

```

0001 // This is source of Leprechaun revision 16FIX: Leprechaun_x-leton.c, copleft Sanmayce, 2012-Dec-16.
0002 // 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 retraverse.
0003
0004 // This is source of Leprechaun revision 16: Leprechaun_x-leton.c, copleft Sanmayce, 2012-Dec-13.
0005 // The new feature is the ability to reuse the external hash-tree structure.
0006 // The option is w/w similar to Z/z. This way the latency i.e. the response time is <1s.
0007
0008 // This is source of Leprechaun revision 15FIXFIX+: Leprechaun_x-leton.c, copleft Sanmayce, 2012-Dec-11.
0009 // The new feature is the ability to command Leprechaun (from inside the list file with 2 metacommands) to enter/exit INSERT mode.
0010 // This allows to control whether new (to current hash-tree structure) x-grams are to be counted [and] INSERTed.
0011
0012 // Usage:
0013 // E:\_Gamera_r15_12348>type ON.txt
0014 // Leprechaun says x-gram inserting disabled for next files: ON
0015 //
0016 // E:\_Gamera_r15_12348>type OFF.txt
0017 // Leprechaun says x-gram inserting disabled for next files: OFF
0018 //
0019 // E:\_Gamera_r15_12348>dir Your_textual_folders\b\s/a-d>go.lst
0020 // E:\_Gamera_r15_12348>copy go.lst+on.txt+_Gamera.tar.3.sorted.4andabove.lst MetaLep.lst /b
0021 // E:\_Gamera_r15_12348>type MetaLep.lst
0022 // E:\_Gamera_r15_12348>Your_textual_folders\Example.txt
0023 // Leprechaun says x-gram inserting disabled for next files: ON
0024 // _Gamera.tar.3.sorted.4andabove.txt
0025 //
0026 // E:\_Gamera_r15_12348>Leprechaun_x-leton_META_32bit_03_01p.exe MetaLep.lst MetaLep.3.wrd 1234567 Y
0027
0028 // All lines new to r15FIXFIX are with commented part //15FIXFIX+
0029
0030 /*
0031 This is source of Leprechaun revision 15FIXFIX: Leprechaun_x-leton.c, copleft Sanmayce, 2011-Dec-14. 2011-Mar-07: Fixed a small command line
    parsing bug.
0032
0033 The 15FIXFIX differs from 15fix with:
0034 [a bug fixed(REALLY FIXED!): Fixed a nasty bug causing very restrictive way of forming x-grams.]
0035 The 15fix differs from 15 with:
0036 [a bug fixed: division by zero when finishing-starting time is under 1 second
0037 Fixed a nasty bug causing very restrictive way of forming x-grams.]
0038 The 15 differs from 14++++FIXFIX with:
0039 [
0040 Only some more stats at the end.
0041 ]
0042 The 14++++FIXFIX differs from 14++++FIX with:
0043 [
0044 Bugs in LOG stats in r.14++++FIX:
0045 Not nullified variables during passes - must be nullified.
0046 Number of Trees(GREATER THE BETTER): 195,939
0047 Number of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 654,428
0048 Total Attempts to Find/Put WORDs into B-trees order 3: 39,042,828
0049 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 3
0050 ]
0051 The 14++++FIX differs from 14+++ with:
0052 [
0053 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.
0054 2)Ability to rip in passes:
0055 #define HashChunkSizeInBITS 26 // Defines the number of passes. Should be smaller or equal to HashInBITS. If HashInBITS == HashChunkSizeInBITS
    then 2^(HashInBITS-HashChunkSizeInBITS)=2^0=1 passe(s).
0056 ]
0057 The 14+++ differs from 14++ with:
0058 [
0059 //Only one must be uncommented:
0060 //define singleton
0061 //define doubleton
0062 //define tripleton
0063 #define quadrupleton
0064 //define quintupleton
0065 //define sextupleton
0066 //define septupleton
0067 //define octupleton
0068 //define nonupleton
0069 //define decupleton
0070 1 One single, singlet, singleton
0071 2 Two double, doublet, doubleton
0072 3 Three triple, triplet, tripleton
0073 4 Four quadruple, quadruplet, quadrupleton
0074 5 Five quintuple, quintuplet, quintupleton
0075 6 Six sextuple, sextuplet, sextupleton
0076 7 Seven septuple, septuplet, septupleton
0077 8 Eight octuple, octuplet, octupleton
0078 9 Nine nonuple, nonuplet, nonupleton
0079 10 Ten, decuple, decuplet, decupleton
0080 1 One ace, single, singleton, unary, unit, unity
0081 2 Two binary, brace, couple, couplet, distich, deuce, double, doubleton, duad, duality, duet, duo, dyad, pair, snake eyes, span,
    twain, twosome, yoke
0082 3 Three deuce-ace, leash, set, tercet, ternary, ternion, terzetto, threesome, tierce, trey, triad, trine, trinity, trio, triplet,
    troika, hat-trick

```

```

0083 4   Four      foursome, quadruplet, quatern, quaternary, quaternion, quaternity, quartet, tetrad
0084 5   Five      cinque, fin, fivesome, pentad, quint, quintet, quintuplet
0085 6   Six       half dozen, hexad, sestet, sextet, sextuplet, sise
0086 7   Seven     heptad, septet, septuplet
0087 8   Eight     octad, octave, octet, octonary, octuplet, ogdoad
0088 Also, in addition to 'Y' and 'Z', 'y' and 'z' were added in order to be able to dump only n-grams without occurrences.
0089 ]
0090 A lazy approach is applied in order to add occurrences of each 4-gram:
0091 - just reserve the last 4bytes in 'wrđ' for counter as follows:
0092 'LongestLineInclusive' has to be greater than 31 (51 looks good enough) in order not to miss longer 4-grams like:
    encourage_innovative_approaches_to
0093 char FourGram[LongestLineInclusive+1+4]; // 31bytes longest 4-gram + 1byte NULL + 4bytes COUNTER
0094 - the laziness lies here:
0095 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,
0096 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).
0097 - when dumping the format has to be:
0098 0,000,001\ta_b_c_d
0099 in order to sort the whole lines later with external Qsort and have easy screening for rare/wrong/useless 4-grams.
0100
0101 Comment/Uncomment accordingly in order to compile:
0102 //#define _WIN32_ENVIRONMENT_
0103 #define _POSIX_ENVIRONMENT_
0104
0105 windows compile(uncomment #include <io.h> line, ignore warnings):
0106 For Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86 use:
0107 cl /Ox /Wp64 /TcLeprechaun.c /FaLeprechaun
0108
0109 windows compile(comment #include <io.h> line, ignore warnings):
0110 For Intel(R) C++ Compiler Professional for applications running on IA-32, Version 11.1 use:
0111 icl /Ox /Wp64 /TcLeprechaun.c /FaLeprechaun /w /QxHOST
0112
0113 Linux compile(ignore warnings):
0114 gcc -D_FILE_OFFSET_BITS=64 -m64 -static -O3 -mtune=generic Leprechaun_quadruplet.c -o Leprechaun_quadruplet_r14_generic_64bits.elf
0115
0116 !!! 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:
0117 gcc -D_FILE_OFFSET_BITS=64 -m32 -static -O3 -mtune=generic Leprechaun_quadruplet.c -o Leprechaun_quadruplet_r14_generic_32bits.elf
0118
0119 [It's a little weird(Intel boosts the sort while falls behind in parsing, tested on T3400):]
0120
0121 Leprechaun_r13_7pluses_Microsoft_32-bit_16.00.30319.01.exe_vs_ Wikipedia_22,202,980_LATIN-words:
0122 words per second performance: 1,679,585w/s
0123 Time for making unsorted wordlist: 30 second(s)
0124 Time for sorting unsorted wordlist: 25 second(s)
0125
0126 Leprechaun_r13_7pluses_Intel_IA-32_11.1.exe_vs_ wikipedia_22,202,980_LATIN-words:
0127 words per second performance: 1,603,240w/s
0128 Time for making unsorted wordlist: 31 second(s)
0129 Time for sorting unsorted wordlist: 19 second(s)
0130
0131 Due to my ignorance(calderas in my C knowledge): 64bit code cannot be generated, for now.
0132 Any improvement is welcome.
0133 Enjoy!
0134 */
0135
0136 // C:\workTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_r13+++++_C_EXE>cl /Ox /Wp64 /TcLeprechaun.c /FaLeprechaun
0137 // Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
0138 // Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
0139 //
0140 // Leprechaun.c
0141 // Leprechaun.c(829) : warning C4312: 'type cast' : conversion from 'int' to 'string' of greater size
0142 // Leprechaun.c(849) : warning C4312: 'type cast' : conversion from 'int' to 'string *' of greater size
0143 // Leprechaun.c(2048) : warning C4312: 'type cast' : conversion from 'int' to 'char *' of greater size
0144 // Leprechaun.c(2063) : warning C4311: 'type cast' : pointer truncation from 'char *' to 'unsigned long'
0145 // Leprechaun.c(2068) : warning C4311: 'type cast' : pointer truncation from 'char *' to 'unsigned long'
0146 // Leprechaun.c(2371) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0147 // Leprechaun.c(2570) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0148 // Leprechaun.c(2626) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0149 // Leprechaun.c(2657) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0150 // Leprechaun.c(2663) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0151 // Leprechaun.c(2668) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0152 // Leprechaun.c(2696) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0153 // Leprechaun.c(2729) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0154 // Leprechaun.c(2743) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0155 // Leprechaun.c(2755) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0156 // Microsoft (R) Incremental Linker Version 7.10.3077
0157 // Copyright (C) Microsoft Corporation. All rights reserved.
0158 //
0159 // /out:Leprechaun.exe
0160 // Leprechaun.obj
0161 //
0162 // C:\workTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_r13+++++_C_EXE>
0163
0164 /*
0165 Below is the gain in 13++ and 13+++:
0166
0167 words per second performance: 5,974,513w/s

```

```

0168 word count: 4,582,451,898 of them 9,177,221 distinct
0169 Number Of Trees(GREATER THE BETTER): 2855919
0170 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 6321302
0171
0172 words per second performance: 6,329,353w/s
0173 word count: 4,582,451,898 of them 9,177,221 distinct
0174 Number Of Trees(GREATER THE BETTER): 2958681
0175 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 6218540
0176
0177 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.
0178 Thanks to Fowler/Noll/vo hash inventors.
0179 */
0180
0181 /*
0182 Let's see the supplementary-clash on Intel Pentium T3400 Merom-1M 2166MHz:
0183 Binary-Search-Trees vs B-Trees of order 3
0184
0185 C:\workTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST>Leprechaun_Microsoft.exe Leprechaun_vs_wikipedia_en-
WORDS.lst Leprechaun_vs_wikipedia_en-WORDS.wrd 4777 x
0186 Leprechaun(Fast Greedy Word-Ripper), revision 13+++++, written by Svalqyatchx.
0187 Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
0188 Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
0189 also the performance of a 3-way hash + 6,602,752 B-Trees of order 3.
0190 Size of input file with files for Leprechauning: 27
0191 Allocating memory 1863MB ... OK
0192 Size of Input TEXTual file: 146,973,879
0193 \; word count: 12,561,874 of them 12,561,874 distinct; Done: 64/64
0194 Bytes per second performance: 14,697,387B/s
0195 words per second performance: 1,256,187w/s
0196 Flushing unsorted words ...
0197 Time for making unsorted wordlist: 15 second(s)
0198 Deallocated memory in MB: 1863
0199 Allocated memory for words in MB: 141
0200 Allocated memory for pointers-to-words in MB: 48
0201 Sorting(with 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...
0202 Sort pass 26/26 ...
0203 Flushing sorted words ...
0204 Time for sorting unsorted wordlist: 14 second(s)
0205 Leprechaun: Done.
0206
0207 [An excerpt of Leprechaun.LOG:]
0208 Number Of Trees(GREATER THE BETTER): 2786806
0209 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 58,935,172
0210 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,806
0211
0212 C:\workTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST>Leprechaun_Microsoft.exe Leprechaun_vs_wikipedia_en-
WORDS.lst Leprechaun_vs_wikipedia_en-WORDS.wrd 4777 y
0213 Leprechaun(Fast Greedy Word-Ripper), revision 13+++++, written by Svalqyatchx.
0214 Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
0215 Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
0216 also the performance of a 3-way hash + 6,602,752 B-Trees of order 3.
0217 Size of input file with files for Leprechauning: 27
0218 Allocating memory 1863MB ... OK
0219 Size of Input TEXTual file: 146,973,879
0220 \; word count: 12,561,874 of them 12,561,874 distinct; Done: 64/64
0221 Bytes per second performance: 24,495,646B/s
0222 words per second performance: 2,093,645w/s
0223 Flushing unsorted words ...
0224 Time for making unsorted wordlist: 12 second(s)
0225 Deallocated memory in MB: 1863
0226 Allocated memory for words in MB: 141
0227 Allocated memory for pointers-to-words in MB: 48
0228 Sorting(with 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...
0229 Sort pass 26/26 ...
0230 Flushing sorted words ...
0231 Time for sorting unsorted wordlist: 14 second(s)
0232 Leprechaun: Done.
0233
0234 [An excerpt of Leprechaun.LOG:]
0235 Number Of Trees(GREATER THE BETTER): 2786806
0236 Total Attempts to Find/Put WORDs into B-trees order 3: 18,534,910
0237
0238 C:\workTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST>type Leprechaun_vs_wikipedia_en-WORDS.lst
0239 wikipedia-en-html.tar.wrd
0240
0241 C:\workTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST>dir Leprechaun_vs_wikipedia_en-WORDS.*
0242 Volume in drive C is H320_Vo12
0243 Volume Serial Number is A094-FAE2
0244
0245 Directory of C:\workTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST
0246
0247 09/14/2010 06:04 AM 27 Leprechaun_vs_wikipedia_en-WORDS.lst
0248 09/15/2010 02:51 AM 146,973,879 Leprechaun_vs_wikipedia_en-WORDS.wrd
0249 2 File(s) 146,973,906 bytes
0250 0 Dir(s) 965,787,648 bytes free
0251
0252 Conclusion:

```

```

0253 18,534,910/12,561,874=1.475 Average Attempts to Find/Put WORDs into B-trees order 3, not bad at all.
0254 */
0255
0256 // To do: must learn how to align, at last.
0257 /*
0258 Matt Mahoney ZPAQ fragment:
0259 T *data; // allocated memory
0260 int offset;
0261 ...
0262 offset=64-int((long)data&63);
0263 data=(T*)((char*)data+offset); // adjust to 64 byte boundary
0264
0265 quicklz.c fragment:
0266 #define QLZ_ALIGNMENT_PADD 8
0267 unsigned char *scratch_aligned = (unsigned char *)scratch_compress + QLZ_ALIGNMENT_PADD - (((size_t)scratch_compress) % QLZ_ALIGNMENT_PADD);
0268 size_t *buffer_size = (size_t *)scratch_aligned;
0269
0270 minilzo.c fragment:
0271 #define lzo_uintptr_t unsigned long
0272 #define PTR(a) ((lzo_uintptr_t) (a))
0273 #define PTR_LINEAR(a) PTR(a)
0274 #define PTR_ALIGNED_4(a) ((PTR_LINEAR(a) & 3) == 0)
0275 */
0276
0277 //__declspec(align(64)) int BigArray[1024]; // windows syntax
0278 //or
0279 //int BigArray[1024] __attribute__((aligned(64))); // Linux syntax
0280
0281 #if defined(_WIN32_ENVIRONMENT_)
0282 __declspec(align(64))
0283 #else
0284 //__attribute__((aligned(64)));
0285 #endif /* defined(_WIN32_ENVIRONMENT_) */
0286
0287 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:
0288 //; 792 : hash32 = FNV_32A_OP32(hash32, *(UINT*)p&0xFFFF);
0289
0290 typedef unsigned int UINT;
0291 typedef unsigned int DWORD;
0292
0293 /*
0294 Enter-the-BESTer or an alchemical clash of pairs of primes.
0295
0296 When an x-bit hash where x < 16 and is not a power of 2 is needed,
0297 here comes 'FNV1A_Hash_4_OCTETS': a slightly tuned FNV1A hash for a huge(22,202,980) wordlist of latin-letters-words.
0298
0299 Two improvements for the generic(base) FNV1A hash:
0300 - first, better speed: by reducing 'imul' instructions when string is 4++ chars
0301 - second, better dispersion: by experimenting(superficially-lite test done, so far) with 'FNV1_32_PRIME'
0302
0303 Or more concretely:
0304 - For FNV1_32_INIT = 2166136261
0305 - Giving to 'FNV1_32_PRIME' all primes between 2 and 11987
0306 - Shifting by 16bits instead of 13bits, when 8192 slots are used
0307
0308 C code:
0309 typedef unsigned char u_int8_t;
0310 typedef unsigned long u_int32_t;
0311
0312 #define FNV1_32_INIT ((u_int32_t)2166136261)
0313 #define FNV1_32_PRIME ((u_int32_t)1607)
0314
0315 #define FNV_32A_OP(hash, octet) \
0316     (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * FNV1_32_PRIME)
0317
0318 #define FNV_32A_OP32(hash, octet) \
0319     (((u_int32_t)(hash) ^ (u_int32_t)(octet)) * FNV1_32_PRIME)
0320
0321 0800 // Invoking: FNV1A_Hash_4_OCTETS(wrd, wrdlen>2) // = 0,1,2,3,4,5,6,7 [1..31]
0322 0801 int FNV1A_Hash_4_OCTETS(char *str, int wrdlen_QUADRUPLITS)
0323 0802 {
0324 0803 u_int32_t hash;
0325 0804 char *p;
0326 0805
0327 0806 hash = FNV1_32_INIT;
0328 0807 p=str;
0329 0808
0330 0809 // The goal of stage #1: to reduce number of 'imul's.
0331 0810
0332 0811 // Stage #1:
0333 0812 for (; wrdlen_QUADRUPLITS != 0; --wrdlen_QUADRUPLITS) {
0334 0813     hash = FNV_32A_OP32(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
0335 0814     p=p+4; // add eax, 4
0336 0815 }
0337 0816
0338 0817 // Stage #2:
0339 0818 for (; *p; ++p) {

```

```

0340 0819     hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [eax]
0341 0820 }
0342 0821
0343 0822 //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
0344 0823 return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
0345 0824 }
0346
0347 Assembler code:
0348 _FNV1A_Hash_4_OCTETS PROC NEAR
0349 ; Line 812
0350 mov edx, DWORD PTR _wordlen_QUADRUPLETS$[esp-4]
0351 test edx, edx
0352 mov eax, DWORD PTR _str$[esp-4]
0353 push esi
0354 mov esi, DWORD PTR _FNV1_32_PRIME
0355 mov ecx, -2128831035
0356 je SHORT $L1612
0357 push edi
0358 npad 7
0359 $L1610:
0360 ; Line 813
0361 mov edi, DWORD PTR [eax]
0362 xor edi, ecx
0363 imul edi, esi
0364 ; Line 814
0365 add eax, 4
0366 dec edx
0367 mov ecx, edi
0368 jne SHORT $L1610
0369 pop edi
0370 $L1612:
0371 ; Line 818
0372 mov dl, BYTE PTR [eax]
0373 test dl, dl
0374 je SHORT $L1619
0375 $L1617:
0376 ; Line 819
0377 movzx     edx, dl
0378 xor edx, ecx
0379 imul edx, esi
0380 inc eax
0381 mov ecx, edx
0382 mov dl, BYTE PTR [eax]
0383 test dl, dl
0384 jne SHORT $L1617
0385 $L1619:
0386 ; Line 823
0387 mov eax, ecx
0388 shr eax, 16
0389 xor eax, ecx
0390 and eax, 8191
0391 pop esi
0392 ; Line 824
0393 ret 0
0394 _FNV1A_Hash_4_OCTETS ENDP
0395
0396

```

0397 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.

0398
0399 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'.

0400
0401 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.
0402 This is critical for speed performance e.g. when 30,974,750,142 words, the case of wikipedia-en-html.tar, must be hashed.
0403 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.
0404 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.

0405
0406 Between 1..1166136247 there are 58,834,113 primes (inclusive).
0407 Between 1..16777619 there are 1,077,891 primes (inclusive).
0408 Or 58834113*1077891 = 63,416,760,895,683 pairs or 2,010,932 years needed at one-pair-per-second rate.

0409
0410 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.
0411 Since the magnum corpus of words is static-enough, THE pair is worthy to be found.

0412
0413 It doesn't take a think-tank to see the superiority of FNV, Fowler/Noll/Vo did reveal a thing of beauty.

0414
0415 Performance of 'FNV1A_Hash_4_OCTETS': 10236 words/clock or 105 MB/s|3,549,448 used slots (best)

0416
0417 CASE #1: with 'if (strlen(backup[j]) != 0)' before each execution
0418 Performance of 'KuxHash3plus' aka '2in1': 8076 words/clock or 82 MB/s|3,410,463 used slots (worst)
0419 Performance of 'FNV1A_Hash': 8079 words/clock or 83 MB/s|3,527,916 used slots
0420 Performance of 'FNV1A_Hash_SHIFTless_XORless': 8109 words/clock or 83 MB/s|3,540,323 used slots

0421
0422 CASE #2: without 'if (strlen(backup[j]) != 0)' before each execution
0423 Performance of 'KuxHash3plus' aka '2in1': 11673 words/clock or 119 MB/s|3,410,463 used slots (worst)
0424 Performance of 'FNV1A_Hash': 11558 words/clock or 118 MB/s|3,527,916 used slots
0425 Performance of 'FNV1A_Hash_SHIFTless_XORless': 11570 words/clock or 118 MB/s|3,540,323 used slots

```

0426
0427 Note:
0428 The 'strlen' overhead(CASE #1) is necessary due to priorly(before hash invocation) needed len-of-string for 'FNV1A_Hash_4_OCTETS'.
0429 Almost always, that is the case, since parsing of incoming text must know length of words/lines/files.
0430 In case of not knowing this length: ((119-105)/105)*100% = 13% degradation is unacceptable.
0431 The 'strlen' is an awful brake.
0432 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.
0433 */
0434
0435 /*
0436 FNV1_32_PRIME: //?: 16777619
0437
0438 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
0439 words per second performance: 1,046,822w/s
0440 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0441 Size of all TEXTual Files: 146,973,879
0442 word count: 12,561,874 of them 12,561,874 distinct
0443 Number Of Trees(GREATER THE BETTER): 2775839
0444
0445 Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 72 and LEAFs = 15
0446 words per second performance: 1,356,588w/s
0447 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0448 Size of all TEXTual Files: 415,982,896
0449 word count: 35,271,297 of them 22,202,980 distinct
0450 Number Of Trees(GREATER THE BETTER): 3539690
0451
0452 FNV1_32_PRIME: //3549448: 1607
0453
0454 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
0455 words per second performance: 1,046,822w/s
0456 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0457 Size of all TEXTual Files: 146,973,879
0458 word count: 12,561,874 of them 12,561,874 distinct
0459 Number Of Trees(GREATER THE BETTER): 2783970
0460
0461 Above Binary-Search-Tree with MaxPEAK = 38 has NODEs = 50 and LEAFs = 11
0462 words per second performance: 1,410,851w/s
0463 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0464 Size of all TEXTual Files: 415,982,896
0465 word count: 35,271,297 of them 22,202,980 distinct
0466 Number Of Trees(GREATER THE BETTER): 3549395
0467
0468 FNV1_32_PRIME: //3550132: 175757909
0469
0470 Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
0471 words per second performance: 966,298w/s
0472 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0473 Size of all TEXTual Files: 146,973,879
0474 word count: 12,561,874 of them 12,561,874 distinct
0475 Number Of Trees(GREATER THE BETTER): 2784479
0476
0477 Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 64 and LEAFs = 12
0478 words per second performance: 1,410,851w/s
0479 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0480 Size of all TEXTual Files: 415,982,896
0481 word count: 35,271,297 of them 22,202,980 distinct
0482 Number Of Trees(GREATER THE BETTER): 3550115
0483
0484 FNV1_32_PRIME: //3550687: 201887489
0485
0486 Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
0487 words per second performance: 966,298w/s
0488 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0489 Size of all TEXTual Files: 146,973,879
0490 word count: 12,561,874 of them 12,561,874 distinct
0491 Number Of Trees(GREATER THE BETTER): 2784377
0492
0493 Above Binary-Search-Tree with MaxPEAK = 40 has NODEs = 55 and LEAFs = 11
0494 words per second performance: 1,356,588w/s
0495 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0496 Size of all TEXTual Files: 415,982,896
0497 word count: 35,271,297 of them 22,202,980 distinct
0498 Number Of Trees(GREATER THE BETTER): 3550528
0499
0500 FNV1_32_PRIME: //3550733: 172783361
0501
0502 Above Binary-Search-Tree with MaxPEAK = 59 has NODEs = 59 and LEAFs = 1
0503 words per second performance: 1,046,822w/s
0504 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0505 Size of all TEXTual Files: 146,973,879
0506 word count: 12,561,874 of them 12,561,874 distinct
0507 Number Of Trees(GREATER THE BETTER): 2786362
0508
0509 Above Binary-Search-Tree with MaxPEAK = 38 has NODEs = 70 and LEAFs = 17
0510 words per second performance: 1,410,851w/s
0511 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0512 Size of all TEXTual Files: 415,982,896

```

0513 word count: 35,271,297 of them 22,202,980 distinct
 0514 Number Of Trees(GREATER THE BETTER): 3550746
 0515
 0516 FNV1_32_PRIME: //3550929: 204312319
 0517
 0518 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
 0519 words per second performance: 966,298w/s
 0520 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
 0521 Size of all TEXTual Files: 146,973,879
 0522 word count: 12,561,874 of them 12,561,874 distinct
 0523 Number Of Trees(GREATER THE BETTER): 2785581
 0524
 0525 Above Binary-Search-Tree with MaxPEAK = 37 has NODEs = 55 and LEAFs = 12
 0526 words per second performance: 1,356,588w/s
 0527 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0528 Size of all TEXTual Files: 415,982,896
 0529 word count: 35,271,297 of them 22,202,980 distinct
 0530 Number Of Trees(GREATER THE BETTER): 3550886
 0531
 0532 Leprechaun_Microsoft.exe: FNV1_32_PRIME: //3551736: 107712257
 0533
 0534 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
 0535 words per second performance: 1,046,822w/s
 0536 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
 0537 Size of all TEXTual Files: 146,973,879
 0538 word count: 12,561,874 of them 12,561,874 distinct
 0539 Number Of Trees(GREATER THE BETTER): 2786515
 0540
 0541 Above Binary-Search-Tree with MaxPEAK = 36 has NODEs = 64 and LEAFs = 15
 0542 words per second performance: 1,356,588w/s
 0543 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0544 Size of all TEXTual Files: 415,982,896
 0545 word count: 35,271,297 of them 22,202,980 distinct
 0546 Number Of Trees(GREATER THE BETTER): 3551744
 0547
 0548 Leprechaun_Intel.exe: FNV1_32_PRIME: //3551736: 107712257
 0549
 0550 Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
 0551 words per second performance: 1,256,187w/s
 0552 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
 0553 Size of all TEXTual Files: 146,973,879
 0554 word count: 12,561,874 of them 12,561,874 distinct
 0555 Number Of Trees(GREATER THE BETTER): 2786515
 0556
 0557 Above Binary-Search-Tree with MaxPEAK = 36 has NODEs = 64 and LEAFs = 15
 0558 words per second performance: 1,603,240w/s
 0559 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0560 Size of all TEXTual Files: 415,982,896
 0561 word count: 35,271,297 of them 22,202,980 distinct
 0562 Number Of Trees(GREATER THE BETTER): 3551744
 0563
 0564 wow: 1,603,240w/s vs 1,356,588w/s respectively Leprechaun_Intel.exe vs Leprechaun_Microsoft.exe, i.e. 18% betterment, no joke!
 0565
 0566 Alchemical search for best PRIME-PAIR revision uses next line:
 0567 Slot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13+++
 0568 This revision uses next lines:
 0569 if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
 0570 Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>2, 2)<<2; //13++++
 0571 else // 2x8+4=20 i.e. first contains 6 clashes
 0572 Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>3, 3)<<2; //13++++
 0573
 0574 ! 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.
 0575
 0576 Leprechaun.exe: FNV1_32_PRIME: //3551961: 428904191
 0577
 0578 Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
 0579 words per second performance: 966,298w/s
 0580 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
 0581 Size of all TEXTual Files: 146,973,879
 0582 word count: 12,561,874 of them 12,561,874 distinct
 0583 Number Of Trees(GREATER THE BETTER): 2786383
 0584
 0585 Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 71 and LEAFs = 16
 0586 words per second performance: 1,410,851w/s
 0587 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
 0588 Size of all TEXTual Files: 415,982,896
 0589 word count: 35,271,297 of them 22,202,980 distinct
 0590 Number Of Trees(GREATER THE BETTER): 3551503
 0591
 0592 Leprechaun.exe: FNV1_32_PRIME: //3552103: 588411137
 0593
 0594 Above Binary-Search-Tree with MaxPEAK = 6 has NODEs = 6 and LEAFs = 1
 0595 Size of all TEXTual Files: 4,067,439
 0596 word count: 358,798 of them 351,116 distinct
 0597 Number Of Trees(GREATER THE BETTER): 310622
 0598 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 310,622
 0599

```

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

```



```

0688 // _CRTIMP __int64 __cdecl _lseeki64(int, __int64, int);
0689 // _CRTIMP __int64 __cdecl _telli64(int);
0690 // _CRTIMP __int64 __cdecl _filelengthi64(int);
0691 // above 3 are in 'io.h'
0692
0693 // _CRTIMP int __cdecl fseek(FILE *, long, int);
0694 // _CRTIMP long __cdecl ftell(FILE *);
0695 // _CRTIMP int __cdecl fclose(FILE *);
0696
0697 // #ifndef _SIZE_T_DEFINED
0698 // #ifdef _WIN64
0699 // typedef unsigned __int64 size_t;
0700 // #else
0701 // typedef _w64 unsigned int size_t;
0702 // #endif
0703 // #define _SIZE_T_DEFINED
0704 // #endif
0705
0706 // typedef __int64 fpos_t;
0707
0708 // Linux: ~~~~~
0709 // size_t fread (void *data, size_t size, size_t count, FILE *stream)
0710 // size_t fwrite (const void *data, size_t size, size_t count, FILE *stream)
0711 // int fgetpos (FILE *stream, fpos_t *position)
0712 // int fsetpos (FILE *stream, const fpos_t *position)
0713
0714 // FILE * fopen64 (const char *filename, const char *opentype)
0715 // int fseeko64 (FILE *stream, off64_t offset, int whence)
0716 // off64_t ftello64 (FILE *stream)
0717 // int fclose (FILE *stream)
0718
0719 // off_t lseek (int filedes, off_t offset, int whence)
0720 // above 1 is in 'unistd.h'
0721
0722
0723 // ===== MUST work both for windows and Linux =====
0724 //Only one must be uncommented:
0725 #define _WIN32_ENVIRONMENT_
0726 // #define _POSIX_ENVIRONMENT_
0727
0728 //Only one must be uncommented:
0729 // #define singleton
0730 // #define doubleton
0731 #define tripleton
0732 // #define quadruplet
0733 // #define quintuplet
0734 // #define sextuplet
0735 // #define septuplet
0736 // #define octuplet
0737 // #define nonuplet
0738 // #define decuplet
0739
0740 #ifdef singleton
0741 #define _ngram_ 1
0742 #endif
0743 #ifdef doubleton
0744 #define _ngram_ 2
0745 #endif
0746 #ifdef tripleton
0747 #define _ngram_ 3
0748 #endif
0749 #ifdef quadruplet
0750 #define _ngram_ 4
0751 #endif
0752 #ifdef quintuplet
0753 #define _ngram_ 5
0754 #endif
0755 #ifdef sextuplet
0756 #define _ngram_ 6
0757 #endif
0758 #ifdef septuplet
0759 #define _ngram_ 7
0760 #endif
0761 #ifdef octuplet
0762 #define _ngram_ 8
0763 #endif
0764 #ifdef nonuplet
0765 #define _ngram_ 9
0766 #endif
0767 #ifdef decuplet
0768 #define _ngram_ 10
0769 #endif
0770
0771 #ifndef NULL
0772 #ifdef __cplusplus
0773 #define NULL 0
0774 #else
0775 #define NULL ((void*)0)

```

```

0776 #endif
0777 #endif
0778
0779 #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)!
0780 #define HashChunkSizeInBITS 24 // Defines the number of passes. Should be smaller or equal to HashInBITS. If HashInBITS == HashChunkSizeInBITS
    then 2^(HashInBITS-HashChunkSizeInBITS)=2^0=1 pass(es).
0781 /*
0782 Tests done on super-speed-ramdisk 1800MB:
0783 Leprechaun_quadrupleton 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.
0784
0785 [Variant (HashInBITS 26 - 0) with 512MB hash:]
0786
0787 Leprechaun_quadrupleton (Fast Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
0788 Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
0789 Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
0790 Feature2: The bottleneck is seek-time, if the external memory has latency 100+microseconds then look further.
0791 Size of input file with files for Leprechauning: 19
0792 Allocating HASH memory 536,870,977 bytes ... OK
0793 Allocating/ZEROing 1,292,478,478 bytes swap file ... OK
0794 Size of Input TEXTual file: 206,908,949
0795 |; 0,065,139P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
0796 Bytes per second performance: 718,433B/s
0797 Phrases per second performance: 65,139P/s
0798 Time for putting phrases into trees: 288 second(s)
0799 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,014,439P/s
0800 Time for shaking phrases from trees: 704 second(s)
0801 Dump LEAFwise also [
0802 Bytes per second performance: 736,330B/s
0803 Phrases per second performance: 66,762P/s
0804 Time for putting phrases into trees: 281 second(s)
0805 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,023,807P/s
0806 Time for shaking phrases from trees: 427 second(s)
0807 Dump LEAFwise also ]
0808 Leprechaun: Done.
0809
0810 Leprechaun report:
0811 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 731,746
0812 Number Of Trees(GREATER THE BETTER): 9,433,894
0813 Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 69,623
0814 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 1
0815 Used value for third parameter in KB: 1,262,186
0816 Use next time as third parameter: 1,262,186
0817 Total Attempts to Find/Put WORDs into B-trees order 3: 365,283
0818
0819 [Variant (HashInBITS 26 - 10) with 512KB hash:]
0820
0821 Leprechaun_quadrupleton (Fast Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
0822 Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
0823 Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
0824 Feature2: The bottleneck is seek-time, if the external memory has latency 100+microseconds then look further.
0825 Size of input file with files for Leprechauning: 19
0826 Allocating HASH memory 524,353 bytes ... OK
0827 Allocating/ZEROing 1,292,478,478 bytes swap file ... OK
0828 Size of Input TEXTual file: 206,908,949
0829 |; 0,014,331P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
0830 Bytes per second performance: 158,066B/s
0831 Phrases per second performance: 14,331P/s
0832 Time for putting phrases into trees: 1309 second(s)
0833 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,019,739P/s
0834 Time for shaking phrases from trees: 515 second(s)
0835 Dump LEAFwise also [
0836 Bytes per second performance: 158,429B/s
0837 Phrases per second performance: 14,364P/s
0838 Time for putting phrases into trees: 1306 second(s)
0839 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,041,492P/s
0840 Time for shaking phrases from trees: 245 second(s)
0841 Dump LEAFwise also ]
0842 Dump & Insert LEAFwise also [
0843 Bytes per second performance: 174,459B/s
0844 Phrases per second performance: 15,818P/s
0845 Time for putting phrases into trees: 1186 second(s)
0846 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,041,323P/s
0847 Time for shaking phrases from trees: 246 second(s)
0848 Dump & Insert LEAFwise also ]
0849 Leprechaun: Done.
0850
0851 Leprechaun report:
0852 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 10,100,104
0853 Number Of Trees(GREATER THE BETTER): 65,536
0854 Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 7,522,788
0855 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 6
0856 Used value for third parameter in KB: 1,262,186
0857 Use next time as third parameter: 1,007,825
0858 Total Attempts to Find/Put WORDs into B-trees order 3: 84,868,241
0859
0860 [Variant (HashInBITS 26 - 20) with 512 hash:]

