```
/*===========================================================================*/
/* MODULE                          BM.C                                      */
/*===========================================================================*/
/* FUNCTION    This module implements the Boyer-Moore algorithm
 *             for single substring search.
 *
 * SYSTEM      Standard C (ANSI/ISO).
 *             Tested on PC/MS DOS V.5.0.
 *
 * SEE ALSO    general.h  error.c/.h  bool.c/.h
 *
 * PROGRAMMER  Allan Dystrup
 *
 * COPYRIGHT   (c) Allan Dystrup
 *
 * VERSION     $Header: d:/cwk/kf/bm/RCS/bm.c 1.1 92/10/25 17:02:46
 *             Allan_Dystrup Exp Locker: Allan_Dystrup $
 *             ----------------------------------------------------------------
 *             $Log:    bm.c $
 *             Revision 1.1  92/10/25  17:02:46  Allan_Dystrup
 *             Initial revision
 *
 * REFERENCES
 *             [1] Donald E. Knuth, James H. Morris, Jr. and Vaughan R. Pratt :
 *                 "Fast pattern matching in strings",
 *                 SIAM J. COMPUT. Vol. 6, No. 2, June 1977.
 *             [2] Robert S. Boyer andj. Strother Moore :
 *                 "A fast string searching algorithm",
 *                 Communications of the ACM, October 1977, Vol. 20, No. 10.
 *             [3] Donald M. Sunday :
 *                 "A very fast substring search algorithm",
 *                 Communications of the ACM, August 1990, Vol. 33, No. 8.
 *
 * USAGE
 *
```

```
*========================== SUBSTRING SEARCH ================================
*
* An elementary problem in information retrieval is searching for a specific
* substring or "pattern" (length p) in a larger string of text (length t),
* where pattern and text are composed of characters from a fixed alphabet
* (length a).
*
*                            SF, Straight Forward
* The SF (or "brute force") method consists of first aligning the start of
* the pattern with the start of the text, and then performing a left-right
* scan of the pattern and a char. by char. comparison with the text.
* In case of a mismatch the pattern is shifted right one step, and the text-
* pointer is "backed up" to the new position of the pattern start.
* The time complexity of SF is :
*    setup time : zero
*    worst-case : quadratic in O(p*t)
*                 (ex. find "aaaab" in "aaaaa....aaaaab",
*                 ie. match whole pattern FOR EACH char. in the text)
*    average    : linear in O(t), with const. factor [1...p] near 1.
*                 (mismatch usually expected at first char. comparison,
*                 in which case there is NO backtracking in the text)
*
*
*                       KMP, Knuth-Morris-Pratt - ref. [1]
* The KMP method performs a left-right pattern scan like SF, but eliminates
* "backing up" the text-pointer at character mismatch by first setting up a
* transition graph including each character in the pattern string.
* The graph is represented by a "next[] array" with an index for each pattern
* character giving the new position IN THE PATTERN STRING for backing up THE
* PATTERN POINTER in case of character mismatch, thereby shifting the pattern
* forward by "delta" characters to align the already MATCHED textstring
* (=pattern suffix) with the first leftmost matching pattern prefix/infix.
* The KMP delta shift precalculation thus utilizes any reoccuring substrings
* in the pattern string, and also eliminates backtracking the textpointer
* (if mismatch and no delta shift : start new scan from next textchar)
* KMP time complexity :
*    setup time : linear in O(p)
*    worst-case : linear in O(t), with const. factor = 1
*                 (ex. find "abcde" in "fffff....fffabcde")
*    average    : linear in O(t), with const. factor [1/p...1] near 1.
*                 (mismatch usually expected at first char. comparison,
*                 in which case KMP delta shift IS NOT utilized at all).
*
*
*                       BM, Boyer-Moore - ref. [2]
* The original BM method performs a "REVERSE" pattern scan (ie. right-left,
* unlike SF and KMP), using two predefined shift functions :
*  - delta1 : For each char 'c' in the alphabet, find the rightmost position
*             (i) of c in the pattern (i=0 if c not in pat). Delta1(c) = p-i.
*  - delta2 : Analogous to the KMP delta function for reoccuring substrings
*             in the pattern : for each pattern suffix find first leftward
*             reoccurence of it (infix or prefix) in the pattern.
