```
0001 // This is source of Leprechaun revision 16FIXFIX: Leprechaun_x-leton.c, copyleft Sammayce, 2013-Mar-31.
0002 // Grrr;;; Next two variables were not nullified between passes:
0003 // 16fixfix [
0004 // PLE_words_INITflag = 0;
0005 // PLE_words = 0;
0006 // 16fixfix ]
0007
0008 // This is source of Leprechaun revision 16FIX: Leprechaun_x-leton.c, copyleft Sammayce, 2012-Dec-16.
0009 // How embarrassing! A stupid bug was fixed, namely one missed 'if ( REUSE == 0 ) {}' holding the TRAVERSE segment - this segment nullifies
     LEAF addresses thus making W/W unable to retraverse.
0010
0011 // This is source of Leprechaun revision 16: Leprechaun_x-leton.c, copyleft Sammayce, 2012-Dec-13.
0012 // The new feature is the ability to reuse the external hash-tree structure.
0013 // The option is W/W similar to Z/Z. This way the latency i.e. the response time is <1s.
0014
0015 // This is source of Leprechaun revision 15FIXFIX+: Leprechaun_x-leton.c, copyleft Sammayce, 2012-Dec-11.
0016 // The new feature is the ability to command Leprechaun (from inside the list file with 2 metacommands) to enter/exit INSERT mode.
0017 // This allows to control whether new (to current hash-tree structure) x-grams are to be counted [and] INSERTed.
0018
0019 // Usage:
0020 // E:\_Gamera_r15_12348>type ON.txt
0021 // Leprechaun says x-gram inserting disabled for next files: ON
0022 //
0023 // E:\_Gamera_r15_12348>type OFF.txt
0024 // Leprechaun says x-gram inserting disabled for next files: OFF
0025 //
0026 // E:\_Gamera_r15_12348>dir Your_textual_folders/b/s/a-d>go.lst
0027 // E:\_Gamera_r15_12348>copy go.lst+on.txt+_Gamera.tar.3.sorted.4andabove.lst MetaLep.lst /b
0028 // E:\_Gamera_r15_12348>type MetaLep.lst
0029 // E:\_Gamera_r15_12348\Your_textual_folders\Example.txt
0030 // Leprechaun says x-gram inserting disabled for next files: ON
0031 // _Gamera.tar.3.sorted.4andabove.txt
0032 //
0033 // E:\_Gamera_r15_12348>Leprechaun_x-leton_META_32bit_03_01p.exe MetaLep.lst MetaLep.3.wrd 1234567 Y
0034
0035 // All lines new to r15FIXFIX are with commented part //15FIXFIX+
0036 /*
0038 This is source of Leprechaun revision 15FIXFIX: Leprechaun_x-leton.c, copyleft Sammayce, 2011-Dec-14. 2011-Mar-07: Fixed a small command line
     parsing bug.
0039
0040 The 15FIXFIX differs from 15fix with:
0041 [a bug fixed(REALLY FIXED!): Fixed a nasty bug causing very restrictive way of forming x-grams.]
0042 The 15fix differs from 15 with:
0043 [a bug fixed: division by zero when finishing-starting time is under 1 second
0044 Fixed a nasty bug causing very restrictive way of forming x-grams.]
0045 The 15 differs from 14++++FIXFIX with:
0046 [
0047 Only some more stats at the end.
0048 ]
0049 The 14++++FIXFIX differs from 14+++FIX with:
0050 [
0051 Bugs in LOG stats in r.14+++FIX:
0052 Not nullified variables during passes - must be nullified.
0053 Number of Trees(GREATER THE BETTER): 195,939
0054 Number of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 654,428
0055 Total Attempts to Find/Put WORDS into B-trees order 3: 391,042,828
0056 Highest Tree not counting ROOT Level i.e. CORONA Level(littler THE BETTER): 3
0057 ]
0058 The 14+++FIX differs from 14+++ with:
0059 [
0060 1)Fixed occurrencies bug due to not NULLifying the field housing the occurrencies, a nasty thing: all the revisions 14??? were buggy, how
     stupid from my side, grrrr.
0061 2)Ability to rip in passes:
0062 #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).
0063 ]
0064 The 14+++ differs from 14++ with:
0065 [
0066 //only one must be uncommented:
0067 //#define singleton
0068 //#define doubleton
0069 //#define tripleton
0070 //#define quadrupleton
0071 //#define quintupleton
0072 //#define sextupleton
0073 //#define septupleton
0074 //#define octupleton
0075 //#define nonupleton
0076 //#define decupleton
0077 1   One     single,    singleton
0078 2   Two     double,    doubleton
0079 3   Three   triple,    tripleton
0080 4   Four    quadruple, quadrupleton
0081 5   Five    quintuple, quintupleton
0082 6   Six     sextuple,  sextupleton
0083 7   Seven   septuple,  septupleton
0084 8   Eight   octuple,   octupleton
```

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

```
9       Nine    nonuple,   nonuplet,   nonupleton
10      Ten,    decuple,   decuplet,   decupleton
1    One       ace, single, singleton, unary, unit, unity
2    Two       binary, brace, couple, couplet, distich, deuce, double, doubleton, duad, duality, duet, duo, dyad, pair, snake eyes, span, twain, twosome, yoke
3    Three     deuce-ace, leash, set, tercet, ternary, ternion, terzetto, threesome, tierce, trey, triad, trine, trinity, trio, triplet, troika, hat-trick
4    Four      foursome, quadruplet, quatern, quaternary, quaternion, quaternity, quartet, tetrad
5    Five      cinque, fin, fivesome, pentad, quint, quintet, quintuplet
6    Six       half dozen, hexad, sestet, sextet, sextuplet, sise
7    Seven     heptad, septet, septuplet
8    Eight     octad, octave, octet, octonary, octuplet, ogdoad
Also, in addition to 'Y' and 'Z', 'y' and 'z' were added in order to be able to dump only n-grams without occurrencies.
]
A lazy approach is applied in order to add occurrences of each 4-gram:
- just reserve the last 4bytes in 'wrd' for counter as follows:
  'LongestLineInclusive' has to be greater than 31 (51 looks good enough) in order not to miss longer 4-grams like: encourage_innovative_approaches_to
  char FourGram[LongestLineInclusive+1+4]; // 31bytes longest 4-gram + 1byte NULL + 4bytes COUNTER
- the laziness lies here:
  no need to make the four bytes to house the value 1 when a new 'wrd' is being inserted (either in step 1 or step 3) just add 1 at final traverse dump,
  in step 1 when a 'wrd' is found then add 1 to the counter only if it is not 9,999,999 already (limitation enforced on counter).
- when dumping the format has to be:
  0,000,001\ta_b_c_d
  in order to sort the whole lines later with external Qsort and have easy screening for rare/wrong/useless 4-grams.

Comment/Uncomment accordingly in order to compile:
#define _WIN32_ENVIRONMENT_
//#define _POSIX_ENVIRONMENT_

Windows compile(uncomment #include <io.h> line, ignore warnings):
For Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86 use:
cl /Ox /Wp64 /TcLeprechaun.c /FaLeprechaun

Windows compile(comment #include <io.h> line, ignore warnings):
For Intel(R) C++ Compiler Professional for applications running on IA-32, Version 11.1 use:
icl /Ox /Wp64 /TcLeprechaun.c /FaLeprechaun /w /QxHOST

Linux compile(ignore warnings):
gcc -D_FILE_OFFSET_BITS=64 -m64 -static -O3 -mtune=generic Leprechaun_quadrupleton.c -o Leprechaun_quadrupleton_r14_generic_64bits.elf

!!! 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:
gcc -D_FILE_OFFSET_BITS=64 -m32 -static -O3 -mtune=generic Leprechaun_quadrupleton.c -o Leprechaun_quadrupleton_r14_generic_32bits.elf

[It's a little weird(Intel boosts the sort while falls behind in parsing, tested on T3400):]

Leprechaun_r13_7pluses_Microsoft_32-bit_16.00.30319.01.exe _vs_ Wikipedia_22,202,980_LATIN-Words:
Words per second performance: 1,679,585W/s
Time for making unsorted wordlist: 30 second(s)
Time for sorting unsorted wordlist: 25 second(s)

Leprechaun_r13_7pluses_Intel_IA-32_11.1.exe _vs_ Wikipedia_22,202,980_LATIN-Words:
Words per second performance: 1,603,240W/s
Time for making unsorted wordlist: 31 second(s)
Time for sorting unsorted wordlist: 19 second(s)

Due to my ignorance(calderas in my C knowledge): 64bit code cannot be generated, for now.
Any improvement is welcome.
Enjoy!
*/

// C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_r13++++++_C_EXE>cl /Ox /Wp64 /TcLeprechaun.c /FaLeprechaun
// Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
// Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
//
// Leprechaun.c
// Leprechaun.c(829) : warning C4312: 'type cast' : conversion from 'int' to 'string' of greater size
// Leprechaun.c(849) : warning C4312: 'type cast' : conversion from 'int' to 'string * ' of greater size
// Leprechaun.c(2048) : warning C4312: 'type cast' : conversion from 'int' to 'char *' of greater size
// Leprechaun.c(2063) : warning C4311: 'type cast' : pointer truncation from 'char *' to 'unsigned long'
// Leprechaun.c(2068) : warning C4311: 'type cast' : pointer truncation from 'char *' to 'unsigned long'
// Leprechaun.c(2371) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Leprechaun.c(2570) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Leprechaun.c(2626) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Leprechaun.c(2657) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Leprechaun.c(2663) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Leprechaun.c(2668) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Leprechaun.c(2696) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Leprechaun.c(2729) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Leprechaun.c(2743) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Leprechaun.c(2755) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
// Microsoft (R) Incremental Linker Version 7.10.3077
// Copyright (C) Microsoft Corporation.  All rights reserved.
//
// /out:Leprechaun.exe
// Leprechaun.obj
```



```
//
// C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_r13++++++_C_EXE>

/*
Below is the gain in 13++ and 13+++:

Words per second performance: 5,974,513W/s
Word count: 4,582,451,898 of them 9,177,221 distinct
Number Of Trees(GREATER THE BETTER): 2855919
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 6321302

Words per second performance: 6,329,353W/s
Word count: 4,582,451,898 of them 9,177,221 distinct
Number Of Trees(GREATER THE BETTER): 2958681
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 6218540

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.
Thanks to Fowler/Noll/Vo hash inventors.
*/

/*
Let's see the supplementary-clash on Intel Pentium T3400 Merom-1M 2166MHz:
Binary-Search-Trees vs B-Trees of order 3

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
Leprechaun(Fast Greedy Word-Ripper), revision 13++++++, written by Svalqyatchx.
Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
      also the performance of a 3-way hash + 6,602,752 B-Trees of order 3.
Size of input file with files for Leprechauning: 27
Allocating memory 1863MB ... OK
Size of Input TEXTual file: 146,973,879
\; Word count: 12,561,874 of them 12,561,874 distinct; Done: 64/64
Bytes per second performance: 14,697,387B/s
Words per second performance: 1,256,187W/s
Flushing unsorted words ...
Time for making unsorted wordlist: 15 second(s)
Deallocated memory in MB: 1863
Allocated memory for words in MB: 141
Allocated memory for pointers-to-words in MB: 48
Sorting(with 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...
Sort pass 26/26 ...
Flushing sorted words ...
Time for sorting unsorted wordlist: 14 second(s)
Leprechaun: Done.

[An excerpt of Leprechaun.LOG:]
Number Of Trees(GREATER THE BETTER): 2786806
Total Attempts to Find/Put WORDs into Binary-Search-Trees: 58,935,172
Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,806

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
Leprechaun(Fast Greedy Word-Ripper), revision 13++++++, written by Svalqyatchx.
Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
      also the performance of a 3-way hash + 6,602,752 B-Trees of order 3.
Size of input file with files for Leprechauning: 27
Allocating memory 1863MB ... OK
Size of Input TEXTual file: 146,973,879
\; Word count: 12,561,874 of them 12,561,874 distinct; Done: 64/64
Bytes per second performance: 24,495,646B/s
Words per second performance: 2,093,645W/s
Flushing unsorted words ...
Time for making unsorted wordlist: 12 second(s)
Deallocated memory in MB: 1863
Allocated memory for words in MB: 141
Allocated memory for pointers-to-words in MB: 48
Sorting(with 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...
Sort pass 26/26 ...
Flushing sorted words ...
Time for sorting unsorted wordlist: 14 second(s)
Leprechaun: Done.

[An excerpt of Leprechaun.LOG:]
Number Of Trees(GREATER THE BETTER): 2786806
Total Attempts to Find/Put WORDs into B-trees order 3: 18,534,910

C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST>type Leprechaun_vs_Wikipedia_en-WORDS.lst
wikipedia-en-html.tar.wrd

C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST>dir Leprechaun_vs_Wikipedia_en-WORDS.*
 Volume in drive C is H320_Vol2
 Volume Serial Number is A094-FAE2

 Directory of C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST
```

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



```
0340 0812 for (; wrdlen_QUADRUPLETS != 0; --wrdlen_QUADRUPLETS) {
0341 0813     hash = FNV_32A_OP32(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
0342 0814     p=p+4; // add eax, 4
0343 0815 }
0344 0816
0345 0817 // Stage #2:
0346 0818 for (; *p; ++p) {
0347 0819     hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [eax]
0348 0820 }
0349 0821
0350 0822   //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
0351 0823   return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
0352 0824 }
0353
0354 Assembler code:
0355 _FNV1A_Hash_4_OCTETS PROC NEAR
0356 ; Line 812
0357 mov  edx, DWORD PTR _wrdlen_QUADRUPLETS$[esp-4]
0358 test edx, edx
0359 mov  eax, DWORD PTR _str$[esp-4]
0360 push esi
0361 mov  esi, DWORD PTR _FNV1_32_PRIME
0362 mov  ecx, -2128831035
0363 je   SHORT $L1612
0364 push edi
0365 npad 7
0366 $L1610:
0367 ; Line 813
0368 mov  edi, DWORD PTR [eax]
0369 xor  edi, ecx
0370 imul edi, esi
0371 ; Line 814
0372 add  eax, 4
0373 dec  edx
0374 mov  ecx, edi
0375 jne  SHORT $L1610
0376 pop  edi
0377 $L1612:
0378 ; Line 818
0379 mov  dl, BYTE PTR [eax]
0380 test dl, dl
0381 je   SHORT $L1619
0382 $L1617:
0383 ; Line 819
0384 movzx         edx, dl
0385 xor  edx, ecx
0386 imul edx, esi
0387 inc  eax
0388 mov  ecx, edx
0389 mov  dl, BYTE PTR [eax]
0390 test dl, dl
0391 jne  SHORT $L1617
0392 $L1619:
0393 ; Line 823
0394 mov  eax, ecx
0395 shr  eax, 16
0396 xor  eax, ecx
0397 and  eax, 8191
0398 pop  esi
0399 ; Line 824
0400 ret  0
0401 _FNV1A_Hash_4_OCTETS ENDP
0402
0403
0404 So, 'FNV1A_Hash_4_OCTETS' calculates faster and gives better distribution(3549448 for 1607), which is 0.6% better(less collisions), than
     generic 'FNV1A_Hash' with 3527916.
0405
0406 FNV proves to be great, dealing with 4x8bits(four octets) at once doesn't hurt distribution at all, I was amazed by consistency(stable
     behaviour) of 'FNV1A_Hash_4_OCTETS'.
0407
0408 I want to make a total clash of all possible pairs 'FNV1_32_INIT' & 'FNV1_32_PRIME' in order to lessen even a few thousand collisions.
0409 This is critical for speed performance e.g. when 30,974,750,142 words, the case of wikipedia-en-html.tar, must be hashed.
0410 The current obstacle is needed-time: each filling (26 slots x 31 sub-slots x 8192 sub-sub-slots) executes in 32-36 seconds for each pair.
0411 Such an easy task, but I can't see how to get done, it is not hard but slow even with 15 times faster testbed.
0412
0413 Between 1..1166136247 there are 58,834,113 primes (inclusive).
0414 Between 1..16777619 there are 1,077,891 primes (inclusive).
0415 Or 58834113*1077891 = 63,416,760,895,683 pairs or 2,010,932 years needed at one-pair-per-second rate.
0416
0417 Finding THE best pair in my opinion is a total alchemy, due to the very nature of hashing: which is mainly alchemical and partly scientific.
0418 Since the magnum corpus of words is static-enough, THE pair is worthy to be found.
0419
0420 It doesn't take a think-tank to see the superiority of FNV, Fowler/Noll/Vo did reveal a thing of beauty.
0421
0422 Performance of 'FNV1A_Hash_4_OCTETS': 10236 words/clock or 105 MB/s|3,549,448 used slots (best)
0423
0424 CASE #1: with 'if (strlen(backup[j]) != 0)' before each execution
0425 Performance of 'KuxHash3plus' aka '2in1': 8076 words/clock or 82 MB/s|3,410,463 used slots (worst)
```

```
Performance of 'FNV1A_Hash': 8079 words/clock or 83 MB/s|3,527,916 used slots
Performance of 'FNV1A_Hash_SHIFTless_XORless': 8109 words/clock or 83 MB/s|3,540,323 used slots

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

Note:
The 'strlen' overhead(CASE #1) is necessary due to priorly(before hash invocation) needed len-of-string for 'FNV1A_Hash_4_OCTETS'.
Almost always, that is the case, since parsing of incoming text must know length of words/lines/files.
In case of not knowing this length: ((119-105)/105)*100% = 13% degradation is unacceptable.
The 'strlen' is an awful brake.
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.
*/

/*
FNV1_32_PRIME: //?: 16777619

Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
Words per second performance: 1,046,822W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
Size of all TEXTual Files: 146,973,879
Word count: 12,561,874 of them 12,561,874 distinct
Number Of Trees(GREATER THE BETTER): 2775839

Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 72 and LEAFs = 15
Words per second performance: 1,356,588W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Trees(GREATER THE BETTER): 3539690

FNV1_32_PRIME: //3549448: 1607

Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
Words per second performance: 1,046,822W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
Size of all TEXTual Files: 146,973,879
Word count: 12,561,874 of them 12,561,874 distinct
Number Of Trees(GREATER THE BETTER): 2783970

Above Binary-Search-Tree with MaxPEAK = 38 has NODEs = 50 and LEAFs = 11
Words per second performance: 1,410,851W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Trees(GREATER THE BETTER): 3549395

FNV1_32_PRIME: //3550132: 175757909

Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
Words per second performance: 966,298W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
Size of all TEXTual Files: 146,973,879
Word count: 12,561,874 of them 12,561,874 distinct
Number Of Trees(GREATER THE BETTER): 2784479

Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 64 and LEAFs = 12
Words per second performance: 1,410,851W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Trees(GREATER THE BETTER): 3550115

FNV1_32_PRIME: //3550687: 201887489

Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
Words per second performance: 966,298W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
Size of all TEXTual Files: 146,973,879
Word count: 12,561,874 of them 12,561,874 distinct
Number Of Trees(GREATER THE BETTER): 2784377

Above Binary-Search-Tree with MaxPEAK = 40 has NODEs = 55 and LEAFs = 11
Words per second performance: 1,356,588W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Trees(GREATER THE BETTER): 3550528

FNV1_32_PRIME: //3550733: 172783361

Above Binary-Search-Tree with MaxPEAK = 59 has NODEs = 59 and LEAFs = 1
Words per second performance: 1,046,822W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
Size of all TEXTual Files: 146,973,879
```



```
Word count: 12,561,874 of them 12,561,874 distinct
Number Of Trees(GREATER THE BETTER): 2786362

Above Binary-Search-Tree with MaxPEAK = 38 has NODEs = 70 and LEAFs = 17
Words per second performance: 1,410,851W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Trees(GREATER THE BETTER): 3550746

FNV1_32_PRIME: //3550929: 204312319

Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
Words per second performance: 966,298W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
Size of all TEXTual Files: 146,973,879
Word count: 12,561,874 of them 12,561,874 distinct
Number Of Trees(GREATER THE BETTER): 2785581

Above Binary-Search-Tree with MaxPEAK = 37 has NODEs = 55 and LEAFs = 12
Words per second performance: 1,356,588W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Trees(GREATER THE BETTER): 3550886

Leprechaun_Microsoft.exe: FNV1_32_PRIME: //3551736: 107712257

Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
Words per second performance: 1,046,822W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
Size of all TEXTual Files: 146,973,879
Word count: 12,561,874 of them 12,561,874 distinct
Number Of Trees(GREATER THE BETTER): 2786515

Above Binary-Search-Tree with MaxPEAK = 36 has NODEs = 64 and LEAFs = 15
Words per second performance: 1,356,588W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Trees(GREATER THE BETTER): 3551744

Leprechaun_Intel.exe: FNV1_32_PRIME: //3551736: 107712257

Above Binary-Search-Tree with MaxPEAK = 61 has NODEs = 61 and LEAFs = 1
Words per second performance: 1,256,187W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
Size of all TEXTual Files: 146,973,879
Word count: 12,561,874 of them 12,561,874 distinct
Number Of Trees(GREATER THE BETTER): 2786515

Above Binary-Search-Tree with MaxPEAK = 36 has NODEs = 64 and LEAFs = 15
Words per second performance: 1,603,240W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Trees(GREATER THE BETTER): 3551744

Wow: 1,603,240W/s vs 1,356,588W/s respectively Leprechaun_Intel.exe vs Leprechaun_Microsoft.exe, i.e. 18% betterment, no joke!

Alchemical search for best PRIME-PAIR revision uses next line:
Slot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13++++
This revision uses next lines:
if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
                Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>2, 2)<<2; //13+++++
else            // 2x8+4=20 i.e. first contains 6 clashes
                Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>3, 3)<<2; //13+++++

! 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.

Leprechaun.exe: FNV1_32_PRIME: //3551961: 428904191

Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
Words per second performance: 966,298W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
Size of all TEXTual Files: 146,973,879
Word count: 12,561,874 of them 12,561,874 distinct
Number Of Trees(GREATER THE BETTER): 2786383

Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 71 and LEAFs = 16
Words per second performance: 1,410,851W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Trees(GREATER THE BETTER): 3551503

Leprechaun.exe: FNV1_32_PRIME: //3552103: 588411137
```

```
0600
0601 Above Binary-Search-Tree with MaxPEAK = 6 has NODEs = 6 and LEAFs = 1
0602 Size of all TEXTual Files: 4,067,439
0603 Word count: 358,798 of them 351,116 distinct
0604 Number Of Trees(GREATER THE BETTER): 310622
0605 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 310,622
0606
0607 Above Binary-Search-Tree with MaxPEAK = 60 has NODEs = 60 and LEAFs = 1
0608 Size of all TEXTual Files: 146,973,879
0609 Word count: 12,561,874 of them 12,561,874 distinct
0610 Number Of Trees(GREATER THE BETTER): 2786485
0611 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,485
0612
0613 Above Binary-Search-Tree with MaxPEAK = 39 has NODEs = 62 and LEAFs = 15
0614 Size of all TEXTual Files: 415,982,896
0615 Word count: 35,271,297 of them 22,202,980 distinct
0616 Number Of Trees(GREATER THE BETTER): 3551956
0617 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,131
0618
0619 Leprechaun.exe: FNV1_32_PRIME: //3552039: 602173697 !!!GOODEST so far!!!
0620
0621 Above Binary-Search-Tree with MaxPEAK = 6 has NODEs = 6 and LEAFs = 1
0622 Size of all TEXTual Files: 4,067,439
0623 Word count: 358,798 of them 351,116 distinct
0624 Number Of Trees(GREATER THE BETTER): 310948
0625 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 310,948
0626
0627 Above Binary-Search-Tree with MaxPEAK = 63 has NODEs = 63 and LEAFs = 1
0628 Size of all TEXTual Files: 146,973,879
0629 Word count: 12,561,874 of them 12,561,874 distinct
0630 Number Of Trees(GREATER THE BETTER): 2786806
0631 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,806
0632
0633 Above Binary-Search-Tree with MaxPEAK = 36 has NODEs = 52 and LEAFs = 9
0634 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0635 Size of all TEXTual Files: 415,982,896
0636 Word count: 35,271,297 of them 22,202,980 distinct
0637 Number Of Trees(GREATER THE BETTER): 3552296
0638 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,899
0639
0640 Between 1 and 602392027 at step 100 following FNV1_32_PRIMEs(for FNV1_32_INIT=2166136261) give(FNV1A_Hash_4_OCTETS) dispersion:
0641 3550022: 423779327
0642 3550028: 513793537
0643 3550053: 434840321
0644 3550067: 437062229
0645 3550080: 420344321
0646 3550090: 304777471
0647 3550097: 496547839
0648 3550129: 390809599
0649 3550132: 175757909
0650 3550163: 353712127
0651 3550231: 334434817
0652 3550237: 272789761
0653 3550247: 590341121
0654 3550255: 358814207
0655 3550277: 437182721
0656 3550326: 521795327
0657 3550347: 311867393
0658 3550447: 456137729
0659 3550458: 418208767
0660 3550516: 602048767
0661 3550525: 513597697
0662 3550526: 347283199
0663 3550528: 598773503
0664 3550592: 598139137
0665 3550598: 242448127
0666 3550611: 571481087
0667 3550628: 457012993
0668 3550664: 482822143
0669 3550666: 249098753
0670 3550687: 201887489
0671 3550702: 489976063
0672 3550710: 272961023
0673 3550733: 172783361
0674 3550734: 431562497
0675 3550929: 204312319
0676 3550984: 562853633
0677 3550991: 551362303
0678 3551359: 332820737
0679 3551484: 354126079
0680 3551514: 407138561
0681 3551523: 442058753
0682 3551701: 449230849
0683 3551736: 107712257
0684 3551961: 428904191
0685 3552039: 602173697
0686 3552103: 588411137
0687 */
```



```
0688
0689 // Windows: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
0690 //  _CRTIMP size_t __cdecl fread(void *, size_t, size_t, FILE *);
0691 //  _CRTIMP size_t __cdecl fwrite(const void *, size_t, size_t, FILE *);
0692 //  _CRTIMP int __cdecl fgetpos(FILE *, fpos_t *);
0693 //  _CRTIMP int __cdecl fsetpos(FILE *, const fpos_t *);
0694
0695 //  _CRTIMP __int64 __cdecl _lseeki64(int, __int64, int);
0696 //  _CRTIMP __int64 __cdecl _telli64(int);
0697 //  _CRTIMP __int64 __cdecl _filelengthi64(int);
0698 //  above 3 are in 'io.h'
0699
0700 //  _CRTIMP int __cdecl fseek(FILE *, long, int);
0701 //  _CRTIMP long __cdecl ftell(FILE *);
0702 //  _CRTIMP int __cdecl fclose(FILE *);
0703
0704 //  #ifndef _SIZE_T_DEFINED
0705 //  #ifdef  _WIN64
0706 //  typedef unsigned __int64    size_t;
0707 //  #else
0708 //  typedef _W64 unsigned int   size_t;
0709 //  #endif
0710 //  #define _SIZE_T_DEFINED
0711 //  #endif
0712
0713 //  typedef __int64 fpos_t;
0714
0715 // Linux: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
0716 //  size_t fread (void *data, size_t size, size_t count, FILE *stream)
0717 //  size_t fwrite (const void *data, size_t size, size_t count, FILE *stream)
0718 //  int fgetpos (FILE *stream, fpos_t *position)
0719 //  int fsetpos (FILE *stream, const fpos_t *position)
0720
0721 //  FILE * fopen64 (const char *filename, const char *opentype)
0722 //  int fseeko64 (FILE *stream, off64_t offset, int whence)
0723 //  off64_t ftello64 (FILE *stream)
0724 //  int fclose (FILE *stream)
0725
0726 //  off_t lseek (int filedes, off_t offset, int whence)
0727 //  above 1 is in 'unistd.h'
0728
0729
0730 // ============ MUST work both for Windows and Linux =============
0731 //Only one must be uncommented:
0732 #define _WIN32_ENVIRONMENT_
0733 //#define _POSIX_ENVIRONMENT_
0734
0735 //Only one must be uncommented:
0736 //#define singleton
0737 //#define doubleton
0738 #define tripleton
0739 //#define quadrupleton
0740 //#define quintupleton
0741 //#define sextupleton
0742 //#define septupleton
0743 //#define octupleton
0744 //#define nonupleton
0745 //#define decupleton
0746
0747 #ifdef singleton
0748 #define _ngram_ 1
0749 #endif
0750 #ifdef doubleton
0751 #define _ngram_ 2
0752 #endif
0753 #ifdef tripleton
0754 #define _ngram_ 3
0755 #endif
0756 #ifdef quadrupleton
0757 #define _ngram_ 4
0758 #endif
0759 #ifdef quintupleton
0760 #define _ngram_ 5
0761 #endif
0762 #ifdef sextupleton
0763 #define _ngram_ 6
0764 #endif
0765 #ifdef septupleton
0766 #define _ngram_ 7
0767 #endif
0768 #ifdef octupleton
0769 #define _ngram_ 8
0770 #endif
0771 #ifdef nonupleton
0772 #define _ngram_ 9
0773 #endif
0774 #ifdef decupleton
0775 #define _ngram_ 10
```

```
#endif

#ifndef NULL
#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void*)0)
#endif
#endif

#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)!
#define HashChunkSizeInBITS 19 // Defines the number of passes. Should be smaller or equal to HashInBITS. If HashInBITS == HashChunkSizeInBITS then 2^(HashInBITS-HashChunkSizeInBITS)=2^0=1 pass(es).
/*
Tests done on super-speed-ramdisk 1800MB:
Leprechaun_quadrupleton rev. 14+ in fact differs from r.14 only with optimized(LEAFwise) fragment 1] and 2]. Fragmet 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.

[Variant (HashInBITS 26 - 0) with 512MB hash:]

Leprechaun_quadrupleton (Fast Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
Feature2: The bottleneck is seek-time, if the external memory has latency 100+microseconds then look further.
Size of input file with files for Leprechauning: 19
Allocating HASH memory 536,870,977 bytes ... OK
Allocating/ZEROing 1,292,478,478 bytes swap file ... OK
Size of Input TEXTual file: 206,908,949
|; 0,065,139P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
Bytes per second performance: 718,433B/s
Phrases per second performance: 65,139P/s
Time for putting phrases into trees: 288 second(s)
Flushing UNsorted phrases: 100%; Shaking trees performance: 0,014,439P/s
Time for shaking phrases from trees: 704 second(s)
Dump LEAFwise also [
Bytes per second performance: 736,330B/s
Phrases per second performance: 66,762P/s
Time for putting phrases into trees: 281 second(s)
Flushing UNsorted phrases: 100%; Shaking trees performance: 0,023,807P/s
Time for shaking phrases from trees: 427 second(s)
Dump LEAFwise also ]
Leprechaun: Done.

Leprechaun report:
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 731,746
Number Of Trees(GREATER THE BETTER): 9,433,894
Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 69,623
Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 1
Used value for third parameter in KB: 1,262,186
Use next time as third parameter: 1,262,186
Total Attempts to Find/Put WORDs into B-trees order 3: 365,283

[Variant (HashInBITS 26 - 10) with 512KB hash:]

Leprechaun_quadrupleton (Fast Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
Feature2: The bottleneck is seek-time, if the external memory has latency 100+microseconds then look further.
Size of input file with files for Leprechauning: 19
Allocating HASH memory 524,353 bytes ... OK
Allocating/ZEROing 1,292,478,478 bytes swap file ... OK
Size of Input TEXTual file: 206,908,949
|; 0,014,331P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
Bytes per second performance: 158,066B/s
Phrases per second performance: 14,331P/s
Time for putting phrases into trees: 1309 second(s)
Flushing UNsorted phrases: 100%; Shaking trees performance: 0,019,739P/s
Time for shaking phrases from trees: 515 second(s)
Dump LEAFwise also [
Bytes per second performance: 158,429B/s
Phrases per second performance: 14,364P/s
Time for putting phrases into trees: 1306 second(s)
Flushing UNsorted phrases: 100%; Shaking trees performance: 0,041,492P/s
Time for shaking phrases from trees: 245 second(s)
Dump LEAFwise also ]
Dump & Insert LEAFwise also [
Bytes per second performance: 174,459B/s
Phrases per second performance: 15,818P/s
Time for putting phrases into trees: 1186 second(s)
Flushing UNsorted phrases: 100%; Shaking trees performance: 0,041,323P/s
Time for shaking phrases from trees: 246 second(s)
Dump & Insert LEAFwise also ]
Leprechaun: Done.

Leprechaun report:
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 10,100,104
Number Of Trees(GREATER THE BETTER): 65,536
```



```
Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 7,522,788
Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 6
Used value for third parameter in KB: 1,262,186
Use next time as third parameter: 1,007,825
Total Attempts to Find/Put WORDs into B-trees order 3: 84,868,241

[Variant (HashInBITS 26 - 20) with 512 hash:]

Leprechaun_quadrupleton (Fast Greedy Phrase-Ripper), rev. 14+, written by Svalqyatchx.
Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
Feature2: The bottleneck is seek-time, if the external memory has latency 100+microseconds then look further.
Size of input file with files for Leprechauning: 19
Allocating HASH memory 577 bytes ... OK
Allocating/ZEROing 1,292,478,478 bytes swap file ... OK
Size of Input TEXTual file: 206,908,949
|; 0,007,717P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
Bytes per second performance: 85,112B/s
Phrases per second performance: 7,717P/s
Time for putting phrases into trees: 2431 second(s)
Flushing UNsorted phrases: 100%; Shaking trees performance: 0,019,777P/s
Time for shaking phrases from trees: 514 second(s)
Leprechaun: Done.

Leprechaun report:
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 10,165,576
Number Of Trees(GREATER THE BETTER): 64
Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 7,592,585
Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 14
Used value for third parameter in KB: 1,262,186
Use next time as third parameter: 1,008,399
Total Attempts to Find/Put WORDs into B-trees order 3: 271,393,689

r.14++ physical memory test [Variant (HashInBITS 26 - 0) with 512MB hash:]:
D:\_KAZE_new-stuff\Leprechaun_quadrupleton_r14++_64bit_Physical-n-Virtual>OSHO-TEST_INTERNAL.BAT
Leprechaun_quadrupleton (Fast-In-Future Greedy Phrase-Ripper), rev. 14++, written by Svalqyatchx.
Purpose: Rips all distinct 4-grams (4-word phrases) with length 12..51 chars from incoming texts.
Feature1: In this revision 512MB 1-way hash is used which results in 67,108,864 external B-Trees of order 3.
Feature2: If the external memory has latency 99+microseconds then !(look no further), IOPS(seek-time) rules.
Size of input file with files for Leprechauning: 19
Allocating HASH memory 536,870,977 bytes ... OK
Allocating memory 1233MB ... OK
Size of Input TEXTual file: 206,908,949
|; 1,042,234P/s; Phrase count: 18,760,213 of them 10,165,640 distinct; Done: 64/64
Bytes per second performance: 11,494,941B/s
Phrases per second performance: 1,042,234P/s
Time for putting phrases into trees: 18 second(s)
Flushing UNsorted phrases: 100%; Shaking trees performance: 0,597,978P/s
Time for shaking phrases from trees: 17 second(s)
Leprechaun: Done.

D:\_KAZE_new-stuff\Leprechaun_quadrupleton_r14++_64bit_Physical-n-Virtual>type Leprechaun.LOG
Leprechaun report:
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 731,746
Number Of Trees(GREATER THE BETTER): 9,433,894
Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: 69,623
Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): 1
Used value for third parameter in KB: 1,262,186
Use next time as third parameter: 1,262,186
Total Attempts to Find/Put WORDs into B-trees order 3: 365,283

D:\_KAZE_new-stuff\Leprechaun_quadrupleton_r14++_64bit_Physical-n-Virtual>
*/

// To do #1: Put this 31 in MAXwl: 'int MAXwl = 31;'
// To do #2: No need of flushing unsorted words to file: make backup[] array
//           instead of writing. And mostly sort 26 times!
// HEAVY BUG in r.7: unsigned long Hill(unsigned long n)
//                   is NOT identical with
//                   unsigned long GRMBLhill[32]; // 00 not used, only 01..31
//                   BECAUSE DUMBEST DUMB Array GRMBLhill expects 'int' not
//                   'unsigned long' !!!

#include <stdio.h>
#include <ctype.h>
#include <time.h>
#if defined(_WIN32_ENVIRONMENT_)
#include <io.h> // needed for Windows' 'lseeki64' and 'telli64'
//Above line must be commented in order to compile with Intel C compiler: an error "can't find io.h" occurs.
#else
#endif /* defined(_WIN32_ENVIRONMENT_)  */

typedef unsigned char char_t;
typedef char_t *string;

clock_t clocks1, clocks2;
int Bozan;
```

