



```
_FNV1A_Hash_Jesteress PROC
...
$LL6@FNV1A_Hash@8:
mov     edi, DWORD PTR [eax]
rol     edi, 5
xor     edi, DWORD PTR [eax+4]
sub     edx, 8
xor     ecx, edi
imul   ecx, 709607
add     eax, 8
dec     esi
jne     SHORT $LL6@FNV1A_Hash@8
pop     edi
$LN4@FNV1A_Hash@8:
test    dl, 4
je      SHORT $LN3@FNV1A_Hash@8
xor     ecx, DWORD PTR [eax]
imul   ecx, 709607
add     eax, 4
$LN3@FNV1A_Hash@8:
test    dl, 2
je      SHORT $LN2@FNV1A_Hash@8
movzx  esi, WORD PTR [eax]
xor     ecx, esi
imul   ecx, 709607
add     eax, 2
$LN2@FNV1A_Hash@8:
pop     esi
test    dl, 1
je      SHORT $LN1@FNV1A_Hash@8
movsx  eax, BYTE PTR [eax]
xor     ecx, eax
imul   ecx, 709607
$LN1@FNV1A_Hash@8:
...
_FNV1A_Hash_Jesteress ENDP
```

LEPRECHAUN QUADRUPLETON

AN ENGLISH-WORDLIST 4GRAM RIPPER, REVISION 13++++++FIX

Free download at www.sanmayce.com — on Intel Merom-1M 2166 MHz it rips **wikipedia** at 2,860,880++ words per second.

```

0001 /*
0002 This is source of Leprechaun revision 13_7pluses_FIX quadrupleton_r1, copyleft Sammayce, 2011-Jan-26. 2011-Mar-07: Fixed a small command line
parsing bug.
0003
0004 Comment/Uncomment accordingly in order to compile:
0005 //define _WIN32_ENVIRONMENT_
0006 #define _POSIX_ENVIRONMENT_
0007
0008 Linux compile(uncomment #include <i.o.h> line, ignore warnings):
0009 gcc -D_FILE_OFFSET_BITS=64 -m32 -static -O3 -mtune=generic Leprechaun.c -o Leprechaun_r13_7pluses_generic_32bits.elf
0010
0011 Windows compile(uncomment #include <i.o.h> line, ignore warnings):
0012 For Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86 use:
0013 cl /ox /wp64 /TcLeprechaun.c /FaLeprechaun
0014
0015 Windows compile(comment #include <i.o.h> line, ignore warnings):
0016 For Intel(R) C++ Compiler Professional for applications running on IA-32, Version 11.1 use:
0017 icl /ox /wp64 /TcLeprechaun.c /FaLeprechaun /w /QXHOST
0018
0019 [It's a little weird(Intel boosts the sort while falls behind in parsing, tested on T3400):]