* In case of mismatch BM uses the larger value of delta1 and delta2 to shift
* the pattern forward :
*  - either delta1(c) : to the rightmost matching char. 'c' in the pattern,
*    however if delta1(c) is to the right of the mismatch-position, then
*    the pattern is shifted right by 1,
*  - or delta2(c) : to the first leftward pattern prefix/infix matching the
*    already matched textstring, cf. KMP algorithm.
* and restarting the scan from the pattern end,
* BM time complexity :
*    setup time : linear in O(a+p)
*    worst-case : - using delta1 alone... : quadratic in O(p*t)
*                 - using delta1 & delta2 : linear in O(t)
*    average    : sub-linear in O(t/p), w. const. factor [1/p...1] near 1/p.
*                 (mismatch usually expected at first char. comparison,
*                 in which case BM delta shifts ARE fully utilized).
*
```

```
*========================== CHOICE OF ALGORITHM ============================
*
* The only advantage of SF is that it doesn't require any setup time. For
* very simple search situations (small and heterogenous pattern and text)
* SF may be worth considering, but as a general basis for bibliographic
* search SF is totally inadequate.
*
* KMP has an advantage when "backing up" the textpointer is inconvenient, or
* when searching for a highly self-repetitive pattern in a self-repetitive
* text. Under these circumstances the linear worst-case search time makes it
* more attractive than SF, but in most actual applications (as for instance
* bibliographic search for a single word) KMP is not likely to be signifi-
* cantly faster than SF.
*
* BM is our candidate for a bibliographic one-word search algortihm. The
* "sublinear" search time of delta1 makes BM more than three times as fast
* as SF or KMP in most practical situations, and with the addition of delta2
* we get an algorithm with the same desirable worst-case performance as KMP.
* With a small modification of the original BM algorithm it is even possible
* to get a simpler and faster implementation that does not depend on a
* specific pattern scan order and thus may be coded to avoid backtracking!
* The advantage of BM increases with the length of the search pattern (p) and
* of the alphabet (a). A bibliographic search for a long keyword or phrase
* in a text over the full (255 char) alphabet is an ideal application for BM.
*
* BM however has one drawback : it can not easily be extended to search for
* multiple patterns in a text. For this type of application it is better to
* use the technique applied by KMP : building a transition graph (also
* called a "Finite State Automaton") to perform the search. This is precisely
* the scope of the Aho-Corasic algorithm for multiple string search (see
* module ac.c).
*
*===================== AN IMPROVED BM IMPLEMENTATION =======================
*
* An improved implementation of the BM algorithm, - cf. ref. [3]
*
* Delta1.
* First note that the pattern string always shifts right by at least one
* char. Hence the char. in the text string IMMEDIATELY PAST THE END of the
* pattern string must be involved for testing at the next pattern position.
* Thus delta1 can be computed for the whole alphabet to be the index of the
* first leftward occurence of each char. FROM THE END of the pattern string.
* This slightly modified delta1 has the following big advantage :
* the shift defined by the new delta1(c) is an ABSOLUTE SHIFT relating to
* the text character immediately after the pattern (always >= 1), hence
*  (1) delta1 MAY BE USED "STAND ALONE" to code a "quick and clean" BM
*      search (as opposed to using the original BM delta1 SHIFT, which was
*      relative to the mismatch position, requiring a test and shift by
*      max(1, delta2) when the pos. of mismatch was to the left of delta1.
*  (2) delta1 does NOT DEPEND ON A SPECIFIC PATTERN SCAN ORDER (as opposed
*      to the original BM, which required a strict right-left "backtracking"
*      textpointer)
*
* Delta2
* For any specific order of scanning the pattern string, one can define a
* delta2 shift similar to the KMP delta (left-right scan) or the BM delta2
* (right-left scan), thus avoiding the undesirable "quadratic worst case
* behaviour" of a BM delta1 stand-alone solution (but at the expense of a
* substantially larger setup time).
*
* Delta1 vs. Delta2.
* Empirical evidence indicates a "break even" between delta1 standalone and
* delta1+delta2 at a pattern length of about 10-15 characters (in biblio-
* graphic search), - ie. a search for a short keyword is best performed by
* a delta1 standalone algortihm, while searching for a longer word or phrase
* often justifies the initial setup time for delta2.
