



```
0,000,490 i_m_not_gonna
0,000,447 i_need_you_to
0,000,436 what_do_you_mean
0,000,396 i_didn_t_know
0,000,385 what_do_you_want
0,000,384 are_you_doing_here
0,000,378 we_don_t_have
0,000,376 i_m_so_sorry
0,000,368 that_s_what_i
0,000,359 what_s_wrong_with
0,000,357 i_don_t_wanna
0,000,356 i_m_not_sure
0,000,350 don_t_have_a
0,000,348 i_don_t_need
0,000,339 you_re_going_to
0,000,333 i_m_gonna_go
0,000,331 i_think_it_s
0,000,317 don_t_know_how
0,000,312 what_s_why_i
0,000,308 i_m_trying_to
0,000,307 you_re_not_gonna
0,000,306 i_ll_see_you
0,000,302 i_don_t_even
0,000,295 get_out_of_here
0,000,292 i_ll_tell_you
```

```
_FNVL1_Hash_Jesteress PROC
    .i
    $LN6@FNVL1_Hash@8:
    mov edi, DWORD PTR [eax]
    rol edi, 5
    xor edi, DWORD PTR [eax+4]
    sub edx, 8
    xor ecx, edi
    imul ecx, 709607
    add eax, 8
    dec esi
    jne SHORT $LN6@FNVL1_Hash@8
    pop edi
    $LN4@FNVL1_Hash@8:
    test dl, 4
    je SHORT $LN3@FNVL1_Hash@8
    xor ecx, DWORD PTR [eax]
    imul ecx, 709607
    add eax, 4
    $LN3@FNVL1_Hash@8:
    test dl, 2
    je SHORT $LN2@FNVL1_Hash@8
    movzx esi, WORD PTR [eax]
    xor ecx, esi
    imul ecx, 709607
    add eax, 2
    $LN2@FNVL1_Hash@8:
    pop esi
    test dl, 1
    je SHORT $LN1@FNVL1_Hash@8
    movsx eax, BYTE PTR [eax]
    xor ecx, eax
    imul ecx, 709607
    $LN1@FNVL1_Hash@8:
    .i
    _FNVL1_Hash_Jesteress ENDP
```

LEPRECHAUN X - LETON

A 32BIT/64BIT LINUX/WINDOWS ENGLISH X-GRAM WORDLIST RIPPER, REVISION 16FIXFIX

Free download at www.sanmayce.com — in multi-pass mode IT can rip the whole written English using a simple net-book.

```

0001 // This is source of Leprechaun revision 16FIXFIX: Leprechaun_x-leton.c, copyleft Sanmayce, 2013-Mar-31.
0002 // Grrr... Next two variables were not nullified between passes:
0003 // 16fixfix [
0004 // PLE_words_INITflag = 0;
0005 // PLE_words = 0;
0006 // 16fixfix ]
0007
0008 // This is source of Leprechaun revision 16FIX: Leprechaun_x-leton.c, copyleft Sanmayce, 2012-Dec-16.
0009 // How embarrassing! A stupid bug was fixed, namely one missed 'if ( REUSE == 0 ) {}' holding the TRAVERSE segment - this segment nullifies
    LEAF addresses thus making W/w unable to retrace.
0010
0011 // This is source of Leprechaun revision 16: Leprechaun_x-leton.c, copyleft Sanmayce, 2012-Dec-13.
0012 // The new feature is the ability to reuse the external hash-tree structure.
0013 // The option is W/w similar to Z/z. This way the latency i.e. the response time is <1s.
0014
0015 // This is source of Leprechaun revision 15FIXFIX+: Leprechaun_x-leton.c, copyleft Sanmayce, 2012-Dec-11.
0016 // The new feature is the ability to command Leprechaun (from inside the list file with 2 metacommands) to enter/exit INSERT mode.
0017 // This allows to control whether new (to current hash-tree structure) x-grams are to be counted [and] INSERTed.
0018
0019 // Usage:
0020 // E:\_Gamera_r15_12348>type ON.txt
0021 // Leprechaun says x-gram inserting disabled for next files: ON
0022 //
0023 // E:\_Gamera_r15_12348>type OFF.txt
0024 // Leprechaun says x-gram inserting disabled for next files: OFF
0025 //
0026 // E:\_Gamera_r15_12348>dir Your_textual_folders\b\s/a-d>go.lst
0027 // E:\_Gamera_r15_12348>copy go.lst+on.txt+_Gamera.tar.3.sorted.4andabove.lst MetaLep.lst /b
0028 // E:\_Gamera_r15_12348>type MetaLep.lst
0029 // E:\_Gamera_r15_12348>Your_textual_folders\Example.txt
0030 // Leprechaun says x-gram inserting disabled for next files: ON
0031 // _Gamera.tar.3.sorted.4andabove.txt
0032 //
0033 // E:\_Gamera_r15_12348>Leprechaun_x-leton_META_32bit_03_01p.exe MetaLep.lst MetaLep.3.wrd 1234567 Y
0034
0035 // All lines new to r15FIXFIX are with commented part //15FIXFIX+
0036
0037 /*
0038 This is source of Leprechaun revision 15FIXFIX: Leprechaun_x-leton.c, copyleft Sanmayce, 2011-Dec-14. 2011-Mar-07: Fixed a small command line
    parsing bug.
0039
0040 The 15FIXFIX differs from 15fix with:
0041 [a bug fixed(REALLY FIXED!): Fixed a nasty bug causing very restrictive way of forming x-grams.]
0042 The 15fix differs from 15 with:
0043 [a bug fixed: division by zero when finishing-starting time is under 1 second
0044 Fixed a nasty bug causing very restrictive way of forming x-grams.]
0045 The 15 differs from 14++++FIXFIX with:
0046 [
0047 Only some more stats at the end.
0048 ]
0049 The 14++++FIXFIX differs from 14++++FIX with:
0050 [
0051 Bugs in LOG stats in r.14++++FIX:
0052 Not nullified variables during passes - must be nullified.
0053 Number Of Trees(GREATER THE BETTER): 195,939
0054 Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 654,428
0055 Total Attempts to Find/Put WORDs into B-trees order 3: 39,042,828
0056 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 3
0057 ]
0058 The 14++++FIX differs from 14+++ with:
0059 [
0060 1)Fixed occurrences bug due to not NULLifying the field housing the occurrences, a nasty thing: all the revisions 14??? were buggy, how
    stupid from my side, grrrr.
0061 2)Ability to rip in passes:
0062 #define HashChunkSizeNBITS 26 // Defines the number of passes. Should be smaller or equal to HashNBITS. If HashNBITS == HashChunkSizeNBITS
    then 2^(HashNBITS-HashChunkSizeNBITS)=2^0=1 passe(s).
0063 ]
0064 The 14+++ differs from 14++ with:
0065 [
0066 //Only one must be uncommented:
0067 //define singleton
0068 //define doubleton
0069 //define tripleton
0070 #define quadruplet
0071 //define quintuplet
0072 //define sextuplet
0073 //define septuplet
0074 //define octuplet
0075 //define nonuplet
0076 //define decuplet
0077 1 One singleton, singleton
0078 2 Two double, doublet, doubleton
0079 3 Three triple, triplet, tripleton
0080 4 Four quadruple, quadruplet, quadruplet
0081 5 Five quintuple, quintuplet, quintuplet
0082 6 Six sextuple, sextuplet, sextuplet
0083 7 Seven septuple, septuplet, septuplet
0084 8 Eight octuple, octuplet, octuplet

```

```

0085 9      Ni ne      nonuple, nonuplet, nonuplet on
0086 10     Ten,      decuple, decuplet, decuplet on
0087 1      One      ace, single, singleton, unary, unit, unity
0088 2      Two      binary, brace, couple, couplet, distich, deuce, double, doubleton, duad, duality, duet, duo, dyad, pair, snake eyes, span,
        twain, twosome, yoke
0089 3      Three    deuce-ace, leash, set, tercet, ternary, ternion, terzetto, threesome, tierce, trey, triad, trine, trinity, trio, triplet,
        troika, hat-trick
0090 4      Four     foursome, quadruplet, quatern, quaternary, quaternion, quaternity, quartet, tetrad
0091 5      Five     cinque, fin, fivesome, pentad, quint, quintet, quintuplet
0092 6      Six      half dozen, hexad, sestet, sextet, sextuplet, sise
0093 7      Seven    heptad, septet, septuplet
0094 8      Eight    octad, octave, octet, octonary, octuplet, ogdoad
0095 Also, in addition to 'Y' and 'Z', 'y' and 'z' were added in order to be able to dump only n-grams without occurrences.
0096 ]
0097 A lazy approach is applied in order to add occurrences of each 4-gram:
0098 - just reserve the last 4bytes in 'wrđ' for counter as follows:
0099 'LongestLineInclusive' has to be greater than 31 (51 looks good enough) in order not to miss longer 4-grams like:
        encourage_innovative_approaches_to
0100 char FourGram[LongestLineInclusive+1+4]; // 31bytes longest 4-gram + 1byte NULL + 4bytes COUNTER
0101 - the laziness lies here:
0102 no need to make the four bytes to house the value 1 when a new 'wrđ' is being inserted (either in step 1 or step 3) just add 1 at final
        traverse dump,
0103 in step 1 when a 'wrđ' is found then add 1 to the counter only if it is not 9,999,999 already (limitation enforced on counter).
0104 - when dumping the format has to be:
0105 0,000,001\ta_b_c_d
0106 in order to sort the whole lines later with external Qsort and have easy screening for rare/wrong/useless 4-grams.
0107
0108 Comment/Uncomment accordingly in order to compile:
0109 #define _WIN32_ENVIRONMENT_
0110 // #define _POSIX_ENVIRONMENT_
0111
0112 Windows compile(uncomment #include <i.o.h> line, ignore warnings):
0113 For Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86 use:
0114 cl /Ox /Wp64 /TcLeprechaun.c /FaLeprechaun
0115
0116 Windows compile(comment #include <i.o.h> line, ignore warnings):
0117 For Intel (R) C++ Compiler Professional for applications running on IA-32, Version 11.1 use:
0118 icl /Ox /Wp64 /TcLeprechaun.c /FaLeprechaun /w /QxHOST
0119
0120 Linux compile(ignore warnings):
0121 gcc -D_FILE_OFFSET_BITS=64 -m64 -static -O3 -mtune=generic Leprechaun_quadruplet on.c -o Leprechaun_quadruplet on_r14_generic_64bits.e lf
0122
0123 !!! For some reason a nasty bug (some UFO/wrong occurrences before phrases in the resultant file) occurs when 32bit (supposedly the opposite
        of the expected) code is generated:
0124 gcc -D_FILE_OFFSET_BITS=64 -m32 -static -O3 -mtune=generic Leprechaun_quadruplet on.c -o Leprechaun_quadruplet on_r14_generic_32bits.e lf
0125
0126 [It's a little weird(Intel boosts the sort while falls behind in parsing, tested on T3400):]
0127
0128 Leprechaun_r13_7pluses_Microsoft_32-bit_16.00.30319.01.exe_vs_Wikipedia_22,202,980_LATIN-Words:
0129 Words per second performance: 1,679,585W/s
0130 Time for making unsorted wordlist: 30 second(s)
0131 Time for sorting unsorted wordlist: 25 second(s)
0132
0133 Leprechaun_r13_7pluses_Intel_IA-32_11.1.exe_vs_Wikipedia_22,202,980_LATIN-Words:
0134 Words per second performance: 1,603,240W/s
0135 Time for making unsorted wordlist: 31 second(s)
0136 Time for sorting unsorted wordlist: 19 second(s)
0137
0138 Due to my ignorance(calderas in my C knowledge): 64bit code cannot be generated, for now.
0139 Any improvement is welcome.
0140 Enjoy!
0141 */
0142
0143 // C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_r13+++++_C_EXE>cl /Ox /Wp64 /TcLeprechaun.c /FaLeprechaun
0144 // Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
0145 // Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
0146 //
0147 // Leprechaun.c
0148 // Leprechaun.c(829) : warning C4312: 'type cast' : conversion from 'int' to 'string' of greater size
0149 // Leprechaun.c(849) : warning C4312: 'type cast' : conversion from 'int' to 'string *' of greater size
0150 // Leprechaun.c(2048) : warning C4312: 'type cast' : conversion from 'int' to 'char *' of greater size
0151 // Leprechaun.c(2063) : warning C4311: 'type cast' : pointer truncation from 'char *' to 'unsigned long'
0152 // Leprechaun.c(2068) : warning C4311: 'type cast' : pointer truncation from 'char *' to 'unsigned long'
0153 // Leprechaun.c(2371) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0154 // Leprechaun.c(2570) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0155 // Leprechaun.c(2626) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0156 // Leprechaun.c(2657) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0157 // Leprechaun.c(2663) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0158 // Leprechaun.c(2668) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0159 // Leprechaun.c(2696) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0160 // Leprechaun.c(2729) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0161 // Leprechaun.c(2743) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0162 // Leprechaun.c(2755) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0163 // Microsoft (R) Incremental Linker Version 7.10.3077
0164 // Copyright (C) Microsoft Corporation. All rights reserved.
0165 //
0166 // /out:Leprechaun.exe
0167 // Leprechaun.obj

```

```

0168 //
0169 // C:\WorkTemp\Leprechaun_r13++\Visual C++ Tool kit 2003\Leprechaun_r13+++++_C_EXE>
0170
0171 /*
0172 Below is the gain in 13++ and 13+++:
0173
0174 Words per second performance: 5,974,513W/s
0175 Word count: 4,582,451,898 of them 9,177,221 distinct
0176 Number Of Trees(GREATER THE BETTER): 2855919
0177 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 6321302
0178
0179 Words per second performance: 6,329,353W/s
0180 Word count: 4,582,451,898 of them 9,177,221 distinct
0181 Number Of Trees(GREATER THE BETTER): 2958681
0182 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 6218540
0183
0184 The aftermath: 6,321,302 - 6,218,540 = 102,762 less collisions while the speed of hash is not slower for sure - I call this: double trouble
avoidance.
0185 Thanks to Fowler/Noll/Vo hash inventors.
0186 */
0187
0188 /*
0189 Let's see the supplementary-clash on Intel Pentium T3400 Merom-1M 2166MHz:
0190 Binary-Search-Trees vs B-Trees of order 3
0191
0192 C:\WorkTemp\Leprechaun_r13++\Visual C++ Tool kit 2003\Leprechaun_step_1_PAIR-QUEST>Leprechaun_Microsoft.exe Leprechaun_vs_Wikipedia_en-
WORDS.lst Leprechaun_vs_Wikipedia_en-WORDS.wrd 4777 x
0193 Leprechaun(Fast Greedy Word-Ripper), revision 13+++++, written by Svalqyatchx.
0194 Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
0195 Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
0196 also the performance of a 3-way hash + 6,602,752 B-Trees of order 3.
0197 Size of input file with files for Leprechauning: 27
0198 Allocating memory 1863MB ... OK
0199 Size of Input TEXTual file: 146,973,879
0200 \; Word count: 12,561,874 of them 12,561,874 distinct; Done: 64/64
0201 Bytes per second performance: 14,697,387B/s
0202 Words per second performance: 1,256,187W/s
0203 Flushing unsorted words ...
0204 Time for making unsorted wordlist: 15 second(s)
0205 Deallocated memory in MB: 1863
0206 Allocated memory for words in MB: 141
0207 Allocated memory for pointers-to-words in MB: 48
0208 Sorting(with 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...
0209 Sort pass 26/26 ...
0210 Flushing sorted words ...
0211 Time for sorting unsorted wordlist: 14 second(s)
0212 Leprechaun: Done.
0213
0214 [An excerpt of Leprechaun.LOG:]
0215 Number Of Trees(GREATER THE BETTER): 2786806
0216 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 58,935,172
0217 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,806
0218
0219 C:\WorkTemp\Leprechaun_r13++\Visual C++ Tool kit 2003\Leprechaun_step_1_PAIR-QUEST>Leprechaun_Microsoft.exe Leprechaun_vs_Wikipedia_en-
WORDS.lst Leprechaun_vs_Wikipedia_en-WORDS.wrd 4777 y
0220 Leprechaun(Fast Greedy Word-Ripper), revision 13+++++, written by Svalqyatchx.
0221 Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
0222 Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
0223 also the performance of a 3-way hash + 6,602,752 B-Trees of order 3.
0224 Size of input file with files for Leprechauning: 27
0225 Allocating memory 1863MB ... OK
0226 Size of Input TEXTual file: 146,973,879
0227 \; Word count: 12,561,874 of them 12,561,874 distinct; Done: 64/64
0228 Bytes per second performance: 24,495,646B/s
0229 Words per second performance: 2,093,645W/s
0230 Flushing unsorted words ...
0231 Time for making unsorted wordlist: 12 second(s)
0232 Deallocated memory in MB: 1863
0233 Allocated memory for words in MB: 141
0234 Allocated memory for pointers-to-words in MB: 48
0235 Sorting(with 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...
0236 Sort pass 26/26 ...
0237 Flushing sorted words ...
0238 Time for sorting unsorted wordlist: 14 second(s)
0239 Leprechaun: Done.
0240
0241 [An excerpt of Leprechaun.LOG:]
0242 Number Of Trees(GREATER THE BETTER): 2786806
0243 Total Attempts to Find/Put WORDs into B-trees order 3: 18,534,910
0244
0245 C:\WorkTemp\Leprechaun_r13++\Visual C++ Tool kit 2003\Leprechaun_step_1_PAIR-QUEST>type Leprechaun_vs_Wikipedia_en-WORDS.lst
0246 wikipedia-en-html.tar.wrd
0247
0248 C:\WorkTemp\Leprechaun_r13++\Visual C++ Tool kit 2003\Leprechaun_step_1_PAIR-QUEST>dir Leprechaun_vs_Wikipedia_en-WORDS.*
0249 Volume in drive C is H320_Vol2
0250 Volume Serial Number is A094-FAE2
0251
0252 Directory of C:\WorkTemp\Leprechaun_r13++\Visual C++ Tool kit 2003\Leprechaun_step_1_PAIR-QUEST

```

```

0253
0254 09/14/2010 06:04 AM 27 Leprechaun_vs_Wikipedia_en-WORDS.lst
0255 09/15/2010 02:51 AM 146,973,879 Leprechaun_vs_Wikipedia_en-WORDS.wrd
0256 2 File(s) 146,973,906 bytes
0257 0 Dir(s) 965,787,648 bytes free
0258
0259 Conclusion:
0260 18,534,910/12,561,874=1.475 Average Attempts to Find/Put WORDs into B-trees order 3, not bad at all.
0261 */
0262
0263 // To do: must learn how to align, at last.
0264 /*
0265 Matt Mahoney ZPAQ fragment:
0266 T *data; // allocated memory
0267 int offset;
0268 ...
0269 offset=64-int((long)data&63);
0270 data=(T*)((char*)data+offset); // adjust to 64 byte boundary
0271
0272 quicklz.c fragment:
0273 #define QLZ_ALIGNMENT_PADD 8
0274 unsigned char *scratch_aligned = (unsigned char *)scratch_compress + QLZ_ALIGNMENT_PADD - (((size_t)scratch_compress) % QLZ_ALIGNMENT_PADD);
0275 size_t *buffersize = (size_t *)scratch_aligned;
0276
0277 minilzo.c fragment:
0278 #define lzo_uintptr_t unsigned long
0279 #define PTR(a) ((lzo_uintptr_t) (a))
0280 #define PTR_LINEAR(a) PTR(a)
0281 #define PTR_ALIGNED_4(a) ((PTR_LINEAR(a) & 3) == 0)
0282 */
0283
0284 //__declspec(align(64)) int BigArray[1024]; // Windows syntax
0285 //or
0286 //int BigArray[1024] __attribute__((aligned(64))); // Linux syntax
0287
0288 #if defined(_WIN32_ENVIRONMENT_)
0289 __declspec(align(64))
0290 #else
0291 //__attribute__((aligned(64)));
0292 #endif /* defined(_WIN32_ENVIRONMENT_) */
0293
0294 typedef unsigned short WORD; // As for 'With *(DWORD*)', a buffer overrun is possible at the end of a memory page.' I knew about it but was
    fooled by assembly code generated by VS2010 which translates it to a word access:
0295 //; 792 : hash32 = FNV_32A_OP32(hash32, *(UINT*)p&0xFFFF);
0296
0297 typedef unsigned int UINT;
0298 typedef unsigned int DWORD;
0299
0300 /*
0301 Enter-the-BESTer or an alchemical clash of pairs of primes.
0302
0303 When an x-bit hash where x < 16 and is not a power of 2 is needed,
0304 here comes 'FNV1A_Hash_4_OCTETS': a slightly tuned FNV1A hash for a huge(22,202,980) wordlist of latin-letters-words.
0305
0306 Two improvements for the generic(base) FNV1A hash:
0307 - first, better speed: by reducing 'imul' instructions when string is 4++ chars
0308 - second, better dispersion: by experimenting(superficially-lite test done, so far) with 'FNV1_32_PRIME'
0309
0310 Or more concretely:
0311 - For FNV1_32_INIT = 2166136261
0312 - Giving to 'FNV1_32_PRIME' all primes between 2 and 11987
0313 - Shifting by 16bits instead of 13bits, when 8192 slots are used
0314
0315 C code:
0316 typedef unsigned char u_int8_t;
0317 typedef unsigned long u_int32_t;
0318
0319 #define FNV1_32_INIT ((u_int32_t)2166136261)
0320 #define FNV1_32_PRIME ((u_int32_t)1607)
0321
0322 #define FNV_32A_OP(hash, octet) \
0323     (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * FNV1_32_PRIME)
0324
0325 #define FNV_32A_OP32(hash, octet) \
0326     (((u_int32_t)(hash) ^ (u_int32_t)(octet)) * FNV1_32_PRIME)
0327
0328 0800 // Invoking: FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2) // = 0,1,2,3,4,5,6,7 [1..31]
0329 0801 int FNV1A_Hash_4_OCTETS(char *str, int wrdlen_QUADRUPLETS)
0330 0802 {
0331 0803 u_int32_t hash;
0332 0804 char *p;
0333 0805
0334 0806 hash = FNV1_32_INIT;
0335 0807 p=str;
0336 0808
0337 0809 // The goal of stage #1: to reduce number of 'imul's.
0338 0810
0339 0811 // Stage #1:

```

```

0340 0812 for (; wrdlen_QUADRUPLTS != 0; --wrdlen_QUADRUPLTS) {
0341 0813     hash = FNV_32A_OP32(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
0342 0814     p=p+4; // add eax, 4
0343 0815 }
0344 0816
0345 0817 // Stage #2:
0346 0818 for (; *p; ++p) {
0347 0819     hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [eax]
0348 0820 }
0349 0821
0350 0822 //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
0351 0823 return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
0352 0824 }

```

0353

0354 Assembler code:

0355 _FNV1A_Hash_4_OCTETS PROC NEAR

0356 ; Line 812

0357 mov edx, DWORD PTR _wrdlen_QUADRUPLTS\$[esp-4]

0358 test edx, edx

0359 mov eax, DWORD PTR _str\$[esp-4]

0360 push esi

0361 mov esi, DWORD PTR _FNV1_32_PRIME

0362 mov ecx, -2128831035

0363 je SHORT \$L1612

0364 push edi

0365 npad 7

0366 \$L1610:

0367 ; Line 813

0368 mov edi, DWORD PTR [eax]

0369 xor edi, ecx

0370 imul edi, esi

0371 ; Line 814

0372 add eax, 4

0373 dec edx

0374 mov ecx, edi

0375 jne SHORT \$L1610

0376 pop edi

0377 \$L1612:

0378 ; Line 818

0379 mov dl, BYTE PTR [eax]

0380 test dl, dl

0381 je SHORT \$L1619

0382 \$L1617:

0383 ; Line 819

0384 movzx edx, dl

0385 xor edx, ecx

0386 imul edx, esi

0387 inc eax

0388 mov ecx, edx

0389 mov dl, BYTE PTR [eax]

0390 test dl, dl

0391 jne SHORT \$L1617

0392 \$L1619:

0393 ; Line 823

0394 mov eax, ecx

0395 shr eax, 16

0396 xor eax, ecx

0397 and eax, 8191

0398 pop esi

0399 ; Line 824

0400 ret 0

0401 _FNV1A_Hash_4_OCTETS ENDP

0402

0403

0404 So, 'FNV1A_Hash_4_OCTETS' calculates faster and gives better distribution(3549448 for 1607), which is 0.6% better(less collisions), than generic 'FNV1A_Hash' with 3527916.

0405

0406 FNV proves to be great, dealing with 4x8bits(four octets) at once doesn't hurt distribution at all, I was amazed by consistency(stable behaviour) of 'FNV1A_Hash_4_OCTETS'.

0407

0408 I want to make a total clash of all possible pairs 'FNV1_32_INIT' & 'FNV1_32_PRIME' in order to lessen even a few thousand collisions.

0409 This is critical for speed performance e.g. when 30,974,750,142 words, the case of wikipedia-en-html.tar, must be hashed.

0410 The current obstacle is needed-time: each filling (26 slots x 31 sub-slots x 8192 sub-sub-slots) executes in 32-36 seconds for each pair.

0411 Such an easy task, but I can't see how to get done, it is not hard but slow even with 15 times faster testbed.

0412

0413 Between 1..1166136247 there are 58,834,113 primes (inclusive).

0414 Between 1..16777619 there are 1,077,891 primes (inclusive).

0415 Or 58834113*1077891 = 63,416,760,895,683 pairs or 2,010,932 years needed at one-pair-per-second rate.

0416

0417 Finding THE best pair in my opinion is a total alchemy, due to the very nature of hashing: which is mainly alchemical and partly scientific.

0418 Since the magnum corpus of words is static-enough, THE pair is worthy to be found.

0419

0420 It doesn't take a think-tank to see the superiority of FNV, Fowler/Noll/Vo did reveal a thing of beauty.

0421

0422 Performance of 'FNV1A_Hash_4_OCTETS': 10236 words/clock or 105 MB/s|3,549,448 used slots (best)

0423

0424 CASE #1: with 'if (strlen(backup[j]) != 0)' before each execution

0425 Performance of 'KuxHash3plus' aka '2in1': 8076 words/clock or 82 MB/s|3,410,463 used slots (worst)

0426 Performance of 'FNV1A_Hash': 8079 words/clock or 83 MB/s|3,527,916 used slots
0427 Performance of 'FNV1A_Hash_SHIFTLess_XORless': 8109 words/clock or 83 MB/s|3,540,323 used slots
0428
0429 CASE #2: without 'if (strlen(backup[j]) != 0)' before each execution
0430 Performance of 'KuxHash3plus' aka '2in1': 11673 words/clock or 119 MB/s|3,410,463 used slots (worst)
0431 Performance of 'FNV1A_Hash': 11558 words/clock or 118 MB/s|3,527,916 used slots
0432 Performance of 'FNV1A_Hash_SHIFTLess_XORless': 11570 words/clock or 118 MB/s|3,540,323 used slots
0433
0434 Note:
0435 The 'strlen' overhead(CASE #1) is necessary due to priorly(before hash invocation) needed len-of-string for 'FNV1A_Hash_4_OCTETS'.
0436 Almost always, that is the case, since parsing of incoming text must know length of words/lines/files.
0437 In case of not knowing this length: ((119-105)/105)*100% = 13% degradation is unacceptable.
0438 The 'strlen' is an awful brake.
0439 Also whether the code overhead(one additional cycle) of 'FNV1A_Hash_4_OCTETS' is so successful (as a trade-off) or the testbed is deceiving I do not know, here I am not so sure regardless of notorious delays caused by 'imul' and 'div' instructions.
0440 */
0441
0442 /*
0443 FNV1_32_PRIME: //?: 16777619
0444
0445 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
0446 Words per second performance: 1,046,822W/s
0447 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0448 Size of all TEXTual Files: 146,973,879
0449 Word count: 12,561,874 of them 12,561,874 distinct
0450 Number Of Trees(GREATER THE BETTER): 2775839
0451
0452 Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 72 and LEAFs = 15
0453 Words per second performance: 1,356,588W/s
0454 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0455 Size of all TEXTual Files: 415,982,896
0456 Word count: 35,271,297 of them 22,202,980 distinct
0457 Number Of Trees(GREATER THE BETTER): 3539690
0458
0459 FNV1_32_PRIME: //3549448: 1607
0460
0461 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
0462 Words per second performance: 1,046,822W/s
0463 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0464 Size of all TEXTual Files: 146,973,879
0465 Word count: 12,561,874 of them 12,561,874 distinct
0466 Number Of Trees(GREATER THE BETTER): 2783970
0467
0468 Above Binary-Search-Tree with MaxPEAK = 38 has NODEs = 50 and LEAFs = 11
0469 Words per second performance: 1,410,851W/s
0470 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0471 Size of all TEXTual Files: 415,982,896
0472 Word count: 35,271,297 of them 22,202,980 distinct
0473 Number Of Trees(GREATER THE BETTER): 3549395
0474
0475 FNV1_32_PRIME: //3550132: 175757909
0476
0477 Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
0478 Words per second performance: 966,298W/s
0479 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0480 Size of all TEXTual Files: 146,973,879
0481 Word count: 12,561,874 of them 12,561,874 distinct
0482 Number Of Trees(GREATER THE BETTER): 2784479
0483
0484 Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 64 and LEAFs = 12
0485 Words per second performance: 1,410,851W/s
0486 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0487 Size of all TEXTual Files: 415,982,896
0488 Word count: 35,271,297 of them 22,202,980 distinct
0489 Number Of Trees(GREATER THE BETTER): 3550115
0490
0491 FNV1_32_PRIME: //3550687: 201887489
0492
0493 Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
0494 Words per second performance: 966,298W/s
0495 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0496 Size of all TEXTual Files: 146,973,879
0497 Word count: 12,561,874 of them 12,561,874 distinct
0498 Number Of Trees(GREATER THE BETTER): 2784377
0499
0500 Above Binary-Search-Tree with MaxPEAK = 40 has NODEs = 55 and LEAFs = 11
0501 Words per second performance: 1,356,588W/s
0502 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0503 Size of all TEXTual Files: 415,982,896
0504 Word count: 35,271,297 of them 22,202,980 distinct
0505 Number Of Trees(GREATER THE BETTER): 3550528
0506
0507 FNV1_32_PRIME: //3550733: 172783361
0508
0509 Above Binary-Search-Tree with MaxPEAK = 59 has NODEs = 59 and LEAFs = 1
0510 Words per second performance: 1,046,822W/s
0511 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0512 Size of all TEXTual Files: 146,973,879

0513 Word count: 12,561,874 of them 12,561,874 distinct
 0514 Number Of Trees(GREATER THE BETTER): 2786362
 0515
 0516 Above Binary-Search-Tree with MaxPEAK = 38 has NODEs = 70 and LEAFs = 17
 0517 Words per second performance: 1,410,851W/s
 0518 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0519 Size of all TEXTual Files: 415,982,896
 0520 Word count: 35,271,297 of them 22,202,980 distinct
 0521 Number Of Trees(GREATER THE BETTER): 3550746
 0522
 0523 FNV1_32_PRIME: //3550929: 204312319
 0524
 0525 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
 0526 Words per second performance: 966,298W/s
 0527 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
 0528 Size of all TEXTual Files: 146,973,879
 0529 Word count: 12,561,874 of them 12,561,874 distinct
 0530 Number Of Trees(GREATER THE BETTER): 2785581
 0531
 0532 Above Binary-Search-Tree with MaxPEAK = 37 has NODEs = 55 and LEAFs = 12
 0533 Words per second performance: 1,356,588W/s
 0534 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0535 Size of all TEXTual Files: 415,982,896
 0536 Word count: 35,271,297 of them 22,202,980 distinct
 0537 Number Of Trees(GREATER THE BETTER): 3550886
 0538
 0539 Leprechaun_Microsoft.exe: FNV1_32_PRIME: //3551736: 107712257
 0540
 0541 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
 0542 Words per second performance: 1,046,822W/s
 0543 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
 0544 Size of all TEXTual Files: 146,973,879
 0545 Word count: 12,561,874 of them 12,561,874 distinct
 0546 Number Of Trees(GREATER THE BETTER): 2786515
 0547
 0548 Above Binary-Search-Tree with MaxPEAK = 36 has NODEs = 64 and LEAFs = 15
 0549 Words per second performance: 1,356,588W/s
 0550 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0551 Size of all TEXTual Files: 415,982,896
 0552 Word count: 35,271,297 of them 22,202,980 distinct
 0553 Number Of Trees(GREATER THE BETTER): 3551744
 0554
 0555 Leprechaun_Intel.exe: FNV1_32_PRIME: //3551736: 107712257
 0556
 0557 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
 0558 Words per second performance: 1,256,187W/s
 0559 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
 0560 Size of all TEXTual Files: 146,973,879
 0561 Word count: 12,561,874 of them 12,561,874 distinct
 0562 Number Of Trees(GREATER THE BETTER): 2786515
 0563
 0564 Above Binary-Search-Tree with MaxPEAK = 36 has NODEs = 64 and LEAFs = 15
 0565 Words per second performance: 1,603,240W/s
 0566 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0567 Size of all TEXTual Files: 415,982,896
 0568 Word count: 35,271,297 of them 22,202,980 distinct
 0569 Number Of Trees(GREATER THE BETTER): 3551744
 0570
 0571 Wow: 1,603,240W/s vs 1,356,588W/s respectively Leprechaun_Intel.exe vs Leprechaun_Microsoft.exe, i.e. 18% betterment, no joke!
 0572
 0573 Al chemical search for best PRIME-PAIR revision uses next line:
 0574 Slot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13+++
 0575 This revision uses next lines:
 0576 if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
 0577 Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>2, 2)<<2; //13++++
 0578 else // 2x8+4=20 i.e. first contains 6 clashes
 0579 Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>3, 3)<<2; //13++++
 0580
 0581 ! An expected but unpleasant degradation for 3551961: 428904191 compared to 3551736: 107712257, this shows 'FNV1A_Hash_4_OCTETS' has only figurative purpose - the 4 lines of 'FNV1A_Hash_Granularity' decide the last usefulness.
 0582
 0583 Leprechaun.exe: FNV1_32_PRIME: //3551961: 428904191
 0584
 0585 Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
 0586 Words per second performance: 966,298W/s
 0587 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
 0588 Size of all TEXTual Files: 146,973,879
 0589 Word count: 12,561,874 of them 12,561,874 distinct
 0590 Number Of Trees(GREATER THE BETTER): 2786383
 0591
 0592 Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 71 and LEAFs = 16
 0593 Words per second performance: 1,410,851W/s
 0594 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0595 Size of all TEXTual Files: 415,982,896
 0596 Word count: 35,271,297 of them 22,202,980 distinct
 0597 Number Of Trees(GREATER THE BETTER): 3551503
 0598
 0599 Leprechaun.exe: FNV1_32_PRIME: //3552103: 588411137

0600
 0601 Above Binary-Search-Tree with MaxPEAK = 6 has NODEs = 6 and LEAFs = 1
 0602 Size of all TEXTual Files: 4,067,439
 0603 Word count: 358,798 of them 351,116 distinct
 0604 Number Of Trees(GREATER THE BETTER): 310622
 0605 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 310,622
 0606
 0607 Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
 0608 Size of all TEXTual Files: 146,973,879
 0609 Word count: 12,561,874 of them 12,561,874 distinct
 0610 Number Of Trees(GREATER THE BETTER): 2786485
 0611 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,485
 0612
 0613 Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 62 and LEAFs = 15
 0614 Size of all TEXTual Files: 415,982,896
 0615 Word count: 35,271,297 of them 22,202,980 distinct
 0616 Number Of Trees(GREATER THE BETTER): 3551956
 0617 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,131
 0618
 0619 Leprechaun.exe: FNV1_32_PRIME: //3552039: 602173697 !!!GOODEST so far!!!
 0620
 0621 Above Binary-Search-Tree with MaxPEAK = 6 has NODEs = 6 and LEAFs = 1
 0622 Size of all TEXTual Files: 4,067,439
 0623 Word count: 358,798 of them 351,116 distinct
 0624 Number Of Trees(GREATER THE BETTER): 310948
 0625 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 310,948
 0626
 0627 Above Binary-Search-Tree with MaxPEAK = 63 has NODEs = 63 and LEAFs = 1
 0628 Size of all TEXTual Files: 146,973,879
 0629 Word count: 12,561,874 of them 12,561,874 distinct
 0630 Number Of Trees(GREATER THE BETTER): 2786806
 0631 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,806
 0632
 0633 Above Binary-Search-Tree with MaxPEAK = 36 has NODEs = 52 and LEAFs = 9
 0634 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0635 Size of all TEXTual Files: 415,982,896
 0636 Word count: 35,271,297 of them 22,202,980 distinct
 0637 Number Of Trees(GREATER THE BETTER): 3552296
 0638 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,899
 0639
 0640 Between 1 and 602392027 at step 100 following FNV1_32_PRIMES(for FNV1_32_INIT=2166136261) give(FNV1A_Hash_4_OCTETS) dispersion:
 0641 3550022: 423779327
 0642 3550028: 513793537
 0643 3550053: 434840321
 0644 3550067: 437062229
 0645 3550080: 420344321
 0646 3550090: 304777471
 0647 3550097: 496547839
 0648 3550129: 390809599
 0649 3550132: 175757909
 0650 3550163: 353712127
 0651 3550231: 334434817
 0652 3550237: 272789761
 0653 3550247: 590341121
 0654 3550255: 358814207
 0655 3550277: 437182721
 0656 3550326: 521795327
 0657 3550347: 311867393
 0658 3550447: 456137729
 0659 3550458: 418208767
 0660 3550516: 602048767
 0661 3550525: 513597697
 0662 3550526: 347283199
 0663 3550528: 598773503
 0664 3550592: 598139137
 0665 3550598: 242448127
 0666 3550611: 571481087
 0667 3550628: 457012993
 0668 3550664: 482822143
 0669 3550666: 249098753
 0670 3550687: 201887489
 0671 3550702: 489976063
 0672 3550710: 272961023
 0673 3550733: 172783361
 0674 3550734: 431562497
 0675 3550929: 204312319
 0676 3550984: 562853633
 0677 3550991: 551362303
 0678 3551359: 332820737
 0679 3551484: 354126079
 0680 3551514: 407138561
 0681 3551523: 442058753
 0682 3551701: 449230849
 0683 3551736: 107712257
 0684 3551961: 428904191
 0685 3552039: 602173697
 0686 3552103: 588411137
 0687 */

```

0688
0689 // Windows: ~~~~~
0690 // _CRTIMP size_t __cdecl fread(void *, size_t, size_t, FILE *);
0691 // _CRTIMP size_t __cdecl fwrite(const void *, size_t, size_t, FILE *);
0692 // _CRTIMP int __cdecl fgetpos(FILE *, fpos_t *);
0693 // _CRTIMP int __cdecl fsetpos(FILE *, const fpos_t *);
0694
0695 // _CRTIMP __int64 __cdecl _lseeki64(int, __int64, int);
0696 // _CRTIMP __int64 __cdecl _telli64(int);
0697 // _CRTIMP __int64 __cdecl _filengthi64(int);
0698 // above 3 are in 'io.h'
0699
0700 // _CRTIMP int __cdecl fseek(FILE *, long, int);
0701 // _CRTIMP long __cdecl ftell(FILE *);
0702 // _CRTIMP int __cdecl fclose(FILE *);
0703
0704 // #ifndef _SIZE_T_DEFINED
0705 // #if defined _WIN64
0706 //     typedef unsigned __int64    size_t;
0707 // #else
0708 //     typedef _W64 unsigned int    size_t;
0709 // #endif
0710 // #define _SIZE_T_DEFINED
0711 // #endif
0712
0713 // typedef __int64 fpos_t;
0714
0715 // Linux: ~~~~~
0716 // size_t fread (void *data, size_t size, size_t count, FILE *stream)
0717 // size_t fwrite (const void *data, size_t size, size_t count, FILE *stream)
0718 // int fgetpos (FILE *stream, fpos_t *position)
0719 // int fsetpos (FILE *stream, const fpos_t *position)
0720
0721 // FILE * fopen64 (const char *filename, const char *opentype)
0722 // int fseeko64 (FILE *stream, off64_t offset, int whence)
0723 // off64_t ftello64 (FILE *stream)
0724 // int fclose (FILE *stream)
0725
0726 // off_t lseek (int filedes, off_t offset, int whence)
0727 // above 1 is in 'unistd.h'
0728
0729
0730 // ===== MUST work both for Windows and Linux =====
0731 //Only one must be uncommented:
0732 #define _WIN32_ENVIRONMENT_
0733 // #define _POSIX_ENVIRONMENT_
0734
0735 //Only one must be uncommented:
0736 // #define singleton
0737 // #define doubleton
0738 #define tripleton
0739 // #define quadrupleton
0740 // #define quintupleton
0741 // #define sextupleton
0742 // #define septupleton
0743 // #define octupleton
0744 // #define nonupleton
0745 // #define decupleton
0746
0747 #if defined singleton
0748 #define _ngram_ 1
0749 #endif
0750 #if defined doubleton
0751 #define _ngram_ 2
0752 #endif
0753 #if defined tripleton
0754 #define _ngram_ 3
0755 #endif
0756 #if defined quadrupleton
0757 #define _ngram_ 4
0758 #endif
0759 #if defined quintupleton
0760 #define _ngram_ 5
0761 #endif
0762 #if defined sextupleton
0763 #define _ngram_ 6
0764 #endif
0765 #if defined septupleton
0766 #define _ngram_ 7
0767 #endif
0768 #if defined octupleton
0769 #define _ngram_ 8
0770 #endif
0771 #if defined nonupleton
0772 #define _ngram_ 9
0773 #endif
0774 #if defined decupleton
0775 #define _ngram_ 10