```

```

0861
0862 Leprechaun_quadrupleton (Fast Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
0863 Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
0864 Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
0865 Feature2: The bottleneck is seek-time, if the external memory has latency 100+microseconds then look further.
0866 Size of input file with files for Leprechauning: 19
0867 Allocating HASH memory 577 bytes ... OK
0868 Allocating/ZEROing 1,292,478,478 bytes swap file ... OK
0869 Size of Input TEXTual file: 206,908,949
0870 |; 0,007,717P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
0871 Bytes per second performance: 85,112B/s
0872 Phrases per second performance: 7,717P/s
0873 Time for putting phrases into trees: 2431 second(s)
0874 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,019,777P/s
0875 Time for shaking phrases from trees: 514 second(s)
0876 Leprechaun: Done.
0877
0878 Leprechaun report:
0879 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 10,165,576
0880 Number Of Trees(GREATER THE BETTER): 64
0881 Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 7,592,585
0882 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 14
0883 Used value for third parameter in KB: 1,262,186
0884 Use next time as third parameter: 1,008,399
0885 Total Attempts to Find/Put WORDs into B-trees order 3: 271,393,689
0886
0887 r.14++ physical memory test [Variant (HashInBITS 26 - 0) with 512MB hash]:
0888 D:\_KAZE_new-stuff\Leprechaun_quadrupleton_r14+_64bit_Physical-n-Virtual>OSHO-TEST_INTERNAL.BAT
0889 Leprechaun_quadrupleton (Fast-In-Future Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
0890 Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
0891 Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
0892 Feature2: If the external memory has latency 99+microseconds then !(look no further), IOPS(seek-time) rules.
0893 Size of input file with files for Leprechauning: 19
0894 Allocating HASH memory 536,870,977 bytes ... OK
0895 Allocating memory 1233MB ... OK
0896 Size of Input TEXTual file: 206,908,949
0897 |; 1,042,234P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
0898 Bytes per second performance: 11,494,941B/s
0899 Phrases per second performance: 1,042,234P/s
0900 Time for putting phrases into trees: 18 second(s)
0901 Flushing UNSorted phrases: 100%; Shaking trees performance: 0,597,978P/s
0902 Time for shaking phrases from trees: 17 second(s)
0903 Leprechaun: Done.
0904
0905 D:\_KAZE_new-stuff\Leprechaun_quadrupleton_r14+_64bit_Physical-n-Virtual>type Leprechaun.LOG
0906 Leprechaun report:
0907 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 731,746
0908 Number Of Trees(GREATER THE BETTER): 9,433,894
0909 Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 69,623
0910 Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 1
0911 Used value for third parameter in KB: 1,262,186
0912 Use next time as third parameter: 1,262,186
0913 Total Attempts to Find/Put WORDs into B-trees order 3: 365,283
0914
0915 D:\_KAZE_new-stuff\Leprechaun_quadrupleton_r14+_64bit_Physical-n-Virtual>
0916 */
0917
0918 // To do #1: Put this 31 in MAXw1: 'int MAXw1 = 31;'
0919 // To do #2: No need of flushing unsorted words to file: make backup[] array
0920 //         instead of writing. And mostly sort 26 times!
0921 // HEAVY BUG in r.7: unsigned long Hill(unsigned long n)
0922 //         is NOT identical with
0923 //         unsigned long GRMBLhill[32]; // 00 not used, only 01..31
0924 //         BECAUSE DUMBEST DUMB Array GRMBLhill expects 'int' not
0925 //         'unsigned long' !!!
0926
0927 #include <stdio.h>
0928 #include <ctype.h>
0929 #include <time.h>
0930 #if defined(_WIN32_ENVIRONMENT_)
0931 #include <io.h> // needed for Windows 'lseeki64' and 'telli64'
0932 //Above line must be commented in order to compile with Intel C compiler: an error "can't find io.h" occurs.
0933 #else
0934 #endif /* defined(_WIN32_ENVIRONMENT_) */
0935
0936 typedef unsigned char char_t;
0937 typedef char_t *string;
0938
0939 clock_t clocks1, clocks2;
0940 int Bozan;
0941
0942 typedef unsigned char u_int8_t; //FNV only
0943 typedef unsigned long u_int32_t; //FNV only
0944 typedef unsigned long long u_int64_t; //FNV only
0945
0946 // SINHA fragment[
0947
0948 #define swapKAZE(a, b) { t = *(a); *(a) = *(b); *(b) = t; }

```

```

0949
0950 static void InsertSortKAZE(string *a, int n, int d) //void inssort(unsigned char **a, int n, int d)
0951 { string *pi, *pj, s, t; //unsigned char **pi, **pj, *s, *t;
0952   for (pi = a + 1; --n > 0; pi++)
0953     for (pj = pi; pj > a; pj--) {
0954       /* Inline strcmp: break if *(pj-1) <= *pj */
0955       for (s=*(pj-1)+d, t=*pj+d; *s==*t && *s!=0; s++, t++)
0956         ;
0957       if (*s <= *t)
0958         break;
0959       swapKAZE(pj, pj-1);
0960     }
0961 }
0962
0963 //int cmpit(unsigned char **h1, unsigned char **h2)
0964 //{
0965 //  return( strcmp(*h1, *h2) );
0966 //}
0967
0968 //int scmp( unsigned char *s1, unsigned char *s2 )
0969 //{
0970 //  while( *s1 != '\0' && *s1 == *s2 )
0971 //    {
0972 //      s1++;
0973 //      s2++;
0974 //    }
0975 //  return( *s1-*s2 );
0976 //}
0977
0978 //static void simplesort(string a[], int n, int b)
0979 //{
0980 //  int i, j;
0981 //  string tmp;
0982 //  for (i = 1; i < n; i++)
0983 //    for (j = i; j > 0 && scmp(a[j-1]+b, a[j]+b) > 0; j--)
0984 //      { tmp = a[j]; a[j] = a[j-1]; a[j-1] = tmp; }
0985 //}
0986
0987
0988 // SINHA fragment]
0989
0990 // mkqsort.c BEGIN *****
0991 /*
0992  Multikey quicksort, a radix sort algorithm for arrays of character
0993  strings by Bentley and Sedgewick.
0994
0995  J. Bentley and R. Sedgewick. Fast algorithms for sorting and
0996  searching strings. In Proceedings of 8th Annual ACM-SIAM Symposium
0997  on Discrete Algorithms, 1997.
0998
0999  http://www.CS.Princeton.EDU/~rs/strings/index.html
1000
1001  The code presented in this file has been tested with care but is
1002  not guaranteed for any purpose. The writer does not offer any
1003  warranties nor does he accept any liabilities with respect to
1004  the code.
1005
1006  Ranjan Sinha, 1 jan 2003.
1007
1008  School of Computer Science and Information Technology,
1009  RMIT University, Melbourne, Australia
1010  rsinha@cs.rmit.edu.au
1011
1012 */
1013
1014 //include "sortstring.h"
1015
1016 /* MULTIKEY QUICKSORT */
1017
1018 #ifndef min
1019 #define min(a, b) ((a)<=(b) ? (a) : (b))
1020 #endif
1021
1022
1023 // ----- BTREE [
1024 #define false -1
1025 #define true 0
1026
1027 struct nodeBTREE {
1028   int data;
1029   struct nodeBTREE* left;
1030   struct nodeBTREE* right;
1031 };
1032
1033 // ----- BTREE ]
1034
1035
1036 /* ssort2 -- Faster Version of Multikey Quicksort */

```

```

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

```

```

1125 {
1126     int nlines = 0;
1127     size_t size;
1128     FILE *in_file;
1129     string basep, cur, next;
1130     string *ASbackup;
1131
1132     if (!(in_file = fopen(file_name, "rb"))) {
1133         printf( "Leprechaun: Can't open file %s \n", file_name );
1134         exit(-1);
1135     }
1136     fseek(in_file, 0, SEEK_END);
1137     size = ftell(in_file);
1138     fseek(in_file, 0, SEEK_SET);
1139     if (!(basep = (string) malloc(size*sizeof(char_t)))) return -1;
1140     printf( "Allocated memory for words in MB: %lu\n", ((size*sizeof(char_t))>>20)+1 );
1141     if (fread(basep, 1, size, in_file) < size) {
1142         printf( "Leprechaun: Can't read file %s \n", file_name );
1143         exit(-1);
1144     }
1145     fclose(in_file);
1146
1147     // GET nlines:
1148     cur = basep;
1149     while (cur < basep + size) {
1150         next = cur;
1151         while ((next < basep + size) && (*next != '\n')) {next++;}
1152         *--next = '\0';          // This is ala DOS i.e. windows
1153                                // 1310 not 10(\n=10)
1154         cur = next + 2;
1155         nlines++;
1156     }
1157
1158     // printf("%lu\n", (unsigned long)*lines); -> backup = *lines = 0
1159     ASbackup = (string *)malloc( nlines*sizeof(string) ); // sizeof(string) is 4
1160     if( ASbackup == NULL )
1161     { puts( "Leprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
1162     printf( "Allocated memory for pointers-to-words in MB: %lu\n", ((nlines*sizeof(string))>>20)+1 );
1163     *lines = ASbackup;
1164     //printf("%lu\n", (unsigned long)*lines); -> backup = *lines = ASbackup = 6946888
1165
1166     // Upload nlines times:
1167     nlines = 0;
1168     cur = basep;
1169     while (cur < basep + size) {
1170         next = cur;
1171         while ((next < basep + size) && (*next != '\n')) {next++;}
1172         *--next = '\0';          // This is ala DOS i.e. windows
1173                                // 1310 not 10(\n=10)
1174         ASbackup[nlines] = cur;
1175         cur = next + 2;
1176         nlines++;
1177     }
1178     return nlines;
1179 }
1180
1181 void x64toakAZE ( /* stdcall is faster and smaller... Might as well use it for the helper. */
1182     unsigned long long val,
1183     char *buf,
1184     unsigned radix,
1185     int is_neg
1186 )
1187 {
1188     char *p;          /* pointer to traverse string */
1189     char *firstdig;   /* pointer to first digit */
1190     char temp;        /* temp char */
1191     unsigned digval;  /* value of digit */
1192
1193     p = buf;
1194
1195     if ( is_neg )
1196     {
1197         *p++ = '-';    /* negative, so output '-' and negate */
1198         val = (unsigned long long)(-(long long)val);
1199     }
1200
1201     firstdig = p;      /* save pointer to first digit */
1202
1203     do {
1204         digval = (unsigned) (val % radix);
1205         val /= radix;  /* get next digit */
1206
1207         /* convert to ascii and store */
1208         if (digval > 9)
1209             *p++ = (char) (digval - 10 + 'a'); /* a letter */
1210         else
1211             *p++ = (char) (digval + '0');      /* a digit */
1212     } while (val > 0);

```

```

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

```

```

1301         int pxnman;
1302         x64toAKAZE(val, buf, radix, 0);
1303         p = buf;
1304         do {
1305             } while (*++p != '\0');
1306         p--; // p points to last digit
1307         // buf points to first digit
1308         buf[26] = 0;
1309         txpman = 1;
1310         pxnman = 0;
1311         while (buf <= p)
1312         {
1313             temp = *p;
1314             buf[26-txpman] = temp; pxnman++;
1315             p--;
1316             if (pxnman % 3 == 0 && buf <= p)
1317             {
1318                 txpman++;
1319                 buf[26-txpman] = (char) (' ');
1320             }
1321             txpman++;
1322         }
1323         return buf+26-(txpman-1);
1324 }
1325
1326 unsigned char KuxHash(char *str)
1327 {
1328     unsigned char h = 0;
1329     int max31 = 0;
1330     //while (*str)
1331     while (str[max31])
1332     {
1333         h = h ^ str[max31++];
1334         //h = h ^ *str++; // I am not sure 'str' is returned changed after return?!
1335     }
1336     return h; // 00..255 i.e. 2^8=256
1337 }
1338
1339 int KuxHash2(char *str)
1340 {
1341     int h = 0;
1342     unsigned long h2 = 0; // must be long: 31*'z'=31*122
1343     int max31 = 0;
1344     while (str[max31])
1345     {
1346         h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1347         //h2 = h2 + str[max31++]; // [113s]
1348         h2 = h2 + max31 * str[max31++];
1349     }
1350     h=h<<4; // 00..15 i.e. 2^4=16
1351     //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
1352     h = h|( h2%((1<<4)-1) );
1353     return h; // 00..4095 i.e. 2^12=4096
1354 }
1355
1356 //OSHO test - Attempts to Find/Put a WORD into linked list count: 32,011,937
1357 int KuxHash3(char *str)
1358 {
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     {
1364         h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1365         //h2 = h2 + str[max31++]; // [113s]
1366         h2 = h2 + str[max31++] * (max31+1);
1367     }
1368     // Result is: 7bits in 'h' and 32bits in 'h2'.
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 {
1383     int h = 0;
1384     unsigned long h2 = 0; // must be long: 31*'z'=31*122
1385     int max31 = 0;
1386     while (str[max31])
1387     {
1388         h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1389         //h2 = h2 + str[max31++]; // [113s]
1390         h2 = h2 + str[max31++] * (max31+1);
1391     }
1392     // Result is: 7bits in 'h' and 32bits in 'h2'.
1393     //printf("%s:\n",str);
1394     //printf("%d ",h);
1395     // a in ASCII is 097 = 0110 0001
1396     // z in ASCII is 122 = 0111 1010

```



```

1389 // Above two lines show that bits 8-7-6 are always 0-1-1 so need for low 5 bits.
1390 //h=h<<8; // 00..15 i.e. 5bits + 00-07bits=13bits
1391 //printf("%d ",h);
1392 //printf("%d ",h2);
1393 //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
1394 h = (( h<<8 )|( h2%(251) ))&8191; // 251 prime
1395 //printf("%d \n",h);
1396 return h; // 00..8191 i.e. 2^13=8192
1397 }
1398
1399 /*
1400 PUBLIC      _KuxHash3plus
1401 ; Function compile flags: /Ogty
1402 _TEXT      SEGMENT
1403 _str$ = 8                                ; size = 4
1404 _KuxHash3plus PROC NEAR
1405 ; Line 511
1406 mov ecx, DWORD PTR _str$[esp-4]
1407 mov dl, BYTE PTR [ecx]
1408 pushesi
1409 xor esi, esi
1410 xor eax, eax
1411 test dl, dl
1412 je SHORT $L1561
1413 pushebx
1414 pushedi
1415 mov edi, 1
1416 sub edi, ecx
1417 npad 8
1418 $L1560:
1419 ; Line 512
1420 movsx     edx, BYTE PTR [ecx]
1421 ; Line 514
1422 lea ebx, DWORD PTR [edi+ecx]
1423 imul ebx, edx
1424 xor esi, edx
1425 mov dl, BYTE PTR [ecx+1]
1426 add eax, ebx
1427 inc ecx
1428 test dl, dl
1429 jne SHORT $L1560
1430 pop edi
1431 pop ebx
1432 $L1561:
1433 ; Line 527
1434 xor edx, edx
1435 mov ecx, 251                                ; 000000fbh
1436 div ecx
1437 shl esi, 8
1438 mov eax, edx
1439 ; Line 529
1440 or  eax, esi
1441 and eax, 8191                                ; 00001ffffh
1442 pop esi
1443 ; Line 530
1444 ret 0
1445 _KuxHash3plus ENDP
1446 _TEXT      ENDS
1447 */
1448
1449 //OSHO test - Attempts to Find/Put a WORD into a linked list count: 32,021,975
1450 int KuxHash4(char *str)
1451 {
1452     int h2 = 0;
1453     for (; *str != 0; str++) {
1454         //h2 = (127*h2 + *str) % (8192-1); // 2^13-1 = 8191 is Mersenne prime
1455         h2 = ((h2<<7) + *str) % (8192-1); // 2^13-1 = 8191 is Mersenne prime
1456     }
1457
1458     return h2; // 00..8191 i.e. 2^13=8192
1459 }
1460
1461 /*
1462 int hash(char *v, int M)
1463 { int h = 0, a = 127;
1464   for (; *v != 0; v++)
1465       h = (a*h + *v) % M;
1466   return h;
1467 }
1468
1469 int hashU(char *v, int M)
1470 { int h, a = 31415, b = 27183;
1471   for (h = 0; *v != 0; v++, a = a*b % (M-1))
1472       h = (a*h + *v) % M;
1473   return (h < 0) ? (h + M) : h;
1474 }
1475 */
1476

```

```

1477 // Kaze: My appreciation of FNV is far beyond C code optimization, it is alchemical, and why not, magical.
1478
1479 /*
1480 FNV hash history
1481     The basis of the FNV hash algorithm was taken from an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee
1482 by Glenn Fowler and Phong Vo back in 1991. In a subsequent ballot round: Landon Curt Noll improved on their algorithm.
1483 Some people tried this hash and found that it worked rather well. In an EMail message to Landon, they named it
1484 the 'Fowler/Noll/vo'' or FNV hash.
1485     FNV hashes are designed to be fast while maintaining a low collision rate. The FNV speed allows one to quickly hash
1486 lots of data while maintaining a reasonable collision rate. The high dispersion of the FNV hashes makes them well suited
1487 for hashing nearly identical strings such as URLs, hostnames, filenames, text, IP addresses, etc.
1488 */
1489
1490 /* NOTE: u_int64_t is a 64 bit unsigned type */
1491 /* NOTE: u_int32_t is a 32 bit unsigned type */
1492 /* NOTE: u_int16_t is a 16 bit unsigned type */
1493 /* NOTE: u_int8_t is a 8 bit unsigned type */
1494
1495 //typedef unsigned char u_int8_t; //FNV only
1496 //typedef unsigned long u_int32_t; //FNV only
1497 //typedef unsigned long long u_int64_t; //FNV only
1498
1499 // 32 bit FNV_prime = 2^24 + 2^8 + 0x93 = 16777619
1500 // 64 bit FNV_prime = 2^40 + 2^8 + 0xb3 = 1099511628211
1501
1502 // 32 bit offset_basis = 2166136261
1503 // 64 bit offset_basis = 14695981039346656037
1504
1505 #define FNV1_64_INIT ((u_int64_t)14695981039346656037)
1506 #define FNV1_64_PRIME ((u_int64_t)1099511628211)
1507 #define FNV1_32_INIT ((u_int32_t)2166136261)
1508 #define FNV1_32_PRIME ((u_int32_t)602173697)
1509 // FNV1A_Hash_4_OCTETS gives dispersion as follows:
1510 //3549448: 1607
1511 //3549669: 171072511
1512 //3550710: 272961023
1513 //3550733: 172783361
1514 //3550734: 431562497
1515 //3550929: 204312319
1516 //3550984: 562853633
1517 //3550991: 551362303
1518 //3551359: 332820737
1519 //3551484: 354126079
1520 //3551514: 407138561
1521 //3551523: 442058753
1522 //3551701: 449230849
1523 //3551736: 107712257
1524 //3551961: 428904191
1525 //3552039: 602173697
1526 //3552103: 588411137
1527
1528 #define FNV_64A_OP(hash, octet) \
1529     (((u_int64_t)(hash) ^ (u_int8_t)(octet)) * FNV1_64_PRIME)
1530
1531 #define FNV_64A_OP64(hash, octet) \
1532     (((u_int64_t)(hash) ^ (u_int64_t)(octet)) * FNV1_64_PRIME)
1533
1534 #define FNV_32A_OP_GENERIC(hash, octet) \
1535     (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * 16777619)
1536
1537 #define FNV_32A_OP(hash, octet) \
1538     (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * FNV1_32_PRIME)
1539
1540 #define FNV_32A_OP_MULless_core(hash, octet) \
1541     ( (u_int32_t)(hash) ^ (u_int8_t)(octet) )
1542
1543 #define FNV_32A_OP_MULless(hash, octet) \
1544     ( (FNV_32A_OP_MULless_core(hash, octet)<<5) - FNV_32A_OP_MULless_core(hash, octet) )
1545
1546 #define FNV_32A_OP32(hash, octet) \
1547     (((u_int32_t)(hash) ^ (u_int32_t)(octet)) * FNV1_32_PRIME)
1548
1549 #define FNV_32A_OP64(hash, octet) \
1550     (((u_int64_t)(hash) ^ (u_int64_t)(octet)) * FNV1_32_PRIME)
1551
1552 #define FNV_32A_OP32_MULless_core(hash, octet) \
1553     ( (u_int32_t)(hash) ^ (u_int32_t)(octet) )
1554
1555 #define FNV_32A_OP32_MULless(hash, octet) \
1556     ( (FNV_32A_OP32_MULless_core(hash, octet)<<5) - FNV_32A_OP32_MULless_core(hash, octet) )
1557
1558
1559 // Invoking: FNV1A_Hash_4_OCTETS_31(wrd, wrdlen>>2) // = 0,1,2,3,4,5,6,7 [1..31]
1560 int FNV1A_Hash_4_OCTETS_31(char *str, int wrdlen_QUADRUPLTS)
1561 {
1562     u_int32_t hash;
1563     char *p;
1564

```

```

1565 hash = FNV1_32_INIT;
1566 p=str;
1567
1568 // The goal of stage #1: to reduce number of 'imul's in fact to reduce loops.
1569
1570 // Stage #1:
1571 for (; wrdlen_QUADRUPLTS != 0; --wrdlen_QUADRUPLTS) {
1572     hash = FNV_32A_OP32_MULless(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
1573     p=p+4; // add eax, 4
1574 }
1575
1576 // Stage #2:
1577 for (; *p; ++p) {
1578     hash = FNV_32A_OP_MULless(hash, *p); // mov dl, BYTE PTR [ecx]
1579 }
1580
1581 //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1582 return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1583 }
1584
1585
1586 // Invoking: FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2) // = 0,1,2,3,4,5,6,7 [1..31]
1587 int FNV1A_Hash_4_OCTETS(char *str, int wrdlen_QUADRUPLTS)
1588 {
1589     u_int32_t hash;
1590     char *p;
1591
1592     hash = FNV1_32_INIT;
1593     p=str;
1594
1595     // The goal of stage #1: to reduce number of 'imul's.
1596
1597     // Stage #1:
1598     for (; wrdlen_QUADRUPLTS != 0; --wrdlen_QUADRUPLTS) {
1599         hash = FNV_32A_OP32(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
1600         p=p+4; // add eax, 4
1601     }
1602
1603     // Stage #2:
1604     for (; *p; ++p) {
1605         hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [ecx]
1606     }
1607
1608     //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1609     return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1610 }
1611
1612 /*
1613 Results for 'FNV1A_Hash_8_OCTETS':
1614 Bytes per second performance: 23,110,160B/s
1615 words per second performance: 1,959,516W/s
1616 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
1617 Size of all TEXTual Files: 415,982,896
1618 Word count: 35,271,297 of them 22,202,980 distinct
1619 Number Of Files: 8
1620 Number Of Lines: 35271297
1621 Allocated memory in MB: 1950
1622 Number Of Trees(GREATER THE BETTER): 3419429
1623 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 51%
1624 Number of Hash Collisions(Distinct WORDs - Number Of Trees): 18783551
1625 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '1,119'
1626 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 268,085,505
1627 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 7,690,615
1628 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 2,622 must have PEAK = 12 = rounding down of integer (1+lb(2,622))
1629 Binary-Search-Tree(1st out of 1) with MaxNODEs = 2,622 has PEAK = 592 and LEAFs = 689
1630 Binary-Search-Tree(1st out of 1) with MaxPEAK = '1,119' has NODEs = 1,537 and LEAFs = 287
1631 Binary-Search-Tree(1st out of 1) with MaxLEAFs = 731 has NODEs = 2,517 and PEAK = 448
1632 */
1633 // Invoking: FNV1A_Hash_8_OCTETS(wrd, wrdlen>>3) // = 0,1,2,3 [1..31]
1634 int FNV1A_Hash_8_OCTETS(char *str, int wrdlen_OCTETS)
1635 {
1636     u_int32_t hash;
1637     char *p;
1638
1639     hash = FNV1_32_INIT;
1640     p=str;
1641
1642     // The goal of stage #1: to reduce number of 'imul's.
1643
1644     // Stage #1:
1645     for (; wrdlen_OCTETS != 0; --wrdlen_OCTETS) {
1646         hash = FNV_32A_OP64(hash, (unsigned long long)*(long *)p); // mov edi, DWORD PTR [eax]
1647         p=p+8; // add eax, 4
1648     }
1649
1650     // Stage #2:
1651     for (; *p; ++p) {
1652         hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [ecx]

```

```

1653 }
1654
1655 //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1656 return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1657 }
1658
1659
1660 // Invoking: FNV1A_Hash_Granularity(wrd, wrdlen>>0|2|3, 0|2|3)
1661 int FNV1A_Hash_Granularity(char *str, int wrdlen_granulated, int Granularity) // wrdlen>>0=wrdlen
1662 {
1663     u_int32_t hash;
1664     u_int64_t hash64;
1665     char *p;
1666
1667     hash = FNV1_32_INIT;
1668     p=str;
1669
1670     // The goal of stage #1: to reduce number of 'imul's and mainly: the number of loops.
1671
1672     // Stage #1:
1673     if (Granularity == 2) {
1674         for (; wrdlen_granulated != 0; --wrdlen_granulated) {
1675             hash = FNV_32A_OP32(hash, (u_int32_t)*(u_int32_t *)p);
1676             p=p+4; // (1<<Granularity): 1<<0=1, 1<<2=4, 1<<3=8
1677         }
1678     }
1679     if (Granularity == 3) {
1680         hash64 = FNV1_64_INIT;
1681         for (; wrdlen_granulated != 0; --wrdlen_granulated) {
1682             hash64 = FNV_64A_OP64(hash64, (u_int64_t)*(u_int64_t *)p);
1683             p=p+8; // (1<<Granularity): 1<<0=1, 1<<2=4, 1<<3=8
1684         }
1685         for (; *p; ++p) {
1686             hash64 = FNV_64A_OP(hash64, (u_int8_t)*(u_int8_t *)p);
1687         }
1688
1689         //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1690         return ((hash64>>51) ^ hash64) & 8191; // 00..8191 i.e. 2^13=8192
1691         // probably better shifting is not by 16 bits but ...
1692         //hash64>>16: 3,544,160 just bad
1693         //hash64>>33: 3,547,854
1694         //hash64>>34: 3,547,266
1695         //hash64>>35: 3,547,453
1696         //hash64>>36: 3,547,242
1697         //hash64>>40: 3,548,263
1698         //hash64>>44: 3,548,242
1699         //hash64>>45: 3,549,056
1700         //hash64>>46: 3,549,207
1701         //hash64>>47: 3,549,094
1702         //hash64>>50: 3,549,392
1703         //hash64>>51: 3,549,395 i.e. maximum shift: the 13 most significant bits i.e. (64-13); closest to 3,549,448
1704
1705         // Above results are obtained for following set:
1706         //if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
1707         //    slot = FNV1A_Hash_Granularity(wrd, wrdlen>>2, 2)<<2; //13++++
1708         //else // 2x8+4=20 i.e. first contains 6 clashes
1709         //    slot = FNV1A_Hash_Granularity(wrd, wrdlen>>3, 3)<<2; //13++++
1710     }
1711
1712     //if (Granularity != 3) {
1713     // Stage #2:
1714     for (; *p; ++p) {
1715         hash = FNV_32A_OP(hash, (u_int8_t)*(u_int8_t *)p);
1716     }
1717
1718     //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1719     return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1720     //}
1721 }
1722
1723
1724 // char *string;          /* the string to 64 bit FNV-1a hash */
1725 // u_int64_t hash;        /* will hold the final value of the hash */
1726 // char *p;
1727 //
1728 // hash = FNV1_64_INIT;
1729 // for (p=string; *p; ++p) {
1730 //     hash = FNV_64A_OP(hash, *p);
1731 // }
1732
1733
1734 // If you need an x-bit hash where x is not a power of 2,
1735 // 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.
1736 // By xor-folding we mean shift the excess high order bits down and xor them with the lower x-bits.
1737 // For tiny x < 16 bit values, we recommend using a 32 bit FNV-1 hash as follows:
1738
1739 // /* NOTE: for 0 < x < 16 ONLY!!! */
1740 // #define TINY_MASK(x) (((u_int32_t)1<<(x))-1)

```

```

1741 // #define FNV1_32_INIT ((u_int32_t)2166136261)
1742 // u_int32_t hash;
1743 // void *data;
1744 // size_t data_len;
1745 //
1746 // hash = fnv_32_buf(data, data_len, FNV1_32_INIT);
1747 // hash = (((hash>>x) ^ hash) & TINY_MASK(x));
1748
1749
1750 int FNV1A_Hash_SHIFTless_XORless(char *str)
1751 {
1752     u_int32_t hash; /* will hold the final value of the hash */
1753     char *p;
1754
1755     hash = FNV1_32_INIT;
1756     for (p=str; *p; ++p) {
1757         hash = FNV_32A_OP(hash, *p);
1758     }
1759     //hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1760
1761     return hash & 8191; // 00..8191 i.e. 2^13=8192
1762 }
1763
1764 /*
1765 _FNV1A_Hash_SHIFTless_XORless PROC NEAR
1766 ; Line 721
1767 mov     edx, DWORD PTR _str$[esp-4]
1768 mov     cl, BYTE PTR [edx]
1769 test    cl, cl
1770 mov     eax, -2128831035 ; 811c9dc5H
1771 je      SHORT $L1582
1772 npad1
1773 $L1580:
1774 ; Line 722
1775 movzx   ecx, cl
1776 xor     ecx, eax
1777 imul    ecx, 16777619 ; 01000193H
1778 inc     edx
1779 mov     eax, ecx
1780 mov     cl, BYTE PTR [edx]
1781 test    cl, cl
1782 jne     SHORT $L1580
1783 $L1582:
1784 ; Line 726
1785 and     eax, 8191 ; 00001ffffH
1786 ; Line 727
1787 ret     0
1788 _FNV1A_Hash_SHIFTless_XORless ENDP
1789 */
1790
1791
1792 int FNV1A_Hash(char *str)
1793 {
1794     u_int32_t hash; /* will hold the final value of the hash */
1795     char *p;
1796
1797     hash = FNV1_32_INIT;
1798     for (p=str; *p; ++p) {
1799         hash = FNV_32A_OP(hash, *p);
1800     }
1801     //hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1802
1803     return ((hash>>13) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1804 }
1805
1806 /*
1807 _FNV1A_Hash PROC NEAR
1808 ; Line 722
1809 mov     edx, DWORD PTR _str$[esp-4]
1810 mov     al, BYTE PTR [edx]
1811 test    al, al
1812 mov     ecx, -2128831035 ; 811c9dc5H
1813 je      SHORT $L1582
1814 npad1
1815 $L1580:
1816 ; Line 723
1817 movzx   eax, al
1818 xor     eax, ecx
1819 imul    eax, 16777619 ; 01000193H
1820 inc     edx
1821 mov     ecx, eax
1822 mov     al, BYTE PTR [edx]
1823 test    al, al
1824 jne     SHORT $L1580
1825 $L1582:
1826 ; Line 727
1827 mov     eax, ecx
1828 shr     eax, 13 ; 0000000dH

```

```

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

```

```

1917     dec     ecx
1918     jnz     next
1919     pop     edi
1920     pop     esi
1921     pop     ebx
1922     retn    0ch
1923
1924     fast_fnv32a: ;FNV1a 32bit (size: 36 bytes)
1925     ;         Intel Core 2 Duo E6600: 574.95 mb/s
1926     push    ebx
1927     push    esi
1928     push    edi
1929     mov     esi, [esp + 10h] ;buffer
1930     mov     ecx, [esp + 14h] ;length
1931     mov     eax, [esp + 18h] ;basis
1932     mov     edi, 01000193h ;fnv_32_prime
1933     xor     ebx, ebx
1934     nexta:
1935     mov     bl, [esi]
1936     xor     eax, ebx
1937     mul     edi
1938     inc     esi
1939     dec     ecx
1940     jnz     nexta
1941     pop     edi
1942     pop     esi
1943     pop     ebx
1944     retn    0ch
1945 */
1946
1947 //Number Of Trees(GREATER THE BETTER): 3525737
1948 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1949 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18677243
1950 int Hash17_unrolled(const char *key, int wrdlen)
1951 {
1952     int hash = 1;
1953     int i;
1954     for(i = 0; i < (wrdlen & -2); i += 2) {
1955         hash = (17) * hash + (key[i] - ' ');
1956         hash = (17) * hash + (key[i+1] - 'i');
1957     }
1958     if(wrdlen & 1)
1959         hash = (17) * hash + (key[wrdlen-1] - ' ');
1960     return ( hash ^ (hash >> 16) ) & 8191;
1961 }
1962
1963 //hash = 1:
1964 //Number Of Trees(GREATER THE BETTER): 3556516
1965 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1966 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18646464
1967 //hash = 13:
1968 //Number Of Trees(GREATER THE BETTER): 3556755
1969 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1970 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18646225
1971 //hash = 11:
1972 //Number Of Trees(GREATER THE BETTER): 3557011
1973 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1974 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18645969
1975 //hash = 7:
1976 //Number Of Trees(GREATER THE BETTER): 3557181
1977 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1978 //Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18645799
1979 int Alfalfa(const char *key, int wrdlen)
1980 {
1981     int hash = 7;
1982     int i;
1983     for(i = 0; i < (wrdlen & -2); i += 2) {
1984         hash = (17+9) * ((17+9) * hash + (key[i])) + (key[i+1]);
1985     }
1986     if(wrdlen & 1)
1987         hash = (17+9) * hash + (key[wrdlen-1]);
1988     return ( hash ^ (hash >> 16) ) & 8191;
1989 }
1990
1991 /*
1992 [FNV1A 'shift-less-&-xor-less' hash used in Leprechaun r.13++:]
1993
1994 int FNV1A_Hash_SHIFTless_XORless(char *str)
1995 {
1996     u_int32_t hash;
1997     char *p;
1998
1999     hash = FNV1_32_INIT;
2000     for (p=str; *p; ++p) {
2001         hash = FNV_32A_OP(hash, *p);
2002     }
2003     //hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
2004