0020
0021 Leprechaun_r13_7pluses_Microsoft_32-bit_16.00.30319.01.exe _vs_ wikipedia_22,202,980_LATIN-words:
0022 words per second performance: 1,679,585w/s
0023 Time for making unsorted wordlist: 30 second(s)
0024 Time for sorting unsorted wordlist: 25 second(s)
0025
0026 Leprechaun_r13_7pluses_Intel_IA-32_11.1.exe _vs_ wikipedia_22,202,980_LATIN-words:
0027 words per second performance: 1,603,240w/s
0028 Time for making unsorted wordlist: 31 second(s)
0029 Time for sorting unsorted wordlist: 19 second(s)
0030
0031 Due to my ignorance(calderas in my C knowledge): 64bit code cannot be generated, for now.
0032 Any improvement is welcome.
0033 Enjoy!
0034 */
0035
0036 // C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_r13+++++_C_EXE\cl /ox /wp64 /TcLeprechaun.c /FaLeprechaun
0037 // Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
0038 // Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
0039 //
0040 // Leprechaun.c
0041 // Leprechaun.c(829) : warning C4312: 'type cast' : conversion from 'int' to 'string' of greater size
0042 // Leprechaun.c(849) : warning C4312: 'type cast' : conversion from 'int' to 'string *' of greater size
0043 // Leprechaun.c(2048) : warning C4312: 'type cast' : conversion from 'int' to 'char *' of greater size
0044 // Leprechaun.c(2063) : warning C4311: 'type cast' : pointer truncation from 'char *' to 'unsigned long'
0045 // Leprechaun.c(2068) : warning C4311: 'type cast' : pointer truncation from 'char *' to 'unsigned long'
0046 // Leprechaun.c(2371) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0047 // Leprechaun.c(2570) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0048 // Leprechaun.c(2626) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0049 // Leprechaun.c(2657) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0050 // Leprechaun.c(2663) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0051 // Leprechaun.c(2668) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0052 // Leprechaun.c(2696) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0053 // Leprechaun.c(2729) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0054 // Leprechaun.c(2743) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0055 // Leprechaun.c(2755) : warning C4312: 'type cast' : conversion from 'unsigned long' to 'char *' of greater size
0056 // Microsoft (R) Incremental Linker Version 7.10.3077
0057 // Copyright (C) Microsoft Corporation. All rights reserved.
0058 //
0059 // /out:Leprechaun.exe
0060 // Leprechaun.obj
0061 //
0062 // C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_r13+++++_C_EXE>
0063
0064 /*
0065 Below is the gain in 13++ and 13+++
0066
0067 words per second performance: 5,974,513w/s
0068 word count: 4,582,451,898 of them 9,177,221 distinct
0069 Number of Trees(GREATER THE BETTER) : 2855919
0070 Number of Hash Collisions(Distinct WORDS - Number of Trees) : 6321302
0071
0072 words per second performance: 6,329,353w/s
0073 word count: 4,582,451,898 of them 9,177,221 distinct
0074 Number of Trees(GREATER THE BETTER) : 2958681
0075 Number of Hash Collisions(Distinct WORDS - Number of Trees) : 6218540
0076
0077 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.
0078 Thanks to Fowler/Noll/vo hash inventors.
0079 */
0080
0081 /*
0082 Let's see the supplementary-clash on Intel Pentium T3400 Merom-1M 2166MHz:
0083 Binary-Search-Trees vs B-Trees of order 3
0084
0085 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

```

```

0086 Leprechaun(Fast Greedy Word-Ripper), revision 13+++++, written by Svalqyatchx.
0087 Leprechaun: 'oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
0088 kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
0089 also the performance of a 3-way hash + 6,602,752 B-Trees of order 3.
0090 Size of input file with files for Leprechauning: 27
0091 Allocating memory 1863MB ... OK
0092 Size of Input TEXTUAL file: 146,973,879
0093 \; word count: 12,561,874 of them 12,561,874 distinct; Done: 64/64
0094 Bytes per second performance: 14,697,387B/s
0095 words per second performance: 1,256,187w/s
0096 Flushing unsorted words ...
0097 Time for making unsorted wordlist: 15 second(s)
0098 Deallocated memory in MB: 1863
0099 Allocated memory for words in MB: 141
0100 Allocated memory for pointers-to-words in MB: 48
0101 Sorting(with 'MultikeyQuickSortx26Sort' by J. Bentley and R. Sedgewick) ...
0102 Sort pass 26/26 ...
0103 Flushing sorted words ...
0104 Time for sorting unsorted wordlist: 14 second(s)
0105 Leprechaun: Done.
0106
0107 [An excerpt of Leprechaun.LOG:]
0108 Number of Trees(GREATER THE BETTER): 2786806
0109 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 58,935,172
0110 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,806
0111
0112 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
0113 Leprechaun(Fast Greedy Word-Ripper), revision 13+++++, written by Svalqyatchx.
0114 Leprechaun: 'oh, well, didn't you hear? Bigger is good, but jumbo is dear.'
0115 kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us,
0116 also the performance of a 3-way hash + 6,602,752 B-Trees of order 3.