```

```

0776 #endif
0777
0778 #ifndef NULL
0779 #define __cplusplus
0780 #define NULL 0
0781 #else
0782 #define NULL ((void*)0)
0783 #endif
0784 #endif
0785
0786 #define HashInBITS 24 // default 26 i.e. 2^26 i.e. 64MS(Mega Slots); slots contain 8bytes pointers or 512MB, because many netbooks have 512MB
    free (1GB in total)!
0787 #define HashChunkSizeInBITS 19 // Defines the number of passes. Should be smaller or equal to HashInBITS. If HashInBITS == HashChunkSizeInBITS
    then 2^(HashInBITS-HashChunkSizeInBITS)=2^0=1 pass(es).
0788 /*
0789 Tests done on super-speed-ramdisk 1800MB:
0790 Leprechaun_quadrupton rev. 14+ in fact differs from r.14 only with optimized(LEAFwise) fragment 1] and 2]. Fragment 3] and dump are not still
    optimized. The goal is to track how this partial tweak will affect 64KS(Kilo Slots) or 512KB hash or 1000 times smaller hash variant.
0791
0792 [Variant (HashInBITS 26 - 0) with 512MB hash:]
0793
0794 Leprechaun_quadrupton (Fast Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
0795 Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
0796 Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
0797 Feature2: The bottleneck is seek-time, if the external memory has latency 100+microseconds then look further.
0798 Size of input file with files for Leprechauning: 19
0799 Allocating HASH memory 536,870,977 bytes ... OK
0800 Allocating/ZEROing 1,292,478,478 bytes swap file ... OK
0801 Size of Input TEXTual file: 206,908,949
0802 |; 0,065,139P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
0803 Bytes per second performance: 718,433B/s
0804 Phrases per second performance: 65,139P/s
0805 Time for putting phrases into trees: 288 second(s)
0806 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,014,439P/s
0807 Time for shaking phrases from trees: 704 second(s)
0808 Dump LEAFwise also [
0809 Bytes per second performance: 736,330B/s
0810 Phrases per second performance: 66,762P/s
0811 Time for putting phrases into trees: 281 second(s)
0812 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,023,807P/s
0813 Time for shaking phrases from trees: 427 second(s)
0814 Dump LEAFwise also ]
0815 Leprechaun: Done.
0816
0817 Leprechaun report:
0818 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 731,746
0819 Number Of Trees(GREATER THE BETTER): 9,433,894
0820 Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 69,623
0821 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 1
0822 Used value for third parameter in KB: 1,262,186
0823 Use next time as third parameter: 1,262,186
0824 Total Attempts to Find/Put WORDs into B-trees order 3: 365,283
0825
0826 [Variant (HashInBITS 26 - 10) with 512KB hash:]
0827
0828 Leprechaun_quadrupton (Fast Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
0829 Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
0830 Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
0831 Feature2: The bottleneck is seek-time, if the external memory has latency 100+microseconds then look further.
0832 Size of input file with files for Leprechauning: 19
0833 Allocating HASH memory 524,353 bytes ... OK
0834 Allocating/ZEROing 1,292,478,478 bytes swap file ... OK
0835 Size of Input TEXTual file: 206,908,949
0836 |; 0,014,331P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
0837 Bytes per second performance: 158,066B/s
0838 Phrases per second performance: 14,331P/s
0839 Time for putting phrases into trees: 1309 second(s)
0840 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,019,739P/s
0841 Time for shaking phrases from trees: 515 second(s)
0842 Dump LEAFwise also [
0843 Bytes per second performance: 158,429B/s
0844 Phrases per second performance: 14,364P/s
0845 Time for putting phrases into trees: 1306 second(s)
0846 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,041,492P/s
0847 Time for shaking phrases from trees: 245 second(s)
0848 Dump LEAFwise also ]
0849 Dump & Insert LEAFwise also [
0850 Bytes per second performance: 174,459B/s
0851 Phrases per second performance: 15,818P/s
0852 Time for putting phrases into trees: 1186 second(s)
0853 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,041,323P/s
0854 Time for shaking phrases from trees: 246 second(s)
0855 Dump & Insert LEAFwise also ]
0856 Leprechaun: Done.
0857
0858 Leprechaun report:
0859 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 10,100,104
0860 Number Of Trees(GREATER THE BETTER): 65,536

```

```

0861 Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 7,522,788
0862 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 6
0863 Used value for third parameter in KB: 1,262,186
0864 Use next time as third parameter: 1,007,825
0865 Total Attempts to Find/Put WORDs into B-trees order 3: 84,868,241
0866
0867 [Variant (HashInBITS 26 - 20) with 512 hash:]
0868
0869 Leprechaun_quadruplet (Fast Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
0870 Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
0871 Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
0872 Feature2: The bottleneck is seek-time, if the external memory has latency 100+microseconds then look further.
0873 Size of input file with files for Leprechauning: 19
0874 Allocating HASH memory 577 bytes ... OK
0875 Allocating/ZEROing 1,292,478,478 bytes swap file ... OK
0876 Size of Input TEXTual file: 206,908,949
0877 |; 0,007,717P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
0878 Bytes per second performance: 85,112B/s
0879 Phrases per second performance: 7,717P/s
0880 Time for putting phrases into trees: 2431 second(s)
0881 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,019,777P/s
0882 Time for shaking phrases from trees: 514 second(s)
0883 Leprechaun: Done.
0884
0885 Leprechaun report:
0886 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 10,165,576
0887 Number Of Trees(GREATER THE BETTER): 64
0888 Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 7,592,585
0889 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 14
0890 Used value for third parameter in KB: 1,262,186
0891 Use next time as third parameter: 1,008,399
0892 Total Attempts to Find/Put WORDs into B-trees order 3: 271,393,689
0893
0894 r.14++ physical memory test [Variant (HashInBITS 26 - 0) with 512MB hash:]:
0895 D:\_KAZE_new-stuff\Leprechaun_quadruplet_r14+_64bit_Physical -n-Virtual>OSH0-TEST_INTERNAL.BAT
0896 Leprechaun_quadruplet (Fast-In-Future Greedy Phrase-Ripper), rev. 14++, written by Svalqyatchx.
0897 Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
0898 Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
0899 Feature2: If the external memory has latency 99+microseconds then !(look no further), IOPS(seek-time) rules.
0900 Size of input file with files for Leprechauning: 19
0901 Allocating HASH memory 536,870,977 bytes ... OK
0902 Allocating memory 1233MB ... OK
0903 Size of Input TEXTual file: 206,908,949
0904 |; 1,042,234P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
0905 Bytes per second performance: 11,494,941B/s
0906 Phrases per second performance: 1,042,234P/s
0907 Time for putting phrases into trees: 18 second(s)
0908 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,597,978P/s
0909 Time for shaking phrases from trees: 17 second(s)
0910 Leprechaun: Done.
0911
0912 D:\_KAZE_new-stuff\Leprechaun_quadruplet_r14+_64bit_Physical -n-Virtual>type Leprechaun.LOG
0913 Leprechaun report:
0914 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 731,746
0915 Number Of Trees(GREATER THE BETTER): 9,433,894
0916 Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 69,623
0917 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 1
0918 Used value for third parameter in KB: 1,262,186
0919 Use next time as third parameter: 1,262,186
0920 Total Attempts to Find/Put WORDs into B-trees order 3: 365,283
0921
0922 D:\_KAZE_new-stuff\Leprechaun_quadruplet_r14+_64bit_Physical -n-Virtual>
0923 */
0924
0925 // To do #1: Put this 31 in MAXw: 'int MAXw = 31;'
0926 // To do #2: No need of flushing unsorted words to file: make backup[] array
0927 // instead of writing. And mostly sort 26 times!
0928 // HEAVY BUG in r.7: unsigned long Hll(unsigned long n)
0929 // is NOT identical with
0930 // unsigned long GRMBLhll[32]; // 00 not used, only 01..31
0931 // BECAUSE DUMBEST DUMB Array GRMBLhll expects 'int' not
0932 // 'unsigned long' !!!
0933
0934 #include <stdio.h>
0935 #include <ctype.h>
0936 #include <time.h>
0937 #if defined(_WIN32_ENVIRONMENT_)
0938 #include <i.o.h> // needed for Windows' 'Iseeki64' and 'telli64'
0939 //Above line must be commented in order to compile with Intel C compiler: an error "can't find i.o.h" occurs.
0940 #else
0941 #endif /* defined(_WIN32_ENVIRONMENT_) */
0942
0943 typedef unsigned char char_t;
0944 typedef char_t *string;
0945
0946 clock_t clocks1, clocks2;
0947 int Bozan;
0948

```

```

0949 typedef unsigned char u_int8_t; //FNV only
0950 typedef unsigned long u_int32_t; //FNV only
0951 typedef unsigned long long u_int64_t; //FNV only
0952
0953 // SINHA fragment[
0954
0955 #define swapKAZE(a, b) { t = *(a); *(a) = *(b); *(b) = t; }
0956
0957 static void InsertSortKAZE(string *a, int n, int d) //void inssort(unsigned char **a, int n, int d)
0958 { string *pi, *pj, s, t; //unsigned char **pi, **pj, *s, *t;
0959   for (pi = a + 1; --n > 0; pi++)
0960     for (pj = pi; pj > a; pj--) {
0961       /* Inline strcmp: break if *(pj-1) <= *pj */
0962       for (s=*(pj-1)+d, t=*pj+d; *s==*t && *s!=0; s++, t++)
0963         ;
0964       if (*s <= *t)
0965         break;
0966       swapKAZE(pj, pj-1);
0967     }
0968 }
0969
0970 //int cmpit(unsigned char **h1, unsigned char **h2)
0971 //{
0972 //   return( strcmp(*h1, *h2) );
0973 //}
0974
0975 //int scmp( unsigned char *s1, unsigned char *s2 )
0976 //{
0977 //   while( *s1 != '\0' && *s1 == *s2 )
0978 //   {
0979 //       s1++;
0980 //       s2++;
0981 //   }
0982 //   return( *s1-*s2 );
0983 //}
0984
0985 //static void simplesort(string a[], int n, int b)
0986 //{
0987 //   int i, j;
0988 //   string tmp;
0989 //
0990 //   for (i = 1; i < n; i++)
0991 //       for (j = i; j > 0 && scmp(a[j-1]+b, a[j]+b) > 0; j--)
0992 //           { tmp = a[j]; a[j] = a[j-1]; a[j-1] = tmp; }
0993 //}
0994
0995 // SINHA fragment]
0996
0997 // mkqsort.c BEGIN *****
0998 /*
0999  Multikey quicksort, a radix sort algorithm for arrays of character
1000  strings by Bentley and Sedgewick.
1001
1002  J. Bentley and R. Sedgewick. Fast algorithms for sorting and
1003  searching strings. In Proceedings of 8th Annual ACM-SIAM Symposium
1004  on Discrete Algorithms, 1997.
1005
1006  http://www.CS.Princeton.EDU/~rs/strings/index.html
1007
1008  The code presented in this file has been tested with care but is
1009  not guaranteed for any purpose. The writer does not offer any
1010  warranties nor does he accept any liabilities with respect to
1011  the code.
1012
1013  Ranjan Sinha, 1 jan 2003.
1014
1015  School of Computer Science and Information Technology,
1016  RMIT University, Melbourne, Australia
1017  rsinha@cs.rmit.edu.au
1018
1019 */
1020
1021 //include "sortstring.h"
1022
1023 /* MULTIKEY QUICKSORT */
1024
1025 #ifndef min
1026 #define min(a, b) ((a)<=(b) ? (a) : (b))
1027 #endif
1028
1029
1030 // ----- BTREE [
1031 #define false -1
1032 #define true 0
1033
1034 struct nodeBTREE {
1035   int data;
1036   struct nodeBTREE* left;

```

```

1037 struct nodeBTREE* right;
1038 };
1039
1040 // ----- BTREE ]
1041
1042
1043 /* ssort2 -- Faster Version of Multikey Quicksort */
1044
1045 void vecswap2(unsigned char **a, unsigned char **b, int n)
1046 { while (n-- > 0) {
1047     unsigned char *t = *a;
1048     *a++ = *b;
1049     *b++ = t;
1050 }
1051 }
1052
1053 #define swap2(a, b) { t = *(a); *(a) = *(b); *(b) = t; }
1054 #define ptr2char(i) (*(i) + depth)
1055
1056 unsigned char **med3func(unsigned char **a, unsigned char **b, unsigned char **c, int depth)
1057 { int va, vb, vc;
1058   if ((va=ptr2char(a)) == (vb=ptr2char(b)))
1059     return a;
1060   if ((vc=ptr2char(c)) == va || vc == vb)
1061     return c;
1062   return va < vb ?
1063     (vb < vc ? b : (va < vc ? c : a)) :
1064     (vb > vc ? b : (va < vc ? a : c));
1065 }
1066 #define med3(a, b, c) med3func(a, b, c, depth)
1067
1068 void insort(unsigned char **a, int n, int d)
1069 { unsigned char **pi, **pj, *s, *t;
1070   for (pi = a + 1; --n > 0; pi++)
1071     for (pj = pi; pj > a; pj--) {
1072       /* Inline strcmp: break if *(pj-1) <= *pj */
1073       for (s=*(pj-1)+d, t=*pj+d; *s==*t && *s!=0; s++, t++)
1074         ;
1075       if (*s <= *t)
1076         break;
1077       swap2(pj, pj-1);
1078     }
1079 }
1080
1081 void mkqsort(unsigned char **a, int n, int depth)
1082 { int d, r, partval;
1083   unsigned char **pa, **pb, **pc, **pd, **pl, **pm, **pn, *t;
1084   if (n < 20) {
1085     insort(a, n, depth);
1086     return;
1087   }
1088   pl = a;
1089   pm = a + (n/2);
1090   pn = a + (n-1);
1091   if (n > 30) { /* On big arrays, pseudomedian of 9 */
1092     d = (n/8);
1093     pl = med3(pl, pl+d, pl+2*d);
1094     pm = med3(pm-d, pm, pm+d);
1095     pn = med3(pn-2*d, pn-d, pn);
1096   }
1097   pm = med3(pl, pm, pn);
1098   swap2(a, pm);
1099   partval = ptr2char(a);
1100   pa = pb = a + 1;
1101   pc = pd = a + n-1;
1102   for (;;) {
1103     while (pb <= pc && (r = ptr2char(pb)-partval) <= 0) {
1104       if (r == 0) { swap2(pa, pb); pa++; }
1105       pb++;
1106     }
1107     while (pb <= pc && (r = ptr2char(pc)-partval) >= 0) {
1108       if (r == 0) { swap2(pc, pd); pd--; }
1109       pc--;
1110     }
1111     if (pb > pc) break;
1112     swap2(pb, pc);
1113     pb++;
1114     pc--;
1115   }
1116   pn = a + n;
1117   r = min(pa-a, pb-pa); vecswap2(a, pb-r, r);
1118   r = min(pd-pc, pn-pd-1); vecswap2(pb, pn-r, r);
1119   if ((r = pb-pa) > 1)
1120     mkqsort(a, r, depth);
1121   if (ptr2char(a+r) != 0)
1122     mkqsort(a+r, pa-a+pn-pd-1, depth+1);
1123   if ((r = pd-pc) > 1)
1124     mkqsort(a+n-r, r, depth);

```

```

1125 }
1126
1127 void mkqsort_main(unsigned char **a, int n) { mkqsort(a, n, 0); }
1128 // mkqsort.c END *****
1129
1130 // Why Sinha uses int instead of long??!!
1131 static int readlines(char *file_name, string **lines)
1132 {
1133     int nlines = 0;
1134     size_t size;
1135     FILE *in_file;
1136     string basep, cur, next;
1137     string *ASbackup;
1138
1139     if (!(in_file = fopen(file_name, "rb"))) {
1140         printf("Leprechaun: Can't open file %s\n", file_name);
1141         exit(-1);
1142     }
1143     fseek(in_file, 0, SEEK_END);
1144     size = ftell(in_file);
1145     fseek(in_file, 0, SEEK_SET);
1146     if (!(basep = (string) malloc(size*sizeof(char_t)))) return -1;
1147     printf("Allocated memory for words in MB: %lu\n", ((size*sizeof(char_t))>>20)+1);
1148     if (fread(basep, 1, size, in_file) < size) {
1149         printf("Leprechaun: Can't read file %s\n", file_name);
1150         exit(-1);
1151     }
1152     fclose(in_file);
1153
1154     // GET nlines:
1155     cur = basep;
1156     while (cur < basep + size) {
1157         next = cur;
1158         while ((next < basep + size) && (*next != '\n')) {next++;}
1159         *--next = '\0'; // This is ala DOS i.e. Windows
1160                        // 1310 not 10(\n=10)
1161         cur = next + 2;
1162         nlines++;
1163     }
1164
1165     // printf("%lu\n", (unsigned long)*lines); -> backup = *lines = 0
1166     ASbackup = (string *)malloc( nlines*sizeof(string) ); // sizeof(string) is 4
1167     if( ASbackup == NULL )
1168     { puts("Leprechaun: Needed memory allocation denied!\n"); return( 1 ); }
1169     printf("Allocated memory for pointers-to-words in MB: %lu\n", ((nlines*sizeof(string))>>20)+1);
1170     *lines = ASbackup;
1171     //printf("%lu\n", (unsigned long)*lines); -> backup = *lines = ASbackup = 6946888
1172
1173     // Upload nlines times:
1174     nlines = 0;
1175     cur = basep;
1176     while (cur < basep + size) {
1177         next = cur;
1178         while ((next < basep + size) && (*next != '\n')) {next++;}
1179         *--next = '\0'; // This is ala DOS i.e. Windows
1180                        // 1310 not 10(\n=10)
1181         ASbackup[nlines] = cur;
1182         cur = next + 2;
1183         nlines++;
1184     }
1185     return nlines;
1186 }
1187
1188 void x64toaKAZE ( /* stdcall is faster and smaller... Might as well use it for the helper. */
1189     unsigned long long val,
1190     char *buf,
1191     unsigned radix,
1192     int is_neg
1193 )
1194 {
1195     char *p; /* pointer to traverse string */
1196     char *firstdig; /* pointer to first digit */
1197     char temp; /* temp char */
1198     unsigned digval; /* value of digit */
1199
1200     p = buf;
1201
1202     if ( is_neg )
1203     {
1204         *p++ = '-'; /* negative, so output '-' and negate */
1205         val = (unsigned long long)(-(long long)val);
1206     }
1207
1208     firstdig = p; /* save pointer to first digit */
1209
1210     do {
1211         digval = (unsigned) (val % radix);
1212         val /= radix; /* get next digit */

```

```

1213
1214     /* convert to ascii and store */
1215     if (digval > 9)
1216         *p++ = (char) (digval - 10 + 'a'); /* a letter */
1217     else
1218         *p++ = (char) (digval + '0');      /* a digit */
1219     } while (val > 0);
1220
1221     /* We now have the digit of the number in the buffer, but in reverse
1222     order. Thus we reverse them now. */
1223
1224     *p-- = '\0'; /* terminate string; p points to last digit */
1225
1226     do {
1227         temp = *p;
1228         *p = *firstdig;
1229         *firstdig = temp; /* swap *p and *firstdig */
1230         --p;
1231         ++firstdig; /* advance to next two digits */
1232     } while (firstdig < p); /* repeat until halfway */
1233 }
1234
1235 /* Actual functions just call conversion helper with neg flag set correctly,
1236 and return pointer to buffer. */
1237
1238 char * _i64toaKAZE (
1239     long long val,
1240     char *buf,
1241     int radix
1242 )
1243 {
1244     x64toaKAZE((unsigned long long)val, buf, radix, (radix == 10 && val < 0));
1245     return buf;
1246 }
1247
1248 char * _ui64toaKAZE (
1249     unsigned long long val,
1250     char *buf,
1251     int radix
1252 )
1253 {
1254     x64toaKAZE(val, buf, radix, 0);
1255     return buf;
1256 }
1257
1258 char * _ui64toaKAZEzerocomma (
1259     unsigned long long val,
1260     char *buf,
1261     int radix
1262 )
1263 {
1264     char *p;
1265     char temp;
1266     int txpman;
1267     int pxnman;
1268     x64toaKAZE(val, buf, radix, 0);
1269     p = buf;
1270     do {
1271     } while (*++p != '\0');
1272     p--; // p points to last digit
1273         // buf points to first digit
1274     buf[26] = 0;
1275     txpman = 1;
1276     pxnman = 0;
1277     do
1278     { if (buf <= p)
1279       { temp = *p;
1280         buf[26-txpman] = temp; pxnman++;
1281         p--;
1282         if (pxnman % 3 == 0)
1283         { txpman++;
1284           buf[26-txpman] = (char) (',' );
1285         }
1286       }
1287     else
1288     { buf[26-txpman] = (char) ('0'); pxnman++;
1289       if (pxnman % 3 == 0)
1290       { txpman++;
1291         buf[26-txpman] = (char) (',' );
1292       }
1293     }
1294     txpman++;
1295     } while (txpman <= 26);
1296     return buf;
1297 }
1298
1299 char * _ui64toaKAZEcomma (
1300     unsigned long long val,

```



```

1301     char *buf,
1302     int radix
1303 )
1304 {
1305     char *p;
1306     char temp;
1307     int txpman;
1308     int pxnman;
1309     x64toaKAZE(val, buf, radix, 0);
1310     p = buf;
1311     do {
1312     } while (*++p != '\0');
1313     p--; // p points to last digit
1314         // buf points to first digit
1315     buf[26] = 0;
1316     txpman = 1;
1317     pxnman = 0;
1318     while (buf <= p)
1319     { temp = *p;
1320       buf[26-txpman] = temp; pxnman++;
1321       p--;
1322       if (pxnman % 3 == 0 && buf <= p)
1323       { txpman++;
1324         buf[26-txpman] = (char) ('.');
1325       }
1326       txpman++;
1327     }
1328     return buf+26-(txpman-1);
1329 }
1330
1331 unsigned char KuxHash(char *str)
1332 { unsigned char h = 0;
1333   int max31 = 0;
1334   //while (*str)
1335   while (str[max31])
1336   { h = h ^ str[max31++];
1337     //h = h ^ *str++; // I am not sure 'str' is returned changed after return?!
1338   }
1339   return h; // 00..255 i.e. 2^8=256
1340 }
1341
1342 int KuxHash2(char *str)
1343 { int h = 0;
1344   unsigned long h2 = 0; // must be long: 31*'z'=31*122
1345   int max31 = 0;
1346   while (str[max31])
1347   { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1348     //h2 = h2 + str[max31++]; // [113s]
1349     h2 = h2 + max31 * str[max31++];
1350   }
1351   h=h<<4; // 00..15 i.e. 2^4=16
1352   //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
1353   h = h|( h2%((1<<4)-1) );
1354   return h; // 00..4095 i.e. 2^12=4096
1355 }
1356
1357 //OSHO test - Attempts to Find/Put a WORD into linked list count: 32,011,937
1358 int KuxHash3(char *str)
1359 { int h = 0;
1360   unsigned long h2 = 0; // must be long: 31*'z'=31*122
1361   int max31 = 0;
1362   while (str[max31])
1363   { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1364     //h2 = h2 + str[max31++]; // [113s]
1365     h2 = h2 + str[max31++] * (max31+1);
1366   }
1367   // Result is: 7bits in 'h' and 32bits in 'h2'.
1368
1369     //printf("%s:\n ", str);
1370     //printf("%d ", h);
1371   h=h<<6; // 00..15 i.e. 00-05+7bits=13bits
1372     //printf("%d ", h);
1373     //printf("%d ", h2);
1374   //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
1375   h = h|( h2%((1<<6)-1) ); // 64-1=63=9*7; 61 is prime
1376     //printf("%d \n", h);
1377   return h; // 00..8191 i.e. 2^13=8192
1378 }
1379
1380 //OSHO test - Attempts to Find/Put a WORD into a linked list count: 31,927,285
1381 int KuxHash3plus(char *str)
1382 { int h = 0;
1383   unsigned long h2 = 0; // must be long: 31*'z'=31*122
1384   int max31 = 0;
1385   while (str[max31])
1386   { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1387     //h2 = h2 + str[max31++]; // [113s]
1388     h2 = h2 + str[max31++] * (max31+1);

```

```

1389 }
1390 // Result is: 7bits in 'h' and 32bits in 'h2'.
1391
1392 //printf("%s:\n",str);
1393 //printf("%d ",h);
1394 // a in ASCII is 097 = 0110 0001
1395 // z in ASCII is 122 = 0111 1010
1396 // Above two lines show that bits 8-7-6 are always 0-1-1 so need for low 5 bits.
1397 //h=h<<8; // 00..15 i.e. 5bits + 00-07bits=13bits
1398 //printf("%d ",h);
1399 //printf("%d ",h2);
1400 //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
1401 h = (( h<<8 )|( h2%(251) ))&8191; // 251 prime
1402 //printf("%d \n",h);
1403 return h; // 00..8191 i.e. 2^13=8192
1404 }
1405
1406 /*
1407 PUBLIC _KuxHash3plus
1408 ; Function compile flags: /Ogt
1409 _TEXT SEGMENT
1410 _str$ = 8 ; size = 4
1411 _KuxHash3plus PROC NEAR
1412 ; Line 511
1413 mov ecx, DWORD PTR _str$(esp-4)
1414 mov dl, BYTE PTR [ecx]
1415 pushesi
1416 xor esi, esi
1417 xor eax, eax
1418 test dl, dl
1419 je SHORT $L1561
1420 pushebx
1421 pushedi
1422 mov edi, 1
1423 sub edi, ecx
1424 npad8
1425 $L1560:
1426 ; Line 512
1427 movsx edx, BYTE PTR [ecx]
1428 ; Line 514
1429 lea ebx, DWORD PTR [edi+ecx]
1430 imul ebx, edx
1431 xor esi, edx
1432 mov dl, BYTE PTR [ecx+1]
1433 add eax, ebx
1434 inc ecx
1435 test dl, dl
1436 jne SHORT $L1560
1437 pop edi
1438 pop ebx
1439 $L1561:
1440 ; Line 527
1441 xor edx, edx
1442 mov ecx, 251 ; 000000fbH
1443 div ecx
1444 shl esi, 8
1445 mov eax, edx
1446 ; Line 529
1447 or eax, esi
1448 and eax, 8191 ; 00001fffH
1449 pop esi
1450 ; Line 530
1451 ret 0
1452 _KuxHash3plus ENDP
1453 _TEXT ENDS
1454 */
1455
1456 //OSHO test - Attempts to Find/Put a WORD into a linked list count: 32,021,975
1457 int KuxHash4(char *str)
1458 {
1459 int h2 = 0;
1460 for (; *str != 0; str++) {
1461 //h2 = (127*h2 + *str) % (8192-1); // 2^13-1 = 8191 is Mersenne prime
1462 h2 = ((h2<<7) + *str) % (8192-1); // 2^13-1 = 8191 is Mersenne prime
1463 }
1464
1465 return h2; // 00..8191 i.e. 2^13=8192
1466 }
1467
1468 /*
1469 int hash(char *v, int M)
1470 { int h = 0, a = 127;
1471 for (; *v != 0; v++)
1472 h = (a*h + *v) % M;
1473 return h;
1474 }
1475
1476 int hashU(char *v, int M)

```

```

1477 { int h, a = 31415, b = 27183;
1478     for (h = 0; *v != 0; v++, a = a*b % (M-1))
1479         h = (a*h + *v) % M;
1480     return (h < 0) ? (h + M) : h;
1481 }
1482 */
1483
1484 // Kaze: My appreciation of FNV is far beyond C code optimization, it is alchemical, and why not, magical.
1485
1486 /*
1487 FNV hash history
1488     The basis of the FNV hash algorithm was taken from an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee
1489     by Glenn Fowler and Phong Vo back in 1991. In a subsequent ballot round: Landon Curt Noll improved on their algorithm.
1490     Some people tried this hash and found that it worked rather well. In an EMail message to Landon, they named it
1491     the ``Fowler/Noll/Vo'' or FNV hash.
1492     FNV hashes are designed to be fast while maintaining a low collision rate. The FNV speed allows one to quickly hash
1493     lots of data while maintaining a reasonable collision rate. The high dispersion of the FNV hashes makes them well suited
1494     for hashing nearly identical strings such as URLs, hostnames, filenames, text, IP addresses, etc.
1495 */
1496
1497 /* NOTE: u_int64_t is a 64 bit unsigned type */
1498 /* NOTE: u_int32_t is a 32 bit unsigned type */
1499 /* NOTE: u_int16_t is a 16 bit unsigned type */
1500 /* NOTE: u_int8_t is a 8 bit unsigned type */
1501
1502 //typedef unsigned char u_int8_t; //FNV only
1503 //typedef unsigned long u_int32_t; //FNV only
1504 //typedef unsigned long long u_int64_t; //FNV only
1505
1506 // 32 bit FNV_prime = 2^24 + 2^8 + 0x93 = 16777619
1507 // 64 bit FNV_prime = 2^40 + 2^8 + 0xb3 = 1099511628211
1508
1509 // 32 bit offset_basis = 2166136261
1510 // 64 bit offset_basis = 14695981039346656037
1511
1512 #define FNV1_64_INIT ((u_int64_t)14695981039346656037)
1513 #define FNV1_64_PRIME ((u_int64_t)1099511628211)
1514 #define FNV1_32_INIT ((u_int32_t)2166136261)
1515 #define FNV1_32_PRIME ((u_int32_t)602173697)
1516 // FNV1A_Hash_4_OCTETS gives dispersion as follows:
1517 //3549448: 1607
1518 //3549669: 171072511
1519 //3550710: 272961023
1520 //3550733: 172783361
1521 //3550734: 431562497
1522 //3550929: 204312319
1523 //3550984: 562853633
1524 //3550991: 551362303
1525 //3551359: 332820737
1526 //3551484: 354126079
1527 //3551514: 407138561
1528 //3551523: 442058753
1529 //3551701: 449230849
1530 //3551736: 107712257
1531 //3551961: 428904191
1532 //3552039: 602173697
1533 //3552103: 588411137
1534
1535 #define FNV_64A_OP(hash, octet) \
1536     (((u_int64_t)(hash) ^ (u_int8_t)(octet)) * FNV1_64_PRIME)
1537
1538 #define FNV_64A_OP64(hash, octet) \
1539     (((u_int64_t)(hash) ^ (u_int64_t)(octet)) * FNV1_64_PRIME)
1540
1541 #define FNV_32A_OP_GENERIC(hash, octet) \
1542     (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * 16777619)
1543
1544 #define FNV_32A_OP(hash, octet) \
1545     (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * FNV1_32_PRIME)
1546
1547 #define FNV_32A_OP_MULTiless_core(hash, octet) \
1548     ( (u_int32_t)(hash) ^ (u_int8_t)(octet) )
1549
1550 #define FNV_32A_OP_MULTiless(hash, octet) \
1551     ( (FNV_32A_OP_MULTiless_core(hash, octet)<<5) - FNV_32A_OP_MULTiless_core(hash, octet) )
1552
1553 #define FNV_32A_OP32(hash, octet) \
1554     (((u_int32_t)(hash) ^ (u_int32_t)(octet)) * FNV1_32_PRIME)
1555
1556 #define FNV_32A_OP64(hash, octet) \
1557     (((u_int64_t)(hash) ^ (u_int64_t)(octet)) * FNV1_32_PRIME)
1558
1559 #define FNV_32A_OP32_MULTiless_core(hash, octet) \
1560     ( (u_int32_t)(hash) ^ (u_int32_t)(octet) )
1561
1562 #define FNV_32A_OP32_MULTiless(hash, octet) \
1563     ( (FNV_32A_OP32_MULTiless_core(hash, octet)<<5) - FNV_32A_OP32_MULTiless_core(hash, octet) )
1564

```

```

1565
1566 // Invoking: FNV1A_Hash_4_OCTETS_31(wrd, wrdlen>>2) // = 0,1,2,3,4,5,6,7 [1..31]
1567 int FNV1A_Hash_4_OCTETS_31(char *str, int wrdlen_QUADRUPLTS)
1568 {
1569     u_int32_t hash;
1570     char *p;
1571
1572     hash = FNV1_32_INIT;
1573     p=str;
1574
1575     // The goal of stage #1: to reduce number of 'imul's in fact to reduce loops.
1576
1577     // Stage #1:
1578     for (; wrdlen_QUADRUPLTS != 0; --wrdlen_QUADRUPLTS) {
1579         hash = FNV_32A_OP32_MULLess(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
1580         p=p+4; // add eax, 4
1581     }
1582
1583     // Stage #2:
1584     for (; *p; ++p) {
1585         hash = FNV_32A_OP_MULLess(hash, *p); // mov dl, BYTE PTR [ecx]
1586     }
1587
1588     //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1589     return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1590 }
1591
1592
1593 // Invoking: FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2) // = 0,1,2,3,4,5,6,7 [1..31]
1594 int FNV1A_Hash_4_OCTETS(char *str, int wrdlen_QUADRUPLTS)
1595 {
1596     u_int32_t hash;
1597     char *p;
1598
1599     hash = FNV1_32_INIT;
1600     p=str;
1601
1602     // The goal of stage #1: to reduce number of 'imul's.
1603
1604     // Stage #1:
1605     for (; wrdlen_QUADRUPLTS != 0; --wrdlen_QUADRUPLTS) {
1606         hash = FNV_32A_OP32(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
1607         p=p+4; // add eax, 4
1608     }
1609
1610     // Stage #2:
1611     for (; *p; ++p) {
1612         hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [ecx]
1613     }
1614
1615     //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1616     return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1617 }
1618
1619 /*
1620 Results for 'FNV1A_Hash_8_OCTETS':
1621 Bytes per second performance: 23,110,160B/s
1622 Words per second performance: 1,959,516W/s
1623 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
1624 Size of all TEXTual Files: 415,982,896
1625 Word count: 35,271,297 of them 22,202,980 distinct
1626 Number Of Files: 8
1627 Number Of Lines: 35271297
1628 Allocated memory in MB: 1950
1629 Number Of Trees(GREATER THE BETTER): 3419429
1630 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 51%
1631 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18783551
1632 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '1,119'
1633 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 268,085,505
1634 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 7,690,615
1635 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 2,622 must have PEAK = 12 = rounding down of integer (1+lb(2,622))
1636 Binary-Search-Tree(1st out of 1) with MaxNODEs = 2,622 has PEAK = 592 and LEAFs = 689
1637 Binary-Search-Tree(1st out of 1) with MaxPEAK = '1,119' has NODEs = 1,537 and LEAFs = 287
1638 Binary-Search-Tree(1st out of 1) with MaxLEAFs = 731 has NODEs = 2,517 and PEAK = 448
1639 */
1640 // Invoking: FNV1A_Hash_8_OCTETS(wrd, wrdlen>>3) // = 0,1,2,3 [1..31]
1641 int FNV1A_Hash_8_OCTETS(char *str, int wrdlen_OCTETS)
1642 {
1643     u_int32_t hash;
1644     char *p;
1645
1646     hash = FNV1_32_INIT;
1647     p=str;
1648
1649     // The goal of stage #1: to reduce number of 'imul's.
1650
1651     // Stage #1:
1652     for (; wrdlen_OCTETS != 0; --wrdlen_OCTETS) {

```

```

1653     hash = FNV_32A_OP64(hash, (unsigned long long)*(long *)p); // mov edi, DWORD PTR [eax]
1654     p=p+8; // add eax, 4
1655 }
1656
1657 // Stage #2:
1658 for (; *p; ++p) {
1659     hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [ecx]
1660 }
1661
1662 //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1663 return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1664 }
1665
1666
1667 // Invoking: FNV1A_Hash_Granularity(wrd, wrdlen>>0|2|3, 0|2|3)
1668 int FNV1A_Hash_Granularity(char *str, int wrdlen_granulated, int Granularity) // wrdlen>>0=wrdlen
1669 {
1670     u_int32_t hash;
1671     u_int64_t hash64;
1672     char *p;
1673
1674     hash = FNV1_32_INIT;
1675     p=str;
1676
1677     // The goal of stage #1: to reduce number of 'imul's and mainly: the number of loops.
1678
1679     // Stage #1:
1680     if (Granularity == 2) {
1681         for (; wrdlen_granulated != 0; --wrdlen_granulated) {
1682             hash = FNV_32A_OP32(hash, (u_int32_t)*(u_int32_t *)p);
1683             p=p+4; // (1<<Granularity): 1<<0=1, 1<<2=4, 1<<3=8
1684         }
1685     }
1686     if (Granularity == 3) {
1687         hash64 = FNV1_64_INIT;
1688         for (; wrdlen_granulated != 0; --wrdlen_granulated) {
1689             hash64 = FNV_64A_OP64(hash64, (u_int64_t)*(u_int64_t *)p);
1690             p=p+8; // (1<<Granularity): 1<<0=1, 1<<2=4, 1<<3=8
1691         }
1692         for (; *p; ++p) {
1693             hash64 = FNV_64A_OP(hash64, (u_int8_t)*(u_int8_t *)p);
1694         }
1695
1696         //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1697         return ((hash64>>51) ^ hash64) & 8191; // 00..8191 i.e. 2^13=8192
1698         // probably better shifting is not by 16 bits but ...
1699         //hash64>>16: 3,544,160 just bad
1700         //hash64>>33: 3,547,854
1701         //hash64>>34: 3,547,266
1702         //hash64>>35: 3,547,453
1703         //hash64>>36: 3,547,242
1704         //hash64>>40: 3,548,263
1705         //hash64>>44: 3,548,242
1706         //hash64>>45: 3,549,056
1707         //hash64>>46: 3,549,207
1708         //hash64>>47: 3,549,094
1709         //hash64>>50: 3,549,392
1710         //hash64>>51: 3,549,395 i.e. maximum shift: the 13 most significant bits i.e. (64-13); closest to 3,549,448
1711
1712         // Above results are obtained for following set:
1713         //if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
1714         //    Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>2, 2)<<2; //13++++
1715         //else // 2x8+4=20 i.e. first contains 6 clashes
1716         //    Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>3, 3)<<2; //13++++
1717     }
1718
1719     //if (Granularity != 3) {
1720     // Stage #2:
1721     for (; *p; ++p) {
1722         hash = FNV_32A_OP(hash, (u_int8_t)*(u_int8_t *)p);
1723     }
1724
1725     //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1726     return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1727     //}
1728 }
1729
1730
1731 // char *string; /* the string to 64 bit FNV-1a hash */
1732 // u_int64_t hash; /* will hold the final value of the hash */
1733 // char *p;
1734 //
1735 // hash = FNV1_64_INIT;
1736 // for (p=string; *p; ++p) {
1737 //     hash = FNV_64A_OP(hash, *p);
1738 // }
1739
1740

```

```

1741 // If you need an x-bit hash where x is not a power of 2,
1742 // then we recommend that you compute the FNV hash that is just larger than x-bits and xor-fold the result down to x-bits.
1743 // By xor-folding we mean shift the excess high order bits down and xor them with the lower x-bits.
1744 // For tiny x < 16 bit values, we recommend using a 32 bit FNV-1 hash as follows:
1745
1746 // /* NOTE: for 0 < x < 16 ONLY!!! */
1747 // #define TINY_MASK(x) (((u_int32_t)1<<(x))-1)
1748 // #define FNV1_32_INIT ((u_int32_t)2166136261)
1749 // u_int32_t hash;
1750 // void *data;
1751 // size_t data_len;
1752 //
1753 // hash = fnv_32_buf(data, data_len, FNV1_32_INIT);
1754 // hash = (((hash>>x) ^ hash) & TINY_MASK(x));
1755
1756
1757 int FNV1A_Hash_SHIFTless_XORless(char *str)
1758 {
1759     u_int32_t hash; /* will hold the final value of the hash */
1760     char *p;
1761
1762     hash = FNV1_32_INIT;
1763     for (p=str; *p; ++p) {
1764         hash = FNV_32A_OP(hash, *p);
1765     }
1766     //hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1767
1768     return hash & 8191; // 00..8191 i.e. 2^13=8192
1769 }
1770
1771 /*
1772 _FNV1A_Hash_SHIFTless_XORless PROC NEAR
1773 ; Line 721
1774 mov edx, DWORD PTR _str$[esp-4]
1775 mov cl, BYTE PTR [edx]
1776 test cl, cl
1777 mov eax, -2128831035 ; 811c9dc5H
1778 je SHORT $L1582
1779 npad 1
1780 $L1580:
1781 ; Line 722
1782 movzx ecx, cl
1783 xor ecx, eax
1784 imul ecx, 16777619 ; 01000193H
1785 inc edx
1786 mov eax, ecx
1787 mov cl, BYTE PTR [edx]
1788 test cl, cl
1789 jne SHORT $L1580
1790 $L1582:
1791 ; Line 726
1792 and eax, 8191 ; 00001fffH
1793 ; Line 727
1794 ret 0
1795 _FNV1A_Hash_SHIFTless_XORless ENDP
1796 */
1797
1798
1799 int FNV1A_Hash(char *str)
1800 {
1801     u_int32_t hash; /* will hold the final value of the hash */
1802     char *p;
1803
1804     hash = FNV1_32_INIT;
1805     for (p=str; *p; ++p) {
1806         hash = FNV_32A_OP(hash, *p);
1807     }
1808     //hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1809
1810     return ((hash>>13) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1811 }
1812
1813 /*
1814 _FNV1A_Hash PROC NEAR
1815 ; Line 722
1816 mov edx, DWORD PTR _str$[esp-4]
1817 mov al, BYTE PTR [edx]
1818 test al, al
1819 mov ecx, -2128831035 ; 811c9dc5H
1820 je SHORT $L1582
1821 npad 1
1822 $L1580:
1823 ; Line 723
1824 movzx eax, al
1825 xor eax, ecx
1826 imul eax, 16777619 ; 01000193H
1827 inc edx
1828 mov ecx, eax