```

```

2005 return hash & 8191; // 00..8191 i.e. 2^13=8192
2006 }
2007
2008 words per second performance: 837,458w/s
2009 Input File with a list of TEXTual Files: wikipedia-en-html.tar.wrd.lst
2010 word count: 12,561,874 of them 12,561,874 distinct
2011 Number Of Trees(GREATER THE BETTER): 2772875
2012 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 41%
2013 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 9788999
2014
2015 words per second performance: 1,007,751w/s
2016 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2017 word count: 35,271,297 of them 22,202,980 distinct
2018 Number Of Trees(GREATER THE BETTER): 3537061
2019 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2020 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18665919
2021
2022 [My '2in1' hash used in Leprechaun r.13++:]
2023
2024 int KuxHash3plus(char *str)
2025 { int h = 0;
2026   unsigned long h2 = 0; // must be long: 31*'z'=31*122
2027   int max31 = 0;
2028   while (str[max31])
2029   { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
2030     //h2 = h2 + str[max31++]; // [113s]
2031     h2 = h2 + str[max31++] * (max31+1);
2032   }
2033   // Result is: 7bits in 'h' and 32bits in 'h2'.
2034
2035   //printf("%s:\n ",str);
2036   //printf("%d ",h);
2037   // a in ASCII is 097 = 0110 0001
2038   // z in ASCII is 122 = 0111 1010
2039   // Above two lines show that bits 8-7-6 are always 0-1-1 so need for low 5 bits.
2040   //h=h<<8; // 00..15 i.e. 5bits + 00-07bits=13bits
2041   //printf("%d ",h);
2042   //printf("%d ",h2);
2043   //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
2044   h = (( h<<8 )|( h2%(251) ))&8191; // 251 prime
2045   //printf("%d\n",h);
2046   return h; // 00..8191 i.e. 2^13=8192
2047 }
2048
2049 words per second performance: 785,117w/s
2050 Input File with a list of TEXTual Files: wikipedia-en-html.tar.wrd.lst
2051 word count: 12,561,874 of them 12,561,874 distinct
2052 Number Of Trees(GREATER THE BETTER): 2663566
2053 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 40%
2054 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 9898308
2055
2056 words per second performance: 979,758w/s
2057 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2058 word count: 35,271,297 of them 22,202,980 distinct
2059 Number Of Trees(GREATER THE BETTER): 3410463
2060 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 51%
2061 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18792517
2062
2063 [Last standing for English(en)-Wikipedia's wordlist:]
2064 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%
2065 chongo's hash has better distribution than Kaze's hash by ((9898308 - 9788999)/9788999)*100% = 1.1%
2066
2067 [Last standing for LATIN(de,en,es,fr,it,nl,pt,ro)-Wikipedia's wordlist:]
2068 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%
2069 chongo's hash has better distribution than Kaze's hash by ((18792517 - 18665919)/18665919)*100% = 0.6%
2070
2071 Bottomline is:
2072 Your hash thrash, my hash for trash, he-he.
2073 Thanks a lot, again, Mr. Noll.
2074
2075 Yummy little file: http://www.isthe.com/chongo/src/fnv/fnv-5.0.2.tar.gz
2076 */
2077
2078 /*
2079 // Paul Larson (http://research.microsoft.com/~PALARSON/)
2080 UINT HashLarson(const CHAR *key, SIZE_T len) {
2081   UINT hash = 0;
2082   for(UINT i = 0; i < len; ++i)
2083     hash = 101 * hash + key[i];
2084   return hash ^ (hash >> 16);
2085 }
2086
2087 // Kernighan & Ritchie, "The C programming Language", 3rd edition.
2088 UINT HashKernighanRitchie(const CHAR *key, SIZE_T len) {
2089   UINT hash = 0;
2090   for(UINT i = 0; i < len; ++i)
2091     hash = 31 * hash + key[i];
2092   return hash;

```



```

2093 }
2094
2095 // A hash function with multiplier 65599 (from Red Dragon book)
2096 UINT Hash65599(const CHAR *key, SIZE_T len) {
2097     UINT hash = 0;
2098     for(UINT i = 0; i < len; ++i)
2099         hash = 65599 * hash + key[i];
2100     return hash ^ (hash >> 16);
2101 }
2102
2103 // FNV hash, http://isthe.com/chongo/tech/comp/fnv/
2104 UINT HashFNV1a(const CHAR *key, SIZE_T len) {
2105     UINT hash = 2166136261;
2106     for(UINT i = 0; i < len; ++i)
2107         hash = 16777619 * (hash ^ key[i]);
2108     return hash ^ (hash >> 16);
2109 }
2110
2111 // Ramakrishna hash
2112 UINT HashRamakrishna(const CHAR *key, SIZE_T len) {
2113     UINT h = 0;
2114     for(UINT i = 0; i < len; ++i) {
2115         h ^= (h << 5) + (h >> 2) + key[i];
2116     }
2117     return h;
2118 }
2119 */
2120
2121 /*
2122 Results for 'Hash_Alfalfa':
2123 Bytes per second performance: 19,808,709B/s
2124 Words per second performance: 1,679,585W/s
2125 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2126 Size of all TEXTual Files: 415,982,896
2127 Word count: 35,271,297 of them 22,202,980 distinct
2128 Number Of Files: 8
2129 Number Of Lines: 35271297
2130 Allocated memory in MB: 1950
2131 Number Of Trees(GREATER THE BETTER): 3549079
2132 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2133 Number Of Hash Collisions(Distinct WORDS - Number Of Trees): 18653901
2134 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '37'
2135 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 117,063,824
2136 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,279
2137 Perfectly-Balanced-Binary-Search-Tree for MaxNODES = 84 must have PEAK = 7 = rounding down of integer (1+lb(84))
2138 Binary-Search-Tree(1st out of 2) with MaxNODES = 84 has PEAK = 20 and LEAFs = 24
2139 Binary-Search-Tree(1st out of 3) with MaxPEAK = '37' has NODES = 67 and LEAFs = 17
2140 Binary-Search-Tree(1st out of 1) with MaxLEAFs = 28 has NODES = 78 and PEAK = 22
2141 */
2142 UINT Hash_Alfalfa(const char *key, unsigned int wrdlen)
2143 {
2144     UINT hash = 7;
2145     unsigned int i;
2146     for (i = 0; i < (wrdlen & -2); i += 2) {
2147         hash = (53) * ((53) * hash + (key[i])) + (key[i+1]);
2148     }
2149     if (wrdlen & 1)
2150         hash = (53) * hash + (key[wrdlen-1]);
2151     return ((hash>>16) ^ hash) & 8191;
2152 }
2153
2154 /*
2155 Results for 'HashAlfalfa_HALF':
2156 Bytes per second performance: 19,808,709B/s
2157 Words per second performance: 1,679,585W/s
2158 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2159 Size of all TEXTual Files: 415,982,896
2160 Word count: 35,271,297 of them 22,202,980 distinct
2161 Number Of Files: 8
2162 Number Of Lines: 35271297
2163 Allocated memory in MB: 1950
2164 Number Of Trees(GREATER THE BETTER): 3550665
2165 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2166 Number Of Hash Collisions(Distinct WORDS - Number Of Trees): 18652315
2167 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '39'
2168 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 117,053,918
2169 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,259
2170 Perfectly-Balanced-Binary-Search-Tree for MaxNODES = 87 must have PEAK = 7 = rounding down of integer (1+lb(87))
2171 Binary-Search-Tree(1st out of 1) with MaxNODES = 87 has PEAK = 21 and LEAFs = 27
2172 Binary-Search-Tree(1st out of 2) with MaxPEAK = '39' has NODES = 65 and LEAFs = 18
2173 Binary-Search-Tree(1st out of 4) with MaxLEAFs = 27 has NODES = 77 and PEAK = 23
2174 */
2175 UINT HashAlfalfa_HALF(const char *key, unsigned int wrdlen)
2176 {
2177     UINT hash = 12;
2178     UINT hashBUFFER;
2179     unsigned int i,j;
2180     for(i = 0; i < (wrdlen & -4); i += 4) {

```

```

2181 //hash = (( (hash<<5)-hash) + key[i] )<<5) - ( (hash<<5)-hash) + key[i] ) + (key[i+1]);
2182 hashBUFFER = ((hash<<5)-hash) + key[i];
2183 hash = (( hashBUFFER )<<5) - ( hashBUFFER ) + (key[i+1]);
2184 //hash = (( (hash<<5)-hash) + key[i+2] )<<5) - ( (hash<<5)-hash) + key[i+2] ) + (key[i+3]);
2185 hashBUFFER = ((hash<<5)-hash) + key[i+2];
2186 hash = (( hashBUFFER )<<5) - ( hashBUFFER ) + (key[i+3]);
2187 }
2188 for(j = 0; j < (wrklen & 3); j += 1) {
2189     hash = ((hash<<5)-hash) + key[i+j];
2190 }
2191 return ((hash>>16) ^ hash) & 8191;
2192 }
2193
2194 /*
2195 Results for 'HashFNV1A_unrolled_Final':
2196 Bytes per second performance: 19,808,709B/s
2197 words per second performance: 1,679,585w/s
2198 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2199 Size of all TEXTual Files: 415,982,896
2200 word count: 35,271,297 of them 22,202,980 distinct
2201 Number Of Files: 8
2202 Number Of Lines: 35271297
2203 Allocated memory in MB: 1950
2204 Number Of Trees(GREATER THE BETTER): 3445337
2205 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 52%
2206 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18757643
2207 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '43'
2208 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 118,349,998
2209 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 7,997,033
2210 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 89 must have PEAK = 7 = rounding down of integer (1+lb(89))
2211 Binary-Search-Tree(1st out of 1) with MaxNODEs = 89 has PEAK = 28 and LEAFs = 28
2212 Binary-Search-Tree(1st out of 1) with MaxPEAK = '43' has NODEs = 65 and LEAFs = 11
2213 Binary-Search-Tree(1st out of 2) with MaxLEAFs = 28 has NODEs = 78 and PEAK = 24
2214 */
2215 UINT HashFNV1A_unrolled_Final(char *str, unsigned int wrklen)
2216 {
2217     //const UINT PRIME = 31;
2218     unsigned int hash = 2166136261;
2219     char * p = str;
2220
2221     /*
2222     // Reduce the number of multiplications by unrolling the loop
2223     for (SIZE_T ndwords = wrklen / sizeof(DWORD); ndwords; --ndwords) {
2224         //hash = (hash ^ *(DWORD*)p) * PRIME;
2225         hash = ((hash ^ *(DWORD*)p)<<5) - (hash ^ *(DWORD*)p);
2226
2227         p += sizeof(DWORD);
2228     }
2229     */
2230     for(; wrklen >= 4; wrklen -= 4, p += 4) {
2231         hash = ((hash ^ *(unsigned int*)p)<<5) - (hash ^ *(unsigned int*)p);
2232     }
2233
2234     // Process the remaining bytes
2235     /*
2236     for (SIZE_T i = 0; i < (wrklen & (sizeof(DWORD) - 1)); i++) {
2237         //hash = (hash ^ *p++) * PRIME;
2238         hash = ((hash ^ *p)<<5) - (hash ^ *p);
2239         p++;
2240     }
2241     */
2242     if (wrklen & -2) {
2243         hash = ((hash ^ (*(unsigned int*)p&0xFFFF))<<5) - (hash ^ (*(unsigned int*)p&0xFFFF));
2244         p++;p++;
2245     }
2246     if (wrklen & 1)
2247         hash = ((hash ^ *p)<<5) - (hash ^ *p);
2248
2249     return ((hash>>16) ^ hash) & 8191;
2250 }
2251
2252 /*
2253 Results for 'Sixtinsensitive':
2254 Bytes per second performance: 19,808,709B/s
2255 words per second performance: 1,679,585w/s
2256 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
2257 Size of all TEXTual Files: 415,982,896
2258 word count: 35,271,297 of them 22,202,980 distinct
2259 Number Of Files: 8
2260 Number Of Lines: 35271297
2261 Allocated memory in MB: 1950
2262 Number Of Trees(GREATER THE BETTER): 3531949
2263 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2264 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18671031
2265 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '38'
2266 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 118,959,016
2267 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,047,983
2268 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 98 must have PEAK = 7 = rounding down of integer (1+lb(98))

```

```

2269 Binary-Search-Tree(1st out of 1) with MaxNODEs = 98 has PEAK = 36 and LEAFs = 30
2270 Binary-Search-Tree(1st out of 1) with MaxPEAK = '38' has NODEs = 54 and LEAFs = 11
2271 Binary-Search-Tree(1st out of 2) with MaxLEAFs = 30 has NODEs = 98 and PEAK = 36
2272 */
2273 // Tuned for lowercase-and-uppercase letters i.e. 26 ASCII symbols 65-90 and 97-122 decimal.
2274 UINT Sixtinsensitive(const char *str, unsigned int wrdlen)
2275 {
2276     UINT hash = 2166136261;
2277     UINT hashBUFFER_EAX, hashBUFFER_BH, hashBUFFER_BL;
2278     const char * p = str;
2279
2280     // 0x41 = 065 'A' 010 [0 0001]
2281     // 0x5A = 090 'Z' 010 [1 1010]
2282     // 0x61 = 097 'a' 011 [0 0001]
2283     // 0x7A = 122 'z' 011 [1 1010]
2284
2285     // Reduce the number of multiplications by unrolling the loop
2286     for(; wrdlen >= 6; wrdlen -= 6, p += 6) {
2287         //hashBUFFER_EAX = (*(DWORD*)(p+0)&0xFFFF);
2288         hashBUFFER_EAX = (*(DWORD*)(p+0)&0x1F1F1F1F);
2289         hashBUFFER_BL = (*(p+4)&0x1F);
2290         hashBUFFER_BH = (*(p+5)&0x1F);
2291         //6bytes-in-4bytes or 48bits-to-30bits
2292         // Two times next:
2293         //3bytes-in-2bytes or 24bits-to-15bits
2294         //EAX          BL          BH
2295         //[5bit][3bit][5bit][3bit][5bit][3bit][5bit][3bit]
2296         //      5th[0..15] 13th[0..15]
2297         //      BL lower 3  BL higher 2bits
2298         // OR or XOR no difference
2299         hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BL&0x07)<<5); // BL lower 3bits of 5bits
2300         hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BL&0x18)<<(2+8)); // BL higher 2bits of 5bits
2301         hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BH&0x07)<<(5+16)); // BH lower 3bits of 5bits
2302         hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BH&0x18)<<((2+8)+16)); // BH higher 2bits of 5bits
2303         //hash = (hash ^ hashBUFFER_EAX)*1607; //what a mess: <<7 becomes imul but <<5 not!
2304         hash = ((hash ^ hashBUFFER_EAX)<<5) - (hash ^ hashBUFFER_EAX);
2305         //1607:[2118599]
2306         // 127:[2121081]
2307         // 31:[2139242]
2308         // 17:[2150803]
2309         // 7:[2166336]
2310         // 5:[2183044]
2311         //8191:[2200477]
2312         // 3:[2205095]
2313         // 257:[2206188]
2314     }
2315     // Post-Variant #1:
2316     for(; wrdlen; wrdlen--, p++) {
2317         hash = ((hash ^ (*p&0x1F))<<5) - (hash ^ (*p&0x1F));
2318     }
2319     /*
2320     // Post-Variant #2:
2321     for(; wrdlen >= 2; wrdlen -= 2, p += 2) {
2322         hash = ((hash ^ (*(DWORD*)p&0xFFFF))<<5) - (hash ^ (*(DWORD*)p&0xFFFF));
2323     }
2324     if (wrdlen & 1)
2325         hash = ((hash ^ *p)<<5) - (hash ^ *p);
2326     */
2327     /*
2328     // Post-Variant #3:
2329     for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
2330         hash = ((hash ^ *(DWORD*)p)<<5) - (hash ^ *(DWORD*)p);
2331     }
2332     if (wrdlen & 2) {
2333         hash = ((hash ^ *(DWORD*)p&0xFFFF)<<5) - (hash ^ *(DWORD*)p&0xFFFF);
2334         p++;p++;
2335     }
2336     if (wrdlen & 1)
2337         hash = ((hash ^ *p)<<5) - (hash ^ *p);
2338     */
2339     return ((hash>>16) ^ hash) & 8191;
2340 }
2341
2342 /*
2343 #define FNV1_32_INIT ((UINT)2166136261)
2344 #define FNV1_32_PRIME ((UINT)1709)
2345
2346 #define FNV_32A_OP(hash, octet) \
2347     (((UINT)(hash) ^ (unsigned char)(octet)) * FNV1_32_PRIME)
2348
2349 #define FNV_32A_OP32(hash, octet) \
2350     (((UINT)(hash) ^ (UINT)(octet)) * FNV1_32_PRIME)
2351
2352 UINT FNV1A_Hash_WHIZ(const char *str, SIZE_T wrdlen)
2353 {
2354
2355     UINT hash32;
2356     const char *p;

```

```

2357
2358 hash32 = FNV1_32_INIT;
2359 p=str;
2360
2361 for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
2362 hash32 = FNV_32A_OP32(hash32, (UINT)*(UINT *)p);
2363 }
2364 if (wrdlen & -2) {
2365     hash32 = FNV_32A_OP32(hash32, *(UINT*)p&0xFFFF);
2366     p++;p++;
2367 }
2368 if (wrdlen & 1)
2369     hash32 = FNV_32A_OP(hash32, *p);
2370
2371 return hash32 ^ (hash32 >> 16);
2372 }
2373 */
2374
2375 /*
2376 Results for 'FNV1A_Hash_Jester':
2377 Bytes per second performance: 19,808,709B/s
2378 Words per second performance: 1,679,585W/s
2379 Input File with a list of TEXTual Files: Leprechaun_vs_wikipedia_LATIN-WORDS.lst
2380 Size of all TEXTual Files: 415,982,896
2381 Word count: 35,271,297 of them 22,202,980 distinct
2382 Number Of Files: 8
2383 Number Of Lines: 35271297
2384 Allocated memory in MB: 1950
2385 Number Of Trees(GREATER THE BETTER): 3537352
2386 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2387 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18665628
2388 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '37'
2389 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 117,243,563
2390 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,063,361
2391 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 87 must have PEAK = 7 = rounding down of integer (1+lb(87))
2392 Binary-Search-Tree(1st out of 2) with MaxNODEs = 87 has PEAK = 27 and LEAFs = 23
2393 Binary-Search-Tree(1st out of 1) with MaxPEAK = '37' has NODEs = 66 and LEAFs = 18
2394 Binary-Search-Tree(1st out of 3) with MaxLEAFs = 27 has NODEs = 84 and PEAK = 27
2395 */
2396 UINT FNV1A_Hash_Jester(const char *str, unsigned int wrdlen)
2397 {
2398     const UINT PRIME = 709607;
2399     UINT hash32 = 2166136261;
2400     const char *p = str;
2401
2402     // Idea comes from Igor Pavlov's 7zCRC, thanks.
2403     /*
2404     for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
2405         hash32 = (hash32 ^ *p) * PRIME;
2406     }
2407     */
2408     for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
2409         hash32 = (hash32 ^ *(DWORD *)p) * PRIME;
2410         hash32 = (hash32 ^ *(DWORD *)p) * PRIME;
2411     }
2412     // Cases: 0,1,2,3,4,5,6,7
2413     if (wrdlen & sizeof(DWORD)) {
2414         hash32 = (hash32 ^ *(DWORD *)p) * PRIME;
2415         p += sizeof(DWORD);
2416     }
2417     if (wrdlen & sizeof(WORD)) {
2418         hash32 = (hash32 ^ *(WORD *)p) * PRIME;
2419         p += sizeof(WORD);
2420     }
2421     if (wrdlen & 1)
2422         hash32 = (hash32 ^ *p) * PRIME;
2423
2424     return (hash32 ^ (hash32 >> 16)) & 8191;
2425 }
2426
2427 /*
2428 Results for 'FNV1A_Hash_Jesteress':
2429 Bytes per second performance: 19,808,709B/s
2430 Words per second performance: 1,679,585W/s
2431 Input File with a list of TEXTual Files: Leprechaun_vs_wikipedia_LATIN-WORDS.lst
2432 Size of all TEXTual Files: 415,982,896
2433 Word count: 35,271,297 of them 22,202,980 distinct
2434 Number Of Files: 8
2435 Number Of Lines: 35271297
2436 Allocated memory in MB: 1950
2437 Number Of Trees(GREATER THE BETTER): 3537293
2438 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2439 Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18665687
2440 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '40'
2441 Total Attempts to Find/Put WORDs into Binary-Search-Trees: 117,526,680
2442 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,051,512
2443 Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 89 must have PEAK = 7 = rounding down of integer (1+lb(89))
2444 Binary-Search-Tree(1st out of 1) with MaxNODEs = 89 has PEAK = 25 and LEAFs = 23

```

```

2445 Binary-Search-Tree(1st out of 1) with MaxPEAK = '40' has NODES = 49 and LEAFs = 8
2446 Binary-Search-Tree(1st out of 1) with MaxLEAFs = 28 has NODES = 72 and PEAK = 21
2447 */
2448 #define ROL(x, n) (((x) << (n)) | ((x) >> (32-(n))))
2449 UINT FNV1A_Hash_Jesteress(const char *str, unsigned int wrdlen)
2450 {
2451     const UINT PRIME = 709607;
2452     UINT hash32 = 2166136261;
2453     const char *p = str;
2454
2455     // Idea comes from Igor Pavlov's 7zCRC, thanks.
2456     /*
2457     for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
2458         hash32 = (hash32 ^ *p) * PRIME;
2459     }
2460     */
2461     for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
2462         hash32 = (hash32 ^ (ROL(*(DWORD *)p,5)^(DWORD *)p+4))) * PRIME;
2463     }
2464     // Cases: 0,1,2,3,4,5,6,7
2465     if (wrdlen & sizeof(DWORD)) {
2466         hash32 = (hash32 ^ *(DWORD*)p) * PRIME;
2467         p += sizeof(DWORD);
2468     }
2469     if (wrdlen & sizeof(WORD)) {
2470         hash32 = (hash32 ^ *(WORD*)p) * PRIME;
2471         p += sizeof(WORD);
2472     }
2473     if (wrdlen & 1)
2474         hash32 = (hash32 ^ *p) * PRIME;
2475
2476     return (hash32 ^ (hash32 >> 16)) & 8191;
2477 }
2478
2479 UINT FNV1A_Hash_Jesteress_27bit(const char *str, unsigned int wrdlen)
2480 {
2481     const UINT PRIME = 709607;
2482     UINT hash32 = 2166136261;
2483     const char *p = str;
2484
2485     // Idea comes from Igor Pavlov's 7zCRC, thanks.
2486     /*
2487     for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
2488         hash32 = (hash32 ^ *p) * PRIME;
2489     }
2490     */
2491     for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
2492         hash32 = (hash32 ^ (ROL(*(DWORD *)p,5)^(DWORD *)p+4))) * PRIME;
2493     }
2494     // Cases: 0,1,2,3,4,5,6,7
2495     if (wrdlen & sizeof(DWORD)) {
2496         hash32 = (hash32 ^ *(DWORD*)p) * PRIME;
2497         p += sizeof(DWORD);
2498     }
2499     if (wrdlen & sizeof(WORD)) {
2500         hash32 = (hash32 ^ *(WORD*)p) * PRIME;
2501         p += sizeof(WORD);
2502     }
2503     if (wrdlen & 1)
2504         hash32 = (hash32 ^ *p) * PRIME;
2505
2506     return (hash32 ^ (hash32 >> 16)) & ((1<<HashInBITS)-1);
2507 }
2508
2509 /*
2510 UINT NextPowerOfTwo(UINT x) {
2511     // Henry Warren, "Hacker's Delight", ch. 3.2
2512     x--;
2513     x |= (x >> 1);
2514     x |= (x >> 2);
2515     x |= (x >> 4);
2516     x |= (x >> 8);
2517     x |= (x >> 16);
2518     return x + 1;
2519 }
2520
2521 UINT NextLog2(UINT x) {
2522     // Henry Warren, "Hacker's Delight", ch. 5.3
2523     if(x <= 1) return x;
2524     x--;
2525     UINT n = 0;
2526     UINT y;
2527     y = x >> 16; if(y) {n += 16; x = y;}
2528     y = x >> 8;  if(y) {n += 8;  x = y;}
2529     y = x >> 4;  if(y) {n += 4;  x = y;}
2530     y = x >> 2;  if(y) {n += 2;  x = y;}
2531     y = x >> 1;  if(y) return n + 2;
2532     return n + x;

```

```

2533 }
2534 */
2535
2536 // 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.
2537 /**
2538  * Returns the floor form of binary logarithm for a 32 bit integer.
2539  * -1 is returned if n is 0.
2540  */
2541 int floorLog2(unsigned int n) {
2542     int pos = 0;
2543     if (n >= 1<<16) { n >>= 16; pos += 16; }
2544     if (n >= 1<< 8) { n >>= 8; pos += 8; }
2545     if (n >= 1<< 4) { n >>= 4; pos += 4; }
2546     if (n >= 1<< 2) { n >>= 2; pos += 2; }
2547     if (n >= 1<< 1) { pos += 1; }
2548     return ((n == 0) ? (-1) : pos);
2549 }
2550
2551 // QuickSortExternal_4+GB.c [
2552
2553 int strcmpKAZE13 (
2554     const char * src,
2555     const char * dst
2556 )
2557 {
2558     int ret = 0 ;
2559
2560     while( ! (ret = *(unsigned char *)src - *(unsigned char *)dst) && (*dst!=13-13))
2561         ++src, ++dst;
2562
2563     if ( ret < 0 )
2564         ret = -1 ;
2565     else if ( ret > 0 )
2566         ret = 1 ;
2567
2568     return( ret );
2569 }
2570
2571 // #define LongestLineInclusive 51 //31 former, CAUTION: for command line options 'x' and 'y' it cannot be other than 31 [YET]!
2572
2573 #ifdef singleton
2574 #define LongestLineInclusive 31
2575 #endif
2576 #ifdef doubleton
2577 #define LongestLineInclusive 41
2578 #endif
2579 #ifdef tripleton
2580 #define LongestLineInclusive 41
2581 #endif
2582 #ifdef quadruplet
2583 #define LongestLineInclusive 51
2584 #endif
2585 #ifdef quintuplet
2586 #define LongestLineInclusive 61
2587 #endif
2588 #ifdef sextuplet
2589 #define LongestLineInclusive 71
2590 #endif
2591 #ifdef septuplet
2592 #define LongestLineInclusive 81
2593 #endif
2594 #ifdef octuplet
2595 #define LongestLineInclusive 91
2596 #endif
2597 #ifdef nonuplet
2598 #define LongestLineInclusive 101
2599 #endif
2600 #ifdef decuplet
2601 #define LongestLineInclusive 111
2602 #endif
2603
2604 // _ngram_ 1 1-31
2605 // _ngram_ 2 5-41
2606 // _ngram_ 3 9-41
2607 // _ngram_ 4 13-51
2608 // _ngram_ 5 17-61
2609 // _ngram_ 6 21-71
2610 // _ngram_ 7 25-81
2611 // _ngram_ 8 29-91
2612 // _ngram_ 9 33-101
2613 // _ngram_ 10 37-111
2614 // For Leaf of 256bytes LongestLineInclusive should be 256 = 8+8+8+2*(LongestLineInclusive+1+4) or LongestLineInclusive = (256 - (8+8+8) -
    2*(1+4))/2 = 111
2615
2616 char FourGramL[LongestLineInclusive+1+4]; // 31bytes longest 4-gram + 1byte NULL + 4bytes COUNTER
2617 char FourGramR[LongestLineInclusive+1+4]; // 31bytes longest 4-gram + 1byte NULL + 4bytes COUNTER

```

```

2618 char LEAF[8+8+8+2*(LongestLineInclusive+1+4)]; // 136bytes = 3 pointers + 2 keys
2619 char LEAFNEW[8+8+8+2*(LongestLineInclusive+1+4)]; // 136bytes = 3 pointers + 2 keys
2620 FILE *fp_outRG; // Global - not to burden the extract/compare function with one more parameter
2621 int CompareStringsEndingWith13_EXTERNAL(unsigned long long AtPosition64L, unsigned long long AtPosition64R) {
2622
2623 int i;
2624 unsigned long long *AtPosition64Lpointer=&AtPosition64L;
2625 unsigned long long *AtPosition64Rpointer=&AtPosition64R;
2626
2627 // Caramba: seek and tell report OK but in fact they lie, only setpos works?!?!?!
2628
2629 // #if defined(_WIN32_ENVIRONMENT_)
2630 // _lseeki64( fileno(fp_outRG), AtPosition64L, 0 );
2631 // #else
2632 // fseeko( fp_outRG, AtPosition64L, SEEK_SET );
2633 // #endif /* defined(_WIN32_ENVIRONMENT_) */
2634
2635 // _CRTIMP __int64 __cdecl _telli64(int);
2636 // off64_t ftello64 (FILE *stream)
2637
2638
2639 fsetpos(fp_outRG, AtPosition64Lpointer);
2640 for (i=0; i<(LongestLineInclusive+1+4); i++) {fread(&FourGramL[i], 1, 1, fp_outRG); if (FourGramL[i]==13-13) break;}
2641 //Commented line below is slower than the one above: 778156 clocks vs 756297 clocks.
2642 //fread(&FourGramL[0], 31+1, 1, fp_outRG);
2643
2644 fsetpos(fp_outRG, AtPosition64Rpointer);
2645 for (i=0; i<(LongestLineInclusive+1+4); i++) {fread(&FourGramR[i], 1, 1, fp_outRG); if (FourGramR[i]==13-13) break;}
2646 //Commented line below is slower than the one above: 778156 clocks vs 756297 clocks.
2647 //fread(&FourGramR[0], 31+1, 1, fp_outRG);
2648
2649 return(strcmpKAZE13(FourGramL, FourGramR));
2650 }
2651
2652 int CompareStringsEndingWith13_INTERNAL(unsigned long long AtPosition64L, unsigned long long AtPosition64R, char *POOLinternal) {
2653
2654 int i;
2655 //char FourGramL[LongestLineInclusive+2]; // 31 longest 4-gram + CR + LF
2656 //char FourGramR[LongestLineInclusive+2]; // 31 longest 4-gram + CR + LF
2657
2658 for (i=0; i<(LongestLineInclusive+1+4); i++) {
2659 //fread(&FourGramL[i], 1, 1, fp_in);
2660 FourGramL[i] = *(char *) (POOLinternal + AtPosition64L);
2661 if (FourGramL[i]==13-13) break;
2662 }
2663
2664 for (i=0; i<(LongestLineInclusive+1+4); i++) {
2665 //fread(&FourGramR[i], 1, 1, fp_in);
2666 FourGramR[i] = *(char *) (POOLinternal + AtPosition64R);
2667 if (FourGramR[i]==13-13) break;
2668 }
2669
2670 return(strcmpKAZE13(FourGramL, FourGramR));
2671 }
2672
2673 // QuickSortExternal_4+GB.c ]
2674
2675
2676 int main( argc, argv )
2677 int argc; char *argv[];
2678 {
2679     int nlines;
2680     string *backup = NULL;
2681
2682     FILE *fp_in, *fp_out, *fp_outLOG, *fp_inLINE;
2683     int LetterOffset;
2684     unsigned long long FilesLEN;
2685     unsigned long long WORDcount;
2686     unsigned long long WORDcountBOTTOM;
2687     unsigned long long WORDcountAttemptsToPut;
2688     int Thunderwith;
2689     unsigned long NumberOfFiles, WORDcountDistinct, WORDcountDistinctTOTAL = 0, TotalMemoryNeededForOnePass = 0;
2690     unsigned long long NumberOfLines; // rev. 12+
2691     unsigned long WHOLEletter_BufferSize;
2692     unsigned long long WHOLEletter_BufferSize_L14;
2693     unsigned long memory_size, LetterBuffer, j, k, LINE10len, wrdlen;
2694     unsigned long k_FIX;
2695     unsigned long long i; // rev. 12+
2696     //unsigned long size_in, size_out, size_inLINE;
2697     unsigned long size_in; // rev. 12+
2698 #if defined(_WIN32_ENVIRONMENT_)
2699     unsigned long long size_inLINESIXFOUR;
2700 #else
2701     size_t size_inLINESIXFOUR;
2702 #endif /* defined(_WIN32_ENVIRONMENT_) */
2703
2704     //unsigned long t1, t2, t3;
2705     time_t t1, t2, t3, t4, tMainB, tMainE;

```

```

2706
2707     const int NumberOfSLOTS = 4096*2; // Since r.12+ in rev.12 it was 4096
2708     unsigned long StackPtr;
2709     //unsigned long BSTstack [65536*3]; // BST in worst case could become a LL.
2710     unsigned long long BSTstack [8192*3]; // BST in worst case could become a LL.
2711     unsigned long NumberOfTrees=0, NumberOfHashCollisions=0;
2712     unsigned long iBSTwithMAXpeak, jBSTwithMAXpeak;
2713     unsigned int PEAKibBST;
2714     unsigned long BSTsTotalLEAFs=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur below where 'break'
is ?!
2715     unsigned long BSTwithMAXnode=0, BSTcurrentNode=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2716     unsigned long BSTcurrentNodeMAXQUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
BSTcurrent occur below where 'break' is ?!
2717     unsigned long BSTwithMAXnodePEAK=1, BSTwithMAXnodeLEAF=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
occur below where 'break' is ?!
2718     unsigned long BSTwithMAXpeak=0, BSTcurrentPeak=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2719     unsigned long BSTcurrentPeakMAX=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2720     unsigned long BSTcurrentPeakMAXQUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
BSTcurrent occur below where 'break' is ?!
2721     unsigned long BSTwithMAXpeakNODE=1, BSTwithMAXpeakLEAF=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
occur below where 'break' is ?!
2722     unsigned long BSTwithMAXleaf=0, BSTcurrentLeaf=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2723     unsigned long BSTcurrentLeafMAXQUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
BSTcurrent occur below where 'break' is ?!
2724     unsigned long BSTwithMAXleafNODE=1, BSTwithMAXleafPEAK=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
occur below where 'break' is ?!
2725
2726     char *pointerflush, *pointerflushUNALIGN, *BufStart, *Flushing;
2727     char *pointerflush_64, *pointerflushUNALIGN_64; // r.14++
2728     unsigned long PseudoLinkedPointer, PseudoLinkedPointerNEW, PseudoLinkedPointerROOT, PseudoLinkedPointerNEWold;
2729     unsigned long PseudoLinkedPointerNEWleft, PseudoLinkedPointerNEWright;
2730     unsigned long PseudoLinkedPointerNEWMiddle;
2731     char *bufend[ 806 ]; // 'a'=0, ... 'z'=25 - 26 letters x 31 lengths
2732     long bufNumberOfWords[ 806 ]; // 'a'=0, ... 'z'=25 - 26 letters x 31 lengths
2733     // long bufNowps[ 806 ][ 8192 ]; // ?! crashes below when an attempt to use it occur
2734     char wrd[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2735     char wrdUP[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2736     char wrdUPold[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2737     char LINE10[257]; // 000..255, 256 = 0
2738     char ZEROS[4]; // 0..3, 0 = 0, 1 = 0, 2 = 0, 3 = 0
2739     char CRdLFa[2]; // 0..1, 0 = 13, 1 = 10
2740     char workbyte;
2741     char workk[1024*128];
2742     long workkoffset = -1;
2743     int FoundInLinkedList, slot;
2744     unsigned long OffsetsInBuffer[31]; // 00..30
2745     unsigned long MAXusedBuffer[32]; // 00 not used, only 01..31
2746     unsigned long GRMBLhill[32]; // 00..31
2747     unsigned long GRMBLFoolAgain[32]; // 00..31
2748     int Melnitcka;
2749     unsigned long MAXusedBufferABS = 0;
2750     unsigned long Utilizal = 0;
2751     unsigned long Utiliza2 = 0;
2752     unsigned long TotalWLchars = 0;
2753
2754     /* minimum signed 64 bit value */
2755     #define _I64_MIN    (-9223372036854775807i64 - 1)
2756     /* maximum signed 64 bit value */
2757     #define _I64_MAX    9223372036854775807i64
2758     /* maximum unsigned 64 bit value */
2759     #define _UI64_MAX    0xffffffffffffffffui64
2760
2761     /* minimum signed 128 bit value */
2762     #define _I128_MIN    (-170141183460469231731687303715884105727i128 - 1)
2763     /* maximum signed 128 bit value */
2764     #define _I128_MAX    170141183460469231731687303715884105727i128
2765     /* maximum unsigned 128 bit value */
2766     #define _UI128_MAX    0xffffffffffffffffffffffffffffffffui128
2767
2768     char llToaDigits[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
2769     // below duplicates are needed because of one_line_invoking need different buffers.
2770     char llToaDigits2[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
2771     char llToaDigits3[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
2772     char llToaDigits4[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
2773     unsigned long HEADOffsetFromStartBUKVA = 0;
2774     unsigned long TAILOffsetFromStartBUKVA = 0;
2775     int BSTorBtree = 0;
2776     int SplitOccured;
2777     int PoffsetInLEAF;
2778     char *Auberge[4] = {"|\\0", "\\0", "-\\0", "\\0\\0"};
2779     int hashAlfalfa, iAlfalfa;
2780     int PLE_words=0; // Quadruple!
2781     char wrd1st[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2782     char wrd2nd[LongestLineInclusive+1+4]; // 0..30, 31 = 0