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



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

```
}
void mkqsort_main(unsigned char **a, int n) { mkqsort(a, n, 0); }
// mkqsort.c END ********************************************************************

// Why Sinha uses int instead of long??!!
static int readlines(char *file_name, string **lines)
{
   int nlines = 0;
   size_t size;
   FILE *in_file;
   string basep, cur, next;
   string *ASbackup;

   if (!(in_file = fopen(file_name, "rb"))) {
      printf( "Leprechaun: Can't open file %s \n", file_name );
      exit(-1);
   }
   fseek(in_file, 0, SEEK_END);
   size = ftell(in_file);
   fseek(in_file, 0, SEEK_SET);
   if (!(basep = (string) malloc(size*sizeof(char_t)))) return -1;
   printf( "Allocated memory for words in MB: %lu\n", ((size*sizeof(char_t))>>20)+1 );
   if (fread(basep, 1, size, in_file) < size) {
      printf( "Leprechaun: Can't read file %s \n", file_name );
      exit(-1);
   }
   fclose(in_file);

// GET nlines:
   cur = basep;
   while (cur < basep + size) {
      next = cur;
      while ((next < basep + size) && (*next != '\n')) {next++;}
      *--next = '\0';          // This is ala DOS i.e. Windows
                               // 1310 not 10(\n=10)
      cur = next + 2;
      nlines++;
   }

// printf("%lu\n",(unsigned long)*lines); -> backup = *lines = 0
ASbackup = (string *)malloc( nlines*sizeof(string) ); // sizeof(string) is 4
if( ASbackup == NULL )
{ puts( "Leprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
printf( "Allocated memory for pointers-to-words in MB: %lu\n", ((nlines*sizeof(string))>>20)+1 );
*lines = ASbackup;
//printf("%lu\n",(unsigned long)*lines); -> backup = *lines = ASbackup = 6946888

// Upload nlines times:
   nlines = 0;
   cur = basep;
   while (cur < basep + size) {
      next = cur;
      while ((next < basep + size) && (*next != '\n')) {next++;}
      *--next = '\0';          // This is ala DOS i.e. Windows
                               // 1310 not 10(\n=10)
      ASbackup[nlines] = cur;
      cur = next + 2;
      nlines++;
   }
   return nlines;
}

void x64toaKAZE (      /* stdcall is faster and smaller... Might as well use it for the helper. */
        unsigned long long val,
        char *buf,
        unsigned radix,
        int is_neg
        )
{
        char *p;                /* pointer to traverse string */
        char *firstdig;         /* pointer to first digit */
        char temp;              /* temp char */
        unsigned digval;        /* value of digit */

        p = buf;

        if ( is_neg )
        {
            *p++ = '-';         /* negative, so output '-' and negate */
            val = (unsigned long long)(-(long long)val);
        }

        firstdig = p;           /* save pointer to first digit */

        do {
            digval = (unsigned) (val % radix);
            val /= radix;       /* get next digit */
```



```

            /* convert to ascii and store */
            if (digval > 9)
                *p++ = (char) (digval - 10 + 'a');  /* a letter */
            else
                *p++ = (char) (digval + '0');       /* a digit */
        } while (val > 0);

        /* We now have the digit of the number in the buffer, but in reverse
           order.  Thus we reverse them now. */

        *p-- = '\0';            /* terminate string; p points to last digit */

        do {
            temp = *p;
            *p = *firstdig;
            *firstdig = temp;   /* swap *p and *firstdig */
            --p;
            ++firstdig;         /* advance to next two digits */
        } while (firstdig < p); /* repeat until halfway */
}

/* Actual functions just call conversion helper with neg flag set correctly,
   and return pointer to buffer. */

char * _i64toaKAZE (
        long long val,
        char *buf,
        int radix
        )
{
        x64toaKAZE((unsigned long long)val, buf, radix, (radix == 10 && val < 0));
        return buf;
}

char * _ui64toaKAZE (
        unsigned long long val,
        char *buf,
        int radix
        )
{
        x64toaKAZE(val, buf, radix, 0);
        return buf;
}

char * _ui64toaKAZEzerocomma (
        unsigned long long val,
        char *buf,
        int radix
        )
{
                        char *p;
                        char temp;
                        int txpman;
                        int pxnman;
        x64toaKAZE(val, buf, radix, 0);
                        p = buf;
                        do {
                        } while (*++p != '\0');
                        p--; // p points to last digit
                             // buf points to first digit
                        buf[26] = 0;
                        txpman = 1;
                        pxnman = 0;
                        do
                        { if (buf <= p)
                          { temp = *p;
                            buf[26-txpman] = temp; pxnman++;
                            p--;
                            if (pxnman % 3 == 0)
                            { txpman++;
                              buf[26-txpman] = (char) (',');
                            }
                          }
                          else
                          { buf[26-txpman] = (char) ('0'); pxnman++;
                            if (pxnman % 3 == 0)
                            { txpman++;
                              buf[26-txpman] = (char) (',');
                            }
                          }
                          txpman++;
                        } while (txpman <= 26);
        return buf;
}

char * _ui64toaKAZEcomma (
        unsigned long long val,
```

```
        char *buf,
        int radix
        )
{
                        char *p;
                        char temp;
                        int txpman;
                        int pxnman;
        x64toaKAZE(val, buf, radix, 0);
                        p = buf;
                        do {
                        } while (*++p != '\0');
                        p--; // p points to last digit
                             // buf points to first digit
                        buf[26] = 0;
                        txpman = 1;
                        pxnman = 0;
                        while (buf <= p)
                        { temp = *p;
                          buf[26-txpman] = temp; pxnman++;
                          p--;
                          if (pxnman % 3 == 0 && buf <= p)
                          { txpman++;
                            buf[26-txpman] = (char) (',');
                          }
                          txpman++;
                        }
        return buf+26-(txpman-1);
}

unsigned char KuxHash(char *str)
{ unsigned char h = 0;
  int max31 = 0;
  //while (*str)
  while (str[max31])
  { h = h ^ str[max31++];
    //h = h ^ *str++; // I am not sure 'str' is returned changed after return?!
  }
  return h; // 00..255 i.e. 2^8=256
}

int KuxHash2(char *str)
{ int h = 0;
  unsigned long h2 = 0; // must be long: 31*'z'=31*122
  int max31 = 0;
  while (str[max31])
  { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
    //h2 = h2 + str[max31++];       // [113s]
    h2 = h2 + max31 * str[max31++];
  }
  h=h<<4; // 00..15 i.e. 2^4=16
  //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
  h = h|( h2%((1<<4)-1) );
  return h; // 00..4095 i.e. 2^12=4096
}

//OSHO test - Attempts to Find/Put a WORD into linked list count: 32,011,937
int KuxHash3(char *str)
{ int h = 0;
  unsigned long h2 = 0; // must be long: 31*'z'=31*122
  int max31 = 0;
  while (str[max31])
  { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
    //h2 = h2 + str[max31++];       // [113s]
    h2 = h2 + str[max31++] * (max31+1);
  }
// Result is: 7bits in 'h' and 32bits in 'h2'.

                        //printf("%s:\n ",str);
                        //printf("%d ",h);
  h=h<<6; // 00..15 i.e. 00-05+7bits=13bits
                        //printf("%d ",h);
                        //printf("%d ",h2);
  //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
  h = h|( h2%((1<<6)-1) ); // 64-1=63=9*7; 61 is prime
                        //printf("%d \n",h);
  return h; // 00..8191 i.e. 2^13=8192
}

//OSHO test - Attempts to Find/Put a WORD into a linked list count: 31,927,285
int KuxHash3plus(char *str)
{ int h = 0;
  unsigned long h2 = 0; // must be long: 31*'z'=31*122
  int max31 = 0;
  while (str[max31])
  { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
    //h2 = h2 + str[max31++];       // [113s]
    h2 = h2 + str[max31++] * (max31+1);
```



```
  }
// Result is: 7bits in 'h' and 32bits in 'h2'.

                        //printf("%s:\n ",str);
                        //printf("%d ",h);
// a in ASCII is 097 = 0110 0001
// z in ASCII is 122 = 0111 1010
// Above two lines show that bits 8-7-6 are always 0-1-1 so need for low 5 bits.
  //h=h<<8; // 00..15 i.e. 5bits + 00-07bits=13bits
                        //printf("%d ",h);
                        //printf("%d ",h2);
  //h = h|( str[0] ^ str[max31-1] ); // [111s] a..z: each XOR each gives 00..31
  h = (( h<<8 )|( h2%(251) ))&8191; // 251 prime
                        //printf("%d \n",h);
  return h; // 00..8191 i.e. 2^13=8192
}

/*
PUBLIC        _KuxHash3plus
; Function compile flags: /Ogty
_TEXT         SEGMENT
_str$ = 8                                               ; size = 4
_KuxHash3plus PROC NEAR
; Line 511
mov  ecx, DWORD PTR _str$[esp-4]
mov  dl, BYTE PTR [ecx]
push esi
xor  esi, esi
xor  eax, eax
test dl, dl
je   SHORT $L1561
push ebx
push edi
mov  edi, 1
sub  edi, ecx
npad 8
$L1560:
; Line 512
movsx         edx, BYTE PTR [ecx]
; Line 514
lea  ebx, DWORD PTR [edi+ecx]
imul ebx, edx
xor  esi, edx
mov  dl, BYTE PTR [ecx+1]
add  eax, ebx
inc  ecx
test dl, dl
jne  SHORT $L1560
pop  edi
pop  ebx
$L1561:
; Line 527
xor  edx, edx
mov  ecx, 251                           ; 000000fbH
div  ecx
shl  esi, 8
mov  eax, edx
; Line 529
or   eax, esi
and  eax, 8191                          ; 00001fffH
pop  esi
; Line 530
ret  0
_KuxHash3plus ENDP
_TEXT         ENDS
*/

//OSHO test - Attempts to Find/Put a WORD into a linked list count: 32,021,975
int KuxHash4(char *str)
{
  int h2 = 0;
    for (; *str != 0; str++) {
      //h2 = (127*h2 + *str) % (8192-1); // 2^13-1 = 8191 is Mersenne prime
      h2 = ((h2<<7) + *str) % (8192-1); // 2^13-1 = 8191 is Mersenne prime
    }

  return h2; // 00..8191 i.e. 2^13=8192
}

/*
int hash(char *v, int M)
  { int h = 0, a = 127;
    for (; *v != 0; v++)
      h = (a*h + *v) % M;
    return h;
  }

int hashU(char *v, int M)
```

```
  { int h, a = 31415, b = 27183;
    for (h = 0; *v != 0; v++, a = a*b % (M-1))
        h = (a*h + *v) % M;
    return (h < 0) ? (h + M) : h;
  }
*/

// Kaze: My appreciation of FNV is far beyound C code optimization, it is alchemical, and why not, magical.

/*
FNV hash history
    The basis of the FNV hash algorithm was taken from an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee
by Glenn Fowler and Phong Vo back in 1991. In a subsequent ballot round: Landon Curt Noll improved on their algorithm.
Some people tried this hash and found that it worked rather well. In an EMail message to Landon, they named it
the ``Fowler/Noll/Vo'' or FNV hash.
    FNV hashes are designed to be fast while maintaining a low collision rate. The FNV speed allows one to quickly hash
lots of data while maintaining a reasonable collision rate. The high dispersion of the FNV hashes makes them well suited
for hashing nearly identical strings such as URLs, hostnames, filenames, text, IP addresses, etc.
*/

/* NOTE: u_int64_t is a 64 bit unsigned type */
/* NOTE: u_int32_t is a 32 bit unsigned type */
/* NOTE: u_int16_t is a 16 bit unsigned type */
/* NOTE: u_int8_t is a 8 bit unsigned type */

//typedef unsigned char u_int8_t; //FNV only
//typedef unsigned long u_int32_t; //FNV only
//typedef unsigned long long u_int64_t; //FNV only

// 32 bit FNV_prime = 2^24 + 2^8 + 0x93 = 16777619
// 64 bit FNV_prime = 2^40 + 2^8 + 0xb3 = 1099511628211

// 32 bit offset_basis = 2166136261
// 64 bit offset_basis = 14695981039346656037

#define FNV1_64_INIT ((u_int64_t)14695981039346656037)
#define FNV1_64_PRIME ((u_int64_t)1099511628211)
#define FNV1_32_INIT ((u_int32_t)2166136261)
#define FNV1_32_PRIME ((u_int32_t)602173697)
// FNV1A_Hash_4_OCTETS gives dispersion as follows:
//3549448: 1607
//3549669: 171072511
//3550710: 272961023
//3550733: 172783361
//3550734: 431562497
//3550929: 204312319
//3550984: 562853633
//3550991: 551362303
//3551359: 332820737
//3551484: 354126079
//3551514: 407138561
//3551523: 442058753
//3551701: 449230849
//3551736: 107712257
//3551961: 428904191
//3552039: 602173697
//3552103: 588411137

#define FNV_64A_OP(hash, octet) \
    (((u_int64_t)(hash) ^ (u_int8_t)(octet)) * FNV1_64_PRIME)

#define FNV_64A_OP64(hash, octet) \
    (((u_int64_t)(hash) ^ (u_int64_t)(octet)) * FNV1_64_PRIME)

#define FNV_32A_OP_GENERIC(hash, octet) \
    (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * 16777619)

#define FNV_32A_OP(hash, octet) \
    (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * FNV1_32_PRIME)

#define FNV_32A_OP_MULless_core(hash, octet) \
    ( (u_int32_t)(hash) ^ (u_int8_t)(octet) )

#define FNV_32A_OP_MULless(hash, octet) \
    ( (FNV_32A_OP_MULless_core(hash, octet)<<5) - FNV_32A_OP_MULless_core(hash, octet) )

#define FNV_32A_OP32(hash, octet) \
    (((u_int32_t)(hash) ^ (u_int32_t)(octet)) * FNV1_32_PRIME)

#define FNV_32A_OP64(hash, octet) \
    (((u_int64_t)(hash) ^ (u_int64_t)(octet)) * FNV1_32_PRIME)

#define FNV_32A_OP32_MULless_core(hash, octet) \
    ( (u_int32_t)(hash) ^ (u_int32_t)(octet) )

#define FNV_32A_OP32_MULless(hash, octet) \
    ( (FNV_32A_OP32_MULless_core(hash, octet)<<5) - FNV_32A_OP32_MULless_core(hash, octet) )
```



```


// Invoking: FNV1A_Hash_4_OCTETS_31(wrd, wrdlen>>2) // = 0,1,2,3,4,5,6,7 [1..31]
int FNV1A_Hash_4_OCTETS_31(char *str, int wrdlen_QUADRUPLETS)
{
u_int32_t hash;
char *p;

hash = FNV1_32_INIT;
p=str;

// The goal of stage #1: to reduce number of 'imul's in fact to reduce loops.

// Stage #1:
for (; wrdlen_QUADRUPLETS != 0; --wrdlen_QUADRUPLETS) {
    hash = FNV_32A_OP32_MULless(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
    p=p+4; // add eax, 4
}

// Stage #2:
for (; *p; ++p) {
    hash = FNV_32A_OP_MULless(hash, *p); // mov dl, BYTE PTR [ecx]
}

  //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
  return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
}


// Invoking: FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2) // = 0,1,2,3,4,5,6,7 [1..31]
int FNV1A_Hash_4_OCTETS(char *str, int wrdlen_QUADRUPLETS)
{
u_int32_t hash;
char *p;

hash = FNV1_32_INIT;
p=str;

// The goal of stage #1: to reduce number of 'imul's.

// Stage #1:
for (; wrdlen_QUADRUPLETS != 0; --wrdlen_QUADRUPLETS) {
    hash = FNV_32A_OP32(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
    p=p+4; // add eax, 4
}

// Stage #2:
for (; *p; ++p) {
    hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [ecx]
}

  //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
  return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
}

/*
Results for 'FNV1A_Hash_8_OCTETS':
Bytes per second performance: 23,110,160B/s
Words per second performance: 1,959,516W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Files: 8
Number Of Lines: 35271297
Allocated memory in MB: 1950
Number Of Trees(GREATER THE BETTER): 3419429
Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 51%
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18783551
Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '1,119'
Total Attempts to Find/Put WORDs into Binary-Search-Trees: 268,085,505
Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 7,690,615
Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 2,622 must have PEAK = 12 = rounding down of integer (1+lb(2,622))
Binary-Search-Tree(1st out of 1) with MaxNODEs = 2,622 has PEAK = 592 and LEAFs = 689
Binary-Search-Tree(1st out of 1) with MaxPEAK = '1,119' has NODEs = 1,537 and LEAFs = 287
Binary-Search-Tree(1st out of 1) with MaxLEAFs = 731 has NODEs = 2,517 and PEAK = 448
*/
// Invoking: FNV1A_Hash_8_OCTETS(wrd, wrdlen>>3) // = 0,1,2,3 [1..31]
int FNV1A_Hash_8_OCTETS(char *str, int wrdlen_OCTETS)
{
u_int32_t hash;
char *p;

hash = FNV1_32_INIT;
p=str;

// The goal of stage #1: to reduce number of 'imul's.

// Stage #1:
for (; wrdlen_OCTETS != 0; --wrdlen_OCTETS) {
```

```
    hash = FNV_32A_OP64(hash, (unsigned long long)*(long *)p); // mov edi, DWORD PTR [eax]
    p=p+8; // add eax, 4
}

// Stage #2:
for (; *p; ++p) {
    hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [ecx]
}

  //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
  return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
}


// Invoking: FNV1A_Hash_Granularity(wrd, wrdlen>>0|2|3, 0|2|3)
int FNV1A_Hash_Granularity(char *str, int wrdlen_granulated, int Granularity) // wrdlen>>0=wrdlen
{
u_int32_t hash;
u_int64_t hash64;
char *p;

hash = FNV1_32_INIT;
p=str;

// The goal of stage #1: to reduce number of 'imul's and mainly: the number of loops.

// Stage #1:
        if (Granularity == 2) {
for (; wrdlen_granulated != 0; --wrdlen_granulated) {
    hash = FNV_32A_OP32(hash, (u_int32_t)*(u_int32_t *)p);
    p=p+4; // (1<<Granularity): 1<<0=1, 1<<2=4, 1<<3=8
}
        }
        if (Granularity == 3) {
hash64 = FNV1_64_INIT;
for (; wrdlen_granulated != 0; --wrdlen_granulated) {
    hash64 = FNV_64A_OP64(hash64, (u_int64_t)*(u_int64_t *)p);
    p=p+8; // (1<<Granularity): 1<<0=1, 1<<2=4, 1<<3=8
}
for (; *p; ++p) {
    hash64 = FNV_64A_OP(hash64, (u_int8_t)*(u_int8_t *)p);
}

  //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
  return ((hash64>>51) ^ hash64) & 8191; // 00..8191 i.e. 2^13=8192
  // probably better shifting is not by 16 bits but ...
  //hash64>>16: 3,544,160 just bad
  //hash64>>33: 3,547,854
  //hash64>>34: 3,547,266
  //hash64>>35: 3,547,453
  //hash64>>36: 3,547,242
  //hash64>>40: 3,548,263
  //hash64>>44: 3,548,242
  //hash64>>45: 3,549,056
  //hash64>>46: 3,549,207
  //hash64>>47: 3,549,094
  //hash64>>50: 3,549,392
  //hash64>>51: 3,549,395 i.e. maximum shift: the 13 most significant bits i.e. (64-13); closest to 3,549,448

// Above results are obtained for following set:
//if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
//              Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>2, 2)<<2; //13+++++
//else            // 2x8+4=20 i.e. first contains 6 clashes
//              Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>3, 3)<<2; //13+++++
        }

        //if (Granularity != 3) {
// Stage #2:
for (; *p; ++p) {
    hash = FNV_32A_OP(hash, (u_int8_t)*(u_int8_t *)p);
}

  //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
  return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
        //}
}


// char *string;                /* the string to 64 bit FNV-1a hash */
// u_int64_t hash;              /* will hold the final value of the hash */
// char *p;
//
// hash = FNV1_64_INIT;
// for (p=string; *p; ++p) {
//     hash = FNV_64A_OP(hash, *p);
// }
```



```
// If you need an x-bit hash where x is not a power of 2,
// 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.
// By xor-folding we mean shift the excess high order bits down and xor them with the lower x-bits.
// For tiny x < 16 bit values, we recommend using a 32 bit FNV-1 hash as follows:

//    /* NOTE: for 0 < x < 16 ONLY!!! */
//    #define TINY_MASK(x) (((u_int32_t)1<<(x))-1)
//    #define FNV1_32_INIT ((u_int32_t)2166136261)
//    u_int32_t hash;
//    void *data;
//    size_t data_len;
//
//    hash = fnv_32_buf(data, data_len, FNV1_32_INIT);
//    hash = (((hash>>x) ^ hash) & TINY_MASK(x));


int FNV1A_Hash_SHIFTless_XORless(char *str)
{
u_int32_t hash;                 /* will hold the final value of the hash */
char *p;

hash = FNV1_32_INIT;
for (p=str; *p; ++p) {
    hash = FNV_32A_OP(hash, *p);
}
//hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13

  return hash & 8191; // 00..8191 i.e. 2^13=8192
}

/*
_FNV1A_Hash_SHIFTless_XORless PROC NEAR
; Line 721
mov  edx, DWORD PTR _str$[esp-4]
mov  cl, BYTE PTR [edx]
test cl, cl
mov  eax, -2128831035                       ; 811c9dc5H
je   SHORT $L1582
npad 1
$L1580:
; Line 722
movzx        ecx, cl
xor  ecx, eax
imul ecx, 16777619                                  ; 01000193H
inc  edx
mov  eax, ecx
mov  cl, BYTE PTR [edx]
test cl, cl
jne  SHORT $L1580
$L1582:
; Line 726
and  eax, 8191                              ; 00001fffH
; Line 727
ret  0
_FNV1A_Hash_SHIFTless_XORless ENDP
*/


int FNV1A_Hash(char *str)
{
u_int32_t hash;                 /* will hold the final value of the hash */
char *p;

hash = FNV1_32_INIT;
for (p=str; *p; ++p) {
    hash = FNV_32A_OP(hash, *p);
}
//hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13

  return ((hash>>13) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
}

/*
_FNV1A_Hash PROC NEAR
; Line 722
mov  edx, DWORD PTR _str$[esp-4]
mov  al, BYTE PTR [edx]
test al, al
mov  ecx, -2128831035                       ; 811c9dc5H
je   SHORT $L1582
npad 1
$L1580:
; Line 723
movzx        eax, al
xor  eax, ecx
imul eax, 16777619                                  ; 01000193H
inc  edx
mov  ecx, eax
```

```
mov al, BYTE PTR [edx]
test al, al
jne SHORT $L1580
$L1582:
; Line 727
mov eax, ecx
shr eax, 13                                     ; 0000000dH
xor eax, ecx
and eax, 8191                           ; 00001fffH
; Line 728
ret 0
_FNV1A_Hash ENDP
*/

/*
Wayne Diamond implemented 32-bit FNV algorithm in PowerBASIC inline x86 assembly:


    FUNCTION FNV32(BYVAL dwOffset AS DWORD, BYVAL dwLen AS DWORD, BYVAL offset_basis AS DWORD) AS DWORD
    #REGISTER NONE
    ! mov esi, dwOffset      ;esi = ptr to buffer
    ! mov ecx, dwLen         ;ecx = length of buffer (counter)
    ! mov eax, offset_basis  ;set to 2166136261 for FNV-1
    ! mov edi, &h01000193    ;FNV_32_PRIME = 16777619
    ! xor ebx, ebx           ;ebx = 0
    nextbyte:
    ! mul edi                ;eax = eax * FNV_32_PRIME
    ! mov bl, [esi]          ;bl = byte from esi
    ! xor eax, ebx           ;al = al xor bl
    ! inc esi                ;esi = esi + 1 (buffer pos)
    ! dec ecx                ;ecx = ecx - 1 (counter)
    ! jnz nextbyte           ;if ecx is 0, jmp to NextByte
    ! mov FUNCTION, eax      ;else, function = eax
    END FUNCTION

Wayne said:

    ''Just thought I should let you know that I've ported the 32-bit FNV algorithm over to inline assembly.
It's actually in PowerBASIC (www.powerbasic.com) format - a compiler I use, but the main function is all assembly.
It could be optimized further in terms of saving a couple of clock cycles,
but it's fairly optimized al ready - only 6 instructions in the main loop, plus 5 setup instructions,
and compiles to just 33 bytes.''

M.S.Schulte sent us these 32-bit FNV-1 and FNV-1a x86 assembler implementations (written in flat assembler),
half of which were optimized for speed, the other half were optimized for size:

    small_fnv32:    ;FNV1 32bit (size: 31 bytes)
    ;                Intel Core 2 Duo E6600: 354.20 mb/s
       push    esi
       push    edi
       mov     esi, [esp + 0ch] ;buffer
       mov     ecx, [esp + 10h] ;length
       mov     eax, [esp + 14h] ;basis
       mov     edi, 01000193h   ;fnv_32_prime
     next:
       mul     edi
       xor     al, [esi]
       inc     esi
       loop    snext
       pop     edi
       pop     esi
       retn    0ch

     small_fnv32a:   ;FNV1a 32bit (size: 31 bytes)
    ;                 Intel Core 2 Duo E6600: 327.68 mb/s
       push    esi
       push    edi
       mov     esi, [esp + 0ch] ;buffer
       mov     ecx, [esp + 10h] ;length
       mov     eax, [esp + 14h] ;basis
       mov     edi, 01000193h   ;fnv_32_prime
     nexta:
       xor     al, [esi]
       mul     edi
       inc     esi
       loop    nexta
       pop     edi
       pop     esi
       retn    0ch

    fast_fnv32:    ;FNV1 32bit (size: 36 bytes)
    ;               Intel Core 2 Duo E6600: 565.12 mb/s
       push    ebx
       push    esi
       push    edi
       mov     esi, [esp + 10h] ;buffer
       mov     ecx, [esp + 14h] ;length
       mov     eax, [esp + 18h] ;basis
```



```
       mov     edi, 01000193h   ;fnv_32_prime
       xor     ebx, ebx
     next:
       mul     edi
       mov     bl, [esi]
       xor     eax, ebx
       inc     esi
       dec     ecx
       jnz     next
       pop     edi
       pop     esi
       pop     ebx
       retn    0ch

     fast_fnv32a:   ;FNV1a 32bit (size: 36 bytes)
    ;                Intel Core 2 Duo E6600: 574.95 mb/s
       push    ebx
       push    esi
       push    edi
       mov     esi, [esp + 10h] ;buffer
       mov     ecx, [esp + 14h] ;length
       mov     eax, [esp + 18h] ;basis
       mov     edi, 01000193h   ;fnv_32_prime
       xor     ebx, ebx
     nexta:
       mov     bl, [esi]
       xor     eax, ebx
       mul     edi
       inc     esi
       dec     ecx
       jnz     nexta
       pop     edi
       pop     esi
       pop     ebx
       retn    0ch
*/

//Number Of Trees(GREATER THE BETTER): 3525737
//Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
//Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18677243
int Hash17_unrolled(const char *key, int wrdlen)
{
int hash = 1;
        int i;
for(i = 0; i < (wrdlen & -2); i += 2) {
     hash = (17) * hash + (key[i] - ' ');
     hash = (17) * hash + (key[i+1] - ' ');
}
if(wrdlen & 1)
     hash = (17) * hash + (key[wrdlen-1] - ' ');
return ( hash ^ (hash >> 16) ) & 8191;
}

//hash = 1:
//Number Of Trees(GREATER THE BETTER): 3556516
//Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
//Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18646464
//hash = 13:
//Number Of Trees(GREATER THE BETTER): 3556755
//Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
//Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18646225
//hash = 11:
//Number Of Trees(GREATER THE BETTER): 3557011
//Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
//Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18645969
//hash = 7:
//Number Of Trees(GREATER THE BETTER): 3557181
//Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
//Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18645799
int Alfalfa(const char *key, int wrdlen)
{
int hash = 7;
        int i;
for(i = 0; i < (wrdlen & -2); i += 2) {
     hash = (17+9) * ((17+9) * hash + (key[i])) + (key[i+1]);
}
if(wrdlen & 1)
     hash = (17+9) * hash + (key[wrdlen-1]);
return ( hash ^ (hash >> 16) ) & 8191;
}

/*
[FNV1A 'shift-less-&-xor-less' hash used in Leprechaun r.13+++:]

int FNV1A_Hash_SHIFTless_XORless(char *str)
{
u_int32_t hash;
char *p;
```