```

```

1829 mov al, BYTE PTR [edx]
1830 test al, al
1831 jne SHORT $L1580
1832 $L1580:
1833 ; Line 727
1834 mov eax, ecx
1835 shr eax, 13 ; 0000000dH
1836 xor eax, ecx
1837 and eax, 8191 ; 00001ffffH
1838 ; Line 728
1839 ret 0
1840 _FNV1A_Hash ENDP
1841 */
1842
1843 /*
1844 Wayne Diamond implemented 32-bit FNV algorithm in PowerBASIC inline x86 assembly:
1845
1846
1847 FUNCTION FNV32(BYVAL dwOffset AS DWORD, BYVAL dwLen AS DWORD, BYVAL offset_basis AS DWORD) AS DWORD
1848 #REGISTER NONE
1849 ! mov esi, dwOffset ; esi = ptr to buffer
1850 ! mov ecx, dwLen ; ecx = length of buffer (counter)
1851 ! mov eax, offset_basis ; set to 2166136261 for FNV-1
1852 ! mov edi, &h01000193 ; FNV_32_PRIME = 16777619
1853 ! xor ebx, ebx ; ebx = 0
1854 nextbyte:
1855 ! mul edi ; eax = eax * FNV_32_PRIME
1856 ! mov bl, [esi] ; bl = byte from esi
1857 ! xor eax, ebx ; al = al xor bl
1858 ! inc esi ; esi = esi + 1 (buffer pos)
1859 ! dec ecx ; ecx = ecx - 1 (counter)
1860 ! jnz nextbyte ; if ecx is 0, jmp to NextByte
1861 ! mov FUNCTION, eax ; else, function = eax
1862 END FUNCTION
1863
1864 Wayne said:
1865
1866 ''Just thought I should let you know that I've ported the 32-bit FNV algorithm over to inline assembly.
1867 It's actually in PowerBASIC (www.powerbasic.com) format - a compiler I use, but the main function is all assembly.
1868 It could be optimized further in terms of saving a couple of clock cycles,
1869 but it's fairly optimized al ready - only 6 instructions in the main loop, plus 5 setup instructions,
1870 and compiles to just 33 bytes.''
1871
1872 M.S.Schulte sent us these 32-bit FNV-1 and FNV-1a x86 assembler implementations (written in flat assembler),
1873 half of which were optimized for speed, the other half were optimized for size:
1874
1875 small_fnv32: ;FNV1 32bit (size: 31 bytes)
1876 ; Intel Core 2 Duo E6600: 354.20 mb/s
1877 push esi
1878 push edi
1879 mov esi, [esp + 0ch] ; buffer
1880 mov ecx, [esp + 10h] ; length
1881 mov eax, [esp + 14h] ; basis
1882 mov edi, 01000193h ; fnv_32_prime
1883 next:
1884 mul edi
1885 xor al, [esi]
1886 inc esi
1887 loop snext
1888 pop edi
1889 pop esi
1890 retn 0ch
1891
1892 small_fnv32a: ;FNV1a 32bit (size: 31 bytes)
1893 ; Intel Core 2 Duo E6600: 327.68 mb/s
1894 push esi
1895 push edi
1896 mov esi, [esp + 0ch] ; buffer
1897 mov ecx, [esp + 10h] ; length
1898 mov eax, [esp + 14h] ; basis
1899 mov edi, 01000193h ; fnv_32_prime
1900 nexta:
1901 xor al, [esi]
1902 mul edi
1903 inc esi
1904 loop nexta
1905 pop edi
1906 pop esi
1907 retn 0ch
1908
1909 fast_fnv32: ;FNV1 32bit (size: 36 bytes)
1910 ; Intel Core 2 Duo E6600: 565.12 mb/s
1911 push ebx
1912 push esi
1913 push edi
1914 mov esi, [esp + 10h] ; buffer
1915 mov ecx, [esp + 14h] ; length
1916 mov eax, [esp + 18h] ; basis

```

```

1917     mov     edi, 01000193h ; fnv_32_prime
1918     xor     ebx, ebx
1919 next:
1920     mul     edi
1921     mov     bl, [esi]
1922     xor     eax, ebx
1923     inc     esi
1924     dec     ecx
1925     jnz     next
1926     pop     edi
1927     pop     esi
1928     pop     ebx
1929     retn    0ch
1930
1931     fast_fnv32a: ; FNV1a 32bit (size: 36 bytes)
1932 ;             Intel Core 2 Duo E6600: 574.95 mb/s
1933     push    ebx
1934     push    esi
1935     push    edi
1936     mov     esi, [esp + 10h] ; buffer
1937     mov     ecx, [esp + 14h] ; length
1938     mov     eax, [esp + 18h] ; basis
1939     mov     edi, 01000193h ; fnv_32_prime
1940     xor     ebx, ebx
1941 nexta:
1942     mov     bl, [esi]
1943     xor     eax, ebx
1944     mul     edi
1945     inc     esi
1946     dec     ecx
1947     jnz     nexta
1948     pop     edi
1949     pop     esi
1950     pop     ebx
1951     retn    0ch
1952 */
1953
1954 //Number Of Trees(GREATER THE BETTER): 3525737
1955 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1956 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18677243
1957 int Hash17_unrolled(const char *key, int wrdlen)
1958 {
1959     int hash = 1;
1960     int i;
1961     for(i = 0; i < (wrdlen & -2); i += 2) {
1962         hash = (17) * hash + (key[i] - ' ');
1963         hash = (17) * hash + (key[i+1] - ' ');
1964     }
1965     if(wrdlen & 1)
1966         hash = (17) * hash + (key[wrdlen-1] - ' ');
1967     return ( hash ^ (hash >> 16) ) & 8191;
1968 }
1969
1970 //hash = 1:
1971 //Number Of Trees(GREATER THE BETTER): 3556516
1972 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1973 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18646464
1974 //hash = 13:
1975 //Number Of Trees(GREATER THE BETTER): 3556755
1976 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1977 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18646225
1978 //hash = 11:
1979 //Number Of Trees(GREATER THE BETTER): 3557011
1980 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1981 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18645969
1982 //hash = 7:
1983 //Number Of Trees(GREATER THE BETTER): 3557181
1984 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1985 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18645799
1986 int Alfa17(const char *key, int wrdlen)
1987 {
1988     int hash = 7;
1989     int i;
1990     for(i = 0; i < (wrdlen & -2); i += 2) {
1991         hash = (17+9) * ((17+9) * hash + (key[i])) + (key[i+1]);
1992     }
1993     if(wrdlen & 1)
1994         hash = (17+9) * hash + (key[wrdlen-1]);
1995     return ( hash ^ (hash >> 16) ) & 8191;
1996 }
1997
1998 /*
1999 [FNV1A 'shift-less-&-xor-less' hash used in Leprechaun r.13++:]
2000
2001 int FNV1A_Hash_SHIFTless_XORless(char *str)
2002 {
2003     u_int32_t hash;
2004     char *p;

```



```

2005
2006 hash = FNV1_32_INIT;
2007 for (p=str; *p; ++p) {
2008     hash = FNV_32A_OP(hash, *p);
2009 }
2010 //hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
2011
2012 return hash & 8191; // 00..8191 i.e. 2^13=8192
2013 }
2014
2015 Words per second performance: 837,458W/s
2016 Input File with a list of TEXTual Files: wikipedia-en-html.tar.wrd.lst
2017 Word count: 12,561,874 of them 12,561,874 distinct
2018 Number Of Trees(GREATER THE BETTER): 2772875
2019 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 41%
2020 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 9788999
2021
2022 Words per second performance: 1,007,751W/s
2023 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2024 Word count: 35,271,297 of them 22,202,980 distinct
2025 Number Of Trees(GREATER THE BETTER): 3537061
2026 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2027 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18665919
2028
2029 [My '2in1' hash used in Leprechaun r.13++]
2030
2031 int KuxHash3plus(char *str)
2032 { int h = 0;
2033   unsigned long h2 = 0; // must be long: 31*'z'=31*122
2034   int max31 = 0;
2035   while (str[max31])
2036   { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
2037     //h2 = h2 + str[max31++]; // [113s]
2038     h2 = h2 + str[max31++] * (max31+1);
2039   }
2040   // Result is: 7bits in 'h' and 32bits in 'h2'.
2041
2042   //printf("%s: \n ", str);
2043   //printf("%d ", h);
2044   // a in ASCII is 097 = 0110 0001
2045   // z in ASCII is 122 = 0111 1010
2046   // Above two lines show that bits 8-7-6 are always 0-1-1 so need for low 5 bits.
2047   //h=h<<8; // 00..15 i.e. 5bits + 00-07bits=13bits
2048   //printf("%d ", h);
2049   //printf("%d ", h2);
2050   //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
2051   h = (( h<<8 )|( h2%(251) ))&8191; // 251 prime
2052   //printf("%d \n", h);
2053   return h; // 00..8191 i.e. 2^13=8192
2054 }
2055
2056 Words per second performance: 785,117W/s
2057 Input File with a list of TEXTual Files: wikipedia-en-html.tar.wrd.lst
2058 Word count: 12,561,874 of them 12,561,874 distinct
2059 Number Of Trees(GREATER THE BETTER): 2663566
2060 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 40%
2061 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 9898308
2062
2063 Words per second performance: 979,758W/s
2064 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2065 Word count: 35,271,297 of them 22,202,980 distinct
2066 Number Of Trees(GREATER THE BETTER): 3410463
2067 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 51%
2068 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18792517
2069
2070 [Last standing for English(en)-Wikipedia's wordlist:]
2071 chongo's hash is faster(in total, not the function itself) than Kaze's hash by ((837,458W/s - 785,117W/s)/785,117W/s)*100% = 6.6%
2072 chongo's hash has better distribution than Kaze's hash by ((9898308 - 9788999)/9788999)*100% = 1.1%
2073
2074 [Last standing for LATIN(de,en,es,fr,it,nl,pt,ro)-Wikipedia's wordlist:]
2075 chongo's hash is faster(in total, not the function itself) than Kaze's hash by ((1,007,751W/s - 979,758W/s)/979,758W/s)*100% = 2.8%
2076 chongo's hash has better distribution than Kaze's hash by ((18792517 - 18665919)/18665919)*100% = 0.6%
2077
2078 Bottomline is:
2079 Your hash thrash, my hash for trash, he-he.
2080 Thanks a lot, again, Mr. Noll.
2081
2082 Yummy little file: http://www.isthe.com/chongo/src/fnv/fnv-5.0.2.tar.gz
2083 */
2084
2085 /*
2086 // Paul Larson (http://research.microsoft.com/~PALARSON/)
2087 UINT HashLarson(const CHAR *key, SIZE_T len) {
2088     UINT hash = 0;
2089     for(UINT i = 0; i < len; ++i)
2090         hash = 101 * hash + key[i];
2091     return hash ^ (hash >> 16);
2092 }

```

```

2093
2094 // Kernighan & Ritchie, "The C programming Language", 3rd edition.
2095 UINT HashKernighanRitchie(const CHAR *key, SIZE_T len) {
2096     UINT hash = 0;
2097     for(UINT i = 0; i < len; ++i)
2098         hash = 31 * hash + key[i];
2099     return hash;
2100 }
2101
2102 // A hash function with multiplier 65599 (from Red Dragon book)
2103 UINT Hash65599(const CHAR *key, SIZE_T len) {
2104     UINT hash = 0;
2105     for(UINT i = 0; i < len; ++i)
2106         hash = 65599 * hash + key[i];
2107     return hash ^ (hash >> 16);
2108 }
2109
2110 // FNV hash, http://isthe.com/chongo/tech/comp/fnv/
2111 UINT HashFNV1a(const CHAR *key, SIZE_T len) {
2112     UINT hash = 2166136261;
2113     for(UINT i = 0; i < len; ++i)
2114         hash = 16777619 * (hash ^ key[i]);
2115     return hash ^ (hash >> 16);
2116 }
2117
2118 // Ramakrishna hash
2119 UINT HashRamakrishna(const CHAR *key, SIZE_T len) {
2120     UINT h = 0;
2121     for(UINT i = 0; i < len; ++i) {
2122         h ^= (h << 5) + (h >> 2) + key[i];
2123     }
2124     return h;
2125 }
2126 */
2127
2128 /*
2129 Results for 'HashAlfa':
2130 Bytes per second performance: 19,808,709B/s
2131 Words per second performance: 1,679,585W/s
2132 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2133 Size of all TEXTual Files: 415,982,896
2134 Word count: 35,271,297 of them 22,202,980 distinct
2135 Number Of Files: 8
2136 Number Of Lines: 35271297
2137 Allocated memory in MB: 1950
2138 Number Of Trees(GREATER THE BETTER): 3549079
2139 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2140 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18653901
2141 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '37'
2142 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 117,063,824
2143 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,279
2144 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 84 must have PEAK = 7 = rounding down of integer (1+lb(84))
2145 Binary-Search-Tree(1st out of 2) with MaxNODEs = 84 has PEAK = 20 and LEAFs = 24
2146 Binary-Search-Tree(1st out of 3) with MaxPEAK = '37' has NODEs = 67 and LEAFs = 17
2147 Binary-Search-Tree(1st out of 1) with MaxLEAFs = 28 has NODEs = 78 and PEAK = 22
2148 */
2149 UINT HashAlfa(const char *key, unsigned int wrdlen)
2150 {
2151     UINT hash = 7;
2152     unsigned int i;
2153     for(i = 0; i < (wrdlen & -2); i += 2) {
2154         hash = (53) * ((53) * hash + (key[i])) + (key[i+1]);
2155     }
2156     if (wrdlen & 1)
2157         hash = (53) * hash + (key[wrdlen-1]);
2158     return ((hash>>16) ^ hash) & 8191;
2159 }
2160
2161 /*
2162 Results for 'HashAlfa_HALF':
2163 Bytes per second performance: 19,808,709B/s
2164 Words per second performance: 1,679,585W/s
2165 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2166 Size of all TEXTual Files: 415,982,896
2167 Word count: 35,271,297 of them 22,202,980 distinct
2168 Number Of Files: 8
2169 Number Of Lines: 35271297
2170 Allocated memory in MB: 1950
2171 Number Of Trees(GREATER THE BETTER): 3550665
2172 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2173 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18652315
2174 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '39'
2175 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 117,053,918
2176 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,259
2177 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 87 must have PEAK = 7 = rounding down of integer (1+lb(87))
2178 Binary-Search-Tree(1st out of 1) with MaxNODEs = 87 has PEAK = 21 and LEAFs = 27
2179 Binary-Search-Tree(1st out of 2) with MaxPEAK = '39' has NODEs = 65 and LEAFs = 18
2180 Binary-Search-Tree(1st out of 4) with MaxLEAFs = 27 has NODEs = 77 and PEAK = 23

```

```

2181 */
2182 UINT HashAlfa_HALF(const char *key, unsigned int wrdlen)
2183 {
2184     UINT hash = 12;
2185     UINT hashBUFFER;
2186     unsigned int i,j;
2187     for(i = 0; i < (wrdlen & -4); i += 4) {
2188         //hash = (( (hash<<5)-hash) + key[i] )<<5) - ( ((hash<<5)-hash) + key[i] ) + (key[i+1]);
2189         hashBUFFER = ((hash<<5)-hash) + key[i];
2190         hash = (( hashBUFFER )<<5) - ( hashBUFFER ) + (key[i+1]);
2191         //hash = (( (hash<<5)-hash) + key[i+2] )<<5) - ( ((hash<<5)-hash) + key[i+2] ) + (key[i+3]);
2192         hashBUFFER = ((hash<<5)-hash) + key[i+2];
2193         hash = (( hashBUFFER )<<5) - ( hashBUFFER ) + (key[i+3]);
2194     }
2195     for(j = 0; j < (wrdlen & 3); j += 1) {
2196         hash = ((hash<<5)-hash) + key[i+j];
2197     }
2198     return ((hash>>16) ^ hash) & 8191;
2199 }
2200
2201 /*
2202 Results for 'HashFNV1A_unrolled_Final':
2203 Bytes per second performance: 19,808,709B/s
2204 Words per second performance: 1,679,585W/s
2205 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2206 Size of all TEXTual Files: 415,982,896
2207 Word count: 35,271,297 of them 22,202,980 distinct
2208 Number Of Files: 8
2209 Number Of Lines: 35271297
2210 Allocated memory in MB: 1950
2211 Number Of Trees(GREATER THE BETTER): 3445337
2212 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 52%
2213 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18757643
2214 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '43'
2215 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 118,349,998
2216 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 7,997,033
2217 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 89 must have PEAK = 7 = rounding down of integer (1+lb(89))
2218 Binary-Search-Tree(1st out of 1) with MaxNODEs = 89 has PEAK = 28 and LEAFs = 28
2219 Binary-Search-Tree(1st out of 1) with MaxPEAK = '43' has NODEs = 65 and LEAFs = 11
2220 Binary-Search-Tree(1st out of 2) with MaxLEAFs = 28 has NODEs = 78 and PEAK = 24
2221 */
2222 UINT HashFNV1A_unrolled_Final(char *str, unsigned int wrdlen)
2223 {
2224     //const UINT PRIME = 31;
2225     unsigned int hash = 2166136261;
2226     char *p = str;
2227
2228     /*
2229     // Reduce the number of multiplications by unrolling the loop
2230     for (SIZE_T ndwords = wrdlen / sizeof(DWORD); ndwords; --ndwords) {
2231         //hash = (hash ^ *(DWORD*)p) * PRIME;
2232         hash = ((hash ^ *(DWORD*)p)<<5) - (hash ^ *(DWORD*)p);
2233     }
2234     p += sizeof(DWORD);
2235     */
2236     /*
2237     for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
2238         hash = ((hash ^ *(unsigned int*)p)<<5) - (hash ^ *(unsigned int*)p);
2239     }
2240
2241     // Process the remaining bytes
2242     */
2243     for (SIZE_T i = 0; i < (wrdlen & (sizeof(DWORD) - 1)); i++) {
2244         //hash = (hash ^ *p++) * PRIME;
2245         hash = ((hash ^ *p)<<5) - (hash ^ *p);
2246         p++;
2247     }
2248     /*
2249     if (wrdlen & -2) {
2250         hash = ((hash ^ (*(unsigned int*)p&0xFFFF))<<5) - (hash ^ (*(unsigned int*)p&0xFFFF));
2251         p++;p++;
2252     }
2253     if (wrdlen & 1)
2254         hash = ((hash ^ *p)<<5) - (hash ^ *p);
2255
2256     return ((hash>>16) ^ hash) & 8191;
2257     */
2258
2259     /*
2260     Results for 'Sixtinsensitive':
2261     Bytes per second performance: 19,808,709B/s
2262     Words per second performance: 1,679,585W/s
2263     Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2264     Size of all TEXTual Files: 415,982,896
2265     Word count: 35,271,297 of them 22,202,980 distinct
2266     Number Of Files: 8
2267     Number Of Lines: 35271297
2268     Allocated memory in MB: 1950

```

```

2269 Number Of Trees(GREATER THE BETTER): 3531949
2270 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2271 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18671031
2272 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '38'
2273 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 118,959,016
2274 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,047,983
2275 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 98 must have PEAK = 7 = rounding down of integer (1+lb(98))
2276 Binary-Search-Tree(1st out of 1) with MaxNODEs = 98 has PEAK = 36 and LEAFs = 30
2277 Binary-Search-Tree(1st out of 1) with MaxPEAK = '38' has NODEs = 54 and LEAFs = 11
2278 Binary-Search-Tree(1st out of 2) with MaxLEAFs = 30 has NODEs = 98 and PEAK = 36
2279 */
2280 // Tuned for lowercase-and-uppercase letters i.e. 26 ASCII symbols 65-90 and 97-122 decimal.
2281 UINT Sixtinsensitive(const char *str, unsigned int wrdlen)
2282 {
2283     UINT hash = 2166136261;
2284     UINT hashBUFFER_EAX, hashBUFFER_BH, hashBUFFER_BL;
2285     const char * p = str;
2286
2287     // 0x41 = 065 'A' 010 [0 0001]
2288     // 0x5A = 090 'Z' 010 [1 1010]
2289     // 0x61 = 097 'a' 011 [0 0001]
2290     // 0x7A = 122 'z' 011 [1 1010]
2291
2292     // Reduce the number of multiplications by unrolling the loop
2293     for(; wrdlen >= 6; wrdlen -= 6, p += 6) {
2294         //hashBUFFER_AX = (*(DWORD*)(p+0)&0xFFFF);
2295         hashBUFFER_EAX = (*(DWORD*)(p+0)&0x1F1F1F1F);
2296         hashBUFFER_BL = (*(p+4)&0x1F);
2297         hashBUFFER_BH = (*(p+5)&0x1F);
2298         //6bytes-in-4bytes or 48bits-to-30bits
2299         // Two times next:
2300         //3bytes-in-2bytes or 24bits-to-15bits
2301         //EAX BL BH
2302         //[5bit][3bit][5bit][3bit][5bit][3bit][5bit][3bit]
2303         // 5th[0..15] 13th[0..15]
2304         // BL lower 3 BL higher 2bits
2305         // OR or XOR no difference
2306         hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BL&0x07)<<5); // BL lower 3bits of 5bits
2307         hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BL&0x18)<<(2+8)); // BL higher 2bits of 5bits
2308         hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BH&0x07)<<(5+16)); // BH lower 3bits of 5bits
2309         hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BH&0x18)<<((2+8)+16)); // BH higher 2bits of 5bits
2310         //hash = (hash ^ hashBUFFER_EAX)*1607; //What a mess: <<7 becomes imul but <<5 not!
2311         hash = ((hash ^ hashBUFFER_EAX)<<5) - (hash ^ hashBUFFER_EAX);
2312         //1607: [2118599]
2313         // 127: [2121081]
2314         // 31: [2139242]
2315         // 17: [2150803]
2316         // 7: [2166336]
2317         // 5: [2183044]
2318         //8191: [2200477]
2319         // 3: [2205095]
2320         // 257: [2206188]
2321     }
2322     // Post-Variant #1:
2323     for(; wrdlen; wrdlen--, p++) {
2324         hash = ((hash ^ (*p&0x1F))<<5) - (hash ^ (*p&0x1F));
2325     }
2326     /*
2327     // Post-Variant #2:
2328     for(; wrdlen >= 2; wrdlen -= 2, p += 2) {
2329         hash = ((hash ^ (*(DWORD*)p&0xFFFF))<<5) - (hash ^ (*(DWORD*)p&0xFFFF));
2330     }
2331     if (wrdlen & 1)
2332         hash = ((hash ^ *p)<<5) - (hash ^ *p);
2333     */
2334     /*
2335     // Post-Variant #3:
2336     for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
2337         hash = ((hash ^ *(DWORD*)p)<<5) - (hash ^ *(DWORD*)p);
2338     }
2339     if (wrdlen & -2) {
2340         hash = ((hash ^ *(DWORD*)p&0xFFFF)<<5) - (hash ^ *(DWORD*)p&0xFFFF);
2341         p++;p++;
2342     }
2343     if (wrdlen & 1)
2344         hash = ((hash ^ *p)<<5) - (hash ^ *p);
2345     */
2346     return ((hash>>16) ^ hash) & 8191;
2347 }
2348
2349 /*
2350 #define FNV1_32_INIT ((UINT)2166136261)
2351 #define FNV1_32_PRIME ((UINT)1709)
2352
2353 #define FNV_32A_OP(hash, octet) \
2354     (((UINT)(hash) ^ (unsigned char)(octet)) * FNV1_32_PRIME)
2355
2356 #define FNV_32A_OP32(hash, octet) \

```

```

2357      (((UINT)(hash) ^ (UINT)(octet)) * FNV1_32_PRIME)
2358
2359  UINT FNV1A_Hash_WHI Z(const char *str, SIZE_T wrdlen)
2360  {
2361
2362  UINT hash32;
2363  const char *p;
2364
2365  hash32 = FNV1_32_INIT;
2366  p=str;
2367
2368  for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
2369  hash32 = FNV_32A_OP32(hash32, (UINT)*(UINT *)p);
2370  }
2371  if (wrdlen & -2) {
2372      hash32 = FNV_32A_OP32(hash32, *(UINT*)p&0xFFFF);
2373      p++;p++;
2374  }
2375  if (wrdlen & 1)
2376      hash32 = FNV_32A_OP(hash32, *p);
2377
2378  return hash32 ^ (hash32 >> 16);
2379  }
2380  */
2381
2382  /*
2383  Results for 'FNV1A_Hash_Jester':
2384  Bytes per second performance: 19,808,709B/s
2385  Words per second performance: 1,679,585W/s
2386  Input File with a list of TEXTual Files: Leprechaun_vs_Wi ki pedi a_LATIN-WORDS. l st
2387  Size of all TEXTual Files: 415,982,896
2388  Word count: 35,271,297 of them 22,202,980 distinct
2389  Number Of Files: 8
2390  Number Of Lines: 35271297
2391  Allocated memory in MB: 1950
2392  Number Of Trees(GREATER THE BETTER): 3537352
2393  Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2394  Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18665628
2395  Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '37'
2396  Total Attempts to Find/Put WORDs into Binary-Search-Trees: 117,243,563
2397  Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,063,361
2398  Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 87 must have PEAK = 7 = rounding down of integer (1+lb(87))
2399  Binary-Search-Tree(1st out of 2) with MaxNODEs = 87 has PEAK = 27 and LEAFs = 23
2400  Binary-Search-Tree(1st out of 1) with MaxPEAK = '37' has NODEs = 66 and LEAFs = 18
2401  Binary-Search-Tree(1st out of 3) with MaxLEAFs = 27 has NODEs = 84 and PEAK = 27
2402  */
2403  UINT FNV1A_Hash_Jester(const char *str, unsigned int wrdlen)
2404  {
2405  const UINT PRIME = 709607;
2406  UINT hash32 = 2166136261;
2407  const char *p = str;
2408
2409  // Idea comes from Igor Pavlov's 7zCRC, thanks.
2410  /*
2411  for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
2412      hash32 = (hash32 ^ *p) * PRIME;
2413  }
2414  */
2415  for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
2416      hash32 = (hash32 ^ *(DWORD *)p) * PRIME;
2417      hash32 = (hash32 ^ *(DWORD *) (p+4)) * PRIME;
2418  }
2419  // Cases: 0,1,2,3,4,5,6,7
2420  if (wrdlen & sizeof(DWORD)) {
2421      hash32 = (hash32 ^ *(DWORD*)p) * PRIME;
2422      p += sizeof(DWORD);
2423  }
2424  if (wrdlen & sizeof(WORD)) {
2425      hash32 = (hash32 ^ *(WORD*)p) * PRIME;
2426      p += sizeof(WORD);
2427  }
2428  if (wrdlen & 1)
2429      hash32 = (hash32 ^ *p) * PRIME;
2430
2431  return (hash32 ^ (hash32 >> 16)) & 8191;
2432  }
2433
2434  /*
2435  Results for 'FNV1A_Hash_Jesteress':
2436  Bytes per second performance: 19,808,709B/s
2437  Words per second performance: 1,679,585W/s
2438  Input File with a list of TEXTual Files: Leprechaun_vs_Wi ki pedi a_LATIN-WORDS. l st
2439  Size of all TEXTual Files: 415,982,896
2440  Word count: 35,271,297 of them 22,202,980 distinct
2441  Number Of Files: 8
2442  Number Of Lines: 35271297
2443  Allocated memory in MB: 1950
2444  Number Of Trees(GREATER THE BETTER): 3537293

```

```

2445 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2446 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18665687
2447 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '40'
2448 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 117,526,680
2449 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,051,512
2450 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 89 must have PEAK = 7 = rounding down of integer (1+lb(89))
2451 Binary-Search-Tree(1st out of 1) with MaxNODEs = 89 has PEAK = 25 and LEAFs = 23
2452 Binary-Search-Tree(1st out of 1) with MaxPEAK = '40' has NODEs = 49 and LEAFs = 8
2453 Binary-Search-Tree(1st out of 1) with MaxLEAFs = 28 has NODEs = 72 and PEAK = 21
2454 */
2455 #define ROL(x, n) (((x) << (n)) | ((x) >> (32-(n))))
2456 UINT FNV1A_Hash_Jesteress(const char *str, unsigned int wrdlen)
2457 {
2458     const UINT PRIME = 709607;
2459     UINT hash32 = 2166136261;
2460     const char *p = str;
2461
2462     // Idea comes from Igor Pavlov's 7zCRC, thanks.
2463     /*
2464     for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
2465         hash32 = (hash32 ^ *p) * PRIME;
2466     }
2467     */
2468     for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
2469         hash32 = (hash32 ^ (ROL(*(DWORD *)p, 5)^(DWORD *) (p+4))) * PRIME;
2470     }
2471     // Cases: 0, 1, 2, 3, 4, 5, 6, 7
2472     if (wrdlen & sizeof(DWORD)) {
2473         hash32 = (hash32 ^ *(DWORD *)p) * PRIME;
2474         p += sizeof(DWORD);
2475     }
2476     if (wrdlen & sizeof(WORD)) {
2477         hash32 = (hash32 ^ *(WORD *)p) * PRIME;
2478         p += sizeof(WORD);
2479     }
2480     if (wrdlen & 1)
2481         hash32 = (hash32 ^ *p) * PRIME;
2482
2483     return (hash32 ^ (hash32 >> 16)) & 8191;
2484 }
2485
2486 UINT FNV1A_Hash_Jesteress_27bit(const char *str, unsigned int wrdlen)
2487 {
2488     const UINT PRIME = 709607;
2489     UINT hash32 = 2166136261;
2490     const char *p = str;
2491
2492     // Idea comes from Igor Pavlov's 7zCRC, thanks.
2493     /*
2494     for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
2495         hash32 = (hash32 ^ *p) * PRIME;
2496     }
2497     */
2498     for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
2499         hash32 = (hash32 ^ (ROL(*(DWORD *)p, 5)^(DWORD *) (p+4))) * PRIME;
2500     }
2501     // Cases: 0, 1, 2, 3, 4, 5, 6, 7
2502     if (wrdlen & sizeof(DWORD)) {
2503         hash32 = (hash32 ^ *(DWORD *)p) * PRIME;
2504         p += sizeof(DWORD);
2505     }
2506     if (wrdlen & sizeof(WORD)) {
2507         hash32 = (hash32 ^ *(WORD *)p) * PRIME;
2508         p += sizeof(WORD);
2509     }
2510     if (wrdlen & 1)
2511         hash32 = (hash32 ^ *p) * PRIME;
2512
2513     return (hash32 ^ (hash32 >> 16)) & ((1<<HashInBITS)-1);
2514 }
2515
2516 /*
2517 UINT NextPowerOfTwo(UINT x) {
2518     // Henry Warren, "Hacker's Delight", ch. 3.2
2519     x--;
2520     x |= (x >> 1);
2521     x |= (x >> 2);
2522     x |= (x >> 4);
2523     x |= (x >> 8);
2524     x |= (x >> 16);
2525     return x + 1;
2526 }
2527
2528 UINT NextLog2(UINT x) {
2529     // Henry Warren, "Hacker's Delight", ch. 5.3
2530     if(x <= 1) return x;
2531     x--;
2532     UINT n = 0;

```

```

2533 unsigned int y;
2534 y = x >> 16; if(y) {n += 16; x = y;}
2535 y = x >> 8; if(y) {n += 8; x = y;}
2536 y = x >> 4; if(y) {n += 4; x = y;}
2537 y = x >> 2; if(y) {n += 2; x = y;}
2538 y = x >> 1; if(y) return n + 2;
2539 return n + x;
2540 }
2541 */
2542
2543 // The following example code in the C language computes the binary logarithm (rounding down) of an integer, rounded down. [2] The operator
// '>>' represents 'unsigned right shift'. The rounding down form of binary logarithm is identical to computing the position of the most
// significant 1 bit.
2544 /**
2545  * Returns the floor form of binary logarithm for a 32 bit integer.
2546  * -1 is returned if n is 0.
2547  */
2548 int floorLog2(unsigned int n) {
2549     int pos = 0;
2550     if (n >= 1<<16) { n >>= 16; pos += 16; }
2551     if (n >= 1<< 8) { n >>= 8; pos += 8; }
2552     if (n >= 1<< 4) { n >>= 4; pos += 4; }
2553     if (n >= 1<< 2) { n >>= 2; pos += 2; }
2554     if (n >= 1<< 1) { pos += 1; }
2555     return ((n == 0) ? (-1) : pos);
2556 }
2557
2558 // QuickSortExternal_4GB.c [
2559
2560 int strcmpKAZE13 (
2561     const char * src,
2562     const char * dst
2563 )
2564 {
2565     int ret = 0 ;
2566
2567     while( ! (ret = *(unsigned char *)src - *(unsigned char *)dst) && (*dst!=13-13))
2568         ++src, ++dst;
2569
2570     if ( ret < 0 )
2571         ret = -1 ;
2572     else if ( ret > 0 )
2573         ret = 1 ;
2574
2575     return( ret );
2576 }
2577
2578 //define LongestLineInclusive 51 //31 former, CAUTION: for command line options 'x' and 'y' it cannot be other than 31 [YET]!
2579
2580 #ifndef singleton
2581 #define LongestLineInclusive 31
2582 #endif
2583 #ifndef doubleton
2584 #define LongestLineInclusive 41
2585 #endif
2586 #ifndef tripleton
2587 #define LongestLineInclusive 41
2588 #endif
2589 #ifndef quadrupleton
2590 #define LongestLineInclusive 51
2591 #endif
2592 #ifndef quintupleton
2593 #define LongestLineInclusive 61
2594 #endif
2595 #ifndef sextupleton
2596 #define LongestLineInclusive 71
2597 #endif
2598 #ifndef septupleton
2599 #define LongestLineInclusive 81
2600 #endif
2601 #ifndef octupleton
2602 #define LongestLineInclusive 91
2603 #endif
2604 #ifndef nonupleton
2605 #define LongestLineInclusive 101
2606 #endif
2607 #ifndef decupleton
2608 #define LongestLineInclusive 111
2609 #endif
2610
2611 // _ngram_ 1 1-31
2612 // _ngram_ 2 5-41
2613 // _ngram_ 3 9-41
2614 // _ngram_ 4 13-51
2615 // _ngram_ 5 17-61
2616 // _ngram_ 6 21-71
2617 // _ngram_ 7 25-81
2618 // _ngram_ 8 29-91

```

```

2619 // _ngram_ 9 33-101
2620 // _ngram_ 10 37-111
2621 // For Leaf of 256bytes LongestLineInclusive should be 256 = 8+8+2*(LongestLineInclusive+1+4) or LongestLineInclusive = (256 - (8+8+8) -
2*(1+4))/2 = 111
2622
2623 char FourGramL[LongestLineInclusive+1+4]; // 31bytes longest 4-gram + 1byte NULL + 4bytes COUNTER
2624 char FourGramR[LongestLineInclusive+1+4]; // 31bytes longest 4-gram + 1byte NULL + 4bytes COUNTER
2625 char LEAF[8+8+8+2*(LongestLineInclusive+1+4)]; // 136bytes = 3 pointers + 2 keys
2626 char LEAFNEW[8+8+8+2*(LongestLineInclusive+1+4)]; // 136bytes = 3 pointers + 2 keys
2627 FILE *fp_outRG; // Global - not to burden the extract/compare function with one more parameter
2628 int CompareStringsEndingWith13_EXTERNAL(unsigned long long AtPosition64L, unsigned long long AtPosition64R) {
2629
2630 int i;
2631 unsigned long long *AtPosition64Lpointer=&AtPosition64L;
2632 unsigned long long *AtPosition64Rpointer=&AtPosition64R;
2633
2634 // Caramba: seek and tell report OK but in fact they lie, only setpos works?!?!?
2635
2636 //if defined(_WIN32_ENVIRONMENT_)
2637 //_lseeki64( fileno(fp_outRG), AtPosition64L, 0 );
2638 //else
2639 //fseeko( fp_outRG, AtPosition64L, SEEK_SET );
2640 //endif /* defined(_WIN32_ENVIRONMENT_) */
2641
2642 // _CRTIMP __int64 __cdecl _telli64(int);
2643 // off64_t ftello64 (FILE *stream)
2644
2645
2646 fsetpos(fp_outRG, AtPosition64Lpointer);
2647 for (i=0; i<(LongestLineInclusive+1+4); i++) {fread(&FourGramL[i], 1, 1, fp_outRG); if (FourGramL[i]==13-13) break;}
2648 //Commented line below is slower than the one above: 778156 clocks vs 756297 clocks.
2649 //fread(&FourGramL[0], 31+1, 1, fp_outRG);
2650
2651 fsetpos(fp_outRG, AtPosition64Rpointer);
2652 for (i=0; i<(LongestLineInclusive+1+4); i++) {fread(&FourGramR[i], 1, 1, fp_outRG); if (FourGramR[i]==13-13) break;}
2653 //Commented line below is slower than the one above: 778156 clocks vs 756297 clocks.
2654 //fread(&FourGramR[0], 31+1, 1, fp_outRG);
2655
2656 return(strcmpKAZE13(FourGramL, FourGramR));
2657 }
2658
2659 int CompareStringsEndingWith13_INTERNAL(unsigned long long AtPosition64L, unsigned long long AtPosition64R, char *POOLInternal) {
2660
2661 int i;
2662 //char FourGramL[LongestLineInclusive+2]; // 31 longest 4-gram + CR + LF
2663 //char FourGramR[LongestLineInclusive+2]; // 31 longest 4-gram + CR + LF
2664
2665 for (i=0; i<(LongestLineInclusive+1+4); i++) {
2666 //fread(&FourGramL[i], 1, 1, fp_in);
2667 FourGramL[i] = *(char *) (POOLInternal + AtPosition64L);
2668 if (FourGramL[i]==13-13) break;
2669 }
2670
2671 for (i=0; i<(LongestLineInclusive+1+4); i++) {
2672 //fread(&FourGramR[i], 1, 1, fp_in);
2673 FourGramR[i] = *(char *) (POOLInternal + AtPosition64R);
2674 if (FourGramR[i]==13-13) break;
2675 }
2676
2677 return(strcmpKAZE13(FourGramL, FourGramR));
2678 }
2679
2680 // QuickSortExternal_4+GB.c ]
2681
2682
2683 int main( argc, argv )
2684 int argc; char *argv[];
2685 {
2686 int nlines;
2687 string *backup = NULL;
2688
2689 FILE *fp_in, *fp_out, *fp_outLOG, *fp_inLINE;
2690 int LetterOffset;
2691 unsigned long long FilesLEN;
2692 unsigned long long WORDcount;
2693 unsigned long long WORDcountBOTTOM;
2694 unsigned long long WORDcountAttemptsToPut;
2695 int Thunderwith;
2696 unsigned long NumberOfFiles, WORDcountDistinct, WORDcountDistinctTOTAL = 0, TotalMemoryNeededForOnePass = 0;
2697 unsigned long long NumberOfLines; // rev. 12+
2698 unsigned long WHOLELetter_BufferSize;
2699 unsigned long long WHOLELetter_BufferSize_L14;
2700 unsigned long memory_size, LetterBuffer, j, k, LINE10len, wrdlen;
2701 unsigned long k_FIX;
2702 unsigned long long i; // rev. 12+
2703 //unsigned long size_in, size_out, size_inLINE;
2704 unsigned long size_in; // rev. 12+
2705 #if defined(_WIN32_ENVIRONMENT_)