```



```

2783 char wrd3rd[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2784 char wrd4th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2785 char wrd5th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2786 char wrd6th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2787 char wrd7th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2788 char wrd8th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2789 char wrd9th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2790 char wrd10th[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2791 char *DelimiterUnderscore = "_\0";
2792 int PLE_words_INITflag = 0;
2793
2794 // QuickSortExternal_4GB [
2795 unsigned long long ThunderwithL64_L14;
2796 unsigned long long Strnglen64_L14;
2797 unsigned long long size_in64_L14, size_in2_L14;
2798 unsigned long long Over4billionLines, j_Over4billion;
2799 char OneChar_ieByte = '\0';
2800 char CR_ieByte = '\r';
2801 char SomeByte;
2802 unsigned long long BufEnd_64;
2803 unsigned long long SeekPosition;
2804 unsigned long long *PointerToSeekPosition;
2805 char FourGram[LongestLineInclusive+2]; // 31 longest 4-gram + CR + LF
2806 char *PoolPhysical;
2807 unsigned long long fsetpos_ZERO=0;
2808 char OneClusterZEROES[1024*4]; // Caution: must be ZEROed(NULLlified)!
2809 char *FileSwapTag = "LEPRECHAUNISH";
2810 char EOFcode = 0x1A;
2811 unsigned long long PseudoLinkedPointer_64, PseudoLinkedPointerNEW_64, PseudoLinkedPointerROOT_64, PseudoLinkedPointerNEWold_64;
2812     unsigned long long PseudoLinkedPointerNEWleft_64, PseudoLinkedPointerNEWright_64;
2813     unsigned long long PseudoLinkedPointerNEWmiddle_64;
2814     unsigned long long NULLs_64 = 0;
2815 unsigned long long PseudoLinkedPointerAUX_64;
2816 unsigned long long PseudoLinkedPointerAUXdumbo_64;
2817     char wrdAUX[LongestLineInclusive+1+4]; // 0..30, 31 = 0
2818 // QuickSortExternal_4GB ]
2819
2820 unsigned long CounterOccurrences;
2821 unsigned long long NumberOfLEAFs=0;
2822 unsigned long LevelsInCorona_Not_Counting_ROOT=0;
2823 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"};
2824
2825 unsigned long RipPasses;
2826 unsigned long long NULLSForWRD=0;
2827
2828 //15+
2829 int DoNotInsertFlag = 0;
2830 int METACOMMANDFlag;
2831
2832 //16
2833 int REUSE=0;
2834 int HSHexist;
2835 int SWPexist;
2836
2837 // INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT
2838 printf ("Leprechaun_%s (Fast-In-Future Greedy n-gram-Ripper), rev. 16FIX, written by Svalqyatchx.\n", ngram[_ngram_]);
2839 //puts("Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'");
2840
2841 #ifdef singleton
2842 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 1..31 chars from incoming texts.\n", _ngram_, _ngram_);
2843 #endif
2844 #ifdef doubleton
2845 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 5..41 chars from incoming texts.\n", _ngram_, _ngram_);
2846 #endif
2847 #ifdef tripleton
2848 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 9..41 chars from incoming texts.\n", _ngram_, _ngram_);
2849 #endif
2850 #ifdef quadrupleton
2851 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 13..51 chars from incoming texts.\n", _ngram_, _ngram_);
2852 #endif
2853 #ifdef quintupleton
2854 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 17..61 chars from incoming texts.\n", _ngram_, _ngram_);
2855 #endif
2856 #ifdef sextupleton
2857 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 21..71 chars from incoming texts.\n", _ngram_, _ngram_);
2858 #endif
2859 #ifdef septupleton
2860 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 25..81 chars from incoming texts.\n", _ngram_, _ngram_);
2861 #endif
2862 #ifdef octupleton
2863 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 29..91 chars from incoming texts.\n", _ngram_, _ngram_);
2864 #endif
2865 #ifdef nonupleton
2866 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 33..101 chars from incoming texts.\n", _ngram_, _ngram_);
2867 #endif
2868 #ifdef decupleton

```

```

2869 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 37..111 chars from incoming texts.\n", _ngram_, _ngram_);
2870 #endif
2871 puts( "Feature1: All words within x-lets/n-grams are in range 1..31 chars inclusive." );
2872 //puts( "Feature2: In this revision 128MB 1-way hash is used which results in 16,777,216 external B-Trees of order 3." );
2873
2874 if (HashInBITS+3<10)
2875 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), llToADigits, 10), _ui64toaKAZEcomma((1<<HashInBITS), llToADigits2, 10) );
2876 else if (HashInBITS+3>=10 && HashInBITS+3<20)
2877 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, llToADigits, 10), _ui64toaKAZEcomma((1<<HashInBITS), llToADigits2, 10) );
2878 else
2879 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, llToADigits, 10), _ui64toaKAZEcomma((1<<HashInBITS), llToADigits2, 10) );
2880 if (HashInBITS-HashChunkSizeInBITS==0)
2881 printf ("Feature3: In this revision %s pass is to be made.\n", _ui64toaKAZEcomma(1<<(HashInBITS-HashChunkSizeInBITS), llToADigits, 10));
2882 else
2883 printf ("Feature3: In this revision %s passes are to be made.\n", _ui64toaKAZEcomma(1<<(HashInBITS-HashChunkSizeInBITS), llToADigits, 10));
2884
2885 puts( "Feature4: If the external memory has latency 99+microseconds then !(look no further), IOPS(seek-time) rules." );
2886 // The phrase 'look no further' was used in amazon.com review meaning 'stop searching for better thing this is it'.
2887 //puts( "Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us," );
2888 //puts( "    also the performance of a 3-way hash + 6,602,752 B-Trees of order 3," );
2889 //puts( "    also the performance of a 1-way hash + 134,217,728 external B-Trees of order 3." );
2890 //puts( "Note1: Compiled with Microsoft C v. 13.10.3077: 'cl /ox /TcLeprechaun.c'." );
2891 //puts( "Note2: This WORDLISTER makes as output pseudo(unsorted)_wordlist_CRLF_file." );
2892 if( argc != 3 && argc != 4 && argc != 5 && argc != 6 ) // +1 for program name
2893 {
2894     puts( "" );
2895     puts( "'The Little Monster' short notes:" );
2896     puts( "Note1: I wish to thank to R.N. Horspool, Ranjan Sinha, Dmitry Shkarin," );
2897     puts( "    Michael Abrash, J. Bentley, R. Sedgewick, Igor Pavlov, Lasse Reinhold," );
2898     puts( "    Landon Noll, Peter Kankowski for sharing their knowledge to public." );
2899     puts( "Note2: Run it without parameters to get usage and short notes." );
2900     puts( "Note3: This simple amateurish(more over I am not versed well neither in C nor" );
2901     puts( "    in mathematics nor in English language, but I am persistent in INDEXING" );
2902     puts( "    GBS of english TEXTS) tool is written in ANSI C(at least its source is" );
2903     puts( "    compileable for CL(windows) and GCC(Linux)), and its purpose is to" );
2904     puts( "    create a wordList for a group of files(given via fileList)." );
2905     puts( "    Its name comes(according to Heritage Dictionary) from 'low corpus' or" );
2906     puts( "    'little body', in fact from amazing movie saga 'Leprechaun 1-2-3-4-5-6'" );
2907     puts( "    starring by Warwick Davis." );
2908     puts( "Note4: Only words up to 31 chars are proceeded - the reason is 'DDT'(the" );
2909     puts( "    longest word in Heritage Dictionary 3rd edition) or" );
2910     puts( "    'dichlorodiphenyltrichloroethane.'" );
2911     puts( "Note5: Cursor hiding in C - mission impossible for me." );
2912     puts( "Note6: By default(third parameter is 1023) allocated memory is 393MB." );
2913     puts( "    Due to 'malloc()' limitation under WINDOWS, maximum value of third" );
2914     puts( "    parameter is 5174 which is 1988MB allocated block." );
2915     puts( "Note7: File Leprechaun.LOG is a log, where new statistics are appended." );
2916     puts( "Note8: Revision 12+ can handle files larger than 4GB." );
2917     puts( "Note9: Revision 12++ has a buffered 'fread()' - therefore I/O READ-BURST SPEED" );
2918     puts( "    is the first(worst) bottleneck, as a result r.12++ is much-much faster;" );
2919     puts( "    the second(worse) bottleneck: the linked lists - the b-trees" );
2920     puts( "    might be the answer; the third(bad) bottleneck: the amateurish author." );
2921     puts( "NoteA: Revision 12+++ has an improved(2 bits were used doltishly) main hash" );
2922     puts( "    function - therefore less collisions, for example:" );
2923     puts( "    for file 'wikipedia-de-html.tar' 42,291,855,360 bytes with" );
2924     puts( "    5,750,179,678 words of them 7,375,373 distinct attempts to Find/Put" );
2925     puts( "    a WORD into a linked list are 6,117,675,470(r.12++) and 5,845,989,790" );
2926     puts( "    (r.12+++); also two 'if' sections were moved because they were executed" );
2927     puts( "    unnecessarily many times." );
2928     puts( "NoteB: Revision 13 uses BSTs instead of LLs, that is Linked-Lists were" );
2929     puts( "    replaced by Binary-Search-Trees, as a result for 22,202,980 distinct" );
2930     puts( "    words(out of 35,271,297) r.12+++ needs 225,548,268 total attempts to" );
2931     puts( "    Find/Put WORDS into linked lists where r.13 needs 121,674,042 total" );
2932     puts( "    attempts to Find/Put WORDS into Binary-Search-Trees. But this is a" );
2933     puts( "    significant boost in performance only for wordlists of million words." );
2934     puts( "NoteC: Revision 13+ gives only more statistics. Future revisions could lessen" );
2935     puts( "    number of attempts to Find/Put WORDS into Binary-Search-Trees" );
2936     puts( "    furthermore by making them at some point Perfectly-Balanced. But" );
2937     puts( "    for huge amount(multi-(m|b)illion) of distinct words the b-tree family" );
2938     puts( "    must come in, until then this is the leprechaunish niche." );
2939     puts( "NoteD: Revision 13++ has a little fix(2 unnecessary ZEROings, when a new word" );
2940     puts( "    is inserted, were deleted) and a fixed bug(13+ adds stupidly the" );
2941     puts( "    highest BST to the wordlist). Also B-Tree of order 3 is added as a" );
2942     puts( "    searching method. Main goal of B-Tree is to reduce number of" );
2943     puts( "    comparisons but at nasty cost: a precious time wasted to construct it" );
2944     puts( "    and twice more memory, i.e. one step forward two backward: this tree is" );
2945     puts( "    more effective than BST in cases of 2++ billion/million" );
2946     puts( "    different/distinct words." );
2947     puts( "    The improvement which comes from using B-Tree of order 3 is about 200%" );
2948     puts( "    much more pleasing than I expected, for wikipedia-en-html.tar.wrd with" );
2949     puts( "    12,561,874 distinct words Total Attempts to Find/Put WORDS into:" );
2950     puts( "    Binary-Search-Trees was 61,895,043 while for" );
2951     puts( "    B-trees order 3 was 19,295,791." );
2952     puts( "NoteE: Revision 13+++ has a faster(not heavily tested yet) and with" );
2953     puts( "    better(0.6% to 1.1%) dispersion Fowler/Noll/Vo hash," );

```

```

2954 puts( "    so called FNV1a hash. Revision 13++++ boosting: Leprechaun_Intel.exe" );
2955 puts( "    gives 1,256,187w/s for wikipedia-en-html.tar.wrd with FNV1_32_PRIME:" );
2956 puts( "    107712257 with 3,551,736 dispersion for 'FNV1A_Hash_Granularity'." );
2957 puts( "NoteF: For old r.12+ a USB connected HDD crippled test:" );
2958 puts( "    for 'H:\>Leprechaun.exe static.wikipedia.org_downloads_2008-06_en.1st" );
2959 puts( "    wikipedia-en-html.tar.wrd 5400'" );
2960 puts( "    where 223,674,511,360 wikipedia-en-html.tar" );
2961 puts( "    on laptop Toshiba Pentium T3400 2166 MHz with" );
2962 puts( "    Motherboard Name: Toshiba Satellite L305" );
2963 puts( "    CPU Type: Mobile DualCore Intel Pentium, 2166 MHz (13 x 167)" );
2964 puts( "    CPU Alias: Merom-1M" );
2965 puts( "    L1 Code Cache: 32 KB per core" );
2966 puts( "    L1 Data Cache: 32 KB per core" );
2967 puts( "    L2 Cache: 1 MB (On-Die, ECC, ASC, Full-Speed)" );
2968 puts( "    Bus Type: Dual DDR2 SDRAM" );
2969 puts( "    Bus Width: 128-bit" );
2970 puts( "    Real Clock: 333 MHz (DDR)" );
2971 puts( "    Effective Clock: 666 MHz" );
2972 puts( "    EVEREST v5.00.1650 Memory Copy: 3725MB/s with timings 5-5-5-13" );
2973 puts( "    result is logged to 'Leprechaun.LOG':" );
2974 puts( "    Bytes per second performance: 20,658,955B/s" );
2975 puts( "    Words per second performance: 2,860,880W/s" );
2976 puts( "    Input File with a list of TEXTual Files:" );
2977 puts( "    static.wikipedia.org_downloads_2008-06_en.1st" );
2978 puts( "    Size of all TEXTual Files: 223,674,511,360" );
2979 puts( "    Word count: 30,974,750,142 of them 12,561,874 distinct" );
2980 puts( "    Number Of Files: 1" );
2981 puts( "    Number Of Lines: 2088618575" );
2982 puts( "    Allocated memory in MB: 1920" );
2983 puts( "    Words with length 01 occupy 0,033KB of 0,349KB given i.e. 09% utilization" );
2984 puts( "    Words with length 02 occupy 0,033KB of 0,349KB given i.e. 09% utilization" );
2985 puts( "    Words with length 03 occupy 0,037KB of 0,697KB given i.e. 05% utilization" );
2986 puts( "    Words with length 04 occupy 0,151KB of 0,871KB given i.e. 17% utilization" );
2987 puts( "    Words with length 05 occupy 0,744KB of 1,568KB given i.e. 47% utilization" );
2988 puts( "    Words with length 06 occupy 1,470KB of 3,136KB given i.e. 46% utilization" );
2989 puts( "    Words with length 07 occupy 2,605KB of 5,923KB given i.e. 43% utilization" );
2990 puts( "    Words with length 08 occupy 3,296KB of 6,968KB given i.e. 47% utilization" );
2991 puts( "    Words with length 09 occupy 3,714KB of 6,968KB given i.e. 53% utilization" );
2992 puts( "    Words with length 10 occupy 3,483KB of 6,968KB given i.e. 49% utilization" );
2993 puts( "    Words with length 11 occupy 3,235KB of 5,923KB given i.e. 54% utilization" );
2994 puts( "    Words with length 12 occupy 2,691KB of 4,181KB given i.e. 64% utilization" );
2995 puts( "    Words with length 13 occupy 2,230KB of 3,484KB given i.e. 64% utilization" );
2996 puts( "    Words with length 14 occupy 1,718KB of 3,484KB given i.e. 49% utilization" );
2997 puts( "    Words with length 15 occupy 1,357KB of 2,613KB given i.e. 51% utilization" );
2998 puts( "    Words with length 16 occupy 1,063KB of 2,613KB given i.e. 40% utilization" );
2999 puts( "    Words with length 17 occupy 0,814KB of 1,742KB given i.e. 46% utilization" );
3000 puts( "    Words with length 18 occupy 0,617KB of 1,742KB given i.e. 35% utilization" );
3001 puts( "    Words with length 19 occupy 0,485KB of 1,742KB given i.e. 27% utilization" );
3002 puts( "    Words with length 20 occupy 0,402KB of 1,742KB given i.e. 23% utilization" );
3003 puts( "    Words with length 21 occupy 0,327KB of 1,742KB given i.e. 18% utilization" );
3004 puts( "    Words with length 22 occupy 0,274KB of 1,742KB given i.e. 15% utilization" );
3005 puts( "    Words with length 23 occupy 0,224KB of 1,394KB given i.e. 16% utilization" );
3006 puts( "    Words with length 24 occupy 0,190KB of 1,394KB given i.e. 13% utilization" );
3007 puts( "    Words with length 25 occupy 0,162KB of 1,394KB given i.e. 11% utilization" );
3008 puts( "    Words with length 26 occupy 0,136KB of 1,220KB given i.e. 11% utilization" );
3009 puts( "    Words with length 27 occupy 0,119KB of 1,046KB given i.e. 11% utilization" );
3010 puts( "    Words with length 28 occupy 0,107KB of 0,871KB given i.e. 12% utilization" );
3011 puts( "    Words with length 29 occupy 0,091KB of 0,697KB given i.e. 13% utilization" );
3012 puts( "    Words with length 30 occupy 0,080KB of 0,523KB given i.e. 15% utilization" );
3013 puts( "    Words with length 31 occupy 0,076KB of 0,523KB given i.e. 14% utilization" );
3014 puts( "    Total pseudo(including hash table) memory utilization: 42%" );
3015 puts( "    Total real(wordlist's words vs allocated block) memory utilization: 60/1000" );
3016 puts( "    Used value for third parameter in KB: 5400" );
3017 puts( "    Use next time as third parameter: 3475-" );
3018 puts( "    Time for making unsorted wordlist: 10827 second(s)" );
3019 puts( "    Time for sorting unsorted wordlist: 10 second(s)" );
3020 puts( "NoteG: 2011-Mar-07: Fixed a small command line parsing bug." );
3021 puts( "NoteH: A heavy blow for my illusions(regarding speed performance of external b-trees)," );
3022 puts( "    desperate results for ripping on HDD 7200rpm:" );
3023 puts( "    20,000,000 distinct 4-grams per 5 hours." );
3024 puts( "    D:\>Leprechaun_quadupleton_r14_minus>Leprechaun_quadupleton GRAFFITH_2048.1st GRAFFITH_2048.wrd 48000000 z" );
3025 puts( "    Leprechaun(Fast Greedy Word-Ripper), rev. 14_minus_quadupleton, written by Svalqyatchx." );
3026 puts( "    Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'" );
3027 puts( "    Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us," );
3028 puts( "    also the performance of a 3-way hash + 6,602,752 B-Trees of order 3." );
3029 puts( "    also the performance of a 1-way hash + 134,217,728 external B-Trees of order 3." );
3030 puts( "    Size of input file with files for Leprechauning: 42140" );
3031 puts( "    Allocating HASH memory 1,073,741,889 bytes ... OK" );
3032 puts( "    Allocating/ZEROing 49,152,000,014 bytes swap file ... OK" );
3033 puts( "    Size of Input TEXTual file: 33,470,581" );
3034 puts( "    |; word count: 3,045,077 of them 2,597,942 distinct; Done: 64/64" );
3035 puts( "    Size of Input TEXTual file: 17,229,900" );
3036 puts( "    -; word count: 4,235,032 of them 3,588,757 distinct; Done: 64/64" );
3037 puts( "    Size of Input TEXTual file: 19,191,256" );
3038 puts( "    |; word count: 5,803,400 of them 4,866,213 distinct; Done: 64/64" );
3039 puts( "    Size of Input TEXTual file: 34,651,077" );
3040 puts( "    \\; word count: 8,714,961 of them 6,941,108 distinct; Done: 64/64" );
3041 puts( "    Size of Input TEXTual file: 26,875,458" );

```

```

3042 puts( "      /; word count: 11,022,830 of them 8,579,931 distinct; Done: 64/64" );
3043 puts( "      Size of Input TEXTual file: 19,605,129" );
3044 puts( "      -; word count: 12,924,821 of them 10,078,191 distinct; Done: 64/64" );
3045 puts( "      Size of Input TEXTual file: 17,053,521" );
3046 puts( "      /; word count: 14,577,010 of them 11,455,983 distinct; Done: 64/64" );
3047 puts( "      Size of Input TEXTual file: 44,087,709" );
3048 puts( "      -; word count: 18,953,280 of them 15,010,569 distinct; Done: 64/64" );
3049 puts( "      Size of Input TEXTual file: 32,796,705" );
3050 puts( "      |; word count: 22,412,912 of them 17,621,649 distinct; Done: 64/64" );
3051 puts( "      Size of Input TEXTual file: 19,538,360" );
3052 puts( "      /; word count: 24,381,005 of them 19,137,701 distinct; Done: 64/64" );
3053 puts( "      Size of Input TEXTual file: 29,565,366" );
3054 puts( "      \\; word count: 26,214,400 of them 20,528,357 distinct; Done: 40/64" );
3055 puts( "      ..." );
3056 puts( "NoteI: In revision 14- the resultant wordlist is NOT sorted when 'z' is used." );
3057 puts( "NoteJ: In revision 14 'x' and 'y' options are disabled, for 7++ million phrases their usefulness is no more." );
3058 puts( "      the real loads are of order 800+ million, too many limitations exist, they must be rewritten as 64bit." );
3059 puts( "NoteK: Ripping OSHO.TXT (10,165,640 4-grams) on HDD daunts because of 6+hours needed." );
3060 puts( "      Number Of Trees(GREATER THE BETTER): 9,433,894" );
3061 puts( "      Used value for third parameter in KB: 3,145,728" );
3062 puts( "      Use next time as third parameter: 1,262,186" );
3063 puts( "      One leaf has size: 8+8+8+(51+1+4)+(51+1+4)=136bytes." );
3064 puts( "      or MAX (one 4-gram per leaf) 10,165,640*136=1,382,527,040bytes." );
3065 puts( "NoteL: Each phrase in extracted file is preceded by TAB ASCII code, this (TAB being a delimiter symbol) allows" );
3066 puts( "      the phrase-list to be ripped again i.e. to treat already ripped files as any other text." );
3067 puts( "NoteM: Too many 'fsetpos', 'fread', 'fwrite' invocations were put in the straight port (from 32bit internal memory to" );
3068 puts( "      64bit external memory), a optimization is needed, something like reading/writing a LEAF at once." );
3069 puts( "NoteN: Since revision 14+: Optimized(LEAFwise) search (fragment 1] and 2]), insert (fragment 3]) and dump." );
3070 puts( "NoteO: In next revisions a 2in1 is to be done i.e. one code fragment will deal with virtual and physical memory." );
3071 puts( "      thus establishing pure 64bit mode of operation, a single flag will decide whether 'memcpy' or" );
3072 puts( "      the slow I/O triad sub-fragments will be used. DONE." );
3073 puts( "NoteP: In next revisions a multi-pass (by chunking the hash table) mode is to be added in order to avoid" );
3074 puts( "      these sick-seeks. DONE." );
3075 puts( "NoteQ: Fixed occurencies bug due to not NULLifying the field housing the occurencies, a nasty thing: all" );
3076 puts( "      the revisions 14??? were buggy, how stupid from my side, grumble." );
3077 puts( "NoteR: In r.14++++FIXFIX were fixed STATS(Leprechaun.LOG) bugs (appearing only in multi-pass mode) due to not" );
3078 puts( "      NULLifying the variables housing the stats, they do not affect the results - they are for informative use." );
3079 puts( "Notes: Fixed a division-by-zero bug, occurs when finishing-starting time is under 1 second." );
3080 puts( "      Fixed a nasty bug causing very restrictive way of forming x-grams." );
3081 puts( "NoteT: At last and finally the nasty bug causing very restrictive way of forming x-grams was REALLY fixed - lack of" );
3082 puts( "      calmness jammed (again) my actions - a lesson to be learnt." );
3083
3084 puts( "NoteU: Since r.15FIXFIX+ the ability to command Leprechaun (from inside the list file with 2 metacommands) to enter/exit" );
3085 puts( "      INSERT mode was added. This allows to control whether new (to current hash-tree structure) x-grams are to be counted" );
3086 puts( "      [and] INSERTED. These two metacommands are:" );
3087 puts( "      Leprechaun says x-gram inserting disabled for next files: ON" );
3088 puts( "      Leprechaun says x-gram inserting disabled for next files: OFF" );
3089
3090 puts( "Notev: When w/w option is used multiple-passes shouldn't be dumped - it is meaningless, dump when only one pass" );
3091 puts( "      that is, use w/w only in ONE-PASS mode otherwise it behaves as Z/z but DOES NOT dump to OutFile." );
3092 puts( "      It uses in READ mode the two HASH+TREES output files: 'Leprechaun_64bit.hsh' and 'Leprechaun_64bit.swp'." );
3093 puts( "      If during the start one of them is missing then Z/z behaviour is on, at end 'Leprechaun_64bit.hsh' is dumped." );
3094 puts( "      Also the OutFile has all incoming x-grams which are present in the corpus (i.e. HASH+TREES structure)." );
3095 puts( "" );
3096 puts( "Usage: Leprechaun InFile OutFile [BufferSize] [SortMethod] [TreeMethod]" );
3097 puts( "      <InFile>: Input file with files for Leprechauning, in WINDOWS console" );
3098 puts( "      you can create it by 'E:\\KAZEHOME>dir *.txt/s/b>Leprechaun.lst'" );
3099 puts( "      <OutFile>: Output WORDLIST(sorted since r.9, CRLF) file" );
3100 puts( "      <BufferSize>: Optional Dynamic RAM buffer in KB, default(and minimum" );
3101 puts( "      in the same time) is 1023, i.e. omit or specify greater one" );
3102 puts( "      <SortMethod>: Optional Sort Method, default is 'D'," );
3103 puts( "      A - InsertionSort" );
3104 puts( "      B - InsertionX26Sort" );
3105 puts( "      C - MultiKeyQuickSortSort by J. Bentley, R. Sedgewick" );
3106 puts( "      D - MultiKeyQuickSortX26Sort' by J. Bentley, R. Sedgewick" );
3107 puts( "      <TreeMethod>: Optional Tree Method, default is 'X'," );
3108 puts( "      X - Binary-Search-Trees" );
3109 puts( "      y - B-Trees of order 3, INTERNAL/fast memory digitless i.e. no repetitions, 64bit addressing!" );
3110 puts( "      Y - B-Trees of order 3, INTERNAL/fast memory, 64bit addressing!" );
3111 puts( "      z - B-Trees of order 3, EXTERNAL/slow memory digitless i.e. no repetitions, 64bit addressing!" );
3112 puts( "      Z - B-Trees of order 3, EXTERNAL/slow memory, 64bit addressing!" );
3113 puts( "      w - B-Trees of order 3, EXTERNAL/slow memory digitless i.e. no repetitions, 64bit addressing! REUSE!" );
3114 puts( "      W - B-Trees of order 3, EXTERNAL/slow memory, 64bit addressing! REUSE!" );
3115 puts( "" );
3116 puts( "Have a nice Leprechauning." );
3117 puts( "For contacts: sanmayce@sanmayce.com" );
3118 puts( "Sanmayce Svalgyatchx 'Kaze', 2005 Feb 07. Last revision: 2012 Dec 16." );
3119 return( 1 );
3120 }
3121
3122 GRMBLhill[0]=0;
3123 GRMBLhill[1]=1;
3124 GRMBLhill[2]=1;
3125 GRMBLhill[3]=1;
3126 GRMBLhill[4]=1;
3127 GRMBLhill[5]=1;
3128 GRMBLhill[6]=1;
3129 GRMBLhill[7]=1;

```

```

3130 GRMBLhi11[8]=1;
3131 GRMBLhi11[9]=1;
3132 GRMBLhi11[10]=1;
3133 GRMBLhi11[11]=1;
3134 GRMBLhi11[12]=15;
3135 GRMBLhi11[13]=15;
3136 GRMBLhi11[14]=15;
3137 GRMBLhi11[15]=20;
3138 GRMBLhi11[16]=30;
3139 GRMBLhi11[17]=40;
3140 GRMBLhi11[18]=50;
3141 GRMBLhi11[19]=50;
3142 GRMBLhi11[20]=50;
3143 GRMBLhi11[21]=40;
3144 GRMBLhi11[22]=40;
3145 GRMBLhi11[23]=40;
3146 GRMBLhi11[24]=30;
3147 GRMBLhi11[25]=20;
3148 GRMBLhi11[26]=20;
3149 GRMBLhi11[27]=20;
3150 GRMBLhi11[28]=20;
3151 GRMBLhi11[29]=20;
3152 GRMBLhi11[30]=10;
3153 GRMBLhi11[31]=10;
3154
3155 (void) time(&tMainB);
3156
3157 if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
3158 { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
3159 fclose(fp_out); // The file must be with size 0 because it is opened for appending down below.
3160
3161 // 2^(HashInBITS-HashChunkSizeInBITS)=2^0=1 passe(s).
3162 // 14++++ [
3163 //for( RipPasses = 1-1; RipPasses <= (1<<(HashInBITS-HashChunkSizeInBITS))-1; RipPasses++ )
3164 //{
3165 // 14++++ ]
3166 RipPasses = 1-1;
3167 whyTheHellForIsNotWorking:
3168 printf( "Pass #%lu of %lu:\n", RipPasses+1, (1<<(HashInBITS-HashChunkSizeInBITS)));
3169
3170 if( ( fp_in = fopen( argv[1], "rb" ) ) == NULL )
3171 { printf( "Leprechaun: Can't open file %s \n", argv[1] ); return( 1 ); }
3172
3173 fseek( fp_in, 0L, SEEK_END );
3174 size_in = ftell( fp_in );
3175 fseek( fp_in, 0L, SEEK_SET );
3176 printf( "Size of input file with files for Leprechauning: %lu\n", size_in );
3177
3178 if( ( fp_outLOG = fopen( "Leprechaun.LOG", "a+" ) ) == NULL )
3179 { printf( "Leprechaun: Can't open file Leprechaun.LOG.\n" ); return( 1 ); }
3180
3181 // argc is 4|5|6 due to eventual missing BufferSize
3182 if( argc == 4 ) // not 6 due to eventual missing BufferSize and SortMethod
3183     k_FIX = 3;
3184 if( argc == 5 ) // not 6 due to eventual missing BufferSize or SortMethod
3185     k_FIX = 4;
3186 if( argc == 6 )
3187     k_FIX = 5;
3188 if ( *argv[k_FIX] == 'y' || *argv[k_FIX] == 'y') BStorBtree = 1+2; // +2 since r.14++
3189 if ( *argv[k_FIX] == 'z' || *argv[k_FIX] == 'z') BStorBtree = 2;
3190 if ( *argv[k_FIX] == 'w' || *argv[k_FIX] == 'w') {BStorBtree = 2; REUSE=1;}
3191
3192 if( argc == 4 || argc == 5 || argc == 6 ) Thunderwith = atoi( argv[3] );
3193 else Thunderwith = 527; // for r.12: 527=17*31 this is minimum because of 4096*1*4=16KB+ needed for each buffer!
3194 // for r.12+: 1023=33*31 this is minimum because of 4096*2*4=32KB+ needed for each buffer!
3195 if (Thunderwith < 1023) {Thunderwith = 1023;}
3196
3197 //printf( "Use next time as third parameter: %s\n", _ui64toaKAZEcomma((25123456789>>10)+1, 11ToaDigits, 10) );
3198 //printf( "Use next time as third parameter: %s\n", _ui64toaKAZEcomma((25123456789/1024)+1, 11ToaDigits, 10) );
3199
3200 if (BStorBtree < 2) { printf( "Leprechaun: In this particular revision 'x' option is disabled.\n" ); return( 1 ); }
3201
3202 if (BStorBtree < 2) {
3203     LetterBuffer = Thunderwith * 1024;
3204     WHOLEletter_BufferSize = 0;
3205     for( i = 1; i <= 31; i++ )
3206     { OffsetsInBuffer[i-1] = 0;
3207       for( j = 1; j <= i; j++ )
3208       { OffsetsInBuffer[i-1] = OffsetsInBuffer[i-1] + (GRMBLhi11[(int)(j-1)] * LetterBuffer)/31;
3209       }
3210       WHOLEletter_BufferSize = WHOLEletter_BufferSize + (GRMBLhi11[(int)i] * LetterBuffer)/31;
3211       GRMBLFoolAgain[(int)i] = (GRMBLhi11[(int)i] * LetterBuffer)/31;
3212     }
3213     memory_size = 26 * WHOLEletter_BufferSize + 1 + 64;
3214     printf( "Allocating memory %luMB ... ", (memory_size>>20)+1 );
3215     pointerflushUNALIGN = (char *)malloc( memory_size );
3216     if( pointerflushUNALIGN == NULL )
3217     { puts( "\nLeprechaun: Needed memory allocation denied!\n" ); return( 1 ); }

