(.34.) + 1

        else if Count_Arr(.i.) = 35 then
            Count_Keys(.35.) := Count_Keys(.35.) + 1

        else if Count_Arr(.i.) = 36 then
            Count_Keys(.36.) := Count_Keys(.36.) + 1

        else if Count_Arr(.i.) = 37 then
            Count_Keys(.37.) := Count_Keys(.37.) + 1

        else if Count_Arr(.i.) = 38 then
            Count_Keys(.38.) := Count_Keys(.38.) + 1

        else if Count_Arr(.i.) = 39 then
            Count_Keys(.39.) := Count_Keys(.39.) + 1

        else if Count_Arr(.i.) = 40 then
            Count_Keys(.40.) := Count_Keys(.40.) + 1;


        if (Count_Arr(.i.) > 40) then
            Keys_Over_40 := Keys_Over_40 + 1;

        Var_Sum := Var_Sum + sqr(Count_Arr(.i.) - mean);

      end;  {for}

    variance := Var_Sum / (BUCKET_SIZE + 1);

    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln;
    writeln('--------------------< STATISTICS >---------------------');
    writeln;
    writeln(' THE ',HASH_NAME,' HASH METHOD IS APPLIED TO ');
    writeln;
    writeln(' THE ',DATA_NAME,' DATA.');
    writeln;

    if Hash_Type = FOLD_FOUR_SHIFTED_WORDS then
      begin
```

```
          writeln(' FS(0,2,4,6) => FS(0,10,20,30)');
        { writeln(' FS(0,3,6,1) => FS(0,11,22,25)');
          writeln(' FS(0,4,1,5) => FS(0,12,17,29)');
          writeln(' FS(0,5,2,7) => FS(0,13,18,31)');
          writeln(' FS(0,7,6,5) => FS(0,15,22,29)'); }
          write(' THE SELECTED BITS FROM RESULTING WORD ARE ');

          case FS_ADDR_RANGE of

               1 : writeln('1 THROUGH 8.');

               2 : writeln('9 THROUGH 16.');

               3 : writeln('17 THROUGH 24');

               4 : writeln('25 THROUGH 32');

          end;
      end;

    writeln;
    writeln(' MEAN SQUARE DEVIATION :   ',
              variance:8:2);
    writeln;
    writeln;
    writeln;

    for i := 0 to MAX_KEYS do
      begin
        write(' ',i:2,' KEYS BUCKET =',Count_Keys(.i.):3,' : ');

        for idx := 1 to Count_Keys (.i.) do
          begin
            {if idx mod LINE_LIMIT = 0 then}
                 write ('*');
          end;
        writeln;
      end;


    writeln;
    writeln(' OVER 40 KEYS BUCKET ');
    write('                =', Keys_Over_40:3,' : ');
    for idx := 1 to Keys_Over_40 do
      begin
        if idx mod LINE_LIMIT = 0 then
             write ('*');
      end;
    writeln;
end;



{*************** MAIN PROGRAM STARTS HERE ******************}
```

```
begin

     Initialization;

     {Read Keys one by one converting them to hashed addresses}

     for Key_No := 1 to MAX_NUM_KEYS do

       begin

         Init_For_Do_Loop;

         {Read character by character for a key looking up the
          corresponding ASCII number and convert it to binary number}

         while More_Chars_Left_For_Key do
           begin
              Read_A_Char;
              Look_Up_Char_In_ASCII_Num_Table(index);
              Save_Binary_Num(index);
              Char_No := Char_No + 1;
           end;

         Do_Folding;

         Get_Number_From_Folded_Key (Key_Number);

         case Hash_Type of
              ALGEBRAIC_CODING : Hash_Addr := Hash_Algebraic_Coding
                                              (Key_Number);
              DIGIT_ANALYSIS   : Hash_Addr := Hash_Digit_Analysis;

              DIVISION         : Hash_Addr := Hash_Division
                                              (Key_Number);
              FOLDING          : Hash_Addr := Hash_Folding;

              FOLD_FOUR_SHIFTED_WORDS
                               : Hash_Addr := Hash_FS_4;

              FOLD_SHIFTING    : Hash_Addr := Hash_Shift_Folding;