```



```

2706 unsigned long long size_inLINESEXFOUR;
2707 #else
2708 size_t size_inLINESEXFOUR;
2709 #endif /* defined(_WIN32_ENVIRONMENT_) */
2710
2711 //unsigned long t1, t2, t3;
2712 time_t t1, t2, t3, tMainB, tMainE;
2713
2714 const int NumberOfSLOTS = 4096*2; // Since r.12+ in rev.12 it was 4096
2715 unsigned long StackPtr;
2716 //unsigned long BSTstack [65536*3]; // BST in worst case could become a LL.
2717 unsigned long long BSTstack [8192*3]; // BST in worst case could become a LL.
2718 unsigned long NumberOfTrees=0, NumberOfHashCollisions=0;
2719 unsigned long iBSTwithMAXpeak, jBSTwithMAXpeak;
2720 unsigned int PEAKinBST;
2721 unsigned long BSTsTotalLEAFs=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur below where 'break'
is ?!
2722 unsigned long BSTwithMAXnode=0, BSTcurrentNode=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2723 unsigned long BSTcurrentNodeMAXqUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
BSTcurrent occur below where 'break' is ?!
2724 unsigned long BSTwithMAXnodePEAK=1, BSTwithMAXnodeLEAF=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
occur below where 'break' is ?!
2725 unsigned long BSTwithMAXpeak=0, BSTcurrentPeak=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2726 unsigned long BSTcurrentPeakMAX=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2727 unsigned long BSTcurrentPeakMAXqUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
BSTcurrent occur below where 'break' is ?!
2728 unsigned long BSTwithMAXpeakNODE=1, BSTwithMAXpeakLEAF=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
occur below where 'break' is ?!
2729 unsigned long BSTwithMAXleaf=0, BSTcurrentLeaf=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2730 unsigned long BSTcurrentLeafMAXqUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
BSTcurrent occur below where 'break' is ?!
2731 unsigned long BSTwithMAXleafNODE=1, BSTwithMAXleafPEAK=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
occur below where 'break' is ?!
2732
2733 char *pointerflush, *pointerflushUNALIGN, *BufStart, *Flushing;
2734 char *pointerflush_64, *pointerflushUNALIGN_64; // r.14++
2735 unsigned long PseudoLinkedPointer, PseudoLinkedPointerNEW, PseudoLinkedPointerROOT, PseudoLinkedPointerNEWold;
2736 unsigned long PseudoLinkedPointerNEWleft, PseudoLinkedPointerNEWright;
2737 unsigned long PseudoLinkedPointerNEWmiddle;
2738 char *bufend[ 806 ]; // 'a'=0, ... 'z'=25 - 26 letters x 31 lengths
2739 long bufNumberOfWords[ 806 ]; // 'a'=0, ... 'z'=25 - 26 letters x 31 lengths
2740 // long bufNoWpS[ 806 ][ 8192 ]; // ?! crashes below when an attempt to use it occur
2741 char wrd[LongestLineInclusiv+1+4]; // 0..30, 31 = 0
2742 char wrdUP[LongestLineInclusiv+1+4]; // 0..30, 31 = 0
2743 char wrdUPold[LongestLineInclusiv+1+4]; // 0..30, 31 = 0
2744 char LINE10[257]; // 000..255, 256 = 0
2745 char ZEROS[4]; // 0..3, 0 = 0, 1 = 0, 2 = 0, 3 = 0
2746 char CRdLfA[2]; // 0..1, 0 = 13, 1 = 10
2747 char workbyte;
2748 char workK[1024*128];
2749 long workKoffset = -1;
2750 int FoundInLinkedList, Slot;
2751 unsigned long OffsetsInBuffer[31]; // 00..30
2752 unsigned long MAXusedBuffer[32]; // 00 not used, only 01..31
2753 unsigned long GRMBLhlll[32]; // 00..31
2754 unsigned long GRMBLFoolAgain[32]; // 00..31
2755 int Melnitcka;
2756 unsigned long MAXusedBufferABS = 0;
2757 unsigned long Utiliza1 = 0;
2758 unsigned long Utiliza2 = 0;
2759 unsigned long TotalWLchars = 0;
2760
2761 /* minimum signed 64 bit value */
2762 #define _I64_MIN (-9223372036854775807i64 - 1)
2763 /* maximum signed 64 bit value */
2764 #define _I64_MAX 9223372036854775807i64
2765 /* maximum unsigned 64 bit value */
2766 #define _UI64_MAX 0xffffffffffffffffui64
2767
2768 /* minimum signed 128 bit value */
2769 #define _I128_MIN (-170141183460469231731687303715884105727i128 - 1)
2770 /* maximum signed 128 bit value */
2771 #define _I128_MAX 170141183460469231731687303715884105727i128
2772 /* maximum unsigned 128 bit value */
2773 #define _UI128_MAX 0xffffffffffffffffffffffffffffffffui128
2774
2775 char IIT0aDigiTs[27]; // 9, 223, 372, 036, 854, 775, 807: 1(sign or carry)+19(digits)+1('\'0')+6(,)
2776 // below duplicates are needed because of one_line_invoking need different buffers.
2777 char IIT0aDigiTs2[27]; // 9, 223, 372, 036, 854, 775, 807: 1(sign or carry)+19(digits)+1('\'0')+6(,)
2778 char IIT0aDigiTs3[27]; // 9, 223, 372, 036, 854, 775, 807: 1(sign or carry)+19(digits)+1('\'0')+6(,)
2779 char IIT0aDigiTs4[27]; // 9, 223, 372, 036, 854, 775, 807: 1(sign or carry)+19(digits)+1('\'0')+6(,)
2780 unsigned long HEADOffsetFromStartBUKVA = 0;
2781 unsigned long TAILOffsetFromStartBUKVA = 0;
2782 int BSTorBtree = 0;

```

```

2783     int SplitOccured;
2784     int POffsetInLEAF;
2785 char *Auberge[4] = {"|\\0", "\\0", "-\\0", "\\0\\0"};
2786 int hashAlfa, iAlfa;
2787 int PLE_words=0; // Quadruple!
2788 char wrd1st[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2789 char wrd2nd[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2790 char wrd3rd[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2791 char wrd4th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2792 char wrd5th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2793 char wrd6th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2794 char wrd7th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2795 char wrd8th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2796 char wrd9th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2797 char wrd10th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2798 char *DelimiterUnderscore = "\\0";
2799 int PLE_words_INITflag = 0;
2800
2801 // QuickSortExternal_4GB [
2802 unsigned long long ThunderwithL64_L14;
2803 unsigned long long Strnglen64_L14;
2804 unsigned long long size_in64_L14, size_in2_L14;
2805 unsigned long long Over4billionLines, j_Over4billion;
2806 char OneChar_iByte = '\\0';
2807 char CR_iByte = '\\r';
2808 char SomeByte;
2809 unsigned long long BufEnd_64;
2810 unsigned long long SeekPosition;
2811 unsigned long long *PointerToSeekPosition;
2812 char FourGram[LongestLineInclusive+2]; // 31 longest 4-gram + CR + LF
2813 char *PoolPhysical;
2814 unsigned long long fsetpos_ZERO=0;
2815 char OneClusterZEROS[1024*4]; // Caution: must be ZEROed(NULLified)!
2816 char *FileSwapTag = "LEPRECHAUNISH";
2817 char EOFcode = 0x1A;
2818 unsigned long long PseudoLinkedPointer_64, PseudoLinkedPointerNEW_64, PseudoLinkedPointerROOT_64, PseudoLinkedPointerNEWold_64;
2819 unsigned long long PseudoLinkedPointerNEWleft_64, PseudoLinkedPointerNEWright_64;
2820 unsigned long long PseudoLinkedPointerNEWmiddle_64;
2821 unsigned long long NULLs_64 = 0;
2822 unsigned long long PseudoLinkedPointerAUX_64;
2823 unsigned long long PseudoLinkedPointerAUXdumbo_64;
2824 char wrdAUX[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2825 // QuickSortExternal_4GB ]
2826
2827 unsigned long CounterOccurrences;
2828 unsigned long long NumberOfLEAFs=0;
2829 unsigned long LevelInCorona_Not_Counting_ROOT=0;
2830 char *ngram[11] =
    {"NULLleton\\0", "singleton\\0", "doubleton\\0", "tripleton\\0", "quadrupleton\\0", "quintupleton\\0", "sextupleton\\0", "septupleton\\0", "octupleton\\0", "nonupleton\\0", "decupleton\\0"};
2831
2832 unsigned long RipPasses;
2833 unsigned long long NULLsForWRD=0;
2834
2835 //15+
2836 int DoNotInsertFlag = 0;
2837 int METACOMMANDFlag;
2838
2839 //16
2840 int REUSE=0;
2841 int HSHexist;
2842 int SWPexist;
2843
2844 // INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT
2845 printf ("Leprechaun_%s (Fast-In-Future Greedy n-gram-Ripper), rev. 16FIXFIX, written by Svalqatchx.\\n", ngram[_ngram_]);
2846 //puts ("Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'");
2847
2848 #ifdef singleton
2849 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 1..31 chars from incoming texts.\\n", _ngram_, _ngram_);
2850 #endif
2851 #ifdef doubleton
2852 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 5..41 chars from incoming texts.\\n", _ngram_, _ngram_);
2853 #endif
2854 #ifdef tripleton
2855 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 9..41 chars from incoming texts.\\n", _ngram_, _ngram_);
2856 #endif
2857 #ifdef quadrupleton
2858 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 13..51 chars from incoming texts.\\n", _ngram_, _ngram_);
2859 #endif
2860 #ifdef quintupleton
2861 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 17..61 chars from incoming texts.\\n", _ngram_, _ngram_);
2862 #endif
2863 #ifdef sextupleton
2864 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 21..71 chars from incoming texts.\\n", _ngram_, _ngram_);
2865 #endif
2866 #ifdef septupleton
2867 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 25..81 chars from incoming texts.\\n", _ngram_, _ngram_);
2868 #endif

```

```

2869 #ifdef octupleton
2870 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 29..91 chars from incoming texts.\n", _ngram_, _ngram_);
2871 #endif
2872 #ifdef nonupletone
2873 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 33..101 chars from incoming texts.\n", _ngram_, _ngram_);
2874 #endif
2875 #ifdef decupletone
2876 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 37..111 chars from incoming texts.\n", _ngram_, _ngram_);
2877 #endif
2878 puts( "Feature1: All words within x-lets/n-grams are in range 1..31 chars inclusive." );
2879 //puts( "Feature2: In this revision 128MB 1-way hash is used which results in 16,777,216 external B-Trees of order 3." );
2880
2881 if (HashInBITS<3<10)
2882 printf ("Feature2: In this revision %sbytes 1-way hash is used which results in %s external B-Trees of order 3.\n",
        _ui64toaKAZEcomma(((1<<HashInBITS)<<3), 11T0aDigits, 10), _ui64toaKAZEcomma((1<<HashInBITS), 11T0aDigits2, 10) );
2883 else if (HashInBITS+3>=10 && HashInBITS+3<20)
2884 printf ("Feature2: In this revision %sKB 1-way hash is used which results in %s external B-Trees of order 3.\n", _ui64toaKAZEcomma(
        (((1<<HashInBITS)<<3))>>10, 11T0aDigits, 10), _ui64toaKAZEcomma((1<<HashInBITS), 11T0aDigits2, 10) );
2885 else
2886 printf ("Feature2: In this revision %sMB 1-way hash is used which results in %s external B-Trees of order 3.\n", _ui64toaKAZEcomma(
        (((1<<HashInBITS)<<3))>>20, 11T0aDigits, 10), _ui64toaKAZEcomma((1<<HashInBITS), 11T0aDigits2, 10) );
2887 if (HashInBITS-HashChunkSizeInBITS==0)
2888 printf ("Feature3: In this revision, %s pass is to be made.\n", _ui64toaKAZEcomma(1<<(HashInBITS-HashChunkSizeInBITS), 11T0aDigits, 10));
2889 else
2890 printf ("Feature3: In this revision, %s passes are to be made.\n", _ui64toaKAZEcomma(1<<(HashInBITS-HashChunkSizeInBITS), 11T0aDigits, 10));
2891
2892 puts( "Feature4: If the external memory has latency 99+microseconds then !(look no further), IOPS(seek-time) rules." );
2893 // The phrase 'look no further' was used in amazon.com review meaning 'stop searching for better thing this is it'.
2894 //puts( "Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us." );
2895 //puts( "      also the performance of a 3-way hash + 6,602,752 B-Trees of order 3." );
2896 //puts( "      also the performance of a 1-way hash + 134,217,728 external B-Trees of order 3." );
2897 //puts( "Note1: Compiled with Microsoft C v. 13.10.3077: 'cl /Ox /TcLeprechaun.c'." );
2898 //puts( "Note2: This WORDLISTER makes as output pseudo(unsorted)_wordlist_CRLF_file." );
2899 if( argc != 3 && argc != 4 && argc != 5 && argc != 6 ) // +1 for program name
2900 {
2901     puts( "" );
2902     puts( "'The Little Monster' short notes:" );
2903     puts( "Note1: I wish to thank to R.N. Horspool, Ranjan Sinha, Dmitry Shkarin." );
2904     puts( "      Michael Abrash, J. Bentley, R. Sedgewick, Igor Pavlov, Lasse Reinhold." );
2905     puts( "      Landon Noll, Peter Kankowski for sharing their knowledge to public." );
2906     puts( "Note2: Run it without parameters to get usage and short notes." );
2907     puts( "Note3: This simple amateurish(more over I am not versed well neither in C nor" );
2908     puts( "      in mathematics nor in English language, but I am persistent in INDEXING" );
2909     puts( "      GBs of english TEXTS) tool is written in ANSI C(at least its source is)" );
2910     puts( "      compileable for CL(Windows) and GCC(Linux), and its purpose is to" );
2911     puts( "      create a WordList for a group of files(given via filelist)." );
2912     puts( "      Its name comes(according to Heritage Dictionary) from 'low corpus' or" );
2913     puts( "      'little body', in fact from amazing movie saga 'Leprechaun 1-2-3-4-5-6'" );
2914     puts( "      starring by Warwick Davis." );
2915     puts( "Note4: Only words up to 31 chars are proceeded - the reason is 'DDT'(the" );
2916     puts( "      longest word in Heritage Dictionary 3rd edition) or" );
2917     puts( "      'di chl orodi phenyl tri chl oroethane'." );
2918     puts( "Note5: Cursor hiding in C - mission impossible for me." );
2919     puts( "Note6: By default(third parameter is 1023) allocated memory is 393MB." );
2920     puts( "      Due to 'malloc()' limitation under WINDOWS, maximum value of third" );
2921     puts( "      parameter is 5174 which is 1988MB allocated block." );
2922     puts( "Note7: File Leprechaun.LOG is a log, where new statistics are appended." );
2923     puts( "Note8: Revision 12+ can handle files larger than 4GB." );
2924     puts( "Note9: Revision 12++ has a buffered 'fread()' - therefore I/O READ-BURST SPEED" );
2925     puts( "      is the first(worst) bottleneck, as a result r.12++ is much-much faster;" );
2926     puts( "      the second(worse) bottleneck: the linked lists - the b-trees" );
2927     puts( "      might be the answer; the third(bad) bottleneck: the amateurish author." );
2928     puts( "NoteA: Revision 12+++ has an improved(2 bits were used dolitshly) main hash" );
2929     puts( "      function - therefore less collisions, for example:" );
2930     puts( "      for file 'wikipedia-de-html.tar' 42,291,855,360 bytes with" );
2931     puts( "      5,750,179,678 words of them 7,375,373 distinct attempts to Find/Put" );
2932     puts( "      a WORD into a linked list are 6,117,675,470(r.12++) and 5,845,989,790" );
2933     puts( "      (r.12+++); also two 'if' sections were moved because they were executed" );
2934     puts( "      unnecessarily many times." );
2935     puts( "NoteB: Revision 13 uses BSTs instead of LLs, that is Linked-Lists were" );
2936     puts( "      replaced by Binary-Search-Trees, as a result for 22,202,980 distinct" );
2937     puts( "      words(out of 35,271,297) r.12+++ needs 225,548,268 total attempts to" );
2938     puts( "      Find/Put WORDs into linked lists where r.13 needs 121,674,042 total" );
2939     puts( "      attempts to Find/Put WORDs into Binary-Search-Trees. But this is a" );
2940     puts( "      significant boost in performance only for wordlists of million words." );
2941     puts( "NoteC: Revision 13+ gives only more statistics. Future revisions could lessen" );
2942     puts( "      number of attempts to Find/Put WORDs into Binary-Search-Trees" );
2943     puts( "      furthermore by making them at some point Perfectly-Balanced. But" );
2944     puts( "      for huge amount(multi-(m)billion) of distinct words the b-tree family" );
2945     puts( "      must come in, until then this is the leprechaunish niche." );
2946     puts( "NoteD: Revision 13++ has a little fix(2 unnecessary ZEROings, when a new word" );
2947     puts( "      is inserted, were deleted) and a fixed bug(13+ adds stupidly the" );
2948     puts( "      highest BST to the wordlist). Also B-Tree of order 3 is added as a" );
2949     puts( "      searching method. Main goal of B-Tree is to reduce number of" );
2950     puts( "      comparisons but at nasty cost: a precious time wasted to construct it" );
2951     puts( "      and twice more memory, i.e. one step forward two backward: this tree is" );
2952     puts( "      more effective than BST in cases of 2++ billion/million" );
2953     puts( "      different/distinct words." );

```

```

2954 puts( " The improvement which comes from using B-Tree of order 3 is about 200%" );
2955 puts( " much more pleasing than I expected, for wikipedia-en-html.tar.wrd with" );
2956 puts( " 12,561,874 distinct words Total Attempts to Find/Put WORDS into:" );
2957 puts( " Binary-Search-Trees was 61,895,043 while for" );
2958 puts( " B-trees order 3 was 19,295,791." );
2959 puts( "NoteE: Revision 13+++ has a faster(not heavily tested yet) and with" );
2960 puts( " better(0.6% to 1.1%) dispersion Fowler/Noll/Vo hash." );
2961 puts( " so called FNV1a hash. Revision 13++++ boosting: Leprechaun_Intel.exe" );
2962 puts( " gives 1,256,187W/s for wikipedia-en-html.tar.wrd with FNV1_32_PRIME:" );
2963 puts( " 107712257 with 3,551,736 dispersion for 'FNV1A_Hash_Granularity'." );
2964 puts( "NoteF: For old r.12+ a USB connected HDD crippled test:" );
2965 puts( " for 'H:\>Leprechaun.exe static.wikipedia.org_downloads_2008-06_en.lst" );
2966 puts( " wikipedia-en-html.tar.wrd 5400" );
2967 puts( " where 223,674,511,360 wikipedia-en-html.tar" );
2968 puts( " on laptop Toshiba Pentium T3400 2166 MHz with" );
2969 puts( " Motherboard Name: Toshiba Satellite L305" );
2970 puts( " CPU Type: Mobile DualCore Intel Pentium, 2166 MHz (13 x 167)" );
2971 puts( " CPU Alias: Merom-1M" );
2972 puts( " L1 Code Cache: 32 KB per core" );
2973 puts( " L1 Data Cache: 32 KB per core" );
2974 puts( " L2 Cache: 1 MB (On-Die, ECC, ASC, Full-Speed)" );
2975 puts( " Bus Type: Dual DDR2 SDRAM" );
2976 puts( " Bus Width: 128-bit" );
2977 puts( " Real Clock: 333 MHz (DDR)" );
2978 puts( " Effective Clock: 666 MHz" );
2979 puts( " EVEREST v5.00.1650 Memory Copy: 3725MB/s with timings 5-5-5-13" );
2980 puts( " result is logged to 'Leprechaun.LOG':" );
2981 puts( " Bytes per second performance: 20,658,955B/s" );
2982 puts( " Words per second performance: 2,860,880W/s" );
2983 puts( " Input File with a list of TEXTual Files:" );
2984 puts( " static.wikipedia.org_downloads_2008-06_en.lst" );
2985 puts( " Size of all TEXTual Files: 223,674,511,360" );
2986 puts( " Word count: 30,974,750,142 of them 12,561,874 distinct" );
2987 puts( " Number Of Files: 1" );
2988 puts( " Number Of Lines: 2088618575" );
2989 puts( " Allocated memory in MB: 1920" );
2990 puts( " Words with length 01 occupy 0,033KB of 0,349KB given i.e. 09% utilization" );
2991 puts( " Words with length 02 occupy 0,033KB of 0,349KB given i.e. 09% utilization" );
2992 puts( " Words with length 03 occupy 0,037KB of 0,697KB given i.e. 05% utilization" );
2993 puts( " Words with length 04 occupy 0,151KB of 0,871KB given i.e. 17% utilization" );
2994 puts( " Words with length 05 occupy 0,744KB of 1,568KB given i.e. 47% utilization" );
2995 puts( " Words with length 06 occupy 1,470KB of 3,136KB given i.e. 46% utilization" );
2996 puts( " Words with length 07 occupy 2,605KB of 5,923KB given i.e. 43% utilization" );
2997 puts( " Words with length 08 occupy 3,296KB of 6,968KB given i.e. 47% utilization" );
2998 puts( " Words with length 09 occupy 3,714KB of 6,968KB given i.e. 53% utilization" );
2999 puts( " Words with length 10 occupy 3,483KB of 6,968KB given i.e. 49% utilization" );
3000 puts( " Words with length 11 occupy 3,235KB of 5,923KB given i.e. 54% utilization" );
3001 puts( " Words with length 12 occupy 2,691KB of 4,181KB given i.e. 64% utilization" );
3002 puts( " Words with length 13 occupy 2,230KB of 3,484KB given i.e. 64% utilization" );
3003 puts( " Words with length 14 occupy 1,718KB of 3,484KB given i.e. 49% utilization" );
3004 puts( " Words with length 15 occupy 1,357KB of 2,613KB given i.e. 51% utilization" );
3005 puts( " Words with length 16 occupy 1,063KB of 2,613KB given i.e. 40% utilization" );
3006 puts( " Words with length 17 occupy 0,814KB of 1,742KB given i.e. 46% utilization" );
3007 puts( " Words with length 18 occupy 0,617KB of 1,742KB given i.e. 35% utilization" );
3008 puts( " Words with length 19 occupy 0,485KB of 1,742KB given i.e. 27% utilization" );
3009 puts( " Words with length 20 occupy 0,402KB of 1,742KB given i.e. 23% utilization" );
3010 puts( " Words with length 21 occupy 0,327KB of 1,742KB given i.e. 18% utilization" );
3011 puts( " Words with length 22 occupy 0,274KB of 1,742KB given i.e. 15% utilization" );
3012 puts( " Words with length 23 occupy 0,224KB of 1,394KB given i.e. 16% utilization" );
3013 puts( " Words with length 24 occupy 0,190KB of 1,394KB given i.e. 13% utilization" );
3014 puts( " Words with length 25 occupy 0,162KB of 1,394KB given i.e. 11% utilization" );
3015 puts( " Words with length 26 occupy 0,136KB of 1,220KB given i.e. 11% utilization" );
3016 puts( " Words with length 27 occupy 0,119KB of 1,046KB given i.e. 11% utilization" );
3017 puts( " Words with length 28 occupy 0,107KB of 0,871KB given i.e. 12% utilization" );
3018 puts( " Words with length 29 occupy 0,091KB of 0,697KB given i.e. 13% utilization" );
3019 puts( " Words with length 30 occupy 0,080KB of 0,523KB given i.e. 15% utilization" );
3020 puts( " Words with length 31 occupy 0,076KB of 0,523KB given i.e. 14% utilization" );
3021 puts( " Total pseudo(including hash table) memory utilization: 42%" );
3022 puts( " Total real(wordlist's words VS allocated block) memory utilization: 60/1000" );
3023 puts( " Used value for third parameter in KB: 5400" );
3024 puts( " Use next time as third parameter: 3475-" );
3025 puts( " Time for making unsorted wordlist: 10827 second(s)" );
3026 puts( " Time for sorting unsorted wordlist: 10 second(s)" );
3027 puts( "NoteG: 2011-Mar-07: Fixed a small command line parsing bug." );
3028 puts( "NoteH: A heavy blow for my illusions(regarding speed performance of external b-trees)," );
3029 puts( " desperate results for ripping on HDD 7200rpm:" );
3030 puts( " 20,000,000 distinct 4-grams per 5 hours." );
3031 puts( " D:\>Leprechaun_quadupleton_r14_minus>Leprechaun_quadupleton GRAFFITH_2048.lst GRAFFITH_2048.wrd 48000000 z" );
3032 puts( " Leprechaun(Fast Greedy Word-Ripper), rev. 14_minus_quadupleton, written by Svalqyatch." );
3033 puts( " Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'" );
3034 puts( " Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us," );
3035 puts( " also the performance of a 3-way hash + 6,602,752 B-Trees of order 3," );
3036 puts( " also the performance of a 1-way hash + 134,217,728 external B-Trees of order 3." );
3037 puts( " Size of input file with files for Leprechauning: 42140" );
3038 puts( " Allocating HASH memory 1,073,741,889 bytes ... OK" );
3039 puts( " Allocating/ZEROing 49,152,000,014 bytes swap file ... OK" );
3040 puts( " Size of Input TEXTual file: 33,470,581" );
3041 puts( " |; Word count: 3,045,077 of them 2,597,942 distinct; Done: 64/64" );

```

```

3042 puts( "      Size of Input TEXTual file: 17,229,900" );
3043 puts( "      -; Word count: 4,235,032 of them 3,588,757 distinct; Done: 64/64" );
3044 puts( "      Size of Input TEXTual file: 19,191,256" );
3045 puts( "      |; Word count: 5,803,400 of them 4,866,213 distinct; Done: 64/64" );
3046 puts( "      Size of Input TEXTual file: 34,651,077" );
3047 puts( "      \\; Word count: 8,714,961 of them 6,941,108 distinct; Done: 64/64" );
3048 puts( "      Size of Input TEXTual file: 26,875,458" );
3049 puts( "      /; Word count: 11,022,830 of them 8,579,931 distinct; Done: 64/64" );
3050 puts( "      Size of Input TEXTual file: 19,605,129" );
3051 puts( "      -; Word count: 12,924,821 of them 10,078,191 distinct; Done: 64/64" );
3052 puts( "      Size of Input TEXTual file: 17,053,521" );
3053 puts( "      /; Word count: 14,577,010 of them 11,455,983 distinct; Done: 64/64" );
3054 puts( "      Size of Input TEXTual file: 44,087,709" );
3055 puts( "      -; Word count: 18,953,280 of them 15,010,569 distinct; Done: 64/64" );
3056 puts( "      Size of Input TEXTual file: 32,796,705" );
3057 puts( "      |; Word count: 22,412,912 of them 17,621,649 distinct; Done: 64/64" );
3058 puts( "      Size of Input TEXTual file: 19,538,360" );
3059 puts( "      /; Word count: 24,381,005 of them 19,137,701 distinct; Done: 64/64" );
3060 puts( "      Size of Input TEXTual file: 29,565,366" );
3061 puts( "      \\; Word count: 26,214,400 of them 20,528,357 distinct; Done: 40/64" );
3062 puts( "      ..." );
3063 puts( "NoteI: In revision 14- the resultant wordlist is NOT sorted when 'Z' is used." );
3064 puts( "NoteJ: In revision 14 'x' and 'y' options are disabled, for 7++ million phrases their usefulness is no more." );
3065 puts( "      the real loads are of order 800+ million, too many limitations exist, they must be rewritten as 64bit." );
3066 puts( "NoteK: Ripping OSHO.TXT (10,165,640 4-grams) on HDD daunts because of 6+hours needed." );
3067 puts( "      Number Of Trees(GREATER THE BETTER): 9,433,894" );
3068 puts( "      Used value for third parameter in KB: 3,145,728" );
3069 puts( "      Use next time as third parameter: 1,262,186" );
3070 puts( "      One leaf has size: 8+8+8+(51+1+4)+(51+1+4)=136bytes." );
3071 puts( "      or MAX (one 4-gram per leaf) 10,165,640*136=1,382,527,040bytes." );
3072 puts( "NoteL: Each phrase in extracted file is preceded by TAB ASCII code, this (TAB being a delimiter symbol) allows" );
3073 puts( "      the phrase-list to be ripped again i.e. to treat already ripped files as any other text." );
3074 puts( "NoteM: Too many 'fsetpos', 'fread', 'fwrite' invocations were put in the straight port (from 32bit internal memory to" );
3075 puts( "      64bit external memory), a optimization is needed, something like reading/writing a LEAF at once." );
3076 puts( "NoteN: Since revision 14+: Optimized(LEAFwise) search (fragment 1] and 2]), insert (fragment 3]) and dump." );
3077 puts( "NoteO: In next revisions a 2in1 is to be done i.e. one code fragment will deal with virtual and physical memory." );
3078 puts( "      thus establishing pure 64bit mode of operation, a single flag will decide whether 'memcpy' or" );
3079 puts( "      the slow I/O triad sub-fragments will be used. DONE." );
3080 puts( "NoteP: In next revisions a multi-pass (by chunking the hash table) mode is to be added in order to avoid" );
3081 puts( "      these sick-seeks. DONE." );
3082 puts( "NoteQ: Fixed occurrences bug due to not NULLifying the field housing the occurrences, a nasty thing: all" );
3083 puts( "      the revisions 14??? were buggy, how stupid from my side, grumble." );
3084 puts( "NoteR: In r.14+++++FIXFIX were fixed STATS(Leprechaun.LOG) bugs (appearing only in multi-pass mode) due to not" );
3085 puts( "      NULLifying the variables housing the stats, they do not affect the results - they are for informative use." );
3086 puts( "NoteS: Fixed a division-by-zero bug, occurs when finishing-starting time is under 1 second." );
3087 puts( "      Fixed a nasty bug causing very restrictive way of forming x-grams." );
3088 puts( "NoteT: At last and finally the nasty bug causing very restrictive way of forming x-grams was REALLY fixed - lack of" );
3089 puts( "      calmness jammed (again) my actions - a lesson to be learnt." );
3090
3091 puts( "NoteU: Since r.15FIXFIX+ the ability to command Leprechaun (from inside the list file with 2 metacommands) to enter/exit" );
3092 puts( "      INSERT mode was added. This allows to control whether new (to current hash-tree structure) x-grams are to be counted" );
3093 puts( "      [and] INSERTed. These two metacommands are:" );
3094 puts( "      Leprechaun says x-gram inserting disabled for next files: ON" );
3095 puts( "      Leprechaun says x-gram inserting disabled for next files: OFF" );
3096
3097 puts( "NoteV: When W/w option is used multiple-passes shouldn't be dumped - it is meaningless, dump when only one pass," );
3098 puts( "      that is, use W/w only in ONE-PASS mode otherwise it behaves as Z/z but DOES NOT dump to OutFile." );
3099 puts( "      It uses in READ mode the two HASH+TREES output files: 'Leprechaun_64bit.hsh' and 'Leprechaun_64bit.swp'." );
3100 puts( "      If during the start one of them is missing then Z/z behaviour is on, at end 'Leprechaun_64bit.hsh' is dumped." );
3101 puts( "      Also the OutFile has all incoming x-grams which are present in the corpus (i.e. HASH+TREES structure)." );
3102 puts( "" );
3103 puts( "Usage: Leprechaun InFile OutFile [BufferSize] [SortMethod] [TreeMethod]" );
3104 puts( "      <InFile>: Input file with files for Leprechauning, in WINDOWS console" );
3105 puts( "      you can create it by 'E:\\KAZEHOME>dir *.txt/s/b>Leprechaun.lst'" );
3106 puts( "      <OutFile>: Output WORDLIST(sorted since r.9, CRLF) file" );
3107 puts( "      <BufferSize>: Optional Dynamic RAM buffer in KB, default(and minimum" );
3108 puts( "      in the same time) is 1023, i.e. omit or specify greater one" );
3109 puts( "      <SortMethod>: Optional Sort Method, default is 'D'," );
3110 puts( "      A - InsertionSort" );
3111 puts( "      B - InsertionX26Sort" );
3112 puts( "      C - MultiKeyQuickSortSort by J. Bentley, R. Sedgewick" );
3113 puts( "      D - MultiKeyQuickSortX26Sort' by J. Bentley, R. Sedgewick" );
3114 puts( "      <TreeMethod>: Optional Tree Method, default is 'X'," );
3115 puts( "      X - Binary-Search-Trees" );
3116 puts( "      y - B-Trees of order 3, INTERNAL/fast memory digi tless i.e. no repetitions, 64bit addressing!" );
3117 puts( "      Y - B-Trees of order 3, INTERNAL/fast memory, 64bit addressing!" );
3118 puts( "      z - B-Trees of order 3, EXTERNAL/slow memory digi tless i.e. no repetitions, 64bit addressing!" );
3119 puts( "      Z - B-Trees of order 3, EXTERNAL/slow memory, 64bit addressing!" );
3120 puts( "      w - B-Trees of order 3, EXTERNAL/slow memory digi tless i.e. no repetitions, 64bit addressing! REUSE!" );
3121 puts( "      W - B-Trees of order 3, EXTERNAL/slow memory, 64bit addressing! REUSE!" );
3122 puts( "" );
3123 puts( "Have a nice Leprechauning." );
3124 puts( "For contacts: sanmayce@sanmayce.com" );
3125 puts( "Sanmayce Svalqyatchx 'Kaze', 2005 Feb 07. Last revision: 2013 Mar 31." );
3126 return( 1 );
3127 }
3128
3129 GRMBLhi || [0]=0;

```

```

3130 GRMBLhi1[1]=1;
3131 GRMBLhi1[2]=1;
3132 GRMBLhi1[3]=1;
3133 GRMBLhi1[4]=1;
3134 GRMBLhi1[5]=1;
3135 GRMBLhi1[6]=1;
3136 GRMBLhi1[7]=1;
3137 GRMBLhi1[8]=1;
3138 GRMBLhi1[9]=1;
3139 GRMBLhi1[10]=1;
3140 GRMBLhi1[11]=1;
3141 GRMBLhi1[12]=15;
3142 GRMBLhi1[13]=15;
3143 GRMBLhi1[14]=15;
3144 GRMBLhi1[15]=20;
3145 GRMBLhi1[16]=30;
3146 GRMBLhi1[17]=40;
3147 GRMBLhi1[18]=50;
3148 GRMBLhi1[19]=50;
3149 GRMBLhi1[20]=50;
3150 GRMBLhi1[21]=40;
3151 GRMBLhi1[22]=40;
3152 GRMBLhi1[23]=40;
3153 GRMBLhi1[24]=30;
3154 GRMBLhi1[25]=20;
3155 GRMBLhi1[26]=20;
3156 GRMBLhi1[27]=20;
3157 GRMBLhi1[28]=20;
3158 GRMBLhi1[29]=20;
3159 GRMBLhi1[30]=10;
3160 GRMBLhi1[31]=10;
3161
3162 (void) time(&tMainB);
3163
3164 if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
3165 { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
3166 fclose(fp_out); // The file must be with size 0 because it is opened for appending down below.
3167
3168 // 2^(Hash1nBITs-HashChunkSize1nBITs)=2^0=1 passe(s).
3169 // 14++++ [
3170 //for( RipPasses = 1-1; RipPasses <= (1<<(Hash1nBITs-HashChunkSize1nBITs))-1; RipPasses++ )
3171 //{
3172 // 14++++ ]
3173 RipPasses = 1-1;
3174 WhyTheHellForIsNotWorking:
3175 printf( "Pass #%u of %u: \n", RipPasses+1, (1<<(Hash1nBITs-HashChunkSize1nBITs)));
3176
3177 if( ( fp_in = fopen( argv[1], "rb" ) ) == NULL )
3178 { printf( "Leprechaun: Can't open file %s \n", argv[1] ); return( 1 ); }
3179
3180 fseek( fp_in, 0L, SEEK_END );
3181 size_in = ftell( fp_in );
3182 fseek( fp_in, 0L, SEEK_SET );
3183 printf( "Size of input file with files for Leprechauning: %lu\n", size_in );
3184
3185 if( ( fp_outLOG = fopen( "Leprechaun.LOG", "a+" ) ) == NULL )
3186 { printf( "Leprechaun: Can't open file Leprechaun.LOG.\n" ); return( 1 ); }
3187
3188 // argc is 4|5|6 due to eventual missing BufferSize
3189 if( argc == 4 ) // not 6 due to eventual missing BufferSize and SortMethod
3190     k_FIX = 3;
3191 if( argc == 5 ) // not 6 due to eventual missing BufferSize or SortMethod
3192     k_FIX = 4;
3193 if( argc == 6 )
3194     k_FIX = 5;
3195 if ( *argv[k_FIX] == 'Y' || *argv[k_FIX] == 'y' ) BStorBtree = 1+2; // +2 since r.14++
3196 if ( *argv[k_FIX] == 'Z' || *argv[k_FIX] == 'z' ) BStorBtree = 2;
3197 if ( *argv[k_FIX] == 'W' || *argv[k_FIX] == 'w' ) {BStorBtree = 2; REUSE=1;}
3198
3199 if( argc == 4 || argc == 5 || argc == 6 ) Thunderwith = atoi( argv[3] );
3200 else Thunderwith = 527; // for r.12: 527=17*31 this is minimum because of 4096*1*4=16KB+ needed for each buffer!
3201 // for r.12+: 1023=33*31 this is minimum because of 4096*2*4=32KB+ needed for each buffer!
3202 if (Thunderwith < 1023) {Thunderwith = 1023;}
3203
3204 //printf( "Use next time as third parameter: %s\n", _ui64toaKAZEcomma((25123456789>>10)+1, IIT0aDigits, 10) );
3205 //printf( "Use next time as third parameter: %s\n", _ui64toaKAZEcomma((25123456789/1024)+1, IIT0aDigits, 10) );
3206
3207 if (BStorBtree < 2) { printf( "Leprechaun: In this particular revision 'x' option is disabled.\n" ); return( 1 ); }
3208
3209 if (BStorBtree < 2) {
3210     LetterBuffer = Thunderwith * 1024;
3211     WHOLEletter_BufferSize = 0;
3212     for( i = 1; i <= 31; i++ )
3213     { OffsetsInBuffer[i-1] = 0;
3214       for( j = 1; j <= i; j++ )
3215       { OffsetsInBuffer[i-1] = OffsetsInBuffer[i-1] + (GRMBLhi1[(int)(j-1)] * LetterBuffer)/31;
3216       }
3217       WHOLEletter_BufferSize = WHOLEletter_BufferSize + (GRMBLhi1[(int)i] * LetterBuffer)/31;

```

```

3218     GRMBLFoolAgain[(int)i] = (GRMBLhi11[(int)i] * LetterBuffer)/31;
3219 }
3220 memory_size = 26 * WHOLEletter_BufferSize + 1 + 64;
3221 printf( "Allocating memory %luMB ... ", (memory_size>>20)+1 );
3222 pointerflushUNALIGN = (char *)malloc( memory_size );
3223 if( pointerflushUNALIGN == NULL )
3224 { puts( "\nLeprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
3225 pointerflush = pointerflushUNALIGN + 64 - (((size_t)pointerflushUNALIGN) % 64); // 13_6+
3226 //offset=64-int((long)data&63);
3227
3228 printf( "OK\n");
3229     fprintf( fp_outLOG, "Leprechaun report:\n" );
3230
3231 // Check once for ever whether allocated memory is ZEROed? Answer: YES
3232 //for( i = 0; i < memory_size; i++ )
3233 // if (*(char *) (pointerflush+i)!=0) printf("NON-ZERO encountered, so 'NO'.");
3234
3235 for( i = 0; i < 26; i++ )
3236 { for( k = 1; k <= 31; k++ )
3237 { bufend[i*31+k-1] = pointerflush + i * WHOLEletter_BufferSize + OffsetsInBuffer[k-1]; // i*31+k-1 must be 0..805
3238     if (i==25) { MAXusedBuffer[k] = (unsigned long)bufend[i*31+k-1]; }
3239     for( j = 0; j < (NumberOfSLOTS+1)*4; j++ ) // ? memset(bufend[i],0,(NumberOfSLOTS+1)*4);
3240     { *bufend[i*31+k-1]++ = 0;
3241       //++bufend[i*31+k-1];
3242     }
3243     if (i==25) { MAXusedBuffer[k] = (unsigned long)bufend[i*31+k-1]-MAXusedBuffer[k]; }
3244     bufNumberOfWords[i*31+k-1]=0;
3245 //for( j = 0; j < NumberOfSLOTS; j++ )
3246 //bufNolpS[i*31+k-1][j]=0;
3247 }
3248 }
3249
3250 } else { //if (BSTorBtree != 2) {
3251 // - ASCII code 095
3252 // - ASCII code 096 \
3253 // a ASCII code 097 / In total 26+1+1 radix instead of 27 to avoid +1 for each '-', code 096 not used.
3254 // z ASCII code 122
3255 // The hash for 'a_quadruplet_for_example' will be calculated for first 5 chars:
3256 // = (byte1-'_')*28*28*28*28 + (byte2-'_')*28*28*28 + (byte3-'_')*28*28 + (byte4-'_')*28 + (byte5-'_')
3257 // Hash slots are 28*28*28*28 = 17,210,368 each containing one 64bit pointer i.e. 8bytes in length.
3258 // Hash size = 17,210,368*8 = 137,682,944 bytes
3259 // When at end all these slots(17,210,368- Btrees) are traversed the outcome is a sorted wordlist - no need of sorting.
3260 //unsigned long long SeekPosition;
3261 //unsigned long long *PointerToSeekPosition;
3262 // The 64bit external pool will be addressed via fsetpos(fp_outRG, PointerToSeekPosition); similarly to bufend approach from r.13 - that is
3263 // bufend points to first(always following the last used btree leaf) free position in the pool.
3264 // For final stats all non-zero slots point to one btree.
3265 // printf( "Allocating HASH memory %s bytes ... ", _ui64toaKAZEcomma( (17210368*8) + 1 + 64 , 11ToDigits, 10) );
3266 // pointerflushUNALIGN = (char *)malloc( (17210368*8) + 1 + 64 );
3267 // Hash slots are 27bit = 2^27 = 134,217,728 each containing one 64bit pointer i.e. 8bytes in length.
3268 printf( "Allocating HASH memory %s bytes ... ", _ui64toaKAZEcomma( ((1<<HashInBITS)*8) + 1 + 64 , 11ToDigits, 10) );
3269 pointerflushUNALIGN = (char *)malloc( (1<<HashInBITS)*8 + 1 + 64 );
3270 if( pointerflushUNALIGN == NULL )
3271 { puts( "\nLeprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
3272 // r16
3273 pointerflush = pointerflushUNALIGN;
3274 //pointerflush = pointerflushUNALIGN + 64 - (((size_t)pointerflushUNALIGN) % 64); // 13_6+
3275 //offset=64-int((long)data&63);
3276 printf( "OK\n");
3277 // memset(pointerflush,0,17210368*8);
3278 memset(pointerflush,0,(1<<HashInBITS)*8);
3279     if (BSTorBtree == 2) {
3280 if( ( fp_outRG = fopen( "Leprechaun_64bit.hsh", "rb" ) ) == NULL )
3281 {
3282     HSHexist=0;
3283     if ( REUSE && ((HashInBITS-HashChunkSizeInBITS)==0) ) // Multiple-passes shouldn't be uploaded - it is meaningless, dump when only one
3284     pass.
3285     printf( "Leprechaun: Can't find file 'Leprechaun_64bit.hsh'.\n" );
3286 } else {
3287     HSHexist=1;
3288     fclose(fp_outRG);
3289 }
3290 if( ( fp_outRG = fopen( "Leprechaun_64bit.swp", "rb" ) ) == NULL )
3291 {
3292     SWPexist=0;
3293     if ( REUSE && ((HashInBITS-HashChunkSizeInBITS)==0) )
3294     printf( "Leprechaun: Can't find file 'Leprechaun_64bit.swp'.\n" );
3295 } else {
3296     SWPexist=1;
3297     fclose(fp_outRG);
3298 }
3299
3300 if ( REUSE && ((HashInBITS-HashChunkSizeInBITS)==0) && (SWPexist+HSHexist == 2) ) {
3301     REUSE=2;
3302     if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
3303     { printf( "Leprechaun: Can't create file %s\n", argv[2] ); return( 1 ); }