```

```

3218 pointerflush = pointerflushUNALIGN + 64 - (((size_t)pointerflushUNALIGN) % 64); // 13_6+
3219 //offset=64-int((long)data&63);
3220
3221 printf( "OK\n");
3222         fprintf( fp_outLOG, "Leprechaun report:\n" );
3223
3224 // Check once for ever whether allocated memory is ZEROed!? Answer: YES
3225 //for( i = 0; i < memory_size; i++ )
3226 // if (*(char *) (pointerflush+i)!=0) printf("NON-ZERO encountered, so 'NO'.");
3227
3228 for( i = 0; i < 26; i++ )
3229 { for( k = 1; k <= 31; k++ )
3230   { bufend[i*31+k-1] = pointerflush + i * WHOLEletter_BufferSize + OffsetsInBuffer[k-1]; // i*31+k-1 must be 0..805
3231     if (i==25) { MAXusedBuffer[k] = (unsigned long)bufend[i*31+k-1]; }
3232     for( j = 0; j < (NumberOfSLOTS+1)*4; j++ ) // ? memset(bufend[i],0,(NumberOfSLOTS+1)*4);
3233     { *bufend[i*31+k-1]++ = 0;
3234       //++bufend[i*31+k-1];
3235     }
3236     if (i==25) { MAXusedBuffer[k] = (unsigned long)bufend[i*31+k-1]-MAXusedBuffer[k]; }
3237     bufNumberOfWords[i*31+k-1]=0;
3238 //for( j = 0; j < NumberOfSLOTS; j++ )
3239 //bufNowps[i*31+k-1][j]=0;
3240   }
3241 }
3242
3243 } else { //if (BSTorBtree != 2) {
3244 // _ ASCII code 095
3245 // _ ASCII code 096 \
3246 // a ASCII code 097 / In total 26+1+1 radix instead of 27 to avoid +1 for each '_', code 096 not used.
3247 // z ASCII code 122
3248 // The hash for 'a quadruplet_for_example' will be calculated for first 5 chars:
3249 // = (byte1-'_')*28*28*28*28 + (byte2-'_')*28*28*28 + (byte3-'_')*28*28 + (byte4-'_')*28 + (byte5-'_')
3250 // Hash slots are 28*28*28*28 = 17,210,368 each containing one 64bit pointer i.e. 8bytes in length.
3251 // Hash size = 17,210,368*8 = 137,682,944 bytes
3252 // When at end all these slots(17,210,368- Btrees) are traversed the outcome is a sorted wordlist - no need of sorting.
3253 //unsigned long long SeekPosition;
3254 //unsigned long long *PointerToSeekPosition;
3255 // The 64bit external pool will be addressed via fsetpos(fp_outRG, PointerToSeekPosition); similarly to bufend approach from r.13 - that is
//bufend points to first(always following the last used btree leaf) free position in the pool.
3256 // For final stats all non-zero slots point to one btree.
3257 // printf( "Allocating HASH memory %s bytes ... ", _ui64toaKAZEcomma( (17210368*8) + 1 + 64 , llToADigits, 10) );
3258 // pointerflushUNALIGN = (char *)malloc( (17210368*8) + 1 + 64 );
3259 // Hash slots are 27bit = 2^27 = 134,217,728 each containing one 64bit pointer i.e. 8bytes in length.
3260 printf( "Allocating HASH memory %s bytes ... ", _ui64toaKAZEcomma( ((1<<HashInBITS)*8) + 1 + 64 , llToADigits, 10) );
3261 pointerflushUNALIGN = (char *)malloc( (1<<HashInBITS)*8 + 1 + 64 );
3262 if( pointerflushUNALIGN == NULL )
3263 { puts( "\nLeprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
3264 // r16
3265 pointerflush = pointerflushUNALIGN;
3266 //pointerflush = pointerflushUNALIGN + 64 - (((size_t)pointerflushUNALIGN) % 64); // 13_6+
3267 //offset=64-int((long)data&63);
3268 printf( "OK\n");
3269 // memset(pointerflush,0,17210368*8);
3270 memset(pointerflush,0,(1<<HashInBITS)*8);
3271         if (BSTorBtree == 2) {
3272
3273 if( ( fp_outRG = fopen( "Leprechaun_64bit.hsh", "rb" ) ) == NULL )
3274 {
3275     HSHexist=0;
3276     if ( REUSE && ((HashInBITS-HashChunkSizeInBITS)==0) ) // Multiple-passes shouldn't be uploaded - it is meaningless, dump when only one
pass.
3277         printf( "Leprechaun: Can't find file 'Leprechaun_64bit.hsh'.\n" );
3278 } else {
3279     HSHexist=1;
3280     fclose(fp_outRG);
3281 }
3282
3283 if( ( fp_outRG = fopen( "Leprechaun_64bit.swp", "rb" ) ) == NULL )
3284 {
3285     SWPexist=0;
3286     if ( REUSE && ((HashInBITS-HashChunkSizeInBITS)==0) )
3287         printf( "Leprechaun: Can't find file 'Leprechaun_64bit.swp'.\n" );
3288 } else {
3289     SWPexist=1;
3290     fclose(fp_outRG);
3291 }
3292
3293 if ( REUSE && ((HashInBITS-HashChunkSizeInBITS)==0) && (SWPexist+HSHexist == 2) ) {
3294     REUSE=2;
3295     if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
3296     { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
3297 }
3298
3299 if( ( fp_outRG = fopen( "Leprechaun_64bit.hsh", "rb" ) ) != NULL ) {
3300     if ( REUSE == 2 ) { // REUSE [
3301
3302 #if defined(_WIN32_ENVIRONMENT_)
3303     // 64bit:

```

```

3304 _lseeki64( fileno(fp_outRG), 0L, SEEK_END );
3305 size_inLINESIXFOUR = _telli64( fileno(fp_outRG) );
3306 _lseeki64( fileno(fp_outRG), 0L, SEEK_SET );
3307 #else
3308     // 64bit:
3309     fseeko( fp_outRG, 0L, SEEK_END );
3310     size_inLINESIXFOUR = ftello( fp_outRG );
3311     fseeko( fp_outRG, 0L, SEEK_SET );
3312 #endif /* defined(_WIN32_ENVIRONMENT_) */
3313 printf( "Uploading-n-Reusing 'Leprechaun_64bit.hsh' file: %s bytes\n", _ui64toaKAZEcomma(size_inLINESIXFOUR, l1ToaDigits, 10) );
3314
3315     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.
3316     }
3317     fclose(fp_outRG);
3318 }
3319
3320 // Tag for the swap file is: LEPRECHAUNISH{ASCIIcode26}
3321 // or 14bytes, then when type of the swap is requested:
3322 // D:\_KAZE~1\LEPREC~1>type Leprechaun_64bit.swp
3323 // LEPRECHAUNISH
3324 // D:\_KAZE~1\LEPREC~1>
3325 size_in64_L14 = 1024 * (unsigned long long)Thunderwith + 14;
3326 BufEnd_64 = 0+14;
3327 // 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.
3328
3329 /*
3330 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:
3331
3332 'r'
3333     Open an existing file for reading only.
3334 'w'
3335     Open the file for writing only. If the file already exists, it is truncated to zero length. Otherwise a new file is created.
3336 'a'
3337     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.
3338 'r+'
3339     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.
3340 'w+'
3341     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.
3342 'a+'
3343     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.
3344 */
3345
3346     // 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!
3347     if ( REUSE == 2 ) { // REUSE [
3348         if( ( fp_outRG = fopen( "Leprechaun_64bit.swp", "rb+" ) ) == NULL )
3349             { printf( "Leprechaun: Can't create file 'Leprechaun_64bit.swp'.\n" ); return( 1 ); }
3350
3351         #if defined(_WIN32_ENVIRONMENT_)
3352             // 64bit:
3353             _lseeki64( fileno(fp_outRG), 0L, SEEK_END );
3354             size_inLINESIXFOUR = _telli64( fileno(fp_outRG) );
3355             _lseeki64( fileno(fp_outRG), 0L, SEEK_SET );
3356         #else
3357             // 64bit:
3358             fseeko( fp_outRG, 0L, SEEK_END );
3359             size_inLINESIXFOUR = ftello( fp_outRG );
3360             fseeko( fp_outRG, 0L, SEEK_SET );
3361         #endif /* defined(_WIN32_ENVIRONMENT_) */
3362         printf( "Reusing 'Leprechaun_64bit.swp' file: %s bytes\n", _ui64toaKAZEcomma(size_inLINESIXFOUR, l1ToaDigits, 10) );
3363
3364         fsetpos(fp_outRG, &BufEnd_64); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ???! So DO-IT-OVER.
3365     }
3366     else { // REUSE
3367         if( ( fp_outRG = fopen( "Leprechaun_64bit.swp", "wb+" ) ) == NULL )
3368             { printf( "Leprechaun: Can't create file 'Leprechaun_64bit.swp'.\n" ); return( 1 ); }
3369         printf( "Allocating/ZEROing %s bytes swap file ... ", _ui64toaKAZEcomma(size_in64_L14, l1ToaDigits, 10) );
3370         fsetpos(fp_outRG, &fsetpos_ZERO); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ???! So DO-IT-OVER.
3371         memset(OneckusterZEROES, 0, 1024*4);
3372         for (ThunderwithL64_L14=0; ThunderwithL64_L14 < size_in64_L14/(1024*4); ThunderwithL64_L14++)
3373             fwrite(OneckusterZEROES, 1024*4, 1, fp_outRG);
3374         for (ThunderwithL64_L14=0; ThunderwithL64_L14 < size_in64_L14%(1024*4); ThunderwithL64_L14++)
3375             fwrite(&onechar_ieByte, 1, 1, fp_outRG);
3376         fsetpos(fp_outRG, &fsetpos_ZERO); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ???! So DO-IT-OVER.
3377         fwrite(FileSwapTag, 13, 1, fp_outRG);
3378         fwrite(&EOFcode, 1, 1, fp_outRG);
3379         fsetpos(fp_outRG, &BufEnd_64); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ???! So DO-IT-OVER.
3380         printf( "OK\n" );
3381     } // REUSE ]
3382
3383     } else { // ##### 64bit memory manipulations [
3384         size_in64_L14 = 1024 * (unsigned long long)Thunderwith + 14 + 1 + 64;
3385         printf( "Allocating memory %lUMB ... ", (size_in64_L14>>20)+1 );

```

```

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

```



```

3471     for( i = 0; i < size_inLINESIXFOUR; i++ )
3472 {
3473
3474     // ~~~~~ Buffering fread [
3475     if (workKoffset == -1) {
3476         if (i + 1024*128 < size_inLINESIXFOUR) {
3477             fread( &workK[0], 1, 1024*128, fp_inLINE );
3478             workKoffset = 0;
3479             workbyte = workK[workKoffset];
3480         } else
3481             fread( &workbyte, 1, 1, fp_inLINE );
3482     } else {
3483         workKoffset++;
3484         workbyte = workK[workKoffset];
3485         if (workKoffset == 1024*128 - 1) workKoffset = -1;
3486     }
3487     // ~~~~~ Buffering fread ]
3488
3489     // if( isalpha( workbyte ) )
3490     // {
3491     //     if( wrdlen < 31 )
3492     //     { wrd[ wrdlen ] = tolower( workbyte ); }
3493     //     wrdlen++;
3494     // }
3495
3496     if ( workbyte < 'A' ) // Most characters are under alphabet - only one if
3497     {
3498 Elstupido:
3499         // This fragment is MIRROred: #1 copy [
3500         if (workbyte == 10) {NumberOfLines++;}
3501
3502 // Quadruple! [
3503 // Sliding window for ' wrd ': The incoming string 'a lot of things must' becomes 'a_lot_of_things' and 'lot_of_things_must':
3504
3505 // ain_t_that_a
3506 // didn_t_feel_a
3507 // i_didn_t_feel
3508 // t_feel_a_thing
3509 // t_that_a_cake
3510
3511 // 316
3512 // 00:17:55,859 --> 00:17:58,447
3513 // Ain't that a cake ? I didn't feel a thing !
3514
3515         if ( PLE_words_INITflag == 0 && ( (PLE_words != 0) || (PLE_words == 0 && wrdlen != 0) ) )
3516         if ( workbyte == '.' || workbyte == '!' || workbyte == '?' || workbyte == ':' || workbyte == ';' || workbyte == ','
|| workbyte == '\t' ) {
3517             PLE_words_INITflag = 1;
3518         }
3519 // Quadruple! ]
3520
3521 //r.15fixfix [
3522         if( wrdlen > 31 ) PLE_words_INITflag = 1;
3523 //r.15fixfix ]
3524
3525         //if ( 1 <= wrdlen && wrdlen <= LongestLineInclusive ) // Enforce no word with length greater than 31 with below line
to enter x-lets.
3526         if ( 1 <= wrdlen && wrdlen <= 31 )
3527         {
3528             wrd[ wrdlen ] = 0;
3529             // OTKACHAM: 1<<17-1 gives 65536 i.e. '-' have had high priority than '<<'
3530             //Next line gives error due to mix of '&' and 'double'
3531             if ((WORDcount & ((1<<18)-1)) == 0)
3532             { // _ui64toaKAZEzerocomma(WORDcount, llToaDigits, 10);
3533                 //printf( "word count: %s(%lu/128 done)\r", llToaDigits, ((long long)i*100) / size_inLINESIXFOUR );
3534 //++MeInitchka;
3535 //MeInitchka = MeInitchka % 4;
3536 //if (MeInitchka == 0){ printf( "|; word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEcomma(WORDcount, llToaDigits, 10),
_ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llToaDigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
3537 //if (MeInitchka == 1){ printf( "/; word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEcomma(WORDcount, llToaDigits, 10),
_ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llToaDigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
3538 //if (MeInitchka == 2){ printf( "-; word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEcomma(WORDcount, llToaDigits, 10),
_ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llToaDigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
3539 //if (MeInitchka == 3){ printf( "\\; word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEcomma(WORDcount, llToaDigits, 10),
_ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llToaDigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
3540 MeInitchka = MeInitchka & 3; // 0 1 2 3: 00 01 10 11
3541         (void) time(&t4);
3542         if (t4 <= t1) {t4 = t1; t4++;}
3543 printf( "%s; %sP/s; Phrase count: %s of them %s distinct; Done: %lu/64\r", Auberge[MeInitchka++], _ui64toaKAZEzerocomma(WORDcount/((int) t4-
t1), llToaDigits3, 10)+(26-10), _ui64toaKAZEcomma(WORDcount, llToaDigits, 10), _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct,
llToaDigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR );
3544         }
3545
3546 //14+++ [
3547 PLE_words++;
3548
3549 #ifdef singleton
3550         PLE_words_INITflag = 1;

```

```

3551 #endif
3552 #ifdef doubleton
3553 if (PLE_words == 1)
3554     strcpy( wrd1st, wrd );
3555 else if (PLE_words == 2) {
3556     strcpy( wrd2nd, wrd );
3557     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+1; // '_'
3558     //wrdlen = strlen(wrd);
3559     //if ( wrdlen <= 31 ) {
3560         if ( wrdlen <= LongestLineInclusive ) {
3561             strcpy(wrd, wrd1st);
3562             strcat(wrd, DelimiterUnderscore);
3563             strcat(wrd, wrd2nd);
3564         }
3565     }
3566 else {
3567     PLE_words = 2;
3568     strcpy( wrd1st, wrd2nd );
3569     strcpy( wrd2nd, wrd );
3570     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+1; // '_'
3571     //wrdlen = strlen(wrd);
3572     //if ( wrdlen <= 31 ) {
3573         if ( wrdlen <= LongestLineInclusive ) {
3574             strcpy(wrd, wrd1st);
3575             strcat(wrd, DelimiterUnderscore);
3576             strcat(wrd, wrd2nd);
3577         }
3578     }
3579 #endif
3580 #ifdef tripleton
3581 if (PLE_words == 1)
3582     strcpy( wrd1st, wrd );
3583 else if (PLE_words == 2)
3584     strcpy( wrd2nd, wrd );
3585 else if (PLE_words == 3) {
3586     strcpy( wrd3rd, wrd );
3587     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+1+1; // '_''_'
3588     //wrdlen = strlen(wrd);
3589     //if ( wrdlen <= 31 ) {
3590         if ( wrdlen <= LongestLineInclusive ) {
3591             strcpy(wrd, wrd1st);
3592             strcat(wrd, DelimiterUnderscore);
3593             strcat(wrd, wrd2nd);
3594             strcat(wrd, DelimiterUnderscore);
3595             strcat(wrd, wrd3rd);
3596         }
3597     }
3598 else {
3599     PLE_words = 3;
3600     strcpy( wrd1st, wrd2nd );
3601     strcpy( wrd2nd, wrd3rd );
3602     strcpy( wrd3rd, wrd );
3603     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+1+1; // '_''_'
3604     //wrdlen = strlen(wrd);
3605     //if ( wrdlen <= 31 ) {
3606         if ( wrdlen <= LongestLineInclusive ) {
3607             strcpy(wrd, wrd1st);
3608             strcat(wrd, DelimiterUnderscore);
3609             strcat(wrd, wrd2nd);
3610             strcat(wrd, DelimiterUnderscore);
3611             strcat(wrd, wrd3rd);
3612         }
3613     }
3614 #endif
3615 #ifdef quadruplet
3616 // Quadruple! [
3617 if (PLE_words == 1)
3618     strcpy( wrd1st, wrd );
3619 else if (PLE_words == 2)
3620     strcpy( wrd2nd, wrd );
3621 else if (PLE_words == 3)
3622     strcpy( wrd3rd, wrd );
3623 else if (PLE_words == 4) {
3624     strcpy( wrd4th, wrd );
3625     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+1+1+1; // '_''''_'
3626     //wrdlen = strlen(wrd);
3627     //if ( wrdlen <= 31 ) {
3628         if ( wrdlen <= LongestLineInclusive ) {
3629             strcpy(wrd, wrd1st);
3630             strcat(wrd, DelimiterUnderscore);
3631             strcat(wrd, wrd2nd);
3632             strcat(wrd, DelimiterUnderscore);
3633             strcat(wrd, wrd3rd);
3634             strcat(wrd, DelimiterUnderscore);
3635             strcat(wrd, wrd4th);
3636         }
3637     }
3638 else {

```

```

3639     PLE_words = 4;
3640     strcpy( wrd1st, wrd2nd );
3641     strcpy( wrd2nd, wrd3rd );
3642     strcpy( wrd3rd, wrd4th );
3643     strcpy( wrd4th, wrd );
3644     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+1+1+1; // '_''_''_'
3645     //wrdlen = strlen(wrd);
3646     //if ( wrdlen <= 31 ) {
3647         if ( wrdlen <= LongestLineInclusive ) {
3648         strcpy(wrd, wrd1st);
3649         strcat(wrd, DelimiterUnderscore);
3650         strcat(wrd, wrd2nd);
3651         strcat(wrd, DelimiterUnderscore);
3652         strcat(wrd, wrd3rd);
3653         strcat(wrd, DelimiterUnderscore);
3654         strcat(wrd, wrd4th);
3655         }
3656     }
3657 // Quadruple! ]
3658 #endif
3659 #ifdef quintuplet
3660 if (PLE_words == 1)
3661     strcpy( wrd1st, wrd );
3662 else if (PLE_words == 2)
3663     strcpy( wrd2nd, wrd );
3664 else if (PLE_words == 3)
3665     strcpy( wrd3rd, wrd );
3666 else if (PLE_words == 4)
3667     strcpy( wrd4th, wrd );
3668 else if (PLE_words == 5) {
3669     strcpy( wrd5th, wrd );
3670     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+1+1+1+1; // '_''_''_''_'
3671     //wrdlen = strlen(wrd);
3672     //if ( wrdlen <= 31 ) {
3673         if ( wrdlen <= LongestLineInclusive ) {
3674         strcpy(wrd, wrd1st);
3675         strcat(wrd, DelimiterUnderscore);
3676         strcat(wrd, wrd2nd);
3677         strcat(wrd, DelimiterUnderscore);
3678         strcat(wrd, wrd3rd);
3679         strcat(wrd, DelimiterUnderscore);
3680         strcat(wrd, wrd4th);
3681         strcat(wrd, DelimiterUnderscore);
3682         strcat(wrd, wrd5th);
3683         }
3684     }
3685 else {
3686     PLE_words = 5;
3687     strcpy( wrd1st, wrd2nd );
3688     strcpy( wrd2nd, wrd3rd );
3689     strcpy( wrd3rd, wrd4th );
3690     strcpy( wrd4th, wrd5th );
3691     strcpy( wrd5th, wrd );
3692     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+1+1+1+1; // '_''_''_''_'
3693     //wrdlen = strlen(wrd);
3694     //if ( wrdlen <= 31 ) {
3695         if ( wrdlen <= LongestLineInclusive ) {
3696         strcpy(wrd, wrd1st);
3697         strcat(wrd, DelimiterUnderscore);
3698         strcat(wrd, wrd2nd);
3699         strcat(wrd, DelimiterUnderscore);
3700         strcat(wrd, wrd3rd);
3701         strcat(wrd, DelimiterUnderscore);
3702         strcat(wrd, wrd4th);
3703         strcat(wrd, DelimiterUnderscore);
3704         strcat(wrd, wrd5th);
3705         }
3706     }
3707 #endif
3708 #ifdef sextuplet
3709 if (PLE_words == 1)
3710     strcpy( wrd1st, wrd );
3711 else if (PLE_words == 2)
3712     strcpy( wrd2nd, wrd );
3713 else if (PLE_words == 3)
3714     strcpy( wrd3rd, wrd );
3715 else if (PLE_words == 4)
3716     strcpy( wrd4th, wrd );
3717 else if (PLE_words == 5)
3718     strcpy( wrd5th, wrd );
3719 else if (PLE_words == 6) {
3720     strcpy( wrd6th, wrd );
3721     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+1+1+1+1+1; // '_''_''_''_''_'
3722     //wrdlen = strlen(wrd);
3723     //if ( wrdlen <= 31 ) {
3724         if ( wrdlen <= LongestLineInclusive ) {
3725         strcpy(wrd, wrd1st);
3726         strcat(wrd, DelimiterUnderscore);

```

```

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

```

```

3813 strcat(wrd, wrd2nd);
3814 strcat(wrd, DelimiterUnderscore);
3815 strcat(wrd, wrd3rd);
3816 strcat(wrd, DelimiterUnderscore);
3817 strcat(wrd, wrd4th);
3818 strcat(wrd, DelimiterUnderscore);
3819 strcat(wrd, wrd5th);
3820 strcat(wrd, DelimiterUnderscore);
3821 strcat(wrd, wrd6th);
3822 strcat(wrd, DelimiterUnderscore);
3823 strcat(wrd, wrd7th);
3824     }
3825 }
3826 #endif
3827 #ifdef octupleton
3828 if (PLE_words == 1)
3829     strcpy( wrd1st, wrd );
3830 else if (PLE_words == 2)
3831     strcpy( wrd2nd, wrd );
3832 else if (PLE_words == 3)
3833     strcpy( wrd3rd, wrd );
3834 else if (PLE_words == 4)
3835     strcpy( wrd4th, wrd );
3836 else if (PLE_words == 5)
3837     strcpy( wrd5th, wrd );
3838 else if (PLE_words == 6)
3839     strcpy( wrd6th, wrd );
3840 else if (PLE_words == 7)
3841     strcpy( wrd7th, wrd );
3842 else if (PLE_words == 8) {
3843     strcpy( wrd8th, wrd );
3844     wrdlen =
        strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+strlen(wrd8th)+1+1+1+1+1+1; //
        - - - - - //wrdlen = strlen(wrd);
3845     //if ( wrdlen <= 31 ) {
3846         if ( wrdlen <= LongestLineInclusive ) {
3847             strcpy(wrd, wrd1st);
3848             strcat(wrd, DelimiterUnderscore);
3849             strcat(wrd, wrd2nd);
3850             strcat(wrd, DelimiterUnderscore);
3851             strcat(wrd, wrd3rd);
3852             strcat(wrd, DelimiterUnderscore);
3853             strcat(wrd, wrd4th);
3854             strcat(wrd, DelimiterUnderscore);
3855             strcat(wrd, wrd5th);
3856             strcat(wrd, DelimiterUnderscore);
3857             strcat(wrd, wrd6th);
3858             strcat(wrd, DelimiterUnderscore);
3859             strcat(wrd, wrd7th);
3860             strcat(wrd, DelimiterUnderscore);
3861             strcat(wrd, wrd8th);
3862         }
3863     }
3864 }
3865 else {
3866     PLE_words = 8;
3867     strcpy( wrd1st, wrd2nd );
3868     strcpy( wrd2nd, wrd3rd );
3869     strcpy( wrd3rd, wrd4th );
3870     strcpy( wrd4th, wrd5th );
3871     strcpy( wrd5th, wrd6th );
3872     strcpy( wrd6th, wrd7th );
3873     strcpy( wrd7th, wrd8th );
3874     strcpy( wrd8th, wrd );
3875     wrdlen =
        strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+strlen(wrd8th)+1+1+1+1+1+1; //
        - - - - - //wrdlen = strlen(wrd);
3876     //if ( wrdlen <= 31 ) {
3877         if ( wrdlen <= LongestLineInclusive ) {
3878             strcpy(wrd, wrd1st);
3879             strcat(wrd, DelimiterUnderscore);
3880             strcat(wrd, wrd2nd);
3881             strcat(wrd, DelimiterUnderscore);
3882             strcat(wrd, wrd3rd);
3883             strcat(wrd, DelimiterUnderscore);
3884             strcat(wrd, wrd4th);
3885             strcat(wrd, DelimiterUnderscore);
3886             strcat(wrd, wrd5th);
3887             strcat(wrd, DelimiterUnderscore);
3888             strcat(wrd, wrd6th);
3889             strcat(wrd, DelimiterUnderscore);
3890             strcat(wrd, wrd7th);
3891             strcat(wrd, DelimiterUnderscore);
3892             strcat(wrd, wrd8th);
3893         }
3894     }
3895 }
3896 #endif

```

```

3897 #ifndef nonpleton
3898 if (PLE_words == 1)
3899     strcpy( wrd1st, wrd );
3900 else if (PLE_words == 2)
3901     strcpy( wrd2nd, wrd );
3902 else if (PLE_words == 3)
3903     strcpy( wrd3rd, wrd );
3904 else if (PLE_words == 4)
3905     strcpy( wrd4th, wrd );
3906 else if (PLE_words == 5)
3907     strcpy( wrd5th, wrd );
3908 else if (PLE_words == 6)
3909     strcpy( wrd6th, wrd );
3910 else if (PLE_words == 7)
3911     strcpy( wrd7th, wrd );
3912 else if (PLE_words == 8)
3913     strcpy( wrd8th, wrd );
3914 else if (PLE_words == 9) {
3915     strcpy( wrd9th, wrd );
3916     wrdlen =
        strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+strlen(wrd8th)+strlen(wrd9th)+1+1+1+1
        +1+1+1+1; // '_ _ _ _ _'
3917     //wrdlen = strlen(wrd);
3918     //if ( wrdlen <= 31 ) {
3919         if ( wrdlen <= LongestLineInclusive ) {
3920             strcpy(wrd, wrd1st);
3921             strcat(wrd, DelimiterUnderscore);
3922             strcat(wrd, wrd2nd);
3923             strcat(wrd, DelimiterUnderscore);
3924             strcat(wrd, wrd3rd);
3925             strcat(wrd, DelimiterUnderscore);
3926             strcat(wrd, wrd4th);
3927             strcat(wrd, DelimiterUnderscore);
3928             strcat(wrd, wrd5th);
3929             strcat(wrd, DelimiterUnderscore);
3930             strcat(wrd, wrd6th);
3931             strcat(wrd, DelimiterUnderscore);
3932             strcat(wrd, wrd7th);
3933             strcat(wrd, DelimiterUnderscore);
3934             strcat(wrd, wrd8th);
3935             strcat(wrd, DelimiterUnderscore);
3936             strcat(wrd, wrd9th);
3937         }
3938     }
3939 else {
3940     PLE_words = 9;
3941     strcpy( wrd1st, wrd2nd );
3942     strcpy( wrd2nd, wrd3rd );
3943     strcpy( wrd3rd, wrd4th );
3944     strcpy( wrd4th, wrd5th );
3945     strcpy( wrd5th, wrd6th );
3946     strcpy( wrd6th, wrd7th );
3947     strcpy( wrd7th, wrd8th );
3948     strcpy( wrd8th, wrd9th );
3949     strcpy( wrd9th, wrd );
3950     wrdlen =
        strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+strlen(wrd8th)+strlen(wrd9th)+1+1+1+1
        +1+1+1+1; // '_ _ _ _ _'
3951     //wrdlen = strlen(wrd);
3952     //if ( wrdlen <= 31 ) {
3953         if ( wrdlen <= LongestLineInclusive ) {
3954             strcpy(wrd, wrd1st);
3955             strcat(wrd, DelimiterUnderscore);
3956             strcat(wrd, wrd2nd);
3957             strcat(wrd, DelimiterUnderscore);
3958             strcat(wrd, wrd3rd);
3959             strcat(wrd, DelimiterUnderscore);
3960             strcat(wrd, wrd4th);
3961             strcat(wrd, DelimiterUnderscore);
3962             strcat(wrd, wrd5th);
3963             strcat(wrd, DelimiterUnderscore);
3964             strcat(wrd, wrd6th);
3965             strcat(wrd, DelimiterUnderscore);
3966             strcat(wrd, wrd7th);
3967             strcat(wrd, DelimiterUnderscore);
3968             strcat(wrd, wrd8th);
3969             strcat(wrd, DelimiterUnderscore);
3970             strcat(wrd, wrd9th);
3971         }
3972     }
3973 #endif
3974 #ifdef decupleton
3975 if (PLE_words == 1)
3976     strcpy( wrd1st, wrd );
3977 else if (PLE_words == 2)
3978     strcpy( wrd2nd, wrd );
3979 else if (PLE_words == 3)
3980     strcpy( wrd3rd, wrd );

```



```

4065         if ( ( PLE_words == 2 ) && ( 5 <= wrdlen ) && ( wrdlen <= 41 ) ) {
4066 #endif
4067 #ifdef tripleteon
4068         if ( ( PLE_words == 3 ) && ( 9 <= wrdlen ) && ( wrdlen <= 41 ) ) {
4069 #endif
4070 #ifdef quadrupleton
4071         if ( ( PLE_words == 4 ) && ( 13 <= wrdlen ) && ( wrdlen <= 51 ) ) {
4072 #endif
4073 #ifdef quintupleton
4074         if ( ( PLE_words == 5 ) && ( 17 <= wrdlen ) && ( wrdlen <= 61 ) ) {
4075 #endif
4076 #ifdef sextupleton
4077         if ( ( PLE_words == 6 ) && ( 21 <= wrdlen ) && ( wrdlen <= 71 ) ) {
4078 #endif
4079 #ifdef septupleton
4080         if ( ( PLE_words == 7 ) && ( 25 <= wrdlen ) && ( wrdlen <= 81 ) ) {
4081 #endif
4082 #ifdef octupleton
4083         if ( ( PLE_words == 8 ) && ( 29 <= wrdlen ) && ( wrdlen <= 91 ) ) {
4084 #endif
4085 #ifdef nonupleton
4086         if ( ( PLE_words == 9 ) && ( 33 <= wrdlen ) && ( wrdlen <= 101 ) ) {
4087 #endif
4088 #ifdef decupleton
4089         if ( ( PLE_words == 10 ) && ( 37 <= wrdlen ) && ( wrdlen <= 111 ) ) {
4090 #endif
4091 //14+++ ]
4092 WORDcount++;
4093 if (BSTorBtree < 2) {
4094
4095     LetterOffset = (int)( wrd[0] - 'a' ) * 31 + (wrdlen-1); // 0..805
4096     //BufStart = pointerflush + LetterOffset * LetterBuffer; // OLD
4097
4098     BufStart = pointerflush + (int)( wrd[0] - 'a' ) * WHOLEletter_BufferSize + OffsetsInBuffer[wrdlen-1];
4099     // Above line and Below line are equal
4100     //BufStart = pointerflush + (LetterOffset / 31) * WHOLEletter_BufferSize + OffsetsInBuffer[LetterOffset % 31];
4101
4102     //Slot = KuxHash3plus(wrd)<<2; //13++
4103     //Slot = FNV1A_Hash_SHIFTless_XORless(wrd)<<2; //13+++
4104     //Slot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13++++
4105     //Slot = FNV1A_Hash_4_OCTETS_31(wrd, wrdlen>>2)<<2; //13+++++
4106 /*
4107 if (wrdlen<=19) // 4x4+3=19
4108     Slot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13++++
4109 else
4110     // 2x8+4=20 i.e. first contains 5 clashes
4111     Slot = FNV1A_Hash_8_OCTETS(wrd, wrdlen>>3)<<2; //13+++++
4112 //if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
4113 //    Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>2, 2)<<2; //13+++++
4114 //else
4115 //    // 2x8+4=20 i.e. first contains 6 clashes
4116 //    Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>3, 3)<<2; //13+++++
4117 //Slot = FNV1A_Hash_8_OCTETS(wrd, wrdlen>>3)<<2; //13_7p
4118 //Slot = HashFNV1A_unrolled_Final(wrd, wrdlen)<<2; //13_7p
4119 //Slot = HashAlfalpa_HALF(wrd, wrdlen)<<2; //13_7p
4120 //Slot = HashAlfalpa(wrd, wrdlen)<<2; //13_7p
4121 //    Slot = Sixtinsensitive(wrd, wrdlen)<<2; //13_7p
4122 //    Slot = FNV1A_Hash_Jesteress(wrd, wrdlen)<<2; //13_7p
4123 //    Slot = FNV1A_Hash_Jester(wrd, wrdlen)<<2; //13_7p
4124
4125 /*
4126 hashAlfalpa = 7;
4127 for(iAlfalpa = 0; iAlfalpa < (wrdlen & -2); iAlfalpa += 2) {
4128     hashAlfalpa = (17+9) * ((17+9) * hashAlfalpa + (wrd[iAlfalpa])) + (wrd[iAlfalpa+1]);
4129 }
4130 if(wrdlen & 1)
4131     hashAlfalpa = (17+9) * hashAlfalpa + (wrd[wrdlen-1]);
4132
4133     Slot = (( hashAlfalpa ^ (hashAlfalpa >> 16) ) & 8191)<<2; //13_7p
4134 */
4135
4136     memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
4137 //; Line 917
4138 // mov     edx, DWORD PTR [eax+ebp]
4139 // add     esp, 4
4140 // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
4141 //PseudoLinkedPointer = (unsigned long)*(long *) (BufStart+Slot);
4142 //; Line 919
4143 // mov     edx, DWORD PTR [eax+ebp]
4144 // add     esp, 4
4145 //         while (count--) {
4146 //             *(char *)dst = *(char *)src;
4147 //             dst = (char *)dst + 1;
4148 //             src = (char *)src + 1;
4149 //         }
4150
4151 if (BSTorBtree == 0)
4152 {

```



```

4153 // @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ BST fragment [
4154 if (PseudoLinkedPointer == 0) // means EMPTY-SLOT
4155 {
4156 //if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 < (GRMBLhill[(int)wrdlen] * LetterBuffer)/31 ) // OLD slower
4157 if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST instead
of LL
4158 {
4159 memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
4160 //; Line 932
4161 // mov     DWORD PTR [eax+ebp], esi
4162 // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
4163 //*(long *) (BufStart+Slot) = *(long *)&bufend[LetterOffset];
4164 //; Line 936
4165 // mov     DWORD PTR [eax+ebp], esi
4166
4167 // Below 3 lines are commented due to experiment below malloc which shows that allocated memory is ZEROed.
4168 //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not: Means PseudoLinkedPointerL =
0
4169 //; Line 940
4170 // mov     ecx, DWORD PTR [ebp+32768]
4171 // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
4172 //*(long *)bufend[LetterOffset] = *(long *)&BufStart[NumberOfSLOTS*4];
4173 //; Line 944
4174 // mov     ecx, DWORD PTR [ebp+32768]
4175 //bufend[LetterOffset] = bufend[LetterOffset] + 4;
4176 //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not: Means PseudoLinkedPointerR =
0
4177 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4; // + 4 due to above commenting
4178 memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4179 //bufNowPS[LetterOffset][Slot]++; // ?! crashes
4180 bufend[LetterOffset] = bufend[LetterOffset] + wrdlen;
4181 if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4182 }
4183 else
4184 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4185 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4186 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, llToaDigits, 10) );
4187 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llToaDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, llToaDigits2, 10) );
4188 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4189 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4190 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4191 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDS into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llToaDigits,
10) );
4192 for( k = 1; k < 32; k++ )
4193 { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s utilization\n", _ui64toaKAZEcomma(k, llToaDigits, 10)+(26-
2), _ui64toaKAZEcomma((MAXusedBuffer[k]>>10)+1, llToaDigits2, 10)+(26-5), _ui64toaKAZEcomma((((GRMBLhill[(int)k] *
LetterBuffer)/31)>>10)+1, llToaDigits3, 10)+(26-5), _ui64toaKAZEcomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLhill[(int)k] *
LetterBuffer)/31), llToaDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
4194 }
4195 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4196 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4197 return( 1 );
4198 }
4199 }
4200 else // means USED-SLOT
4201 { FoundInLinkedList = 0;
4202 while (PseudoLinkedPointer != 0 && FoundInLinkedList == 0)
4203 {
4204 if (memcmp(PseudoLinkedPointer+4+4, wrd, wrdlen) == 0)
4205 // while ( --count && *(char *)buf1 == *(char *)buf2 ) {
4206 // buf1 = (char *)buf1 + 1;
4207 // buf2 = (char *)buf2 + 1;
4208 // }
4209 // return( *((unsigned char *)buf1) - *((unsigned char *)buf2) );
4210 { FoundInLinkedList = 1;
4211 }
4212 else // i.e < or >
4213 {
4214 if (memcmp(PseudoLinkedPointer+4+4, wrd, wrdlen) > 0)
4215 memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
4216 else
4217 {
4218 PseudoLinkedPointer = PseudoLinkedPointer + 4;
4219 memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
4220 }
4221 }
4222 if (PseudoLinkedPointerNEW == 0)
4223 {
4224 //if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 < (GRMBLhill[(int)wrdlen] * LetterBuffer)/31 ) // OLD slower
4225 if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] )
4226 { memcpy( PseudoLinkedPointer, &bufend[LetterOffset], 4 );
4227 // Below 3 lines are commented due to experiment below malloc which shows that allocated memory is ZEROed.
4228 //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not
4229 //bufend[LetterOffset] = bufend[LetterOffset] + 4;
4230 //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not
4231 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4; // + 4 due to above commenting