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



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

```
*/
UINT HashAlfalfa_HALF(const char *key, unsigned int wrdlen)
{
UINT hash = 12;
UINT hashBUFFER;
unsigned int i,j;
for(i = 0; i < (wrdlen & -4); i += 4) {
    //hash = (( ((hash<<5)-hash) + key[i] )<<5) - ( ((hash<<5)-hash) + key[i] ) + (key[i+1]);
    hashBUFFER = ((hash<<5)-hash) + key[i];
    hash = (( hashBUFFER )<<5) - ( hashBUFFER ) + (key[i+1]);
    //hash = (( ((hash<<5)-hash) + key[i+2] )<<5) - ( ((hash<<5)-hash) + key[i+2] ) + (key[i+3]);
    hashBUFFER = ((hash<<5)-hash) + key[i+2];
    hash = (( hashBUFFER )<<5) - ( hashBUFFER ) + (key[i+3]);
}
for(j = 0; j < (wrdlen & 3); j += 1) {
    hash = ((hash<<5)-hash) + key[i+j];
}
return ((hash>>16) ^ hash) & 8191;
}

/*
Results for 'HashFNV1A_unrolled_Final':
Bytes per second performance: 19,808,709B/s
Words per second performance: 1,679,585W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Files: 8
Number Of Lines: 35271297
Allocated memory in MB: 1950
Number Of Trees(GREATER THE BETTER): 3445337
Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 52%
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18757643
Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '43'
Total Attempts to Find/Put WORDs into Binary-Search-Trees: 118,349,998
Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 7,997,033
Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 89 must have PEAK = 7 = rounding down of integer (1+lb(89))
Binary-Search-Tree(1st out of 1) with MaxNODEs = 89 has PEAK = 28 and LEAFs = 28
Binary-Search-Tree(1st out of 1) with MaxPEAK = '43' has NODEs = 65 and LEAFs = 11
Binary-Search-Tree(1st out of 2) with MaxLEAFs = 28 has NODEs = 78 and PEAK = 24
*/
UINT HashFNV1A_unrolled_Final(char *str, unsigned int wrdlen)
{
//const UINT PRIME = 31;
unsigned int hash = 2166136261;
char * p = str;

/*
// Reduce the number of multiplications by unrolling the loop
for (SIZE_T ndwords = wrdlen / sizeof(DWORD); ndwords; --ndwords) {
    //hash = (hash ^ *(DWORD*)p) * PRIME;
    hash = ((hash ^ *(DWORD*)p)<<5) - (hash ^ *(DWORD*)p);

    p += sizeof(DWORD);
}
*/
for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
    hash = ((hash ^ *(unsigned int*)p)<<5) - (hash ^ *(unsigned int*)p);
}

// Process the remaining bytes
/*
for (SIZE_T i = 0; i < (wrdlen & (sizeof(DWORD) - 1)); i++) {
    //hash = (hash ^ *p++) * PRIME;
    hash = ((hash ^ *p)<<5) - (hash ^ *p);
    p++;
}
*/
if (wrdlen & -2) {
    hash = ((hash ^ (*(unsigned int*)p&0xFFFF))<<5) - (hash ^ (*(unsigned int*)p&0xFFFF));
    p++;p++;
}
if (wrdlen & 1)
    hash = ((hash ^ *p)<<5) - (hash ^ *p);

return ((hash>>16) ^ hash) & 8191;
}

/*
Results for 'Sixtinsensitive':
Bytes per second performance: 19,808,709B/s
Words per second performance: 1,679,585W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Files: 8
Number Of Lines: 35271297
Allocated memory in MB: 1950
```



```
Number Of Trees(GREATER THE BETTER): 3531949
Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18671031
Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '38'
Total Attempts to Find/Put WORDs into Binary-Search-Trees: 118,959,016
Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,047,983
Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 98 must have PEAK = 7 = rounding down of integer (1+lb(98))
Binary-Search-Tree(1st out of 1) with MaxNODEs = 98 has PEAK = 36 and LEAFs = 30
Binary-Search-Tree(1st out of 1) with MaxPEAK = '38' has NODEs = 54 and LEAFs = 11
Binary-Search-Tree(1st out of 2) with MaxLEAFs = 30 has NODEs = 98 and PEAK = 36
*/
// Tuned for lowercase-and-uppercase letters i.e. 26 ASCII symbols 65-90 and 97-122 decimal.
UINT Sixtinsensitive(const char *str, unsigned int wrdlen)
{
UINT hash = 2166136261;
UINT hashBUFFER_EAX, hashBUFFER_BH, hashBUFFER_BL;
const char * p = str;

// 0x41 = 065 'A' 010 [0 0001]
// 0x5A = 090 'Z' 010 [1 1010]
// 0x61 = 097 'a' 011 [0 0001]
// 0x7A = 122 'z' 011 [1 1010]

// Reduce the number of multiplications by unrolling the loop
for(; wrdlen >= 6; wrdlen -= 6, p += 6) {
    //hashBUFFER_AX = (*(DWORD*)(p+0)&0xFFFF);
    hashBUFFER_EAX = (*(DWORD*)(p+0)&0x1F1F1F1F);
    hashBUFFER_BL = (*(p+4)&0x1F);
    hashBUFFER_BH = (*(p+5)&0x1F);
    //6bytes-in-4bytes or 48bits-to-30bits
    // Two times next:
    //3bytes-in-2bytes or 24bits-to-15bits
    //EAX                       BL          BH
    //[5bit][3bit][5bit][3bit][5bit][3bit][5bit][3bit]
    //       5th[0..15]  13th[0..15]
    //       BL lower 3  BL higher 2bits
    // OR or XOR no difference
    hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BL&0x07)<<5); // BL lower 3bits of 5bits
    hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BL&0x18)<<(2+8)); // BL higher 2bits of 5bits
    hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BH&0x07)<<(5+16)); // BH lower 3bits of 5bits
    hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BH&0x18)<<((2+8)+16)); // BH higher 2bits of 5bits
    //hash = (hash ^ hashBUFFER_EAX)*1607; //What a mess: <<7 becomes imul but <<5 not!?
    hash = ((hash ^ hashBUFFER_EAX)<<5) - (hash ^ hashBUFFER_EAX);
    //1607:[2118599]
    // 127:[2121081]
    //  31:[2139242]
    //  17:[2150803]
    //   7:[2166336]
    //   5:[2183044]
    //8191:[2200477]
    //   3:[2205095]
    // 257:[2206188]
}
// Post-Variant #1:
for(; wrdlen; wrdlen--, p++) {
    hash = ((hash ^ (*p&0x1F))<<5) - (hash ^ (*p&0x1F));
}
/*
// Post-Variant #2:
for(; wrdlen >= 2; wrdlen -= 2, p += 2) {
    hash = ((hash ^ (*(DWORD*)p&0xFFFF))<<5) - (hash ^ (*(DWORD*)p&0xFFFF));
}
if (wrdlen & 1)
    hash = ((hash ^ *p)<<5) - (hash ^ *p);
*/
/*
// Post-Variant #3:
for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
    hash = ((hash ^ *(DWORD*)p)<<5) - (hash ^ *(DWORD*)p);
}
if (wrdlen & -2) {
    hash = ((hash ^ (*(DWORD*)p&0xFFFF))<<5) - (hash ^ (*(DWORD*)p&0xFFFF));
    p++;p++;
}
if (wrdlen & 1)
    hash = ((hash ^ *p)<<5) - (hash ^ *p);
*/
return ((hash>>16) ^ hash) & 8191;
}

/*
#define FNV1_32_INIT ((UINT)2166136261)
#define FNV1_32_PRIME ((UINT)1709)

#define FNV_32A_OP(hash, octet) \
    (((UINT)(hash) ^ (unsigned char)(octet)) * FNV1_32_PRIME)

#define FNV_32A_OP32(hash, octet) \
```

```
    (((UINT)(hash) ^ (UINT)(octet)) * FNV1_32_PRIME)

UINT FNV1A_Hash_WHIZ(const char *str, SIZE_T wrdlen)
{

UINT hash32;
const char *p;

hash32 = FNV1_32_INIT;
p=str;

for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
hash32 = FNV_32A_OP32(hash32, (UINT)*(UINT *)p);
}
if (wrdlen & -2) {
    hash32 = FNV_32A_OP32(hash32, *(UINT*)p&0xFFFF);
    p++;p++;
}
if (wrdlen & 1)
    hash32 = FNV_32A_OP(hash32, *p);

return hash32 ^ (hash32 >> 16);
}
*/

/*
Results for 'FNV1A_Hash_Jester':
Bytes per second performance: 19,808,709B/s
Words per second performance: 1,679,585W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Files: 8
Number Of Lines: 35271297
Allocated memory in MB: 1950
Number Of Trees(GREATER THE BETTER): 3537352
Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18665628
Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '37'
Total Attempts to Find/Put WORDs into Binary-Search-Trees: 117,243,563
Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,063,361
Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 87 must have PEAK = 7 = rounding down of integer (1+lb(87))
Binary-Search-Tree(1st out of 2) with MaxNODEs = 87 has PEAK = 27 and LEAFs = 23
Binary-Search-Tree(1st out of 1) with MaxPEAK = '37' has NODEs = 66 and LEAFs = 18
Binary-Search-Tree(1st out of 3) with MaxLEAFs = 27 has NODEs = 84 and PEAK = 27
*/
UINT FNV1A_Hash_Jester(const char *str, unsigned int wrdlen)
{
const UINT PRIME = 709607;
UINT hash32 = 2166136261;
const char *p = str;

// Idea comes from Igor Pavlov's 7zCRC, thanks.
/*
for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
    hash32 = (hash32 ^ *p) * PRIME;
}
*/
for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
    hash32 = (hash32 ^ *(DWORD *)p) * PRIME;
    hash32 = (hash32 ^ *(DWORD *)(p+4)) * PRIME;
}
// Cases: 0,1,2,3,4,5,6,7
if (wrdlen & sizeof(DWORD)) {
    hash32 = (hash32 ^ *(DWORD*)p) * PRIME;
    p += sizeof(DWORD);
}
if (wrdlen & sizeof(WORD)) {
    hash32 = (hash32 ^ *(WORD*)p) * PRIME;
    p += sizeof(WORD);
}
if (wrdlen & 1)
    hash32 = (hash32 ^ *p) * PRIME;

return (hash32 ^ (hash32 >> 16)) & 8191;
}

/*
Results for 'FNV1A_Hash_Jesteress':
Bytes per second performance: 19,808,709B/s
Words per second performance: 1,679,585W/s
Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
Size of all TEXTual Files: 415,982,896
Word count: 35,271,297 of them 22,202,980 distinct
Number Of Files: 8
Number Of Lines: 35271297
Allocated memory in MB: 1950
Number Of Trees(GREATER THE BETTER): 3537293
```



```
Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
Number Of Hash Collisions(Distinct WORDs - Number Of Trees): 18665687
Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '40'
Total Attempts to Find/Put WORDs into Binary-Search-Trees: 117,526,680
Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,051,512
Perfectly-Balanced-Binary-Search-Tree for MaxNODEs = 89 must have PEAK = 7 = rounding down of integer (1+lb(89))
Binary-Search-Tree(1st out of 1) with MaxNODEs = 89 has PEAK = 25 and LEAFs = 23
Binary-Search-Tree(1st out of 1) with MaxPEAK = '40' has NODEs = 49 and LEAFs = 8
Binary-Search-Tree(1st out of 1) with MaxLEAFs = 28 has NODEs = 72 and PEAK = 21
*/
#define ROL(x, n) (((x) << (n)) | ((x) >> (32-(n))))
UINT FNV1A_Hash_Jesteress(const char *str, unsigned int wrdlen)
{
const UINT PRIME = 709607;
UINT hash32 = 2166136261;
const char *p = str;

// Idea comes from Igor Pavlov's 7zCRC, thanks.
/*
for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
    hash32 = (hash32 ^ *p) * PRIME;
}
*/
for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
    hash32 = (hash32 ^ (ROL(*(DWORD *)p,5)^*(DWORD *)(p+4))) * PRIME;
}
// Cases: 0,1,2,3,4,5,6,7
if (wrdlen & sizeof(DWORD)) {
    hash32 = (hash32 ^ *(DWORD*)p) * PRIME;
    p += sizeof(DWORD);
}
if (wrdlen & sizeof(WORD)) {
    hash32 = (hash32 ^ *(WORD*)p) * PRIME;
    p += sizeof(WORD);
}
if (wrdlen & 1)
    hash32 = (hash32 ^ *p) * PRIME;

return (hash32 ^ (hash32 >> 16)) & 8191;
}

UINT FNV1A_Hash_Jesteress_27bit(const char *str, unsigned int wrdlen)
{
const UINT PRIME = 709607;
UINT hash32 = 2166136261;
const char *p = str;

// Idea comes from Igor Pavlov's 7zCRC, thanks.
/*
for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
    hash32 = (hash32 ^ *p) * PRIME;
}
*/
for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
    hash32 = (hash32 ^ (ROL(*(DWORD *)p,5)^*(DWORD *)(p+4))) * PRIME;
}
// Cases: 0,1,2,3,4,5,6,7
if (wrdlen & sizeof(DWORD)) {
    hash32 = (hash32 ^ *(DWORD*)p) * PRIME;
    p += sizeof(DWORD);
}
if (wrdlen & sizeof(WORD)) {
    hash32 = (hash32 ^ *(WORD*)p) * PRIME;
    p += sizeof(WORD);
}
if (wrdlen & 1)
    hash32 = (hash32 ^ *p) * PRIME;

return (hash32 ^ (hash32 >> 16)) & ((1<<HashInBITS)-1);
}

/*
UINT NextPowerOfTwo(UINT x) {
// Henry Warren, "Hacker's Delight", ch. 3.2
x--;
x |= (x >> 1);
x |= (x >> 2);
x |= (x >> 4);
x |= (x >> 8);
x |= (x >> 16);
return x + 1;
}

UINT NextLog2(UINT x) {
// Henry Warren, "Hacker's Delight", ch. 5.3
if(x <= 1) return x;
x--;
UINT n = 0;
```

```
UINT y;
y = x >>16; if(y) {n += 16; x = y;}
y = x >> 8; if(y) {n +=  8; x = y;}
y = x >> 4; if(y) {n +=  4; x = y;}
y = x >> 2; if(y) {n +=  2; x = y;}
y = x >> 1; if(y) return n + 2;
return n + x;
}
*/

// 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.
/**
 * Returns the floor form of binary logarithm for a 32 bit integer.
 * -1 is returned if n is 0.
 */
int floorLog2(unsigned int n) {
  int pos = 0;
  if (n >= 1<<16) { n >>= 16; pos += 16; }
  if (n >= 1<< 8) { n >>=  8; pos +=  8; }
  if (n >= 1<< 4) { n >>=  4; pos +=  4; }
  if (n >= 1<< 2) { n >>=  2; pos +=  2; }
  if (n >= 1<< 1) {           pos +=  1; }
  return ((n == 0) ? (-1) : pos);
}

// QuickSortExternal_4+GB.c [

int strcmpKAZE13 (
        const char * src,
        const char * dst
        )
{
        int ret = 0 ;

        while( ! (ret = *(unsigned char *)src - *(unsigned char *)dst) && (*dst!=13-13))
                ++src, ++dst;

        if ( ret < 0 )
                ret = -1 ;
        else if ( ret > 0 )
                ret = 1 ;

        return( ret );
}

//#define LongestLineInclusive 51 //31 former, CAUTION: for command line options 'x' and 'y' it cannot be other than 31 [YET]!

#ifdef singleton
#define LongestLineInclusive 31
#endif
#ifdef doubleton
#define LongestLineInclusive 41
#endif
#ifdef tripleton
#define LongestLineInclusive 41
#endif
#ifdef quadrupleton
#define LongestLineInclusive 51
#endif
#ifdef quintupleton
#define LongestLineInclusive 61
#endif
#ifdef sextupleton
#define LongestLineInclusive 71
#endif
#ifdef septupleton
#define LongestLineInclusive 81
#endif
#ifdef octupleton
#define LongestLineInclusive 91
#endif
#ifdef nonupleton
#define LongestLineInclusive 101
#endif
#ifdef decupleton
#define LongestLineInclusive 111
#endif

// _ngram_  1  1-31
// _ngram_  2  5-41
// _ngram_  3  9-41
// _ngram_  4 13-51
// _ngram_  5 17-61
// _ngram_  6 21-71
// _ngram_  7 25-81
// _ngram_  8 29-91
```



```
// _ngram_  9 33-101
// _ngram_ 10 37-111
// 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

char FourGramL[LongestLineInclusive+1+4]; // 31bytes longest 4-gram + 1byte NULL + 4bytes COUNTER
char FourGramR[LongestLineInclusive+1+4]; // 31bytes longest 4-gram + 1byte NULL + 4bytes COUNTER
char LEAF[8+8+8+2*(LongestLineInclusive+1+4)]; // 136bytes = 3 pointers + 2 keys
char LEAFNEW[8+8+8+2*(LongestLineInclusive+1+4)]; // 136bytes = 3 pointers + 2 keys
FILE *fp_outRG; // Global - not to burden the extract/compare function with one more parameter
int CompareStringsEndingWith13_EXTERNAL(unsigned long long AtPosition64L, unsigned long long AtPosition64R) {

int i;
unsigned long long *AtPosition64Lpointer=&AtPosition64L;
unsigned long long *AtPosition64Rpointer=&AtPosition64R;

// Caramba: seek and tell report OK but in fact they lie, only setpos works?!?!?!

//#if defined(_WIN32_ENVIRONMENT_)
//_lseeki64( fileno(fp_outRG), AtPosition64L, 0 );
//#else
//fseeko( fp_outRG, AtPosition64L, SEEK_SET );
//#endif /* defined(_WIN32_ENVIRONMENT_)  */

//  _CRTIMP __int64 __cdecl _telli64(int);
//  off64_t ftello64 (FILE *stream)


fsetpos(fp_outRG, AtPosition64Lpointer);
for (i=0; i<(LongestLineInclusive+1+4); i++) {fread(&FourGramL[i], 1, 1, fp_outRG); if (FourGramL[i]==13-13) break;}
//Commented line below is slower than the one above: 778156 clocks vs 756297 clocks.
//fread(&FourGramL[0], 31+1, 1, fp_outRG);

fsetpos(fp_outRG, AtPosition64Rpointer);
for (i=0; i<(LongestLineInclusive+1+4); i++) {fread(&FourGramR[i], 1, 1, fp_outRG); if (FourGramR[i]==13-13) break;}
//Commented line below is slower than the one above: 778156 clocks vs 756297 clocks.
//fread(&FourGramR[0], 31+1, 1, fp_outRG);

return(strcmpKAZE13(FourGramL, FourGramR));
}

int CompareStringsEndingWith13_INTERNAL(unsigned long long AtPosition64L, unsigned long long AtPosition64R, char *POOLinternal) {

int i;
//char FourGramL[LongestLineInclusive+2]; // 31 longest 4-gram + CR + LF
//char FourGramR[LongestLineInclusive+2]; // 31 longest 4-gram + CR + LF

for (i=0; i<(LongestLineInclusive+1+4); i++) {
//fread(&FourGramL[i], 1, 1, fp_in);
FourGramL[i] = *(char *)(POOLinternal + AtPosition64L);
if (FourGramL[i]==13-13) break;
}

for (i=0; i<(LongestLineInclusive+1+4); i++) {
//fread(&FourGramR[i], 1, 1, fp_in);
FourGramR[i] = *(char *)(POOLinternal + AtPosition64R);
if (FourGramR[i]==13-13) break;
}

return(strcmpKAZE13(FourGramL, FourGramR));
}

// QuickSortExternal_4+GB.c ]


int main( argc, argv )
int argc; char *argv[];
{
   int nlines;
   string *backup = NULL;

      FILE *fp_in, *fp_out, *fp_outLOG, *fp_inLINE;
      int LetterOffset;
      unsigned long long FilesLEN;
      unsigned long long WORDcount;
      unsigned long long WORDcountBOTTOM;
      unsigned long long WORDcountAttemptsToPut;
      int Thunderwith;
        unsigned long NumberOfFiles, WORDcountDistinct, WORDcountDistinctTOTAL = 0, TotalMemoryNeededForOnePass = 0;
        unsigned long long NumberOfLines;  // rev. 12+
      unsigned long WHOLEletter_BufferSize;
      unsigned long long WHOLEletter_BufferSize_L14;
        unsigned long memory_size, LetterBuffer, j, k, LINE10len, wrdlen;
        unsigned long k_FIX;
        unsigned long long i;              // rev. 12+
        //unsigned long size_in, size_out, size_inLINE;
        unsigned long size_in;             // rev. 12+
#if defined(_WIN32_ENVIRONMENT_)
```

```
2706        unsigned long long size_inLINESIXFOUR;
2707 #else
2708        size_t size_inLINESIXFOUR;
2709 #endif /* defined(_WIN32_ENVIRONMENT_)  */
2710
2711        //unsigned long t1, t2, t3;
2712        time_t t1, t2, t3, t4,tMainB,tMainE;
2713
2714        const int NumberOfSLOTs = 4096*2; // Since r.12+ in rev.12 it was 4096
2715    unsigned long StackPtr;
2716    //unsigned long BSTstack [65536*3]; // BST in worst case could become a LL.
2717    unsigned long long BSTstack [8192*3]; // BST in worst case could become a LL.
2718 unsigned long NumberOfTrees=0, NumberOfHashCollisions=0;
2719 unsigned long iBSTwithMAXpeak, jBSTwithMAXpeak;
2720    unsigned int PEAKibBST;
2721    unsigned long BSTsTotalLEAFs=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur below where 'break'
     is ?!
2722    unsigned long BSTwithMAXnode=0, BSTcurrentNode=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
     below where 'break' is ?!
2723    unsigned long                  BSTcurrentNodeMAXqUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
     BSTcurrent occur below where 'break' is ?!
2724    unsigned long BSTwithMAXnodePEAK=1, BSTwithMAXnodeLEAF=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
     occur below where 'break' is ?!
2725    unsigned long BSTwithMAXpeak=0, BSTcurrentPeak=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
     below where 'break' is ?!
2726    unsigned long                  BSTcurrentPeakMAX=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
     below where 'break' is ?!
2727    unsigned long                  BSTcurrentPeakMAXqUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
     BSTcurrent occur below where 'break' is ?!
2728    unsigned long BSTwithMAXpeakNODE=1, BSTwithMAXpeakLEAF=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
     occur below where 'break' is ?!
2729    unsigned long BSTwithMAXleaf=0, BSTcurrentLeaf=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
     below where 'break' is ?!
2730    unsigned long                  BSTcurrentLeafMAXqUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
     BSTcurrent occur below where 'break' is ?!
2731    unsigned long BSTwithMAXleafNODE=1, BSTwithMAXleafPEAK=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
     occur below where 'break' is ?!
2732
2733        char *pointerflush, *pointerflushUNALIGN, *BufStart, *Flushing;
2734        char *pointerflush_64, *pointerflushUNALIGN_64; // r.14++
2735        unsigned long PseudoLinkedPointer, PseudoLinkedPointerNEW, PseudoLinkedPointerROOT, PseudoLinkedPointerNEWold;
2736        unsigned long PseudoLinkedPointerNEWleft, PseudoLinkedPointerNEWright;
2737        unsigned long PseudoLinkedPointerNEWmiddle;
2738        char *bufend[ 806 ]; // 'a'=0, ... 'z'=25 - 26 letters x 31 lengths
2739        long bufNumberOfWords[ 806 ]; // 'a'=0, ... 'z'=25 - 26 letters x 31 lengths
2740        // long bufNoWpS[ 806 ][ 8192 ]; // ?! crashes below when an attempt to use it occur
2741        char wrd[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2742        char wrdUP[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2743        char wrdUPold[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2744        char LINE1O[257]; // OOO..255, 256 = 0
2745        char ZEROS[4]; // 0..3, 0 = 0, 1 = 0, 2 = 0, 3 = 0
2746        char CRdLFa[2];  // 0..1, 0 = 13, 1 = 10
2747        char workbyte;
2748        char workK[1024*128];
2749        long workKoffset = -1;
2750        int FoundInLinkedList, Slot;
2751        unsigned long OffsetsInBuffer[31]; // OO..30
2752        unsigned long MAXusedBuffer[32]; // OO not used, only O1..31
2753        unsigned long GRMBLhill[32]; // OO..31
2754        unsigned long GRMBLFoolAgain[32]; // OO..31
2755        int Melnitchka;
2756        unsigned long MAXusedBufferABS = 0;
2757        unsigned long Utiliza1 = 0;
2758        unsigned long Utiliza2 = 0;
2759        unsigned long TotalWLchars = 0;
2760
2761 /* minimum signed 64 bit value */
2762 #define _I64_MIN    (-9223372036854775807i64 - 1)
2763 /* maximum signed 64 bit value */
2764 #define _I64_MAX      9223372036854775807i64
2765 /* maximum unsigned 64 bit value */
2766 #define _UI64_MAX     0xffffffffffffffffui64
2767
2768 /* minimum signed 128 bit value */
2769 #define _I128_MIN   (-170141183460469231731687303715884105727i128 - 1)
2770 /* maximum signed 128 bit value */
2771 #define _I128_MAX     170141183460469231731687303715884105727i128
2772 /* maximum unsigned 128 bit value */
2773 #define _UI128_MAX    0xffffffffffffffffffffffffffffffffui128
2774
2775        char llTOaDigits[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
2776        // below duplicates are needed because of one_line_invoking need different buffers.
2777        char llTOaDigits2[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
2778        char llTOaDigits3[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
2779        char llTOaDigits4[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('\0')+6(,)
2780        unsigned long HEADOffetFromStartBUKVA = 0;
2781        unsigned long TAILOffetFromStartBUKVA = 0;
2782        int BSTorBtree = 0;
```



```
2783        int SplitOccured;
2784        int POffsetInLEAF;
2785 char *Auberge[4] = {"!\0","/\0","-\0","\\\0"};
2786 int hashAlfalfa, iAlfalfa;
2787 int PLE_words=0; // Quadruple!
2788 char wrd1st[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2789 char wrd2nd[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2790 char wrd3rd[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2791 char wrd4th[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2792 char wrd5th[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2793 char wrd6th[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2794 char wrd7th[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2795 char wrd8th[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2796 char wrd9th[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2797 char wrd10th[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2798 char *DelimiterUnderscore = "_\0";
2799 int PLE_words_INITflag = 0;
2800
2801 // QuickSortExternal_4+GB [
2802 unsigned long long ThunderwithL64_L14;
2803 unsigned long long Strnglen64_L14;
2804 unsigned long long size_in64_L14, size_in2_L14;
2805 unsigned long long Over4billionLines, j_Over4billion;
2806 char OneChar_ieByte = '\0';
2807 char CR_ieByte = '\r';
2808 char SomeByte;
2809 unsigned long long BufEnd_64;
2810 unsigned long long SeekPosition;
2811 unsigned long long *PointerToSeekPosition;
2812 char FourGram[LongestLineInclusive+2]; // 31 longest 4-gram + CR + LF
2813 char *PoolPhysical;
2814 unsigned long long fsetpos_ZERO=0;
2815 char OneCkusterZEROES[1024*4]; // Caution: must be ZEROed(NULLified)!
2816 char *FileSwapTag = "LEPRECHAUNISH";
2817 char EOFcode = 0x1A;
2818 unsigned long long PseudoLinkedPointer_64, PseudoLinkedPointerNEW_64, PseudoLinkedPointerROOT_64, PseudoLinkedPointerNEWold_64;
2819        unsigned long long PseudoLinkedPointerNEWleft_64, PseudoLinkedPointerNEWright_64;
2820        unsigned long long PseudoLinkedPointerNEWmiddle_64;
2821        unsigned long long NULLs_64 = 0;
2822 unsigned long long PseudoLinkedPointerAUX_64;
2823 unsigned long long PseudoLinkedPointerAUXdumbo_64;
2824        char wrdAUX[LongestLineInclusive+1+4];  // 0..30, 31 = 0