0117 Size of input file with files for Leprechauning: 27
0118 Allocating memory 1863MB ... OK
0119 Size of Input TEXTUAL file: 146,973,879
0120 \; word count: 12,561,874 of them 12,561,874 distinct; Done: 64/64
0121 Bytes per second performance: 24,495,646B/s
0122 words per second performance: 2,093,645w/s
0123 Flushing unsorted words ...
0124 Time for making unsorted wordlist: 12 second(s)
0125 Deallocated memory in MB: 1863
0126 Allocated memory for words in MB: 141
0127 Allocated memory for pointers-to-words in MB: 48
0128 Sorting(with 'MultikeyQuickSortx26Sort' by J. Bentley and R. Sedgewick) ...
0129 Sort pass 26/26 ...
0130 Flushing sorted words ...
0131 Time for sorting unsorted wordlist: 14 second(s)
0132 Leprechaun: Done.
0133
0134 [An excerpt of Leprechaun.LOG:]
0135 Number of Trees(GREATER THE BETTER): 2786806
0136 Total Attempts to Find/Put WORDS into B-trees order 3: 18,534,910
0137
0138 C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST\type Leprechaun_vs_Wikipedia_en-WORDS.lst
0139 wikipedia-en-html.tar.wrd
0140
0141 C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST\dir Leprechaun_vs_Wikipedia_en-WORDS.*
0142 Volume in drive C is H320_Vol2
0143 Volume Serial Number is A094-FAE2
0144
0145 Directory of C:\WorkTemp\Leprechaun_r13++\Visual C++ Toolkit 2003\Leprechaun_step_1_PAIR-QUEST
0146
0147 09/14/2010 06:04 AM          27 Leprechaun_vs_wikipedia_en-WORDS.lst
0148 09/15/2010 02:51 AM    146,973,879 Leprechaun_vs_wikipedia_en-WORDS.wrd
0149                2 File(s)    146,973,906 bytes
0150                0 Dir(s)      965,787,648 bytes free
0151
0152 Conclusion:
0153 18,534,910/12,561,874=1.475 Average Attempts to Find/Put WORDS into B-trees order 3, not bad at all.
0154 */
0155
0156 // To do: must learn how to align, at last.
0157 /*
0158 Matt Mahoney ZPAQ fragment:
0159  * data; // allocated memory
0160  int offset;
0161  ...
0162  offset=64-int((long)data&63);
0163  data=(T*)((char*)data+offset); // adjust to 64 byte boundary
0164
0165 quicklz.c fragment:
0166 #define QLZ_ALIGNMENT_PADD 8
0167 unsigned char *scratch_aligned = (unsigned char *)scratch_compress + QLZ_ALIGNMENT_PADD - (((size_t)scratch_compress) % QLZ_ALIGNMENT_PADD);
0168 size_t *buffersize = (size_t *)scratch_aligned;
0169
0170 minilzo.c fragment:
0171 #define lzo_uintptr_t          unsigned long
0172 #define PTR(a)                ((lzo_uintptr_t) a)

```

```

0173 #define PTR_LINEAR(a) PTR(a)
0174 #define PTR_ALIGNED_4(a) ((PTR_LINEAR(a) & 3) == 0)
0175 */
0176
0177 //__declspec(align(64)) int BigArray[1024]; // windows syntax
0178 //or
0179 //int BigArray[1024] __attribute__((aligned(64))); // Linux syntax
0180
0181 #if defined(_WIN32_ENVIRONMENT_)
0182 __declspec(align(64))
0183 #else
0184 //__attribute__((aligned(64)));
0185 #endif /* defined(_WIN32_ENVIRONMENT_) */
0186
0187 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:
0188 //; 792 : hash32 = FNV_32A_OP32(hash32, *(UINT*)p&0xFFFF);
0189
0190 typedef unsigned int UINT;
0191 typedef unsigned int DWORD;
0192
0193 /*
0194 Enter-the-BESTer or an alchemical clash of pairs of primes.
0195
0196 When an x-bit hash where x < 16 and is not a power of 2 is needed,
0197 here comes 'FNV1A_Hash_4_OCTETS': a slightly tuned FNV1A hash for a huge(22,202,980) wordlist of latin-letters-words.