*
*/
```

```
/*============================================================================*/
/*                              Includes                                      */
/*============================================================================*/
/* Standard ANSI/ISO header files */
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <stdlib.h>

/* Module header file */
#define _BM_ALLOC              /* bm.c (Boyer-Moore) header file */
#include "bm.h"



/*============================================================================*/
/*                           Typedefs                                         */
/*============================================================================*/
typedef struct patScanElem {  /* Struct. of element in scan ordered pattern */
    int       loc;             /* Location of the character c in the pattern */
    BYTE      c;               /* Value of character in pattern at location  */
} sPAT;



/*============================================================================*/
/*                           Global data objects                              */
/*============================================================================*/
PRIVATE BYTE *pzPat;           /* Ptr. to pattern string */
PRIVATE int iPatLen;           /* Length of pattern string */
PRIVATE int *piDelta1;         /* Ptr. to Delta1 shift array for all chars  */
PRIVATE float *pfScanOrd;     /* Ptr. to scan priorrity array for chars    */
PRIVATE sPAT *psOrdPat;       /* Ptr. to scan orederd elements of pattern  */
PRIVATE int *piDelta2;         /* Ptr. to Delta2 shift array for pat. chars */
PRIVATE enum scanType eScanOrder = eScanUD;     /* Initially undefined */



/*============================================================================*/
/*                         Function Prototypes                                */
/*============================================================================*/
PRIVATE int iShCompMS(sPAT * psPat1, sPAT * psPat2);
PRIVATE int iShCompOM(sPAT * psPat1, sPAT * psPat2);
PRIVATE int iShFind(int iPatLoc, int iLShift);


/*============================================================================*/
/*                           Trace macros                                     */
/*============================================================================*/
/* The option of using an arbitrary scan order in the improved BM algorithm
 * requires a rather convoluted procedure for setting up the piDelta2-shifts.
 * I have defined the following macros to allow tracing and verification of
 * the BM table-setup. The notation of the macro parameters uses the follo-
 * wing "shorthand":
 *    _P   :  pointer to start of zero-terminated pattern string
 *    _p   :  pointer for scanning the pattern string
 *    _s   :  pointer to array of scan order priority for characters
 *    _OP :  pointer to start of list of ordered structures 'patScanElem'
 *    _o   :  pointer for scanning the _OP list
 *    _arr :  array
 *    _len :  length of array or string
 *    _i   :  scratch index variable
 */
int       _i;

#define DUMPD1(_arr,_len)                                                   \
D(     printf("\n\nDUMP of BM table "#_arr"[] ...\n");                      \
```

```
       for (_i=0; _i < _len; _i++)                                          \
          printf(" %x[%c]:%04.1f%c",                                        \
                 _i, _i, (float) _arr[_i], !(_i%6) ? '\n' : ' '); );

#define DUMPMS(_typ,_o,_p,_s,_len)                                          \
D(     printf("\n\nDUMP of "#_typ" scan ordered string "#_o" ...\n");      \
       for(_i=0; _i < _len; _i++)                                           \
          printf("\t"#_o"[%02d]="#_p"[%02d]=%c  "#_s"[%02d]=%04.1f\n",     \
                 _i, _o[_i].loc, _o[_i].c, _i, _s[_o[_i].loc] );           );

#define DUMPOM(_typ,_o,_p,_s,_len)                                          \
D(     printf("\n\nDUMP of "#_typ" scan ordered string "#_o" ...\n");      \
       for(_i=0; _i < _len; _i++)                                           \
          printf("\t"#_o"[%02d]="#_p"[%02d]=%c  "#_s"[%c]=%04.1f\n",       \
                 _i, _o[_i].loc, _o[_i].c, _o[_i].c, _s[_o[_i].c] );        );

#define DUMPD2(_typ,_arr,_len)                                              \
D(     printf("\nDUMP of "#_typ" BM table "#_arr"[] ...\n");               \
       for(_i=0; _i < _len; _i++)                                           \
          printf("\t"#_arr"[%02d]=%02d\n", _i, _arr[_i]);                   );

#define DUMPSH(_typ,_len,_OP,_o,_P,_j)                                       \
D(     printf("\t   iLShift=%d OP[%02d]=P[%02d]=%c P[%02d]=%c\t"#_typ"\n", \
              _len, _o-_OP, _o->loc, _o->c, _j, (_j>=0 ? _P[_j] : '*') ); );
```

```
#ifdef MAIN
/*=========================================================================*/
/*                        Main (Module Testdriver)                         */
/*=========================================================================*/
#define MAX_LINE        256

int
main(int iArgc, BYTE ** ppzArgv)
/* Input is a Boolean search expression "str" provided on the command line.