2825 // QuickSortExternal_4+GB ]
2826
2827 unsigned long CounterOccurrencies;
2828 unsigned long long NumberOfLEAFs=0;
2829 unsigned long LevelsInCorona_Not_Counting_ROOT=0;
2830 char *ngram[11] =
     {"NULLleton\0","singleton\0","doubleton\0","tripleton\0","quadrupleton\0","quintupleton\0","sextupleton\0","septupleton\0","octupleton\0","non
     upleton\0","decupleton\0"};
2831
2832 unsigned long RipPasses;
2833 unsigned long long NULLsForWRD=0;
2834
2835 //15+
2836 int DoNotInsertFlag = 0;
2837 int METACOMMANDflag;
2838
2839 //16
2840 int REUSE=0;
2841 int HSHexist;
2842 int SWPexist;
2843
2844 // INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT
2845 printf ("Leprechaun_%s (Fast-In-Future Greedy n-gram-Ripper), rev. 16FIXFIX, written by Svalqyatchx.\n", ngram[_ngram_]);
2846 //puts( "Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'" );
2847
2848 #ifdef singleton
2849 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 1..31 chars from incoming texts.\n", _ngram_, _ngram_);
2850 #endif
2851 #ifdef doubleton
2852 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 5..41 chars from incoming texts.\n", _ngram_, _ngram_);
2853 #endif
2854 #ifdef tripleton
2855 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 9..41 chars from incoming texts.\n", _ngram_, _ngram_);
2856 #endif
2857 #ifdef quadrupleton
2858 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 13..51 chars from incoming texts.\n", _ngram_, _ngram_);
2859 #endif
2860 #ifdef quintupleton
2861 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 17..61 chars from incoming texts.\n", _ngram_, _ngram_);
2862 #endif
2863 #ifdef sextupleton
2864 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 21..71 chars from incoming texts.\n", _ngram_, _ngram_);
2865 #endif
2866 #ifdef septupleton
2867 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 25..81 chars from incoming texts.\n", _ngram_, _ngram_);
2868 #endif
```

```
2869 #ifdef octupleton
2870 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 29..91 chars from incoming texts.\n", _ngram_, _ngram_);
2871 #endif
2872 #ifdef nonupleton
2873 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 33..101 chars from incoming texts.\n", _ngram_, _ngram_);
2874 #endif
2875 #ifdef decupleton
2876 printf ("Purpose: Rips all distinct %d-grams (%d-word phrases) with length 37..111 chars from incoming texts.\n", _ngram_, _ngram_);
2877 #endif
2878 puts( "Feature1: All words within x-lets/n-grams are in range 1..31 chars inclusive." );
2879 //puts( "Feature2: In this revision 128MB 1-way hash is used which results in 16,777,216 external B-Trees of order 3." );
2880
2881 if (HashInBITS+3<10)
2882 printf ("Feature2: In this revision %sbytes 1-way hash is used which results in %s external B-Trees of order 3.\n",
     _ui64toaKAZEcomma(((1<<HashInBITS)<<3), llTOaDigits, 10), _ui64toaKAZEcomma((1<<HashInBITS), llTOaDigits2, 10) );
2883 else if (HashInBITS+3>=10 && HashInBITS+3<20)
2884 printf ("Feature2: In this revision %sKB 1-way hash is used which results in %s external B-Trees of order 3.\n", _ui64toaKAZEcomma(
     (((1<<HashInBITS)<<3))>>10, llTOaDigits, 10), _ui64toaKAZEcomma((1<<HashInBITS), llTOaDigits2, 10) );
2885 else
2886 printf ("Feature2: In this revision %sMB 1-way hash is used which results in %s external B-Trees of order 3.\n", _ui64toaKAZEcomma(
     (((1<<HashInBITS)<<3))>>20, llTOaDigits, 10), _ui64toaKAZEcomma((1<<HashInBITS), llTOaDigits2, 10) );
2887 if (HashInBITS-HashChunkSizeInBITS==0)
2888 printf ("Feature3: In this revision, %s pass is to be made.\n", _ui64toaKAZEcomma(1<<(HashInBITS-HashChunkSizeInBITS), llTOaDigits, 10));
2889 else
2890 printf ("Feature3: In this revision, %s passes are to be made.\n", _ui64toaKAZEcomma(1<<(HashInBITS-HashChunkSizeInBITS), llTOaDigits, 10));
2891
2892 puts( "Feature4: If the external memory has latency 99+microseconds then !(look no further), IOPS(seek-time) rules." );
2893 // The phrase 'look no further' was used in amazon.com review meaning 'stop searching for better thing this is it'.
2894 //puts( "Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us," );
2895 //puts( "      also the performance of a 3-way hash + 6,602,752 B-Trees of order 3," );
2896 //puts( "      also the performance of a 1-way hash + 134,217,728 external B-Trees of order 3." );
2897 //puts( "Note1: Compiled with Microsoft C v. 13.10.3077: 'cl /Ox /TcLeprechaun.c'." );
2898 //puts( "Note2: This WORDLISTER makes as output pseudo(unsorted)_wordlist_CRLF_file." );
2899 if( argc != 3 && argc != 4 && argc != 5 && argc != 6 ) // +1 for program name
2900 {
2901   puts( "" );
2902 puts( "'The Little Monster' short notes:" );
2903 puts( "Note1: I wish to thank to R.N. Horspool, Ranjan Sinha, Dmitry Shkarin," );
2904 puts( "       Michael Abrash, J. Bentley, R. Sedgewick, Igor Pavlov, Lasse Reinhold," );
2905 puts( "       Landon Noll, Peter Kankowski for sharing their knowledge to public." );
2906 puts( "Note2: Run it without parameters to get usage and short notes." );
2907 puts( "Note3: This simple amateurish(more over I am not versed well neither in C nor" );
2908 puts( "       in mathematics nor in English language, but I am persistent in INDEXING" );
2909 puts( "       GBs of english TEXTS) tool is written in ANSI C(at least its source is" );
2910 puts( "       compileable for CL(Windows) and GCC(Linux)), and its purpose is to" );
2911 puts( "       create a Wordlist for a group of files(given via filelist)." );
2912 puts( "       Its name comes(according to Heritage Dictionary) from 'low corpus' or" );
2913 puts( "       'little body', in fact from amazing movie saga 'Leprechaun 1-2-3-4-5-6'" );
2914 puts( "       starring by Warwick Davis." );
2915 puts( "Note4: Only words up to 31 chars are proceeded - the reason is 'DDT'(the" );
2916 puts( "       longest word in Heritage Dictionary 3rd edition) or" );
2917 puts( "       'dichlorodiphenyltrichloroethane'." );
2918 puts( "Note5: Cursor hiding in C - mission impossible for me." );
2919 puts( "Note6: By default(third parameter is 1023) allocated memory is 393MB." );
2920 puts( "       Due to 'malloc()' limitation under WINDOWS, maximum value of third" );
2921 puts( "       parameter is 5174 which is 1988MB allocated block." );
2922 puts( "Note7: File Leprechaun.LOG is a log, where new statistics are appended." );
2923 puts( "Note8: Revision 12+ can handle files larger than 4GB." );
2924 puts( "Note9: Revision 12++ has a buffered 'fread()' - therefore I/O READ-BURST SPEED" );
2925 puts( "       is the first(worst) bottleneck, as a result r.12++ is much-much faster;" );
2926 puts( "       the second(worse) bottleneck: the linked lists - the b-trees" );
2927 puts( "       might be the answer; the third(bad) bottleneck: the amateurish author." );
2928 puts( "NoteA: Revision 12+++ has an improved(2 bits were used doltishly) main hash" );
2929 puts( "       function - therefore less collisions, for example:" );
2930 puts( "       for file 'wikipedia-de-html.tar' 42,291,855,360 bytes with" );
2931 puts( "       5,750,179,678 words of them 7,375,373 distinct attempts to Find/Put" );
2932 puts( "       a WORD into a linked list are 6,117,675,470(r.12++) and 5,845,989,790" );
2933 puts( "       (r.12+++); also two 'if' sections were moved because they were executed" );
2934 puts( "       unnecessarily many times." );
2935 puts( "NoteB: Revision 13 uses BSTs instead of LLs, that is Linked-Lists were" );
2936 puts( "       replaced by Binary-Search-Trees, as a result for 22,202,980 distinct" );
2937 puts( "       words(out of 35,271,297) r.12+++ needs 225,548,268 total attempts to" );
2938 puts( "       Find/Put WORDs into linked lists where r.13 needs 121,674,042 total" );
2939 puts( "       attempts to Find/Put WORDs into Binary-Search-Trees. But this is a" );
2940 puts( "       significant boost in performance only for wordlists of million words." );
2941 puts( "NoteC: Revision 13+ gives only more statistics. Future revisions could lessen" );
2942 puts( "       number of attempts to Find/Put WORDs into Binary-Search-Trees" );
2943 puts( "       furthermore by making them at some point Perfectly-Balanced. But" );
2944 puts( "       for huge amount(multi-(m|b)illion) of distinct words the b-tree family" );
2945 puts( "       must come in, until then this is the leprechaunish niche." );
2946 puts( "NoteD: Revision 13++ has a little fix(2 unnecessary ZEROings, when a new word" );
2947 puts( "       is inserted, were deleted) and a fixed bug(13+ adds stupidly the" );
2948 puts( "       highest BST to the wordlist). Also B-Tree of order 3 is added as a" );
2949 puts( "       searching method. Main goal of B-Tree is to reduce number of" );
2950 puts( "       comparisons but at nasty cost: a precious time wasted to construct it" );
2951 puts( "       and twice more memory, i.e. one step forward two backward: this tree is" );
2952 puts( "       more effective than BST in cases of 2++ billion/million" );
2953 puts( "       different/distinct words." );
```



```
2954 puts( "       The improvement which comes from using B-Tree of order 3 is about 200%" );
2955 puts( "       much more pleasing than I expected, for wikipedia-en-html.tar.wrd with" );
2956 puts( "       12,561,874 distinct words Total Attempts to Find/Put WORDs into:" );
2957 puts( "       Binary-Search-Trees was 61,895,043 while for" );
2958 puts( "       B-trees order 3 was 19,295,791." );
2959 puts( "NoteE: Revision 13+++ has a faster(not heavily tested yet) and with" );
2960 puts( "       better(0.6% to 1.1%) dispersion Fowler/Noll/Vo hash," );
2961 puts( "       so called FNV1a hash. Revision 13+++++ boosting: Leprechaun_Intel.exe" );
2962 puts( "       gives 1,256,187W/s for wikipedia-en-html.tar.wrd with FNV1_32_PRIME:" );
2963 puts( "       107712257 with 3,551,736 dispersion for 'FNV1A_Hash_Granularity'." );
2964 puts( "NoteF: For old r.12+ a USB connected HDD crippled test:" );
2965 puts( "       for 'H:\\>Leprechaun.exe static.wikipedia.org_downloads_2008-06_en.lst" );
2966 puts( "       wikipedia-en-html.tar.wrd 5400'" );
2967 puts( "       where 223,674,511,360 wikipedia-en-html.tar" );
2968 puts( "       on laptop Toshiba Pentium T3400 2166 MHz with" );
2969 puts( "       Motherboard Name:                                  Toshiba Satellite L305" );
2970 puts( "       CPU Type:          Mobile DualCore Intel Pentium, 2166 MHz (13 x 167)" );
2971 puts( "       CPU Alias:                                                 Merom-1M" );
2972 puts( "       L1 Code Cache:                                          32 KB per core" );
2973 puts( "       L1 Data Cache:                                          32 KB per core" );
2974 puts( "       L2 Cache:                        1 MB (On-Die, ECC, ASC, Full-Speed)" );
2975 puts( "       Bus Type:                                            Dual DDR2 SDRAM" );
2976 puts( "       Bus Width:                                                128-bit" );
2977 puts( "       Real Clock:                                        333 MHz (DDR)" );
2978 puts( "       Effective Clock:                                          666 MHz" );
2979 puts( "       EVEREST v5.00.1650 Memory Copy:       3725MB/s with timings 5-5-5-13" );
2980 puts( "       result is logged to 'Leprechaun.LOG':" );
2981 puts( "  Bytes per second performance: 20,658,955B/s" );
2982 puts( "  Words per second performance: 2,860,880W/s" );
2983 puts( "  Input File with a list of TEXTual Files:" );
2984 puts( "   static.wikipedia.org_downloads_2008-06_en.lst" );
2985 puts( "  Size of all TEXTual Files: 223,674,511,360" );
2986 puts( "  Word count: 30,974,750,142 of them 12,561,874 distinct" );
2987 puts( "  Number Of Files: 1" );
2988 puts( "  Number Of Lines: 2088618575" );
2989 puts( "  Allocated memory in MB: 1920" );
2990 puts( "  Words with length 01 occupy 0,033KB of 0,349KB given i.e. 09% utilization" );
2991 puts( "  Words with length 02 occupy 0,033KB of 0,349KB given i.e. 09% utilization" );
2992 puts( "  Words with length 03 occupy 0,037KB of 0,697KB given i.e. 05% utilization" );
2993 puts( "  Words with length 04 occupy 0,151KB of 0,871KB given i.e. 17% utilization" );
2994 puts( "  Words with length 05 occupy 0,744KB of 1,568KB given i.e. 47% utilization" );
2995 puts( "  Words with length 06 occupy 1,470KB of 3,136KB given i.e. 46% utilization" );
2996 puts( "  Words with length 07 occupy 2,605KB of 5,923KB given i.e. 43% utilization" );
2997 puts( "  Words with length 08 occupy 3,296KB of 6,968KB given i.e. 47% utilization" );
2998 puts( "  Words with length 09 occupy 3,714KB of 6,968KB given i.e. 53% utilization" );
2999 puts( "  Words with length 10 occupy 3,483KB of 6,968KB given i.e. 49% utilization" );
3000 puts( "  Words with length 11 occupy 3,235KB of 5,923KB given i.e. 54% utilization" );
3001 puts( "  Words with length 12 occupy 2,691KB of 4,181KB given i.e. 64% utilization" );
3002 puts( "  Words with length 13 occupy 2,230KB of 3,484KB given i.e. 64% utilization" );
3003 puts( "  Words with length 14 occupy 1,718KB of 3,484KB given i.e. 49% utilization" );
3004 puts( "  Words with length 15 occupy 1,357KB of 2,613KB given i.e. 51% utilization" );
3005 puts( "  Words with length 16 occupy 1,063KB of 2,613KB given i.e. 40% utilization" );
3006 puts( "  Words with length 17 occupy 0,814KB of 1,742KB given i.e. 46% utilization" );
3007 puts( "  Words with length 18 occupy 0,617KB of 1,742KB given i.e. 35% utilization" );
3008 puts( "  Words with length 19 occupy 0,485KB of 1,742KB given i.e. 27% utilization" );
3009 puts( "  Words with length 20 occupy 0,402KB of 1,742KB given i.e. 23% utilization" );
3010 puts( "  Words with length 21 occupy 0,327KB of 1,742KB given i.e. 18% utilization" );
3011 puts( "  Words with length 22 occupy 0,274KB of 1,742KB given i.e. 15% utilization" );
3012 puts( "  Words with length 23 occupy 0,224KB of 1,394KB given i.e. 16% utilization" );
3013 puts( "  Words with length 24 occupy 0,190KB of 1,394KB given i.e. 13% utilization" );
3014 puts( "  Words with length 25 occupy 0,162KB of 1,394KB given i.e. 11% utilization" );
3015 puts( "  Words with length 26 occupy 0,136KB of 1,220KB given i.e. 11% utilization" );
3016 puts( "  Words with length 27 occupy 0,119KB of 1,046KB given i.e. 11% utilization" );
3017 puts( "  Words with length 28 occupy 0,107KB of 0,871KB given i.e. 12% utilization" );
3018 puts( "  Words with length 29 occupy 0,091KB of 0,697KB given i.e. 13% utilization" );
3019 puts( "  Words with length 30 occupy 0,080KB of 0,523KB given i.e. 15% utilization" );
3020 puts( "  Words with length 31 occupy 0,076KB of 0,523KB given i.e. 14% utilization" );
3021 puts( "  Total pseudo(including hash table) memory utilization: 42%" );
3022 puts( "  Total real(wordlist's words VS allocated block) memory utilization: 60/100" );
3023 puts( "  Used value for third parameter in KB: 5400" );
3024 puts( "  Use next time as third parameter: 3475-" );
3025 puts( "  Time for making unsorted wordlist: 10827 second(s)" );
3026 puts( "  Time for sorting unsorted wordlist: 10 second(s)" );
3027 puts( "NoteG: 2011-Mar-07: Fixed a small command line parsing bug." );
3028 puts( "NoteH: A heavy blow for my illusions(regarding speed performance of external b-trees)," );
3029 puts( "       desperate results for ripping on HDD 7200rpm:" );
3030 puts( "       20,000,000 distinct 4-grams per 5 hours." );
3031 puts( "       D:\\Leprechaun_quadrupleton_r14_minus>Leprechaun_quadrupleton GRAFFITH_2048.lst GRAFFITH_2048.wrd 48000000 z" );
3032 puts( "       Leprechaun(Fast Greedy Word-Ripper), rev. 14_minus_quadrupleton, written by Svalqyatchx." );
3033 puts( "       Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'" );
3034 puts( "       Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us," );
3035 puts( "             also the performance of a 3-way hash + 6,602,752 B-Trees of order 3," );
3036 puts( "             also the performance of a 1-way hash + 134,217,728 external B-Trees of order 3." );
3037 puts( "       Size of input file with files for Leprechauning: 42140" );
3038 puts( "       Allocating HASH memory 1,073,741,889 bytes ... OK" );
3039 puts( "       Allocating/ZEROing 49,152,000,014 bytes swap file ... OK" );
3040 puts( "       Size of Input TEXTual file: 33,470,581" );
3041 puts( "       |; Word count: 3,045,077 of them 2,597,942 distinct; Done: 64/64" );
```

```
3042 puts( "       Size of Input TEXTual file: 17,229,900" );
3043 puts( "       -; Word count: 4,235,032 of them 3,588,757 distinct; Done: 64/64" );
3044 puts( "       Size of Input TEXTual file: 19,191,256" );
3045 puts( "       |; Word count: 5,803,400 of them 4,866,213 distinct; Done: 64/64" );
3046 puts( "       Size of Input TEXTual file: 34,651,077" );
3047 puts( "       \\; Word count: 8,714,961 of them 6,941,108 distinct; Done: 64/64" );
3048 puts( "       Size of Input TEXTual file: 26,875,458" );
3049 puts( "       /; Word count: 11,022,830 of them 8,579,931 distinct; Done: 64/64" );
3050 puts( "       Size of Input TEXTual file: 19,605,129" );
3051 puts( "       -; Word count: 12,924,821 of them 10,078,191 distinct; Done: 64/64" );
3052 puts( "       Size of Input TEXTual file: 17,053,521" );
3053 puts( "       /; Word count: 14,577,010 of them 11,455,983 distinct; Done: 64/64" );
3054 puts( "       Size of Input TEXTual file: 44,087,709" );
3055 puts( "       -; Word count: 18,953,280 of them 15,010,569 distinct; Done: 64/64" );
3056 puts( "       Size of Input TEXTual file: 32,796,705" );
3057 puts( "       |; Word count: 22,412,912 of them 17,621,649 distinct; Done: 64/64" );
3058 puts( "       Size of Input TEXTual file: 19,538,360" );
3059 puts( "       /; Word count: 24,381,005 of them 19,137,701 distinct; Done: 64/64" );
3060 puts( "       Size of Input TEXTual file: 29,565,366" );
3061 puts( "       \\; Word count: 26,214,400 of them 20,528,357 distinct; Done: 40/64" );
3062 puts( "       ..." );
3063 puts( "NoteI: In revision 14- the resultant wordlist is NOT sorted when 'Z' is used." );
3064 puts( "NoteJ: In revision 14 'x' and 'y' options are disabled, for 7++ million phrases their usefulness is no more," );
3065 puts( "       the real loads are of order 800+ million, too many limitations exist, they must be rewritten as 64bit." );
3066 puts( "NoteK: Ripping OSHO.TXT (10,165,640 4-grams) on HDD daunts because of 6+hours needed:" );
3067 puts( "       Number Of Trees(GREATER THE BETTER): 9,433,894" );
3068 puts( "       Used value for third parameter in KB: 3,145,728" );
3069 puts( "       Use next time as third parameter: 1,262,186" );
3070 puts( "       One leaf has size: 8+8+8+(51+1+4)+(51+1+4)=136bytes," );
3071 puts( "       or MAX (one 4-gram per leaf) 10,165,640*136=1,382,527,040bytes." );
3072 puts( "NoteL: Each phrase in extracted file is preceded by TAB ASCII code, this (TAB being a delimiter symbol) allows" );
3073 puts( "       the phrase-list to be ripped again i.e. to treat already ripped files as any other text." );
3074 puts( "NoteM: Too many 'fsetpos', 'fread', 'fwrite' invocations were put in the straight port (from 32bit internal memory to" );
3075 puts( "       64bit external memory), a optimization is needed, something like reading/writing a LEAF at once." );
3076 puts( "NoteN: Since revision 14+: Optimized(LEAFwise) search (fragment 1] and 2]), insert (fragment 3]) and dump." );
3077 puts( "NoteO: In next revisions a 2in1 is to be done i.e. one code fragment will deal with virtual and physical memory," );
3078 puts( "       thus establishing pure 64bit mode of operation, a single flag will decide whether 'memcpy' or" );
3079 puts( "       the slow I/O triad sub-fragments will be used. DONE." );
3080 puts( "NoteP: In next revisions a multi-pass (by chunking the hash table) mode is to be added in order to avoid" );
3081 puts( "       these sick-seeks. DONE." );
3082 puts( "NoteQ: Fixed occurrencies bug due to not NULLifying the field housing the occurrencies, a nasty thing: all" );
3083 puts( "       the revisions 14??? were buggy, how stupid from my side, grumble." );
3084 puts( "NoteR: In r.14++++FIXFIX were fixed STATS(Leprechaun.LOG) bugs (appearing only in multi-pass mode) due to not" );
3085 puts( "       NULLifying the variables housing the stats, they do not affect the results - they are for informative use." );
3086 puts( "NoteS: Fixed a division-by-zero bug, occurs when finishing-starting time is under 1 second." );
3087 puts( "       Fixed a nasty bug causing very restrictive way of forming x-grams." );
3088 puts( "NoteT: At last and finally the nasty bug causing very restrictive way of forming x-grams was REALLY fixed - lack of" );
3089 puts( "       calmness jammed (again) my actions - a lesson to be relearnt." );
3090
3091 puts( "NoteU: Since r.15FIXFIX+ the ability to command Leprechaun (from inside the list file with 2 metacommands) to enter/exit" );
3092 puts( "       INSERT mode was added. This allows to control whether new (to current hash-tree structure) x-grams are to be counted" );
3093 puts( "       [and] INSERTed. These two metacommands are:" );
3094 puts( "       Leprechaun says x-gram inserting disabled for next files: ON" );
3095 puts( "       Leprechaun says x-gram inserting disabled for next files: OFF" );
3096
3097 puts( "NoteV: When W/w option is used multiple-passes shouldn't be dumped - it is meaningless, dump when only one pass," );
3098 puts( "       that is, use W/w only in ONE-PASS mode otherwise it behaves as Z/z but DOES NOT dump to OutFile." );
3099 puts( "       It uses in READ mode the two HASH+TREES output files: 'Leprechaun_64bit.hsh' and 'Leprechaun_64bit.swp'." );
3100 puts( "       If during the start one of them is missing then Z/z behaviour is on, at end 'Leprechaun_64bit.hsh' is dumped." );
3101 puts( "       Also the OutFile has all incoming x-grams which are present in the corpus (i.e. HASH+TREES structure)." );
3102   puts( "" );
3103   puts( "Usage: Leprechaun InFile OutFile [BufferSize] [SortMethod] [TreeMethod]" );
3104   puts( "      <InFile>: Input file with files for Leprechauning, in WINDOWS console" );
3105   puts( "                you can create it by 'E:\\KAZEHOME>dir *.txt/s/b>Leprechaun.lst'" );
3106   puts( "      <OutFile>: Output WORDLIST(sorted since r.9, CRLF) file" );
3107   puts( "      <BufferSize>: Optional Dynamic RAM buffer in KB, default(and minimum" );
3108   puts( "                    in the same time) is 1023, i.e. omit or specify greater one" );
3109   puts( "      <SortMethod>: Optional Sort Method, default is 'D'," );
3110   puts( "                    A - InsertionSort" );
3111   puts( "                    B - InsertionX26Sort" );
3112   puts( "                    C - MultiKeyQuickSortSort by J. Bentley, R. Sedgewick" );
3113   puts( "                    D - MultiKeyQuickSortx26Sort' by J. Bentley, R. Sedgewick" );
3114   puts( "      <TreeMethod>: Optional Tree Method, default is 'X'," );
3115   puts( "                    X - Binary-Search-Trees" );
3116   puts( "                    y - B-Trees of order 3, INTERNAL/fast memory digitless i.e. no repetitions, 64bit addressing!" );
3117   puts( "                    Y - B-Trees of order 3, INTERNAL/fast memory, 64bit addressing!" );
3118   puts( "                    z - B-Trees of order 3, EXTERNAL/slow memory digitless i.e. no repetitions, 64bit addressing!" );
3119   puts( "                    Z - B-Trees of order 3, EXTERNAL/slow memory, 64bit addressing!" );
3120   puts( "                    w - B-Trees of order 3, EXTERNAL/slow memory digitless i.e. no repetitions, 64bit addressing! REUSE!" );
3121   puts( "                    W - B-Trees of order 3, EXTERNAL/slow memory, 64bit addressing! REUSE!" );
3122   puts( "" );
3123 puts( "Have a nice Leprechauning." );
3124 puts( "For contacts: sanmayce@sanmayce.com" );
3125 puts( "Sanmayce Svalqyatchx 'Kaze', 2005 Feb 07. Last revision: 2013 Mar 31." );
3126   return( 1 );
3127 }
3128
3129   GRMBLhill[0]=0;
```