```

```

4232 memcp( bufend[LetterOffset], wrd, wrdlen ; WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4233 //bufNowPS[LetterOffset][Slot]++; // ?! crashes
4234 bufend[LetterOffset] = bufend[LetterOffset] + wrdlen;
4235 if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4236 }
4237 else
4238 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4239 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4240 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, 11ToaDigits, 10) );
4241 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, 11ToaDigits2, 10) );
4242 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4243 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4244 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4245 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, 11ToaDigits,
10) );
4246 for( k = 1; k < 32; k++ )
4247 { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s utilization\n", _ui64toaKAZEcomma(k, 11ToaDigits, 10)+(26-
2), _ui64toaKAZEcomma((MAXusedBuffer[k]>>10)+1, 11ToaDigits2, 10)+(26-5), _ui64toaKAZEcomma((((GRMBLh11[(int)k] *
LetterBuffer)/31)>>10)+1, 11ToaDigits3, 10)+(26-5), _ui64toaKAZEcomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLh11[(int)k] *
LetterBuffer)/31), 11ToaDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
4248 }
4249 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4250 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4251 return( 1 );
4252 }
4253 }
4254 PseudoLinkedPointer = PseudoLinkedPointerNEW;
4255 }
4256 WORDcountAttemptsToPut++;
4257 } // while
4258 }
4259 // @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ BST fragment ]
4260 } else
4261 {
4262 // ##### B-tree order 3 fragment [
4263 //
4264 // LEAF structure: [LeftPointer][MiddlePointer][RightPointer][Leftword][Rightword]
4265 //                  4bytes      4bytes      4bytes      wrdlen      wrdlen
4266 //                                     *          *
4267 // ALL B-tree order 3 fragment consists of 3 sub-fragments:
4268 // 1] Search 2] if Search failed Trasirascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search 3] Insert Iterative
4269
4270 // 1] Search [ _____1407 line in C - see below: whole Search in assembler_____
4271 if (PseudoLinkedPointer == 0) // means EMPTY-SLOT
4272 {
4273 if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST
instead of LL; + more(see LEAF)
4274 {
4275 memcp( BufStart+Slot, &bufend[LetterOffset], 4 );
4276 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
4277 memcp( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4278 bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
4279 if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4280 }
4281 else
4282 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4283 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4284 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, 11ToaDigits, 10) );
4285 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, 11ToaDigits2, 10) );
4286 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4287 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4288 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4289 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, 11ToaDigits, 10)
);
4290 for( k = 1; k < 32; k++ )
4291 { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s utilization\n", _ui64toaKAZEcomma(k, 11ToaDigits, 10)+(26-
2), _ui64toaKAZEcomma((MAXusedBuffer[k]>>10)+1, 11ToaDigits2, 10)+(26-5), _ui64toaKAZEcomma((((GRMBLh11[(int)k] *
LetterBuffer)/31)>>10)+1, 11ToaDigits3, 10)+(26-5), _ui64toaKAZEcomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLh11[(int)k] *
LetterBuffer)/31), 11ToaDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
4292 }
4293 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4294 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4295 return( 1 );
4296 }
4297 }
4298 FoundInLinkedList = 1;
4299 }
4300 else // means USED-SLOT
4301 { FoundInLinkedList = 0;
4302 while (PseudoLinkedPointer != 0 && FoundInLinkedList == 0)
4303 {
4304 // ***** 'P W P' section [
4305 // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
4306 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
4307 // here ALWAYS LW exists: no need for existence check - line below

```

```

4307 // if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
4308 if (memcmp(PseudoLinkedPointer+4+4+4, wrd, wrdlen) > 0) // go LP
4309 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 0, 4 ); //LP
4310   PseudoLinkedPointer = PseudoLinkedPointerNEW;
4311 }
4312 else if (memcmp(PseudoLinkedPointer+4+4+4, wrd, wrdlen) < 0) // go RP or MP
4313 { // RW existence check - line below:
4314   if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // RW exists
4315     { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
4316 // ++++++
4317 // ***** 'P W P' section 2 [
4318 // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
4319 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
4320 // here ALWAYS RW exists: no need for existence check - line below
4321 // if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
4322 if (memcmp(PseudoLinkedPointer+4+4+4+wrdlen, wrd, wrdlen) > 0) // go MP
4323 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
4324   PseudoLinkedPointer = PseudoLinkedPointerNEW;
4325 }
4326 else if (memcmp(PseudoLinkedPointer+4+4+4+wrdlen, wrd, wrdlen) < 0) // go RP
4327 { // No ?w after RW - go RP
4328   memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
4329   PseudoLinkedPointer = PseudoLinkedPointerNEW;
4330 }
4331 else FoundInLinkedList = 1; // wrd is RW
4332   WORDcountAttemptsToPut++;
4333 // ***** 'P W P' section 2 ]
4334 // ++++++
4335 }
4336 else // RW empty - go MP
4337 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
4338   PseudoLinkedPointer = PseudoLinkedPointerNEW;
4339 }
4340 }
4341 else FoundInLinkedList = 1; // wrd is LW
4342   WORDcountAttemptsToPut++;
4343 // ***** 'P W P' section ]
4344 } // while
4345   WORDcountAttemptsToPut--; // - 1 due to BST way of counting i.e. direct hash hit is not counted only successors
4346 }
4347 // 1] Search ] _____1484 line in C - see below: whole search in assembler_____
4348
4349 /*
4350 ; Line 1397
4351 jmp $L2139
4352 $L2042:
4353 ; Line 1408
4354 test edx, edx
4355 jne SHORT $L2110
4356 ; Line 1410
4357 mov ecx, DWORD PTR _bufend$[esp+esi*4+892340]
4358 mov edi, DWORD PTR _GRMBLFoolAgain$[esp+ebx*4+892340]
4359 lea edx, DWORD PTR _bufend$[esp+esi*4+892340]
4360 lea esi, DWORD PTR [ebx+ebx+12]
4361 sub esi, ebp
4362 add esi, ecx
4363 cmp esi, edi
4364 mov DWORD PTR tv4122[esp+892340], edx
4365 jae $L2113
4366 ; Line 1412
4367 mov DWORD PTR [eax+ebp], ecx
4368 ; Line 1413
4369 lea eax, DWORD PTR [ecx+12]
4370 ; Line 1436
4371 jmp $L2749
4372 $L2110:
4373 ; Line 1437
4374 mov DWORD PTR _FoundInLinkedList$[esp+892340], 0
4375 npad11
4376 $L2141:
4377 ; Line 1438
4378 mov eax, DWORD PTR _FoundInLinkedList$[esp+892340]
4379 test eax, eax
4380 jne $L2142
4381 ; Line 1445
4382 lea ebp, DWORD PTR [edx+12]
4383 mov ecx, ebx
4384 lea edi, DWORD PTR _wrd$[esp+892340]
4385 mov esi, ebp
4386 xor eax, eax
4387 repe cmpsb
4388 je SHORT $L2682
4389 sbb eax, eax
4390 sbb eax, -1
4391 $L2682:
4392 test eax, eax
4393 jle SHORT $L2143
4394 ; Line 1447

```

```

4395 mov edx, DWORD PTR [edx]
4396 ; Line 1449
4397 jmp $L2153
4398 $L2143:
4399 mov ecx, ebx
4400 lea edi, DWORD PTR _word$[esp+892340]
4401 mov esi, ebp
4402 xor eax, eax
4403 repe cmpsb
4404 je SHORT $L2640
4405 sbb eax, eax
4406 sbb eax, -1
4407 $L2640:
4408 test eax, eax
4409 jge SHORT $L2145
4410 ; Line 1451
4411 mov cl, BYTE PTR [edx+ebx+12]
4412 test cl, cl
4413 lea eax, DWORD PTR [edx+ebx+12]
4414 je SHORT $L2147
4415 ; Line 1459
4416 mov ecx, ebx
4417 lea edi, DWORD PTR _word$[esp+892340]
4418 mov esi, eax
4419 xor ebp, ebp
4420 repe cmpsb
4421 je SHORT $L2695
4422 sbb ebp, ebp
4423 sbb ebp, -1
4424 $L2695:
4425 test ebp, ebp
4426 jle SHORT $L2148
4427 ; Line 1461
4428 mov edx, DWORD PTR [edx+4]
4429 ; Line 1463
4430 jmp SHORT $L2151
4431 $L2148:
4432 mov esi, eax
4433 mov ecx, ebx
4434 lea edi, DWORD PTR _word$[esp+892340]
4435 xor eax, eax
4436 repe cmpsb
4437 je SHORT $L2642
4438 sbb eax, eax
4439 sbb eax, -1
4440 $L2642:
4441 test eax, eax
4442 jge SHORT $L2150
4443 ; Line 1466
4444 mov edx, DWORD PTR [edx+8]
4445 ; Line 1468
4446 jmp SHORT $L2151
4447 $L2150:
4448 mov DWORD PTR _FoundInLinkedList$[esp+892340], 1
4449 $L2151:
4450 ; Line 1469
4451 mov ecx, DWORD PTR _wordcountAttemptsToPut$[esp+892340]
4452 mov eax, DWORD PTR _wordcountAttemptsToPut$[esp+892344]
4453 add ecx, 1
4454 adc eax, 0
4455 mov DWORD PTR _wordcountAttemptsToPut$[esp+892340], ecx
4456 mov DWORD PTR _wordcountAttemptsToPut$[esp+892344], eax
4457 ; Line 1473
4458 jmp SHORT $L2153
4459 $L2147:
4460 ; Line 1475
4461 mov edx, DWORD PTR [edx+4]
4462 ; Line 1478
4463 jmp SHORT $L2153
4464 $L2145:
4465 mov DWORD PTR _FoundInLinkedList$[esp+892340], 1
4466 $L2153:
4467 ; Line 1479
4468 mov esi, DWORD PTR _wordcountAttemptsToPut$[esp+892340]
4469 mov ecx, DWORD PTR _wordcountAttemptsToPut$[esp+892344]
4470 add esi, 1
4471 adc ecx, 0
4472 test edx, edx
4473 mov DWORD PTR _wordcountAttemptsToPut$[esp+892340], esi
4474 mov DWORD PTR _wordcountAttemptsToPut$[esp+892344], ecx
4475 jne $L2141
4476 $L2142:
4477 ; Line 1482
4478 mov edx, DWORD PTR _wordcountAttemptsToPut$[esp+892340]
4479 mov ecx, DWORD PTR _wordcountAttemptsToPut$[esp+892344]
4480 or eax, -1
4481 add edx, eax
4482 adc ecx, eax

```

```

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

```

```

4571 // ALlocate NEW LEAF:
4572 if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wordlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wordlen] ) // +4 more for BST
instead of LL; + more(see LEAF)
4573 {
4574     memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
4575     bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
4576     bufend[LetterOffset] = bufend[LetterOffset] + 2*wordlen;
4577     if (MAXusedBuffer[wordlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wordlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4578 }
4579 else
4580 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4581 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4582 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, llToaDigits, 10) );
4583 fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDCOUNT, llToaDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDCOUNTDISTINCT, llToaDigits2, 10) );
4584 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4585 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4586 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4587 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDS into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDCOUNTATTEMPTSTOPUT, llToaDigits, 10)
);
4588 for( k = 1; k < 32; k++ )
4589 { fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s utilization\n", _ui64toaKAZEzerocomma(k, llToaDigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, llToaDigits2, 10)+(26-5), _ui64toaKAZEzerocomma((((GRMBLhill[(int)k] *
LetterBuffer)/31)>>10)+1, llToaDigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLhill[(int)k] *
LetterBuffer)/31), llToaDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
4590 }
4591 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4592 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4593 return( 1 );
4594 }
4595 if (POffsetInLEAF == 0) // wrd < LW
4596 {
4597     memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
4598     memcpy( PseudoLinkedPointer+4+4+4, wrd, wrdlen ); // wrd go to OLD LEAF
4599     memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
4600     *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
4601 }
4602 if (POffsetInLEAF == 4) // LW < wrd < RW
4603 {
4604     memcpy( wrdUP, wrd, wrdlen ); // wrd up
4605     memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
4606     *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
4607 }
4608 if (POffsetInLEAF == 8) // wrd > RW
4609 {
4610     memcpy( wrdUP, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW up
4611     *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
4612     memcpy( PseudoLinkedPointerNEW+4+4+4, wrd, wrdlen ); // wrd go to NEW LEAF
4613 }
4614 }
4615 else // If LEAF is not full: Case #1
4616 { SplitOccured = 0; WORDCOUNTDISTINCT++; bufNumberOfWords[LetterOffset]++;
4617 if (POffsetInLEAF == 0) // wrd < [LW] so [LW] -> [ ] [LW] -> [wrd] [LW]
4618 {
4619     memcpy( PseudoLinkedPointer+4+4+4+wrdlen, PseudoLinkedPointer+4+4+4, wrdlen );
4620     memcpy( PseudoLinkedPointer+4+4+4, wrd, wrdlen );
4621 }
4622 if (POffsetInLEAF == 4) // wrd > [LW] so [LW] -> [LW] [wrd]
4623 {
4624     memcpy( PseudoLinkedPointer+4+4+4+wrdlen, wrd, wrdlen );
4625 }
4626 }
4627 }
4628 if (SplitOccured != 0)
4629 {
4630 // ~ Second deal with the INNER NODE(S) i.e Case #3 & Case #4:
4631 while (StackPtr != 0 || SplitOccured != 0)
4632 {
4633 // 'PseudoLinkedPointerNEW' is new LEAF to be inserted
4634 // 'wrdUP' is NEW word to be inserted
4635 if (StackPtr != 0)
4636 {
4637     POffsetInLEAF = BSTstack[--StackPtr];
4638     PseudoLinkedPointer = BSTstack[--StackPtr];
4639 if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // If LEAF is full: Case #4
4640 { SplitOccured = 1;
4641     memcpy( wrdUPold, wrdUP, wrdlen ); // LW up
4642     PseudoLinkedPointerNEWold = PseudoLinkedPointerNEW;
4643 // ALlocate NEW LEAF:
4644 if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wordlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wordlen] ) // +4 more for BST
instead of LL; + more(see LEAF)
4645 {
4646     memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
4647     bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
4648     bufend[LetterOffset] = bufend[LetterOffset] + 2*wordlen;
4649     if (MAXusedBuffer[wordlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wordlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}

```

```

4650     }
4651     else
4652     { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4653     fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4654     fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, 11ToADigits, 10) );
4655     fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, 11ToADigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, 11ToADigits2, 10) );
4656     fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4657     fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4658     fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4659     fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, 11ToADigits, 10)
);
4660     for( k = 1; k < 32; k++ )
4661     { fprintf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s%s utilization\n", _ui64toaKAZEzerocomma(k, 11ToADigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, 11ToADigits2, 10)+(26-5), _ui64toaKAZEzerocomma(((GRMBLh11[(int)k] *
LetterBuffer)/31)>>10)+1, 11ToADigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLh11[(int)k] *
LetterBuffer)/31), 11ToADigits4, 10)+(26-2), "%\0" ); // 26 are all 26-DESIRED=24
4662     }
4663     fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4664     fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4665     return( 1 );
4666     }
4667     if (POffsetInLEAF == 0) // wrdUPold < LW
4668     {
4669         memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
4670         memcpy( PseudoLinkedPointer+4+4+4, wrdUPold, wrdlen ); // wrdUPold go to OLD LEAF
4671         memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
4672         *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
4673         // [LP](PseudoLinkedPointerNEWold)[MP][RP](wrdUPold)[LW][RW] -----
4674         // pair [LW] PseudoLinkedPointerNEW goes up |
4675         // PseudoLinkedPointer: PseudoLinkedPointerNEW: |
4676         // [LP](PseudoLinkedPointerNEWold)[ ](wrdUPold) [MP][RP][ ] [RW] <----
4677         // no need to put zero in RP because logic is based on words existence:
4678         memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+4, 4 );
4679         memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
4680         memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEWold, 4 );
4681     }
4682     if (POffsetInLEAF == 4) // LW < wrdUPold < RW
4683     {
4684         memcpy( wrdUP, wrdUPold, wrdlen ); // wrdUPold up
4685         memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
4686         *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
4687         // [LP][MP](PseudoLinkedPointerNEWold)[RP][LW](wrdUPold)[RW] -----
4688         // pair [wrdUPold] PseudoLinkedPointerNEW goes up |
4689         // PseudoLinkedPointer: PseudoLinkedPointerNEW: |
4690         // [LP][MP][ ] [LW] (PseudoLinkedPointerNEWold)[RP][ ] [RW] <----
4691         // no need to put zero in RP because logic is based on words existence:
4692         memcpy( PseudoLinkedPointerNEW+0, &PseudoLinkedPointerNEWold, 4 );
4693         memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
4694     }
4695     if (POffsetInLEAF == 8) // wrdUPold > RW
4696     {
4697         memcpy( wrdUP, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW up
4698         *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
4699         memcpy( PseudoLinkedPointerNEW+4+4+4, wrdUPold, wrdlen ); // wrdUPold go to NEW LEAF
4700         // [LP][MP][RP](PseudoLinkedPointerNEWold)[LW][RW](wrdUPold) -----
4701         // pair [RW] PseudoLinkedPointerNEW goes up |
4702         // PseudoLinkedPointer: PseudoLinkedPointerNEW: |
4703         // [LP][MP][ ] [LW] [RP](PseudoLinkedPointerNEWold)[ ](wrdUPold) <----
4704         // no need to put zero in RP because logic is based on words existence:
4705         memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+8, 4 );
4706         memcpy( PseudoLinkedPointerNEW+4, &PseudoLinkedPointerNEWold, 4 );
4707     }
4708 }
4709 else // If LEAF is not full: Case #3
4710 { SplitOccured = 0;
4711     if (POffsetInLEAF == 0) // wrdUP < [LW][ ] so [LW][ ] -> [ ] [LW] -> [wrdUP][LW]
4712     {
4713         memcpy( PseudoLinkedPointer+4+4+4+wrdlen, PseudoLinkedPointer+4+4+4, wrdlen );
4714         memcpy( PseudoLinkedPointer+4+4+4, wrdUP, wrdlen );
4715         // [LP][MP][ ] -> [LP][ ] [MP] -> [LP][np][MP]
4716         memcpy( PseudoLinkedPointer+8, PseudoLinkedPointer+4, 4 );
4717         memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEW, 4 );
4718     }
4719     if (POffsetInLEAF == 4) // wrdUP > [LW][ ] so [LW][ ] -> [LW][wrdUP]
4720     {
4721         memcpy( PseudoLinkedPointer+4+4+4+wrdlen, wrdUP, wrdlen );
4722         // [LP][MP][ ] -> [LP][MP][np]
4723         memcpy( PseudoLinkedPointer+8, &PseudoLinkedPointerNEW, 4 );
4724     }
4725     break;
4726 }
4727 }
4728 else // Empty stack means ROOT and more over ROOT is already splitted(Case #4 is off)
4729 {
4730 // If LEAF is not full: Case #3
4731 // THIS IS WHERE A NEW(SECOND) LEAF 'PseudoLinkedPointerROOT' must be allocated:
4732     if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST

```

```

instead of LL; + more(see LEAF)
4733 {
4734     memcpy( &PseudoLinkedPointerROOT, &bufend[LetterOffset], 4 );
4735     bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
4736     memcpy( bufend[LetterOffset], wrdUP, wrdlen );
4737     bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
4738     if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
4739 }
4740 else
4741 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4742 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
4743 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, 11ToaDigits, 10) );
4744 fprintf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, 11ToaDigits2, 10) );
4745 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
4746 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
4747 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
4748 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, 11ToaDigits, 10)
);
4749 for( k = 1; k < 32; k++ )
4750 { fprintf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s%s utilization\n", _ui64toaKAZEzerocomma(k, 11ToaDigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, 11ToaDigits2, 10)+(26-5), _ui64toaKAZEzerocomma((((GRMBLh1l[(int)k] *
LetterBuffer)/31)>>10)+1, 11ToaDigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLh1l[(int)k] *
LetterBuffer)/31), 11ToaDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
4751 }
4752 fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4753 fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4754 return( 1 );
4755 }
4756 // Here -- 'PseudoLinkedPointerROOT' --
4757 // | (wrdUP)
4758 // 'PseudoLinkedPointer' <-- --> 'PseudoLinkedPointerNEW'
4759 // (LW) (RW)
4760 memcpy( PseudoLinkedPointerROOT, &PseudoLinkedPointer, 4 ); // LP
4761 memcpy( PseudoLinkedPointerROOT+4, &PseudoLinkedPointerNEW, 4 ); // MP
4762 // Here must NEW ROOT be updated i.e. HASH table(SLOT) must point it:
4763 memcpy( BufStart+Slot, &PseudoLinkedPointerROOT, 4 );
4764 break; //because it is ROOT without split
4765 }
4766 } // while
4767 } //if (SplitOccured != 0)
4768 // 3] Insert Iterative ]
4769 //if (FoundInLinkedList == 0)
4770 // ##### B-tree order 3 ]
4771 //if (BSTorBtree == 0)
4772 } else { //if (BSTorBtree != 2) {
4773 // External Btrees [
4774
4775 // - ASCII code 095
4776 // - ASCII code 096 \
4777 // a ASCII code 097 / In total 26+1+1 radix instead of 27 to avoid +1 for each '-', code 096 not used.
4778 // z ASCII code 122
4779 // The hash for 'a_quadruplet_for_example' will be calculated for first 5 chars:
4780 // = (byte1-'_')*28*28*28*28 + (byte2-'_')*28*28*28 + (byte3-'_')*28*28 + (byte4-'_')*28 + (byte5-'_')
4781 // Hash slots are 28*28*28*28 = 17,210,368 each containing one 64bit pointer i.e. 8bytes in length.
4782 // Hash size = 17,210,368*8 = 137,682,944 bytes
4783 // When at end all these slots(17,210,368- Btrees) are traversed the outcome is a sorted wordlist - no need of sorting.
4784
4785 // D:\_KAZE_new-stuff\Leprechaun_quadruplet_r14_minus>Leprechaun_quadruplet.exe GRAFFITH_2048.lst GRAFFITH_2048.wrd z
4786 // Leprechaun(Fast Greedy Word-Ripper), rev. 14_minus_quadruplet, written by svalqatchx.
4787 // Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
4788 // Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
4789 // also the performance of a 3-way hash + 6,602,752 B-Trees of order 3,
4790 // also the performance of a 1-way hash + 17,210,368 external B-Trees of order 3.
4791 // Size of input file with files for Leprechauning: 42140
4792 // Allocating HASH memory 137,683,009 bytes ... OK
4793 // Allocating/ZEROing 1,047,566 bytes swap file ... OK
4794 // Size of Input TEXTual file: 33,470,581
4795 // |; Word count: 3,045,077 of them 0 distinct; Done: 64/64
4796 // ...
4797 // Size of Input TEXTual file: 17,403,406
4798 // /; word count: 2,710,601,882 of them 0 distinct; Done: 64/64
4799 // Bytes per second performance: 17,694,246B/s
4800 // Words per second performance: 1,730,907w/s
4801 // Leprechaun: Done.
4802 //
4803 // Leprechaun report:
4804 // Number Of Trees(GREATER THE BETTER): 1,646,004
4805
4806 // TO DO - it is long overdue: at last make sort stage at end unnecessary - only traversing-and-dumping!
4807
4808 BufStart = pointerflush;
4809 // Slot = ((wrd[0]-'_')*28*28*28*28 + (wrd[1]-'_')*28*28*28 + (wrd[2]-'_')*28*28 + (wrd[3]-'_')*28 + (wrd[4]-'_'))<<3;
4810 // Slot = FNV1A_Hash_Jesteress_27bit(wrd, wrdlen)<<3; // Commented since r.14+++ because of passes.
4811 Slot = FNV1A_Hash_Jesteress_27bit(wrd, wrdlen);
4812
4813 // Bug fix for all r.14+++ and below! [

```



```

4814 memcpy( &wrd[(LongestLineInclusive+1+4)-4], &NULLsForWRD, 4 );
4815 // Bug fix for all r.14+++ and below! ]
4816
4817 // Example: HashInBITS-HashChunkSizeInBITS=2
4818 //         HashInBITS = 5
4819 //         HashChunkSizeInBITS = 3
4820 //         RipPasses = 1<<(HashInBITS-HashChunkSizeInBITS) i.e. 1<<2 which is 4 i.e. 32 slots with 4 passes 8 slots each.
4821 //         00??? 5bits 0-7
4822 //         01??? 5bits 8-15
4823 //         10??? 5bits 16-23
4824 //         11??? 5bits 24-31
4825 if ( (Slot>>HashChunkSizeInBITS) == RipPasses ) {
4826 Slot = Slot<<3;
4827
4828 //Slot = 0; // One Tree only!
4829 memcpy( &PseudoLinkedPointer_64, BufStart+Slot, 8 );
4830
4831 // ##### B-tree order 3 fragment 64bit [
4832 //
4833 // LEAF structure: [LeftPointer][MiddlePointer][RightPointer][Leftword][Rightword]
4834 //                4bytes      4bytes      4bytes      wrdlen      wrdlen
4835 //                *              *              *              *              *      <- if *(char *)==0 means the word cell is empty
4836 // ALL B-tree order 3 fragment consists of 3 sub-fragments:
4837 // 1] Search 2] if Search failed Trasirascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search 3] Insert Iterative
4838
4839 // LEAF_64 structure: [LeftPointer][MiddlePointer][RightPointer][Leftword] [Rightword]
4840 //                   8bytes      8bytes      8bytes      LongestLineInclusive+1+4 LongestLineInclusive+1+4
4841 //                   *              *              *              *              *      <- if *(char *)==0 means the word cell is empty
4842 // 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
4843 // choice ...
4844 // Note: BufEnd_64 in fact is the first free position after the BUFFER END!
4845
4846 // 1] Search [
4847 //         if (PseudoLinkedPointer_64 == 0) // means EMPTY-SLOT
4848 //         {
4849 //         if ( REUSE != 2 ) { // REUSE [ // This line comes since r16
4850 //         if (DoNotInsertFlag == 0)
4851 //         { // This line comes since r15FIXFIX+ [#####
4852 //         //if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST
4853 //         instead of LL; + more(see LEAF)
4854 //         {
4855 //         memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
4856 //         bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
4857 //         memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
4858 //         bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
4859 //         if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
4860 //         long)(bufend[LetterOffset] - BufStart);}
4861 //         if( 8 + 8 + 8 + 2*(LongestLineInclusive+1+4) < size_in64_L14 - (BufEnd_64-(unsigned long long)pointerflush_64) ) // the longest
4862 //         wrdlen is LongestLineInclusive but actual is LongestLineInclusive(+ CR char)
4863 //         {
4864 //         memcpy( BufStart+Slot, &BufEnd_64, 8 );
4865 //         BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
4866 //         if (BStorBtree == 2) {
4867 //         fsetpos(fp_outRG, &BufEnd_64);
4868 //         fwrite(wrd, wrdlen, 1, fp_outRG); WORDcountDistinct++;
4869 //         //fwrite(&OneChar_ieByte, 1, 1, fp_outRG); // Write ZERO ASCII code
4870 //         // r.14+++ The above line was commented because the pool is already ZEROed.
4871 //         } else { // ##### 64bit memory manipulations [
4872 //         memcpy( (char *)BufEnd_64, wrd, wrdlen ); WORDcountDistinct++;
4873 //         } // ##### 64bit memory manipulations ]
4874 //         BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
4875 //         fsetpos(fp_outRG, &BufEnd_64);
4876 //         }
4877 //         else
4878 //         { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
4879 //         fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10), _ui64toaKAZEcomma((unsigned long
4880 //         long)WORDcountDistinct, 11ToaDigits2, 10) );
4881 //         fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, 11ToaDigits, 10) );
4882 //         fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, 11ToaDigits, 10)
4883 //         );
4884 //         fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
4885 //         fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
4886 //         return( 1 );
4887 //         }
4888 //         FoundInLinkedList = 1;
4889 //         } // This line comes since r15FIXFIX+ [#####
4890 //         }
4891 //         else // means USED-SLOT
4892 //         { FoundInLinkedList = 0;
4893 //         StackPtr = 0;
4894 //         while (PseudoLinkedPointer != 0 && FoundInLinkedList == 0)
4895 //         while (PseudoLinkedPointer_64 != 0 && FoundInLinkedList == 0)
4896 //         {
4897 //         // ***** 'P W P' section [
4898 //         // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
4899 //         // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 )

```

```

4896 // here ALWAYS LW exists: no need for existence check - line below
4897 // if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
4898 // if ( memcmp(PseudoLinkedPointer+4+4+4, wrd, wrdlen) > 0 ) // go LP
4899 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4900 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
4901 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
4902 //fread(&FourGramL[0], (LongestLineInclusive+1+4), 1, fp_outRG);
4903 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4904 // [ //r.14+
4905 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
4906 if (BSTorBtree == 2) {
4907 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
4908 fread(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
4909 } else { // ##### 64bit memory manipulations [
4910 memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+2*(LongestLineInclusive+1+4) );
4911 } // ##### 64bit memory manipulations ]
4912 memcpy( &FourGramL[0], &LEAF[8 + 8 + 8], (LongestLineInclusive+1+4) );
4913 // ] //r.14+
4914 if (strcmpKAZE13(FourGramL, wrd) > 0) // go LP
4915 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 0, 4 ); //LP
4916 PseudoLinkedPointer = PseudoLinkedPointerNEW;
4917 }
4918 {
4919 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4920 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 0; //LP
4921 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4922 if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
4923 BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
4924 BSTstack[StackPtr] = 0; ++StackPtr; //LPOffset=0;MPOffset=8;RPOffset=16;
4925 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4926 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
4927 //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
4928 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4929 // [ //r.14+
4930 memcpy( &PseudoLinkedPointer_64, &LEAF[0], 8 );
4931 // ] //r.14+
4932 }
4933 // else if ( memcmp(PseudoLinkedPointer+4+4+4, wrd, wrdlen) < 0 ) // go RP or MP
4934 else if ( strcmpKAZE13(FourGramL, wrd) < 0 ) // go RP or MP
4935 { // RW existence check - line below:
4936 // if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // RW exists
4937 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4938 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4); //RW
4939 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
4940 //fread(&SomeByte, 1, 1, fp_outRG);
4941 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4942 // [ //r.14+
4943 memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
4944 // ] //r.14+
4945 if (SomeByte != 0 ) // RW exists
4946 { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
4947 // ++++++
4948 // ***** 'P W P' section 2 [
4949 // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
4950 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
4951 // here ALWAYS RW exists: no need for existence check - line below
4952 // if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
4953 // if ( memcmp(PseudoLinkedPointer+4+4+4+wrdlen, wrd, wrdlen) > 0 ) // go MP
4954 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4955 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
4956 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
4957 //fread(&FourGramL[0], (LongestLineInclusive+1+4), 1, fp_outRG);
4958 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4959 // [ //r.14+
4960 memcpy( &FourGramL[0], &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], (LongestLineInclusive+1+4) );
4961 // ] //r.14+
4962 if (strcmpKAZE13(FourGramL, wrd) > 0) // go MP
4963 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
4964 PseudoLinkedPointer = PseudoLinkedPointerNEW;
4965 }
4966 {
4967 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4968 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8; //MP
4969 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4970 if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
4971 BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
4972 BSTstack[StackPtr] = 8; ++StackPtr; //LPOffset=0;MPOffset=8;RPOffset=16;
4973 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4974 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
4975 //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
4976 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4977 // [ //r.14+
4978 memcpy( &PseudoLinkedPointer_64, &LEAF[8], 8 );
4979 // ] //r.14+
4980 }
4981 // else if ( memcmp(PseudoLinkedPointer+4+4+4+wrdlen, wrd, wrdlen) < 0 ) // go RP
4982 else if ( strcmpKAZE13(FourGramL, wrd) < 0 ) // go RP
4983 { // No ?W after RW - go RP

```

```

4984 //                memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
4985 //                PseudoLinkedPointer = PseudoLinkedPointerNEW;
4986 //            }
4987 //        }
4988 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4989 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8; //RP
4990 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4991 if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
4992 BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
4993 BSTstack[StackPtr] = 16; ++StackPtr; //LPoffset=0;MPoffset=8;RPoffset=16;
4994 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4995 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
4996 //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
4997 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
4998 // [ //r.14+
4999 memcpy( &PseudoLinkedPointer_64, &LEAF[8 + 8], 8 );
5000 // ] //r.14+
5001 }
5002 else { FoundInLinkedList = 1; // wrd is RW
5003 // Counter [
5004 if (BSTorBtree == 2) {
5005 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5006 }
5007 memcpy( &CounterOccurrences, &FourGramL[(LongestLineInclusive+1+4)-4], 4 );
5008
5009 // r16 [
5010 if ( REUSE == 2 ) {
5011 if (*argv[k_FIX] == 'w')
5012     fprintf(fp_out, "%s\t%s\r\n", _ui64toaKAZEzerocomma(CounterOccurrences+1, llToaDigits2, 10)+(26-9), wrd);
//WORDcountBOTTOM++;
5013 if (*argv[k_FIX] == 'w')
5014     fprintf(fp_out, "%s\r\n", wrd); //WORDcountBOTTOM++;
5015 }
5016 // r16 ]
5017
5018 if ( REUSE != 2 ) { // r16
5019 if (CounterOccurrences<99999999) CounterOccurrences++;
5020 memcpy( &FourGramL[(LongestLineInclusive+1+4)-4], &CounterOccurrences, 4 );
5021 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5022 //fwrite(&FourGramL[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5023 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5024 // [ //r.14+
5025 memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &FourGramL[0], (LongestLineInclusive+1+4) );
5026 if (BSTorBtree == 2) {
5027 fwrite(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5028 } else { // ##### 64bit memory manipulations [
5029 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4) );
5030 } // ##### 64bit memory manipulations ]
5031 // ] //r.14+
5032 } // r16
5033 // Counter ]
5034 }
5035 WORDcountAttemptsToPut++;
5036 // ***** 'P W P' section 2 ]
5037 // ++++++
5038 }
5039 else // RW empty - go MP
5040 {
5041 // { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
5042 // PseudoLinkedPointer = PseudoLinkedPointerNEW;
5043 // }
5044 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5045 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8; //MP
5046 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5047 if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5048 BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
5049 BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0;MPoffset=8;RPoffset=16;
5050 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5051 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5052 //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
5053 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5054 // [ //r.14+
5055 memcpy( &PseudoLinkedPointer_64, &LEAF[8], 8 );
5056 // ] //r.14+
5057 }
5058 }
5059 else { FoundInLinkedList = 1; // wrd is LW
5060 // Counter [
5061 if (BSTorBtree == 2) {
5062 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5063 }
5064 memcpy( &CounterOccurrences, &FourGramL[(LongestLineInclusive+1+4)-4], 4 );
5065
5066 // r16 [
5067 if ( REUSE == 2 ) {
5068 if (*argv[k_FIX] == 'w')
5069     fprintf(fp_out, "%s\t%s\r\n", _ui64toaKAZEzerocomma(CounterOccurrences+1, llToaDigits2, 10)+(26-9), wrd);
//WORDcountBOTTOM++;