 * The Boolean expression is parsed by function pzParse (in bool.c) to :
 *  - a suffix representation of the Boolean logic tree (pzPostfix).
 *  - a set of keywords K = {k1, k2, ... ,kn} to search for (symtable[]).
 * In module bm.c we perform a simple search for a keyword or phrase using
 * the improved Boyer-Moore algorithm. The use of pzParse to scan/parse the
 * single keyword or phrase is thus somewhat "overkill" (we don't actually
 * use pzPostfix in this context), but the driver may easily be extended to
 * run other more advanced search algorithms as for instance the Aho-Corasic
 * search for multiple substrings in a text.
 */
{
    BYTE     *pbArg;                   /* Char in argument string */
    int       eScanOrder = eScanUD;  /* Type of pattern scan order */
    FILE     *fileInpFd;               /* File descriptor for input file */
    BYTE      pzInpBuffer[MAX_LINE];/* String buffer for input file */
    BYTE     *pzPostfix;               /* Ptr. to postfix boolean logic */
    BYTE     *pbNL;                    /* Prt. to newline character */
    FLAG      fMatch;                  /* Flag for match Pattern/Text */
    int       i;                       /* Scratch index variable */

    /* 1: Parse command line option */
    while (--iArgc && (*++ppzArgv)[0] == '-')
            for (pbArg = ppzArgv[0] + 1; *pbArg != '\0'; pbArg++)
                switch (toupper(*pbArg)) {
                        case 'M':     /* Force a Maximal Shift search */
                            eScanOrder = eScanMS;
                            break;
                        case 'O':     /* Force an Optimal Mismatch search */
                            eScanOrder = eScanOM;
                            break;
                        case 'Q':     /* Force a Quick Search */
                            eScanOrder = eScanQS;
                            break;
                        default:      /* Switch must be M, O, or Q */
                            printf("\n\n--- allowed switches : [MOQ] ! ---\n\n");
                            exit(EXIT_FAILURE);
                }


    /* 2: Parse Bool search expression to postfix string and keyword list */
    /* -- (Use only first entry in keyword list : symtable[1] for search) */
    D(printf("\n\n=============== BUILDING NEW BM ================\n\n"));
    D(puts("PARSE INPUT STRING ..."));
    D(printf("\t%s\n", *ppzArgv));
    pzPostfix = pzParse(*ppzArgv++);
    iArgc--;

    for (i = 1; symtable[i].pzLexptr; i++)
       /* count symbol table entries */;
    if (--i > 1 || !symtable[i].pzLexptr) {
            printf("\n\n--- exactly one search phrase, please! ---\n\n");
            exit(EXIT_FAILURE);
    }


    /* 3: Initialize BM-search */
    if (eScanOrder == eScanUD)
            eScanOrder = (strlen(symtable[1].pzLexptr) <= 15 ? eScanQS : eScanMS);
```

```
    vBuildBM(eScanOrder, symtable);


    /* 3: Run BM-search */
    fileInpFd = (iArgc ? fopen(*ppzArgv, "r") : stdin);
    while (fgets(pzInpBuffer, MAX_LINE, fileInpFd)) {

            if (pbNL = strchr(pzInpBuffer, '\n'))/* Replace newline w. string term. */
                *pbNL = '\0';

            if (fMatch = fRunBM(pzInpBuffer)) {
                D(printf("\nOUTPUT ...\n>\t"));
                puts(pzInpBuffer);
            }
    }


    /* 3: Terminate BM search */
    /* Return status to the parent process: 1 if match, 0 if none */
    vDelBM();
    return (fMatch ? 1 : 0);
}
#endif /* #ifdef MAIN */
```

```
/*============================================================================*/
/*                            vBuildBM                                        */
/*============================================================================*/
void
vBuildBM(enum scanType type, struct entry * symtab)
{
/* Construct the BM delta shift table(s) from pattern string pzPat.
 *
 *                      1 Delta1 table : piDelta1[]
 * The delta1 table-setup requires two steps :
 *
 * 1.1 First initialize the table to (iPatLen+1) for all characters in the
 *     alphabet.