0198
0199 Two improvements for the generic(base) FNV1A hash:
0200 - first, better speed: by reducing 'imul' instructions when string is 4++ chars
0201 - second, better dispersion: by experimenting(superficially-lite test done, so far) with 'FNV1_32_PRIME'
0202
0203 Or more concretely:
0204 - For FNV1_32_INIT = 2166136261
0205 - Giving to 'FNV1_32_PRIME' all primes between 2 and 11987
0206 - Shifting by 16bits instead of 13bits, when 8192 slots are used
0207
0208 C code:
0209 typedef unsigned char u_int8_t;
0210 typedef unsigned long u_int32_t;
0211
0212 #define FNV1_32_INIT ((u_int32_t)2166136261)
0213 #define FNV1_32_PRIME ((u_int32_t)1607)
0214
0215 #define FNV_32A_OP(hash, octet) \
0216     (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * FNV1_32_PRIME)
0217
0218 #define FNV_32A_OP32(hash, octet) \
0219     (((u_int32_t)(hash) ^ (u_int32_t)(octet)) * FNV1_32_PRIME)
0220
0221 0800 // Invoking: FNV1A_Hash_4_OCTETS(wrd, wrdlen>>2) // = 0,1,2,3,4,5,6,7 [1..31]
0222 0801 int FNV1A_Hash_4_OCTETS(char *str, int wrdlen<QUADRUPLETS)
0223 0802 {
0224 0803 u_int32_t hash;
0225 0804 char *p;
0226 0805
0227 0806 hash = FNV1_32_INIT;
0228 0807 p=str;
0229 0808
0230 0809 // The goal of stage #1: to reduce number of 'imul's.
0231 0810
0232 0811 // Stage #1:
0233 0812 for (; wrdlen<QUADRUPLETS != 0; --wrdlen<QUADRUPLETS) {
0234 0813     hash = FNV_32A_OP32(hash, (unsigned long)*(long *)p); // mov edi, DWORD PTR [eax]
0235 0814     p=p+4; // add eax, 4
0236 0815 }
0237 0816
0238 0817 // Stage #2:
0239 0818 for (; *p; ++p) {
0240 0819     hash = FNV_32A_OP(hash, *p); // mov dl, BYTE PTR [eax]
0241 0820 }
0242 0821
0243 0822 //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
0244 0823 return ((hash>>16) ^ hash) & 8191; // 00.8191 i.e. 2^13=8192
0245 0824 }
0246
0247 Assembler code:
0248 _FNV1A_Hash_4_OCTETS PROC NEAR
0249 ; Line 812
0250 mov edx, DWORD PTR _wrdlen<QUADRUPLETS$[esp-4]
0251 test edx, edx
0252 mov eax, DWORD PTR _str$[esp-4]
0253 push esi
0254 mov esi, DWORD PTR _FNV1_32_PRIME
0255 mov ecx, -2128831035
0256 je SHORT $L1612
0257 push edi
0258 npad 7
0259 $L1610:

```

```

0260 ; Line 813
0261 mov edi, DWORD PTR [eax]
0262 xor edi, ecx
0263 imul edi, esi
0264 ; Line 814
0265 add eax, 4
0266 dec edx
0267 mov ecx, edi
0268 jne SHORT $L1610
0269 pop edi
0270 $L1612:
0271 ; Line 818
0272 mov dl, BYTE PTR [eax]
0273 test dl, dl
0274 je SHORT $L1619
0275 $L1617:
0276 ; Line 819
0277 movzx     edx, dl
0278 xor     edx, ecx
0279 imul    edx, esi
0280 inc     eax
0281 mov     ecx, edx
0282 mov     dl, BYTE PTR [eax]
0283 test    dl, dl
0284 jne     SHORT $L1617
0285 $L1619:
0286 ; Line 823
0287 mov     eax, ecx
0288 shr     eax, 16
0289 xor     eax, ecx
0290 and     eax, 8191
0291 pop     esi
0292 ; Line 824
0293 ret     0
0294 _FNV1A_Hash_4_OCTETS ENDP
0295
0296
0297 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.
0298
0299 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'.
0300
0301 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.
0302 This is critical for speed performance e.g. when 30,974,750,142 words, the case of wikipedia-en-html.tar, must be hashed.
0303 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.
0304 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.