```

```

3304 }
3305
3306 if( ( fp_outRG = fopen( "Leprechaun_64bit.hsh", "rb" ) ) != NULL ) {
3307     if ( REUSE == 2 ) { // REUSE [
3308
3309 #if defined(_WIN32_ENVIRONMENT_)
3310     // 64bit:
3311     _lseeki64( fileno(fp_outRG), 0L, SEEK_END );
3312     size_inLINESIXFOUR = _telli64( fileno(fp_outRG) );
3313     _lseeki64( fileno(fp_outRG), 0L, SEEK_SET );
3314 #else
3315     // 64bit:
3316     fseeko( fp_outRG, 0L, SEEK_END );
3317     size_inLINESIXFOUR = ftello( fp_outRG );
3318     fseeko( fp_outRG, 0L, SEEK_SET );
3319 #endif /* defined(_WIN32_ENVIRONMENT_) */
3320 printf( "Uploading-n-Reusing 'Leprechaun_64bit.hsh' file: %s bytes\n", _ui64toaKAZEcomma(size_inLINESIXFOUR, IIT0aDigi ts, 10) );
3321
3322     fread( pointerflushUNALIGN, 1, (1<<HashInBITS)*8 + 1 + 64, fp_outRG ); // Notice that the actual size of .HSH file is not
    calculated since it won't work if not the same as during the creation.
3323     }
3324     fclose(fp_outRG);
3325 }
3326
3327 // Tag for the swap file is: LEPRECHAUNISH(ASCIIcode26)
3328 // or 14bytes, then when type of the swap is requested:
3329 // D:\KAZE-1\LEPREC-1>type Leprechaun_64bit.swp
3330 // LEPRECHAUNISH
3331 // D:\KAZE-1\LEPREC-1>
3332 size_in64_L14 = 1024 * (unsigned long long)Thunderwith + 14;
3333 BufEnd_64 = 0+14;
3334 // The tag plays two roles, the second to avoid existence of SeekPosition equal to 0. The 0 cannot be used as a free slot FLAG without the
    TAG.
3335
3336 /*
3337 The opentype argument is a string that controls how the file is opened and specifies attributes of the resulting stream. It must begin with
    one of the following sequences of characters:
3338
3339 'r'
3340     Open an existing file for reading only.
3341
3342 'w'
3343     Open the file for writing only. If the file already exists, it is truncated to zero length. Otherwise a new file is created.
3344
3345 'a'
3346     Open a file for append access; that is, writing at the end of file only. If the file already exists, its initial contents are unchanged
    and output to the stream is appended to the end of the file. Otherwise, a new, empty file is created.
3347
3348 'r+'
3349     Open an existing file for both reading and writing. The initial contents of the file are unchanged and the initial file position is at the
    beginning of the file.
3350
3351 'w+'
3352     Open a file for both reading and writing. If the file already exists, it is truncated to zero length. Otherwise, a new file is created.
3353
3354 'a+'
3355     Open or create file for both reading and appending. If the file exists, its initial contents are unchanged. Otherwise, a new file is
    created. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.
3356 */
3357
3358 // r16: Three+One conditions to reuse: Leprechaun_64bit.swp to exist, Not in multi-pass mode, W/w specified. The last one is the HASH
    upload to have been successful!
3359 if ( REUSE == 2 ) { // REUSE [
3360 if( ( fp_outRG = fopen( "Leprechaun_64bit.swp", "rb+" ) ) == NULL )
3361 { printf( "Leprechaun: Can't create file 'Leprechaun_64bit.swp'.\n" ); return( 1 ); }
3362
3363 #if defined(_WIN32_ENVIRONMENT_)
3364 // 64bit:
3365 _lseeki64( fileno(fp_outRG), 0L, SEEK_END );
3366 size_inLINESIXFOUR = _telli64( fileno(fp_outRG) );
3367 _lseeki64( fileno(fp_outRG), 0L, SEEK_SET );
3368 #else
3369 // 64bit:
3370 fseeko( fp_outRG, 0L, SEEK_END );
3371 size_inLINESIXFOUR = ftello( fp_outRG );
3372 fseeko( fp_outRG, 0L, SEEK_SET );
3373 #endif /* defined(_WIN32_ENVIRONMENT_) */
3374 printf( "Reusing 'Leprechaun_64bit.swp' file: %s bytes\n", _ui64toaKAZEcomma(size_inLINESIXFOUR, IIT0aDigi ts, 10) );
3375
3376 fsetpos(fp_outRG, &BufEnd_64); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ???! So DO-IT-OVER.
3377 }
3378 else { // REUSE
3379 if( ( fp_outRG = fopen( "Leprechaun_64bit.swp", "wb+" ) ) == NULL )
3380 { printf( "Leprechaun: Can't create file 'Leprechaun_64bit.swp'.\n" ); return( 1 ); }
3381 printf( "Allocating/ZEROing %s bytes swap file ... ", _ui64toaKAZEcomma(size_in64_L14, IIT0aDigi ts, 10) );
3382 fsetpos(fp_outRG, &fsetpos_ZERO); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ???! So DO-IT-OVER.
3383 memset(OneCkusterZER0ES, 0, 1024*4);
3384 for (ThunderwithL64_L14=0; ThunderwithL64_L14 < size_in64_L14/(1024*4); ThunderwithL64_L14++)
3385     fwrite(OneCkusterZER0ES, 1024*4, 1, fp_outRG);
3386 for (ThunderwithL64_L14=0; ThunderwithL64_L14 < size_in64_L14%(1024*4); ThunderwithL64_L14++)
3387     fwrite(&OneChar_ieByte, 1, 1, fp_outRG);
3388 fsetpos(fp_outRG, &fsetpos_ZERO); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ???! So DO-IT-OVER.
3389 fwrite(FileSwapTag, 13, 1, fp_outRG);

```



```

3385         fwrite(&EOFcode, 1, 1, fp_outRG);
3386 fsetpos(fp_outRG, &BufEnd_64); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ???! So DO-IT-OVER.
3387 printf( "OK\n");
3388     } // REUSE ]
3389     } else { // ##### 64bit memory manipulations [
3390 size_in64_L14 = 1024 * (unsigned long long)Thunderwith + 14 + 1 + 64;
3391 printf( "Allocating memory %luMB ... ", (size_in64_L14>>20)+1 );
3392 pointerflushUNALIGN_64 = (char *)malloc( size_in64_L14 );
3393 if( pointerflushUNALIGN_64 == NULL )
3394 { puts( "\nLeprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
3395 pointerflush_64 = pointerflushUNALIGN_64 + 64 - (((size_t)pointerflushUNALIGN_64) % 64); // 13_6+
3396 //offset=64-int((long)data&63);
3397 //memset(pointerflush_64, 0, 1024 * (unsigned long long)Thunderwith + 14);
3398 BufEnd_64 = (unsigned long long)pointerflush_64;
3399 /*
3400 printf( "BufEnd_64: %s\n", _ui64toaKAZEcomma(BufEnd_64, IIT0aDigits, 10) );
3401 printf( "pointerflush_64: %s\n", _ui64toaKAZEcomma(pointerflush_64, IIT0aDigits, 10) );
3402 pointerflush_64 = (char *)BufEnd_64;
3403 printf( "pointerflush_64: %s\n", _ui64toaKAZEcomma(pointerflush_64, IIT0aDigits, 10) );
3404 exit( 1);
3405 //BufEnd_64: 541,261,888
3406 //pointerflush_64: 541,261,888
3407 //pointerflush_64: 541,261,888
3408 */
3409 printf( "OK\n");
3410     } // ##### 64bit memory manipulations ]
3411 fprintf( fp_outLOG, "Leprechaun report:\n" );
3412 } //if (BStorBtree != 2) {
3413
3414
3415 // PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM
3416 (void) time(&t1);
3417
3418 // 16fixfix [
3419 PLE_words_INITflag = 0;
3420 PLE_words = 0;
3421 // 16fixfix ]
3422     Melnitcka = 0;
3423     WORDcount = 0; // Total word count i.e. for all files!
3424     WORDcountDistinct = 0;
3425     NumberOfFiles = 0;
3426     NumberOfLines = 0;
3427     FilesLEN = 0;
3428     LINE10len = 0;
3429 // Added in r.14+++++FIXFIX [
3430 NumberOfTrees=0; NumberOfHashCollisions=0;
3431 NumberOfLEAFs=0;
3432 WORDcountAttemptsToPut=0;
3433 LevelInCorona_Not_Counting_ROOT=0;
3434 // Added in r.14+++++FIXFIX ]
3435
3436     for( k = 0; k < size_in; k++ )
3437     {
3438         fread( &workbyte, 1, 1, fp_in );
3439         if( workbyte != 10 )
3440         { if( workbyte != 13 ) // NON UNIX
3441             { if( LINE10len < 255 ) { LINE10[ LINE10len ] = workbyte; }
3442               LINE10len++;
3443             }
3444         } else
3445         {
3446         }
3447     }
3448     else
3449     { if( 1 <= LINE10len && LINE10len <= 255 )
3450       { LINE10[ LINE10len ] = 0;
3451       METACOMMANDflag = 0;
3452       if ( strcmp(LINE10, "Leprechaun says x-gram inserting disabled for next files: ON\0") == 0 ) {DoNotInsertFlag = 1; METACOMMANDflag = 1;}
3453       if ( strcmp(LINE10, "Leprechaun says x-gram inserting disabled for next files: OFF\0") == 0 ) {DoNotInsertFlag = 0; METACOMMANDflag = 1;}
3454
3455       if( METACOMMANDflag == 0 )
3456       { // ~~~~~ IT IS a FILENAME not a METACOMMAND [
3457       if( ( fp_inLINE = fopen( LINE10, "rb" ) ) == NULL ) // Since r15FIXFIX+ a command [METACOMMAND] inside the .LST file is allowed:
3458           'Leprechaun says x-gram inserting disabled for next files: ON'
3459           // To allow again (which is default) use: 'Leprechaun says x-gram inserting disabled for
3460           next files: OFF'
3461       { printf( "Leprechaun: Can't open file %s\n", LINE10 ); return( 1 ); }
3462
3463       //fseek( fp_inLINE, 0L, SEEK_END ); //Rev. 12
3464       //size_inLINE = ftell( fp_inLINE ); //Rev. 12
3465       //fseek( fp_inLINE, 0L, SEEK_SET ); //Rev. 12
3466
3467       #if defined(_WIN32_ENVIRONMENT_)
3468           // 64bit:
3469       _lseeki64( fileno(fp_inLINE), 0L, SEEK_END );
3470       size_inLINESIXFOUR = _telli64( fileno(fp_inLINE) );
3471       _lseeki64( fileno(fp_inLINE), 0L, SEEK_SET );
3472       #else
3473

```

```

3471 // 64bit:
3472 fseeko( fp_inLINE, OL, SEEK_END );
3473 size_inLINESIXFOUR = ftello( fp_inLINE );
3474 fseeko( fp_inLINE, OL, SEEK_SET );
3475 #endif /* defined(_WIN32_ENVIRONMENT_) */
3476
3477 printf( "Size of Input TEXTual file: %s\n", _ui64toaKAZEcomma(size_inLINESIXFOUR, IIT0aDigi ts, 10) );
3478 FilesLEN = FilesLEN + size_inLINESIXFOUR;
3479 NumberOfFiles++;
3480
3481 //-----
3482 wrdlen = 0;
3483 for( i = 0; i < size_inLINESIXFOUR; i++ )
3484 {
3485     // ----- Buffering fread [
3486     if (workKoffset == -1) {
3487         if (i + 1024*128 < size_inLINESIXFOUR) {
3488             fread( &workK[0], 1, 1024*128, fp_inLINE );
3489             workKoffset = 0;
3490             workbyte = workK[workKoffset];
3491         } else
3492             fread( &workbyte, 1, 1, fp_inLINE );
3493     } else {
3494         workKoffset++;
3495         workbyte = workK[workKoffset];
3496         if (workKoffset == 1024*128 - 1) workKoffset = -1;
3497     }
3498     // ----- Buffering fread ]
3499
3500     if( isalpha( workbyte ) )
3501     {
3502         if( wrdlen < 31 )
3503         { wrd[ wrdlen ] = tolower( workbyte ); }
3504         wrdlen++;
3505     }
3506
3507     if ( workbyte < 'A' ) // Most characters are under alphabet - only one if
3508     {
3509         ElStupido:
3510             // This fragment is MIRRORed: #1 copy [
3511             if (workbyte == 10) {NumberOfLines++;}
3512
3513         // Quadruple! [
3514         // Sliding window for ' wrd ': The incoming string 'a lot of things must' becomes 'a_lot_of_things' and 'lot_of_things_must':
3515         // ain_t_that_a
3516         // didn_t_feel_a
3517         // i_didn_t_feel
3518         // t_feel_a_thing
3519         // t_that_a_cake
3520
3521         // 316
3522         // 00:17:55,859 --> 00:17:58,447
3523         // Ain't that a cake ? I didn't feel a thing !
3524
3525         if ( PLE_words_INITflag == 0 && ( PLE_words != 0 ) || ( PLE_words == 0 && wrdlen != 0 ) )
3526         if ( workbyte == '.' || workbyte == '!' || workbyte == '?' || workbyte == ':' || workbyte == ';' || workbyte == ',' || workbyte == '\t' ) {
3527             PLE_words_INITflag = 1;
3528         }
3529
3530         // Quadruple! ]
3531
3532         //r. 15fixfi x [
3533         if( wrdlen > 31 ) PLE_words_INITflag = 1;
3534         //r. 15fixfi x ]
3535
3536         //if ( 1 <= wrdlen && wrdlen <= LongestLineInclusive ) // Enforce no word with length greater than 31 with below line
3537         to enter x-lets.
3538         if ( 1 <= wrdlen && wrdlen <= 31 )
3539         {
3540             wrd[ wrdlen ] = 0;
3541             // OTKACHAM: 1<<17-1 gives 65536 i.e. '-' have had high priority than '<<'
3542             //Next line gives error due to mix of '&' and 'double'
3543             if ((WORDcount & ((1<<18)-1)) == 0)
3544             { //ui64toaKAZEzerocomma(WORDcount, IIT0aDigi ts, 10);
3545                 //printf( "Word count: %s(%lu/128 done)\r", IIT0aDigi ts, ((long long)i*100) / size_inLINESIXFOUR );
3546             }
3547             //Melni tchka;
3548             //Melni tchka = Melni tchka % 4;
3549             //if (Melni tchka == 0){ printf( "|; Word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEcomma(WORDcount, IIT0aDigi ts, 10),
3550                 _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, IIT0aDigi ts2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
3551             //if (Melni tchka == 1){ printf( "/; Word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEcomma(WORDcount, IIT0aDigi ts, 10),
3552                 _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, IIT0aDigi ts2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
3553             //if (Melni tchka == 2){ printf( "-; Word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEcomma(WORDcount, IIT0aDigi ts, 10),
3554                 _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, IIT0aDigi ts2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
3555             //if (Melni tchka == 3){ printf( "\\; Word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEcomma(WORDcount, IIT0aDigi ts, 10),
3556                 _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, IIT0aDigi ts2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
3557             Melni tchka = Melni tchka & 3; // 0 1 2 3: 00 01 10 11

```

```

3553         (void) time(&t4);
3554         if (t4 <= t1) {t4 = t1; t4++;}
3555 printf( "%s: %sP/s; Phrase count: %s of them %s distinct; Done: %lu/64\r", Auberge[Mel ni tchka++], _ui 64toaKAZEzerocomma(WORDcount/((int) t4-
t1), IIT0aDi gi ts3, 10)+(26-10), _ui 64toaKAZEcomma(WORDcount, IIT0aDi gi ts, 10), _ui 64toaKAZEcomma((unsigned long long)WORDcountDi stinct,
IIT0aDi gi ts2, 10), ((long long)i<<6) / size_inLINESIXFOUR );
3556     }
3557
3558 //14+++ [
3559 PLE_words++;
3560
3561 #ifdef singleton
3562         PLE_words_INITflag = 1;
3563 #endif
3564 #ifdef doubleton
3565 if (PLE_words == 1)
3566     strcpy( wrd1st, wrd );
3567 else if (PLE_words == 2) {
3568     strcpy( wrd2nd, wrd );
3569     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+1; // '_'
3570     //wrdlen = strlen(wrd);
3571     //if ( wrdlen <= 31 ) {
3572         if ( wrdlen <= LongestLineInclusive ) {
3573         strcpy(wrd, wrd1st);
3574         strcat(wrd, DelimiterUnderscore);
3575         strcat(wrd, wrd2nd);
3576     }
3577 }
3578 else {
3579     PLE_words = 2;
3580     strcpy( wrd1st, wrd2nd );
3581     strcpy( wrd2nd, wrd );
3582     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+1; // '_'
3583     //wrdlen = strlen(wrd);
3584     //if ( wrdlen <= 31 ) {
3585         if ( wrdlen <= LongestLineInclusive ) {
3586         strcpy(wrd, wrd1st);
3587         strcat(wrd, DelimiterUnderscore);
3588         strcat(wrd, wrd2nd);
3589     }
3590 }
3591 #endif
3592 #ifdef tripleton
3593 if (PLE_words == 1)
3594     strcpy( wrd1st, wrd );
3595 else if (PLE_words == 2)
3596     strcpy( wrd2nd, wrd );
3597 else if (PLE_words == 3) {
3598     strcpy( wrd3rd, wrd );
3599     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+1+1; // '_ _'
3600     //wrdlen = strlen(wrd);
3601     //if ( wrdlen <= 31 ) {
3602         if ( wrdlen <= LongestLineInclusive ) {
3603         strcpy(wrd, wrd1st);
3604         strcat(wrd, DelimiterUnderscore);
3605         strcat(wrd, wrd2nd);
3606         strcat(wrd, DelimiterUnderscore);
3607         strcat(wrd, wrd3rd);
3608     }
3609 }
3610 else {
3611     PLE_words = 3;
3612     strcpy( wrd1st, wrd2nd );
3613     strcpy( wrd2nd, wrd3rd );
3614     strcpy( wrd3rd, wrd );
3615     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+1+1; // '_ _'
3616     //wrdlen = strlen(wrd);
3617     //if ( wrdlen <= 31 ) {
3618         if ( wrdlen <= LongestLineInclusive ) {
3619         strcpy(wrd, wrd1st);
3620         strcat(wrd, DelimiterUnderscore);
3621         strcat(wrd, wrd2nd);
3622         strcat(wrd, DelimiterUnderscore);
3623         strcat(wrd, wrd3rd);
3624     }
3625 }
3626 #endif
3627 #ifdef quadrupleton
3628 // Quadruple! [
3629 if (PLE_words == 1)
3630     strcpy( wrd1st, wrd );
3631 else if (PLE_words == 2)
3632     strcpy( wrd2nd, wrd );
3633 else if (PLE_words == 3)
3634     strcpy( wrd3rd, wrd );
3635 else if (PLE_words == 4) {
3636     strcpy( wrd4th, wrd );
3637     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+1+1+1; // '_ _ _'
3638     //wrdlen = strlen(wrd);

```

```

3639     //if ( wrdlen <= 31 ) {
3640     if ( wrdlen <= LongestLineInclusive ) {
3641         strcpy(wrd, wrd1st);
3642         strcat(wrd, DelimiterUnderscore);
3643         strcat(wrd, wrd2nd);
3644         strcat(wrd, DelimiterUnderscore);
3645         strcat(wrd, wrd3rd);
3646         strcat(wrd, DelimiterUnderscore);
3647         strcat(wrd, wrd4th);
3648     }
3649 }
3650 else {
3651     PLE_words = 4;
3652     strcpy( wrd1st, wrd2nd );
3653     strcpy( wrd2nd, wrd3rd );
3654     strcpy( wrd3rd, wrd4th );
3655     strcpy( wrd4th, wrd );
3656     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+1+1+1; // ' _ ' _ ' _ '
3657     //wrdlen = strlen(wrd);
3658     //if ( wrdlen <= 31 ) {
3659     if ( wrdlen <= LongestLineInclusive ) {
3660         strcpy(wrd, wrd1st);
3661         strcat(wrd, DelimiterUnderscore);
3662         strcat(wrd, wrd2nd);
3663         strcat(wrd, DelimiterUnderscore);
3664         strcat(wrd, wrd3rd);
3665         strcat(wrd, DelimiterUnderscore);
3666         strcat(wrd, wrd4th);
3667     }
3668 }
3669 // Quadruple! ]
3670 #endif
3671 #ifdef quintuplet
3672 if (PLE_words == 1)
3673     strcpy( wrd1st, wrd );
3674 else if (PLE_words == 2)
3675     strcpy( wrd2nd, wrd );
3676 else if (PLE_words == 3)
3677     strcpy( wrd3rd, wrd );
3678 else if (PLE_words == 4)
3679     strcpy( wrd4th, wrd );
3680 else if (PLE_words == 5) {
3681     strcpy( wrd5th, wrd );
3682     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+1+1+1+1; // ' _ ' _ ' _ ' _ '
3683     //wrdlen = strlen(wrd);
3684     //if ( wrdlen <= 31 ) {
3685     if ( wrdlen <= LongestLineInclusive ) {
3686         strcpy(wrd, wrd1st);
3687         strcat(wrd, DelimiterUnderscore);
3688         strcat(wrd, wrd2nd);
3689         strcat(wrd, DelimiterUnderscore);
3690         strcat(wrd, wrd3rd);
3691         strcat(wrd, DelimiterUnderscore);
3692         strcat(wrd, wrd4th);
3693         strcat(wrd, DelimiterUnderscore);
3694         strcat(wrd, wrd5th);
3695     }
3696 }
3697 else {
3698     PLE_words = 5;
3699     strcpy( wrd1st, wrd2nd );
3700     strcpy( wrd2nd, wrd3rd );
3701     strcpy( wrd3rd, wrd4th );
3702     strcpy( wrd4th, wrd5th );
3703     strcpy( wrd5th, wrd );
3704     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+1+1+1+1; // ' _ ' _ ' _ ' _ '
3705     //wrdlen = strlen(wrd);
3706     //if ( wrdlen <= 31 ) {
3707     if ( wrdlen <= LongestLineInclusive ) {
3708         strcpy(wrd, wrd1st);
3709         strcat(wrd, DelimiterUnderscore);
3710         strcat(wrd, wrd2nd);
3711         strcat(wrd, DelimiterUnderscore);
3712         strcat(wrd, wrd3rd);
3713         strcat(wrd, DelimiterUnderscore);
3714         strcat(wrd, wrd4th);
3715         strcat(wrd, DelimiterUnderscore);
3716         strcat(wrd, wrd5th);
3717     }
3718 }
3719 #endif
3720 #ifdef sextuplet
3721 if (PLE_words == 1)
3722     strcpy( wrd1st, wrd );
3723 else if (PLE_words == 2)
3724     strcpy( wrd2nd, wrd );
3725 else if (PLE_words == 3)
3726     strcpy( wrd3rd, wrd );

```

```

3727 else if (PLE_words == 4)
3728     strcpy( wrd4th, wrd );
3729 else if (PLE_words == 5)
3730     strcpy( wrd5th, wrd );
3731 else if (PLE_words == 6) {
3732     strcpy( wrd6th, wrd );
3733     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+1+1+1+1+1; // ' _ ' _ ' _ ' _ ' _ '
3734     //wrdlen = strlen(wrd);
3735     //if ( wrdlen <= 31 ) {
3736         if ( wrdlen <= LongestLineInclusive ) {
3737             strcpy(wrd, wrd1st);
3738             strcat(wrd, DelimiterUnderscore);
3739             strcat(wrd, wrd2nd);
3740             strcat(wrd, DelimiterUnderscore);
3741             strcat(wrd, wrd3rd);
3742             strcat(wrd, DelimiterUnderscore);
3743             strcat(wrd, wrd4th);
3744             strcat(wrd, DelimiterUnderscore);
3745             strcat(wrd, wrd5th);
3746             strcat(wrd, DelimiterUnderscore);
3747             strcat(wrd, wrd6th);
3748         }
3749     }
3750 else {
3751     PLE_words = 6;
3752     strcpy( wrd1st, wrd2nd );
3753     strcpy( wrd2nd, wrd3rd );
3754     strcpy( wrd3rd, wrd4th );
3755     strcpy( wrd4th, wrd5th );
3756     strcpy( wrd5th, wrd6th );
3757     strcpy( wrd6th, wrd );
3758     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+1+1+1+1+1; // ' _ ' _ ' _ ' _ ' _ '
3759     //wrdlen = strlen(wrd);
3760     //if ( wrdlen <= 31 ) {
3761         if ( wrdlen <= LongestLineInclusive ) {
3762             strcpy(wrd, wrd1st);
3763             strcat(wrd, DelimiterUnderscore);
3764             strcat(wrd, wrd2nd);
3765             strcat(wrd, DelimiterUnderscore);
3766             strcat(wrd, wrd3rd);
3767             strcat(wrd, DelimiterUnderscore);
3768             strcat(wrd, wrd4th);
3769             strcat(wrd, DelimiterUnderscore);
3770             strcat(wrd, wrd5th);
3771             strcat(wrd, DelimiterUnderscore);
3772             strcat(wrd, wrd6th);
3773         }
3774     }
3775 #endif
3776 #ifdef septuplet
3777 if (PLE_words == 1)
3778     strcpy( wrd1st, wrd );
3779 else if (PLE_words == 2)
3780     strcpy( wrd2nd, wrd );
3781 else if (PLE_words == 3)
3782     strcpy( wrd3rd, wrd );
3783 else if (PLE_words == 4)
3784     strcpy( wrd4th, wrd );
3785 else if (PLE_words == 5)
3786     strcpy( wrd5th, wrd );
3787 else if (PLE_words == 6)
3788     strcpy( wrd6th, wrd );
3789 else if (PLE_words == 7) {
3790     strcpy( wrd7th, wrd );
3791     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+1+1+1+1+1+1; //
3792     //wrdlen = strlen(wrd);
3793     //if ( wrdlen <= 31 ) {
3794         if ( wrdlen <= LongestLineInclusive ) {
3795             strcpy(wrd, wrd1st);
3796             strcat(wrd, DelimiterUnderscore);
3797             strcat(wrd, wrd2nd);
3798             strcat(wrd, DelimiterUnderscore);
3799             strcat(wrd, wrd3rd);
3800             strcat(wrd, DelimiterUnderscore);
3801             strcat(wrd, wrd4th);
3802             strcat(wrd, DelimiterUnderscore);
3803             strcat(wrd, wrd5th);
3804             strcat(wrd, DelimiterUnderscore);
3805             strcat(wrd, wrd6th);
3806             strcat(wrd, DelimiterUnderscore);
3807             strcat(wrd, wrd7th);
3808         }
3809     }
3810 else {
3811     PLE_words = 7;
3812     strcpy( wrd1st, wrd2nd );
3813     strcpy( wrd2nd, wrd3rd );

```

```

3814     strcpy( wrd3rd, wrd4th );
3815     strcpy( wrd4th, wrd5th );
3816     strcpy( wrd5th, wrd6th );
3817     strcpy( wrd6th, wrd7th );
3818     strcpy( wrd7th, wrd );
3819     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+1+1+1+1+1; //
- - - - -
3820     //wrdlen = strlen(wrd);
3821     //if ( wrdlen <= 31 ) {
3822     if ( wrdlen <= LongestLineInclusive ) {
3823     strcpy(wrd, wrd1st);
3824     strcat(wrd, DelimiterUnderscore);
3825     strcat(wrd, wrd2nd);
3826     strcat(wrd, DelimiterUnderscore);
3827     strcat(wrd, wrd3rd);
3828     strcat(wrd, DelimiterUnderscore);
3829     strcat(wrd, wrd4th);
3830     strcat(wrd, DelimiterUnderscore);
3831     strcat(wrd, wrd5th);
3832     strcat(wrd, DelimiterUnderscore);
3833     strcat(wrd, wrd6th);
3834     strcat(wrd, DelimiterUnderscore);
3835     strcat(wrd, wrd7th);
3836     }
3837 }
3838 #endif
3839 #ifndef octupleton
3840 if ( PLE_words == 1)
3841     strcpy( wrd1st, wrd );
3842 else if ( PLE_words == 2)
3843     strcpy( wrd2nd, wrd );
3844 else if ( PLE_words == 3)
3845     strcpy( wrd3rd, wrd );
3846 else if ( PLE_words == 4)
3847     strcpy( wrd4th, wrd );
3848 else if ( PLE_words == 5)
3849     strcpy( wrd5th, wrd );
3850 else if ( PLE_words == 6)
3851     strcpy( wrd6th, wrd );
3852 else if ( PLE_words == 7)
3853     strcpy( wrd7th, wrd );
3854 else if ( PLE_words == 8) {
3855     strcpy( wrd8th, wrd );
3856     wrdlen =
- - - - -
        strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+strlen(wrd8th)+1+1+1+1+1+1; //
- - - - -
3857     //wrdlen = strlen(wrd);
3858     //if ( wrdlen <= 31 ) {
3859     if ( wrdlen <= LongestLineInclusive ) {
3860     strcpy(wrd, wrd1st);
3861     strcat(wrd, DelimiterUnderscore);
3862     strcat(wrd, wrd2nd);
3863     strcat(wrd, DelimiterUnderscore);
3864     strcat(wrd, wrd3rd);
3865     strcat(wrd, DelimiterUnderscore);
3866     strcat(wrd, wrd4th);
3867     strcat(wrd, DelimiterUnderscore);
3868     strcat(wrd, wrd5th);
3869     strcat(wrd, DelimiterUnderscore);
3870     strcat(wrd, wrd6th);
3871     strcat(wrd, DelimiterUnderscore);
3872     strcat(wrd, wrd7th);
3873     strcat(wrd, DelimiterUnderscore);
3874     strcat(wrd, wrd8th);
3875     }
3876 }
3877 else {
3878     PLE_words = 8;
3879     strcpy( wrd1st, wrd2nd );
3880     strcpy( wrd2nd, wrd3rd );
3881     strcpy( wrd3rd, wrd4th );
3882     strcpy( wrd4th, wrd5th );
3883     strcpy( wrd5th, wrd6th );
3884     strcpy( wrd6th, wrd7th );
3885     strcpy( wrd7th, wrd8th );
3886     strcpy( wrd8th, wrd );
3887     wrdlen =
- - - - -
        strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+strlen(wrd8th)+1+1+1+1+1+1; //
- - - - -
3888     //wrdlen = strlen(wrd);
3889     //if ( wrdlen <= 31 ) {
3890     if ( wrdlen <= LongestLineInclusive ) {
3891     strcpy(wrd, wrd1st);
3892     strcat(wrd, DelimiterUnderscore);
3893     strcat(wrd, wrd2nd);
3894     strcat(wrd, DelimiterUnderscore);
3895     strcat(wrd, wrd3rd);
3896     strcat(wrd, DelimiterUnderscore);

```


Leprechaun_x-leton revision **16FIXFIX**; Both Physical/INTERNAL and Virtual/EXTERNAL modes pure 64bit (32bit code along with 64bit) + Multi-Pass Mode + REUSE; Update: 2013-Mar-31


```

4065   strcat(wrd, Delimi terUnderscore);
4066   strcat(wrd, wrd10th);
4067   }
4068 }
4069 #endi f
4070 //14+++ ]
4071
4072 //14+++ [
4073 #i fdef si ngleton
4074         i f ( ( PLE_words == 1 ) && ( 1 <= wrdlen ) && ( wrdlen <= 31 ) ) {
4075 #endi f
4076 #i fdef doubleton
4077         i f ( ( PLE_words == 2 ) && ( 5 <= wrdlen ) && ( wrdlen <= 41 ) ) {
4078 #endi f
4079 #i fdef tri pleton
4080         i f ( ( PLE_words == 3 ) && ( 9 <= wrdlen ) && ( wrdlen <= 41 ) ) {
4081 #endi f
4082 #i fdef quadrupleton
4083         i f ( ( PLE_words == 4 ) && ( 13 <= wrdlen ) && ( wrdlen <= 51 ) ) {
4084 #endi f
4085 #i fdef qui ntupleton
4086         i f ( ( PLE_words == 5 ) && ( 17 <= wrdlen ) && ( wrdlen <= 61 ) ) {
4087 #endi f
4088 #i fdef sextupleton
4089         i f ( ( PLE_words == 6 ) && ( 21 <= wrdlen ) && ( wrdlen <= 71 ) ) {
4090 #endi f
4091 #i fdef septupleton
4092         i f ( ( PLE_words == 7 ) && ( 25 <= wrdlen ) && ( wrdlen <= 81 ) ) {
4093 #endi f
4094 #i fdef octupleton
4095         i f ( ( PLE_words == 8 ) && ( 29 <= wrdlen ) && ( wrdlen <= 91 ) ) {
4096 #endi f
4097 #i fdef nonupleton
4098         i f ( ( PLE_words == 9 ) && ( 33 <= wrdlen ) && ( wrdlen <= 101 ) ) {
4099 #endi f
4100 #i fdef decupleton
4101         i f ( ( PLE_words == 10 ) && ( 37 <= wrdlen ) && ( wrdlen <= 111 ) ) {
4102 #endi f
4103 //14+++ ]
4104 WORDcount++;
4105 i f (BSTorBtree < 2) {
4106
4107     LetterOffset = (int)( wrd[0] - 'a' ) * 31 + (wrdlen-1); // 0..805
4108     //BufStart = pointerflush + LetterOffset * LetterBuffer; // OLD
4109
4110     BufStart = pointerflush + (int)( wrd[0] - 'a' ) * WHOLEl etter_BufferSi ze + OffsetsI nBuffer[wrdlen-1];
4111     // Above line and Below line are equal
4112     //BufStart = pointerflush + (LetterOffset / 31) * WHOLEl etter_BufferSi ze + OffsetsI nBuffer[LetterOffset % 31];
4113
4114     //Si ot = KuxHash3pl us(wrd)<<2; //13++
4115     //Si ot = FNV1A_Hash_SHI FTl ess_XORl ess(wrd)<<2; //13+++
4116     //Si ot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13++++
4117     //Si ot = FNV1A_Hash_4_OCTETS_31(wrd, wrdlen>>2)<<2; //13+++++
4118 /*
4119 i f (wrdlen<=19) // 4x4+3=19
4120     Si ot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13++++
4121 el se
4122     Si ot = FNV1A_Hash_8_OCTETS(wrd, wrdlen>>3)<<2; //13+++++
4123 */
4124 //i f (wrdlen<=19) // 4x4+3=19 i.e. l ast contains 7 clashes
4125 //     Si ot = FNV1A_Hash_Granul ari ty(wrd, wrdlen>>2, 2)<<2; //13+++++
4126 //el se
4127 //     Si ot = FNV1A_Hash_Granul ari ty(wrd, wrdlen>>3, 3)<<2; //13+++++
4128
4129     //Si ot = FNV1A_Hash_8_OCTETS(wrd, wrdlen>>3)<<2; //13_7p
4130     //Si ot = HashFNV1A_unrol led_Fi nal (wrd, wrdlen)<<2; //13_7p
4131     //Si ot = HashAl fal fa_HALF(wrd, wrdlen)<<2; //13_7p
4132     //Si ot = Hash_Al fal fa(wrd, wrdlen)<<2; //13_7p
4133 //     Si ot = Si xti nsensi ti ve(wrd, wrdlen)<<2; //13_7p
4134     Si ot = FNV1A_Hash_Jesteress(wrd, wrdlen)<<2; //13_7p
4135 //     Si ot = FNV1A_Hash_Jester(wrd, wrdlen)<<2; //13_7p
4136
4137 /*
4138 hashAl fal fa = 7;
4139 for(i Al fal fa = 0; i Al fal fa < (wrdlen & -2); i Al fal fa += 2) {
4140     hashAl fal fa = (17+9) * ((17+9) * hashAl fal fa + (wrd[i Al fal fa])) + (wrd[i Al fal fa+1]);
4141 }
4142 i f(wrdlen & 1)
4143     hashAl fal fa = (17+9) * hashAl fal fa + (wrd[wrdlen-1]);
4144
4145     Si ot = (( hashAl fal fa ^ (hashAl fal fa >> 16) ) & 8191)<<2; //13_7p
4146 */
4147
4148     memcpy( &PseudoLi nkedPoi nter, BufStart+Si ot, 4 );
4149 //; Li ne 917
4150 // mov     edx, DWORD PTR [eax+ebp]
4151 // add     esp, 4
4152 // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical

```

```

4153 //PseudoLinkedPointer = (unsigned long)*(long *) (BufStart+Slot);
4154 //; Line 919
4155 // mov     edx, DWORD PTR [eax+ebp]
4156 // add     esp, 4
4157 //         while (count--) {
4158 //             *(char *)dst = *(char *)src;
4159 //             dst = (char *)dst + 1;
4160 //             src = (char *)src + 1;
4161 //         }
4162
4163 if (BSTorBtree == 0)
4164 {
4165 // ===== BST fragment [
4166 // if (PseudoLinkedPointer == 0) // means EMPTY-SLOT
4167 // {
4168 //     //if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 < (GRMBLhi11[(int)wrdlen] * LetterBuffer)/31 ) // OLD slower
4169 //     if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST instead
of LL
4170 // {
4171 //     memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
4172 //; Line 932
4173 // mov     DWORD PTR [eax+ebp], esi
4174 // // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
4175 // /*(long *) (BufStart+Slot) = *(long *)&bufend[LetterOffset];
4176 //; Line 936
4177 // mov     DWORD PTR [eax+ebp], esi
4178
4179 // Below 3 lines are commented due to experiment below malloc which shows that allocated memory is ZEROed.
4180 // memcpy( bufend[LetterOffset], &BufStart[NumberOfSlots*4], 4 ); // means next exists not: Means PseudoLinkedPointerL =
0
4181 //; Line 940
4182 // mov     ecx, DWORD PTR [ebp+32768]
4183 // // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
4184 // /*(long *)bufend[LetterOffset] = *(long *)&BufStart[NumberOfSlots*4];
4185 //; Line 944
4186 // mov     ecx, DWORD PTR [ebp+32768]
4187 // //bufend[LetterOffset] = bufend[LetterOffset] + 4;
4188 // memcpy( bufend[LetterOffset], &BufStart[NumberOfSlots*4], 4 ); // means next exists not: Means PseudoLinkedPointerR =
0
4189 // bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4; // + 4 due to above commenting
4190 // memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4191 // //bufNoWpS[LetterOffset][Slot]++; // ?! crashes
4192 // bufend[LetterOffset] = bufend[LetterOffset] + wrdlen;
4193 // if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4194 // }
4195 // else
4196 // { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4197 // fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4198 // fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, IIT0aDiGiTs, 10) );
4199 // fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORdcount, IIT0aDiGiTs, 10), _ui64toaKAZEcomma((unsigned long
long)WORdcountDistinct, IIT0aDiGiTs2, 10) );
4200 // fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4201 // fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4202 // fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4203 // fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORdcountAttemptsToPut, IIT0aDiGiTs,
10) );
4204 // for( k = 1; k < 32; k++ )
4205 // { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%s utilization\n", _ui64toaKAZEcomma(k, IIT0aDiGiTs, 10)+(26-
2), _ui64toaKAZEcomma((MAXusedBuffer[k]>>10)+1, IIT0aDiGiTs2, 10)+(26-5), _ui64toaKAZEcomma((((GRMBLhi11[(int)k] *
LetterBuffer)/31)>>10)+1, IIT0aDiGiTs3, 10)+(26-5), _ui64toaKAZEcomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLhi11[(int)k] *
LetterBuffer)/31), IIT0aDiGiTs4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
4206 // }
4207 // fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4208 // fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4209 // return( 1 );
4210 // }
4211 // }
4212 // else // means USED-SLOT
4213 // { FoundInLinkedLiSt = 0;
4214 //     while (PseudoLinkedPointer != 0 && FoundInLinkedLiSt == 0)
4215 //     {
4216 //         if (memcmp(PseudoLinkedPointer+4+4, wrd, wrdlen) == 0)
4217 //         // while ( --count && *(char *)buf1 == *(char *)buf2 ) {
4218 //         //     buf1 = (char *)buf1 + 1;
4219 //         //     buf2 = (char *)buf2 + 1;
4220 //         // }
4221 //         // return( *((unsigned char *)buf1) - *((unsigned char *)buf2) );
4222 //         { FoundInLinkedLiSt = 1;
4223 //         }
4224 //     } else // i.e < or >
4225 //     {
4226 //         if (memcmp(PseudoLinkedPointer+4+4, wrd, wrdlen) > 0)
4227 //         memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
4228 //         else
4229 //         {
4230 //             PseudoLinkedPointer = PseudoLinkedPointer + 4;
4231 //             memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );