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

```
    GRMBLFoolAgain[(int)i] = (GRMBLhill[(int)i] * LetterBuffer)/31;
  }
  memory_size = 26 * WHOLEletter_BufferSize + 1 + 64;
printf( "Allocating memory %luMB ... ", (memory_size>>20)+1 );
pointerflushUNALIGN = (char *)malloc( memory_size );
if( pointerflushUNALIGN == NULL )
{ puts( "\nLeprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
pointerflush = pointerflushUNALIGN + 64 - (((size_t)pointerflushUNALIGN) % 64); // 13_6+
//offset=64-int((long)data&63);

printf( "OK\n");
                fprintf( fp_outLOG, "Leprechaun report:\n" );

// Check once for ever whether allocated memory is ZEROed!? Answer: YES
//for( i = 0; i < memory_size; i++ )
// if (*(char *)(pointerflush+i)!=0) printf("NON-ZERO encountered, so 'NO'.");

for( i = 0; i < 26; i++ )
{ for( k = 1; k <= 31; k++ )
  { bufend[i*31+k-1] = pointerflush + i * WHOLEletter_BufferSize + OffsetsInBuffer[k-1]; // i*31+k-1 must be 0..805
      if (i==25) { MAXusedBuffer[k] = (unsigned long)bufend[i*31+k-1]; }
    for( j = 0; j < (NumberOfSLOTs+1)*4; j++ ) // ? memset(bufend[i],0,(NumberOfSLOTs+1)*4);
    { *bufend[i*31+k-1]++ = 0;
      //++bufend[i*31+k-1];
    }
      if (i==25) { MAXusedBuffer[k] = (unsigned long)bufend[i*31+k-1]-MAXusedBuffer[k]; }
    bufNumberOfWords[i*31+k-1]=0;
//for( j = 0; j < NumberOfSLOTs; j++ )
//bufNoWpS[i*31+k-1][j]=0;
  }
}

} else { //if (BSTorBtree != 2) {
// _ ASCII code 095
// ` ASCII code 096  \
// a ASCII code 097  / In total 26+1+1 radix instead of 27 to avoid +1 for each '_', code 096 not used.
// z ASCII code 122
// The hash  for 'a_quadruplet_for_example' will be calculated for first 5 chars:
// = (byte1-'_')*28*28*28*28 + (byte2-'_')*28*28*28 + (byte3-'_')*28*28 + (byte4-'_')*28 + (byte5-'_')
// Hash slots are 28*28*28*28*28 = 17,210,368 each containing one 64bit pointer i.e. 8bytes in length.
// Hash size = 17,210,368*8 = 137,682,944 bytes
// When at end all these slots(17,210,368- Btrees) are traversed the outcome is a sorted wordlist - no need of sorting.
//unsigned long long SeekPosition;
//unsigned long long *PointerToSeekPosition;
// 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.
// For final stats all non-zero slots point to one btree.
//   printf( "Allocating HASH memory %s bytes ... ", _ui64toaKAZEcomma( (17210368*8) + 1 + 64 , llTOaDigits, 10) );
//   pointerflushUNALIGN = (char *)malloc( (17210368*8) + 1 + 64 );
// Hash slots are 27bit = 2^27 = 134,217,728 each containing one 64bit pointer i.e. 8bytes in length.
printf( "Allocating HASH memory %s bytes ... ", _ui64toaKAZEcomma( ((1<<HashInBITS)*8) + 1 + 64 , llTOaDigits, 10) );
pointerflushUNALIGN = (char *)malloc( (1<<HashInBITS)*8 + 1 + 64 );
if( pointerflushUNALIGN == NULL )
{ puts( "\nLeprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
// r16
pointerflush = pointerflushUNALIGN;
//pointerflush = pointerflushUNALIGN + 64 - (((size_t)pointerflushUNALIGN) % 64); // 13_6+
//offset=64-int((long)data&63);
printf( "OK\n");
//  memset(pointerflush,0,17210368*8);
memset(pointerflush,0,(1<<HashInBITS)*8);
                        if (BSTorBtree == 2) {
if( ( fp_outRG = fopen( "Leprechaun_64bit.hsh", "rb" ) ) == NULL )
{
    HSHexist=0;
    if ( REUSE && ((HashInBITS-HashChunkSizeInBITS)==0) ) // Multiple-passes shouldn't be uploaded - it is meaningless, dump when only one pass.
                printf( "Leprechaun: Can't find file 'Leprechaun_64bit.hsh'.\n" );
} else {
    HSHexist=1;
fclose(fp_outRG);
}

if( ( fp_outRG = fopen( "Leprechaun_64bit.swp", "rb" ) ) == NULL )
{
    SWPexist=0;
    if ( REUSE && ((HashInBITS-HashChunkSizeInBITS)==0) )
                printf( "Leprechaun: Can't find file 'Leprechaun_64bit.swp'.\n" );
} else {
    SWPexist=1;
fclose(fp_outRG);
}

if ( REUSE && ((HashInBITS-HashChunkSizeInBITS)==0) && (SWPexist+HSHexist == 2) ) {
    REUSE=2;
    if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
    { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
```



```
}
if( ( fp_outRG = fopen( "Leprechaun_64bit.hsh", "rb" ) ) != NULL ) {
    if ( REUSE == 2 ) { //  REUSE [

#if defined(_WIN32_ENVIRONMENT_)
   // 64bit:
_lseeki64( fileno(fp_outRG), 0L, SEEK_END );
size_inLINESIXFOUR = _telli64( fileno(fp_outRG) );
_lseeki64( fileno(fp_outRG), 0L, SEEK_SET );
#else
   // 64bit:
fseeko( fp_outRG, 0L, SEEK_END );
size_inLINESIXFOUR = ftello( fp_outRG );
fseeko( fp_outRG, 0L, SEEK_SET );
#endif /* defined(_WIN32_ENVIRONMENT_)  */
printf( "Uploading-n-Reusing 'Leprechaun_64bit.hsh' file: %s bytes\n", _ui64toaKAZEcomma(size_inLINESIXFOUR, llTOaDigits, 10) );

              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.
    }
fclose(fp_outRG);
}

// Tag for the swap file is: LEPRECHAUNISH{ASCIIcode26}
// or 14bytes, then when type of the swap is requested:
// D:\_KAZE_~1\LEPREC~1>type Leprechaun_64bit.swp
// LEPRECHAUNISH
// D:\_KAZE_~1\LEPREC~1>
size_in64_L14 = 1024 * (unsigned long long)Thunderwith + 14;
BufEnd_64 = 0+14;
// 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.

/*
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:

'r'
    Open an existing file for reading only.
'w'
    Open the file for writing only. If the file already exists, it is truncated to zero length. Otherwise a new file is created.
'a'
    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.
'r+'
    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.
'w+'
    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.
'a+'
    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.
*/

    // 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!
    if ( REUSE == 2 ) { //  REUSE [
if( ( fp_outRG = fopen( "Leprechaun_64bit.swp", "rb+" ) ) == NULL )
{ printf( "Leprechaun: Can't create file 'Leprechaun_64bit.swp'.\n" ); return( 1 ); }

#if defined(_WIN32_ENVIRONMENT_)
   // 64bit:
_lseeki64( fileno(fp_outRG), 0L, SEEK_END );
size_inLINESIXFOUR = _telli64( fileno(fp_outRG) );
_lseeki64( fileno(fp_outRG), 0L, SEEK_SET );
#else
   // 64bit:
fseeko( fp_outRG, 0L, SEEK_END );
size_inLINESIXFOUR = ftello( fp_outRG );
fseeko( fp_outRG, 0L, SEEK_SET );
#endif /* defined(_WIN32_ENVIRONMENT_)  */
printf( "Reusing 'Leprechaun_64bit.swp' file: %s bytes\n", _ui64toaKAZEcomma(size_inLINESIXFOUR, llTOaDigits, 10) );

fsetpos(fp_outRG, &BufEnd_64); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ??!! So DO-IT-OVER.
    }
    else { // REUSE
if( ( fp_outRG = fopen( "Leprechaun_64bit.swp", "wb+" ) ) == NULL )
{ printf( "Leprechaun: Can't create file 'Leprechaun_64bit.swp'.\n" ); return( 1 ); }
printf( "Allocating/ZEROing %s bytes swap file ... ", _ui64toaKAZEcomma(size_in64_L14, llTOaDigits, 10) );
fsetpos(fp_outRG, &fsetpos_ZERO); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ??!! So DO-IT-OVER.
memset(OneCkusterZEROES,0,1024*4);
for (ThunderwithL64_L14=0; ThunderwithL64_L14 < size_in64_L14/(1024*4); ThunderwithL64_L14++)
              fwrite(OneCkusterZEROES, 1024*4, 1, fp_outRG);
for (ThunderwithL64_L14=0; ThunderwithL64_L14 < size_in64_L14%(1024*4); ThunderwithL64_L14++)
              fwrite(&OneChar_ieByte, 1, 1, fp_outRG);
fsetpos(fp_outRG, &fsetpos_ZERO); // SOMETHING ROTTEN with lseeki64/fseeko and fsetpos ??!! So DO-IT-OVER.
              fwrite(FileSwapTag, 13, 1, fp_outRG);
```

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



```
3471    // 64bit:
3472 fseeko( fp_inLINE, 0L, SEEK_END );
3473 size_inLINESIXFOUR = ftello( fp_inLINE );
3474 fseeko( fp_inLINE, 0L, SEEK_SET );
3475 #endif /* defined(_WIN32_ENVIRONMENT_)  */
3476
3477 printf( "Size of Input TEXTual file: %s\n", _ui64toaKAZEcomma(size_inLINESIXFOUR, llTOaDigits, 10) );
3478 FilesLEN = FilesLEN + size_inLINESIXFOUR;
3479 NumberOfFiles++;
3480
3481         //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3482 wrdlen = 0;
3483         for( i = 0; i < size_inLINESIXFOUR; i++ )
3484 {
3485
3486                 // ~~~~~~~~~~ Buffering fread [
3487                 if (workKoffset == -1) {
3488                         if (i + 1024*128 < size_inLINESIXFOUR) {
3489                                 fread( &workK[0], 1, 1024*128, fp_inLINE );
3490                                 workKoffset = 0;
3491                                 workbyte = workK[workKoffset];
3492                         } else
3493                         fread( &workbyte, 1, 1, fp_inLINE );
3494                 } else {
3495                         workKoffset++;
3496                         workbyte = workK[workKoffset];
3497                         if (workKoffset == 1024*128 - 1) workKoffset = -1;
3498                 }
3499                 // ~~~~~~~~~~ Buffering fread ]
3500
3501 //                if( isalpha( workbyte ) )
3502 //                {
3503 //                        if( wrdlen < 31 )
3504 //                        { wrd[ wrdlen ] = tolower( workbyte ); }
3505 //                        wrdlen++;
3506 //                }
3507
3508                 if ( workbyte < 'A' ) // Most characters are under alphabet - only one if
3509                 {
3510 ElStupido:
3511                         // This fragment is MIRRORed: #1 copy [
3512                         if (workbyte == 10) {NumberOfLines++;}
3513
3514 // Quadruple! [
3515 // Sliding window for 'wrd': The incoming string 'a lot of things must' becomes 'a_lot_of_things' and 'lot_of_things_must':
3516
3517 // ain_t_that_a
3518 // didn_t_feel_a
3519 // i_didn_t_feel
3520 // t_feel_a_thing
3521 // t_that_a_cake
3522
3523 // 316
3524 // 00:17:55,859 --> 00:17:58,447
3525 // Ain't that a cake ? I didn't feel a thing !
3526
3527                           if ( PLE_words_INITflag == 0 && ( (PLE_words != 0) || (PLE_words == 0 && wrdlen != 0) ) )
3528                           if ( workbyte == '.' || workbyte == '!' || workbyte == '?' || workbyte == ':' || workbyte == ';' || workbyte == ','
     || workbyte == '\t' ) {
3529                             PLE_words_INITflag = 1;
3530                           }
3531 // Quadruple! ]
3532
3533 //r.15fixfix [
3534                         if( wrdlen > 31 ) PLE_words_INITflag = 1;
3535 //r.15fixfix ]
3536
3537                         //if ( 1 <= wrdlen && wrdlen <= LongestLineInclusive ) // Enforce no word with length greater than 31 with below line
     to enter x-lets.
3538                         if ( 1 <= wrdlen && wrdlen <= 31 )
3539                 {
3540                                 wrd[ wrdlen ] = 0;
3541                                 // OTKACHAM: 1<<17-1 gives 65536 i.e. '-' have had high priority than '<<'
3542                                 //Next line gives error due to mix of '&' and 'double'
3543                                 if ((WORDcount & ((1<<18)-1)) == 0)
3544                                 { //_ui64toaKAZEzerocomma(WORDcount, llTOaDigits, 10);
3545                                   //printf( "Word count: %s(%lu/128 done)\r", llTOaDigits, ((long long)i*100) / size_inLINESIXFOUR );
3546 //++Melnitchka;
3547 //Melnitchka = Melnitchka % 4;
3548 //if (Melnitchka == 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 ); }
3549 //if (Melnitchka == 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 ); }
3550 //if (Melnitchka == 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 ); }
3551 //if (Melnitchka == 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 ); }
3552 Melnitchka = Melnitchka & 3; // 0 1 2 3: 00 01 10 11
```

```
                        (void) time(&t4);
                        if (t4 <= t1) {t4 = t1; t4++;}
printf( "%s; %sP/s; Phrase count: %s of them %s distinct; Done: %lu/64\r", Auberge[Melnitchka++], _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 );
                                }

//14+++ [
PLE_words++;

#ifdef singleton
                                PLE_words_INITflag = 1;
#endif
#ifdef doubleton
if (PLE_words == 1)
        strcpy( wrd1st, wrd );
else if (PLE_words == 2) {
        strcpy( wrd2nd, wrd );
        wrdlen = strlen(wrd1st)+strlen(wrd2nd)+1; // '_'
        //wrdlen = strlen(wrd);
        //if ( wrdlen <= 31 ) {
        if ( wrdlen <= LongestLineInclusive ) {
  strcpy(wrd, wrd1st);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd2nd);
        }
}
else {
        PLE_words = 2;
        strcpy( wrd1st, wrd2nd );
        strcpy( wrd2nd, wrd );
        wrdlen = strlen(wrd1st)+strlen(wrd2nd)+1; // '_'
        //wrdlen = strlen(wrd);
        //if ( wrdlen <= 31 ) {
        if ( wrdlen <= LongestLineInclusive ) {
  strcpy(wrd, wrd1st);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd2nd);
        }
}
#endif
#ifdef tripleton
if (PLE_words == 1)
        strcpy( wrd1st, wrd );
else if (PLE_words == 2)
        strcpy( wrd2nd, wrd );
else if (PLE_words == 3) {
        strcpy( wrd3rd, wrd );
        wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+1+1; // '_''_'
        //wrdlen = strlen(wrd);
        //if ( wrdlen <= 31 ) {
        if ( wrdlen <= LongestLineInclusive ) {
  strcpy(wrd, wrd1st);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd2nd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd3rd);
        }
}
else {
        PLE_words = 3;
        strcpy( wrd1st, wrd2nd );
        strcpy( wrd2nd, wrd3rd );
        strcpy( wrd3rd, wrd );
        wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+1+1; // '_''_'
        //wrdlen = strlen(wrd);
        //if ( wrdlen <= 31 ) {
        if ( wrdlen <= LongestLineInclusive ) {
  strcpy(wrd, wrd1st);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd2nd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd3rd);
        }
}
#endif
#ifdef quadrupleton
// Quadruple! [
if (PLE_words == 1)
        strcpy( wrd1st, wrd );
else if (PLE_words == 2)
        strcpy( wrd2nd, wrd );
else if (PLE_words == 3)
        strcpy( wrd3rd, wrd );
else if (PLE_words == 4) {
        strcpy( wrd4th, wrd );
        wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+1+1+1; // '_''_''_'
        //wrdlen = strlen(wrd);
```



```
        //if ( wrdlen <= 31 ) {
        if ( wrdlen <= LongestLineInclusive ) {
  strcpy(wrd, wrd1st);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd2nd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd3rd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd4th);
        }
}
else {
        PLE_words = 4;
        strcpy( wrd1st, wrd2nd );
        strcpy( wrd2nd, wrd3rd );
        strcpy( wrd3rd, wrd4th );
        strcpy( wrd4th, wrd );
        wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+1+1+1; // '_''_''_'
        //wrdlen = strlen(wrd);
        //if ( wrdlen <= 31 ) {
        if ( wrdlen <= LongestLineInclusive ) {
  strcpy(wrd, wrd1st);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd2nd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd3rd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd4th);
        }
}
// Quadruple! ]
#endif
#ifdef quintupleton
if (PLE_words == 1)
        strcpy( wrd1st, wrd );
else if (PLE_words == 2)
        strcpy( wrd2nd, wrd );
else if (PLE_words == 3)
        strcpy( wrd3rd, wrd );
else if (PLE_words == 4)
        strcpy( wrd4th, wrd );
else if (PLE_words == 5) {
        strcpy( wrd5th, wrd );
        wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+1+1+1+1; // '_''_''_''_'
        //wrdlen = strlen(wrd);
        //if ( wrdlen <= 31 ) {
        if ( wrdlen <= LongestLineInclusive ) {
  strcpy(wrd, wrd1st);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd2nd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd3rd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd4th);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd5th);
        }
}
else {
        PLE_words = 5;
        strcpy( wrd1st, wrd2nd );
        strcpy( wrd2nd, wrd3rd );
        strcpy( wrd3rd, wrd4th );
        strcpy( wrd4th, wrd5th );
        strcpy( wrd5th, wrd );
        wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+1+1+1+1; // '_''_''_''_'
        //wrdlen = strlen(wrd);
        //if ( wrdlen <= 31 ) {
        if ( wrdlen <= LongestLineInclusive ) {
  strcpy(wrd, wrd1st);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd2nd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd3rd);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd4th);
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd5th);
        }
}
#endif
#ifdef sextupleton
if (PLE_words == 1)
        strcpy( wrd1st, wrd );
else if (PLE_words == 2)
        strcpy( wrd2nd, wrd );
else if (PLE_words == 3)
        strcpy( wrd3rd, wrd );
```

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



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

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



```
3981   strcat(wrd, DelimiterUnderscore);
3982   strcat(wrd, wrd9th);
3983        }
3984 }
3985 #endif
3986 #ifdef decupleton
3987 if (PLE_words == 1)
3988        strcpy( wrd1st, wrd );
3989 else if (PLE_words == 2)
3990        strcpy( wrd2nd, wrd );
3991 else if (PLE_words == 3)
3992        strcpy( wrd3rd, wrd );
3993 else if (PLE_words == 4)
3994        strcpy( wrd4th, wrd );
3995 else if (PLE_words == 5)
3996        strcpy( wrd5th, wrd );
3997 else if (PLE_words == 6)
3998        strcpy( wrd6th, wrd );
3999 else if (PLE_words == 7)
4000        strcpy( wrd7th, wrd );
4001 else if (PLE_words == 8)
4002        strcpy( wrd8th, wrd );
4003 else if (PLE_words == 9)
4004        strcpy( wrd9th, wrd );
4005 else if (PLE_words == 10) {
4006        strcpy( wrd10th, wrd );
4007        wrdlen =
     strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+strlen(wrd8th)+strlen(wrd9th)+strlen(
     wrd10th)+1+1+1+1+1+1+1+1+1+1; // '_''_''_''_''_''_''_''_''_'
4008        //wrdlen = strlen(wrd);
4009        //if ( wrdlen <= 31 ) {
4010        if ( wrdlen <= LongestLineInclusive ) {
4011   strcpy(wrd, wrd1st);
4012   strcat(wrd, DelimiterUnderscore);
4013   strcat(wrd, wrd2nd);
4014   strcat(wrd, DelimiterUnderscore);
4015   strcat(wrd, wrd3rd);
4016   strcat(wrd, DelimiterUnderscore);
4017   strcat(wrd, wrd4th);
4018   strcat(wrd, DelimiterUnderscore);
4019   strcat(wrd, wrd5th);
4020   strcat(wrd, DelimiterUnderscore);
4021   strcat(wrd, wrd6th);
4022   strcat(wrd, DelimiterUnderscore);
4023   strcat(wrd, wrd7th);
4024   strcat(wrd, DelimiterUnderscore);
4025   strcat(wrd, wrd8th);
4026   strcat(wrd, DelimiterUnderscore);
4027   strcat(wrd, wrd9th);
4028   strcat(wrd, DelimiterUnderscore);
4029   strcat(wrd, wrd10th);
4030        }
4031 }
4032 else {
4033        PLE_words = 10;
4034        strcpy( wrd1st, wrd2nd );
4035        strcpy( wrd2nd, wrd3rd );
4036        strcpy( wrd3rd, wrd4th );
4037        strcpy( wrd4th, wrd5th );
4038        strcpy( wrd5th, wrd6th );
4039        strcpy( wrd6th, wrd7th );
4040        strcpy( wrd7th, wrd8th );
4041        strcpy( wrd8th, wrd9th );
4042        strcpy( wrd9th, wrd10th );
4043        strcpy( wrd10th, wrd );
4044        wrdlen =
     strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+strlen(wrd5th)+strlen(wrd6th)+strlen(wrd7th)+strlen(wrd8th)+strlen(wrd9th)+strlen(
     wrd10th)+1+1+1+1+1+1+1+1+1+1; // '_''_''_''_''_''_''_''_''_'
4045        //wrdlen = strlen(wrd);
4046        //if ( wrdlen <= 31 ) {
4047        if ( wrdlen <= LongestLineInclusive ) {
4048   strcpy(wrd, wrd1st);
4049   strcat(wrd, DelimiterUnderscore);
4050   strcat(wrd, wrd2nd);
4051   strcat(wrd, DelimiterUnderscore);
4052   strcat(wrd, wrd3rd);
4053   strcat(wrd, DelimiterUnderscore);
4054   strcat(wrd, wrd4th);
4055   strcat(wrd, DelimiterUnderscore);
4056   strcat(wrd, wrd5th);
4057   strcat(wrd, DelimiterUnderscore);
4058   strcat(wrd, wrd6th);
4059   strcat(wrd, DelimiterUnderscore);
4060   strcat(wrd, wrd7th);
4061   strcat(wrd, DelimiterUnderscore);
4062   strcat(wrd, wrd8th);
4063   strcat(wrd, DelimiterUnderscore);
4064   strcat(wrd, wrd9th);
```

```
  strcat(wrd, DelimiterUnderscore);
  strcat(wrd, wrd10th);
        }
}
#endif
//14+++ ]

//14+++ [
#ifdef singleton
                        if ( ( PLE_words == 1 ) && ( 1 <= wrdlen ) && ( wrdlen <= 31 ) ) {
#endif
#ifdef doubleton
                        if ( ( PLE_words == 2 ) && ( 5 <= wrdlen ) && ( wrdlen <= 41 ) ) {
#endif
#ifdef tripleton
                        if ( ( PLE_words == 3 ) && ( 9 <= wrdlen ) && ( wrdlen <= 41 ) ) {
#endif
#ifdef quadrupleton
                        if ( ( PLE_words == 4 ) && ( 13 <= wrdlen ) && ( wrdlen <= 51 ) ) {
#endif
#ifdef quintupleton
                        if ( ( PLE_words == 5 ) && ( 17 <= wrdlen ) && ( wrdlen <= 61 ) ) {
#endif
#ifdef sextupleton
                        if ( ( PLE_words == 6 ) && ( 21 <= wrdlen ) && ( wrdlen <= 71 ) ) {
#endif
#ifdef septupleton
                        if ( ( PLE_words == 7 ) && ( 25 <= wrdlen ) && ( wrdlen <= 81 ) ) {
#endif
#ifdef octupleton
                        if ( ( PLE_words == 8 ) && ( 29 <= wrdlen ) && ( wrdlen <= 91 ) ) {
#endif
#ifdef nonupleton
                        if ( ( PLE_words == 9 ) && ( 33 <= wrdlen ) && ( wrdlen <= 101 ) ) {
#endif
#ifdef decupleton
                        if ( ( PLE_words == 10 ) && ( 37 <= wrdlen ) && ( wrdlen <= 111 ) ) {
#endif
//14+++ ]
WORDcount++;
if (BSTorBtree < 2) {

  LetterOffset = (int)( wrd[0] - 'a' ) * 31 + (wrdlen-1); // 0..805
  //BufStart = pointerflush + LetterOffset * LetterBuffer; // OLD

  BufStart = pointerflush + (int)( wrd[0] - 'a' ) * WHOLEletter_BufferSize + OffsetsInBuffer[wrdlen-1];
  // Above line and Below line are equal
  //BufStart = pointerflush + (LetterOffset / 31) * WHOLEletter_BufferSize + OffsetsInBuffer[LetterOffset % 31];

              //Slot = KuxHash3plus(wrd)<<2; //13++
              //Slot = FNV1A_Hash_SHIFTless_XORless(wrd)<<2; //13+++
              //Slot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13++++
              //Slot = FNV1A_Hash_4_OCTETS_31(wrd, wrdlen>>2)<<2; //13+++++
/*
if (wrdlen<=19) // 4x4+3=19
              Slot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13++++
else            // 2x8+4=20 i.e. first contains 5 clashes
              Slot = FNV1A_Hash_8_OCTETS(wrd, wrdlen>>3)<<2; //13+++++
*/
//if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
//              Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>2, 2)<<2; //13+++++
//else            // 2x8+4=20 i.e. first contains 6 clashes
//              Slot = FNV1A_Hash_Granularity(wrd, wrdlen>>3, 3)<<2; //13+++++

              //Slot = FNV1A_Hash_8_OCTETS(wrd, wrdlen>>3)<<2; //13_7p
              //Slot = HashFNV1A_unrolled_Final(wrd, wrdlen)<<2; //13_7p
       //Slot = HashAlfalfa_HALF(wrd, wrdlen)<<2; //13_7p
       //Slot = Hash_Alfalfa(wrd, wrdlen)<<2; //13_7p
//        Slot = Sixtinsensitive(wrd, wrdlen)<<2; //13_7p
       Slot = FNV1A_Hash_Jesteress(wrd, wrdlen)<<2; //13_7p
//        Slot = FNV1A_Hash_Jester(wrd, wrdlen)<<2; //13_7p

/*
hashAlfalfa = 7;
for(iAlfalfa = 0; iAlfalfa < (wrdlen & -2); iAlfalfa += 2) {
    hashAlfalfa = (17+9) * ((17+9) * hashAlfalfa + (wrd[iAlfalfa])) + (wrd[iAlfalfa+1]);
}
if(wrdlen & 1)
    hashAlfalfa = (17+9) * hashAlfalfa + (wrd[wrdlen-1]);

              Slot = (( hashAlfalfa ^ (hashAlfalfa >> 16) ) & 8191)<<2; //13_7p
*/

                   memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
//; Line 917
//  mov     edx, DWORD PTR [eax+ebp]
//  add     esp, 4
                   // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
```



```
                   //PseudoLinkedPointer = (unsigned long)*(long *)(BufStart+Slot);
//; Line 919
//  mov     edx, DWORD PTR [eax+ebp]
//  add     esp, 4
                   //       while (count--) {
                   //       *(char *)dst = *(char *)src;
                   //       dst = (char *)dst + 1;
                   //       src = (char *)src + 1;
                   //       }

if (BSTorBtree == 0)
{
// @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ BST fragment [
                   if (PseudoLinkedPointer == 0) // means EMPTY-SLOT
                   {
           //if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 < (GRMBLhill[(int)wrdlen] * LetterBuffer)/31 ) // OLD slower
           if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST instead of LL
                     {
                     memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
//; Line 932
//  mov     DWORD PTR [eax+ebp], esi
                     // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
                     //*(long *)(BufStart+Slot) = *(long *)&bufend[LetterOffset];
//; Line 936
//  mov     DWORD PTR [eax+ebp], esi

                     // Below 3 lines are commented due to experiment below malloc which shows that allocated memory is ZEROed.
                     //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTs*4], 4 ); // means next exists not: Means PseudoLinkedPointerL = 0
//; Line 940
//  mov     ecx, DWORD PTR [ebp+32768]
                     // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
                     //*(long *)bufend[LetterOffset] = *(long *)&BufStart[NumberOfSLOTs*4];
//; Line 944
//  mov     ecx, DWORD PTR [ebp+32768]
                     //bufend[LetterOffset] = bufend[LetterOffset] + 4;
                     //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTs*4], 4 ); // means next exists not: Means PseudoLinkedPointerR = 0
                     bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4; // + 4 due to above commenting
                     memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
                              //bufNowps[LetterOffset][Slot]++; // ?! crashes
                     bufend[LetterOffset] = bufend[LetterOffset] + wrdlen;
                     if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned long)(bufend[LetterOffset] - BufStart);}
                     }
           else
           { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, llTOaDigits, 10) );
fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llTOaDigits2, 10) );
fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10) );
for( k = 1; k < 32; k++ )
{ fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%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
}
fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
              fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
              return( 1 );
           }
                   }
                   else // means USED-SLOT
                   { FoundInLinkedList = 0;
                     while (PseudoLinkedPointer != 0 && FoundInLinkedList == 0)
                     {
                       if (memcmp(PseudoLinkedPointer+4+4,wrd,wrdlen) == 0)
                   //       while ( --count && *(char *)buf1 == *(char *)buf2 ) {
                   //       buf1 = (char *)buf1 + 1;
                   //       buf2 = (char *)buf2 + 1;
                   //       }
                   //       return( *((unsigned char *)buf1) - *((unsigned char *)buf2) );
                       { FoundInLinkedList = 1;
                       }
                       else // i.e < or >
                       {
                         if (memcmp(PseudoLinkedPointer+4+4,wrd,wrdlen) > 0)
                         memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
                         else
                         {
                         PseudoLinkedPointer = PseudoLinkedPointer + 4;
                         memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
```

```
                        }

                        if (PseudoLinkedPointerNEW == 0)
                  {
           //if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 < (GRMBLhill[(int)wrdlen] * LetterBuffer)/31 ) // OLD slower
           if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] )
                        { memcpy( PseudoLinkedPointer, &bufend[LetterOffset], 4 );
                          // Below 3 lines are commented due to experiment below malloc which shows that allocated memory is ZEROed.
                          //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTs*4], 4 ); // means next exists not
                          //bufend[LetterOffset] = bufend[LetterOffset] + 4;
                          //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTs*4], 4 ); // means next exists not
                          bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4; // + 4 due to above commenting
                          memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
                              //bufNoWpS[LetterOffset][Slot]++; // ?! crashes
                           bufend[LetterOffset] = bufend[LetterOffset] + wrdlen;
                          if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
                        }
           else
           { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, llTOaDigits, 10) );
fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, llTOaDigits2, 10) );
fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits,
10) );
for( k = 1; k < 32; k++ )
{ fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%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
}
fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
             fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
             return( 1 );
           }
                  }
                        PseudoLinkedPointer = PseudoLinkedPointerNEW;
                      }
                      WORDcountAttemptsToPut++;
                    } // while
                  }
// @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ BST fragment ]
} else
{
// ####################################### B-tree order 3 fragment [
//
// LEAF structure: [LeftPointer][MiddlePointer][RightPointer][LeftWord][RightWord]
//                 4bytes       4bytes         4bytes        wrdlen     wrdlen
//                                                           *          *          <- if *(char *)==0 means the word cell is empty
// ALL B-tree order 3 fragment consists of 3 sub-fragments:
// 1] Search 2] if Search failed Trasirascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search 3] Insert Iterative

// 1] Search [ _______1407 line in C - see below: whole Search in assembler_______
                  if (PseudoLinkedPointer == 0) // means EMPTY-SLOT
                  {
           if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST
instead of LL; + more(see LEAF)
                    {
                    memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
                    bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
                    memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
                    bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
                    if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
                    }
           else
           { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, llTOaDigits, 10) );
fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, llTOaDigits2, 10) );
fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10)
);
for( k = 1; k < 32; k++ )
{ fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%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
}
fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
             fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
```



```
             return( 1 );
           }
                    FoundInLinkedList = 1;
                  }
                  else // means USED-SLOT
                  { FoundInLinkedList = 0;
                    while (PseudoLinkedPointer != 0 && FoundInLinkedList == 0)
                    {
// ***** 'P W P' section [
// LW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4) != 0 )
// RW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
                      // here ALWAYS LW exists: no need for existence check - line below
                      // if ( *(char *)(PseudoLinkedPointer+4+4+4) != 0 )
                      if (memcmp(PseudoLinkedPointer+4+4+4,wrd,wrdlen) > 0) // go LP
                        { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 0, 4 ); //LP
                          PseudoLinkedPointer = PseudoLinkedPointerNEW;
                        }
                      else if (memcmp(PseudoLinkedPointer+4+4+4,wrd,wrdlen) < 0) // go RP or MP
                        { // RW existence check - line below:
                          if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // RW exists
                            { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// ***** 'P W P' section 2 [
// LW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4) != 0 )
// RW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
                      // here ALWAYS RW exists: no need for existence check - line below
                      // if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
                      if (memcmp(PseudoLinkedPointer+4+4+4+wrdlen,wrd,wrdlen) > 0) // go MP
                        { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
                          PseudoLinkedPointer = PseudoLinkedPointerNEW;
                        }
                      else if (memcmp(PseudoLinkedPointer+4+4+4+wrdlen,wrd,wrdlen) < 0) // go RP
                        { // No ?W after RW - go RP
                          memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
                          PseudoLinkedPointer = PseudoLinkedPointerNEW;
                        }
                      else FoundInLinkedList = 1; // wrd is RW
                      WORDcountAttemptsToPut++;
// ***** 'P W P' section 2 ]
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                            }
                          else // RW empty - go MP
                            { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
                              PseudoLinkedPointer = PseudoLinkedPointerNEW;
                            }
                        }
                      else FoundInLinkedList = 1; // wrd is LW
                      WORDcountAttemptsToPut++;
// ***** 'P W P' section ]
                    } // while
                    WORDcountAttemptsToPut--; // - 1 due to BST way of counting i.e. direct hash hit is not counted only successors
                  }
// 1] Search ] _______1484 line in C - see below: whole Search in assembler_______

/*
; Line 1397
jmp $L2139
$L2042:
; Line 1408
test edx, edx
jne SHORT $L2110
; Line 1410
mov ecx, DWORD PTR _bufend$[esp+esi*4+892340]
mov edi, DWORD PTR _GRMBLFoolAgain$[esp+ebx*4+892340]
lea edx, DWORD PTR _bufend$[esp+esi*4+892340]
lea esi, DWORD PTR [ebx+ebx+12]
sub esi, ebp
add esi, ecx
cmp esi, edi
mov DWORD PTR tv4122[esp+892340], edx
jae $L2113
; Line 1412
mov DWORD PTR [eax+ebp], ecx
; Line 1413
lea eax, DWORD PTR [ecx+12]
; Line 1436
jmp $L2749
$L2110:
; Line 1437
mov DWORD PTR _FoundInLinkedList$[esp+892340], 0
npad 11
$L2141:
; Line 1438
mov eax, DWORD PTR _FoundInLinkedList$[esp+892340]
test eax, eax
jne $L2142
; Line 1445
lea ebp, DWORD PTR [edx+12]
```