0305
0306 Between 1..1166136247 there are 58,834,113 primes (inclusive).
0307 Between 1..16777619 there are 1,077,891 primes (inclusive).
0308 or 58834113*1077891 = 63,416,760,895,683 pairs or 2,010,932 years needed at one-pair-per-second rate.
0309
0310 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.
0311 Since the magnum corpus of words is static-enough, THE pair is worthy to be found.
0312
0313 It doesn't take a think-tank to see the superiority of FNV, Fowler/Noll/Vo did reveal a thing of beauty.
0314
0315 Performance of 'FNV1A_Hash_4_OCTETS': 10236 words/clock or 105 MB/s|3,549,448 used slots (best)
0316
0317 CASE #1: with 'if (strlen(backup[j]) != 0)' before each execution
0318 Performance of 'kuxHash3plus' aka '2in1': 8076 words/clock or 82 MB/s|3,410,463 used slots (worst)
0319 Performance of 'FNV1A_Hash': 8079 words/clock or 83 MB/s|3,527,916 used slots
0320 Performance of 'FNV1A_Hash_SHIFTless_XORless': 8109 words/clock or 83 MB/s|3,540,323 used slots
0321
0322 CASE #2: without 'if (strlen(backup[j]) != 0)' before each execution
0323 Performance of 'kuxHash3plus' aka '2in1': 11673 words/clock or 119 MB/s|3,410,463 used slots (worst)
0324 Performance of 'FNV1A_Hash': 11558 words/clock or 118 MB/s|3,527,916 used slots
0325 Performance of 'FNV1A_Hash_SHIFTless_XORless': 11570 words/clock or 118 MB/s|3,540,323 used slots
0326
0327 Note:
0328 The 'strlen' overhead(CASE #1) is necessary due to priorly(before hash invocation) needed len-of-string for 'FNV1A_Hash_4_OCTETS'.
0329 Almost always, that is the case, since parsing of incoming text must know length of words/lines/files.
0330 In case of not knowing this length: ((119-105)/105)*100% = 13% degradation is unacceptable.
0331 The 'strlen' is an awful brake.
0332 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.
0333 */
0334
0335 /*
0336 FNV1_32_PRIME: //?: 16777619
0337
0338 Above Binary-Search-Tree with MaxPEAK = 61 has NODES = 61 and LEAFs = 1
0339 words per second performance: 1,046,822w/s
0340 Input File with a list of TEXTUAL Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0341 Size of all TEXTUAL Files: 146,973,879
0342 word count: 12,561,874 of them 12,561,874 distinct
0343 Number of Trees(GREATER THE BETTER): 2775839
0344

```

0345 Above Binary-Search-Tree with MaxPEAK = 39 has NODES = 72 and LEAFs = 15
0346 words per second performance: 1,356,588w/s
0347 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0348 Size of all TEXTual Files: 415,982,896
0349 word count: 35,271,297 of them 22,202,980 distinct
0350 Number Of Trees(GREATER THE BETTER): 3539690
0351
0352 FNV1_32_PRIME: //3549448: 1607
0353
0354 Above Binary-Search-Tree with MaxPEAK = 61 has NODES = 61 and LEAFs = 1
0355 words per second performance: 1,046,822w/s
0356 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0357 Size of all TEXTual Files: 146,973,879
0358 word count: 12,561,874 of them 12,561,874 distinct
0359 Number Of Trees(GREATER THE BETTER): 2783970
0360
0361 Above Binary-Search-Tree with MaxPEAK = 38 has NODES = 50 and LEAFs = 11
0362 words per second performance: 1,410,851w/s
0363 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0364 Size of all TEXTual Files: 415,982,896
0365 word count: 35,271,297 of them 22,202,980 distinct
0366 Number Of Trees(GREATER THE BETTER): 3549395
0367
0368 FNV1_32_PRIME: //3550132: 175757909
0369
0370 Above Binary-Search-Tree with MaxPEAK = 60 has NODES = 60 and LEAFs = 1
0371 words per second performance: 966,298w/s
0372 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0373 Size of all TEXTual Files: 146,973,879
0374 word count: 12,561,874 of them 12,561,874 distinct
0375 Number Of Trees(GREATER THE BETTER): 2784479
0376