```

```

4232     }
4233
4234     if (PseudoLinkedPointerNEW == 0)
4235     {
4236         //if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 < (GRMBLhi11[(int)wrdlen] * LetterBuffer)/31 ) // OLD slower
4237         if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] )
4238         { memcpy( PseudoLinkedPointer, &bufend[LetterOffset], 4 );
4239             // Below 3 lines are commented due to experiment below malloc which shows that allocated memory is ZEROed.
4240             //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not
4241             //bufend[LetterOffset] = bufend[LetterOffset] + 4;
4242             //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not
4243             bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4; // + 4 due to above commenting
4244             memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4245             //bufNoWpS[LetterOffset][Slot]++; // ?! crashes
4246             bufend[LetterOffset] = bufend[LetterOffset] + wrdlen;
4247             if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4248         }
4249     }
4250     else
4251     { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4252     }
4251 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4252 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, IIT0aDigits, 10) );
4253 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, IIT0aDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, IIT0aDigits2, 10) );
4254 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4255 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4256 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4257 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, IIT0aDigits,
10) );
4258 for( k = 1; k < 32; k++ )
4259 { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s utilization\n", _ui64toaKAZEzerocomma(k, IIT0aDigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, IIT0aDigits2, 10)+(26-5), _ui64toaKAZEzerocomma((((GRMBLhi11[(int)k] *
LetterBuffer)/31)>>10)+1, IIT0aDigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLhi11[(int)k] *
LetterBuffer)/31), IIT0aDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRE=24
4260 }
4261 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4262 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4263 return( 1 );
4264 }
4265 }
4266 PseudoLinkedPointer = PseudoLinkedPointerNEW;
4267 }
4268 WORDcountAttemptsToPut++;
4269 } // while
4270 }
4271 // ===== BST fragment ]
4272 } else
4273 {
4274 // ##### B-tree order 3 fragment [
4275 //
4276 // LEAF structure: [LeftPointer][MiddlePointer][RightPointer][LeftWord][RightWord]
4277 //                  4bytes      4bytes      4bytes      * wrdlen      * wrdlen
4278 //
4279 // ALL B-tree order 3 fragment consists of 3 sub-fragments:
4280 // 1] Search 2] if Search failed Trasierascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search 3] Insert Iterative
4281
4282 // 1] Search [ _____1407 line in C - see below: whole Search in assembler_____
4283     if (PseudoLinkedPointer == 0) // means EMPTY-SLOT
4284     {
4285         if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST
instead of LL; + more(see LEAF)
4286         {
4287             memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
4288             bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
4289             memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4290             bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
4291             if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4292         }
4293     }
4294     else
4295     { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4296     }
4295 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4296 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, IIT0aDigits, 10) );
4297 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, IIT0aDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, IIT0aDigits2, 10) );
4298 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4299 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4300 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4301 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, IIT0aDigits, 10)
);
4302 for( k = 1; k < 32; k++ )
4303 { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s utilization\n", _ui64toaKAZEzerocomma(k, IIT0aDigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, IIT0aDigits2, 10)+(26-5), _ui64toaKAZEzerocomma((((GRMBLhi11[(int)k] *
LetterBuffer)/31)>>10)+1, IIT0aDigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLhi11[(int)k] *
LetterBuffer)/31), IIT0aDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRE=24
4304 }
4305 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4306 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");

```

```

4307         return( 1 );
4308     }
4309         FoundInLinkedList = 1;
4310     }
4311     else // means USED-SLOT
4312     { FoundInLinkedList = 0;
4313       while (PseudoLinkedListPointer != 0 && FoundInLinkedList == 0)
4314       {
4315         // ***** 'P W P' section 1 [
4316         // LW: existence check if ( *(char *) (PseudoLinkedListPointer+4+4) != 0 )
4317         // RW: existence check if ( *(char *) (PseudoLinkedListPointer+4+4+wordlen) != 0 )
4318         // here ALWAYS LW exists: no need for existence check - line below
4319         // if ( *(char *) (PseudoLinkedListPointer+4+4) != 0 )
4320         if (memcmp(PseudoLinkedListPointer+4+4, wrd, wordlen) > 0) // go LP
4321         { memcpy( &PseudoLinkedListPointerNEW, PseudoLinkedListPointer + 0, 4 ); //LP
4322           PseudoLinkedListPointer = PseudoLinkedListPointerNEW;
4323         }
4324         else if (memcmp(PseudoLinkedListPointer+4+4, wrd, wordlen) < 0) // go RP or MP
4325         { // RW existence check - line below:
4326           if ( *(char *) (PseudoLinkedListPointer+4+4+wordlen) != 0 ) // RW exists
4327             { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
4328               // ++++++
4329               // ***** 'P W P' section 2 [
4330               // LW: existence check if ( *(char *) (PseudoLinkedListPointer+4+4) != 0 )
4331               // RW: existence check if ( *(char *) (PseudoLinkedListPointer+4+4+wordlen) != 0 )
4332               // here ALWAYS RW exists: no need for existence check - line below
4333               // if ( *(char *) (PseudoLinkedListPointer+4+4+wordlen) != 0 )
4334               if (memcmp(PseudoLinkedListPointer+4+4+wordlen, wrd, wordlen) > 0) // go MP
4335               { memcpy( &PseudoLinkedListPointerNEW, PseudoLinkedListPointer + 4, 4 ); //MP
4336                 PseudoLinkedListPointer = PseudoLinkedListPointerNEW;
4337               }
4338               else if (memcmp(PseudoLinkedListPointer+4+4+wordlen, wrd, wordlen) < 0) // go RP
4339               { // No ?W after RW - go RP
4340                 memcpy( &PseudoLinkedListPointerNEW, PseudoLinkedListPointer + 4 + 4, 4 ); //RP
4341                 PseudoLinkedListPointer = PseudoLinkedListPointerNEW;
4342               }
4343               else FoundInLinkedList = 1; // wrd is RW
4344               WORDcountAttemptsToPut++;
4345               // ***** 'P W P' section 2 ]
4346               // ++++++
4347               }
4348               else // RW empty - go MP
4349               { memcpy( &PseudoLinkedListPointerNEW, PseudoLinkedListPointer + 4, 4 ); //MP
4350                 PseudoLinkedListPointer = PseudoLinkedListPointerNEW;
4351               }
4352             }
4353             else FoundInLinkedList = 1; // wrd is LW
4354             WORDcountAttemptsToPut++;
4355             // ***** 'P W P' section ]
4356             } // while
4357             WORDcountAttemptsToPut--; // - 1 due to BST way of counting i.e. direct hash hit is not counted only successors
4358           }
4359           // 1] Search ] _____ 1484 line in C - see below: whole Search in assembler_____
4360
4361           /*
4362           ; Line 1397
4363           jmp $L2139
4364           $L2042:
4365           ; Line 1408
4366           test edx, edx
4367           jne SHORT $L2110
4368           ; Line 1410
4369           mov ecx, DWORD PTR _bufend$[esp+esi*4+892340]
4370           mov edi, DWORD PTR _GRMBLFoolAgain$[esp+ebx*4+892340]
4371           lea edx, DWORD PTR _bufend$[esp+esi*4+892340]
4372           lea esi, DWORD PTR [ebx+ebx+12]
4373           sub esi, ebp
4374           add esi, ecx
4375           cmp esi, edi
4376           mov DWORD PTR tv4122[esp+892340], edx
4377           jae $L2113
4378           ; Line 1412
4379           mov DWORD PTR [eax+ebp], ecx
4380           ; Line 1413
4381           lea eax, DWORD PTR [ecx+12]
4382           ; Line 1436
4383           jmp $L2749
4384           $L2110:
4385           ; Line 1437
4386           mov DWORD PTR _FoundInLinkedList$[esp+892340], 0
4387           npad11
4388           $L2141:
4389           ; Line 1438
4390           mov eax, DWORD PTR _FoundInLinkedList$[esp+892340]
4391           test eax, eax
4392           jne $L2142
4393           ; Line 1445
4394           lea ebp, DWORD PTR [edx+12]

```

```

4395 mov ecx, ebx
4396 lea edi, DWORD PTR _wrd$[esp+892340]
4397 mov esi, ebp
4398 xor eax, eax
4399 repe cmpsb
4400 je SHORT $L2682
4401 sbb eax, eax
4402 sbb eax, -1
4403 $L2682:
4404 test eax, eax
4405 jle SHORT $L2143
4406 ; Line 1447
4407 mov edx, DWORD PTR [edx]
4408 ; Line 1449
4409 jmp $L2153
4410 $L2143:
4411 mov ecx, ebx
4412 lea edi, DWORD PTR _wrd$[esp+892340]
4413 mov esi, ebp
4414 xor eax, eax
4415 repe cmpsb
4416 je SHORT $L2640
4417 sbb eax, eax
4418 sbb eax, -1
4419 $L2640:
4420 test eax, eax
4421 jge SHORT $L2145
4422 ; Line 1451
4423 mov cl, BYTE PTR [edx+ebx+12]
4424 test cl, cl
4425 lea eax, DWORD PTR [edx+ebx+12]
4426 je SHORT $L2147
4427 ; Line 1459
4428 mov ecx, ebx
4429 lea edi, DWORD PTR _wrd$[esp+892340]
4430 mov esi, eax
4431 xor ebp, ebp
4432 repe cmpsb
4433 je SHORT $L2695
4434 sbb ebp, ebp
4435 sbb ebp, -1
4436 $L2695:
4437 test ebp, ebp
4438 jle SHORT $L2148
4439 ; Line 1461
4440 mov edx, DWORD PTR [edx+4]
4441 ; Line 1463
4442 jmp SHORT $L2151
4443 $L2148:
4444 mov esi, eax
4445 mov ecx, ebx
4446 lea edi, DWORD PTR _wrd$[esp+892340]
4447 xor eax, eax
4448 repe cmpsb
4449 je SHORT $L2642
4450 sbb eax, eax
4451 sbb eax, -1
4452 $L2642:
4453 test eax, eax
4454 jge SHORT $L2150
4455 ; Line 1466
4456 mov edx, DWORD PTR [edx+8]
4457 ; Line 1468
4458 jmp SHORT $L2151
4459 $L2150:
4460 mov DWORD PTR _FoundInLi nkedLi st$[esp+892340], 1
4461 $L2151:
4462 ; Line 1469
4463 mov ecx, DWORD PTR _WORDcountAttemptsToPut$[esp+892340]
4464 mov eax, DWORD PTR _WORDcountAttemptsToPut$[esp+892344]
4465 add ecx, 1
4466 adc eax, 0
4467 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892340], ecx
4468 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892344], eax
4469 ; Line 1473
4470 jmp SHORT $L2153
4471 $L2147:
4472 ; Line 1475
4473 mov edx, DWORD PTR [edx+4]
4474 ; Line 1478
4475 jmp SHORT $L2153
4476 $L2145:
4477 mov DWORD PTR _FoundInLi nkedLi st$[esp+892340], 1
4478 $L2153:
4479 ; Line 1479
4480 mov esi, DWORD PTR _WORDcountAttemptsToPut$[esp+892340]
4481 mov ecx, DWORD PTR _WORDcountAttemptsToPut$[esp+892344]
4482 add esi, 1

```

```

4483 adc ecx, 0
4484 test edx, edx
4485 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892340], esi
4486 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892344], ecx
4487 jne $L2141
4488 $L2142:
4489 ; Line 1482
4490 mov edx, DWORD PTR _WORDcountAttemptsToPut$[esp+892340]
4491 mov ecx, DWORD PTR _WORDcountAttemptsToPut$[esp+892344]
4492 or eax, -1
4493 add edx, eax
4494 adc ecx, eax
4495 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892344], ecx
4496 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892340], edx
4497 $L2139:
4498 */
4499
4500 if (FoundInLinkedList == 0)
4501 {
4502 // 2] if Search failed Trasi rascht (pushing in stack PseudoLinkedPointer (visited LEAFs)) Search [
4503 // 'TracingSearch' is the same as 'Search' except that adds the trail in my simulated stack,
4504 // the goal is not to waste time in 'Search' by dealing with no needed trail in case of not 'Insert'.
4505 // Simulated stack contains pairs of 'Address of ParentLEAF' + 'Offset of ParentPointer in ParentLEAF' i.e. 0 for LP, 4 for MP, 8 for RP'.
4506 // 'Offset ...' saves unnecessary comparisons of NEWword which after splitting goes up.
4507 memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
4508 StackPtr = 0;
4509 while (PseudoLinkedPointer != 0)
4510 {
4511 // ***** 'P W P' section [
4512 // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
4513 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
4514 // here ALWAYS LW exists: no need for existence check - line below
4515 // if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
4516 if (memcmp(PseudoLinkedPointer+4+4+4, wrd, wordlen) > 0) // go LP
4517 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 0, 4 ); //LP
4518 if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 ); }
4519 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
4520 BSTstack[StackPtr] = 0; ++StackPtr; //LPoffset=0; MPoffset=4; RPoffset=8;
4521 PseudoLinkedPointer = PseudoLinkedPointerNEW;
4522 }
4523 else if (memcmp(PseudoLinkedPointer+4+4+4, wrd, wordlen) < 0) // go RP or MP
4524 { // RW existence check - line below:
4525 if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 ) // RW exists
4526 { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
4527 // ***** 'P W P' section 2 [
4528 // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
4529 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
4530 // here ALWAYS RW exists: no need for existence check - line below
4531 // if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
4532 if (memcmp(PseudoLinkedPointer+4+4+4+wordlen, wrd, wordlen) > 0) // go MP
4533 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
4534 if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 ); }
4535 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
4536 BSTstack[StackPtr] = 4; ++StackPtr; //LPoffset=0; MPoffset=4; RPoffset=8;
4537 PseudoLinkedPointer = PseudoLinkedPointerNEW;
4538 }
4539 else if (memcmp(PseudoLinkedPointer+4+4+4+wordlen, wrd, wordlen) < 0) // go RP
4540 { // No ?W after RW - go RP
4541 memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
4542 if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 ); }
4543 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
4544 BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0; MPoffset=4; RPoffset=8;
4545 PseudoLinkedPointer = PseudoLinkedPointerNEW;
4546 }
4547 }
4548 else FoundInLinkedList = 1; // wrd is RW
4549 // ***** 'P W P' section 2 ]
4550 // *****
4551 }
4552 else // RW empty - go MP
4553 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
4554 if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 ); }
4555 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
4556 BSTstack[StackPtr] = 4; ++StackPtr; //LPoffset=0; MPoffset=4; RPoffset=8;
4557 PseudoLinkedPointer = PseudoLinkedPointerNEW;
4558 }
4559 }
4560 else FoundInLinkedList = 1; // wrd is LW
4561 // ***** 'P W P' section ]
4562 } // while
4563 // 2] if Search failed Trasi rascht (pushing in stack PseudoLinkedPointer (visited LEAFs)) Search [
4564
4565 // 3] Insert Iterative [
4566 // There are total 4 situations:
4567 // Case #1: Outer NODE(including ROOT) [ ][ ][ ][LW][ ]
4568 // Case #2: Outer NODE(including ROOT) [ ][ ][ ][LW][RW] Split Occurs -----
4569 // Case #3: ROOT [LP][MP][ ][LW][ ] 'wrdUP' (wordlen bytes)
4570 // Case #4: Inner NODE(including ROOT) [LP][MP][RP][LW][RW] Split Occurs --- | &

```

```

4571 // | | 'PseudoLinkedPointerNEW' (ptr to NEW LEAF)
4572 // There are total 2 situations for PARENT LEAF: <----- ARE GOING UP
4573 // Case #3: [LP][MP][ ] [LW][ ]
4574 // Case #4: [LP][MP][RP][LW][RW] Split Occurs
4575
4576 // ~ First deal alongly with the OUTER NODE(LEAF) where Search stopped i.e Case #1 & Case #2:
4577 POffsetInLEAF = BSTstack[--StackPtr];
4578 PseudoLinkedPointer = BSTstack[--StackPtr];
4579 // NOTE: ONE LEAF IS FULL ONLY WHEN LAST CELL FOR KEY(here RW) EXISTS!
4580 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+wrklen) != 0 )
4581 if ( *(char *) (PseudoLinkedPointer+4+4+wrklen) != 0 ) // If LEAF is full: Case #2
4582 { SplitOccured = 1; WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4583 // ALlocate NEW LEAF:
4584 if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrklen + 4 + 4 + 4 < GRMBLFoolAgain((int)wrklen) ) // +4 more for BST
instead of LL; + more(see LEAF)
4585 {
4586 memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
4587 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
4588 bufend[LetterOffset] = bufend[LetterOffset] + 2*wrklen;
4589 if (MAXusedBuffer[wrklen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrklen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4590 }
4591 else
4592 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4593 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4594 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, IIT0aDigits, 10) );
4595 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, IIT0aDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, IIT0aDigits2, 10) );
4596 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4597 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4598 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4599 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, IIT0aDigits, 10)
);
4600 for( k = 1; k < 32; k++ )
4601 { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s utilization\n", _ui64toaKAZEzerocomma(k, IIT0aDigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, IIT0aDigits2, 10)+(26-5), _ui64toaKAZEzerocomma((((GRMBLhiI[(int)k] *
LetterBuffer)/31)>>10)+1, IIT0aDigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLhiI[(int)k] *
LetterBuffer)/31), IIT0aDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
4602 }
4603 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4604 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4605 return( 1 );
4606 }
4607 if (POffsetInLEAF == 0) // wrd < LW
4608 {
4609 memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrklen ); // LW up
4610 memcpy( PseudoLinkedPointer+4+4+4, wrd, wrklen ); // wrd go to OLD LEAF
4611 memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrklen, wrklen ); // RW go to NEW LEAF
4612 *(char *) (PseudoLinkedPointer+4+4+4+wrklen) = 0; // RW mark unused in OLD LEAF
4613 }
4614 if (POffsetInLEAF == 4) // LW < wrd < RW
4615 {
4616 memcpy( wrdUP, wrd, wrklen ); // wrd up
4617 memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrklen, wrklen ); // RW go to NEW LEAF
4618 *(char *) (PseudoLinkedPointer+4+4+4+wrklen) = 0; // RW mark unused in OLD LEAF
4619 }
4620 if (POffsetInLEAF == 8) // wrd > RW
4621 {
4622 memcpy( wrdUP, PseudoLinkedPointer+4+4+4+wrklen, wrklen ); // RW up
4623 *(char *) (PseudoLinkedPointer+4+4+4+wrklen) = 0; // RW mark unused in OLD LEAF
4624 memcpy( PseudoLinkedPointerNEW+4+4+4, wrd, wrklen ); // wrd go to NEW LEAF
4625 }
4626 }
4627 else // If LEAF is not full: Case #1
4628 { SplitOccured = 0; WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4629 if (POffsetInLEAF == 0) // wrd < [LW][ ] so [LW][ ] -> [ ][LW] -> [wrd][LW]
4630 {
4631 memcpy( PseudoLinkedPointer+4+4+4+wrklen, PseudoLinkedPointer+4+4+4, wrklen );
4632 memcpy( PseudoLinkedPointer+4+4+4, wrd, wrklen );
4633 }
4634 if (POffsetInLEAF == 4) // wrd > [LW][ ] so [LW][ ] -> [LW][wrd]
4635 {
4636 memcpy( PseudoLinkedPointer+4+4+4+wrklen, wrd, wrklen );
4637 }
4638 }
4639
4640 if (SplitOccured != 0)
4641 {
4642 // ~ Second deal with the INNER NODE(S) i.e Case #3 & Case #4:
4643 while (StackPtr != 0 || SplitOccured != 0)
4644 {
4645 // 'PseudoLinkedPointerNEW' is new LEAF to be inserted
4646 // 'wrdUP' is NEW word to be inserted
4647 if (StackPtr != 0)
4648 {
4649 POffsetInLEAF = BSTstack[--StackPtr];
4650 PseudoLinkedPointer = BSTstack[--StackPtr];
4651 if ( *(char *) (PseudoLinkedPointer+4+4+4+wrklen) != 0 ) // If LEAF is full: Case #4

```

```

4652 { SplitOccured = 1;
4653     memcpy( wrdUPold, wrdUP, wrdlen ); // LW up
4654     PseudoLinkedPointerNEWold = PseudoLinkedPointerNEW;
4655     // Allocate NEW LEAF:
4656     if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wordlen + 4 + 4 + 4 < GRMBLFullAgain((int)wordlen) ) // +4 more for BST
instead of LL; + more(see LEAF)
4657     {
4658         memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
4659         bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
4660         bufend[LetterOffset] = bufend[LetterOffset] + 2*wordlen;
4661         if (MAXUsedBuffer[wordlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXUsedBuffer[wordlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4662     }
4663     else
4664     { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4665     fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4666     fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, IIT0aDigits, 10) );
4667     fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, IIT0aDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, IIT0aDigits2, 10) );
4668     fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4669     fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4670     fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4671     fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, IIT0aDigits, 10)
);
4672     for( k = 1; k < 32; k++ )
4673     { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%s utilization\n", _ui64toaKAZEzerocomma(k, IIT0aDigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXUsedBuffer[k]>>10)+1, IIT0aDigits2, 10)+(26-5), _ui64toaKAZEzerocomma((((GRMBLhi11[(int)k] *
LetterBuffer)/31)>>10)+1, IIT0aDigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXUsedBuffer[k]*100)/((GRMBLhi11[(int)k] *
LetterBuffer)/31), IIT0aDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIREd=24
4674 }
4675 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4676     fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4677     return( 1 );
4678 }
4679 if (POffsetInLEAF == 0) // wrdUPold < LW
4680 {
4681     memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
4682     memcpy( PseudoLinkedPointer+4+4+4, wrdUPold, wrdlen ); // wrdUPold go to OLD LEAF
4683     memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wordlen, wrdlen ); // RW go to NEW LEAF
4684     *(char *) (PseudoLinkedPointer+4+4+4+wordlen) = 0; // RW mark unused in OLD LEAF
4685     // [LP](PseudoLinkedPointerNEWold)[MP][RP](wrdUPold)[LW][RW] -----
4686     // pair [LW] PseudoLinkedPointerNEW goes up
4687     // PseudoLinkedPointer: PseudoLinkedPointerNEW:
4688     // [LP](PseudoLinkedPointerNEWold)[wrdUPold] [MP][RP][LW] <----
4689     // no need to put zero in RP because logic is based on words existence:
4690     memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+4, 4 );
4691     memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
4692     memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEWold, 4 );
4693 }
4694 if (POffsetInLEAF == 4) // LW < wrdUPold < RW
4695 {
4696     memcpy( wrdUP, wrdUPold, wrdlen ); // wrdUPold up
4697     memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wordlen, wrdlen ); // RW go to NEW LEAF
4698     *(char *) (PseudoLinkedPointer+4+4+4+wordlen) = 0; // RW mark unused in OLD LEAF
4699     // [LP][MP](PseudoLinkedPointerNEWold)[RP][LW](wrdUPold)[RW] -----
4700     // pair [wrdUPold] PseudoLinkedPointerNEW goes up
4701     // PseudoLinkedPointer: PseudoLinkedPointerNEW:
4702     // [LP][MP][LW] (PseudoLinkedPointerNEWold)[RP][RW] <----
4703     // no need to put zero in RP because logic is based on words existence:
4704     memcpy( PseudoLinkedPointerNEW+0, &PseudoLinkedPointerNEWold, 4 );
4705     memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
4706 }
4707 if (POffsetInLEAF == 8) // wrdUPold > RW
4708 {
4709     memcpy( wrdUP, PseudoLinkedPointer+4+4+4+wordlen, wrdlen ); // RW up
4710     *(char *) (PseudoLinkedPointer+4+4+4+wordlen) = 0; // RW mark unused in OLD LEAF
4711     memcpy( PseudoLinkedPointerNEW+4+4+4, wrdUPold, wrdlen ); // wrdUPold go to NEW LEAF
4712     // [LP][MP][RP](PseudoLinkedPointerNEWold)[LW][RW](wrdUPold) -----
4713     // pair [RW] PseudoLinkedPointerNEW goes up
4714     // PseudoLinkedPointer: PseudoLinkedPointerNEW:
4715     // [LP][MP][LW] [RP](PseudoLinkedPointerNEWold)[wrdUPold] <----
4716     // no need to put zero in RP because logic is based on words existence:
4717     memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+8, 4 );
4718     memcpy( PseudoLinkedPointerNEW+4, &PseudoLinkedPointerNEWold, 4 );
4719 }
4720 }
4721 else // If LEAF is not full: Case #3
4722 { SplitOccured = 0;
4723     if (POffsetInLEAF == 0) // wrdUP < [LW][ ] so [LW][ ] -> [ ][LW] -> [wrdUP][LW]
4724     {
4725         memcpy( PseudoLinkedPointer+4+4+4+wordlen, PseudoLinkedPointer+4+4+4, wrdlen );
4726         memcpy( PseudoLinkedPointer+4+4+4, wrdUP, wrdlen );
4727         // [LP][MP][ ] -> [LP][ ][MP] -> [LP][np][MP]
4728         memcpy( PseudoLinkedPointer+8, PseudoLinkedPointer+4, 4 );
4729         memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEW, 4 );
4730     }
4731     if (POffsetInLEAF == 4) // wrdUP > [LW][ ] so [LW][ ] -> [LW][wrdUP]
4732     {

```



```

4733 memcpy( PseudoLinkedPointer+4+4+wordlen, wrdUP, wrdlen );
4734 // [LP][MP][] -> [LP][MP][np]
4735 memcpy( PseudoLinkedPointer+8, &PseudoLinkedPointerNEW, 4 );
4736 }
4737 break;
4738 }
4739 }
4740 else // Empty stack means ROOT and more over ROOT is already splitted(Case #4 is off)
4741 {
4742 // If LEAF is not full: Case #3
4743 // THIS IS WHERE A NEW(SECOND) LEAF 'PseudoLinkedPointerROOT' must be allocated:
4744 if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wordlen + 4 + 4 + 4 < GRMBLPoolAgain((int)wordlen) ) // +4 more for BST
instead of LL; + more(see LEAF)
4745 {
4746 memcpy( &PseudoLinkedPointerROOT, bufend[LetterOffset], 4 );
4747 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // +4 due to above commenting
4748 memcpy( bufend[LetterOffset], wrdUP, wrdlen );
4749 bufend[LetterOffset] = bufend[LetterOffset] + 2*wordlen;
4750 if (MAXusedBuffer[wordlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wordlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4751 }
4752 else
4753 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4754 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4755 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, IIToADigits, 10) );
4756 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, IIToADigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, IIToADigits2, 10) );
4757 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4758 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4759 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4760 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, IIToADigits, 10)
);
4761 for( k = 1; k < 32; k++)
4762 { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s utilization\n", _ui64toaKAZEzerocomma(k, IIToADigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, IIToADigits2, 10)+(26-5), _ui64toaKAZEzerocomma((((GRMBLhi1[(int)k] *
LetterBuffer)/31)>>10)+1, IIToADigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLhi1[(int)k] *
LetterBuffer)/31), IIToADigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRE=24
4763 }
4764 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4765 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4766 return( 1 );
4767 }
4768 // Here -- 'PseudoLinkedPointerROOT' --
4769 // | (wrdUP)
4770 // 'PseudoLinkedPointer' <-- --> 'PseudoLinkedPointerNEW'
4771 // (LW) (RW)
4772 memcpy( PseudoLinkedPointerROOT, &PseudoLinkedPointer, 4 ); // LP
4773 memcpy( PseudoLinkedPointerROOT+4, &PseudoLinkedPointerNEW, 4 ); // MP
4774 // Here must NEW ROOT be updated i.e. HASH table(SLOT) must point it:
4775 memcpy( BufStart+Slot, &PseudoLinkedPointerROOT, 4 );
4776 break; //because it is ROOT without split
4777 }
4778 } // while
4779 } //if (SplitOccured != 0)
4780 // 3] Insert Iterative ]
4781 //if (FoundInLinkedList == 0)
4782 // ##### B-tree order 3 ]
4783 //if (BSTorBtree == 0)
4784 } else { //if (BSTorBtree != 2) {
4785 // External Btrees [
4786
4787 // _ ASCII code 095
4788 // ^ ASCII code 096 \
4789 // a ASCII code 097 / In total 26+1+1 radix instead of 27 to avoid +1 for each '_', code 096 not used.
4790 // z ASCII code 122
4791 // The hash for 'a_quadruplet_for_example' will be calculated for first 5 chars:
4792 // = (byte1-'_')*28*28*28*28 + (byte2-'_')*28*28*28 + (byte3-'_')*28*28 + (byte4-'_')*28 + (byte5-'_')
4793 // Hash slots are 28*28*28*28 = 17,210,368 each containing one 64bit pointer i.e. 8bytes in length.
4794 // Hash size = 17,210,368*8 = 137,682,944 bytes
4795 // When at end all these slots(17,210,368- Btrees) are traversed the outcome is a sorted wordlist - no need of sorting.
4796
4797 // D:\_KAZE_new-stuff\Leprechaun_quadruplet_r14_minus\Leprechaun_quadruplet.exe GRAFFITH_2048.lst GRAFFITH_2048.wrd z
4798 // Leprechaun(Fast Greedy Word-Ripper), rev. 14_minus_quadruplet, written by Svalqatchx.
4799 // Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
4800 // Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
4801 // also the performance of a 3-way hash + 6,602,752 B-Trees of order 3,
4802 // also the performance of a 1-way hash + 17,210,368 external B-Trees of order 3.
4803 // Size of input file with files for Leprechauning: 42140
4804 // Allocating HASH memory 137,683,009 bytes ... OK
4805 // Allocating/ZEROing 1,047,566 bytes swap file ... OK
4806 // Size of Input TEXTual file: 33,470,581
4807 // |; Word count: 3,045,077 of them 0 distinct; Done: 64/64
4808 // ...
4809 // Size of Input TEXTual file: 17,403,406
4810 // /; Word count: 2,710,601,882 of them 0 distinct; Done: 64/64
4811 // Bytes per second performance: 17,694,246B/s
4812 // Words per second performance: 1,730,907W/s
4813 // Leprechaun: Done.

```

```

4814 //
4815 // Leprechaun report:
4816 // Number Of Trees(GREATER THE BETTER): 1,646,004
4817
4818 // TO DO - it is long overdue: at last make sort stage at end unnecessary - only traversing-and-dumping!
4819
4820 BufStart = pointerflush;
4821 // Slot = ((wrd[0]-' ')*28*28*28*28 + (wrd[1]-' ')*28*28*28 + (wrd[2]-' ')*28*28 + (wrd[3]-' ')*28 + (wrd[4]-' '))<<3;
4822 // Slot = FNV1A_Hash_Jesteress_27bit(wrd, wrdlen)<<3; // Commented since r.14+++ because of passes.
4823 Slot = FNV1A_Hash_Jesteress_27bit(wrd, wrdlen);
4824
4825 // Bug fix for all r.14+++ and below! [
4826 memcpy( &wrd[(LongestLineInclusive+1+4)-4], &NULLsForWRD, 4 );
4827 // Bug fix for all r.14+++ and below! ]
4828
4829 // Example: HashInBITS-HashChunkSizeInBITS=2
4830 // HashInBITS = 5
4831 // HashChunkSizeInBITS = 3
4832 // RipPasses = 1<<(HashInBITS-HashChunkSizeInBITS) i.e. 1<<2 which is 4 i.e. 32 slots with 4 passes 8 slots each.
4833 // 00??? 5bits 0-7
4834 // 01??? 5bits 8-15
4835 // 10??? 5bits 16-23
4836 // 11??? 5bits 24-31
4837 if ( (Slot>>HashChunkSizeInBITS) == RipPasses ) {
4838 Slot = Slot<<3;
4839
4840 //Slot = 0; // One Tree only!
4841 memcpy( &PseudoLinkedPointer_64, BufStart+Slot, 8 );
4842
4843 // ##### B-tree order 3 fragment 64bit [
4844 //
4845 // LEAF structure: [LeftPointer][MiddlePointer][RightPointer][LeftWord][RightWord]
4846 //                4bytes      4bytes      4bytes      * wrdlen      * wrdlen
4847 //
4848 // ALL B-tree order 3 fragment consists of 3 sub-fragments:
4849 // 1] Search 2] if Search failed Trasi rascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search 3] Insert Iterative
4850
4851 // LEAF_64 structure: [LeftPointer][MiddlePointer][RightPointer][LeftWord] [RightWord]
4852 //                   8bytes      8bytes      8bytes      * LongestLineInclusive+1+4 * LongestLineInclusive+1+4
4853 //                   * * * * *
4854 // Note: In order to use one fread(and strcmp) a NULL postfix for LeftWord, RightWord i.e. LeftWord_Length=len(LeftWord)+1 a kinda stupid
4855 // choice ...
4856 // Note: BufEnd_64 in fact is the first free position after the BUFFER END!
4857 // 1] Search [
4858 //         if (PseudoLinkedPointer_64 == 0) // means EMPTY-SLOT
4859 //         {
4860 if ( REUSE != 2 ) { // REUSE [ // This line comes since r16
4861 if (DoNotInsertFlag == 0)
4862 { // This line comes since r15FIXFIX+ [#####
4863 //if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wordlen + 4 + 4 + 4 < GRMBLFoolAgain((int)wordlen) ) // +4 more for BST
4864 //instead of LL; + more(see LEAF)
4865 // {
4866 //     memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
4867 //     bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
4868 //     memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4869 //     bufend[LetterOffset] = bufend[LetterOffset] + 2*wordlen;
4870 //     if (MAXusedBuffer[wordlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wordlen] = (unsigned
4871 //long)(bufend[LetterOffset] - BufStart);}
4872 // }
4873 // if( 8 + 8 + 8 + 2*(LongestLineInclusive+1+4) < size_in64_L14 - (BufEnd_64-(unsigned long long)pointerflush_64) ) // the longest
4874 //wordlen is LongestLineInclusive but actual is LongestLineInclusive(+ CR char)
4875 // {
4876 //     memcpy( BufStart+Slot, &BufEnd_64, 8 );
4877 //     BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
4878 //     if (BSTorBtree == 2) {
4879 //         fsetpos(fp_outRG, &BufEnd_64);
4880 //         fwrite(wrd, wrdlen, 1, fp_outRG); WORDcountDistinct++;
4881 //         //fwrite(&OneChar_1Byte, 1, 1, fp_outRG); // Write ZERO ASCII code
4882 //         // r.14++ The above line was commented because the pool is already ZEROed.
4883 //     } else { // ##### 64bit memory manipulations [
4884 //         memcpy( (char *)BufEnd_64, wrd, wrdlen ); WORDcountDistinct++;
4885 //         // ##### 64bit memory manipulations ]
4886 //         BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
4887 //         //fsetpos(fp_outRG, &BufEnd_64);
4888 //     }
4889 // }
4890 // else
4891 // { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4892 //   fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, IIToADigits, 10), _ui64toaKAZEcomma((unsigned long
4893 //long)WORDcountDistinct, IIToADigits2, 10) );
4894 //   fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, IIToADigits, 10) );
4895 //   fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, IIToADigits, 10)
4896 //);
4897 //   fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4898 //   fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4899 //   return( 1 );
4900 // }
4901 // }
4902 // FoundInLinkedList = 1;

```

Lebrechaun x-leton revision 16FIXFIX: Both Physical/INTERNAL and Virtual/EXTERNAL modes pure 64bit (32bit code along with 64bit) + Multi-Pass Mode + REUSE: Update: 2013-Mar-31

```

4984     BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0;MPoffset=8;RPoffset=16;
4985     // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4986     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
4987     //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
4988     // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4989     // [ //r.14+
4990     memcpy( &PseudoLinkedPointer_64, &LEAF[8], 8 );
4991     // ] //r.14+
4992     }
4993     // else if (memcmp(PseudoLinkedPointer+4+4+4+wordlen, wrd, wordlen) < 0) // go RP
4994     else if (strcmp(KAZE13(FourGramL, wrd) < 0) // go RP
4995     { // No ?W after RW - go RP
4996         memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
4997         PseudoLinkedPointer = PseudoLinkedPointerNEW;
4998     }
4999     {
5000     // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5001     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8; //RP
5002     // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5003     if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5004     BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
5005     BSTstack[StackPtr] = 16; ++StackPtr; //LPoffset=0;MPoffset=8;RPoffset=16;
5006     // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5007     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5008     //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
5009     // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5010     // [ //r.14+
5011     memcpy( &PseudoLinkedPointer_64, &LEAF[8 + 8], 8 );
5012     // ] //r.14+
5013     }
5014     else { FoundInLinkedList = 1; // wrd is RW
5015     // Counter [
5016     if (BSTorBtree == 2) {
5017     fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5018     }
5019     memcpy( &CounterOccurrences, &FourGramL[(LongestLineInclusi ve+1+4)-4], 4 );
5020
5021     // r16 [
5022     if ( REUSE == 2 ) {
5023     if (*argv[k_FIX] == 'W')
5024         fprintf(fp_out, "%s\t%s\r\n", _ui64toaKAZEzerocomma(CounterOccurrences+1, 11ToDigits2, 10)+(26-9), wrd);
5025     //WORDcountBOTTOM++;
5026     if (*argv[k_FIX] == 'w')
5027         fprintf(fp_out, "%s\r\n", wrd); //WORDcountBOTTOM++;
5028     }
5029     // r16 ]
5030
5031     if ( REUSE != 2 ) { // r16
5032     if (CounterOccurrences<99999999) CounterOccurrences++;
5033     memcpy( &FourGramL[(LongestLineInclusi ve+1+4)-4], &CounterOccurrences, 4 );
5034     // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5035     //fwrite(&FourGramL[0], (LongestLineInclusi ve+1+4), 1, fp_outRG);
5036     // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5037     // [ //r.14+
5038     memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusi ve+1+4)], &FourGramL[0], (LongestLineInclusi ve+1+4) );
5039     if (BSTorBtree == 2) {
5040     fwrite(&LEAF[0], 8+8+8+2*(LongestLineInclusi ve+1+4), 1, fp_outRG);
5041     } else { // ##### 64bit memory manipulations [
5042     memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLineInclusi ve+1+4) );
5043     } // ##### 64bit memory manipulations ]
5044     // ] //r.14+
5045     } // r16
5046     // Counter ]
5047     }
5048     WORDcountAttemptsToPut++;
5049     // ***** 'P W P' section 2 ]
5050     // ++++++
5051     }
5052     else // RW empty - go MP
5053     { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
5054     PseudoLinkedPointer = PseudoLinkedPointerNEW;
5055     }
5056     {
5057     // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5058     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8; //MP
5059     // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5060     if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5061     BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
5062     BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0;MPoffset=8;RPoffset=16;
5063     // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5064     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5065     //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
5066     // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5067     // [ //r.14+
5068     memcpy( &PseudoLinkedPointer_64, &LEAF[8], 8 );
5069     // ] //r.14+
5070     }
5071     }

```

```

5071         else { FoundInLinkedList = 1; // wrd is LW
5072         // Counter [
5073         if (BSTorBtree == 2) {
5074         fsetpos(&fp_outRG, &PseudoLinkedListPointerAUX_64);
5075         }
5076         memcpy( &CounterOccurrences, &FourGramL[(LongestLineInclusive+1+4)-4], 4 );
5077
5078         // r16 [
5079         if ( REUSE == 2 ) {
5080         if (*argv[k_FIX] == 'W')
5081             fprintf(fp_out, "%s\\t%s\\r\\n", _ui64toaKAZeZeroComma(CounterOccurrences+1, 11ToDigits2, 10)+(26-9), wrd);
5082         //WORDcountBOTTOM++;
5083         if (*argv[k_FIX] == 'w')
5084             fprintf(fp_out, "%s\\r\\n", wrd); //WORDcountBOTTOM++;
5085         }
5086         // r16 ]
5087
5088         if ( REUSE != 2 ) { // r16
5089         if (CounterOccurrences<99999999) CounterOccurrences++;
5090         memcpy( &FourGramL[(LongestLineInclusive+1+4)-4], &CounterOccurrences, 4 );
5091         // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5092         //fwrite(&FourGramL[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5093         // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5094         // [ //r.14+
5095         memcpy( &LEAF[8 + 8 + 8], &FourGramL[0], (LongestLineInclusive+1+4) );
5096         if (BSTorBtree == 2) {
5097         fwrite(&LEAF[0], 8+8+8*(LongestLineInclusive+1+4), 1, fp_outRG);
5098         } else { // ##### 64bit memory manipulations [
5099         memcpy( (char *)PseudoLinkedListPointerAUX_64, &LEAF[0], 8+8+8*2*(LongestLineInclusive+1+4) );
5100         } // ##### 64bit memory manipulations ]
5101         // ] //r.14+
5102         } // r16
5103         // Counter ]
5104         }
5105         WORDcountAttemptsToPut++;
5106         // ***** 'P W P' section ]
5107         } // while
5108         WORDcountAttemptsToPut--; // - 1 due to BST way of counting i.e. direct hash hit is not counted only successors
5109         }
5110         // 1] Search ]
5111         if ( REUSE != 2 ) { // REUSE [ // This line comes since r16
5112         if (DoNotInsertFlag == 0) { // This line comes since r15FIXFIX+
5113         if (FoundInLinkedList == 0)
5114         {
5115         /*
5116         // ===== [ The whole section/sub-fragment 2 is commented due to great time differences for Internal_vs_External memory
5117         // accesses - it is far more cheap to have the STACK overhead (moved to sub-fragment 1) ] ===== [
5118         // 2] if Search failed Trasi rascht (pushing in stack PseudoLinkedListPointer(visited LEAFs)) Search [
5119         // 'TracingSearch' is the same as 'Search' except that adds the trail in my simulated stack,
5120         // the goal is not to waste time in 'Search' by dealing with no needed trail in case of not 'Insert'.
5121         // Simulated stack contains pairs of 'Address of ParentLEAF' + 'Offset of ParentPointer in ParentLEAF i.e. 0 for LP, 4 for MP, 8 for RP'.
5122         // 'Offset ...' saves unnecessary comparisons of NEWword which after splitting goes up.
5123         memcpy( &PseudoLinkedListPointer, BufStart+Slot, 4 );
5124         StackPtr = 0;
5125         while (PseudoLinkedListPointer != 0)
5126         {
5127         // ***** 'P W P' section [
5128         // LW: existence check if ( *(char *) (PseudoLinkedListPointer+4+4+4) != 0 )
5129         // RW: existence check if ( *(char *) (PseudoLinkedListPointer+4+4+4+wordlen) != 0 )
5130         // here ALWAYS LW exists: no need for existence check - line below
5131         // if ( *(char *) (PseudoLinkedListPointer+4+4+4) != 0 )
5132         if (memcmp(PseudoLinkedListPointer+4+4+4, wrd, wordlen) > 0) // go LP
5133         { memcpy( &PseudoLinkedListPointerNEW, PseudoLinkedListPointer + 0, 4 ); //LP
5134         if (StackPtr > 8192*3-1-1) { printf( "\\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\\n" ); return( 13 ); }
5135         BSTstack[StackPtr] = PseudoLinkedListPointer; ++StackPtr; //pt to visited leaf
5136         BSTstack[StackPtr] = 0; ++StackPtr; //LPoffset=0; MPoffset=4; RPoffset=8;
5137         PseudoLinkedListPointer = PseudoLinkedListPointerNEW;
5138         }
5139         else if (memcmp(PseudoLinkedListPointer+4+4+4, wrd, wordlen) < 0) // go RP or MP
5140         { // RW existence check - line below:
5141         if ( *(char *) (PseudoLinkedListPointer+4+4+4+wordlen) != 0 ) // RW exists
5142         { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
5143         // ***** 'P W P' section 2 [
5144         // LW: existence check if ( *(char *) (PseudoLinkedListPointer+4+4+4) != 0 )
5145         // RW: existence check if ( *(char *) (PseudoLinkedListPointer+4+4+4+wordlen) != 0 )
5146         // here ALWAYS RW exists: no need for existence check - line below
5147         // if ( *(char *) (PseudoLinkedListPointer+4+4+4+wordlen) != 0 )
5148         if (memcmp(PseudoLinkedListPointer+4+4+4+wordlen, wrd, wordlen) > 0) // go MP
5149         { memcpy( &PseudoLinkedListPointerNEW, PseudoLinkedListPointer + 4, 4 ); //MP
5150         if (StackPtr > 8192*3-1-1) { printf( "\\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\\n" ); return( 13 ); }
5151         BSTstack[StackPtr] = PseudoLinkedListPointer; ++StackPtr; //pt to visited leaf
5152         BSTstack[StackPtr] = 4; ++StackPtr; //LPoffset=0; MPoffset=4; RPoffset=8;
5153         PseudoLinkedListPointer = PseudoLinkedListPointerNEW;
5154         }
5155         else if (memcmp(PseudoLinkedListPointer+4+4+4+wordlen, wrd, wordlen) < 0) // go RP
5156         { // No ?W after RW - go RP