 *
 * 1.2 Then reset the table entries for each char in the pattern to the
 *     rightmost position in the pattern of that char.
 *
 *                      2 Delta2 table : piDelta2[]
 * The delta2 shift table piDelta2[] is built in two steps :
 *
 * 2.1 First build the ordered pattern psOrdPat from any ordering spe-
 *     cification; The chosen scan-order (set up in array pfScanOrd) may be :
 *      - forward, giving a piDelta2[] = KMP delta
 *      - reverse, giving a piDelta2[] = original BM delta2
 *      - maximal shift (MS), using the max. left shift to sort psOrdPat
 *      - optimal mismatch (OM), using char-frequency to sort psOrdPat
 *      - any other order suitable for the actual implementation
 *     For bibliographic search we choose to implement the MS scan order.
 *
 * 2.2 Then construct piDelta2[] in two steps, using the following procedure:
 *     If mismatch at scan order position i, ie  psOrdPat[i].c !=
 *     text[ScanStart+psOrdPat[i].loc], then shift ('overlay') the pattern
 *     left to position 'iLShift', so
 *     2.2.1  psOrdPat[0].c ... psOrdPat[i-1].c ALL MATCH aligned characters
 *            in the shifted pat; - iLShift is the max. value in the range
 *            0...psOrdPat[0].loc (= the minimal leftshift to match i chars)
 *     2.2.2  psOrdPat[i].c DOES NOT match the aligned character
 *            pzPat[ psOrdPat[i].loc - iLShift ];  - this may require
 *            repeated left shifting of the pattern.
 *     Then delta2 table piDelta2[i] = (psOrdPat[0].loc - iLShift),
 *     ie. the required right shift to align reoccuring characters in the
 *     pattern (cf. 2.2.1) excluding the mismatched character (cf. 2.2.2)
 *     with already matched characters in the text.
 */
   BYTE     *p;               /* Ptr. to scan through pattern string */
   sPAT     *o;               /* Ptr. to scan through ordered pattern */
   int       iLShift;         /* Left shift for matching reoccuring chars */
   int       i, j;            /* Scratch index variables */

   /* Define global data objects of module */
   pzPat = symtab[1].pzLexptr;/* Pattern, -scanned by module bool.c */
   iPatLen = strlen((char *) pzPat);           /* Pattern length */
   eScanOrder = type;         /* Requested type of pattern scan order */
   pfScanOrd = NULL;          /* Defined in comp. functions below */


   /* -------------------- Build delta1 table ---------------------------- */

   /* Allocate the Delta1 shift table piDelta1[] */
   if (!(piDelta1 = (int *) malloc(sizeof(int) * (UCHAR_MAX + 1))))
           vError(EMEM005, "vBuildBM");

   /* 1.1 Initialize the piDelta1[] table for all characters in the alphabet */
   for (i = 0; i <= UCHAR_MAX; i++)/* mismatch -> 1. char after pattern */
           piDelta1[i] = iPatLen + 1;

   /* 1.2 Reset piDelta1[] entries for each of the pattern characters */
   for (p = pzPat; *p; p++)   /* match -> last position in pattern */
```

```
        piDelta1[*p] = iPatLen - (p - pzPat);

DUMPD1(piDelta1, UCHAR_MAX);


/* -------------------- Build delta2 table ---------------------------- */

if (eScanOrder != eScanQS) {

        /* Alloc. structures for psOrdPat and Delta2 shift table piDelta2[] */
        if (!(psOrdPat = (sPAT *) malloc(sizeof(sPAT) * (iPatLen + 1)))
            || !(piDelta2 = (int *) malloc(sizeof(int) * (iPatLen + 1))))
            vError(EMEM006, "vBuildBM");


        /* 2.1 */
        /* Build the scan-ordered pattern from the pfScanOrd array    */
        /* using qsort to sort the pattern chars after max left shift */
        for (i = 0, p = pzPat, o = psOrdPat; i <= iPatLen; ++i, ++p, ++o) {
            o->loc = i;
            o->c = *p;                  /* \0 for i == iPatLen */
        }

        switch (eScanOrder) {
            case eScanMS:     /* Maximal Shift search */
                    qsort(psOrdPat, iPatLen, sizeof(sPAT), iShCompMS);
                    DUMPMS(MS - sorted, psOrdPat, pzPat, pfScanOrd, iPatLen);
                    break;
            case eScanOM:     /* Optimal Msimatch search */
                    qsort(psOrdPat, iPatLen, sizeof(sPAT), iShCompOM);
                    DUMPOM(OM - sorted, psOrdPat, pzPat, pfScanOrd, iPatLen);
                    break;
                    /* case tNEW : insert call to new scan order sorting here & */