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



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

```
//                                                                          | |  'PseudoLinkedPointerNEW' (ptr to NEW LEAF)
//    There are total 2 situations for PARENT LEAF: <-------------------------------  ARE GOING UP
//    Case #3: [LP][MP][  ][LW][  ]
//    Case #4: [LP][MP][RP][LW][RW]  Split Occurs

// ~  First deal alonely with the OUTER NODE(LEAF) where Search stopped i.e Case #1 & Case #2:
        POffsetInLEAF = BSTstack[--StackPtr];
        PseudoLinkedPointer = BSTstack[--StackPtr];
// NOTE: ONE LEAF IS FULL ONLY WHEN LAST CELL FOR KEY(here RW) EXISTS!
// RW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // If LEAF is full: Case #2
    { SplitOccured = 1; WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
            // ALlocate NEW LEAF:
            if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST instead of LL; + more(see LEAF)
                    {
                    memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
                    bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
                    bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
                    if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned long)(bufend[LetterOffset] - BufStart);}
                    }
            else
            { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, llTOaDigits, 10) );
fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long)WORDcountDistinct, llTOaDigits2, 10) );
fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10) );
for( k = 1; k < 32; k++ )
{ fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%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
}
fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
            fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
            return( 1 );
            }
    if (POffsetInLEAF == 0) // wrd < LW
        {
                    memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
                    memcpy( PseudoLinkedPointer+4+4+4, wrd, wrdlen );   // wrd go to OLD LEAF
                    memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
                    *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) = 0;    // RW mark unused in OLD LEAF
        }
    if (POffsetInLEAF == 4) // LW < wrd < RW
        {
                    memcpy( wrdUP, wrd, wrdlen );                    // wrd up
                    memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
                    *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) = 0;    // RW mark unused in OLD LEAF
        }
    if (POffsetInLEAF == 8) // wrd > RW
        {
                    memcpy( wrdUP, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW up
                    *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) = 0;    // RW mark unused in OLD LEAF
                    memcpy( PseudoLinkedPointerNEW+4+4+4, wrd, wrdlen );   // wrd go to NEW LEAF
        }
    }
else // If LEAF is not full: Case #1
    { SplitOccured = 0; WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
    if (POffsetInLEAF == 0) // wrd < [LW][] so [LW][] -> [][LW] -> [wrd][LW]
        {
                    memcpy( PseudoLinkedPointer+4+4+4+wrdlen, PseudoLinkedPointer+4+4+4, wrdlen );
                    memcpy( PseudoLinkedPointer+4+4+4, wrd, wrdlen );
        }
    if (POffsetInLEAF == 4) // wrd > [LW][] so [LW][] -> [LW][wrd]
        {
                    memcpy( PseudoLinkedPointer+4+4+4+wrdlen, wrd, wrdlen );
        }
    }

if (SplitOccured != 0)
{
// ~  Second deal with the INNER NODE(S) i.e Case #3 & Case #4:
        while (StackPtr != 0 || SplitOccured != 0)
              {
        // 'PseudoLinkedPointerNEW' is new LEAF to be inserted
        // 'wrdUP' is NEW word to be inserted
        if (StackPtr != 0)
        {
          POffsetInLEAF = BSTstack[--StackPtr];
          PseudoLinkedPointer = BSTstack[--StackPtr];
if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // If LEAF is full: Case #4
```



```
    { SplitOccured = 1;
                    memcpy( wrdUPold, wrdUP, wrdlen ); // LW up
                    PseudoLinkedPointerNEWold = PseudoLinkedPointerNEW;
            // ALlocate NEW LEAF:
            if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST instead of LL; + more(see LEAF)
                    {
                    memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
                    bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
                    bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
                    if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned long)(bufend[LetterOffset] - BufStart);}
                    }
            else
            { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, llTOaDigits, 10) );
fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long)WORDcountDistinct, llTOaDigits2, 10) );
fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10) );
for( k = 1; k < 32; k++ )
{ fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%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
}
fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
            fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
            return( 1 );
            }
    if (POffsetInLEAF == 0) // wrdUPold < LW
        {
                    memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
                    memcpy( PseudoLinkedPointer+4+4+4, wrdUPold, wrdlen );   // wrdUPold go to OLD LEAF
                    memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
                    *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) = 0;    // RW mark unused in OLD LEAF
                    // [LP](PseudoLinkedPointerNEWold)[MP][RP](wrdUPold)[LW][RW]       ------
                    //               pair [LW] PseudoLinkedPointerNEW goes up                 |
                    // PseudoLinkedPointer:                         PseudoLinkedPointerNEW: |
                    // [LP](PseudoLinkedPointerNEWold)[](wrdUPold)  [MP][RP][][RW]      <----
                    // no need to put zero in RP because logic is based on words existence:
                    memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+4, 4 );
                    memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
                    memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEWold, 4 );
        }
    if (POffsetInLEAF == 4) // LW < wrdUPold < RW
        {
                    memcpy( wrdUP, wrdUPold, wrdlen );                     // wrdUPold up
                    memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
                    *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) = 0;    // RW mark unused in OLD LEAF
                    // [LP][MP](PseudoLinkedPointerNEWold)[RP][LW](wrdUPold)[RW]       ------
                    //               pair [wrdUPold] PseudoLinkedPointerNEW goes up           |
                    // PseudoLinkedPointer:  PseudoLinkedPointerNEW:                          |
                    // [LP][MP][][LW]        (PseudoLinkedPointerNEWold)[RP][][RW]      <----
                    // no need to put zero in RP because logic is based on words existence:
                    memcpy( PseudoLinkedPointerNEW+0, &PseudoLinkedPointerNEWold, 4 );
                    memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
        }
    if (POffsetInLEAF == 8) // wrdUPold > RW
        {
                    memcpy( wrdUP, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW up
                    *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) = 0;    // RW mark unused in OLD LEAF
                    memcpy( PseudoLinkedPointerNEW+4+4+4, wrdUPold, wrdlen );   // wrdUPold go to NEW LEAF
                    // [LP][MP][RP](PseudoLinkedPointerNEWold)[LW][RW](wrdUPold)       ------
                    //               pair [RW] PseudoLinkedPointerNEW goes up                 |
                    // PseudoLinkedPointer:  PseudoLinkedPointerNEW:                          |
                    // [LP][MP][][LW]        [RP](PseudoLinkedPointerNEWold)[](wrdUPold) <---
                    // no need to put zero in RP because logic is based on words existence:
                    memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+8, 4 );
                    memcpy( PseudoLinkedPointerNEW+4, &PseudoLinkedPointerNEWold, 4 );
        }
    }
else // If LEAF is not full: Case #3
    { SplitOccured = 0;
    if (POffsetInLEAF == 0) // wrdUP < [LW][] so [LW][] -> [][LW] -> [wrdUP][LW]
        {
                    memcpy( PseudoLinkedPointer+4+4+4+wrdlen, PseudoLinkedPointer+4+4+4, wrdlen );
                    memcpy( PseudoLinkedPointer+4+4+4, wrdUP, wrdlen );
                          // [LP][MP][] -> [LP][][MP] -> [LP][np][MP]
                    memcpy( PseudoLinkedPointer+8, PseudoLinkedPointer+4, 4 );
                    memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEW, 4 );
        }
    if (POffsetInLEAF == 4) // wrdUP > [LW][] so [LW][] -> [LW][wrdUP]
        {
```

```
                    memcpy( PseudoLinkedPointer+4+4+4+wrdlen, wrdUP, wrdlen );
                            // [LP][MP][] -> [LP][MP][np]
                    memcpy( PseudoLinkedPointer+8, &PseudoLinkedPointerNEW, 4 );
         }
           break;
    }
        }
        else // Empty stack means ROOT and more over ROOT is already splitted(Case #4 is off)
        {
     // If LEAF is not full: Case #3
     // THIS IS WHERE A NEW(SECOND) LEAF 'PseudoLinkedPointerROOT' must be allocated:
            if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST instead of LL; + more(see LEAF)
                    {
                    memcpy( &PseudoLinkedPointerROOT, &bufend[LetterOffset], 4 );
                    bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
                    memcpy( bufend[LetterOffset], wrdUP, wrdlen );
                    bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
                    if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned long)(bufend[LetterOffset] - BufStart);}
                    }
            else
            { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, llTOaDigits, 10) );
fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llTOaDigits2, 10) );
fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10) );
for( k = 1; k < 32; k++ )
{ fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%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
}
fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
              fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
              return( 1 );
            }
            // Here                    --  'PseudoLinkedPointerROOT' --
            //                         |           (wrdUP)           |
            // 'PseudoLinkedPointer' <--                             --> 'PseudoLinkedPointerNEW'
            //          (LW)                                                   (RW)
            memcpy( PseudoLinkedPointerROOT, &PseudoLinkedPointer, 4 );      // LP
            memcpy( PseudoLinkedPointerROOT+4, &PseudoLinkedPointerNEW, 4 ); // MP
            // Here must NEW ROOT be updated i.e. HASH table(SLOT) must point it:
            memcpy( BufStart+Slot, &PseudoLinkedPointerROOT, 4 );
           break; //because it is ROOT without split
        }
              } // while
} //if (SplitOccured != 0)
// 3] Insert Iterative ]
} //if (FoundInLinkedList == 0)
// ######################################## B-tree order 3 ]
} //if (BSTorBtree == 0)
} else { //if (BSTorBtree != 2) {
// External Btrees [

// _ ASCII code 095
// ` ASCII code 096  \
// a ASCII code 097  / In total 26+1+1 radix instead of 27 to avoid +1 for each '_', code 096 not used.
// z ASCII code 122
// The hash  for 'a_quadruplet_for_example' will be calculated for first 5 chars:
// = (byte1-'_')*28*28*28*28 + (byte2-'_')*28*28*28 + (byte3-'_')*28*28 + (byte4-'_')*28 + (byte5-'_')
// Hash slots are 28*28*28*28*28 = 17,210,368 each containing one 64bit pointer i.e. 8bytes in length.
// Hash size = 17,210,368*8 = 137,682,944 bytes
// When at end all these slots(17,210,368- Btrees) are traversed the outcome is a sorted wordlist - no need of sorting.

// D:\_KAZE_new-stuff\Leprechaun_quadrupleton_r14_minus>Leprechaun_quadrupleton.exe GRAFFITH_2048.lst GRAFFITH_2048.wrd z
// Leprechaun(Fast Greedy Word-Ripper), rev. 14_minus_quadrupleton, written by Svalqyatchx.
// Leprechaun: 'Oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
// Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
//       also the performance of a 3-way hash + 6,602,752 B-Trees of order 3,
//       also the performance of a 1-way hash + 17,210,368 external B-Trees of order 3.
// Size of input file with files for Leprechauning: 42140
// Allocating HASH memory 137,683,009 bytes ... OK
// Allocating/ZEROing 1,047,566 bytes swap file ... OK
// Size of Input TEXTual file: 33,470,581
// |; Word count: 3,045,077 of them 0 distinct; Done: 64/64
// ...
// Size of Input TEXTual file: 17,403,406
// /; Word count: 2,710,601,882 of them 0 distinct; Done: 64/64
// Bytes per second performance: 17,694,246B/s
// Words per second performance: 1,730,907W/s
// Leprechaun: Done.
```



```
//
// Leprechaun report:
// Number Of Trees(GREATER THE BETTER): 1,646,004

// TO DO - it is long overdue: at last make sort stage at end unnecessary - only traversing-and-dumping!

BufStart = pointerflush;
//   Slot = ((wrd[0]-'_')*28*28*28*28 + (wrd[1]-'_')*28*28*28 + (wrd[2]-'_')*28*28 + (wrd[3]-'_')*28 + (wrd[4]-'_'))<<3;
//   Slot = FNV1A_Hash_Jesteress_27bit(wrd, wrdlen)<<3; // Commented since r.14++++ because of passes.
Slot = FNV1A_Hash_Jesteress_27bit(wrd, wrdlen);

// Bug fix for all r.14+++ and below! [
memcpy( &wrd[(LongestLineInclusive+1+4)-4], &NULLsForWRD, 4 );
// Bug fix for all r.14+++ and below! ]

// Example: HashInBITS-HashChunkSizeInBITS=2
//          HashInBITS = 5
//          HashChunkSizeInBITS = 3
//          RipPasses = 1<<(HashInBITS-HashChunkSizeInBITS) i.e. 1<<2 which is 4 i.e. 32 slots with 4 passes 8 slots each.
//          00??? 5bits 0-7
//          01??? 5bits 8-15
//          10??? 5bits 16-23
//          11??? 5bits 24-31
if ( (Slot>>HashChunkSizeInBITS) == RipPasses ) {
Slot = Slot<<3;

//Slot = 0; // One Tree only!
memcpy( &PseudoLinkedPointer_64, BufStart+Slot, 8 );

// ######################################## B-tree order 3 fragment 64bit [
//
// LEAF structure: [LeftPointer][MiddlePointer][RightPointer][LeftWord][RightWord]
//                  4bytes       4bytes        4bytes        wrdlen    wrdlen
//                                                           *         *         <- if *(char *)==0 means the word cell is empty
// ALL B-tree order 3 fragment consists of 3 sub-fragments:
// 1] Search 2] if Search failed Trasirascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search 3] Insert Iterative

// LEAF_64 structure: [LeftPointer][MiddlePointer][RightPointer][LeftWord]              [RightWord]
//                     8bytes       8bytes         8bytes        LongestLineInclusive+1+4 LongestLineInclusive+1+4
//                                                               *                      *         <- if *(char *)==0 means the word cell is empty
// 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 choice ...
// Note: BufEnd_64 in fact is the first free position after the BUFFER END!

// 1] Search [
                  if (PseudoLinkedPointer_64 == 0) // means EMPTY-SLOT
                  {
if ( REUSE != 2 ) { //  REUSE [ // This line comes since r16
if (DoNotInsertFlag == 0)
{ // This line comes since r15FIXFIX+ [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
            //if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST instead of LL; + more(see LEAF)
            //          {
            //          memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
            //          bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
            //          memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
            //          bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
            //          if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned long)(bufend[LetterOffset] - BufStart);}
            //          }
            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)
                    {
                        memcpy( BufStart+Slot, &BufEnd_64, 8 );
              BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
                     if (BSTorBtree == 2) {
              fsetpos(fp_outRG, &BufEnd_64);
              fwrite(wrd, wrdlen, 1, fp_outRG); WORDcountDistinct++;
              //fwrite(&OneChar_ieByte, 1, 1, fp_outRG); // Write ZERO ASCII code
              // r.14++ The above line was commented because the pool is already ZEROed.
                     } else { // ######### 64bit memory manipulations [
              memcpy( (char *)BufEnd_64, wrd, wrdlen ); WORDcountDistinct++;
                     } // ######### 64bit memory manipulations ]
              BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
              //fsetpos(fp_outRG, &BufEnd_64);
                    }
            else
            { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llTOaDigits2, 10) );
fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, llTOaDigits, 10) );
fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10) );
fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
              fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
              return( 1 );
            }
                      FoundInLinkedList = 1;
```

```
} // This line comes since r15FIXFIX+ ]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]
}
                    }
                    else // means USED-SLOT
                    { FoundInLinkedList = 0;
                StackPtr = 0;
//                      while (PseudoLinkedPointer != 0 && FoundInLinkedList == 0)
                      while (PseudoLinkedPointer_64 != 0 && FoundInLinkedList == 0)
                      {
// ***** 'P W P' section [
// LW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4) != 0 )
// RW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
                        // here ALWAYS LW exists: no need for existence check - line below
                        // if ( *(char *)(PseudoLinkedPointer+4+4+4) != 0 )
//                        if (memcmp(PseudoLinkedPointer+4+4+4,wrd,wrdlen) > 0) // go LP
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              //fread(&FourGramL[0], (LongestLineInclusive+1+4), 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
                        if (BSTorBtree == 2) {
              fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              fread(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+8+2*(LongestLineInclusive+1+4) );
                        } // ########## 64bit memory manipulations ]
              memcpy( &FourGramL[0], &LEAF[8 + 8 + 8], (LongestLineInclusive+1+4) );
              // ]  //r.14+
                        if (strcmpKAZE13(FourGramL, wrd) > 0) // go LP
//                          { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 0, 4 ); //LP
//                            PseudoLinkedPointer = PseudoLinkedPointerNEW;
//                          }
                          {
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 0; //LP
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
                if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
                BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
                BSTstack[StackPtr] = 0; ++StackPtr; //LPoffset=0;MPoffset=8;RPoffset=16;
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &PseudoLinkedPointer_64, &LEAF[0], 8 );
              // ]  //r.14+
                          }
//                        else if (memcmp(PseudoLinkedPointer+4+4+4,wrd,wrdlen) < 0) // go RP or MP
                        else if (strcmpKAZE13(FourGramL, wrd) < 0) // go RP or MP
                          { // RW existence check - line below:
//                            if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // RW exists
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4); //RW
              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              //fread(&SomeByte, 1, 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
              // ]  //r.14+
              if (SomeByte != 0 ) // RW exists
                              { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// ***** 'P W P' section 2 [
// LW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4) != 0 )
// RW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
                        // here ALWAYS RW exists: no need for existence check - line below
                        // if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
//                        if (memcmp(PseudoLinkedPointer+4+4+4+wrdlen,wrd,wrdlen) > 0) // go MP
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              //fread(&FourGramL[0], (LongestLineInclusive+1+4), 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &FourGramL[0], &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], (LongestLineInclusive+1+4) );
              // ]  //r.14+
                        if (strcmpKAZE13(FourGramL, wrd) > 0) // go MP
//                          { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
//                            PseudoLinkedPointer = PseudoLinkedPointerNEW;
//                          }
                          {
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8; //MP
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
                if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
                BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
```



```
                BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0;MPoffset=8;RPoffset=16;
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &PseudoLinkedPointer_64, &LEAF[8], 8 );
              // ]  //r.14+
                          }
//                          else if (memcmp(PseudoLinkedPointer+4+4+4+wrdlen,wrd,wrdlen) < 0) // go RP
                        else if (strcmpKAZE13(FourGramL, wrd) < 0) // go RP
//                          { // No ?W after RW - go RP
//                            memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
//                            PseudoLinkedPointer = PseudoLinkedPointerNEW;
//                          }
                          {
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8; //RP
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
                if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
                BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
                BSTstack[StackPtr] = 16; ++StackPtr; //LPoffset=0;MPoffset=8;RPoffset=16;
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &PseudoLinkedPointer_64, &LEAF[8 + 8], 8 );
              // ]  //r.14+
                          }
                          else { FoundInLinkedList = 1; // wrd is RW
              // Counter [
                        if (BSTorBtree == 2) {
              fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
                        }
              memcpy( &CounterOccurrencies, &FourGramL[(LongestLineInclusive+1+4)-4], 4 );

                        // r16 [
                        if ( REUSE == 2 ) {
                        if (*argv[k_FIX] == 'W')
                                fprintf(fp_out, "%s\t%s\r\n", _ui64toaKAZEzerocomma(CounterOccurrencies+1, llTOaDigits2, 10)+(26-9), wrd);
//WORDcountBOTTOM++;
                        if (*argv[k_FIX] == 'w')
                                fprintf(fp_out, "%s\r\n", wrd); //WORDcountBOTTOM++;
                        }
                        // r16 ]

                        if ( REUSE != 2 ) {  // r16
              if (CounterOccurrencies<9999999) CounterOccurrencies++;
              memcpy( &FourGramL[(LongestLineInclusive+1+4)-4], &CounterOccurrencies, 4 );
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //fwrite(&FourGramL[0], (LongestLineInclusive+1+4), 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &FourGramL[0], (LongestLineInclusive+1+4) );
                        if (BSTorBtree == 2) {
              fwrite(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4) );
                        } // ########## 64bit memory manipulations ]
              // ]  //r.14+
                        } // r16
              // Counter ]
                  }
                        WORDcountAttemptsToPut++;
// ***** 'P W P' section 2 ]
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              }
                              else // RW empty - go MP
//                              { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
//                                PseudoLinkedPointer = PseudoLinkedPointerNEW;
//                              }
                              {
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8; //MP
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
                if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
                BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //pt to visited leaf
                BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0;MPoffset=8;RPoffset=16;
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              //fread(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &PseudoLinkedPointer_64, &LEAF[8], 8 );
              // ]  //r.14+
                              }
                          }
```

```
5071                        else { FoundInLinkedList = 1; // wrd is LW
5072                // Counter [
5073                        if (BSTorBtree == 2) {
5074                fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5075                        }
5076                memcpy( &CounterOccurrencies, &FourGramL[(LongestLineInclusive+1+4)-4], 4 );
5077
5078                        // r16 [
5079                        if ( REUSE == 2 ) {
5080                        if (*argv[k_FIX] == 'W')
5081                                fprintf(fp_out, "%s\t%s\r\n", _ui64toaKAZEzerocomma(CounterOccurrencies+1, llTOaDigits2, 10)+(26-9), wrd);
    //WORDcountBOTTOM++;
5082                        if (*argv[k_FIX] == 'W')
5083                                fprintf(fp_out, "%s\r\n", wrd); //WORDcountBOTTOM++;
5084                        }
5085                        // r16 ]
5086
5087                        if ( REUSE != 2 ) {  // r16
5088                if (CounterOccurrencies<9999999) CounterOccurrencies++;
5089                memcpy( &FourGramL[(LongestLineInclusive+1+4)-4], &CounterOccurrencies, 4 );
5090                // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5091                //fwrite(&FourGramL[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5092                // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5093                // [  //r.14+
5094                memcpy( &LEAF[8 + 8 + 8], &FourGramL[0], (LongestLineInclusive+1+4) );
5095                        if (BSTorBtree == 2) {
5096                fwrite(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5097                        } else { // ######### 64bit memory manipulations [
5098                memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4) );
5099                        } // ######### 64bit memory manipulations ]
5100                // ]  //r.14+
5101                        } // r16
5102                // Counter ]
5103                  }
5104                        WORDcountAttemptsToPut++;
5105 // ***** 'P W P' section ]
5106                      } // while
5107                        WORDcountAttemptsToPut--; // - 1 due to BST way of counting i.e. direct hash hit is not counted only successors
5108                    }
5109 // 1] Search ]
5110
5111 if ( REUSE != 2 ) { //  REUSE [ // This line comes since r16
5112 if (DoNotInsertFlag == 0) { // This line comes since r15FIXFIX+
5113 if (FoundInLinkedList == 0)
5114 {
5115 /*
5116 // ==================== [ The whole section/sub-fragment 2 is commented due to great time differences for Internal_vs_External memory
    accesses - it is far more cheap to have the STACK overhead (moved to sub-fragment 1) ] ==================== [
5117 // 2] if Search failed Trasirascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search [
5118 //    'TracingSearch' is the same as 'Search' except that adds the trail in my simulated stack,
5119 //    the goal is not to waste time in 'Search' by dealing with no needed trail in case of not 'Insert'.
5120 //    Simulated stack contains pairs of 'Address of ParentLEAF' + 'Offset of ParentPointer in ParentLEAF i.e. 0 for LP, 4 for MP, 8 for RP'.
5121 //    'Offset ...' saves unnecessary comparisons of NEWword which after splitting goes up.
5122                    memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
5123                StackPtr = 0;
5124                    while (PseudoLinkedPointer != 0)
5125                    {
5126 // ***** 'P W P' section [
5127 // LW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4) != 0 )
5128 // RW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
5129                        // here ALWAYS LW exists: no need for existence check - line below
5130                        // if ( *(char *)(PseudoLinkedPointer+4+4+4) != 0 )
5131                        if (memcmp(PseudoLinkedPointer+4+4+4,wrd,wrdlen) > 0) // go LP
5132                          { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 0, 4 ); //LP
5133                if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5134                BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5135                BSTstack[StackPtr] = 0; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
5136                            PseudoLinkedPointer = PseudoLinkedPointerNEW;
5137                          }
5138                        else if (memcmp(PseudoLinkedPointer+4+4+4,wrd,wrdlen) < 0) // go RP or MP
5139                          { // RW existence check - line below:
5140                            if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // RW exists
5141                              { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
5142 // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
5143 // ***** 'P W P' section 2 [
5144 // LW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4) != 0 )
5145 // RW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
5146                        // here ALWAYS RW exists: no need for existence check - line below
5147                        // if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
5148                        if (memcmp(PseudoLinkedPointer+4+4+4+wrdlen,wrd,wrdlen) > 0) // go MP
5149                          { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
5150                if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5151                BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5152                BSTstack[StackPtr] = 4; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
5153                            PseudoLinkedPointer = PseudoLinkedPointerNEW;
5154                          }
5155                        else if (memcmp(PseudoLinkedPointer+4+4+4+wrdlen,wrd,wrdlen) < 0) // go RP
5156                          { // No ?W after RW - go RP
```



```
5157                            memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
5158                if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5159                BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5160                BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
5161                            PseudoLinkedPointer = PseudoLinkedPointerNEW;
5162                          }
5163                        else FoundInLinkedList = 1; // wrd is RW
5164 // ***** 'P W P' section 2 ]
5165 // ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
5166                              }
5167                            else // RW empty - go MP
5168                               { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
5169                if (StackPtr > 8192*3-1-1) { printf( "\nLeprechaun: Failure! 'B-tree order 3' simulated stack overflow!\n" ); return( 13 );}
5170                BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
5171                BSTstack[StackPtr] = 4; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
5172                                  PseudoLinkedPointer = PseudoLinkedPointerNEW;
5173                               }
5174                          }
5175                        else FoundInLinkedList = 1; // wrd is LW
5176 // ***** 'P W P' section ]
5177                      } // while
5178 // 2] if Search failed Trasirascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search ]
5179 // ==================== [ The whole section/sub-fragment 2 is commented due to great time differences for Internal_vs_External memory
    accesses - it is far more cheap to have the STACK overhead (moved to sub-fragment 1) ] ==================== ]
5180 */
5181
5182 // 3] Insert Iterative [
5183 //    There are total 4 situations:
5184 //    Case #1: Outer NODE(including ROOT) [  ][  ][  ][LW][  ]
5185 //    Case #2: Outer NODE(including ROOT) [  ][  ][  ][LW][RW]  Split Occurs   -----
5186 //    Case #3: ROOT                       [LP][MP][  ][LW][  ]                  |  'wrdUP' (wrdlen bytes)
5187 //    Case #4: Inner NODE(including ROOT) [LP][MP][RP][LW][RW]  Split Occurs   --- |   &
5188 //                                                                              | |  'PseudoLinkedPointerNEW' (ptr to NEW LEAF)
5189 //    There are total 2 situations for PARENT LEAF: <-----------------------------  ARE GOING UP
5190 //    Case #3: [LP][MP][  ][LW][  ]
5191 //    Case #4: [LP][MP][RP][LW][RW]  Split Occurs
5192
5193 // ~  First deal alonely with the OUTER NODE(LEAF) where Search stopped i.e Case #1 & Case #2:
5194       POffsetInLEAF = BSTstack[--StackPtr];
5195       PseudoLinkedPointer_64 = BSTstack[--StackPtr];
5196 // NOTE: ONE LEAF IS FULL ONLY WHEN LAST CELL FOR KEY(here RW) EXISTS!
5197 // RW: existence check if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 )
5198 //if ( *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // If LEAF is full: Case #2
5199             // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5200             //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4); //RW
5201             //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5202             //fread(&SomeByte, 1, 1, fp_outRG);
5203             // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5204             // [  //r.14+
5205             PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
5206                      if (BSTorBtree == 2) {
5207             fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5208             fread(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5209                      } else { // ######### 64bit memory manipulations [
5210             memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+8+2*(LongestLineInclusive+1+4) );
5211                      } // ######### 64bit memory manipulations ]
5212             memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
5213             // ]  //r.14+
5214             if (SomeByte != 0 ) // RW exists
5215    { SplitOccured = 1; WORDcountDistinct++;
5216           // ALlocate NEW LEAF:
5217 //           if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST
    instead of LL; + more(see LEAF)
5218 //                     {
5219 //                     memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
5220 //                     bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
5221 //                     bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
5222 //                     if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
    long)(bufend[LetterOffset] - BufStart);}
5223 //                   }
5224           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)
5225                     {
5226             PseudoLinkedPointerNEW_64 = BufEnd_64;
5227             BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
5228             BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
5229                     }
5230           else
5231           { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
5232       fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned
    long long)WORDcountDistinct, llTOaDigits2, 10) );
5233       fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, llTOaDigits, 10) );
5234       fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut,
    llTOaDigits, 10) );
5235       fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
5236       fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
5237             return( 1 );
5238           }
```