```

```

5157         memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
5158     if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5159     BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5160     BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
5161     PseudoLinkedPointer = PseudoLinkedPointerNEW;
5162 }
5163 else FoundInLinkedList = 1; // wrd is RW
5164 // ***** 'P W P' section 2 ]
5165 // ++++++
5166 }
5167 else // RW empty - go MP
5168 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
5169 if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5170 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5171 BSTstack[StackPtr] = 4; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
5172 PseudoLinkedPointer = PseudoLinkedPointerNEW;
5173 }
5174 }
5175 else FoundInLinkedList = 1; // wrd is LW
5176 // ***** 'P W P' section ]
5177 } // while
5178 // 2] if Search failed Trasi rascht (pushing in stack PseudoLinkedPointer(visited LEAFs)) Search ]
5179 // ===== [ The whole section/sub-fragment 2 is commented due to great time differences for Internal_vs_External memory
5180 // accesses - it is far more cheap to have the STACK overhead (moved to sub-fragment 1) ] ===== ]
5181 */
5182 // 3] Insert Iterative [
5183 // There are total 4 situations:
5184 // Case #1: Outer NODE(including ROOT) [ ][ ][ ][LW][ ]
5185 // Case #2: Outer NODE(including ROOT) [ ][ ][ ][LW][RW] Split Occurs -----
5186 // Case #3: ROOT [LP][MP][ ][LW][ ]
5187 // Case #4: Inner NODE(including ROOT) [LP][MP][RP][LW][RW] Split Occurs ---
5188 // |
5189 // There are total 2 situations for PARENT LEAF: <----- ARE GOING UP
5190 // Case #3: [LP][MP][ ][LW][ ]
5191 // Case #4: [LP][MP][RP][LW][RW] Split Occurs
5192
5193 // ~ First deal alonely with the OUTER NODE(LEAF) where Search stopped i.e Case #1 & Case #2:
5194 POffsetInLEAF = BSTstack[--StackPtr];
5195 PseudoLinkedPointer_64 = BSTstack[--StackPtr];
5196 // NOTE: ONE LEAF IS FULL ONLY WHEN LAST CELL FOR KEY(here RW) EXISTS!
5197 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wrklen) != 0 )
5198 // if ( *(char *) (PseudoLinkedPointer+4+4+4+wrklen) != 0 ) // If LEAF is full: Case #2
5199 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5200 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4); //RW
5201 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5202 //fread(&SomeByte, 1, 1, fp_outRG);
5203 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5204 // [ //r.14+
5205 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
5206 if (BSTorBtree == 2) {
5207 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5208 fread(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5209 } else { // ##### 64bit memory manipulations [
5210 memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+2*(LongestLineInclusive+1+4) );
5211 } // ##### 64bit memory manipulations ]
5212 memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
5213 // ] //r.14+
5214 if (SomeByte != 0) // RW exists
5215 { SplitOccured = 1; WORDcountDistinct++;
5216 // ALlocate NEW LEAF:
5217 // if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrklen + 4 + 4 + 4 < GRMBLFoolAgain((int)wrklen) ) // +4 more for BST
5218 // instead of LL; + more(see LEAF)
5219 {
5220 memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
5221 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
5222 bufend[LetterOffset] = bufend[LetterOffset] + 2*wrklen;
5223 if (MAXusedBuffer[wrklen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrklen] = (unsigned
5224 long)(bufend[LetterOffset] - BufStart);}
5225 }
5226 if( 8 + 8 + 8 + 2*(LongestLineInclusive+1+4) < size_in64_L14 - (BufEnd_64-(unsigned long long)pointerflush_64) ) // the longest
5227 wrklen is LongestLineInclusive but actual is LongestLineInclusive(+ CR char)
5228 {
5229 PseudoLinkedPointerNEW_64 = BufEnd_64;
5230 BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
5231 BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
5232 }
5233 else
5234 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
5235 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, IIToADigits, 10), _ui64toaKAZEcomma((unsigned
5236 long long)WORDcountDistinct, IIToADigits2, 10) );
5237 fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, IIToADigits, 10) );
5238 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut,
5239 IIToADigits, 10) );
5240 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
5241 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n" );
5242 return( 1 );
5243 }

```

```

5239 if (POffsetInLEAF == 0) // wrd < LW
5240 {
5241 // memcopy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
5242 // memcopy( PseudoLinkedPointer+4+4+4, wrd, wrdlen ); // wrd go to OLD LEAF
5243 // memcopy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
5244 // *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
5245 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5246 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5247 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5248 //fread(&wrdUP[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5249 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5250 //fwrite(&wrd[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5251 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5252 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5253 //fread(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5254 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5255 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5256 //fwrite(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5257 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5258 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5259 //fwrite(&OneChar_iByte, 1, 1, fp_outRG); // Write ZERO ASCII code
5260 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5261 // [ //r.14+
5262 memcopy( &wrdUP[0], &LEAF[8 + 8 + 8], (LongestLinel ncl usi ve+1+4) );
5263 memcopy( &LEAF[8 + 8 + 8], &wrd[0], (LongestLinel ncl usi ve+1+4) );
5264 memcopy( &wrdAUX[0], &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], (LongestLinel ncl usi ve+1+4) );
5265 // Here reordering (of writing wrdAUX) is needed to avoid seek the position NEW and stupidly to seek again OLD/current position!
5266 memcopy( &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], &OneChar_iByte, 1 );
5267 if (BSTorBtree == 2) {
5268 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5269 fwrite(&LEAF[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5270 } else { // ##### 64bit memory manipulations [
5271 memcopy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4) );
5272 // ] ##### 64bit memory manipulations ]
5273 PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5274 if (BSTorBtree == 2) {
5275 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5276 fwrite(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5277 } else { // ##### 64bit memory manipulations [
5278 memcopy( (char *)PseudoLinkedPointerAUX_64, &wrdAUX[0], (LongestLinel ncl usi ve+1+4) );
5279 // ] ##### 64bit memory manipulations ]
5280 // ] //r.14+
5281 }
5282 if (POffsetInLEAF == 8) // LW < wrd < RW
5283 {
5284 // memcopy( wrdUP, wrd, wrdlen ); // wrd up
5285 // memcopy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
5286 // *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
5287 // memcopy( wrdUP, wrd, (LongestLinel ncl usi ve+1+4) ); // wrd up
5288 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5289 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5290 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5291 //fread(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5292 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5293 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5294 //fwrite(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5295 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5296 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5297 //fwrite(&OneChar_iByte, 1, 1, fp_outRG); // Write ZERO ASCII code
5298 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5299 // [ //r.14+
5300 memcopy( &wrdAUX[0], &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], (LongestLinel ncl usi ve+1+4) );
5301 // Here reordering (of writing wrdAUX) is needed to avoid seek the position NEW and stupidly to seek again OLD/current position!
5302 memcopy( &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], &OneChar_iByte, 1 );
5303 if (BSTorBtree == 2) {
5304 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5305 fwrite(&LEAF[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5306 } else { // ##### 64bit memory manipulations [
5307 memcopy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4) );
5308 // ] ##### 64bit memory manipulations ]
5309 PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5310 if (BSTorBtree == 2) {
5311 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5312 fwrite(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5313 } else { // ##### 64bit memory manipulations [
5314 memcopy( (char *)PseudoLinkedPointerAUX_64, &wrdAUX[0], (LongestLinel ncl usi ve+1+4) );
5315 // ] ##### 64bit memory manipulations ]
5316 // ] //r.14+
5317 }
5318 if (POffsetInLEAF == 16) // wrd > RW
5319 {
5320 // memcopy( wrdUP, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW up
5321 // *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
5322 // memcopy( PseudoLinkedPointerNEW+4+4+4, wrd, wrdlen ); // wrd go to NEW LEAF
5323 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5324 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5325 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5326 //fread(&wrdUP[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);

```

```

5327 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5328 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5329 //fwrite(&OneChar_iByte, 1, 1, fp_outRG); // Write ZERO ASCII code
5330 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5331 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5332 //fwrite(&wrd[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5333 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5334 // [ //r.14+
5335 memcpy( &wrdUP[0], &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], (LongestLineInclusive+1+4) );
5336 memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &OneChar_iByte, 1 );
5337 if (BSTorBtree == 2) {
5338 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5339 fwrite(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5340 } else { // ##### 64bit memory manipulations [
5341 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+2*(LongestLineInclusive+1+4) );
5342 } // ##### 64bit memory manipulations ]
5343 // ] //r.14+
5344 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one! Here NO need!
5345 PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5346 if (BSTorBtree == 2) {
5347 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5348 fwrite(&wrd[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5349 } else { // ##### 64bit memory manipulations [
5350 memcpy( (char *)PseudoLinkedPointerAUX_64, &wrd[0], (LongestLineInclusive+1+4) );
5351 } // ##### 64bit memory manipulations ]
5352 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one! Here NO need!
5353 }
5354 }
5355 else // If LEAF is not full: Case #1
5356 { SplitOccured = 0; WORDcountDistinct++;
5357 if (POffsetInLEAF == 0) // wrd < [LW][ ] so [LW][ ] -> [ ][LW] -> [wrd][LW]
5358 {
5359 // memcpy( PseudoLinkedPointer+4+4+4+wrdlen, PseudoLinkedPointer+4+4+4, wrdlen );
5360 // memcpy( PseudoLinkedPointer+4+4+4, wrd, wrdlen );
5361 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5362 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5363 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5364 //fread(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5365 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5366 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5367 //fwrite(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5368 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5369 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5370 //fwrite(&wrd[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5371 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5372 // [ //r.14+
5373 memcpy( &wrdAUX[0], &LEAF[8 + 8 + 8], (LongestLineInclusive+1+4) );
5374 memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &wrdAUX[0], (LongestLineInclusive+1+4) );
5375 memcpy( &LEAF[8 + 8 + 8], &wrd[0], (LongestLineInclusive+1+4) );
5376 if (BSTorBtree == 2) {
5377 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5378 fwrite(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5379 } else { // ##### 64bit memory manipulations [
5380 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+2*(LongestLineInclusive+1+4) );
5381 } // ##### 64bit memory manipulations ]
5382 // ] //r.14+
5383 }
5384 }
5385 if (POffsetInLEAF == 8) // wrd > [LW][ ] so [LW][ ] -> [LW][wrd]
5386 {
5387 // memcpy( PseudoLinkedPointer+4+4+4+wrdlen, wrd, wrdlen );
5388 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one! Here NO need!
5389 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5390 if (BSTorBtree == 2) {
5391 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5392 fwrite(&wrd[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5393 } else { // ##### 64bit memory manipulations [
5394 memcpy( (char *)PseudoLinkedPointerAUX_64, &wrd[0], (LongestLineInclusive+1+4) );
5395 } // ##### 64bit memory manipulations ]
5396 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one! Here NO need!
5397 }
5398 }
5399 }
5400 if (SplitOccured != 0)
5401 {
5402 // ~ Second deal with the INNER NODE(S) i.e Case #3 & Case #4:
5403 while (StackPtr != 0 || SplitOccured != 0)
5404 {
5405 // 'PseudoLinkedPointerNEW' is new LEAF to be inserted
5406 // 'wrdUP' is NEW word to be inserted
5407 if (StackPtr != 0)
5408 {
5409 POffsetInLEAF = BSTstack[--StackPtr];
5410 PseudoLinkedPointer_64 = BSTstack[--StackPtr];
5411 //if ( *(char *)PseudoLinkedPointer+4+4+4+wrdlen != 0 ) // If LEAF is full: Case #4
5412 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5413 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4); //RW
5414 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);

```



```

5415 //fread(&SomeByte, 1, 1, fp_outRG);
5416 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5417 // [ //r.14+
5418 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
5419 if (BSTorBtree == 2) {
5420 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5421 fread(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5422 } else { // ##### 64bit memory manipulations [
5423 memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+2*(LongestLineInclusive+1+4) );
5424 } // ##### 64bit memory manipulations ]
5425 memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
5426 // ] //r.14+
5427 if (SomeByte != 0) // RW exists
5428 { SplitOccured = 1;
5429 // memcpy( wrdUPold, wrdUP, wrdlen ); // LW up
5430 // PseudoLinkedPointerNEWold = PseudoLinkedPointerNEW;
5431 // memcpy( wrdUPold, wrdUP, (LongestLineInclusive+1+4) );
5432 // PseudoLinkedPointerNEWold_64 = PseudoLinkedPointerNEW_64;
5433 // Allocate NEW LEAF:
5434 // if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wordlen + 4 + 4 + 4 < GRMBLFoolAgain((int)wordlen) ) // +4 more for BST
instead of LL; + more(see LEAF)
5435 // {
5436 // memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
5437 // bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
5438 // bufend[LetterOffset] = bufend[LetterOffset] + 2*wordlen;
5439 // if (MAXusedBuffer[wordlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wordlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
5440 // }
5441 if( 8 + 8 + 8 + 2*(LongestLineInclusive+1+4) < size_in64_L14 - (BufEnd_64-(unsigned long long)pointerflush_64) ) // the longest
wordlen is LongestLineInclusive but actual is LongestLineInclusive(+ CR char)
5442 {
5443 PseudoLinkedPointerNEW_64 = BufEnd_64;
5444 BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
5445 BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
5446 // [ //r.14+
5447 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64;
5448 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5449 //fread(&LEAFNEW[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5450 // In fact above three lines are slow, the only need is ZEROed LEAFNEW.
5451 memset(&LEAFNEW[0], 0, 8+8+2*(LongestLineInclusive+1+4));
5452 // ] //r.14+
5453 }
5454 else
5455 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
5456 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, IIT0aDigits, 10), _ui64toaKAZEcomma((unsigned
long long)WORDcountDistinct, IIT0aDigits, 10) );
5457 fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, IIT0aDigits, 10) );
5458 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut,
IIT0aDigits, 10) );
5459 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
5460 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
5461 return( 1 );
5462 }
5463 if (POffsetInLEAF == 0) // wrdUPold < LW
5464 {
5465 // memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
5466 // memcpy( PseudoLinkedPointer+4+4+4, wrdUPold, wrdlen ); // wrdUPold go to OLD LEAF
5467 // memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wordlen, wrdlen ); // RW go to NEW LEAF
5468 // *(char *) (PseudoLinkedPointer+4+4+4+wordlen) = 0; // RW mark unused in OLD LEAF
5469 // [LP] (PseudoLinkedPointerNEWold)[MP][RP](wrdUPold)[LW][RW] -----
5470 // // pair [LW] PseudoLinkedPointerNEW goes up
5471 // // PseudoLinkedPointer: PseudoLinkedPointerNEW: |
5472 // // [LP] (PseudoLinkedPointerNEWold)[ ] (wrdUPold) [MP][RP][ ] [RW] <----
5473 // // no need to put zero in RP because logic is based on words existence:
5474 // memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+4, 4 );
5475 // memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
5476 // memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEWold, 4 );
5477 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5478 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5479 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5480 //fread(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5481 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5482 //fwrite(&wrdUPold[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5483 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5484 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5485 //fread(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5486 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5487 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5488 //fwrite(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5489 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5490 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5491 //fwrite(&OneChar_ieByte, 1, 1, fp_outRG); // Write ZERO ASCII code
5492 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
5493 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5494 //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5495 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 0;
5496 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5497 //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);

```

```

5498 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5499 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5500 //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5501 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8;
5502 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5503 //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5504 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
5505 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5506 //fwrite(&PseudoLinkedPointerNEWold_64, 8, 1, fp_outRG);
5507 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5508 // [ //r.14+
5509 memcpy( &wrdUP[0], &LEAF[8 + 8 + 8], (LongestLinel ncl usi ve+1+4) );
5510 memcpy( &LEAF[8 + 8 + 8], &wrdUPold[0], (LongestLinel ncl usi ve+1+4) );
5511 memcpy( &wrdAUX[0], &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], (LongestLinel ncl usi ve+1+4) );
5512 memcpy( &LEAFNEW[8 + 8 + 8], &wrdAUX[0], (LongestLinel ncl usi ve+1+4) );
5513 memcpy( &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], &OneChar_ieByte, 1 );
5514 memcpy( &PseudoLinkedPointerAUXdumbo_64, &LEAF[8], 8 );
5515 memcpy( &LEAFNEW[0], &PseudoLinkedPointerAUXdumbo_64, 8 );
5516 memcpy( &PseudoLinkedPointerAUXdumbo_64, &LEAF[16], 8 );
5517 memcpy( &LEAFNEW[8], &PseudoLinkedPointerAUXdumbo_64, 8 );
5518 memcpy( &LEAF[8], &PseudoLinkedPointerNEWold_64, 8 );
5519 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
5520 if (BSTorBtree == 2) {
5521 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5522 fwrite(&LEAF[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5523 } else { // ##### 64bit memory manipulations [
5524 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4) );
5525 } // ##### 64bit memory manipulations ]
5526 PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64;
5527 if (BSTorBtree == 2) {
5528 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5529 fwrite(&LEAFNEW[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5530 } else { // ##### 64bit memory manipulations [
5531 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAFNEW[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4) );
5532 } // ##### 64bit memory manipulations ]
5533 // ] //r.14+
5534 }
5535 if (PoffsetInLEAF == 8) // LW < wrdUPold < RW
5536 {
5537 // memcpy( wrdUP, wrdUPold, wrdlen ); // wrdUPold up
5538 // memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+ wrdlen, wrdlen ); // RW go to NEW LEAF
5539 // *(char *) (PseudoLinkedPointer+4+4+4+ wrdlen) = 0; // RW mark unused in OLD LEAF
5540 // // [LP][MP] (PseudoLinkedPointerNEWold)[RP][LW] (wrdUPold)[RW] -----
5541 // // pair [wrdUPold] PseudoLinkedPointerNEW goes up |
5542 // // PseudoLinkedPointer: PseudoLinkedPointerNEW:
5543 // // [LP][MP][LW] (PseudoLinkedPointerNEWold)[RP][RW] <----
5544 // // no need to put zero in RP because logic is based on words existence:
5545 // memcpy( PseudoLinkedPointerNEW+0, &PseudoLinkedPointerNEWold, 4 );
5546 // memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
5547 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5548 // memcpy( wrdUP, wrdUPold, (LongestLinel ncl usi ve+1+4) );
5549 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5550 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5551 //fread(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5552 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5553 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5554 //fwrite(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5555 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5556 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5557 //fwrite(&OneChar_ieByte, 1, 1, fp_outRG); // Write ZERO ASCII code
5558 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 0;
5559 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5560 //fwrite(&PseudoLinkedPointerNEWold_64, 8, 1, fp_outRG);
5561 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5562 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5563 //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5564 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8;
5565 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5566 //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5567 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5568 // [ //r.14+
5569 memcpy( &wrdAUX[0], &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], (LongestLinel ncl usi ve+1+4) );
5570 memcpy( &LEAFNEW[8 + 8 + 8], &wrdAUX[0], (LongestLinel ncl usi ve+1+4) );
5571 memcpy( &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], &OneChar_ieByte, 1 );
5572 memcpy( &LEAFNEW[0], &PseudoLinkedPointerNEWold_64, 8 );
5573 memcpy( &PseudoLinkedPointerAUXdumbo_64, &LEAF[16], 8 );
5574 memcpy( &LEAFNEW[8], &PseudoLinkedPointerAUXdumbo_64, 8 );
5575 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
5576 if (BSTorBtree == 2) {
5577 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5578 fwrite(&LEAF[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5579 } else { // ##### 64bit memory manipulations [
5580 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4) );
5581 } // ##### 64bit memory manipulations ]
5582 PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64;
5583 if (BSTorBtree == 2) {
5584 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5585 fwrite(&LEAFNEW[0], 8+8+8+2*(LongestLinel ncl usi ve+1+4), 1, fp_outRG);

```

```

5586         } else { // ##### 64bit memory manipulations [
5587     memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAFNEW[0], 8+8+2*(LongestLinel ncl usi ve+1+4) );
5588         } // ##### 64bit memory manipulations ]
5589     // ] //r.14+
5590 }
5591 if (POffsetInLEAF == 16) // wrdUPol d > RW
5592 {
5593     memcpy( wrdUP, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW up
5594     *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
5595     memcpy( PseudoLinkedPointerNEW+4+4+4, wrdUPol d, wrdlen ); // wrdUPol d go to NEW LEAF
5596     // [LP][MP][RP] (PseudoLinkedPointerNEWol d)[LW][RW] (wrdUPol d) -----
5597     // pair [RW] PseudoLinkedPointerNEW goes up
5598     // PseudoLinkedPointer: PseudoLinkedPointerNEW
5599     // [LP][MP][LW] [RP] (PseudoLinkedPointerNEWol d)[LW] (wrdUPol d) <---
5600     // no need to put zero in RP because logic is based on words existence:
5601     memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+8, 4 );
5602     memcpy( PseudoLinkedPointerNEW+4, &PseudoLinkedPointerNEWol d, 4 );
5603     // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5604     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5605     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5606     //fread(&wrdUP[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5607     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5608     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5609     //fwrite(&OneChar_ieByte, 1, 1, fp_outRG); // Write ZERO ASCII code
5610     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5611     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5612     //fwrite(&wrdUPol d[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5613     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5614     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5615     //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5616     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 0;
5617     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5618     //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5619     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8;
5620     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5621     //fwrite(&PseudoLinkedPointerNEWol d_64, 8, 1, fp_outRG);
5622     // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5623     // [ //r.14+
5624     memcpy( &wrdUP[0], &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], (LongestLinel ncl usi ve+1+4) );
5625     memcpy( &LEAF[8 + 8 + 8 + (LongestLinel ncl usi ve+1+4)], &OneChar_ieByte, 1 );
5626     memcpy( &LEAFNEW[8 + 8 + 8], &wrdUPol d[0], (LongestLinel ncl usi ve+1+4) );
5627     memcpy( &PseudoLinkedPointerAUXdumbo_64, &LEAF[16], 8 );
5628     memcpy( &LEAFNEW[0], &PseudoLinkedPointerAUXdumbo_64, 8 );
5629     memcpy( &LEAFNEW[8], &PseudoLinkedPointerNEWol d_64, 8 );
5630     PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
5631     if (BSTorBtree == 2) {
5632         fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5633         fwrite(&LEAF[0], 8+8+2*(LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5634     } else { // ##### 64bit memory manipulations [
5635     memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+2*(LongestLinel ncl usi ve+1+4) );
5636     } // ##### 64bit memory manipulations ]
5637     PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64;
5638     if (BSTorBtree == 2) {
5639         fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5640         fwrite(&LEAFNEW[0], 8+8+2*(LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5641     } else { // ##### 64bit memory manipulations [
5642     memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAFNEW[0], 8+8+2*(LongestLinel ncl usi ve+1+4) );
5643     } // ##### 64bit memory manipulations ]
5644     // ] //r.14+
5645 }
5646 }
5647 else // If LEAF is not full: Case #3
5648 { SplitOccured = 0;
5649     if (POffsetInLEAF == 0) // wrdUP < [LW] so [LW] -> [][LW] -> [wrdUP][LW]
5650     {
5651         //
5652         memcpy( PseudoLinkedPointer+4+4+4+wrdlen, PseudoLinkedPointer+4+4+4, wrdlen );
5653         //
5654         memcpy( PseudoLinkedPointer+4+4+4, wrdUP, wrdlen );
5655         //
5656         // [LP][MP][LW] -> [LP][MP] -> [LP][np][MP]
5657         //
5658         memcpy( PseudoLinkedPointer+8, PseudoLinkedPointer+4, 4 );
5659         //
5660         memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEW, 4 );
5661         // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5662         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5663         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5664         //fread(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5665         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLinel ncl usi ve+1+4);
5666         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5667         //fwrite(&wrdAUX[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5668         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5669         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5670         //fwrite(&wrdUP[0], (LongestLinel ncl usi ve+1+4), 1, fp_outRG);
5671         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
5672         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5673         //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5674         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5675         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5676         //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5677         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
5678         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5679     }
5680 }

```

```

5674 //fwrite(&PseudoLinkedPointerNEW_64, 8, 1, fp_outRG);
5675 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5676 // [ //r.14+
5677 memcpy( &wrdAUX[0], &LEAF[8 + 8 + 8], (LongestLineInclusive+1+4) );
5678 memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &wrdAUX[0], (LongestLineInclusive+1+4) );
5679 memcpy( &LEAF[8 + 8 + 8], &wrdUP[0], (LongestLineInclusive+1+4) );
5680 memcpy( &PseudoLinkedPointerAUXdumbo_64, &LEAF[8], 8 );
5681 memcpy( &LEAF[8 + 8], &PseudoLinkedPointerAUXdumbo_64, 8 );
5682 memcpy( &LEAF[8], &PseudoLinkedPointerNEW_64, 8 );
5683 if (BSTorBtree == 2) {
5684 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5685 fwrite(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5686 } else { // ##### 64bit memory manipulations [
5687 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+2*(LongestLineInclusive+1+4) );
5688 } // ##### 64bit memory manipulations ]
5689 // ] //r.14+
5690 }
5691 if (POffsetInLEAF == 8) // wrdUP > [LW][ ] so [LW][ ] -> [LW][wrdUP]
5692 {
5693 // memcpy( PseudoLinkedPointer+4+4+wrdlen, wrdUP, wrdlen );
5694 // // [LP][MP][ ] -> [LP][MP][np]
5695 // memcpy( PseudoLinkedPointer+8, &PseudoLinkedPointerNEW, 4 );
5696 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5697 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5698 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5699 //fwrite(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5700 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5701 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5702 //fwrite(&PseudoLinkedPointerNEW_64, 8, 1, fp_outRG);
5703 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5704 // [ //r.14+
5705 memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &wrdUP[0], (LongestLineInclusive+1+4) );
5706 memcpy( &LEAF[8 + 8], &PseudoLinkedPointerNEW_64, 8 );
5707 if (BSTorBtree == 2) {
5708 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5709 fwrite(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5710 } else { // ##### 64bit memory manipulations [
5711 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+2*(LongestLineInclusive+1+4) );
5712 } // ##### 64bit memory manipulations ]
5713 // ] //r.14+
5714 }
5715 break;
5716 }
5717 }
5718 else // Empty stack means ROOT and more over ROOT is already splitted(Case #4 is off)
5719 {
5720 // If LEAF is not full: Case #3
5721 // THIS IS WHERE A NEW(SECOND) LEAF 'PseudoLinkedPointerROOT' must be allocated:
5722 // if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain((int)wrdlen) ) // +4 more for BST
instead of LL; + more(see LEAF)
5723 // {
5724 // memcpy( &PseudoLinkedPointerROOT, &bufend[LetterOffset], 4 );
5725 // bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
5726 // memcpy( bufend[LetterOffset], wrdUP, wrdlen );
5727 // bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
5728 // if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
5729 // }
5730 if( 8 + 8 + 8 + 2*(LongestLineInclusive+1+4) < size_in64_L14 - (BufEnd_64-(unsigned long long)pointerflush_64) ) // the longest
wrdlen is LongestLineInclusive but actual is LongestLineInclusive(+ CR char)
5731 {
5732 PseudoLinkedPointerROOT_64 = BufEnd_64;
5733 BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
5734 BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
5735 PseudoLinkedPointerAUX_64 = PseudoLinkedPointerROOT_64 + 8 + 8 + 8;
5736 if (BSTorBtree == 2) {
5737 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5738 fwrite(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5739 } else { // ##### 64bit memory manipulations [
5740 memcpy( (char *)PseudoLinkedPointerAUX_64, &wrdUP[0], (LongestLineInclusive+1+4) );
5741 } // ##### 64bit memory manipulations ]
5742 }
5743 else
5744 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
5745 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORdcount, IIT0aDigits, 10), _ui64toaKAZEcomma((unsigned
long long)WORdcountDistinct, IIT0aDigits2, 10) );
5746 fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, IIT0aDigits, 10) );
5747 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDS into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORdcountAttemptsToPut,
IIT0aDigits, 10) );
5748 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
5749 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
5750 return( 1 );
5751 }
5752 // Here -- 'PseudoLinkedPointerROOT' --
5753 // | (wrdUP) |
5754 // 'PseudoLinkedPointer' <-- --> 'PseudoLinkedPointerNEW'
5755 // (LW) (RW)
5756 // memcpy( PseudoLinkedPointerROOT, &PseudoLinkedPointer, 4 ); // LP

```

```

5757 //      memcpy( PseudoLinkedPointerROOT+4, &PseudoLinkedPointerNEW, 4 ); // MP
5758 //      // Here must NEW ROOT be updated i.e. HASH table(SLOT) must point it:
5759 //      memcpy( BufStart+Slot, &PseudoLinkedPointerROOT, 4 );
5760 PseudoLinkedPointerAUX_64 = PseudoLinkedPointerROOT_64;
5761      if (BSTorBtree == 2) {
5762 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5763 fwrite(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
5764      } else { // ##### 64bit memory manipulations [
5765      memcpy( (char *)PseudoLinkedPointerAUX_64, &PseudoLinkedPointer_64, 8 );
5766      } // ##### 64bit memory manipulations ]
5767 PseudoLinkedPointerAUX_64 = PseudoLinkedPointerROOT_64 + 8;
5768      if (BSTorBtree == 2) {
5769 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5770 fwrite(&PseudoLinkedPointerNEW_64, 8, 1, fp_outRG);
5771      } else { // ##### 64bit memory manipulations [
5772      memcpy( (char *)PseudoLinkedPointerAUX_64, &PseudoLinkedPointerNEW_64, 8 );
5773      } // ##### 64bit memory manipulations ]
5774 memcpy( BufStart+Slot, &PseudoLinkedPointerROOT_64, 8 );
5775 break; //because it is ROOT without split
5776     }
5777     } // while
5778 } //if (SplitOccured != 0)
5779 // 3] Insert Iterative ]
5780 //if (FoundInLinkedList == 0)
5781 } //if (DoNotInsertFlag == 0) { // This line comes since r15FIXFIX+
5782 //if ( REUSE != 2 ) { // REUSE [ // This line comes since r16
5783 // ##### B-tree order 3 64bit ]
5784 } //if ( (Slot>>HashChunkSizeInBITS) == RipPasses ) {
5785
5786 // External Btrees ]
5787 } //if (BSTorBtree != 2) {
5788     } // if ( ( PLE_words == 4 ) && ( wrdlen <= 31 ) ) {
5789     } // if( 1 <= wrdlen && wrdlen <= 31 )
5790 /*
5791 The Most Stupid Bug I have ever made [
5792     else {
5793 PLE_words_INITflag = 1; // This line fixes the stupidity done since first quadruplet on r1, namely to initialize the sequence
when a word longer than 31 is spotted - it is wrong to slide it.
5794     }
5795 The Most Stupid Bug I have ever made ]
5796 */
5797 //r.15fix [
5798 //This line was intended BUT placed before the fragment 'if( 1 <= wrdlen && wrdlen <= 31 )' [
5799 //      if( wrdlen > 31 ) PLE_words_INITflag = 1; // Not when wrdlen == 0 as the above buggy fragment!
5800 //This line was intended BUT placed before the fragment 'if( 1 <= wrdlen && wrdlen <= 31 )' ]
5801 //r.15fix ]
5802 if ( PLE_words_INITflag == 1 ) { PLE_words = 0; PLE_words_INITflag = 0; } // Quadruple!
5803 wrdlen = 0;
5804     // This fragment is MIRROred: #1 copy ]
5805     }
5806     //else if( workbyte >= 'A' && workbyte <= 'Z' )
5807     else if( workbyte <= 'Z' )
5808     {
5809         if( wrdlen < 31 )
5810         //if( wrdlen < LongestLineInclusive )
5811         { wrd[ wrdlen ] = workbyte + 32 ; }
5812         wrdlen++;
5813     }
5814     else if( workbyte >= 'a' && workbyte <= 'z' )
5815     {
5816         if( wrdlen < 31 )
5817         //if( wrdlen < LongestLineInclusive )
5818         { wrd[ wrdlen ] = workbyte; }
5819         wrdlen++;
5820     }
5821     else
5822     {
5823         // This fragment is MIRROred: #2 copy [
5824         goto ElStupido;
5825         // This fragment is MIRROred: #2 copy ]
5826     }
5827     } // i 'for'
5828     //-----
5829 //++Melnitcka;
5830 //Melnitcka = Melnitcka % 4;
5831 //if (Melnitcka == 0){ printf( "|; Word count: %s of them %s distinct; Done: %lu/64\n", _ui64toaKAZEcomma(WORDcount, IIT0aDigi ts, 10),
    _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, IIT0aDigi ts2, 10), 64 ); }
5832 //if (Melnitcka == 1){ printf( "/; Word count: %s of them %s distinct; Done: %lu/64\n", _ui64toaKAZEcomma(WORDcount, IIT0aDigi ts, 10),
    _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, IIT0aDigi ts2, 10), 64 ); }
5833 //if (Melnitcka == 2){ printf( "-; Word count: %s of them %s distinct; Done: %lu/64\n", _ui64toaKAZEcomma(WORDcount, IIT0aDigi ts, 10),
    _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, IIT0aDigi ts2, 10), 64 ); }
5834 //if (Melnitcka == 3){ printf( "\\; Word count: %s of them %s distinct; Done: %lu/64\n", _ui64toaKAZEcomma(WORDcount, IIT0aDigi ts, 10),
    _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, IIT0aDigi ts2, 10), 64 ); }
5835 Melnitcka = Melnitcka & 3; // 0 1 2 3: 00 01 10 11
5836     (void) time(&t4);
5837     if (t4 <= t1) {t4 = t1; t4++;}
5838 printf( "%s; %sP/s; Phrase count: %s of them %s distinct; Done: %lu/64\n", Auberge[Melnitcka++], _ui64toaKAZEzerocomma(WORDcount/((int) t4-
t1), IIT0aDigi ts3, 10)+(26-10), _ui64toaKAZEcomma(WORDcount, IIT0aDigi ts, 10), _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct,

```

```

5839 IIT0aDigits2, 10), 64 );
5840 fclose( fp_inLINE );
5841 } // ----- IT IS a FILENAME not a METACOMMAND ]
5842     LINE10len = 0;
5843 LINE10[ LINE10len ] = 0;
5844     }
5845     }
5846 } // k 'for'
5847
5848 (void) time(&t3);
5849 if (t3 <= t1) {t3 = t1; t3++;}
5850 printf( "Bytes per second performance: %sB/s\n", _ui64toaKAZEcomma(FilesLEN/((int) t3-t1), IIT0aDigits, 10) ); // Rev. 12+
5851 printf( "Phrases per second performance: %sP/s\n", _ui64toaKAZEcomma(WORDcount/((int) t3-t1), IIT0aDigits, 10) ); // Rev. 12+
5852 printf("Time for putting phrases into trees: %d second(s)\n", (int) t3-t1);
5853
5854 if (BSTorBtree < 2) {
5855     // FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH
5856     printf("Flushing unsorted words...\n");
5857     if ( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
5858     { printf( "Leprechaun: Can't create file %s\n", argv[2] ); return( 1 ); }
5859     ZEROS[0] = 0; ZEROS[1] = 0; ZEROS[2] = 0; ZEROS[3] = 0;
5860     CRdLFa[0] = 13; CRdLFa[1] = 10;
5861
5862     for( i = 0; i < 806; i++ )
5863     { //BufStart = pointerflush + i * LetterBuffer; // OLD
5864         BufStart = pointerflush + (i / 31) * WHOLELetter_BufferSize + OffsetsInBuffer[i % 31];
5865         // for( j = 0; j < NumberOfSLOTS; j++ )
5866         // {
5867         //     Slot = j<<2;
5868         //     memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
5869         //     while (PseudoLinkedPointer != 0)
5870         //     { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
5871         //       memcpy( PseudoLinkedPointer, ZEROS, 4 );
5872         //       PseudoLinkedPointer = PseudoLinkedPointerNEW;
5873         //     }
5874         // }
5875         // // Start of COUPLES [OFFSET: 4byte(ZEROS)][WORD: up to 31bytes]
5876         // //fwrite(BufStart+(NumberOfSLOTS+1)*4, bufend[i] - (BufStart+(NumberOfSLOTS+1)*4), 1, fp_out );
5877         // /* Follows STATE OF UGLINESS: */
5878         // Flushing = BufStart+(NumberOfSLOTS+1)*4 + 4; // '+ 4' in order to skip first 4 zeros
5879         // //in case of current buffer not have been used then NOT entering in this cycle
5880         // while(Flushing < bufend[i])
5881         // { if (*Flushing != 0) {fwrite(Flushing, 1, 1, fp_out ); TotalWLchars++;}
5882         //   // Below 'Flushing-1' works due to skipped first 4 zeros!
5883         //   if (*Flushing-1) != 0 && *Flushing == 0) {fwrite(CRdLFa, 2, 1, fp_out);}
5884         //   //last word must be suffixed with 1310 too
5885         //   if (Flushing == bufend[i]-1) {fwrite(CRdLFa, 2, 1, fp_out );}
5886         //   Flushing++;
5887         // }
5888
5889         for( j = 0; j < NumberOfSLOTS; j++ )
5890         {
5891             Slot = j<<2;
5892             memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
5893             if (PseudoLinkedPointer != 0)
5894             {
5895                 NumberOfTrees++;
5896                 if (BSTorBtree == 0)
5897                 {
5898                     // ===== BST traverse [
5899                     // DONE JOB:
5900                     // Must be written BST traverse ! with simulated stack i.e. non-recursive.
5901                     // ...
5902                     // /*
5903                     // Given a binary search tree, print out
5904                     // its data elements in increasing
5905                     // sorted order.
5906                     // */
5907                     // void printTree(struct node* node) {
5908                     // if (node == NULL) return;
5909                     // printTree(node->left);
5910                     // printf("%d ", node->data);
5911                     // printTree(node->right);
5912                     // }
5913
5914                     // FUTURE JOB:
5915                     // I need functions:
5916                     // BST_LeafNumber() // greater the better
5917                     // BST_NodeNumber() // 'BSTcurrent' below
5918                     // BST_Peak() // i.e. levels, root has height = 1
5919                     // BST_PeakIB() // 1BBST(1deal Balanced BST) has 1 + lgNodeNumber height
5920                     // I need 'Ideal Balancing BST FRAGMENT' with simulated stack:
5921                     // I need 'Ideal Balancing BST FRAGMENT' to be executed when Peak() >= PeakIB()<<1:
5922
5923                     // ----- [
5924                     BSTcurrentNode = 0; BSTcurrentPeak = 0; BSTcurrentLeaf = 0;
5925                     BSTcurrentPeakMAX = 0; // Height of current BST