              MIDSQUARE        : Hash_Addr := Hash_Midsquare
                                              (Key_Number);
              MULTIPLICATIVE   : Hash_Addr := Hash_Multiplicative
                                              (Key_Number);
              RADIX            : Hash_Addr := Hash_Radix
                                              (Key_Number);
              RANDOM           : Hash_Addr := Hash_Random
                                              (Key_Number);
           end;  {case}


         {Using the resulting hashed address, store the input key in
          the corresponding bucket.}
```

```
        Store_Key_In_Addressed_Bucket (Hash_Addr);
      end; {outer for}
    {Print out all the keys in every hashed buckets.}
    Print_Keys_In_Each_Bucket;
    {Print out all the necessary statistics for an analysis.}
    Print_Statistics;
end.
```

FILE: ROMMAKE PASCAL A THE GEORGE WASHINGTON UNIVERSITY COMPUTER CENTER

```
program rommake (input, output);
const
     FIRST_SEED = 6121;
     LINE_END = 10;
var
     Seed_Num, P_Num : integer;
     i, j : integer;

function random (var seed : integer) : integer;

const
     MULTIPLIER = 25173;
     INCREMENT = 13849;
     MODULUS = 65536;

begin
     random := (MULTIPLIER * seed + INCREMENT) mod MODULUS;
end;

begin
     Seed_Num := FIRST_SEED;
     for i := 1 to 16 do
       begin
          for j := 1 to 70 do
            begin
               Seed_Num := random (Seed_Num);
               P_Num := Seed_Num mod 10000;
               write(P_Num:5);
               if (j mod LINE_END) = 0 then
                   writeln;
            end;
          writeln;
       end;
end.
```

```
333333333   8888888         4                EEEEEEEEE  N       NN  DDDDDDDD
333333333  888888888       44                EEEEEEEEE  NN      NN  DDDDDDDDD
      33   88     88      444                EE         NNN     NN  DD      DD
     33    88     88     4444                EE         NNNN    NN  DD      DD
    3333    8888888     44 44      -------   EEEEEE     NN NN   NN  DD      DD
   333333   8888888    44  44      -------   EEEEEE     NN  NN  NN  DD      DD
      33   88     88  444444444              EE         NN   NNNN  DD      DD
33    33   88     88  444444444              EE         NN    NNN  DD      DD
333333333  888888888       44                EEEEEEEEE  NN      NN  DDDDDDDDD
 3333333    8888888        44                EEEEEEEEE  NN       N  DDDDDDDD
```

PRINT COMPLETED AT 21:42:22 FOR USER: RSCSREC DIST: SHIN

```
EEEEEEEEE  N       NN  DDDDDDDD                333333333   8888888         4
EEEEEEEEE  NN      NN  DDDDDDDDD               333333333  888888888       44
EE         NNN     NN  DD      DD              33    33   88     88      444
EE         NNNN    NN  DD      DD                   33   88     88     4444
EEEEEE     NN NN   NN  DD      DD   -------       3333    8888888     44 44
EEEEEE     NN  NN  NN  DD      DD   -------     333333    8888888    44  44
EE         NN   NNNN  DD      DD                    33   88     88  444444444
EE         NN    NNN  DD      DD              33    33   88     88  444444444
EEEEEEEEE  NN      NN  DDDDDDDDD               333333333  888888888       44
EEEEEEEEE  NN       N  DDDDDDDD                 3333333    8888888        44
```

DATE: 29 SEP 91 AT 21:44:22

DEPARTMENT: DFAULT:JDL*

JOB ID: 2409 REPORT NO. 92

FILE ID:

INPUT PROCESSING TIME: 00:01:57

OUTPUT PROCESSING TIME: 00:00:51

REPORT COMPLETION CODE: 4

PAGES TO BIN: 68

PAGES TO TRAY: 0

PAPER PATH HOLES: 0

LINES PRINTED: 3685

ONLINE IDLE (SEC): 96

BLOCKS READ: 0

BLOCKS SKIPPED: 0

RECORDS READ: 3731

DJDE RECORDS READ: 1

MAXIMUM COPY COUNT: 1

OVERPRINTS: 0

COLLATE: YES

SF/MF: MULTI

SIMPLEX/DUPLEX: BOTH

JDE,JDL USED: DFLT,DFAULT

ACCTINFO:

INITIAL FONT LIST: LO112C

INITIAL FORM LIST: -NONE

INITIAL CME LIST: -NONE

XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX
XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX
XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX
XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX EPS XEROX