```

```

5070         if (*argv[k_FIX] == 'w')
5071             fprintf(fp_out, "%s\r\n", wrd); //WORDcountBOTTOM++;
5072     }
5073     // r16 ]
5074
5075     if ( REUSE != 2 ) { // r16
5076         if (CounterOccurrences<9999999) CounterOccurrences++;
5077         memcpy( &FourGramL[(LongestLineInclusive+1+4)-4], &CounterOccurrences, 4 );
5078         // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5079         //fwrite(&FourGramL[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5080         // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5081         // [ //r.14+
5082         memcpy( &LEAF[8 + 8 + 8], &FourGramL[0], (LongestLineInclusive+1+4) );
5083         if (BSTorBtree == 2) {
5084             fwrite(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5085         } else { // ##### 64bit memory manipulations [
5086             memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+2*(LongestLineInclusive+1+4) );
5087             // ##### 64bit memory manipulations ]
5088         } //r.14+
5089     } // r16
5090     // Counter ]
5091 }
5092 WORDcountAttemptsToPut++;
5093 // ***** 'P W P' section ]
5094     } // while
5095     WORDcountAttemptsToPut--; // - 1 due to BST way of counting i.e. direct hash hit is not counted only successors
5096 }
5097 // 1] Search ]
5098
5099 if ( REUSE != 2 ) { // REUSE [ // This line comes since r16
5100     if (DoNotInsertFlag == 0) { // This line comes since r15FIXFIX+
5101         if (FoundInLinkedList == 0)
5102         {
5103             /*
5104             // ===== [ The whole section/sub-fragment 2 is commented due to great time differences for Internal_vs_External memory
5105             // accesses - it is far more cheap to have the STACK overhead (moved to sub-fragment 1) ] ===== [
5106             // 2] if Search failed Trasirascht (pushing in stack PseudoLinkedPointer(visited LEAFs)) Search [
5107             // 'TracingSearch' is the same as 'Search' except that adds the trail in my simulated stack,
5108             // the goal is not to waste time in 'Search' by dealing with no needed trail in case of not 'Insert'.
5109             // Simulated stack contains pairs of 'Address of ParentLEAF' + 'Offset of ParentPointer in ParentLEAF i.e. 0 for LP, 4 for MP, 8 for RP'.
5110             // 'Offset ...' saves unnecessary comparisons of NEWword which after splitting goes up.
5111             memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
5112             StackPtr = 0;
5113             while (PseudoLinkedPointer != 0)
5114             {
5115             // ***** 'P W P' section [
5116             // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
5117             // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
5118             // here ALWAYS LW exists: no need for existence check - line below
5119             // if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
5120             if (memcmp(PseudoLinkedPointer+4+4+4, wrd, wordlen) > 0) // go LP
5121             { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 0, 4 ); //LP
5122             if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5123             BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5124             BSTstack[StackPtr] = 0; ++StackPtr; //LPoffset=0;MPoffset=4;RPOffset=8;
5125             PseudoLinkedPointer = PseudoLinkedPointerNEW;
5126             }
5127             else if (memcmp(PseudoLinkedPointer+4+4+4, wrd, wordlen) < 0) // go RP or MP
5128             { // RW existence check - line below:
5129             if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 ) // RW exists
5130             { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
5131             // ++++++
5132             // ***** 'P W P' section 2 [
5133             // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
5134             // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
5135             // here ALWAYS RW exists: no need for existence check - line below
5136             // if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
5137             if (memcmp(PseudoLinkedPointer+4+4+4+wordlen, wrd, wordlen) > 0) // go MP
5138             { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
5139             if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5140             BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5141             BSTstack[StackPtr] = 4; ++StackPtr; //LPoffset=0;MPoffset=4;RPOffset=8;
5142             PseudoLinkedPointer = PseudoLinkedPointerNEW;
5143             }
5144             else if (memcmp(PseudoLinkedPointer+4+4+4+wordlen, wrd, wordlen) < 0) // go RP
5145             { // No ?w after RW - go RP
5146             memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
5147             if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5148             BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5149             BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0;MPoffset=4;RPOffset=8;
5150             PseudoLinkedPointer = PseudoLinkedPointerNEW;
5151             }
5152             else FoundInLinkedList = 1; // wrd is RW
5153             // ***** 'P W P' section 2 ]
5154             // ++++++
5155             }
5156             else // RW empty - go MP
5157             { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP

```

```

5157         if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5158         BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5159         BSTstack[StackPtr] = 4; ++StackPtr; //LPOffset=0;MPoffset=4;RPOffset=8;
5160             PseudoLinkedPointer = PseudoLinkedPointerNEW;
5161         }
5162     }
5163     else FoundInLinkedList = 1; // wrd is LW
5164 // ***** 'p w p' section ]
5165     } // while
5166 // 2] if Search failed Trasirascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search ]
5167 // ===== [ The whole section/sub-fragment 2 is commented due to great time differences for Internal_vs_External memory
5168 // accesses - it is far more cheap to have the STACK overhead (moved to sub-fragment 1) ] ===== ]
5169 */
5170 // 3] Insert Iterative [
5171 // There are total 4 situations:
5172 // Case #1: Outer NODE(including ROOT) [ ][ ][ ][LW][ ]
5173 // Case #2: Outer NODE(including ROOT) [ ][ ][ ][LW][RW] Split Occurs ----- | 'wrdUP' (wrdlen bytes)
5174 // Case #3: ROOT [LP][MP][ ][LW][ ] Split Occurs --- | &
5175 // Case #4: Inner NODE(including ROOT) [LP][MP][RP][LW][RW] Split Occurs | | 'PseudoLinkedPointerNEW' (ptr to NEW LEAF)
5176 // There are total 2 situations for PARENT LEAF: <----- ARE GOING UP
5177 // Case #3: [LP][MP][ ][LW][ ]
5178 // Case #4: [LP][MP][RP][LW][RW] Split Occurs
5179 //
5180 // ~ First deal alonely with the OUTER NODE(LEAF) where Search stopped i.e Case #1 & Case #2:
5181 // POffsetInLEAF = BSTstack[--StackPtr];
5182 // PseudoLinkedPointer_64 = BSTstack[--StackPtr];
5183 // NOTE: ONE LEAF IS FULL ONLY WHEN LAST CELL FOR KEY(here RW) EXISTS!
5184 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
5185 //if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // If LEAF is full: Case #2
5186 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5187 // PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4); //RW
5188 // fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5189 // fread(&SomeByte, 1, 1, fp_outRG);
5190 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5191 // [ //r.14+
5192 // PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
5193 // if (BSTorBtree == 2) {
5194 //     fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5195 //     fread(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5196 // } else { // ##### 64bit memory manipulations [
5197 //     memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+8+2*(LongestLineInclusive+1+4) );
5198 // } // ##### 64bit memory manipulations ]
5199 // memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
5200 // ] //r.14+
5201 // if (SomeByte != 0 ) // RW exists
5202 { SplitOccured = 1; WORDcountDistinct++;
5203 // Allocate NEW LEAF:
5204 // if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFooAgain[(int)wrdlen] ) // +4 more for BST
5205 // instead of LL; + more(see LEAF)
5206 // {
5207 //     memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
5208 //     bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
5209 //     bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
5210 //     if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
5211 // long)(bufend[LetterOffset] - BufStart);}
5212 //     if( 8 + 8 + 8 + 2*(LongestLineInclusive+1+4) < size_in64_L14 - (BufEnd_64-(unsigned long long)pointerflush_64) ) // the longest
5213 // wrdlen is LongestLineInclusive but actual is LongestLineInclusive(+ CR char)
5214 //     {
5215 //         PseudoLinkedPointerNEW_64 = BufEnd_64;
5216 //         BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
5217 //         BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
5218 //     }
5219 //     else
5220 //     { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
5221 //       fprintf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, l1ToaDigits, 10), _ui64toaKAZEcomma((unsigned
5222 // long long)WORDcountDistinct, l1ToaDigits, 10) );
5223 //       fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, l1ToaDigits, 10) );
5224 //       fprintf( fp_outLOG, "Total Attempts to Find/Put WORDS into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut,
5225 // l1ToaDigits, 10) );
5226 //       fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
5227 //       fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
5228 //       return( 1 );
5229 //     }
5230 //     if (POffsetInLEAF == 0) // wrd < LW
5231 //     {
5232 //         memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
5233 //         memcpy( PseudoLinkedPointer+4+4+4, wrd, wrdlen ); // wrd go to OLD LEAF
5234 //         memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
5235 //         *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
5236 //         [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5237 //         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5238 //         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5239 //         //fread(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5240 //         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5241 //         //fwrite(&wrd[0], (LongestLineInclusive+1+4), 1, fp_outRG);

```

```

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

```

```

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

```

```

5415         if (SomeByte != 0 ) // RW exists
5416     { SplitOccured = 1;
5417 //         memcpy( wrdUPold, wrdUP, wrdlen ); // LW up
5418 //         PseudoLinkedPointerNEWold = PseudoLinkedPointerNEW;
5419 //         memcpy( wrdUPold, wrdUP, (LongestLineInclusive+1+4) );
5420 //         PseudoLinkedPointerNEWold_64 = PseudoLinkedPointerNEW_64;
5421 //         // ALlocate NEW LEAF:
5422 //         if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2* wrdlen + 4 + 4 + 4 < GRMBLFoolAgain((int) wrdlen) ) // +4 more for BST
instead of LL; + more(see LEAF)
5423 //         {
5424 //             memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
5425 //             bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
5426 //             bufend[LetterOffset] = bufend[LetterOffset] + 2* wrdlen;
5427 //             if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
5428 //         }
5429 //         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)
5430 //         {
5431 //             PseudoLinkedPointerNEW_64 = BufEnd_64;
5432 //             BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
5433 //             BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
5434 //             // [ //r.14+
5435 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64;
5436 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5437 //             //fread(&LEAFNEW[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5438 //             // In fact above three lines are slow, the only need is ZEROed LEAFNEW.
5439 //             memset(&LEAFNEW[0], 0, 8+8+2*(LongestLineInclusive+1+4));
5440 //             // ] //r.14+
5441 //         }
5442 //         else
5443 //         { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
5444 //             fprintf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDCount, l1ToaDigits, 10), _ui64toaKAZEcomma((unsigned
long long)WORDCountDistinct, l1ToaDigits2, 10) );
5445 //             fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, l1ToaDigits, 10) );
5446 //             fprintf( fp_outLOG, "Total Attempts to Find/Put WORDS into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDCountAttemptsToPut,
l1ToaDigits, 10) );
5447 //             fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
5448 //             fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
5449 //             return( 1 );
5450 //         }
5451 //         if (POffsetInLEAF == 0) // wrdUPold < LW
5452 //         {
5453 //             memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
5454 //             memcpy( PseudoLinkedPointer+4+4+4, wrdUPold, wrdlen ); // wrdUPold go to OLD LEAF
5455 //             memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
5456 //             *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
5457 //             // [LP](PseudoLinkedPointerNEWold)[MP][RP](wrdUPold)[LW][RW] -----
5458 //             // pair [LW] PseudoLinkedPointerNEW goes up
5459 //             // PseudoLinkedPointer: PseudoLinkedPointerNEW: |
5460 //             // [LP](PseudoLinkedPointerNEWold)[ ](wrdUPold) [MP][RP][ ][RW] <----
5461 //             // no need to put zero in RP because logic is based on words existence:
5462 //             memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+4, 4 );
5463 //             memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
5464 //             memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEWold, 4 );
5465 //             // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5466 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5467 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5468 //             //fread(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5469 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5470 //             //fwrite(&wrdUPold[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5471 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5472 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5473 //             //fread(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5474 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5475 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5476 //             //fwrite(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5477 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5478 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5479 //             //fwrite(&OneChar_ieByte, 1, 1, fp_outRG); // Write ZERO ASCII code
5480 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
5481 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5482 //             //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5483 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 0;
5484 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5485 //             //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5486 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5487 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5488 //             //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5489 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8;
5490 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5491 //             //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5492 //             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
5493 //             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5494 //             //fwrite(&PseudoLinkedPointerNEWold_64, 8, 1, fp_outRG);
5495 //             // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5496 //             // [ //r.14+
5497 //             memcpy( &wrdUP[0], &LEAF[8 + 8 + 8], (LongestLineInclusive+1+4) );

```



```

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

```

```

5586 // // PseudoLinkedPointer: PseudoLinkedPointerNEW: |
5587 // // [LP][MP][LW] [RP](PseudoLinkedPointerNEWold)[(wrdUPold) <---
5588 // // no need to put zero in RP because logic is based on words existence:
5589 // memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+8, 4 );
5590 // memcpy( PseudoLinkedPointerNEW+4, &PseudoLinkedPointerNEWold, 4 );
5591 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5592 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5593 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5594 //fread(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5595 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5596 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5597 //fwrite(&OneChar_ieByte, 1, 1, fp_outRG); // Write ZERO ASCII code
5598 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5599 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5600 //fwrite(&wrdUPold[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5601 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5602 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5603 //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5604 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 0;
5605 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5606 //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5607 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8;
5608 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5609 //fwrite(&PseudoLinkedPointerNEWold_64, 8, 1, fp_outRG);
5610 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5611 // [ //r.14+
5612 memcpy( &wrdUP[0], &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], (LongestLineInclusive+1+4) );
5613 memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &OneChar_ieByte, 1 );
5614 memcpy( &LEAFNEW[8 + 8 + 8], &wrdUPold[0], (LongestLineInclusive+1+4) );
5615 memcpy( &PseudoLinkedPointerAUXdumbo_64, &LEAF[16], 8 );
5616 memcpy( &LEAFNEW[0], &PseudoLinkedPointerAUXdumbo_64, 8 );
5617 memcpy( &LEAFNEW[8], &PseudoLinkedPointerNEWold_64, 8 );
5618 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
5619 if (BStorBtree == 2) {
5620 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5621 fwrite(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5622 } else { // ##### 64bit memory manipulations [
5623 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4) );
5624 } // ##### 64bit memory manipulations ]
5625 PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64;
5626 if (BStorBtree == 2) {
5627 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5628 fwrite(&LEAFNEW[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5629 } else { // ##### 64bit memory manipulations [
5630 memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAFNEW[0], 8+8+8+2*(LongestLineInclusive+1+4) );
5631 } // ##### 64bit memory manipulations ]
5632 // ] //r.14+
5633 }
5634 }
5635 else // If LEAF is not full: Case #3
5636 { SplitOccured = 0;
5637 if (PoffsetInLEAF == 0) // wrdUP < [LW][ ] so [LW][ ] -> [ ][LW] -> [wrdUP][LW]
5638 {
5639 // memcpy( PseudoLinkedPointer+4+4+4+wrdlen, PseudoLinkedPointer+4+4+4, wrdlen );
5640 // memcpy( PseudoLinkedPointer+4+4+4, wrdUP, wrdlen );
5641 // // [LP][MP][ ] -> [LP][ ][MP] -> [LP][np][MP]
5642 // memcpy( PseudoLinkedPointer+8, PseudoLinkedPointer+4, 4 );
5643 // memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEW, 4 );
5644 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5645 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5646 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5647 //fread(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5648 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5649 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5650 //fwrite(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5651 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5652 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5653 //fwrite(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5654 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
5655 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5656 //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5657 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5658 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5659 //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5660 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
5661 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5662 //fwrite(&PseudoLinkedPointerNEW_64, 8, 1, fp_outRG);
5663 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5664 // [ //r.14+
5665 memcpy( &wrdAUX[0], &LEAF[8 + 8 + 8], (LongestLineInclusive+1+4) );
5666 memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &wrdAUX[0], (LongestLineInclusive+1+4) );
5667 memcpy( &LEAF[8 + 8 + 8], &wrdUP[0], (LongestLineInclusive+1+4) );
5668 memcpy( &PseudoLinkedPointerAUXdumbo_64, &LEAF[8], 8 );
5669 memcpy( &LEAF[8 + 8], &PseudoLinkedPointerAUXdumbo_64, 8 );
5670 memcpy( &LEAF[8], &PseudoLinkedPointerNEW_64, 8 );
5671 if (BStorBtree == 2) {
5672 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5673 fwrite(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);

```

```

5674         } else { // ##### 64bit memory manipulations [
5675         memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+2*(LongestLineInclusive+1+4) );
5676         } // ##### 64bit memory manipulations ]
5677         // ] //r.14+
5678     }
5679     if (POffsetInLEAF == 8) // wrdUP > [LW][ ] so [LW][ ] -> [LW][wrdUP]
5680     {
5681         memcpy( PseudoLinkedPointer+4+4+4+wrdlen, wrdUP, wrdlen );
5682         // [LP][MP][ ] -> [LP][MP][np]
5683         // memcpy( PseudoLinkedPointer+8, &PseudoLinkedPointerNEW, 4 );
5684         // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5685         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5686         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5687         //fwrite(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5688         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5689         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5690         //fwrite(&PseudoLinkedPointerNEW_64, 8, 1, fp_outRG);
5691         // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5692         // [ //r.14+
5693         memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &wrdUP[0], (LongestLineInclusive+1+4) );
5694         memcpy( &LEAF[8 + 8], &PseudoLinkedPointerNEW_64, 8 );
5695         if (BSTorBtree == 2) {
5696             fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5697             fwrite(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5698         } else { // ##### 64bit memory manipulations [
5699         memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+2*(LongestLineInclusive+1+4) );
5700         } // ##### 64bit memory manipulations ]
5701         // ] //r.14+
5702     }
5703     break;
5704 }
5705 }
5706 else // Empty stack means ROOT and more over ROOT is already splitted(Case #4 is off)
5707 {
5708     // If LEAF is not full: Case #3
5709     // THIS IS WHERE A NEW(SECOND) LEAF 'PseudoLinkedPointerROOT' must be allocated:
5710     // if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain((int)wrdlen) ) // +4 more for BST
5711     // instead of LL; + more(see LEAF)
5712     {
5713         memcpy( &PseudoLinkedPointerROOT, &bufend[LetterOffset], 4 );
5714         bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
5715         memcpy( bufend[LetterOffset], wrdUP, wrdlen );
5716         bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
5717         if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
5718         long)(bufend[LetterOffset] - BufStart);}
5719         if( 8 + 8 + 8 + 2*(LongestLineInclusive+1+4) < size_in64_L14 - (BufEnd_64-(unsigned long long)pointerflush_64) ) // the longest
5720         wrdlen is LongestLineInclusive but actual is LongestLineInclusive(+ CR char)
5721         {
5722             PseudoLinkedPointerROOT_64 = BufEnd_64;
5723             BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
5724             BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
5725             PseudoLinkedPointerAUX_64 = PseudoLinkedPointerROOT_64 + 8 + 8 + 8;
5726             if (BSTorBtree == 2) {
5727                 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5728                 fwrite(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5729             } else { // ##### 64bit memory manipulations [
5730             memcpy( (char *)PseudoLinkedPointerAUX_64, &wrdUP[0], (LongestLineInclusive+1+4) );
5731             } // ##### 64bit memory manipulations ]
5732         }
5733     }
5734     else
5735     { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
5736     fprintf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDCount, llToADigits, 10), _ui64toaKAZEcomma((unsigned
5737     long long)WORDCountDistinct, llToADigits2, 10) );
5738     fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, llToADigits, 10) );
5739     fprintf( fp_outLOG, "Total Attempts to Find/Put WORDS into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDCountAttemptsToPut,
5740     llToADigits, 10) );
5741     fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
5742     fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
5743     return( 1 );
5744     }
5745     // Here -- 'PseudoLinkedPointerROOT' --
5746     // | (wrdUP) |
5747     // 'PseudoLinkedPointer' <-- --> 'PseudoLinkedPointerNEW'
5748     // (LW) (RW)
5749     memcpy( PseudoLinkedPointerROOT, &PseudoLinkedPointer, 4 ); // LP
5750     memcpy( PseudoLinkedPointerROOT+4, &PseudoLinkedPointerNEW, 4 ); // MP
5751     // Here must NEW ROOT be updated i.e. HASH table(SLOT) must point it:
5752     memcpy( BufStart+Slot, &PseudoLinkedPointerROOT, 4 );
5753     PseudoLinkedPointerAUX_64 = PseudoLinkedPointerROOT_64;
5754     if (BSTorBtree == 2) {
5755         fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5756         fwrite(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
5757     } else { // ##### 64bit memory manipulations [
5758     memcpy( (char *)PseudoLinkedPointerAUX_64, &PseudoLinkedPointer_64, 8 );
5759     } // ##### 64bit memory manipulations ]
5760     PseudoLinkedPointerAUX_64 = PseudoLinkedPointerROOT_64 + 8;
5761     if (BSTorBtree == 2) {

```

```

5757 fsetpos(fp_outRG, &PseudolinkedPointerAUX_64);
5758 fwrite(&PseudolinkedPointerNEW_64, 8, 1, fp_outRG);
5759 } else { // ##### 64bit memory manipulations [
5760 memcopy( (char *)PseudolinkedPointerAUX_64, &PseudolinkedPointerNEW_64, 8 );
5761 } // ##### 64bit memory manipulations ]
5762 memcopy( BufStart+slot, &PseudolinkedPointerROOT_64, 8 );
5763 break; //because it is ROOT without split
5764 }
5765 } // while
5766 } //if (SplitOccured != 0)
5767 // 3] Insert Iterative ]
5768 } //if (FoundInLinkedList == 0)
5769 } //if (DoNotInsertFlag == 0) { // This line comes since r15FIXFIX+
5770 } //if ( REUSE != 2 ) { // REUSE [ // This line comes since r16
5771 // ##### B-tree order 3 64bit ]
5772 } //if ( ( slot>>HashChunkSizeInBITS) == RipPasses ) {
5773
5774 // External Btrees ]
5775 } //if (BSTorBtree != 2) {
5776 } // if ( ( PLE_words == 4 ) && ( wrdlen <= 31 ) ) {
5777 } // if( 1 <= wrdlen && wrdlen <= 31 )
5778 /*
5779 The Most Stupid Bug I have ever made [
5780 else {
5781 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.
5782 }
5783 The Most Stupid Bug I have ever made ]
5784 */
5785 //r.15fix [
5786 //This line was intended BUT placed before the fragment 'if( 1 <= wrdlen && wrdlen <= 31 )' [
5787 // if( wrdlen > 31 ) PLE_words_INITflag = 1; // Not when wrdlen == 0 as the above buggy fragment!
5788 //This line was intended BUT placed before the fragment 'if( 1 <= wrdlen && wrdlen <= 31 )' ]
5789 //r.15fix ]
5790 if ( PLE_words_INITflag == 1 ) { PLE_words = 0; PLE_words_INITflag = 0; } // Quadruple!
5791 wrdlen = 0;
5792 // This fragment is MIRRORed: #1 copy ]
5793 }
5794 //else if( workbyte >= 'A' && workbyte <= 'Z' )
5795 else if( workbyte <= 'Z' )
5796 {
5797 if( wrdlen < 31 )
5798 //if( wrdlen < LongestLineInclusive )
5799 { wrd[ wrdlen ] = workbyte + 32 ; }
5800 wrdlen++;
5801 }
5802 else if( workbyte >= 'a' && workbyte <= 'z' )
5803 {
5804 if( wrdlen < 31 )
5805 //if( wrdlen < LongestLineInclusive )
5806 { wrd[ wrdlen ] = workbyte; }
5807 wrdlen++;
5808 }
5809 else
5810 {
5811 // This fragment is MIRRORed: #2 copy [
5812 goto ElStupido;
5813 // This fragment is MIRRORed: #2 copy ]
5814 }
5815 } // i 'for'
5816 //~~~~~
5817 //++Melnitckka;
5818 //Melnitckka = Melnitckka % 4;
5819 //if (Melnitckka == 0){ printf( "|; word count: %s of them %s distinct; Done: %lu/64\n", _ui64toaKAZEcomma(WORDcount, llToaDigits, 10),
_ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llToaDigits2, 10), 64 ); }
5820 //if (Melnitckka == 1){ printf( "/; word count: %s of them %s distinct; Done: %lu/64\n", _ui64toaKAZEcomma(WORDcount, llToaDigits, 10),
_ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llToaDigits2, 10), 64 ); }
5821 //if (Melnitckka == 2){ printf( "-; word count: %s of them %s distinct; Done: %lu/64\n", _ui64toaKAZEcomma(WORDcount, llToaDigits, 10),
_ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llToaDigits2, 10), 64 ); }
5822 //if (Melnitckka == 3){ printf( "\\; word count: %s of them %s distinct; Done: %lu/64\n", _ui64toaKAZEcomma(WORDcount, llToaDigits, 10),
_ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llToaDigits2, 10), 64 ); }
5823 Melnitckka = Melnitckka & 3; // 0 1 2 3: 00 01 10 11
5824 (void) time(&t4);
5825 if (t4 <= t1) {t4 = t1; t4++;}
5826 printf( "%s; %sP/s; Phrase count: %s of them %s distinct; Done: %lu/64\n", Auberge[Melnitchka++], _ui64toaKAZEzerocomma(WORDcount/((int) t4-
t1), llToaDigits3, 10)+(26-10), _ui64toaKAZEcomma(WORDcount, llToaDigits, 10), _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct,
llToaDigits2, 10), 64 );
5827
5828 fclose( fp_inLINE );
5829 } // ~~~~~ IT IS a FILENAME not a METACOMMAND ]
5830 LINE10len = 0;
5831 LINE10[ LINE10len ] = 0;
5832 }
5833 }
5834 } // k 'for'
5835
5836 (void) time(&t3);
5837 if (t3 <= t1) {t3 = t1; t3++;}

```

```

5838 printf( "Bytes per second performance: %sB/s\n", _ui64toaKAZEcomma(FilesLEN/((int) t3-t1), 11ToADigits, 10) ); // Rev. 12+
5839 printf( "Phrases per second performance: %sP/s\n", _ui64toaKAZEcomma(WORDcount/((int) t3-t1), 11ToADigits, 10) ); // Rev. 12+
5840 printf("Time for putting phrases into trees: %d second(s)\n", (int) t3-t1);
5841
5842 if (BSTorBtree < 2) {
5843     // FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH
5844     printf("Flushing unsorted words ...\n");
5845     if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
5846     { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
5847     ZEROS[0] = 0; ZEROS[1] = 0; ZEROS[2] = 0; ZEROS[3] = 0;
5848     CRdLFa[0] = 13; CRdLFa[1] = 10;
5849
5850     for( i = 0; i < 806; i++ )
5851     { //BufStart = pointerflush + i * LetterBuffer; // OLD
5852         BufStart = pointerflush + (i / 31) * WHOLELetter_BufferSize + OffsetsInBuffer[i % 31];
5853         // for( j = 0; j < NumberOfSLOTS; j++ )
5854         // {
5855             //         Slot = j<<2;
5856             //         memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
5857             //         while (PseudoLinkedPointer != 0)
5858             //         { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
5859                 //         memcpy( PseudoLinkedPointer, ZEROS, 4 );
5860                 //         PseudoLinkedPointer = PseudoLinkedPointerNEW;
5861             //         }
5862         // }
5863         // // Start of COUPLES [OFFSET: 4byte(ZEROS)][WORD:up to 31bytes]
5864         // //fwrite(BufStart+(NumberOfSLOTS+1)*4, bufend[i] - (BufStart+(NumberOfSLOTS+1)*4), 1, fp_out );
5865         // /* Follows STATE OF UGLINESS: */
5866         // Flushing = BufStart+(NumberOfSLOTS+1)*4 + 4; // '+ 4' in order to skip first 4 zeros
5867         // //in case of current buffer not have been used then NOT entering in this cycle
5868         // while(Flushing < bufend[i])
5869         // { if (*Flushing != 0) {fwrite(Flushing, 1, 1, fp_out ); TotalWLchars++;}
5870             // // Below 'Flushing-1' works due to skipped first 4 zeros!
5871             // if (*(Flushing-1) != 0 && *Flushing == 0) {fwrite(CRdLFa, 2, 1, fp_out);}
5872             // //last word must be suffixed with 1310 too
5873             // if (Flushing == bufend[i]-1) {fwrite(CRdLFa, 2, 1, fp_out );}
5874             // Flushing++;
5875         // }
5876
5877         for( j = 0; j < NumberOfSLOTS; j++ )
5878         {
5879             Slot = j<<2;
5880             memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
5881             if (PseudoLinkedPointer != 0)
5882             {
5883                 NumberOfTrees++;
5884                 if (BSTorBtree == 0)
5885                 {
5886                     // ===== BST traverse [
5887                     // DONE JOB:
5888                     // Must be written BST traverse ! with simulated stack i.e. non-recursive.
5889                     // ...
5890                     // /*
5891                     // Given a binary search tree, print out
5892                     // its data elements in increasing
5893                     // sorted order.
5894                     // */
5895                     // void printTree(struct node* node) {
5896                     // if (node == NULL) return;
5897                     // printTree(node->left);
5898                     // printf("%d ", node->data);
5899                     // printTree(node->right);
5900                     // }
5901
5902                     // FUTURE JOB:
5903                     // I need functions:
5904                     // BST_LeafNumber() // greater the better
5905                     // BST_NodeNumber() // 'BSTcurrent' below
5906                     // BST_Peak() // i.e. levels, root has height = 1
5907                     // BST_PeakIB() // IBBST(Ideal Balanced BST) has 1 + lgNodeNumber height
5908                     // I need 'Ideal Balancing BST FRAGMENT' with simulated stack:
5909                     // I need 'Ideal Balancing BST FRAGMENT' to be executed when Peak() >= PeakIB()<<1:
5910
5911                     // ----- [
5912                     BSTcurrentNode = 0; BSTcurrentPeak = 0; BSTcurrentLeaf = 0;
5913                     BSTcurrentPeakMAX = 0; // Height of current BST
5914                     StackPtr = 0;
5915                     while ( 2==2 ) {
5916                         while (PseudoLinkedPointer != 0)
5917                         {
5918                             if (StackPtr > 8192*3-1-3) { printf( "\nLeprechaun: Failure! BST simulated stack overflow, too high BST!\n" ); return( 13 );}
5919                             memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
5920                             PseudoLinkedPointer = PseudoLinkedPointer + 4;
5921                             memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer, 4 );
5922                             BSTstack[StackPtr] = PseudoLinkedPointer + 4; ++StackPtr; //ptr to wrd
5923                             BSTstack[StackPtr] = PseudoLinkedPointerNEWright; ++StackPtr;
5924                             BSTstack[StackPtr] = PseudoLinkedPointerNEWleft; ++StackPtr; //needed for stats not for recursion
5925                             // BST stats [

```

```

5926         if (PseudoLinkedPointerNEWleft == 0 && PseudoLinkedPointerNEWright == 0) {BSTcurrentLeaf++; BSTsTotalLEAFs++;}
5927         BSTcurrentPeak++;
5928         if (BSTcurrentPeakMAX < BSTcurrentPeak) BSTcurrentPeakMAX = BSTcurrentPeak;
5929         BSTstack[StackPtr] = BSTcurrentPeak; ++StackPtr; //needed for stats not for recursion
5930     // BST stats ]
5931     PseudoLinkedPointer = PseudoLinkedPointerNEWright; // choose right instead of 'PseudoLinkedPointerNEWleft' because of stats
print
5932 }
5933 if (StackPtr == 0) break;
5934 BSTcurrentPeak = BSTstack[--StackPtr]; // level of the node(1 is root) needed only for stats(print)
5935 PseudoLinkedPointerNEWleft = BSTstack[--StackPtr]; // left pointer needed only for stats(print)
5936 PseudoLinkedPointerNEWright = BSTstack[--StackPtr]; // right pointer
5937 memcpy( wrd, BSTstack[--StackPtr], i%31+1 );
5938 fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
5939 fwrite(CRdLFa, 2, 1, fp_out);
5940 BSTcurrentNode++;
5941 PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
5942 }
5943 // ----- ]
5944 // BST stats [
5945 if (BSTwithMAXnode < BSTcurrentNode) {
5946     BSTwithMAXnode = BSTcurrentNode;
5947     BSTwithMAXnodePEAK = BSTcurrentPeakMAX;
5948     BSTwithMAXnodeLEAF = BSTcurrentLeaf;
5949     BSTcurrentNodeMAXQUANTITY = 0;
5950 }
5951 if (BSTwithMAXnode == BSTcurrentNode) BSTcurrentNodeMAXQUANTITY++;
5952 if (BSTwithMAXpeak < BSTcurrentPeakMAX) {
5953     BSTwithMAXpeak = BSTcurrentPeakMAX;
5954     BSTwithMAXpeakNODE = BSTcurrentNode;
5955     BSTwithMAXpeakLEAF = BSTcurrentLeaf;
5956     BSTcurrentPeakMAXQUANTITY = 0;      iBSTwithMAXpeak=i; jBSTwithMAXpeak=j;
5957 }
5958 if (BSTwithMAXpeak == BSTcurrentPeakMAX) BSTcurrentPeakMAXQUANTITY++;
5959 if (BSTwithMAXleaf < BSTcurrentLeaf) {
5960     BSTwithMAXleaf = BSTcurrentLeaf;
5961     BSTwithMAXleafNODE = BSTcurrentNode;
5962     BSTwithMAXleafPEAK = BSTcurrentPeakMAX;
5963     BSTcurrentLeafMAXQUANTITY = 0;
5964 }
5965 if (BSTwithMAXleaf == BSTcurrentLeaf) BSTcurrentLeafMAXQUANTITY++;
5966 // BST stats ]
5967 // ===== BST traverse ]
5968 } else
5969 {
5970 // $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse [
5971 // DONE JOB:
5972 // Must be written B-tree traverse ! with simulated stack i.e. non-recursive.
5973 // ...
5974 StackPtr = 0;
5975 while ( 2==2 ) {
5976     while (PseudoLinkedPointer != 0)
5977     {
5978         if (StackPtr > 8192*3-1) { printf( "\nLeprechaun: Failure! B-tree simulated stack overflow, too high B-tree!\n" ); return( 13
5979 );}
5980         BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //ptr to Rwrld
5981         if ( *(char *) (PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) == 0 ) {memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTS*4], 4 );}
5982         memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
5983         memcpy( &PseudoLinkedPointerNEWMiddle, PseudoLinkedPointer + 4, 4 );
5984         memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
5985         // Give first from right to left non-zero PTR
5986         if (PseudoLinkedPointerNEWright !=0 )
5987         { memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTS*4], 4 );
5988           PseudoLinkedPointer = PseudoLinkedPointerNEWright;
5989         }
5990         else if (PseudoLinkedPointerNEWMiddle !=0 )
5991         { memcpy( PseudoLinkedPointer + 4, &BufStart[NumberOfSLOTS*4], 4 );
5992           PseudoLinkedPointer = PseudoLinkedPointerNEWMiddle;
5993         }
5994         else if (PseudoLinkedPointerNEWleft !=0 )
5995         { memcpy( PseudoLinkedPointer, &BufStart[NumberOfSLOTS*4], 4 );
5996           PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
5997         }
5998         else
5999         {
6000             PseudoLinkedPointer = 0;
6001         }
6002     }
6003     if (StackPtr == 0) break;
6004     PseudoLinkedPointer = BSTstack[--StackPtr];
6005     memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
6006     memcpy( &PseudoLinkedPointerNEWMiddle, PseudoLinkedPointer + 4, 4 );
6007     memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
6008     if (PseudoLinkedPointerNEWleft+PseudoLinkedPointerNEWMiddle+PseudoLinkedPointerNEWright == 0) // One LEAF is PRINTED when LP=0 MP=0
6009     {
6010         RP=0
6011         {
6012             memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4, i%31+1 );
6013             fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;