                    /* ----------- place code of the scan order comp.func below */
            default:
                    /* Must be a defined type of search ! */
                    vError(EARG002, "vBuildBM");
        }


        /* 2.2.1 */
        /* First init piDelta2[] with the minimum matching left shift    */
        /* so psOrdPat[0].c ... psOrdPat[i-1].c match aligned characters */
        for (piDelta2[0] = iLShift = i = 1; i < iPatLen; ++i) {
            iLShift = iShFind(i, iLShift);
            piDelta2[i] = iLShift;
        }
        DUMPD2(initialized, piDelta2, iPatLen);

        /* 2.2.2 */
        /* Then for each piDelta2[i] check that :  psOrdPat[i].c !=        */
        /* pzPat[psOrdPat[i].loc-iLShift]; - if not, repeat search for a   */
        /* matching left shift until this condition.                       */
        for (i = 0; i < iPatLen; ++i) {
            iLShift = piDelta2[i];          /* get initial matching shift */
            o = psOrdPat + i;
            D(printf("\n   piDelta2[%02d],INITIAL LSHIFT : %d\n", i, iLShift));

            while (iLShift < iPatLen) {

                    /* Require left shift INSIDE pattern! */
                    if ((j = (psOrdPat[i].loc - iLShift)) < 0) {
                        DUMPSH(Ignore, iLShift, psOrdPat, o, pzPat, j);
                        break;
                    }

                    /* Require current char NOT MATCH after shift! */
                    if (psOrdPat[i].c != pzPat[j]) {
```

```
                            DUMPSH(OK, iLShift, psOrdPat, o, pzPat, j);
                            break;
                        }

                        /* If match, scan iteratively for next matching shift */
                        DUMPSH(RESET, iLShift, psOrdPat, o, pzPat, j);
                        ++iLShift;
                        iLShift = iShFind(i, iLShift);
                    }
                    piDelta2[i] = iLShift;          /* set final shift */

                }
                DUMPD2(corrected, piDelta2, iPatLen);
    }                                       /* end if eScanOrder != eScanQS */
}



/*---------------------------------------------------------------------------*/
/*                                    iShCompMS                              */
/*---------------------------------------------------------------------------*/
int
iShCompMS(sPAT * psPat1, sPAT * psPat2)
{
/* Compare 2 elements of the scan ordered pattern according to a precalcula-
 * ted table 'pfScanOrd' of each element-character's maximal left shift.
 */
    int       dsh;              /* Delta shift value */
    int       i, j;             /* Scratch index variables */

    /* First call: Build the pfScanOrd array for each char in pattern pzPat */
    /* of the minimal left shift to the last position of the char in pzPat  */
    /* (or to the 1. character of pzPat if no match).                       */
    if (pfScanOrd == NULL) {
            if (!(pfScanOrd = (float *) malloc(sizeof(float) * (iPatLen + 1))))
                vError(EMEM007, "iShCompMS");
            for (i = 0; i < iPatLen; ++i) {
                for (j = i - 1; j >= 0 && pzPat[j] != pzPat[i]; --j);
                pfScanOrd[i] = (float) (i - j);
            }
            DUMPMS(unsorted, psOrdPat, pzPat, pfScanOrd, iPatLen);
    }

    /* ascending order */
    dsh = (int) (pfScanOrd[psPat2->loc] - pfScanOrd[psPat1->loc]);
    return (dsh ? dsh : psPat2->loc - psPat1->loc);
}



/*---------------------------------------------------------------------------*/
/*                                    iShCompOM                              */
/*---------------------------------------------------------------------------*/
int
iShCompOM(sPAT * psPat1, sPAT * psPat2)
{
/* Compare 2 elements of the scan ordered pattern according to a precalculated
 * table 'pfScanOrd' of each element-character's frequency in the alphabet. */

    int       i;
    float     fq;

    /* First call: Build the pfScanOrd array for each char in the alphabet */
    /* of freq. of the chars in the targetlanguage based on the alphabet.  */
    if (pfScanOrd == NULL) {
            if (!(pfScanOrd = (float *) malloc(sizeof(float) * (UCHAR_MAX + 1))))
                vError(EMEM008, "iShCompOM");