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



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

```
5415                //fread(&SomeByte, 1, 1, fp_outRG);
5416                // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5417                // [  //r.14+
5418                PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
5419                        if (BSTorBtree == 2) {
5420                fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5421                fread(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5422                        } else { // ########## 64bit memory manipulations [
5423                memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+8+2*(LongestLineInclusive+1+4) );
5424                        } // ########## 64bit memory manipulations ]
5425                memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
5426                // ]  //r.14+
5427                if (SomeByte != 0 ) // RW exists
5428        { SplitOccured = 1;
5429 //                       memcpy( wrdUPold, wrdUP, wrdlen ); // LW up
5430 //                       PseudoLinkedPointerNEWold = PseudoLinkedPointerNEW;
5431                          memcpy( wrdUPold, wrdUP, (LongestLineInclusive+1+4) );
5432                          PseudoLinkedPointerNEWold_64 = PseudoLinkedPointerNEW_64;
5433              // ALlocate NEW LEAF:
5434 //             if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST instead of LL; + more(see LEAF)
5435 //                       {
5436 //                       memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
5437 //                       bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
5438 //                       bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
5439 //                       if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned long)(bufend[LetterOffset] - BufStart);}
5440 //                       }
5441              if( 8 + 8 + 8 + 2*(LongestLineInclusive+1+4) < size_in64_L14 - (BufEnd_64-(unsigned long long)pointerflush_64) ) // the longest wrdlen is LongestLineInclusive but actual is LongestLineInclusive(+ CR char)
5442                        {
5443                PseudoLinkedPointerNEW_64 = BufEnd_64;
5444                BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
5445                BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
5446                // [  //r.14+
5447                //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64;
5448                //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5449                //fread(&LEAFNEW[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5450                // In fact above three lines are slow, the only need is ZEROed LEAFNEW.
5451                memset(&LEAFNEW[0],0,8+8+8+2*(LongestLineInclusive+1+4));
5452                // ]  //r.14+
5453                        }
5454              else
5455              { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
5456        fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(wORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llTOaDigits2, 10) );
5457        fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, llTOaDigits, 10) );
5458        fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10) );
5459        fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
5460        fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
5461                return( 1 );
5462              }
5463        if (POffsetInLEAF == 0) // wrdUPold < LW
5464          {
5465 //                       memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
5466 //                       memcpy( PseudoLinkedPointer+4+4+4, wrdUPold, wrdlen );   // wrdUPold go to OLD LEAF
5467 //                       memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
5468 //                       *(char *)(PseudoLinkedPointer+4+4+4+wrdlen) = 0;    // RW mark unused in OLD LEAF
5469 //                       // [LP](PseudoLinkedPointerNEWold)[MP][RP](wrdUPold)[LW][RW]       ------
5470 //                       //             pair [LW] PseudoLinkedPointerNEW goes up              |
5471 //                       // PseudoLinkedPointer:                          PseudoLinkedPointerNEW: |
5472 //                       // [LP](PseudoLinkedPointerNEWold)[](wrdUPold)  [MP][RP][][RW]      <----
5473 //                       // no need to put zero in RP because logic is based on words existence:
5474 //                       memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+4, 4 );
5475 //                       memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
5476 //                       memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEWold, 4 );
5477              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5478              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
5479              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5480              //fread(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5481               //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5482               //fwrite(&wrdUPold[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5483                //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5484                //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5485                //fread(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5486                //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 8 + 8 + 8;
5487                //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5488                //fwrite(&wrdAUX[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5489                 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5490                 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5491                 //fwrite(&OneChar_ieByte, 1, 1, fp_outRG); // Write ZERO ASCII code
5492              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
5493              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5494              //fread(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
5495              //PseudoLinkedPointerAUX_64 = PseudoLinkedPointerNEW_64 + 0;
5496              //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5497              //fwrite(&PseudoLinkedPointerAUXdumbo_64, 8, 1, fp_outRG);
```