```

```

5926 StackPtr = 0;
5927 while ( 2==2 ) {
5928     while (PseudoLinkedPointer != 0)
5929     {
5930         if (StackPtr > 8192*3-1-3) { printf( "\nLeprechaun: Failure! BST simulated stack overflow, too high BST!\n" ); return( 13 );}
5931         memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
5932         PseudoLinkedPointer = PseudoLinkedPointer + 4;
5933         memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer, 4 );
5934         BSTstack[StackPtr] = PseudoLinkedPointer + 4; ++StackPtr; //ptr to wrd
5935         BSTstack[StackPtr] = PseudoLinkedPointerNEWright; ++StackPtr;
5936         BSTstack[StackPtr] = PseudoLinkedPointerNEWleft; ++StackPtr; //needed for stats not for recursion
5937         // BST stats [
5938             if (PseudoLinkedPointerNEWleft == 0 && PseudoLinkedPointerNEWright == 0) {BSTcurrentLeaf++; BSTsTotalLEAFs++;}
5939             BSTcurrentPeak++;
5940             if (BSTcurrentPeakMAX < BSTcurrentPeak) BSTcurrentPeakMAX = BSTcurrentPeak;
5941             BSTstack[StackPtr] = BSTcurrentPeak; ++StackPtr; //needed for stats not for recursion
5942         // BST stats ]
5943         PseudoLinkedPointer = PseudoLinkedPointerNEWright; // choose right instead of 'PseudoLinkedPointerNEWleft' because of stats
5944     }
5945     if (StackPtr == 0) break;
5946     BSTcurrentPeak = BSTstack[--StackPtr]; // level of the node(1 is root) needed only for stats(print)
5947     PseudoLinkedPointerNEWleft = BSTstack[--StackPtr]; // left pointer needed only for stats(print)
5948     PseudoLinkedPointerNEWright = BSTstack[--StackPtr]; // right pointer
5949     memcpy( wrd, BSTstack[--StackPtr], i%31+1 );
5950     fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
5951     fwrite(CRdLfA, 2, 1, fp_out);
5952     BSTcurrentNode++;
5953     PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
5954 }
5955 // ----- ]
5956 // BST stats [
5957 if (BSTwidthMAXnode < BSTcurrentNode) {
5958     BSTwidthMAXnode = BSTcurrentNode;
5959     BSTwidthMAXnodePEAK = BSTcurrentPeakMAX;
5960     BSTwidthMAXnodeLEAF = BSTcurrentLeaf;
5961     BSTcurrentNodeMAXQUANTITY = 0;
5962 }
5963 if (BSTwidthMAXnode == BSTcurrentNode) BSTcurrentNodeMAXQUANTITY++;
5964 if (BSTwidthMAXpeak < BSTcurrentPeakMAX) {
5965     BSTwidthMAXpeak = BSTcurrentPeakMAX;
5966     BSTwidthMAXpeakNODE = BSTcurrentNode;
5967     BSTwidthMAXpeakLEAF = BSTcurrentLeaf;
5968     BSTcurrentPeakMAXQUANTITY = 0; i BSTwidthMAXpeak=i; j BSTwidthMAXpeak=j;
5969 }
5970 if (BSTwidthMAXpeak == BSTcurrentPeakMAX) BSTcurrentPeakMAXQUANTITY++;
5971 if (BSTwidthMAXleaf < BSTcurrentLeaf) {
5972     BSTwidthMAXleaf = BSTcurrentLeaf;
5973     BSTwidthMAXleafNODE = BSTcurrentNode;
5974     BSTwidthMAXleafPEAK = BSTcurrentPeakMAX;
5975     BSTcurrentLeafMAXQUANTITY = 0;
5976 }
5977 if (BSTwidthMAXleaf == BSTcurrentLeaf) BSTcurrentLeafMAXQUANTITY++;
5978 // BST stats ]
5979 // ===== BST traverse ]
5980 } else
5981 {
5982 // $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse [
5983 // DONE JOB:
5984 // Must be written B-tree traverse ! with simulated stack i.e. non-recursive.
5985 // ...
5986 StackPtr = 0;
5987 while ( 2==2 ) {
5988     while (PseudoLinkedPointer != 0)
5989     {
5990         if (StackPtr > 8192*3-1) { printf( "\nLeprechaun: Failure! B-tree simulated stack overflow, too high B-tree!\n" ); return( 13 );}
5991         BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //ptr to Rword
5992         if ( *(char *) (PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) == 0 ) {memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTS*4], 4 );}
5993         memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
5994         memcpy( &PseudoLinkedPointerNEWmiddle, PseudoLinkedPointer + 4, 4 );
5995         memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
5996         // Give first from right to left non-zero PTR
5997         if (PseudoLinkedPointerNEWright != 0 )
5998         { memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTS*4], 4 );
5999           PseudoLinkedPointer = PseudoLinkedPointerNEWright;
6000         }
6001         else if (PseudoLinkedPointerNEWmiddle != 0 )
6002         { memcpy( PseudoLinkedPointer + 4, &BufStart[NumberOfSLOTS*4], 4 );
6003           PseudoLinkedPointer = PseudoLinkedPointerNEWmiddle;
6004         }
6005         else if (PseudoLinkedPointerNEWleft != 0 )
6006         { memcpy( PseudoLinkedPointer, &BufStart[NumberOfSLOTS*4], 4 );
6007           PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
6008         }
6009         else
6010         {
6011             PseudoLinkedPointer = 0;

```

```

6012     }
6013 }
6014 if (StackPtr == 0) break;
6015 PseudoLinkedPointer = BSTstack[--StackPtr];
6016 memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
6017 memcpy( &PseudoLinkedPointerNEWmiddle, PseudoLinkedPointer + 4, 4 );
6018 memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
6019 if (PseudoLinkedPointerNEWleft+PseudoLinkedPointerNEWmiddle+PseudoLinkedPointerNEWright == 0) // One LEAF is PRINTED when LP=0 MP=0
RP=0
6020 {
6021     memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4, i%31+1 );
6022     fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
6023     fwrite(CRDLFa, 2, 1, fp_out);
6024     if ( *(char *) (PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) != 0 )
6025     { memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4 + i%31+1, i%31+1 );
6026       fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
6027       fwrite(CRDLFa, 2, 1, fp_out);
6028     }
6029     PseudoLinkedPointer = 0;
6030 }
6031 }
6032 // $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse ]
6033 }
6034     }
6035 } // j
6036
6037 } // i
6038
6039 if (BSTorBtree == 0)
6040 {
6041     // ----- Longest path ----- [
6042     i=iBSTwidthMAXpeak; j=jBSTwidthMAXpeak;
6043     BufStart = pointerflush + (i / 31) * WHOLEletter_BufferSize + OffsetsInBuffer[i % 31];
6044     Slot = j<<2;
6045     memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
6046     if (PseudoLinkedPointer != 0)
6047     {
6048         // ----- [
6049         BSTcurrentNode = 0; BSTcurrentPeak = 0; BSTcurrentLeaf = 0;
6050         BSTcurrentPeakMAX = 0; // Height of current BST
6051         StackPtr = 0;
6052         // BST print [
6053         fprintf( fp_outLOG, "A(not always THE) Binary-Search-Tree with the longest path(height, PEAK, number of levels):\n" );
6054         // BST print ]
6055         while ( 2==2 ) {
6056             while (PseudoLinkedPointer != 0)
6057             {
6058                 if (StackPtr > 8192*3-1-3) { printf( "\nLeprechaun: Failure! BST simulated stack overflow, too high BST!\n" ); return( 13 );}
6059                 memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
6060                 PseudoLinkedPointer = PseudoLinkedPointer + 4;
6061                 memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer, 4 );
6062                 BSTstack[StackPtr] = PseudoLinkedPointer + 4; ++StackPtr; //ptr to wrd
6063                 BSTstack[StackPtr] = PseudoLinkedPointerNEWright; ++StackPtr;
6064                 BSTstack[StackPtr] = PseudoLinkedPointerNEWleft; ++StackPtr; //needed for stats not for recursion
6065                 // BST stats [
6066                 if (PseudoLinkedPointerNEWleft == 0 && PseudoLinkedPointerNEWright == 0) {BSTcurrentLeaf++;} //BSTsTotalLEAFs++;} // REMOVED
to avoid mess in TOTAL stats
6067                 BSTcurrentPeak++;
6068                 if (BSTcurrentPeakMAX < BSTcurrentPeak) BSTcurrentPeakMAX = BSTcurrentPeak;
6069                 BSTstack[StackPtr] = BSTcurrentPeak; ++StackPtr; //needed for stats not for recursion
6070                 // BST stats ]
6071                 PseudoLinkedPointer = PseudoLinkedPointerNEWright; // choose right instead of 'PseudoLinkedPointerNEWleft' because of stats
print
6072             }
6073             if (StackPtr == 0) break;
6074             BSTcurrentPeak = BSTstack[--StackPtr]; // level of the node(1 is root) needed only for stats(print)
6075             PseudoLinkedPointerNEWleft = BSTstack[--StackPtr]; // left pointer needed only for stats(print)
6076             PseudoLinkedPointerNEWright = BSTstack[--StackPtr]; // right pointer
6077             memcpy( wrd, BSTstack[--StackPtr], i%31+1 );
6078             //fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
6079             //fwrite(CRDLFa, 2, 1, fp_out);
6080             BSTcurrentNode++;
6081             PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
6082             // BST print [
6083             for( k = 0; k < BSTcurrentPeak; k++ ) fprintf( fp_outLOG, "%c", ' ' );
6084             if (PseudoLinkedPointerNEWleft == 0) fprintf( fp_outLOG, "[" ); else fprintf( fp_outLOG, "]" );
6085             for( k = 0; k < i%31+1; k++ ) fprintf( fp_outLOG, "%c", *(char *) (wrd+k) );
6086             if (PseudoLinkedPointerNEWright == 0) fprintf( fp_outLOG, "]" ); else fprintf( fp_outLOG, "[" );
6087             if (BSTcurrentPeak == 1) fprintf( fp_outLOG, " ROOT" );
6088             fprintf( fp_outLOG, "\n" );
6089             // BST print ]
6090         }
6091         // ----- ]
6092     }
6093     fprintf( fp_outLOG, "Above Binary-Search-Tree with MaxPEAK = %s has NODEs = %s and LEAFs = %s\n", _ui64toaKAZEcomma(BSTwidthMAXpeak,
IIToADigits2, 10), _ui64toaKAZEcomma(BSTwidthMAXpeakNODE, IIToADigits3, 10), _ui64toaKAZEcomma(BSTwidthMAXpeakLEAF, IIToADigits, 10));
6094     fprintf( fp_outLOG, "Legend: \n" );
6095     fprintf( fp_outLOG, "At left side of the word - '[' means no left successor\n" );

```



```

6096         fprintf( fp_outLOG, "At left side of the word - '[' means left successor exists\n" );
6097         fprintf( fp_outLOG, "At right side of the word - ']' means no right successor\n" );
6098         fprintf( fp_outLOG, "At right side of the word - '[' means right successor exists\n" );
6099 // ----- Longest path ----- ]
6100
6101 // BST stats [
6102 PEAKi bBST=1+floorLog2(BSTwi thMAXnode);
6103 //PEAKi bBST=1;
6104 //while (BSTwi thMAXnode>>PEAKi bBST) PEAKi bBST++;
6105 // BST stats ]
6106 }
6107
6108 (void) time(&t2);
6109 if (t2 <= t1) {t2 = t1; t2++;}
6110 printf("Time for making unsorted wordlist: %d second(s)\n", (int) t2-t1);
6111 fprintf( fp_outLOG, "Bytes per second performance: %sB/s\n", _ui64toaKAZEcomma(FilesLEN/((int) t3-t1), IIT0aDi gi ts, 10) ); // Rev. 12+
6112 fprintf( fp_outLOG, "Words per second performance: %sW/s\n", _ui64toaKAZEcomma(WORDcount/((int) t3-t1), IIT0aDi gi ts, 10) ); // Rev. 12+
6113 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
6114 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, IIT0aDi gi ts, 10) );
6115 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, IIT0aDi gi ts, 10), _ui64toaKAZEcomma((unsigned long)WORDcountDistinct, IIT0aDi gi ts2, 10) );
6116 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
6117 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
6118 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
6119 NumberOfHashCollisions = WORDcountDistinct - NumberOfTrees;
6120 fprintf( fp_outLOG, "Number Of Trees(GREATER THE BETTER): %lu\n", NumberOfTrees );
6121 fprintf( fp_outLOG, "Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): %lu%s\n", (NumberOfTrees*100)/(26*31*8192), "%\0" );
6122 fprintf( fp_outLOG, "Number Of Hash Collisions(Distinct WORDs - Number Of Trees): %lu\n", NumberOfHashCollisions );
6123
6124 if (BSTorBtree == 0)
6125 {
6126     fprintf( fp_outLOG, "Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '%s'\n", _ui64toaKAZEcomma(BSTwi thMAXpeak, IIT0aDi gi ts, 10) );
6127     fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, IIT0aDi gi ts, 10) );
6128     fprintf( fp_outLOG, "Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): %s\n", _ui64toaKAZEcomma(BSTsTotal LEAFs, IIT0aDi gi ts, 10) );
6129     fprintf( fp_outLOG, "Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = %s must have PEAK = %s = rounding down of integer (1+lb(%s))\n", _ui64toaKAZEcomma(BSTwi thMAXnode, IIT0aDi gi ts, 10), _ui64toaKAZEcomma(PEAKi bBST, IIT0aDi gi ts2, 10), _ui64toaKAZEcomma(BSTwi thMAXnode, IIT0aDi gi ts3, 10) );
6130     fprintf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxNODEs = %s has PEAK = %s and LEAFs = %s\n", _ui64toaKAZEcomma(BSTcurrentNodeMAXqUANTITy, IIT0aDi gi ts4, 10), _ui64toaKAZEcomma(BSTwi thMAXnode, IIT0aDi gi ts2, 10), _ui64toaKAZEcomma(BSTwi thMAXnodePEAK, IIT0aDi gi ts3, 10), _ui64toaKAZEcomma(BSTwi thMAXnodeLEAF, IIT0aDi gi ts, 10) );
6131     fprintf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxPEAK = '%s' has NODEs = %s and LEAFs = %s\n", _ui64toaKAZEcomma(BSTcurrentPeakMAXqUANTITy, IIT0aDi gi ts4, 10), _ui64toaKAZEcomma(BSTwi thMAXpeak, IIT0aDi gi ts2, 10), _ui64toaKAZEcomma(BSTwi thMAXpeakNODE, IIT0aDi gi ts3, 10), _ui64toaKAZEcomma(BSTwi thMAXpeakLEAF, IIT0aDi gi ts, 10) );
6132     fprintf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxLEAFs = %s has NODEs = %s and PEAK = %s\n", _ui64toaKAZEcomma(BSTcurrentLeafMAXqUANTITy, IIT0aDi gi ts4, 10), _ui64toaKAZEcomma(BSTwi thMAXleaf, IIT0aDi gi ts2, 10), _ui64toaKAZEcomma(BSTwi thMAXleafNODE, IIT0aDi gi ts3, 10), _ui64toaKAZEcomma(BSTwi thMAXleafPEAK, IIT0aDi gi ts, 10) );
6133 } else
6134 {
6135     fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, IIT0aDi gi ts, 10) );
6136 }
6137
6138 for( k = 1; k < 32; k++ )
6139 { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%s utilization\n", _ui64toaKAZEzerocomma(k, IIT0aDi gi ts, 10)+(26-2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, IIT0aDi gi ts2, 10)+(26-5), _ui64toaKAZEzerocomma((((GRMBLhi l l[(int)k] * LetterBuffer)/31)>>10)+1, IIT0aDi gi ts3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLhi l l[(int)k] * LetterBuffer)/31), IIT0aDi gi ts4, 10)+(26-2), "%\0" ); // 26 are all 26-DESIRED=24
6140     if ( MAXusedBufferABS < (31 * ((MAXusedBuffer[k]>>10)+1)) / GRMBLhi l l[(int)k] ) {MAXusedBufferABS = 1+(31 * ((MAXusedBuffer[k]>>10)+1)) / GRMBLhi l l[(int)k];}
6141     Utiliza1 = Utiliza1 + (MAXusedBuffer[k]>>10)+1;
6142     Utiliza2 = Utiliza2 + (((GRMBLhi l l[(int)k] * LetterBuffer)/31)>>10)+1;
6143 }
6144 fprintf( fp_outLOG, "Total pseudo(including hash table) memory utilization: %s%s\n", _ui64toaKAZEzerocomma((Utiliza1*100)/Utiliza2, IIT0aDi gi ts, 10)+(26-2), "%\0" ); // 26 are all 26-DESIRED=24
6145 fprintf( fp_outLOG, "Total real(wordlist's words VS allocated block) memory utilization: %s/1000\n", _i64toaKAZE(((unsigned long long)TotalWLchars*1000)/memory_size, IIT0aDi gi ts, 10) ); // 26 are all 26-DESIRED=24
6146 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
6147 fprintf( fp_outLOG, "Use next time as third parameter: %lu-\n", MAXusedBufferABS ); // 26 are all 26-DESIRED=24
6148 fprintf( fp_outLOG, "Time for making unsorted wordlist: %d second(s)\n", (int) t2-t1);
6149
6150 // EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT
6151 printf("Deallocated memory in MB: %lu\n", (memory_size>>20)+1 );
6152 free(pointerFlushUNALIGN);
6153 fclose(fp_out);
6154 fclose(fp_outLOG);
6155 // SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT
6156 //printf("Uploading unsorted wordlist ... \n");
6157 if ((nlines = readlines(argv[2], &backup)) >= 0)
6158 { //printf("Number of words(lines) uploaded: %lu\n", nlines);
6159     //printf("Note1: Press 'Ctrl+C' to abort sorting, unsorted wordlist(second parameter)\n");
6160     //printf("    will remain intact(unless flushing is in progress) because of\n");
6161     //printf("    pointers-to-data are being sorted not the data itself.\n");
6162     //printf("Note2: In near future 'InsertionX26Sort' will be replaced with 'QuickX26Sort': \n");
6163     //printf("    which is much faster than 'QuickSort' applied for data-at-once!\n");

```

```

6164
6165 // !!!??? I AM DISAPPOINTED x26 is just an illusion !!!???
6166 // X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26
6167 // argc is 4|5|6 due to eventual missing BufferSize
6168 if( argc == 4 ) // not 5 due to eventual missing BufferSize
6169     k_FIX = 3;
6170 if( argc == 5 || argc == 6 )
6171     k_FIX = 4;
6172 if (*argv[k_FIX] != 'A' && *argv[k_FIX] != 'a' && *argv[k_FIX] != 'B' && *argv[k_FIX] != 'b' && *argv[k_FIX] != 'C' && *argv[k_FIX] != 'c' &&
*argv[k_FIX] != 'D' && *argv[k_FIX] != 'd')
6173 { printf("Sorting(wi th 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ... \n");
6174     /* ???!!! What an unexpected behavior! I have been hit: for SOED5.HTM(259,835 distinct words): InsertionX26Sort gives 46s, InsertionSort
gives 28s */
6175     for( k = 0; k < 26; k++ )
6176     { printf( "Sort pass %s/26 ... \r", _ui64toaKAZEzerocomma(k+1, IIT0aDi gi ts, 10)+(26-2));
6177         HEADOffsetFromStartBUKVA = TAILOffsetFromStartBUKVA;
6178         while( (TAILOffsetFromStartBUKVA < nlines) && (*backup[TAILOffsetFromStartBUKVA] - 'a' == k) )
6179             { TAILOffsetFromStartBUKVA++;
6180             }
6181         if (HEADOffsetFromStartBUKVA != TAILOffsetFromStartBUKVA)
6182             { mkqsort_main(backup + HEADOffsetFromStartBUKVA, TAILOffsetFromStartBUKVA - HEADOffsetFromStartBUKVA + 0); // backup[0..nlines-1]
6183             }
6184     }
6185 }
6186 else
6187 {
6188     if (*argv[k_FIX] == 'A' || *argv[k_FIX] == 'a')
6189     { printf("Sorting(wi th 'InsertionSort') ... ");
6190         InsertSortKAZE(backup, nlines, 0); // backup[0..nlines-1]
6191     }
6192     if (*argv[k_FIX] == 'B' || *argv[k_FIX] == 'b')
6193     { printf("Sorting(wi th 'InsertionX26Sort') ... \n");
6194         /* ???!!! What an unexpected behavior! I have been hit: for SOED5.HTM(259,835 distinct words): InsertionX26Sort gives 46s, InsertionSort
gives 28s */
6195         for( k = 0; k < 26; k++ )
6196         { printf( "Sort pass %s/26 ... \r", _ui64toaKAZEzerocomma(k+1, IIT0aDi gi ts, 10)+(26-2));
6197             HEADOffsetFromStartBUKVA = TAILOffsetFromStartBUKVA;
6198             while( (TAILOffsetFromStartBUKVA < nlines) && (*backup[TAILOffsetFromStartBUKVA] - 'a' == k) )
6199                 { TAILOffsetFromStartBUKVA++;
6200                 }
6201             if (HEADOffsetFromStartBUKVA != TAILOffsetFromStartBUKVA)
6202                 { InsertSortKAZE(backup + HEADOffsetFromStartBUKVA, TAILOffsetFromStartBUKVA - HEADOffsetFromStartBUKVA + 0, 0); // backup[0..nlines-1]
6203                 }
6204         }
6205     }
6206     if (*argv[k_FIX] == 'C' || *argv[k_FIX] == 'c')
6207     { printf("Sorting(wi th 'MultiKeyQuickSortSort' by J. Bentley and R. Sedgewick) ... ");
6208         mkqsort_main(backup, nlines); // backup[0..nlines-1]
6209     }
6210     if (*argv[k_FIX] == 'D' || *argv[k_FIX] == 'd')
6211     { printf("Sorting(wi th 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ... \n");
6212         /* ???!!! What an unexpected behavior! I have been hit: for SOED5.HTM(259,835 distinct words): InsertionX26Sort gives 46s, InsertionSort
gives 28s */
6213         for( k = 0; k < 26; k++ )
6214         { printf( "Sort pass %s/26 ... \r", _ui64toaKAZEzerocomma(k+1, IIT0aDi gi ts, 10)+(26-2));
6215             HEADOffsetFromStartBUKVA = TAILOffsetFromStartBUKVA;
6216             while( (TAILOffsetFromStartBUKVA < nlines) && (*backup[TAILOffsetFromStartBUKVA] - 'a' == k) )
6217                 { TAILOffsetFromStartBUKVA++;
6218                 }
6219             if (HEADOffsetFromStartBUKVA != TAILOffsetFromStartBUKVA)
6220                 { mkqsort_main(backup + HEADOffsetFromStartBUKVA, TAILOffsetFromStartBUKVA - HEADOffsetFromStartBUKVA + 0); // backup[0..nlines-1]
6221                 }
6222         }
6223     }
6224 }
6225 // X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26
6226
6227 printf("\nFlushing sorted words ... \n");
6228 if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
6229 { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
6230 for( j = 0; j < nlines; j++ )
6231 { //Slot = KuxHash3plus(backup[j]);
6232     //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, IIT0aDi gi ts, 10)+(26-5));
6233     fprintf(fp_out, "%s", backup[j]);
6234     fwrite(CRDlFa, 2, 1, fp_out );
6235 }
6236 (void) time(&t3);
6237 if (t3 <= t2) {t3 = t2; t3++;}
6238 printf("Time for sorting unsorted wordlist: %d second(s)\n", (int) t3-t2);
6239
6240 /*
6241 // Hash benchmarking ----- [
6242
6243 // 5[
6244 clocks1 = clock();
6245 for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6246 {
6247     for( j = 0; j < nlines; j++ )

```

```

6248     { //Slot = KuxHash3plus(backup[j]);
6249         //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, IIT0aDigits, 10)+(26-5));
6250         Slot = FNV1A_Hash_4_OCTETS(backup[j], (strlen(backup[j])>>2)); //13+++
6251     }
6252 }
6253 clocks2 = clock();
6254 printf( "Performance of 'FNV1A_Hash_4_OCTETS': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
6255         ((TotalWLchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6256 // 5]
6257 // 1[
6258 clocks1 = clock();
6259 for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6260 {
6261     for( j = 0; j < nlines; j++ )
6262     { //Slot = KuxHash3plus(backup[j]);
6263         //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, IIT0aDigits, 10)+(26-5));
6264         // To make it EVEN !!!
6265         //wrdlen = strlen(backup[j]);
6266         //if (strlen(backup[j]) != 0)
6267             Slot = FNV1A_Hash(backup[j]); //13+++
6268     }
6269 }
6270 clocks2 = clock();
6271 printf( "Performance of 'FNV1A_Hash': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
6272         ((TotalWLchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6273 // 1]
6274 // 2[
6275 clocks1 = clock();
6276 for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6277 {
6278     for( j = 0; j < nlines; j++ )
6279     { //Slot = KuxHash3plus(backup[j]);
6280         //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, IIT0aDigits, 10)+(26-5));
6281         Slot = FNV1A_Hash_4_OCTETS_31(backup[j], (strlen(backup[j])>>2)); //13+++
6282     }
6283 }
6284 clocks2 = clock();
6285 printf( "Performance of 'FNV1A_Hash_4_OCTETS_31': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
6286         ((TotalWLchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6287 // 2]
6288 // 4[
6289 clocks1 = clock();
6290 for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6291 {
6292     for( j = 0; j < nlines; j++ )
6293     { //Slot = KuxHash3plus(backup[j]);
6294         //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, IIT0aDigits, 10)+(26-5));
6295         // To make it EVEN !!!
6296         //wrdlen = strlen(backup[j]);
6297         //if (strlen(backup[j]) != 0)
6298             Slot = KuxHash3plus(backup[j]); //13++
6299     }
6300 }
6301 clocks2 = clock();
6302 printf( "Performance of 'KuxHash3plus': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
6303         ((TotalWLchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6304 // 4]
6305 // 6[
6306 clocks1 = clock();
6307 for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6308 {
6309     for( j = 0; j < nlines; j++ )
6310     { //Slot = KuxHash3plus(backup[j]);
6311         //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, IIT0aDigits, 10)+(26-5));
6312         wrdlen = strlen(backup[j]);
6313         if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
6314             Slot = FNV1A_Hash_Granularity(backup[j], wrdlen>>2, 2); //13++++
6315         else // 2x8+4=20 i.e. first contains 6 clashes
6316             Slot = FNV1A_Hash_Granularity(backup[j], wrdlen>>3, 3); //13++++
6317     } // Conclusion: two functions > 64 bytes lead to horrible slowness, so unite them in one: fit in the cache line.
6318 }
6319 clocks2 = clock();
6320 printf( "Performance of 'FNV1A_Hash_Granularity': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
6321         ((TotalWLchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6322 // 6]
6323 // Hash benchmarking ----- ]
6324 */
6325
6326 if( ( fp_outLOG = fopen( "Leprechaun.LOG", "a+" ) ) == NULL )
6327 { printf( "Leprechaun: Can't open file Leprechaun.LOG.\n" ); return( 1 ); }
6328 fprintf( fp_outLOG, "Time for sorting unsorted wordlist: %d second(s)\n\n", (int) t3-t2);
6329 printf( "Leprechaun: Done.\n" );
6330 return 0;

```

```

6331 }
6332 else
6333 { printf("Leprechaun: Input file too large, wordlist remains unsorted!\n");
6334   return 1;
6335 }
6336 // SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT
6337 } else { //if (BSTorBtree != 2) {
6338 // External Btrees [
6339
6340 // r16 [
6341 if (REUSE==2) {
6342   fclose(fp_out);
6343 }
6344 // r16 ]
6345
6346 if ( REUSE == 0 ) { // r16FIX <*><*><*><*><*><*><*><*><*><*><*><*><*><*><*> [
6347
6348 (void) time(&t1);
6349 WORDcountBOTTOM = 0;
6350 //printf("Flushing UNsorted phrases ... \r");
6351 //if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL ) // Commented since r.14+++ because passes need concatenation.
6352
6353 if( ( fp_out = fopen( argv[2], "ab+" ) ) == NULL )
6354 { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
6355 CRdLFa[0] = 13; CRdLFa[1] = 10;
6356
6357 BufStart = pointerflush;
6358 // for( j = 0; j < 28*28*28*28; j++ )
6359 for( j = 0; j < (1<<HashInBITS); j++ )
6360 {
6361     if ((j & ((1<<14)-1)) == 0) {
6362         (void) time(&t3);
6363         if (t3 <= t1) {t3 = t1; t3++;}
6364         printf("Flushing UNsorted phrases: %s%%; Shaking trees performance: %sP/s\r", _ui64toaKAZEzerocomma(((long
long)j*100)/((1<<HashInBITS)), lIT0aDi gi ts, 10)+(26-3), _ui64toaKAZEzerocomma(WORDcountBOTTOM/((int) t3-t1), lIT0aDi gi ts2, 10)+(26-10));
6365     }
6366     Slot = j<<3;
6367     memcpy( &PseudoLi nkedPoi nter_64, BufStart+Slot, 8 );
6368     if (PseudoLi nkedPoi nter_64 != 0)
6369     {
6370         NumberOfTrees++;
6371
6372 // $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse 64bit [
6373 // DONE JOB:
6374 // Must be written B-tree traverse ! with simulated stack i.e. non-recursive.
6375 // ...
6376 StackPtr = 0;
6377 while ( 2==2 ) {
6378     while (PseudoLi nkedPoi nter_64 != 0)
6379     {
6380         if (StackPtr > 8192*3-1) { printf( "\nLeprechaun: Failure! B-tree simulated stack overflow, too high B-tree!\n" ); return( 13
);}
6381         BSTstack[StackPtr] = PseudoLi nkedPoi nter_64; ++StackPtr; //ptr to Rwrld
6382 //if ( *(char *) (PseudoLi nkedPoi nter + 4 + 4 + 4 + i%31+1) == 0 ) {memcpy( PseudoLi nkedPoi nter + 4 + 4, &BufStart[NumberOfSLOTs*4], 4 );}
6383 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6384 //PseudoLi nkedPoi nterAUX_64 = PseudoLi nkedPoi nter_64 + 8 + 8 + 8 + (LongestLi nel ncl usi ve+1+4); //RW
6385 //fsetpos(fp_outRG, &PseudoLi nkedPoi nterAUX_64);
6386 //fread(&SomeByte, 1, 1, fp_outRG);
6387 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6388 // [ //r.14+
6389 PseudoLi nkedPoi nterAUX_64 = PseudoLi nkedPoi nter_64;
6390 if (BSTorBtree == 2) {
6391     fsetpos(fp_outRG, &PseudoLi nkedPoi nterAUX_64);
6392     fread(&LEAF[0], 8+8+2*(LongestLi nel ncl usi ve+1+4), 1, fp_outRG);
6393     } else { // ##### 64bit memory manipulations [
6394     memcpy( &LEAF[0], (char *)PseudoLi nkedPoi nterAUX_64, 8+8+2*(LongestLi nel ncl usi ve+1+4) );
6395     } // ##### 64bit memory manipulations ]
6396     memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLi nel ncl usi ve+1+4)], 1 );
6397     // ] //r.14+
6398 if (SomeByte == 0) // RW exists not
6399 {
6400     PseudoLi nkedPoi nterAUX_64 = PseudoLi nkedPoi nter_64 + 8 + 8;
6401     if (BSTorBtree == 2) {
6402         fsetpos(fp_outRG, &PseudoLi nkedPoi nterAUX_64);
6403         fwrite(&NULLs_64, 8, 1, fp_outRG);
6404     } else { // ##### 64bit memory manipulations [
6405     memcpy( (char *)PseudoLi nkedPoi nterAUX_64, &NULLs_64, 8 );
6406     } // ##### 64bit memory manipulations ]
6407     // [ //r.14+
6408     memcpy( &LEAF[8 + 8], &NULLs_64, 8 );
6409     // ] //r.14+
6410 }
6411 //     memcpy( &PseudoLi nkedPoi nterNEWl eft, PseudoLi nkedPoi nter, 4 );
6412 //     memcpy( &PseudoLi nkedPoi nterNEWm idl e, PseudoLi nkedPoi nter + 4, 4 );
6413 //     memcpy( &PseudoLi nkedPoi nterNEWr ight, PseudoLi nkedPoi nter + 4 + 4, 4 );
6414 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6415 //PseudoLi nkedPoi nterAUX_64 = PseudoLi nkedPoi nter_64;
6416 //fsetpos(fp_outRG, &PseudoLi nkedPoi nterAUX_64);

```

```

6417 //fread(&PseudoLinkedPointerNEWleft_64, 8, 1, fp_outRG);
6418 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
6419 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6420 //fread(&PseudoLinkedPointerNEWmiddle_64, 8, 1, fp_outRG);
6421 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
6422 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6423 //fread(&PseudoLinkedPointerNEWright_64, 8, 1, fp_outRG);
6424 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6425 // [ //r.14+
6426 memcpy( &PseudoLinkedPointerNEWleft_64, &LEAF[0], 8 );
6427 memcpy( &PseudoLinkedPointerNEWmiddle_64, &LEAF[8], 8 );
6428 memcpy( &PseudoLinkedPointerNEWright_64, &LEAF[8+8], 8 );
6429 // ] //r.14+
6430 // Give first from right to left non-zero PTR
6431 if (PseudoLinkedPointerNEWright_64 != 0 )
6432 { memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSlots*4], 4 );
6433 // PseudoLinkedPointer = PseudoLinkedPointerNEWright;
6434 // }
6435 {
6436 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
6437 if (BSTorBtree == 2) {
6438 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6439 fwrite(&NULLs_64, 8, 1, fp_outRG);
6440 } else { // ##### 64bit memory manipulations [
6441 memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLs_64, 8 );
6442 // ] // ##### 64bit memory manipulations ]
6443 PseudoLinkedPointer_64 = PseudoLinkedPointerNEWright_64;
6444 }
6445 else if (PseudoLinkedPointerNEWmiddle_64 != 0 )
6446 { memcpy( PseudoLinkedPointer + 4, &BufStart[NumberOfSlots*4], 4 );
6447 // PseudoLinkedPointer = PseudoLinkedPointerNEWmiddle;
6448 // }
6449 {
6450 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
6451 if (BSTorBtree == 2) {
6452 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6453 fwrite(&NULLs_64, 8, 1, fp_outRG);
6454 } else { // ##### 64bit memory manipulations [
6455 memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLs_64, 8 );
6456 // ] // ##### 64bit memory manipulations ]
6457 PseudoLinkedPointer_64 = PseudoLinkedPointerNEWmiddle_64;
6458 }
6459 else if (PseudoLinkedPointerNEWleft_64 != 0 )
6460 { memcpy( PseudoLinkedPointer, &BufStart[NumberOfSlots*4], 4 );
6461 // PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
6462 // }
6463 {
6464 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
6465 if (BSTorBtree == 2) {
6466 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6467 fwrite(&NULLs_64, 8, 1, fp_outRG);
6468 } else { // ##### 64bit memory manipulations [
6469 memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLs_64, 8 );
6470 // ] // ##### 64bit memory manipulations ]
6471 PseudoLinkedPointer_64 = PseudoLinkedPointerNEWleft_64;
6472 }
6473 else
6474 {
6475 PseudoLinkedPointer_64 = 0;
6476 }
6477 }
6478 if (LevelInCorona_Not_Counting_ROOT < StackPtr) LevelInCorona_Not_Counting_ROOT = StackPtr; //r.14
6479 if (StackPtr == 0) break;
6480 PseudoLinkedPointer_64 = BSTstack[--StackPtr];
6481 // memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
6482 // memcpy( &PseudoLinkedPointerNEWmiddle, PseudoLinkedPointer + 4, 4 );
6483 // memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
6484 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6485 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
6486 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6487 //fread(&PseudoLinkedPointerNEWleft_64, 8, 1, fp_outRG);
6488 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
6489 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6490 //fread(&PseudoLinkedPointerNEWmiddle_64, 8, 1, fp_outRG);
6491 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
6492 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6493 //fread(&PseudoLinkedPointerNEWright_64, 8, 1, fp_outRG);
6494 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6495 // [ //r.14+
6496 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
6497 if (BSTorBtree == 2) {
6498 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6499 fread(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
6500 } else { // ##### 64bit memory manipulations [
6501 memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+2*(LongestLineInclusive+1+4) );
6502 // ] // ##### 64bit memory manipulations ]
6503 memcpy( &PseudoLinkedPointerNEWleft_64, &LEAF[0], 8 );
6504 memcpy( &PseudoLinkedPointerNEWmiddle_64, &LEAF[8], 8 );

```

[illegible]

```

        _ui64toaKAZEcomma(LevelInCorona_Not_Counting_ROOT-1, IIT0aDigits, 10) );
6587
6588 fprintf( fp_outLOG, "Used value for third parameter in KB: %s\n", _ui64toaKAZEcomma(Thunderwith, IIT0aDigits, 10) );
6589 fprintf( fp_outLOG, "Use next time as third parameter: %s\n", _ui64toaKAZEcomma((((BufEnd_64-(unsigned long long)pointerflush_64)+1)>>10)+1,
        IIT0aDigits, 10) );
6590 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, IIT0aDigits, 10)
        );
6591
6592 // External Btrees ]
6593         if (BSTorBtree == 2) {
6594             fclose(fp_outRG);
6595             // r16
6596             if ( (REUSE == 1) && ((HashInBITS-HashChunkSizeInBITS)==0) ) { // Multiple-passes shouldn't be dumped - it is meaningless, dump when only one
                pass.
6597                 if ( ( fp_outRG = fopen( "Leprechaun_64bit.hsh", "wb+" ) ) == NULL )
6598                     { printf( "Leprechaun: Can't create file 'Leprechaun_64bit.hsh'.\n" ); return( 1 ); }
6599                 fwrite(pointerflushUNALIGN, (1<<HashInBITS)*8 + 1 + 64, 1, fp_outRG);
6600                 fclose(fp_outRG);
6601             }
6602             } else { // ##### 64bit memory manipulations [
6603                 free(pointerflushUNALIGN_64);
6604             } // ##### 64bit memory manipulations ]
6605             free(pointerflushUNALIGN);
6606             fclose(fp_outLOG);
6607             printf( "Leprechaun: Current pass done.\n" );
6608
6609             // 14+++ [
6610             TotalMemoryNeededForOnePass += (((BufEnd_64-(unsigned long long)pointerflush_64)+1)>>10)+1;
6611             WORDcountDistinctTOTAL += WORDcountDistinct;
6612             RipPasses++;
6613             if (RipPasses <= (1<<(HashInBITS-HashChunkSizeInBITS))-1) goto WhyTheHellForIsNotWorking;
6614             // // for( RipPasses = 1-1; RipPasses <= (1<<(HashInBITS-HashChunkSizeInBITS))-1; RipPasses++ )
6615             // 14+++ ]
6616             (void) time(&tMainE);
6617             if (tMainE <= tMainB) {tMainE = tMainB; tMainE++;} // This line fixes a bug in r.15
6618             printf( "\nTotal memory needed for one pass: %sKB\n", _ui64toaKAZEcomma(TotalMemoryNeededForOnePass, IIT0aDigits2, 10) );
6619             printf( "Total distinct phrases: %s\n", _ui64toaKAZEcomma(WORDcountDistinctTOTAL, IIT0aDigits2, 10) );
6620             printf( "Total time: %d second(s)\n", (int) tMainE-tMainB);
6621             printf( "Total performance: %sP/s i.e. phrases per second\n", _ui64toaKAZEcomma(WORDcount/((int) tMainE-tMainB), IIT0aDigits2, 10) );
6622
6623             printf( "Leprechaun: Done.\n" );
6624             exit(0);
6625         } //if (BSTorBtree != 2) {
6626     } // main()
6627
6628 /*
6629 TO BE DONE: Ideal Balancing BST [
6630
6631 link rotR(link h)
6632 { link x = h->l; h->l = x->r; x->r = h;
6633     return x; }
6634
6635 link rotL(link h)
6636 { link x = h->r; h->r = x->l; x->l = h;
6637     return x; }
6638
6639 link partr(link h, int k)
6640 { int t = h->l->N;
6641     if (t > k)
6642         { h->l = partr(h->l, k); h = rotR(h); }
6643     if (t < k)
6644         { h->r = partr(h->r, k-t-1); h = rotL(h); }
6645     return h;
6646 }
6647
6648 link balancer(link h)
6649 {
6650     if (h->N < 2) return h;
6651     h = partr(h, h->N/2);
6652     h->l = balancer(h->l);
6653     h->r = balancer(h->r);
6654     return h;
6655 }
6656
6657 TO BE DONE: Ideal Balancing BST ]
6658 */
6659
6660 /*
6661 #include <stdlib.h>
6662 #include "Item.h"
6663 typedef struct STnode* link;
6664 struct STnode { Item item; link l, r; int N };
6665 static link head, z;
6666 link NEW(Item item, link l, link r, int N)
6667 { link x = malloc(sizeof *x);
6668     x->item = item; x->l = l; x->r = r; x->N = N;
6669     return x;
6670 }

```

```

6671 void STinit()
6672 { head = (z = NEW(NULLitem, 0, 0, 0)); }
6673 int STcount() { return head->N; }
6674 Item searchR(link h, Key v)
6675 { Key t = key(h->item);
6676   if (h == z) return NULLitem;
6677   if eq(v, t) return h->item;
6678   if less(v, t) return searchR(h->l, v);
6679   else return searchR(h->r, v);
6680 }
6681 Item STsearch(Key v)
6682 { return searchR(head, v); }
6683 link insertR(link h, Item item)
6684 { Key v = key(item), t = key(h->item);
6685   if (h == z) return NEW(item, z, z, 1);
6686   if less(v, t)
6687     h->l = insertR(h->l, item);
6688   else h->r = insertR(h->r, item);
6689   (h->N)++; return h;
6690 }
6691 void STinsert(Item item)
6692 { head = insertR(head, item); }
6693 */
6694
6695 /*
6696 int count(link h)
6697 {
6698   if (h == NULL) return 0;
6699   return count(h->l) + count(h->r) + 1;
6700 }
6701
6702 int height(link h)
6703 { int u, v;
6704   if (h == NULL) return -1;
6705   u = height(h->l); v = height(h->r);
6706   if (u > v) return u+1; else return v+1;
6707 }
6708
6709 void printnode(char c, int h)
6710 { int i;
6711   for (i = 0; i < h; i++) printf(" ");
6712   printf("%c\n", c);
6713 }
6714
6715 void show(link x, int h)
6716 {
6717   if (x == NULL) { printnode("x", h); return; }
6718   show(x->r, h+1);
6719   printnode(x->item, h);
6720   show(x->l, h+1);
6721 }
6722 */

```