```

```

6011     fwrite(CRdLfa, 2, 1, fp_out);
6012     if ( *(char *) (PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) != 0 )
6013     { memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4 + i%31+1, i%31+1 );
6014         fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
6015         fwrite(CRdLfa, 2, 1, fp_out);
6016     }
6017     PseudoLinkedPointer = 0;
6018 }
6019 }
6020 // $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse ]
6021 }
6022     }
6023 } // j
6024 }
6025 } // i
6026
6027 if (BSTorBtree == 0)
6028 {
6029     // ~~~~ Longest path ~~~~ [
6030     i=iBSTwithMAXpeak; j=jBSTwithMAXpeak;
6031     BufStart = pointerflush + (i / 31) * WHOLEletter_BufferSize + OffsetsInBuffer[i % 31];
6032     Slot = j<<2;
6033     memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
6034     if (PseudoLinkedPointer != 0)
6035     {
6036         // ----- [
6037         BSTcurrentNode = 0; BSTcurrentPeak = 0; BSTcurrentLeaf = 0;
6038         BSTcurrentPeakMAX = 0; // Height of current BST
6039         StackPtr = 0;
6040         // BST print [
6041         fprintf( fp_outLOG, "A(not always THE) Binary-Search-Tree with the longest path(height, PEAK, number of levels):\n" );
6042         // BST print ]
6043         while ( 2==2 ) {
6044             while (PseudoLinkedPointer != 0)
6045             {
6046                 if (StackPtr > 8192*3-1-3) { printf( "\nLeprechaun: Failure! BST simulated stack overflow, too high BST!\n" ); return( 13 );}
6047                 memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
6048                 PseudoLinkedPointer = PseudoLinkedPointer + 4;
6049                 memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer, 4 );
6050                 BSTstack[StackPtr] = PseudoLinkedPointer + 4; ++StackPtr; //ptr to wrd
6051                 BSTstack[StackPtr] = PseudoLinkedPointerNEWright; ++StackPtr;
6052                 BSTstack[StackPtr] = PseudoLinkedPointerNEWleft; ++StackPtr; //needed for stats not for recursion
6053                 // BST stats [
6054                 if (PseudoLinkedPointerNEWleft == 0 && PseudoLinkedPointerNEWright == 0) {BSTcurrentLeaf++;} //BSTsTotalLEAFs++;} // REMOVED
6055                 to avoid mess in TOTAL stats
6056                 BSTcurrentPeak++;
6057                 if (BSTcurrentPeakMAX < BSTcurrentPeak) BSTcurrentPeakMAX = BSTcurrentPeak;
6058                 BSTstack[StackPtr] = BSTcurrentPeak; ++StackPtr; //needed for stats not for recursion
6059                 // BST stats ]
6060                 PseudoLinkedPointer = PseudoLinkedPointerNEWright; // choose right instead of 'PseudoLinkedPointerNEWleft' because of stats
6061                 print
6062             }
6063             if (StackPtr == 0) break;
6064             BSTcurrentPeak = BSTstack[--StackPtr]; // level of the node(1 is root) needed only for stats(print)
6065             PseudoLinkedPointerNEWleft = BSTstack[--StackPtr]; // left pointer needed only for stats(print)
6066             PseudoLinkedPointerNEWright = BSTstack[--StackPtr]; // right pointer
6067             memcpy( wrd, BSTstack[--StackPtr], i%31+1 );
6068             //fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
6069             //fwrite(CRdLfa, 2, 1, fp_out);
6070             BSTcurrentNode++;
6071             PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
6072             // BST print [
6073             for( k = 0; k < BSTcurrentPeak; k++ ) fprintf( fp_outLOG, "%c", ' ' );
6074             if (PseudoLinkedPointerNEWleft == 0) fprintf( fp_outLOG, "[" ); else fprintf( fp_outLOG, "]" );
6075             for( k = 0; k < i%31+1; k++ ) fprintf( fp_outLOG, "%c", *(char *) (wrd+k) );
6076             if (PseudoLinkedPointerNEWright == 0) fprintf( fp_outLOG, "]" ); else fprintf( fp_outLOG, "[" );
6077             if (BSTcurrentPeak == 1) fprintf( fp_outLOG, " ROOT" );
6078             fprintf( fp_outLOG, "\n" );
6079             // BST print ]
6080         }
6081         // ----- ]
6082         fprintf( fp_outLOG, "Above Binary-Search-Tree with MaxPEAK = %s has NODES = %s and LEAFs = %s\n", _ui64toaKAZEcomma(BSTwithMAXpeak,
6083             11ToaDigits2, 10), _ui64toaKAZEcomma(BSTwithMAXpeakNODE, 11ToaDigits3, 10), _ui64toaKAZEcomma(BSTwithMAXpeakLEAF, 11ToaDigits, 10));
6084         fprintf( fp_outLOG, "Legend:\n" );
6085         fprintf( fp_outLOG, "At left side of the word - '[' means no left successor\n" );
6086         fprintf( fp_outLOG, "At left side of the word - '[' means left successor exists\n" );
6087         fprintf( fp_outLOG, "At right side of the word - '[' means no right successor\n" );
6088         fprintf( fp_outLOG, "At right side of the word - '[' means right successor exists\n" );
6089         // ~~~~ Longest path ~~~~ ]
6090
6091         // BST stats [
6092         PEAKibBST=1+floorLog2(BSTwithMAXnode);
6093         //PEAKibBST=1;
6094         //while (BSTwithMAXnode>>PEAKibBST) PEAKibBST++;
6095         // BST stats ]
6096     }
6097 }

```

```

6096 (void) time(&t2);
6097 if (t2 <= t1) {t2 = t1; t2++;}
6098 printf("Time for making unsorted wordlist: %d second(s)\n", (int) t2-t1);
6099 fprintf( fp_outLOG, "Bytes per second performance: %sB/s\n", _ui64toaKAZEcomma(FilesLEN/((int) t3-t1), 11ToaDigits, 10) ); // Rev. 12+
6100 fprintf( fp_outLOG, "Words per second performance: %sW/s\n", _ui64toaKAZEcomma(WORDcount/((int) t3-t1), 11ToaDigits, 10) ); // Rev. 12+
6101 fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
6102 fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, 11ToaDigits, 10) );
6103 fprintf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, 11ToaDigits2, 10) );
6104 fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
6105 fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
6106 fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
6107 NumberOfHashCollisions = WORDcountDistinct - NumberOfTrees;
6108 fprintf( fp_outLOG, "Number Of Trees(GREATER THE BETTER): %lu\n", NumberOfTrees );
6109 fprintf( fp_outLOG, "Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): %lu%s\n",
(NumberOfTrees*100)/(26*31*8192), "%0" );
6110 fprintf( fp_outLOG, "Number Of Hash Collisions(Distinct WORDS - Number Of Trees): %lu\n", NumberOfHashCollisions );
6111
6112 if (BSTorBtree == 0)
6113 {
6114 fprintf( fp_outLOG, "Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '%s'\n", _ui64toaKAZEcomma(BSTwithMAXpeak, 11ToaDigits,
10) );
6115 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDS into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, 11ToaDigits,
10) );
6116 fprintf( fp_outLOG, "Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): %s\n", _ui64toaKAZEcomma(BSTsTotalLEAFs, 11ToaDigits,
10) );
6117 fprintf( fp_outLOG, "Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = %s must have PEAK = %s = rounding down of integer (1+lb(%s))\n",
_ui64toaKAZEcomma(BSTwithMAXnode, 11ToaDigits, 10), _ui64toaKAZEcomma(PEAKibBST, 11ToaDigits2, 10), _ui64toaKAZEcomma(BSTwithMAXnode,
11ToaDigits3, 10));
6118 fprintf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxNODEs = %s has PEAK = %s and LEAFs = %s\n",
_ui64toaKAZEcomma(BSTcurrentNodeMAXQUANTITY, 11ToaDigits4, 10), _ui64toaKAZEcomma(BSTwithMAXnode, 11ToaDigits2, 10),
_ui64toaKAZEcomma(BSTwithMAXnodePEAK, 11ToaDigits3, 10), _ui64toaKAZEcomma(BSTwithMAXnodeLEAF, 11ToaDigits, 10));
6119 fprintf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxPEAK = '%s' has NODEs = %s and LEAFs = %s\n",
_ui64toaKAZEcomma(BSTcurrentPeakMAXQUANTITY, 11ToaDigits4, 10), _ui64toaKAZEcomma(BSTwithMAXpeak, 11ToaDigits2, 10),
_ui64toaKAZEcomma(BSTwithMAXpeakNODE, 11ToaDigits3, 10), _ui64toaKAZEcomma(BSTwithMAXpeakLEAF, 11ToaDigits, 10));
6120 fprintf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxLEAFs = %s has NODEs = %s and PEAK = %s\n",
_ui64toaKAZEcomma(BSTcurrentLeafMAXQUANTITY, 11ToaDigits4, 10), _ui64toaKAZEcomma(BSTwithMAXleaf, 11ToaDigits2, 10),
_ui64toaKAZEcomma(BSTwithMAXleafNODE, 11ToaDigits3, 10), _ui64toaKAZEcomma(BSTwithMAXleafPEAK, 11ToaDigits, 10));
6121 } else
6122 {
6123 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDS into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, 11ToaDigits, 10)
);
6124 }
6125
6126 for( k = 1; k < 32; k++ )
6127 { fprintf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s utilization\n", _ui64toaKAZEzerocomma(k, 11ToaDigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, 11ToaDigits2, 10)+(26-5), _ui64toaKAZEzerocomma((((GRMBLh11[(int)k] *
LetterBuffer)/31)>>10)+1, 11ToaDigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLh11[(int)k] *
LetterBuffer)/31), 11ToaDigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
6128 if ( MAXusedBufferABS < (31 * ((MAXusedBuffer[k]>>10)+1)) / GRMBLh11[(int)k] ) {MAXusedBufferABS = 1+(31 * ((MAXusedBuffer[k]>>10)+1)) /
GRMBLh11[(int)k];}
6129 Utiliza1 = Utiliza1 + (MAXusedBuffer[k]>>10)+1;
6130 Utiliza2 = Utiliza2 + (((GRMBLh11[(int)k] * LetterBuffer)/31)>>10)+1;
6131 }
6132 fprintf( fp_outLOG, "Total pseudo(including hash table) memory utilization: %s%s\n", _ui64toaKAZEzerocomma((Utiliza1*100)/Utiliza2,
11ToaDigits, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
6133 fprintf( fp_outLOG, "Total real(wordlist's words VS allocated block) memory utilization: %s/1000\n", _ui64toaKAZE(((unsigned long
long)TotalWLchars*1000)/memory_size, 11ToaDigits, 10) ); // 26 are all 26-DESIRED=24
6134 fprintf( fp_outLOG, "used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
6135 fprintf( fp_outLOG, "use next time as third parameter: %lu-\n", MAXusedBufferABS ); // 26 are all 26-DESIRED=24
6136 fprintf( fp_outLOG, "Time for making unsorted wordlist: %d second(s)\n", (int) t2-t1);
6137
6138 // EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT
6139 printf( "Deallocated memory in MB: %lu\n", (memory_size>>20)+1 );
6140 free(pointerflushUNALIGN);
6141 fclose(fp_out);
6142 fclose(fp_outLOG);
6143 // SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT
6144 //printf("Uploading unsorted wordlist ...\n");
6145 if ((nlines = readlines(argv[2], &backup)) >= 0)
6146 { //printf("Number of words(lines) uploaded: %lu\n", nlines);
6147 //printf("Note1: Press 'Ctrl+C' to abort sorting, unsorted wordlist(second parameter)\n");
6148 //printf("will remain intact(unless flushing is in progress) because of\n");
6149 //printf("pointers-to-data are being sorted not the data itself.\n");
6150 //printf("Note2: In near future 'InsertionX26Sort' will be replaced with 'QuickX26Sort':\n");
6151 //printf("which is much faster than 'QuickSort' applied for data-at-once!\n");
6152
6153 // !!!!! I AM DISAPPOINTED x26 is just an illusion !!!!!
6154 // x26 x26 x26 x26 x26 x26 x26 x26 x26 x26 x26 x26 x26 x26 x26 x26
6155 // argc is 4|5|6 due to eventual missing BufferSize
6156 if( argc == 4 ) // not 5 due to eventual missing BufferSize
6157 k_FIX = 3;
6158 if( argc == 5 || argc == 6 )
6159 k_FIX = 4;
6160 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')
6161 { printf("Sorting(with 'MultikeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...\n");
6162 /* ?????! what an unexpected behavior! I have been hit: for SOED5.HTM(259,835 distinct words): InsertionX26Sort gives 46s, InsertionSort

```



```

gives 28s */
6163 for( k = 0; k < 26; k++ )
6164 { printf( "Sort pass %s/26 ...\r", _ui64toaKAZEzerocomma(k+1, llToaDigits, 10)+(26-2));
6165   HEADOffsetFromStartBUKVA = TAILOffsetFromStartBUKVA;
6166   while( (TAILOffsetFromStartBUKVA < nlines) && (*backup[TAILOffsetFromStartBUKVA] - 'a' == k) )
6167   { TAILOffsetFromStartBUKVA++;
6168   }
6169   if (HEADOffsetFromStartBUKVA != TAILOffsetFromStartBUKVA)
6170   { mkqsort_main(backup + HEADOffsetFromStartBUKVA, TAILOffsetFromStartBUKVA - HEADOffsetFromStartBUKVA + 0); // backup[0..nlines-1]
6171   }
6172 }
6173 }
6174 else
6175 {
6176 if (*argv[k_FIX] == 'A' || *argv[k_FIX] == 'a')
6177 { printf("Sorting(with 'InsertionSort') ...");
6178   InsertSortKAZE(backup, nlines, 0); // backup[0..nlines-1]
6179 }
6180 if (*argv[k_FIX] == 'B' || *argv[k_FIX] == 'b')
6181 { printf("Sorting(with 'InsertionX26Sort') ...\n");
6182   /* ???!!! what an unexpected behavior! I have been hit: for SOED5.HTM(259,835 distinct words): InsertionX26Sort gives 46s, InsertionSort
gives 28s */
6183 for( k = 0; k < 26; k++ )
6184 { printf( "Sort pass %s/26 ...\r", _ui64toaKAZEzerocomma(k+1, llToaDigits, 10)+(26-2));
6185   HEADOffsetFromStartBUKVA = TAILOffsetFromStartBUKVA;
6186   while( (TAILOffsetFromStartBUKVA < nlines) && (*backup[TAILOffsetFromStartBUKVA] - 'a' == k) )
6187   { TAILOffsetFromStartBUKVA++;
6188   }
6189   if (HEADOffsetFromStartBUKVA != TAILOffsetFromStartBUKVA)
6190   { InsertSortKAZE(backup + HEADOffsetFromStartBUKVA, TAILOffsetFromStartBUKVA - HEADOffsetFromStartBUKVA + 0, 0); // backup[0..nlines-1]
6191   }
6192 }
6193 }
6194 if (*argv[k_FIX] == 'C' || *argv[k_FIX] == 'c')
6195 { printf("Sorting(with 'MultiKeyQuickSortSort' by J. Bentley and R. Sedgewick) ...");
6196   mkqsort_main(backup, nlines); // backup[0..nlines-1]
6197 }
6198 if (*argv[k_FIX] == 'D' || *argv[k_FIX] == 'd')
6199 { printf("Sorting(with 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...\n");
6200   /* ???!!! what an unexpected behavior! I have been hit: for SOED5.HTM(259,835 distinct words): InsertionX26Sort gives 46s, InsertionSort
gives 28s */
6201 for( k = 0; k < 26; k++ )
6202 { printf( "Sort pass %s/26 ...\r", _ui64toaKAZEzerocomma(k+1, llToaDigits, 10)+(26-2));
6203   HEADOffsetFromStartBUKVA = TAILOffsetFromStartBUKVA;
6204   while( (TAILOffsetFromStartBUKVA < nlines) && (*backup[TAILOffsetFromStartBUKVA] - 'a' == k) )
6205   { TAILOffsetFromStartBUKVA++;
6206   }
6207   if (HEADOffsetFromStartBUKVA != TAILOffsetFromStartBUKVA)
6208   { mkqsort_main(backup + HEADOffsetFromStartBUKVA, TAILOffsetFromStartBUKVA - HEADOffsetFromStartBUKVA + 0); // backup[0..nlines-1]
6209   }
6210 }
6211 }
6212 }
6213 // X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26
6214
6215 printf("\nFlushing sorted words ...\n");
6216 if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
6217 { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
6218 for( j = 0; j < nlines; j++ )
6219 { //slot = KuxHash3plus(backup[j]);
6220   //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(slot, llToaDigits, 10)+(26-5));
6221   fprintf(fp_out, "%s", backup[j]);
6222   fwrite(CrDLfa, 2, 1, fp_out );
6223 }
6224 (void) time(&t3);
6225 if (t3 <= t2) {t3 = t2; t3++;}
6226 printf("Time for sorting unsorted wordlist: %d second(s)\n", (int) t3-t2);
6227
6228 /*
6229 // Hash benchmarking ----- [
6230
6231 // 5[
6232 clocks1 = clock();
6233 for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6234 {
6235   for( j = 0; j < nlines; j++ )
6236   { //slot = KuxHash3plus(backup[j]);
6237     //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(slot, llToaDigits, 10)+(26-5));
6238     slot = FNV1A_Hash_4_OCTETS(backup[j], (strlen(backup[j])>>2)); //13+++
6239   }
6240 }
6241 clocks2 = clock();
6242 printf( "Performance of 'FNV1A_Hash_4_OCTETS': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
((TotalWlchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6243 // 5]
6244
6245 // 1[
6246 clocks1 = clock();

```

```

6247     for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6248 {
6249     for( j = 0; j < nlines; j++ )
6250     { //Slot = KuxHash3plus(backup[j]);
6251         //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, 11ToaDigits, 10)+(26-5));
6252 // To make it EVEN !!!
6253 //wrklen = strlen(backup[j]);
6254 //if (strlen(backup[j]) != 0)
6255         Slot = FNV1A_Hash(backup[j]); //13+++
6256     }
6257 }
6258 clocks2 = clock();
6259 printf( "Performance of 'FNV1A_Hash': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
        ((TotalWLchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6260 // 1]
6261
6262 // 2[
6263 clocks1 = clock();
6264     for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6265 {
6266     for( j = 0; j < nlines; j++ )
6267     { //Slot = KuxHash3plus(backup[j]);
6268         //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, 11ToaDigits, 10)+(26-5));
6269         Slot = FNV1A_Hash_4_OCTETS_31(backup[j], (strlen(backup[j])>>2)); //13+++
6270     }
6271 }
6272 clocks2 = clock();
6273 printf( "Performance of 'FNV1A_Hash_4_OCTETS_31': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
        ((TotalWLchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6274 // 2]
6275
6276 // 4[
6277 clocks1 = clock();
6278     for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6279 {
6280     for( j = 0; j < nlines; j++ )
6281     { //Slot = KuxHash3plus(backup[j]);
6282         //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, 11ToaDigits, 10)+(26-5));
6283 // To make it EVEN !!!
6284 //wrklen = strlen(backup[j]);
6285 //if (strlen(backup[j]) != 0)
6286         Slot = KuxHash3plus(backup[j]); //13++
6287     }
6288 }
6289 clocks2 = clock();
6290 printf( "Performance of 'KuxHash3plus': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
        ((TotalWLchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6291 // 4]
6292
6293 // 6[
6294 clocks1 = clock();
6295     for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
6296 {
6297     for( j = 0; j < nlines; j++ )
6298     { //Slot = KuxHash3plus(backup[j]);
6299         //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, 11ToaDigits, 10)+(26-5));
6300 wrklen = strlen(backup[j]);
6301 if (wrklen<=19) // 4x4+3=19 i.e. last contains 7 clashes
6302         Slot = FNV1A_Hash_Granularity(backup[j], wrklen>>2, 2); //13++++
6303 else // 2x8+4=20 i.e. first contains 6 clashes
6304         Slot = FNV1A_Hash_Granularity(backup[j], wrklen>>3, 3); //13++++
6305     } // Conclusion: two functions > 64 bytes lead to horrible slowness, so unite them in one: fit in the cache line.
6306 }
6307 clocks2 = clock();
6308 printf( "Performance of 'FNV1A_Hash_Granularity': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4)) ,
        ((TotalWLchars>>10)/((long)(clocks2 - clocks1 + 1)>>4)) );
6309 // 6]
6310
6311 // Hash benchmarking ----- ]
6312 */
6313
6314 if( ( fp_outLOG = fopen( "Leprechaun.LOG", "a+" ) ) == NULL )
6315 { printf( "Leprechaun: Can't open file Leprechaun.LOG.\n" ); return( 1 ); }
6316 fprintf( fp_outLOG, "Time for sorting unsorted wordlist: %d second(s)\n", (int) t3-t2);
6317 printf( "Leprechaun: Done.\n" );
6318     return 0;
6319 }
6320 else
6321 { printf("Leprechaun: Input file too large, wordlist remains unsorted!\n");
6322     return 1;
6323 }
6324 // SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT
6325 } else { //if (BSTorBtree != 2) {
6326 // External Btrees [
6327
6328 // r16 [
6329 if (REUSE==2) {
6330     fclose(fp_out);

```

```

6331 }
6332 // r16 ]
6333
6334 if ( REUSE == 0 ) { // r16FIX <*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*> [
6335
6336 (void) time(&t1);
6337 WORDcountBOTTOM = 0;
6338 //printf("Flushing UNSorted phrases ...\r");
6339 //if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL ) // Commented since r.14+++ because passes need concatenation.
6340
6341 if( ( fp_out = fopen( argv[2], "ab+" ) ) == NULL )
6342 { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
6343 CRdLfA[0] = 13; CRdLfA[1] = 10;
6344
6345 BufStart = pointerflush;
6346 // for( j = 0; j < 28*28*28*28; j++ )
6347 for( j = 0; j < (1<<HashInBITS); j++ )
6348 {
6349     if ((j & ((1<<14)-1)) == 0) {
6350         (void) time(&t3);
6351         if (t3 <= t1) {t3 = t1; t3++;}
6352         printf("Flushing UNSorted phrases: %s%%; Shaking trees performance: %sP/s\r", _ui64toAKAZEzerocomma(((long
long)j*100)/((1<<HashInBITS)), 11ToADigits, 10)+(26-3), _ui64toAKAZEzerocomma(WORDcountBOTTOM/((int) t3-t1), 11ToADigits2, 10)+(26-10));
6353     }
6354     Slot = j<<3;
6355     memcpy( &PseudoLinkedPointer_64, BufStart+Slot, 8 );
6356     if (PseudoLinkedPointer_64 != 0)
6357     {
6358         NumberOfTrees++;
6359
6360 // $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse 64bit [
6361 // DONE JOB:
6362 // Must be written B-tree traverse ! with simulated stack i.e. non-recursive.
6363 // ...
6364 StackPtr = 0;
6365 while ( 2==2 ) {
6366     while (PseudoLinkedPointer_64 != 0)
6367     {
6368         if (StackPtr > 8192*3-1) { printf( "\nLeprechaun: Failure! B-tree simulated stack overflow, too high B-tree!\n" ); return( 13
);}
6369         BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //ptr to RwrD
6370 //if ( *(char *) (PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) == 0 ) {memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTS*4], 4 );}
6371 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6372 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4); //RW
6373 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6374 //fread(&SomeByte, 1, 1, fp_outRG);
6375 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6376 // [ //r.14+
6377 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
6378 if (BSTorBtree == 2) {
6379     fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6380     fread(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
6381 } else { // ##### 64bit memory manipulations [
6382     memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+8+2*(LongestLineInclusive+1+4) );
6383 } // ##### 64bit memory manipulations ]
6384 memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
6385 // ] //r.14+
6386 if (SomeByte == 0 ) // RW exists not
6387 {
6388     PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
6389     if (BSTorBtree == 2) {
6390         fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6391         fwrite(&NULLs_64, 8, 1, fp_outRG);
6392     } else { // ##### 64bit memory manipulations [
6393         memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLs_64, 8 );
6394     } // ##### 64bit memory manipulations ]
6395 // [ //r.14+
6396 memcpy( &LEAF[8 + 8], &NULLs_64, 8 );
6397 // ] //r.14+
6398 }
6399 // memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
6400 // memcpy( &PseudoLinkedPointerNEWMiddle, PseudoLinkedPointer + 4, 4 );
6401 // memcpy( &PseudoLinkedPointerNEWRright, PseudoLinkedPointer + 4 + 4, 4 );
6402 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6403 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
6404 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6405 //fread(&PseudoLinkedPointerNEWleft_64, 8, 1, fp_outRG);
6406 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
6407 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6408 //fread(&PseudoLinkedPointerNEWMiddle_64, 8, 1, fp_outRG);
6409 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
6410 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6411 //fread(&PseudoLinkedPointerNEWRright_64, 8, 1, fp_outRG);
6412 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6413 // [ //r.14+
6414 memcpy( &PseudoLinkedPointerNEWleft_64, &LEAF[0], 8 );
6415 memcpy( &PseudoLinkedPointerNEWMiddle_64, &LEAF[8], 8 );
6416 memcpy( &PseudoLinkedPointerNEWRright_64, &LEAF[8+8], 8 );

```

```

6417 // ] //r.14+
6418 // Give first from right to left non-zero PTR
6419 if (PseudoLinkedPointerNEWright_64 !=0 )
6420 { memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTS*4], 4 );
6421 // PseudoLinkedPointer = PseudoLinkedPointerNEWright;
6422 // }
6423 {
6424 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
6425 if (BStorBtree == 2) {
6426 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6427 fwrite(&NULLS_64, 8, 1, fp_outRG);
6428 } else { // ##### 64bit memory manipulations [
6429 memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLS_64, 8 );
6430 // ##### 64bit memory manipulations ]
6431 PseudoLinkedPointer_64 = PseudoLinkedPointerNEWright_64;
6432 }
6433 else if (PseudoLinkedPointerNEWmiddle_64 !=0 )
6434 // { memcpy( PseudoLinkedPointer + 4, &BufStart[NumberOfSLOTS*4], 4 );
6435 // PseudoLinkedPointer = PseudoLinkedPointerNEWmiddle;
6436 // }
6437 {
6438 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
6439 if (BStorBtree == 2) {
6440 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6441 fwrite(&NULLS_64, 8, 1, fp_outRG);
6442 } else { // ##### 64bit memory manipulations [
6443 memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLS_64, 8 );
6444 // ##### 64bit memory manipulations ]
6445 PseudoLinkedPointer_64 = PseudoLinkedPointerNEWmiddle_64;
6446 }
6447 else if (PseudoLinkedPointerNEWleft_64 !=0 )
6448 // { memcpy( PseudoLinkedPointer, &BufStart[NumberOfSLOTS*4], 4 );
6449 // PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
6450 // }
6451 {
6452 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
6453 if (BStorBtree == 2) {
6454 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6455 fwrite(&NULLS_64, 8, 1, fp_outRG);
6456 } else { // ##### 64bit memory manipulations [
6457 memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLS_64, 8 );
6458 // ##### 64bit memory manipulations ]
6459 PseudoLinkedPointer_64 = PseudoLinkedPointerNEWleft_64;
6460 }
6461 else
6462 {
6463 PseudoLinkedPointer_64 = 0;
6464 }
6465 }
6466 if (LevelsInCorona_Not_Counting_ROOT < StackPtr) LevelsInCorona_Not_Counting_ROOT = StackPtr; //r.14
6467 if (StackPtr == 0) break;
6468 PseudoLinkedPointer_64 = BSTstack[--StackPtr];
6469 // memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
6470 // memcpy( &PseudoLinkedPointerNEWmiddle, PseudoLinkedPointer + 4, 4 );
6471 // memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
6472 // [ //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6473 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
6474 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6475 //fread(&PseudoLinkedPointerNEWleft_64, 8, 1, fp_outRG);
6476 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
6477 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6478 //fread(&PseudoLinkedPointerNEWmiddle_64, 8, 1, fp_outRG);
6479 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
6480 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6481 //fread(&PseudoLinkedPointerNEWright_64, 8, 1, fp_outRG);
6482 // ] //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
6483 // [ //r.14+
6484 PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
6485 if (BStorBtree == 2) {
6486 fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
6487 fread(&LEAF[0], 8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
6488 } else { // ##### 64bit memory manipulations [
6489 memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+2*(LongestLineInclusive+1+4) );
6490 // ##### 64bit memory manipulations ]
6491 memcpy( &PseudoLinkedPointerNEWleft_64, &LEAF[0], 8 );
6492 memcpy( &PseudoLinkedPointerNEWmiddle_64, &LEAF[8], 8 );
6493 memcpy( &PseudoLinkedPointerNEWright_64, &LEAF[8+8], 8 );
6494 // ] //r.14+
6495 if (PseudoLinkedPointerNEWleft_64 + PseudoLinkedPointerNEWmiddle_64 + PseudoLinkedPointerNEWright_64 == 0) // One LEAF is PRINTED when
LP=0 MP=0
6496 {
6497 // memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4, i%31+1 );
6498 // fwrite(wrd, i%31+1, 1, fp_out);
6499 // fwrite(CRDLFa, 2, 1, fp_out);
6500 // if ( *(char *) (PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) != 0 )
6501 // { memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4 + i%31+1, i%31+1 );
6502 // fwrite(wrd, i%31+1, 1, fp_out);
6503 // fwrite(CRDLFa, 2, 1, fp_out);

```

[illegible]

```

6584 if ( (REUSE == 1) && ((HashInBITS-HashChunkSizeInBITS)==0) ) { // Multiple-passes shouldn't be dumped - it is meaningless, dump when only one
pass.
6585     if( ( fp_outRG = fopen( "Leprechaun_64bit.hsh", "wb+" ) ) == NULL )
6586     { printf( "Leprechaun: Can't create file 'Leprechaun_64bit.hsh'.\n" ); return( 1 ); }
6587     fwrite(pointerflushUNALIGN, (1<<HashInBITS)*8 + 1 + 64 , 1, fp_outRG);
6588 fclose(fp_outRG);
6589 }
6590     } else { // ##### 64bit memory manipulations [
6591 free(pointerflushUNALIGN_64);
6592     } // ##### 64bit memory manipulations ]
6593 free(pointerflushUNALIGN);
6594 fclose(fp_outLOG);
6595 printf( "Leprechaun: Current pass done.\n" );
6596
6597 // 14++++ [
6598 TotalMemoryNeededForOnePass += (((BufEnd_64-(unsigned long long)pointerflush_64)+1)>>10)+1;
6599 WORDcountDistinctTOTAL += WORDcountDistinct;
6600 RipPasses++;
6601 if (RipPasses <= (1<<(HashInBITS-HashChunkSizeInBITS))-1) goto whyTheHellForIsNotWorking;
6602 //} // for( RipPasses = 1-1; RipPasses <= (1<<(HashInBITS-HashChunkSizeInBITS))-1; RipPasses++ )
6603 // 14++++ ]
6604 (void) time(&tMainE);
6605 if (tMainE <= tMainB) {tMainE = tMainB; tMainE++;} // This line fixes a bug in r.15
6606 printf( "\nTotal memory needed for one pass: %sKB\n", _ui64toaKAZEcomma(TotalMemoryNeededForOnePass, 11ToaDigits2, 10) );
6607 printf( "Total distinct phrases: %s\n", _ui64toaKAZEcomma(WORDcountDistinctTOTAL, 11ToaDigits2, 10) );
6608 printf( "Total time: %d second(s)\n", (int) tMainE-tMainB);
6609 printf( "Total performance: %SP/s i.e. phrases per second\n", _ui64toaKAZEcomma(WORDcount/((int) tMainE-tMainB), 11ToaDigits2, 10) );
6610
6611 printf( "Leprechaun: Done.\n" );
6612 exit(0);
6613 } //if (BSTorBtree != 2) {
6614 } // main()
6615
6616 /*
6617 TO BE DONE: Ideal Balancing BST [
6618
6619 link rotR(link h)
6620 { link x = h->l; h->l = x->r; x->r = h;
6621     return x; }
6622
6623 link rotL(link h)
6624 { link x = h->r; h->r = x->l; x->l = h;
6625     return x; }
6626
6627 link parR(link h, int k)
6628 { int t = h->l->N;
6629     if (t > k)
6630         { h->l = parR(h->l, k); h = rotR(h); }
6631     if (t < k)
6632         { h->r = parR(h->r, k-t-1); h = rotL(h); }
6633     return h;
6634 }
6635
6636 link balancer(link h)
6637 {
6638     if (h->N < 2) return h;
6639     h = parR(h, h->N/2);
6640     h->l = balancer(h->l);
6641     h->r = balancer(h->r);
6642     return h;
6643 }
6644
6645 TO BE DONE: Ideal Balancing BST ]
6646 */
6647
6648 /*
6649 #include <stdlib.h>
6650 #include "Item.h"
6651 typedef struct STnode* link;
6652 struct STnode { Item item; link l, r; int N };
6653 static link head, z;
6654 link NEW(Item item, link l, link r, int N)
6655 { link x = malloc(sizeof *x);
6656     x->item = item; x->l = l; x->r = r; x->N = N;
6657     return x;
6658 }
6659 void STinit()
6660 { head = (z = NEW(NULLitem, 0, 0, 0)); }
6661 int STcount() { return head->N; }
6662 Item searchR(link h, Key v)
6663 { Key t = key(h->item);
6664     if (h == z) return NULLitem;
6665     if (eq(v, t) return h->item;
6666     if (less(v, t) return searchR(h->l, v);
6667     else return searchR(h->r, v);
6668 }
6669 Item STsearch(Key v)
6670 { return searchR(head, v); }

```

```

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

```