            for (i = 0; i <= UCHAR_MAX; ++i)
```

```
                pfScanOrd[i] = 0.1f;

            /* English text alphabet frequency */
            pfScanOrd['e'] = 11.1f;
            pfScanOrd['a'] = 8.9f;
            pfScanOrd['i'] = 7.8f;
            pfScanOrd['r'] = 7.4f;
            pfScanOrd['t'] = 7.1f;
            pfScanOrd['o'] = 6.9f;
            pfScanOrd['n'] = 6.8f;
            pfScanOrd['s'] = 5.6f;
            pfScanOrd['l'] = 5.5f;
            pfScanOrd['c'] = 4.5f;
            pfScanOrd['u'] = 3.6f;
            pfScanOrd['m'] = 3.2f;
            pfScanOrd['d'] = 3.2f;
            pfScanOrd['p'] = 3.1f;
            pfScanOrd['h'] = 2.9f;
            pfScanOrd['g'] = 2.4f;
            pfScanOrd['b'] = 2.3f;
            pfScanOrd['y'] = 2.0f;
            pfScanOrd['f'] = 1.5f;
            pfScanOrd['w'] = 1.1f;
            pfScanOrd['k'] = 1.1f;
            pfScanOrd['v'] = 1.0f;
            pfScanOrd['x'] = 0.3f;
            pfScanOrd['j'] = 0.2f;
            pfScanOrd['z'] = 0.2f;
            pfScanOrd['q'] = 0.2f;

            /* Danish text alphabet frequency */
            /* - to be defined -              */

            DUMPD1(pfScanOrd, UCHAR_MAX);
            DUMPOM(unsorted, psOrdPat, pzPat, pfScanOrd, iPatLen);
    }

    /* descending order */
    fq = pfScanOrd[psPat1->c] - pfScanOrd[psPat2->c];
    return (int) (fq ? (fq > 0 ? 1 : -1) : psPat2->loc - psPat1->loc);
}



/*--------------------------------------------------------------------------*/
/*                                    iShFind                               */
/*--------------------------------------------------------------------------*/
int
iShFind(int i, int iLShift)
{
/* Return value of the next valid leftward shift 'iLShift' of the first 'i'
 * chars in psOrdPat, such that all chars from psOrdPat[i-1].c to
 * psOrdPat[0].c in the scan ordered pattern matches the chars in the pattern
 * pzPat after a left shift of 'iLShift' (ie. we match reoccuring chars in the
 * pattern at a distance of iLShift positions).
 */
    sPAT     *o;               /* Ptr. to scan ordered pattern */
    int       j;               /* Scratch index variable */

    D(printf("\n\t   MATCHSHIFT psOrdPat[%02d...00] :\n", i - 1));
    for (; iLShift < iPatLen; ++iLShift) {

            for (o = psOrdPat + i - 1; o >= psOrdPat; o--) {
                /* in range 0...psOrdPat[j].loc */
                if ((j = (o->loc - iLShift)) < 0) {
                        DUMPSH(Ignore, iLShift, psOrdPat, o, pzPat, j);
                        continue;
                }
```

```
                if (o->c != pzPat[j]) {          /* mismatch at position j */
                        DUMPSH(Mismatch, iLShift, psOrdPat, o, pzPat, j);
                        break;
                }

                DUMPSH(MATCH, iLShift, psOrdPat, o, pzPat, j); /* matched one more */
            }

            if (o < psOrdPat)       /* matched all psOrdPat[0]...psOrdPat[i] */
                break;
    }

    D(printf("\t   LSHIFT : %d\n", iLShift));
    return iLShift;
}
```

```
/*=========================================================================*/
/*                              vRunBM                                     */
/*=========================================================================*/
FLAG
fRunBM(BYTE * T)
{
/*                  psOrdPat<----o---->psOrdPat+iPatLen
 * Scan ordered pattern :  [xxxxxxxxxxx]
 *
 *                     pzPat<-----p--->pzPat+iPatLen
 * Pattern :               [xxxxxxxxxxx]
 *
 * Text    :   [yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy]
 *              T          S<-----t--->S+iPatLen               T+Tlen
 */
    BYTE     *p;               /* Ptr. to scan through pattern */
    sPAT     *o;               /* Ptr. to scan ordered pattern */
    BYTE     *S = T;           /* Ptr to shift the pattern through text */
    BYTE     *t;               /* Ptr. to scan through text */
    int       Tlen = strlen((char *) T); /* Length of text */
    int       iD1;              /* Delta1 table shift value. */
    int       iD2;              /* Delta2 table shift value. */


    switch (eScanOrder) {

            case eScanQS:
                /* --------- Perform left-right Quick Search ---------------- */