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

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



```
5674                 //fwrite(&PseudoLinkedPointerNEW_64, 8, 1, fp_outRG);
5675                // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5676                // [  //r.14+
5677                memcpy( &wrdAUX[0], &LEAF[8 + 8 + 8], (LongestLineInclusive+1+4) );
5678                memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &wrdAUX[0], (LongestLineInclusive+1+4) );
5679                memcpy( &LEAF[8 + 8 + 8], &wrdUP[0], (LongestLineInclusive+1+4) );
5680                memcpy( &PseudoLinkedPointerAUXdumbo_64, &LEAF[8], 8 );
5681                memcpy( &LEAF[8 + 8], &PseudoLinkedPointerAUXdumbo_64, 8 );
5682                memcpy( &LEAF[8], &PseudoLinkedPointerNEW_64, 8 );
5683                        if (BSTorBtree == 2) {
5684                fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5685                fwrite(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5686                        } else { // ########## 64bit memory manipulations [
5687                memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4) );
5688                        } // ########## 64bit memory manipulations ]
5689                // ]  //r.14+
5690          }
5691        if (POffsetInLEAF == 8) // wrdUP > [LW][] so [LW][] -> [LW][wrdUP]
5692          {
5693 //                       memcpy( PseudoLinkedPointer+4+4+4+wrdlen, wrdUP, wrdlen );
5694 //                               // [LP][MP][] -> [LP][MP][np]
5695 //                       memcpy( PseudoLinkedPointer+8, &PseudoLinkedPointerNEW, 4 );
5696                // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5697                //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
5698                //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5699                //fwrite(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5700                 //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 16;
5701                 //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5702                 //fwrite(&PseudoLinkedPointerNEW_64, 8, 1, fp_outRG);
5703                // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
5704                // [  //r.14+
5705                memcpy( &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], &wrdUP[0], (LongestLineInclusive+1+4) );
5706                memcpy( &LEAF[8 + 8], &PseudoLinkedPointerNEW_64, 8 );
5707                        if (BSTorBtree == 2) {
5708                fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5709                fwrite(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
5710                        } else { // ########## 64bit memory manipulations [
5711                memcpy( (char *)PseudoLinkedPointerAUX_64, &LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4) );
5712                        } // ########## 64bit memory manipulations ]
5713                // ]  //r.14+
5714          }
5715            break;
5716      }
5717          }
5718          else // Empty stack means ROOT and more over ROOT is already splitted(Case #4 is off)
5719          {
5720       // If LEAF is not full: Case #3
5721       // THIS IS WHERE A NEW(SECOND) LEAF 'PseudoLinkedPointerROOT' must be allocated:
5722 //            if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST instead of LL; + more(see LEAF)
5723 //                    {
5724 //                    memcpy( &PseudoLinkedPointerROOT, &bufend[LetterOffset], 4 );
5725 //                    bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
5726 //                    memcpy( bufend[LetterOffset], wrdUP, wrdlen );
5727 //                    bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
5728 //                    if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned long)(bufend[LetterOffset] - BufStart);}
5729 //                    }
5730            if( 8 + 8 + 8 + 2*(LongestLineInclusive+1+4) < size_in64_L14 - (BufEnd_64-(unsigned long long)pointerflush_64) ) // the longest wrdlen is LongestLineInclusive but actual is LongestLineInclusive(+ CR char)
5731                    {
5732              PseudoLinkedPointerROOT_64 = BufEnd_64;
5733              BufEnd_64 = BufEnd_64 + 8 + 8 + 8;
5734              BufEnd_64 = BufEnd_64 + 2*(LongestLineInclusive+1+4);
5735              PseudoLinkedPointerAUX_64 = PseudoLinkedPointerROOT_64 + 8 + 8 + 8;
5736                      if (BSTorBtree == 2) {
5737              fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
5738              fwrite(&wrdUP[0], (LongestLineInclusive+1+4), 1, fp_outRG);
5739                      } else { // ########## 64bit memory manipulations [
5740              memcpy( (char *)PseudoLinkedPointerAUX_64, &wrdUP[0], (LongestLineInclusive+1+4) );
5741                      } // ########## 64bit memory manipulations ]
5742                    }
5743            else
5744            { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
5745        fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llTOaDigits2, 10) );
5746        fprintf( fp_outLOG, "Allocated memory: %s bytes\n", _ui64toaKAZEcomma(size_in64_L14, llTOaDigits, 10) );
5747        fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10) );
5748        fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
5749        fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
5750              return( 1 );
5751            }
5752            // Here                    --  'PseudoLinkedPointerROOT' --
5753            //                          |          (wrdUP)           |
5754            // 'PseudoLinkedPointer' <--                              --> 'PseudoLinkedPointerNEW'
5755            //          (LW)                                                  (RW)
5756 //           memcpy( PseudoLinkedPointerROOT, &PseudoLinkedPointer, 4 );      // LP
```

```
//           memcpy( PseudoLinkedPointerROOT+4, &PseudoLinkedPointerNEW, 4 ); // MP
//           // Here must NEW ROOT be updated i.e. HASH table(SLOT) must point it:
//           memcpy( BufStart+Slot, &PseudoLinkedPointerROOT, 4 );
      PseudoLinkedPointerAUX_64 = PseudoLinkedPointerROOT_64;
                        if (BSTorBtree == 2) {
      fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
      fwrite(&PseudoLinkedPointer_64, 8, 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( (char *)PseudoLinkedPointerAUX_64, &PseudoLinkedPointer_64, 8 );
                        } // ########## 64bit memory manipulations ]
       PseudoLinkedPointerAUX_64 = PseudoLinkedPointerROOT_64 + 8;
                        if (BSTorBtree == 2) {
       fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
       fwrite(&PseudoLinkedPointerNEW_64, 8, 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( (char *)PseudoLinkedPointerAUX_64, &PseudoLinkedPointerNEW_64, 8 );
                        } // ########## 64bit memory manipulations ]
    memcpy( BufStart+Slot, &PseudoLinkedPointerROOT_64, 8 );
           break; //because it is ROOT without split
        }
              } // while
} //if (SplitOccured != 0)
// 3] Insert Iterative ]
} //if (FoundInLinkedList == 0)
} //if (DoNotInsertFlag == 0) { // This line comes since r15FIXFIX+
} //if ( REUSE != 2 ) { //  REUSE [ // This line comes since r16
// ########################################## B-tree order 3 64bit ]
} //if ( (Slot>>HashChunkSizeInBITS) == RipPasses ) {

// External Btrees ]
} //if (BSTorBtree != 2) {
                        } // if ( ( PLE_words == 4 ) && ( wrdlen <= 31 ) ) {
                        } // if( 1 <= wrdlen && wrdlen <= 31 )
/*
The Most Stupid Bug I have ever made [
              else {
              PLE_words_INITflag = 1; // This line fixes the stupidity done since first quadrupleton r1, namely to initialize the sequence
  when a word longer than 31 is spotted - it is wrong to slide it.
              }
The Most Stupid Bug I have ever made ]
*/
//r.15fix [
//This line was intended BUT placed before the fragment 'if( 1 <= wrdlen && wrdlen <= 31 )' [
//                      if( wrdlen > 31 ) PLE_words_INITflag = 1; // Not when wrdlen == 0 as the above buggy fragment!
//This line was intended BUT placed before the fragment 'if( 1 <= wrdlen && wrdlen <= 31 )' ]
//r.15fix ]
if ( PLE_words_INITflag == 1 ) { PLE_words = 0; PLE_words_INITflag = 0; } // Quadruple!
              wrdlen = 0;
                        // This fragment is MIRRORed: #1 copy ]
               }
               //else if( workbyte >= 'A' &&  workbyte <= 'Z' )
               else if( workbyte <= 'Z' )
     {
                        if( wrdlen < 31 )
                        //if( wrdlen < LongestLineInclusive )
                        { wrd[ wrdlen ] = workbyte + 32 ; }
                        wrdlen++;
     }
               else if( workbyte >= 'a' &&  workbyte <= 'z' )
     {
                        if( wrdlen < 31 )
                        //if( wrdlen < LongestLineInclusive )
                        { wrd[ wrdlen ] = workbyte; }
                        wrdlen++;
     }
     else
               {
                        // This fragment is MIRRORed: #2 copy [
               goto ElStupido;
                        // This fragment is MIRRORed: #2 copy ]
     }
        } // i 'for'
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//++Melnitchka;
//Melnitchka = Melnitchka % 4;
//if (Melnitchka == 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 ); }
//if (Melnitchka == 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 ); }
//if (Melnitchka == 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 ); }
//if (Melnitchka == 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 ); }
Melnitchka = Melnitchka & 3; // 0 1 2 3: 00 01 10 11
                        (void) time(&t4);
                        if (t4 <= t1) {t4 = t1; t4++;}
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 );

fclose( fp_inLINE );
} // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ IT IS a FILENAME not a METACOMMAND ]
        LINE10len = 0;
LINE10[ LINE10len ] = 0;
                  }
              }
        } // k 'for'

(void) time(&t3);
if (t3 <= t1) {t3 = t1; t3++;}
printf( "Bytes per second performance: %sB/s\n", _ui64toaKAZEcomma(FilesLEN/((int) t3-t1), llTOaDigits, 10) );  // Rev. 12+
printf( "Phrases per second performance: %sP/s\n", _ui64toaKAZEcomma(WORDcount/((int) t3-t1), llTOaDigits, 10) );  // Rev. 12+
printf("Time for putting phrases into trees: %d second(s)\n", (int) t3-t1);

if (BSTorBtree < 2) {
        // FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH
printf("Flushing unsorted words ...\n");
if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
{ printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
ZEROS[0] = 0; ZEROS[1] = 0; ZEROS[2] = 0; ZEROS[3] = 0;
CRdLFa[0] = 13; CRdLFa[1] = 10;

for( i = 0; i < 806; i++ )
{ //BufStart = pointerflush + i * LetterBuffer; // OLD
  BufStart = pointerflush + (i / 31) * WHOLEletter_BufferSize + OffsetsInBuffer[i % 31];
//  for( j = 0; j < NumberOfSLOTs; j++ )
//  {
//                      Slot = j<<2;
//                      memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
//                      while (PseudoLinkedPointer != 0)
//                      { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
//                        memcpy( PseudoLinkedPointer, ZEROS, 4 );
//                        PseudoLinkedPointer = PseudoLinkedPointerNEW;
//                      }
//  }
//  // Start of COUPLES [OFFSET: 4byte(ZEROS)][WORD:up to 31bytes]
//  //fwrite(BufStart+(NumberOfSLOTs+1)*4, bufend[i] - (BufStart+(NumberOfSLOTs+1)*4), 1, fp_out );
//  /* Follows STATE OF UGLINESS: */
//  Flushing = BufStart+(NumberOfSLOTs+1)*4 + 4; // '+ 4' in order to skip first 4 zeros
//  //in case of current buffer not have been used then NOT entering in this cycle
//  while(Flushing < bufend[i])
//  { if (*Flushing != 0) {fwrite(Flushing, 1, 1, fp_out ); TotalWLchars++;}
//    // Below 'Flushing-1' works due to skipped first 4 zeros!
//    if (*(Flushing-1) != 0 && *Flushing == 0) {fwrite(CRdLFa, 2, 1, fp_out);}
//    //last word must be suffixed with 1310 too
//    if (Flushing == bufend[i]-1) {fwrite(CRdLFa, 2, 1, fp_out );}
//    Flushing++;
//  }

  for( j = 0; j < NumberOfSLOTs; j++ )
  {
                     Slot = j<<2;
                     memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
                     if (PseudoLinkedPointer != 0)
                     {
NumberOfTrees++;
if (BSTorBtree == 0)
{
// ========================================================== BST traverse [
     // DONE JOB:
     // Must be written BST traverse ! with simulated stack i.e. non-recursive.
     // ...
                                      // /*
                                      // Given a binary search tree, print out
                                      // its data elements in increasing
                                      // sorted order.
                                      // */
                                      // void printTree(struct node* node) {
                                      // if (node == NULL) return;
                                      // printTree(node->left);
                                      // printf("%d ", node->data);
                                      // printTree(node->right);
                                      // }

     // FUTURE JOB:
     // I need functions:
     //                BST_LeafNumber() // greater the better
     //                BST_NodeNumber() // 'BSTcurrent' below
     //                BST_Peak()       // i.e. levels, root has height = 1
     //                BST_PeakIB()     // IBBST(Ideal Balanced BST) has 1 + lgNodeNumber height
     // I need 'Ideal Balancing BST FRAGMENT' with simulated stack:
     // I need 'Ideal Balancing BST FRAGMENT' to be executed when Peak() >= PeakIB()<<1:

// ---------------------------------------------------------------------- [
  BSTcurrentNode = 0; BSTcurrentPeak = 0; BSTcurrentLeaf = 0;
                     BSTcurrentPeakMAX = 0; // Height of current BST
```

```
  StackPtr = 0;
  while ( 2==2 ) {
        while (PseudoLinkedPointer != 0)
               {
                 if (StackPtr > 8192*3-1-3) { printf( "\nLeprechaun: Failure! BST simulated stack overflow, too high BST!\n" ); return( 13 );}
                 memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
                 PseudoLinkedPointer = PseudoLinkedPointer + 4;
                 memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer, 4 );
                 BSTstack[StackPtr] = PseudoLinkedPointer + 4; ++StackPtr; //ptr to wrd
                 BSTstack[StackPtr] = PseudoLinkedPointerNEWright; ++StackPtr;
                 BSTstack[StackPtr] = PseudoLinkedPointerNEWleft; ++StackPtr; //needed for stats not for recursion
        // BST stats [
                   if (PseudoLinkedPointerNEWleft == 0 && PseudoLinkedPointerNEWright == 0) {BSTcurrentLeaf++; BSTsTotalLEAFs++;}
                   BSTcurrentPeak++;
                   if (BSTcurrentPeakMAX < BSTcurrentPeak) BSTcurrentPeakMAX = BSTcurrentPeak;
                 BSTstack[StackPtr] = BSTcurrentPeak; ++StackPtr; //needed for stats not for recursion
        // BST stats ]
                 PseudoLinkedPointer = PseudoLinkedPointerNEWright; // choose right instead of 'PseudoLinkedPointerNEWleft' because of stats
print
               }
        if (StackPtr == 0) break;
        BSTcurrentPeak = BSTstack[--StackPtr]; // level of the node(1 is root) needed only for stats(print)
        PseudoLinkedPointerNEWleft = BSTstack[--StackPtr]; // left pointer needed only for stats(print)
        PseudoLinkedPointerNEWright = BSTstack[--StackPtr]; // right pointer
        memcpy( wrd, BSTstack[--StackPtr], i%31+1 );
        fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
        fwrite(CRdLFa, 2, 1, fp_out);
        BSTcurrentNode++;
        PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
  }
// ----------------------------------------------------------------------- ]
        // BST stats [
        if (BSTwithMAXnode < BSTcurrentNode) {
           BSTwithMAXnode = BSTcurrentNode;
           BSTwithMAXnodePEAK = BSTcurrentPeakMAX;
           BSTwithMAXnodeLEAF = BSTcurrentLeaf;
           BSTcurrentNodeMAXQUANTITY = 0;
        }
        if (BSTwithMAXnode == BSTcurrentNode) BSTcurrentNodeMAXQUANTITY++;
        if (BSTwithMAXpeak < BSTcurrentPeakMAX) {
           BSTwithMAXpeak = BSTcurrentPeakMAX;
           BSTwithMAXpeakNODE = BSTcurrentNode;
           BSTwithMAXpeakLEAF = BSTcurrentLeaf;
           BSTcurrentPeakMAXQUANTITY = 0;       iBSTwithMAXpeak=i; jBSTwithMAXpeak=j;
        }
        if (BSTwithMAXpeak == BSTcurrentPeakMAX) BSTcurrentPeakMAXQUANTITY++;
        if (BSTwithMAXleaf < BSTcurrentLeaf) {
           BSTwithMAXleaf = BSTcurrentLeaf;
           BSTwithMAXleafNODE = BSTcurrentNode;
           BSTwithMAXleafPEAK = BSTcurrentPeakMAX;
           BSTcurrentLeafMAXQUANTITY = 0;
        }
        if (BSTwithMAXleaf == BSTcurrentLeaf) BSTcurrentLeafMAXQUANTITY++;
        // BST stats ]
// ======================================================== BST traverse ]
} else
{
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse [
      // DONE JOB:
      // Must be written B-tree traverse ! with simulated stack i.e. non-recursive.
      // ...
  StackPtr = 0;
  while ( 2==2 ) {
        while (PseudoLinkedPointer != 0)
               {
                 if (StackPtr > 8192*3-1) { printf( "\nLeprechaun: Failure! B-tree simulated stack overflow, too high B-tree!\n" ); return( 13
);}
                 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //ptr to Rwrd
if ( *(char *)(PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) == 0 ) {memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTs*4], 4 );}
                 memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
                 memcpy( &PseudoLinkedPointerNEWmiddle, PseudoLinkedPointer + 4, 4 );
                 memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
// Give first from right to left non-zero PTR
                 if (PseudoLinkedPointerNEWright !=0 )
                 { memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTs*4], 4 );
                   PseudoLinkedPointer = PseudoLinkedPointerNEWright;
                 }
                 else if (PseudoLinkedPointerNEWmiddle !=0 )
                 { memcpy( PseudoLinkedPointer + 4, &BufStart[NumberOfSLOTs*4], 4 );
                   PseudoLinkedPointer = PseudoLinkedPointerNEWmiddle;
                 }
                 else if (PseudoLinkedPointerNEWleft !=0 )
                 { memcpy( PseudoLinkedPointer, &BufStart[NumberOfSLOTs*4], 4 );
                   PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
                 }
                 else
                 {
                   PseudoLinkedPointer = 0;
```



```
                 }
               }
        if (StackPtr == 0) break;
        PseudoLinkedPointer = BSTstack[--StackPtr];
                 memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
                 memcpy( &PseudoLinkedPointerNEWmiddle, PseudoLinkedPointer + 4, 4 );
                 memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
        if (PseudoLinkedPointerNEWleft+PseudoLinkedPointerNEWmiddle+PseudoLinkedPointerNEWright == 0) // One LEAF is PRINTED when LP=0 MP=0
RP=0
           {
        memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4, i%31+1 );
        fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
        fwrite(CRdLFa, 2, 1, fp_out);
        if ( *(char *)(PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) != 0 )
            { memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4 + i%31+1, i%31+1 );
              fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
              fwrite(CRdLFa, 2, 1, fp_out);
            }
        PseudoLinkedPointer = 0;
           }
  }
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse ]
}
                       }
  } // j

} // i

if (BSTorBtree == 0)
{
// ~~~~~ Longest path ~~~~~ [
i=iBSTwithMAXpeak; j=jBSTwithMAXpeak;
  BufStart = pointerflush + (i / 31) * WHOLEletter_BufferSize + OffsetsInBuffer[i % 31];
                     Slot = j<<2;
                     memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
                     if (PseudoLinkedPointer != 0)
                     {
// ----------------------------------------------------------------------- [
  BSTcurrentNode = 0; BSTcurrentPeak = 0; BSTcurrentLeaf = 0;
                     BSTcurrentPeakMAX = 0; // Height of current BST
  StackPtr = 0;
        // BST print [
                fprintf( fp_outLOG, "A(not always THE) Binary-Search-Tree with the longest path(height, PEAK, number of levels):\n" );
        // BST print ]
  while ( 2==2 ) {
        while (PseudoLinkedPointer != 0)
               {
                 if (StackPtr > 8192*3-1-3) { printf( "\nLeprechaun: Failure! BST simulated stack overflow, too high BST!\n" ); return( 13 );}
                 memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
                 PseudoLinkedPointer = PseudoLinkedPointer + 4;
                 memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer, 4 );
                 BSTstack[StackPtr] = PseudoLinkedPointer + 4; ++StackPtr; //ptr to wrd
                 BSTstack[StackPtr] = PseudoLinkedPointerNEWright; ++StackPtr;
                 BSTstack[StackPtr] = PseudoLinkedPointerNEWleft; ++StackPtr; //needed for stats not for recursion
        // BST stats [
                   if (PseudoLinkedPointerNEWleft == 0 && PseudoLinkedPointerNEWright == 0) {BSTcurrentLeaf++;} //BSTsTotalLEAFs++;} // REMOVED
to avoid mess in TOTAL stats
                   BSTcurrentPeak++;
                   if (BSTcurrentPeakMAX < BSTcurrentPeak) BSTcurrentPeakMAX = BSTcurrentPeak;
                 BSTstack[StackPtr] = BSTcurrentPeak; ++StackPtr; //needed for stats not for recursion
        // BST stats ]
                 PseudoLinkedPointer = PseudoLinkedPointerNEWright; // choose right instead of 'PseudoLinkedPointerNEWleft' because of stats
print
               }
        if (StackPtr == 0) break;
        BSTcurrentPeak = BSTstack[--StackPtr]; // level of the node(1 is root) needed only for stats(print)
        PseudoLinkedPointerNEWleft = BSTstack[--StackPtr]; // left pointer needed only for stats(print)
        PseudoLinkedPointerNEWright = BSTstack[--StackPtr]; // right pointer
        memcpy( wrd, BSTstack[--StackPtr], i%31+1 );
        //fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
        //fwrite(CRdLFa, 2, 1, fp_out);
        BSTcurrentNode++;
        PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
        // BST print [
                for( k = 0; k < BSTcurrentPeak; k++ ) fprintf( fp_outLOG, "%c", ' ' );
                   if (PseudoLinkedPointerNEWleft == 0) fprintf( fp_outLOG, "[" ); else fprintf( fp_outLOG, "]" );
                for( k = 0; k < i%31+1; k++ ) fprintf( fp_outLOG, "%c", *(char *)(wrd+k) );
                   if (PseudoLinkedPointerNEWright == 0) fprintf( fp_outLOG, "]" ); else fprintf( fp_outLOG, "[" );
                   if (BSTcurrentPeak == 1) fprintf( fp_outLOG, " ROOT" );
                   fprintf( fp_outLOG, "\n" );
        // BST print ]
  }
// ----------------------------------------------------------------------- ]
                     }
fprintf( fp_outLOG, "Above Binary-Search-Tree with MaxPEAK = %s has NODEs = %s and LEAFs = %s\n", _ui64toaKAZEcomma(BSTwithMAXpeak,
llTOaDigits2, 10), _ui64toaKAZEcomma(BSTwithMAXpeakNODE, llTOaDigits3, 10), _ui64toaKAZEcomma(BSTwithMAXpeakLEAF, llTOaDigits, 10));
                fprintf( fp_outLOG, "Legend:\n" );
                fprintf( fp_outLOG, "At left side of the word - '[' means no left successor\n" );
```

```
                    fprintf( fp_outLOG, "At left side of the word - ']' means left successor exists\n" );
                    fprintf( fp_outLOG, "At right side of the word - ']' means no right successor\n" );
                    fprintf( fp_outLOG, "At right side of the word - '[' means right successor exists\n" );
// ~~~~ Longest path ~~~~ ]

        // BST stats [
PEAKibBST=1+floorLog2(BSTwithMAXnode);
//PEAKibBST=1;
//while (BSTwithMAXnode>>PEAKibBST) PEAKibBST++;
        // BST stats ]
}

(void) time(&t2);
if (t2 <= t1) {t2 = t1; t2++;}
printf("Time for making unsorted wordlist: %d second(s)\n", (int) t2-t1);
fprintf( fp_outLOG, "Bytes per second performance: %sB/s\n", _ui64toaKAZEcomma(FilesLEN/((int) t3-t1), llTOaDigits, 10) );  // Rev. 12+
fprintf( fp_outLOG, "Words per second performance: %sW/s\n", _ui64toaKAZEcomma(WORDcount/((int) t3-t1), llTOaDigits, 10) );  // Rev. 12+
fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, llTOaDigits, 10) );
fprintf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, llTOaDigits, 10), _ui64toaKAZEcomma((unsigned long long)WORDcountDistinct, llTOaDigits2, 10) );
fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
NumberOfHashCollisions = WORDcountDistinct - NumberOfTrees;
fprintf( fp_outLOG, "Number Of Trees(GREATER THE BETTER): %lu\n", NumberOfTrees );
fprintf( fp_outLOG, "Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): %lu%s\n", (NumberOfTrees*100)/(26*31*8192), "%\0" );
fprintf( fp_outLOG, "Number Of Hash Collisions(Distinct WORDs - Number Of Trees): %lu\n", NumberOfHashCollisions );

if (BSTorBtree == 0)
{
fprintf( fp_outLOG, "Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '%s'\n", _ui64toaKAZEcomma(BSTwithMAXpeak, llTOaDigits, 10) );
fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10) );
fprintf( fp_outLOG, "Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): %s\n", _ui64toaKAZEcomma(BSTsTotalLEAFs, llTOaDigits, 10) );
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, llTOaDigits, 10), _ui64toaKAZEcomma(PEAKibBST, llTOaDigits2, 10), _ui64toaKAZEcomma(BSTwithMAXnode, llTOaDigits3, 10));
fprintf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxNODEs = %s has PEAK = %s and LEAFs = %s\n", _ui64toaKAZEcomma(BSTcurrentNodeMAXqUANTITY, llTOaDigits4, 10), _ui64toaKAZEcomma(BSTwithMAXnode, llTOaDigits2, 10), _ui64toaKAZEcomma(BSTwithMAXnodePEAK, llTOaDigits3, 10), _ui64toaKAZEcomma(BSTwithMAXnodeLEAF, llTOaDigits, 10));
fprintf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxPEAK = '%s' has NODEs = %s and LEAFs = %s\n", _ui64toaKAZEcomma(BSTcurrentPeakMAXqUANTITY, llTOaDigits4, 10), _ui64toaKAZEcomma(BSTwithMAXpeak, llTOaDigits2, 10), _ui64toaKAZEcomma(BSTwithMAXpeakNODE, llTOaDigits3, 10), _ui64toaKAZEcomma(BSTwithMAXpeakLEAF, llTOaDigits, 10));
fprintf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxLEAFs = %s has NODEs = %s and PEAK = %s\n", _ui64toaKAZEcomma(BSTcurrentLeafMAXqUANTITY, llTOaDigits4, 10), _ui64toaKAZEcomma(BSTwithMAXleaf, llTOaDigits2, 10), _ui64toaKAZEcomma(BSTwithMAXleafNODE, llTOaDigits3, 10), _ui64toaKAZEcomma(BSTwithMAXleafPEAK, llTOaDigits, 10));
} else
{
fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10) );
}

for( k = 1; k < 32; k++ )
{ fprintf( fp_outLOG, "Words with length %s occupy %sKB of %sKB given i.e. %s%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
  if ( MAXusedBufferABS < (31 * ((MAXusedBuffer[k]>>10)+1)) / GRMBLhill[(int)k] ) {MAXusedBufferABS = 1+(31 * ((MAXusedBuffer[k]>>10)+1)) / GRMBLhill[(int)k];}
  Utiliza1 = Utiliza1 + (MAXusedBuffer[k]>>10)+1;
  Utiliza2 = Utiliza2 + (((GRMBLhill[(int)k] * LetterBuffer)/31)>>10)+1;
}
fprintf( fp_outLOG, "Total pseudo(including hash table) memory utilization: %s%s\n", _ui64toaKAZEzerocomma((Utiliza1*100)/Utiliza2, llTOaDigits, 10)+(26-2), "%\0" ); // 26 are all 26-DESIRED=24
fprintf( fp_outLOG, "Total real(wordlist's words VS allocated block) memory utilization: %s/1000\n", _i64toaKAZE(((unsigned long long)TotalWLchars*1000)/memory_size, llTOaDigits, 10) ); // 26 are all 26-DESIRED=24
fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
fprintf( fp_outLOG, "Use next time as third parameter: %lu-\n", MAXusedBufferABS ); // 26 are all 26-DESIRED=24
fprintf( fp_outLOG, "Time for making unsorted wordlist: %d second(s)\n", (int) t2-t1);

// EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT EXIT
printf( "Deallocated memory in MB: %lu\n", (memory_size>>20)+1 );
free(pointerflushUNALIGN);
fclose(fp_out);
fclose(fp_outLOG);
// SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT
   //printf("Uploading unsorted wordlist ...\n");
   if ((nlines = readlines(argv[2], &backup)) >= 0)
   {  //printf("Number of words(lines) uploaded: %lu\n", nlines);
      //printf("Note1: Press 'Ctrl+C' to abort sorting, unsorted wordlist(second parameter)\n");
      //printf("       will remain intact(unless flushing is in progress) because of\n");
      //printf("       pointers-to-data are being sorted not the data itself.\n");
      //printf("Note2: In near future 'InsertionX26Sort' will be replaced with 'QuickX26Sort':\n");
      //printf("       which is much faster than 'QuickSort' applied for data-at-once!\n");
```



```

      // !!!??? I AM DISAPPOINTED x26 is just an illusion !!!???
      // X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26
// argc is 4|5|6 due to eventual missing BufferSize
if( argc == 4 ) // not 5 due to eventual missing BufferSize
    k_FIX = 3;
if( argc == 5 || argc == 6 )
    k_FIX = 4;
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')
{ printf("Sorting(with 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...\n");
      /* ???!!! What an unexpected behavior! I have been hit: for SOED5.HTM(259,835 distinct words): InsertionX26Sort gives 46s, InsertionSort gives 28s */
      for( k = 0; k < 26; k++ )
      { printf( "Sort pass %s/26 ...\r", _ui64toaKAZEzerocomma(k+1, llTOaDigits, 10)+(26-2));
        HEADOffetFromStartBUKVA = TAILOffetFromStartBUKVA;
        while( (TAILOffetFromStartBUKVA < nlines) && (*backup[TAILOffetFromStartBUKVA] - 'a' == k) )
        { TAILOffetFromStartBUKVA++;
        }
        if (HEADOffetFromStartBUKVA != TAILOffetFromStartBUKVA)
        { mkqsort_main(backup + HEADOffetFromStartBUKVA, TAILOffetFromStartBUKVA - HEADOffetFromStartBUKVA + 0); // backup[0..nlines-1]
        }
      }
}
else
{
if (*argv[k_FIX] == 'A' || *argv[k_FIX] == 'a')
  { printf("Sorting(with 'InsertionSort') ...");
    InsertSortKAZE(backup, nlines, 0); // backup[0..nlines-1]
  }
if (*argv[k_FIX] == 'B' || *argv[k_FIX] == 'b')
  { printf("Sorting(with 'InsertionX26Sort') ...\n");
      /* ???!!! What an unexpected behavior! I have been hit: for SOED5.HTM(259,835 distinct words): InsertionX26Sort gives 46s, InsertionSort gives 28s */
      for( k = 0; k < 26; k++ )
      { printf( "Sort pass %s/26 ...\r", _ui64toaKAZEzerocomma(k+1, llTOaDigits, 10)+(26-2));
        HEADOffetFromStartBUKVA = TAILOffetFromStartBUKVA;
        while( (TAILOffetFromStartBUKVA < nlines) && (*backup[TAILOffetFromStartBUKVA] - 'a' == k) )
        { TAILOffetFromStartBUKVA++;
        }
        if (HEADOffetFromStartBUKVA != TAILOffetFromStartBUKVA)
        { InsertSortKAZE(backup + HEADOffetFromStartBUKVA, TAILOffetFromStartBUKVA - HEADOffetFromStartBUKVA + 0, 0); // backup[0..nlines-1]
        }
      }
  }
if (*argv[k_FIX] == 'C' || *argv[k_FIX] == 'c')
  { printf("Sorting(with 'MultiKeyQuickSortSort' by J. Bentley and R. Sedgewick) ...");
    mkqsort_main(backup, nlines); // backup[0..nlines-1]
  }
if (*argv[k_FIX] == 'D' || *argv[k_FIX] == 'd')
  { printf("Sorting(with 'MultiKeyQuickSortX26Sort' by J. Bentley and R. Sedgewick) ...\n");
      /* ???!!! What an unexpected behavior! I have been hit: for SOED5.HTM(259,835 distinct words): InsertionX26Sort gives 46s, InsertionSort gives 28s */
      for( k = 0; k < 26; k++ )
      { printf( "Sort pass %s/26 ...\r", _ui64toaKAZEzerocomma(k+1, llTOaDigits, 10)+(26-2));
        HEADOffetFromStartBUKVA = TAILOffetFromStartBUKVA;
        while( (TAILOffetFromStartBUKVA < nlines) && (*backup[TAILOffetFromStartBUKVA] - 'a' == k) )
        { TAILOffetFromStartBUKVA++;
        }
        if (HEADOffetFromStartBUKVA != TAILOffetFromStartBUKVA)
        { mkqsort_main(backup + HEADOffetFromStartBUKVA, TAILOffetFromStartBUKVA - HEADOffetFromStartBUKVA + 0); // backup[0..nlines-1]
        }
      }
  }
}

      // X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26 X26

      printf("\nFlushing sorted words ...\n");
      if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
      { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
      for( j = 0; j < nlines; j++ )
      { //Slot = KuxHash3plus(backup[j]);
        //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, llTOaDigits, 10)+(26-5));
        fprintf(fp_out, "%s", backup[j]);
        fwrite(CRdLFa, 2, 1, fp_out );
      }
      (void) time(&t3);
      if (t3 <= t2) {t3 = t2; t3++;}
      printf("Time for sorting unsorted wordlist: %d second(s)\n", (int) t3-t2);

/*
// Hash benchmarking ---------------------------------------------- [

// 5[
clocks1 = clock();
    for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
{
      for( j = 0; j < nlines; j++ )
```

```
        { //Slot = KuxHash3plus(backup[j]);
          //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, llTOaDigits, 10)+(26-5));
                Slot = FNV1A_Hash_4_OCTETS(backup[j], (strlen(backup[j])>>2)); //13+++
        }
}
clocks2 = clock();
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))  );
// 5]

// 1[
clocks1 = clock();
    for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
{
        for( j = 0; j < nlines; j++ )
        { //Slot = KuxHash3plus(backup[j]);
          //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, llTOaDigits, 10)+(26-5));
// To make it EVEN !!!
//wrdlen = strlen(backup[j]);
//if (strlen(backup[j]) != 0)
                Slot = FNV1A_Hash(backup[j]); //13+++
        }
}
clocks2 = clock();
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))  );
// 1]

// 2[
clocks1 = clock();
    for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
{
        for( j = 0; j < nlines; j++ )
        { //Slot = KuxHash3plus(backup[j]);
          //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, llTOaDigits, 10)+(26-5));
                Slot = FNV1A_Hash_4_OCTETS_31(backup[j], (strlen(backup[j])>>2)); //13+++
        }
}
clocks2 = clock();
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))  );
// 2]

// 4[
clocks1 = clock();
    for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
{
        for( j = 0; j < nlines; j++ )
        { //Slot = KuxHash3plus(backup[j]);
          //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, llTOaDigits, 10)+(26-5));
// To make it EVEN !!!
//wrdlen = strlen(backup[j]);
//if (strlen(backup[j]) != 0)
                Slot = KuxHash3plus(backup[j]); //13++
        }
}
clocks2 = clock();
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))  );
// 4]

// 6[
clocks1 = clock();
    for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
{
        for( j = 0; j < nlines; j++ )
        { //Slot = KuxHash3plus(backup[j]);
          //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZEzerocomma(Slot, llTOaDigits, 10)+(26-5));
wrdlen = strlen(backup[j]);
if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
                Slot = FNV1A_Hash_Granularity(backup[j], wrdlen>>2, 2); //13+++++
else            // 2x8+4=20 i.e. first contains 6 clashes
                Slot = FNV1A_Hash_Granularity(backup[j], wrdlen>>3, 3); //13+++++
        } // Conclusion: two functions > 64 bytes lead to horrible slowness, so unite them in one: fit in the cache line.
}
clocks2 = clock();
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))  );
// 6]

// Hash benchmarking ---------------------------------------------- ]
*/

if( ( fp_outLOG = fopen( "Leprechaun.LOG", "a+" ) ) == NULL )
{ printf( "Leprechaun: Can't open file Leprechaun.LOG.\n" ); return( 1 ); }
fprintf( fp_outLOG, "Time for sorting unsorted wordlist: %d second(s)\n\n", (int) t3-t2);
printf( "Leprechaun: Done.\n" );
        return 0;
```



```
    }
    else
    {  printf("Leprechaun: Input file too large, wordlist remains unsorted!\n");
       return 1;
    }
// SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT
} else { //if (BSTorBtree != 2) {
// External Btrees [

// r16 [
if (REUSE==2) {
     fclose(fp_out);
}
// r16 ]

     if ( REUSE == 0 ) { // r16FIX <*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*> [

(void) time(&t1);
WORDcountBOTTOM = 0;
//printf("Flushing UNsorted phrases ...\r");
//if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL ) // Commented since r.14++++ because passes need concatenation.

if( ( fp_out = fopen( argv[2], "ab+" ) ) == NULL )
{ printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
CRdLFa[0] = 13; CRdLFa[1] = 10;

BufStart = pointerflush;
//   for( j = 0; j < 28*28*28*28*28; j++ )
for( j = 0; j < (1<<HashInBITS); j++ )
{
                              if ((j & ((1<<14)-1)) == 0) {
                        (void) time(&t3);
                        if (t3 <= t1) {t3 = t1; t3++;}
                        printf("Flushing UNsorted phrases: %s%%; Shaking trees performance: %sP/s\r", _ui64toaKAZEzerocomma(((long
long)j*100)/((1<<HashInBITS)), llTOaDigits, 10)+(26-3), _ui64toaKAZEzerocomma(WORDcountBOTTOM/((int) t3-t1), llTOaDigits2, 10)+(26-10));
                        }
                       Slot = j<<3;
                       memcpy( &PseudoLinkedPointer_64, BufStart+Slot, 8 );
                       if (PseudoLinkedPointer_64 != 0)
                       {
NumberOfTrees++;

// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse 64bit [
     // DONE JOB:
     // Must be written B-tree traverse ! with simulated stack i.e. non-recursive.
     // ...
  StackPtr = 0;
  while ( 2==2 ) {
        while (PseudoLinkedPointer_64 != 0)
                {
                  if (StackPtr > 8192*3-1) { printf( "\nLeprechaun: Failure! B-tree simulated stack overflow, too high B-tree!\n" ); return( 13
);}
                 BSTstack[StackPtr] = PseudoLinkedPointer_64; ++StackPtr; //ptr to Rwrd
//if ( *(char *)(PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) == 0 ) {memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTs*4], 4 );}
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4); //RW
     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
     //fread(&SomeByte, 1, 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
                        if (BSTorBtree == 2) {
              fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              fread(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+8+2*(LongestLineInclusive+1+4) );
                        } // ########## 64bit memory manipulations ]
              memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
              // ]  //r.14+
     if (SomeByte == 0 ) // RW exists not
      {
       PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
                        if (BSTorBtree == 2) {
       fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
       fwrite(&NULLs_64, 8, 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLs_64, 8 );
                        } // ########## 64bit memory manipulations ]
              // [  //r.14+
              memcpy( &LEAF[8 + 8], &NULLs_64, 8 );
              // ]  //r.14+
      }
//                memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
//                memcpy( &PseudoLinkedPointerNEWmiddle, PseudoLinkedPointer + 4, 4 );
//                memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
       //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
       //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
```

```
       //fread(&PseudoLinkedPointerNEWleft_64, 8, 1, fp_outRG);
        //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
        //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
        //fread(&PseudoLinkedPointerNEWmiddle_64, 8, 1, fp_outRG);
         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
         //fread(&PseudoLinkedPointerNEWright_64, 8, 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &PseudoLinkedPointerNEWleft_64, &LEAF[0], 8 );
              memcpy( &PseudoLinkedPointerNEWmiddle_64, &LEAF[8], 8 );
              memcpy( &PseudoLinkedPointerNEWright_64, &LEAF[8+8], 8 );
              // ]  //r.14+
// Give first from right to left non-zero PTR
                if (PseudoLinkedPointerNEWright_64 !=0 )
//                { memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTs*4], 4 );
//                  PseudoLinkedPointer = PseudoLinkedPointerNEWright;
//                }
                {
     PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
                        if (BSTorBtree == 2) {
     fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
     fwrite(&NULLs_64, 8, 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLs_64, 8 );
                        } // ########## 64bit memory manipulations ]
      PseudoLinkedPointer_64 = PseudoLinkedPointerNEWright_64;
                }
                else if (PseudoLinkedPointerNEWmiddle_64 !=0 )
//                { memcpy( PseudoLinkedPointer + 4, &BufStart[NumberOfSLOTs*4], 4 );
//                  PseudoLinkedPointer = PseudoLinkedPointerNEWmiddle;
//                }
                {
     PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
                        if (BSTorBtree == 2) {
     fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
     fwrite(&NULLs_64, 8, 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLs_64, 8 );
                        } // ########## 64bit memory manipulations ]
      PseudoLinkedPointer_64 = PseudoLinkedPointerNEWmiddle_64;
                }
                else if (PseudoLinkedPointerNEWleft_64 !=0 )
//                { memcpy( PseudoLinkedPointer, &BufStart[NumberOfSLOTs*4], 4 );
//                  PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
//                }
                {
     PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
                        if (BSTorBtree == 2) {
     fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
     fwrite(&NULLs_64, 8, 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( (char *)PseudoLinkedPointerAUX_64, &NULLs_64, 8 );
                        } // ########## 64bit memory manipulations ]
      PseudoLinkedPointer_64 = PseudoLinkedPointerNEWleft_64;
                }
                else
                {
                  PseudoLinkedPointer_64 = 0;
                }
              }
     if (LevelsInCorona_Not_Counting_ROOT < StackPtr) LevelsInCorona_Not_Counting_ROOT = StackPtr; //r.14
        if (StackPtr == 0) break;
        PseudoLinkedPointer_64 = BSTstack[--StackPtr];
//                memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
//                memcpy( &PseudoLinkedPointerNEWmiddle, PseudoLinkedPointer + 4, 4 );
//                memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer + 4 + 4, 4 );
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
       //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
       //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
       //fread(&PseudoLinkedPointerNEWleft_64, 8, 1, fp_outRG);
        //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8;
        //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
        //fread(&PseudoLinkedPointerNEWmiddle_64, 8, 1, fp_outRG);
         //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8;
         //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
         //fread(&PseudoLinkedPointerNEWright_64, 8, 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64;
                        if (BSTorBtree == 2) {
              fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
              fread(&LEAF[0], 8+8+8+2*(LongestLineInclusive+1+4), 1, fp_outRG);
                        } else { // ########## 64bit memory manipulations [
              memcpy( &LEAF[0], (char *)PseudoLinkedPointerAUX_64, 8+8+8+2*(LongestLineInclusive+1+4) );
                        } // ########## 64bit memory manipulations ]
              memcpy( &PseudoLinkedPointerNEWleft_64, &LEAF[0], 8 );
              memcpy( &PseudoLinkedPointerNEWmiddle_64, &LEAF[8], 8 );
```



```
              memcpy( &PseudoLinkedPointerNEWright_64, &LEAF[8+8], 8 );
              // ]  //r.14+
        if (PseudoLinkedPointerNEWleft_64 + PseudoLinkedPointerNEWmiddle_64 + PseudoLinkedPointerNEWright_64 == 0) // One LEAF is PRINTED when
LP=0 MP=0 RP=0
          {
//        memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4, i%31+1 );
//        fwrite(wrd, i%31+1, 1, fp_out);
//        fwrite(CRdLFa, 2, 1, fp_out);
//        if ( *(char *)(PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) != 0 )
//          { memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4 + i%31+1, i%31+1 );
//            fwrite(wrd, i%31+1, 1, fp_out);
//            fwrite(CRdLFa, 2, 1, fp_out);
//          }
//        PseudoLinkedPointer = 0;
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8;
     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
     //fread(&wrd[0], (LongestLineInclusive+1+4), 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &wrd[0], &LEAF[8 + 8 + 8], (LongestLineInclusive+1+4) );
              // ]  //r.14+
              // Counter [
              memcpy( &CounterOccurrencies, &wrd[(LongestLineInclusive+1+4)-4], 4 );
              if (CounterOccurrencies<9999999) CounterOccurrencies++; // Starting from ZERO! Because when insertion happened there was no
setting counter to 1.
              // Counter ]
if (*argv[k_FIX] == 'Y' || *argv[k_FIX] == 'Z') // || *argv[k_FIX] == 'W'
        fprintf(fp_out, "%s\t%s\r\n", _ui64toaKAZEzerocomma(CounterOccurrencies, llTOaDigits2, 10)+(26-9), wrd); WORDcountBOTTOM++;
if (*argv[k_FIX] == 'y' || *argv[k_FIX] == 'z') // || *argv[k_FIX] == 'w'
        fprintf(fp_out, "%s\r\n", wrd); WORDcountBOTTOM++;
              //fwrite(CRdLFa, 2, 1, fp_out);
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4); //RW
     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
     //fread(&SomeByte, 1, 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &SomeByte, &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], 1 );
              // ]  //r.14+
     if (SomeByte != 0 ) // RW exists
     {
              // [  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
     //PseudoLinkedPointerAUX_64 = PseudoLinkedPointer_64 + 8 + 8 + 8 + (LongestLineInclusive+1+4);
     //fsetpos(fp_outRG, &PseudoLinkedPointerAUX_64);
     //fread(&wrd[0], (LongestLineInclusive+1+4), 1, fp_outRG);
              // ]  //r.14+ Optimized I/O i.e. reading a LEAF at once not LEAF's elements one-by-one!
              // [  //r.14+
              memcpy( &wrd[0], &LEAF[8 + 8 + 8 + (LongestLineInclusive+1+4)], (LongestLineInclusive+1+4) );
              // ]  //r.14+
              // Counter [
              memcpy( &CounterOccurrencies, &wrd[(LongestLineInclusive+1+4)-4], 4 );
              if (CounterOccurrencies<9999999) CounterOccurrencies++; // Starting from ZERO! Because when insertion happened there was no
setting counter to 1.
              // Counter ]
if (*argv[k_FIX] == 'Y' || *argv[k_FIX] == 'Z') // || *argv[k_FIX] == 'W'
              fprintf(fp_out, "%s\t%s\r\n", _ui64toaKAZEzerocomma(CounterOccurrencies, llTOaDigits2, 10)+(26-9), wrd); WORDcountBOTTOM++;
if (*argv[k_FIX] == 'y' || *argv[k_FIX] == 'z') // || *argv[k_FIX] == 'w'
              fprintf(fp_out, "%s\r\n", wrd); WORDcountBOTTOM++;
               //fwrite(CRdLFa, 2, 1, fp_out);
     }
       PseudoLinkedPointer_64 = 0;
     NumberOfLEAFs++; //r.14
          }
   }
// $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse 64bit ]

                    }
}

fclose(fp_out);
(void) time(&t2);
if (t2 <= t1) {t2 = t1; t2++;}
printf("Flushing UNsorted phrases: %s%%; Shaking trees performance: %sP/s\n", _ui64toaKAZEzerocomma(100, llTOaDigits, 10)+(26-3),
_ui64toaKAZEzerocomma(WORDcountBOTTOM/((int) t2-t1), llTOaDigits2, 10)+(26-10));
printf("Time for shaking phrases from trees: %d second(s)\n", (int) t2-t1);

// TO DO ...
//printf( "Leprechaun: THE DUMP NOT SORTED?! HASH sucks also - these two issues ought to be fixed in r.14. This is revision 14-.\n" );

     } //if ( REUSE == 0 ) { // r16FIX <*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*> ]

fprintf( fp_outLOG, "Number Of Hash Collisions(Distinct WORDs - Number Of Trees): %s\n", _ui64toaKAZEcomma(WORDcountDistinct - NumberOfTrees,
llTOaDigits, 10) );
fprintf( fp_outLOG, "Number Of Trees(GREATER THE BETTER): %s\n", _ui64toaKAZEcomma(NumberOfTrees, llTOaDigits, 10) );
fprintf( fp_outLOG, "Number Of LEAFs(littler THE BETTER) not counting ROOT LEAFs: %s\n", _ui64toaKAZEcomma(NumberOfLEAFs-NumberOfTrees,
llTOaDigits, 10) );
fprintf( fp_outLOG, "Highest Tree not counting ROOT Level i.e. CORONA levels(littler THE BETTER): %s\n",
```

```
    _ui64toaKAZEcomma(LevelsInCorona_Not_Counting_ROOT-1, llTOaDigits, 10) );
6587
6588 fprintf( fp_outLOG, "Used value for third parameter in KB: %s\n", _ui64toaKAZEcomma(Thunderwith, llTOaDigits, 10) );
6589 fprintf( fp_outLOG, "Use next time as third parameter: %s\n", _ui64toaKAZEcomma((((BufEnd_64-(unsigned long long)pointerflush_64)+1)>>10)+1,
    llTOaDigits, 10) );
6590 fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, llTOaDigits, 10)
    );
6591
6592 // External Btrees ]
6593                         if (BSTorBtree == 2) {
6594 fclose(fp_outRG);
6595 // r16
6596 if ( (REUSE == 1) && ((HashInBITS-HashChunkSizeInBITS)==0) ) { // Multiple-passes shouldn't be dumped - it is meaningless, dump when only one
    pass.
6597     if( ( fp_outRG = fopen( "Leprechaun_64bit.hsh", "wb+" ) ) == NULL )
6598     { printf( "Leprechaun: Can't create file 'Leprechaun_64bit.hsh'.\n" ); return( 1 ); }
6599     fwrite(pointerflushUNALIGN,  (1<<HashInBITS)*8 + 1 + 64 , 1, fp_outRG);
6600 fclose(fp_outRG);
6601 }
6602                         } else { // ########## 64bit memory manipulations [
6603 free(pointerflushUNALIGN_64);
6604                         } // ########## 64bit memory manipulations ]
6605 free(pointerflushUNALIGN);
6606 fclose(fp_outLOG);
6607 printf( "Leprechaun: Current pass done.\n" );
6608
6609 // 14++++ [
6610  TotalMemoryNeededForOnePass += (((BufEnd_64-(unsigned long long)pointerflush_64)+1)>>10)+1;
6611  WORDcountDistinctTOTAL += WORDcountDistinct;
6612  RipPasses++;
6613  if (RipPasses <= (1<<(HashInBITS-HashChunkSizeInBITS))-1) goto WhyTheHellForIsNotWorking;
6614 //} // for( RipPasses = 1-1; RipPasses <= (1<<(HashInBITS-HashChunkSizeInBITS))-1; RipPasses++ )
6615 // 14++++ ]
6616 (void) time(&tMainE);
6617 if (tMainE <= tMainB) {tMainE = tMainB; tMainE++;} // This line fixes a bug in r.15
6618 printf( "\nTotal memory needed for one pass: %sKB\n", _ui64toaKAZEcomma(TotalMemoryNeededForOnePass, llTOaDigits2, 10) );
6619 printf( "Total distinct phrases: %s\n", _ui64toaKAZEcomma(WORDcountDistinctTOTAL, llTOaDigits2, 10) );
6620 printf("Total time: %d second(s)\n", (int) tMainE-tMainB);
6621 printf( "Total performance: %sP/s i.e. phrases per second\n", _ui64toaKAZEcomma(WORDcount/((int) tMainE-tMainB), llTOaDigits2, 10) );
6622
6623 printf( "Leprechaun: Done.\n" );
6624 exit(0);
6625 } //if (BSTorBtree != 2) {
6626 } // main()
6627
6628 /*
6629 TO BE DONE: Ideal Balancing BST [
6630
6631 link rotR(link h)
6632   { link x = h->l; h->l = x->r; x->r = h;
6633     return x; }
6634
6635 link rotL(link h)
6636   { link x = h->r; h->r = x->l; x->l = h;
6637     return x; }
6638
6639 link partR(link h, int k)
6640   { int t = h->l->N;
6641     if (t > k )
6642       { h->l = partR(h->l, k); h = rotR(h); }
6643     if (t < k )
6644       { h->r = partR(h->r, k-t-1); h = rotL(h); }
6645     return h;
6646   }
6647
6648 link balanceR(link h)
6649   {
6650     if (h->N < 2) return h;
6651     h = partR(h, h->N/2);
6652     h->l = balanceR(h->l);
6653     h->r = balanceR(h->r);
6654     return h;
6655   }
6656
6657 TO BE DONE: Ideal Balancing BST ]
6658 */
6659
6660 /*
6661 #include <stdlib.h>
6662 #include "Item.h"
6663 typedef struct STnode* link;
6664 struct STnode { Item item; link l, r; int N };
6665 static link head, z;
6666 link NEW(Item item, link l, link r, int N)
6667   { link x = malloc(sizeof *x);
6668     x->item = item; x->l = l; x->r = r; x->N = N;
6669     return x;
6670   }
```



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