                D(puts("\nTRACE of BM Quick Search ...\n"));

                /* while enough text is still left ... */
                while (S + iPatLen <= T + Tlen) {

                        D(printf("Pat:\t%s\n", pzPat));
                        D(printf("Txt:\t%s\n", S));

                        /* scan through pattern string while match in text */
                        D(printf("\tMATCH:["));
                        for (p = pzPat, t = S; *p && *p == *t; p++, t++)
                            D(printf("%c", *t));

                        /* test for end-of-string */
                        if (*p) {      /* no : MISMATCH */
                            D(printf("]\n\tFAIL.:[%c<>%c]", *p, *t));
                            iD1 = piDelta1[*(S + iPatLen)]; /* get delta1 */
                            D(printf("\n\tSHIFT:piDelta1[%c]=%03d\n", *(S + iPatLen),
iD1));
                            S += iD1;  /* shift pattern */
                        }
                        else {              /* yes : MATCH */
                            D(puts("]\n\tCOMPLETE!\n"));
                            return (TRUE);      /* might return position (1+S-T) */
                        }
                }
                return (FALSE);    /* no match in total text, return FALSE(0) */
                break;          /* unreachable,- defensive programming */

            case eScanMS:
            case eScanOM:
                /* ------- Perform arbitrary scan order search  -------------- */

                D(puts("\nTRACE of BM ScanOrder Search ...\n"));

                /* while enough text is still left ... */
                while (S + iPatLen <= T + Tlen) {

                        D(printf("Pat:\t%s\n", pzPat));
```

```
                        D(printf("Txt:\t%s\n", S));

                        /* scan through ordered pattern while match in text */
                        D(printf("\tMATCH:["));
                        for (o = psOrdPat; o->c && o->c == *(S + o->loc); o++)
                            D(printf("%c", *(S + o->loc)));

                        /* test for end-of-string */
                        if (o->c != 0) {        /* no : MISMATCH */
                            D(printf("]\n\tFAIL.:pos.%03d [%c<>%c]\n",
                                        o->loc, o->c, *(S + o->loc)));
                            iD1 = piDelta1[*(S + iPatLen)]; /* get delta1 */
                            iD2 = piDelta2[o - psOrdPat];   /* get delta2 */
                            D(printf("\tSHIFT:piDelta1[%c]  =%03d\n", *(S + iPatLen),
iD1));
                            D(printf("\t      piDelta2[%03d]=%03d\n", o - psOrdPat, iD2));
                            D(printf("\t      MAX     =%03d\n", (iD1 > iD2 ? iD1 : iD2)));
                            S += (iD1 > iD2 ? iD1 : iD2);   /* use max for pattern shift
*/
                        }
                        else {              /* yes : MATCH */
                            D(puts("]\n\tCOMPLETE!\n"));
                            return (TRUE);      /* might return position (1+S-T) */
                        }
                    }
                    return (FALSE);    /* no match in total text, return FALSE(0) */
                    break;          /* unreachable,- defensive programming */

                default:            /* Must be a defined type of search ! */
                    vError(EARG003, "vRunBM");
        }
}
```

```
/*============================================================================*/
/*                              vDelBM                                        */
/*============================================================================*/
void
vDelBM(void)
{
    switch (eScanOrder) {
       case eScanMS:
       case eScanOM:
                free(piDelta2);
                free(psOrdPat);
                free(pfScanOrd);
                /* !fall through! */
            case eScanQS:
                free(piDelta1);
                break;
            default:           /* Must be a defined type of search ! */
                vError(EARG004, "vDelBM");
    }

    D(puts("\nData structures deallocated ..."));
}


/* END of module BM.C                                                         */
/*============================================================================*/
```