0377 Above Binary-Search-Tree with MaxPEAK = 39 has NODES = 64 and LEAFs = 12
0378 words per second performance: 1,410,851w/s
0379 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0380 Size of all TEXTual Files: 415,982,896
0381 word count: 35,271,297 of them 22,202,980 distinct
0382 Number Of Trees(GREATER THE BETTER): 3550115
0383
0384 FNV1_32_PRIME: //3550687: 201887489
0385
0386 Above Binary-Search-Tree with MaxPEAK = 60 has NODES = 60 and LEAFs = 1
0387 words per second performance: 966,298w/s
0388 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0389 Size of all TEXTual Files: 146,973,879
0390 word count: 12,561,874 of them 12,561,874 distinct
0391 Number Of Trees(GREATER THE BETTER): 2784377
0392
0393 Above Binary-Search-Tree with MaxPEAK = 40 has NODES = 55 and LEAFs = 11
0394 words per second performance: 1,356,588w/s
0395 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0396 Size of all TEXTual Files: 415,982,896
0397 word count: 35,271,297 of them 22,202,980 distinct
0398 Number Of Trees(GREATER THE BETTER): 3550528
0399
0400 FNV1_32_PRIME: //3550733: 172783361
0401
0402 Above Binary-Search-Tree with MaxPEAK = 59 has NODES = 59 and LEAFs = 1
0403 words per second performance: 1,046,822w/s
0404 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0405 Size of all TEXTual Files: 146,973,879
0406 word count: 12,561,874 of them 12,561,874 distinct
0407 Number Of Trees(GREATER THE BETTER): 2786362
0408
0409 Above Binary-Search-Tree with MaxPEAK = 38 has NODES = 70 and LEAFs = 17
0410 words per second performance: 1,410,851w/s
0411 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0412 Size of all TEXTual Files: 415,982,896
0413 word count: 35,271,297 of them 22,202,980 distinct
0414 Number Of Trees(GREATER THE BETTER): 3550746
0415
0416 FNV1_32_PRIME: //3550929: 204312319
0417
0418 Above Binary-Search-Tree with MaxPEAK = 61 has NODES = 61 and LEAFs = 1
0419 words per second performance: 966,298w/s
0420 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0421 Size of all TEXTual Files: 146,973,879
0422 word count: 12,561,874 of them 12,561,874 distinct
0423 Number Of Trees(GREATER THE BETTER): 2785581
0424
0425 Above Binary-Search-Tree with MaxPEAK = 37 has NODES = 55 and LEAFs = 12
0426 words per second performance: 1,356,588w/s
0427 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0428 Size of all TEXTual Files: 415,982,896
0429 word count: 35,271,297 of them 22,202,980 distinct
0430 Number Of Trees(GREATER THE BETTER): 3550886
0431
0432 Leprechaun_Microsoft.exe: FNV1_32_PRIME: //3551736: 107712257

0433
0434 Above Binary-Search-Tree with MaxPEAK = 61 has NODES = 61 and LEAFs = 1
0435 words per second performance: 1,046,822w/s
0436 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0437 Size of all TEXTual Files: 146,973,879
0438 word count: 12,561,874 of them 12,561,874 distinct
0439 Number Of Trees(GREATER THE BETTER): 2786515
0440
0441 Above Binary-Search-Tree with MaxPEAK = 36 has NODES = 64 and LEAFs = 15
0442 words per second performance: 1,356,588w/s
0443 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0444 Size of all TEXTual Files: 415,982,896
0445 word count: 35,271,297 of them 22,202,980 distinct
0446 Number Of Trees(GREATER THE BETTER): 3551744
0447
0448 Leprechaun_Intel.exe: FNV1_32_PRIME: //3551736: 107712257
0449
0450 Above Binary-Search-Tree with MaxPEAK = 61 has NODES = 61 and LEAFs = 1
0451 words per second performance: 1,256,187w/s
0452 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0453 Size of all TEXTual Files: 146,973,879
0454 word count: 12,561,874 of them 12,561,874 distinct
0455 Number Of Trees(GREATER THE BETTER): 2786515
0456
0457 Above Binary-Search-Tree with MaxPEAK = 36 has NODES = 64 and LEAFs = 15
0458 words per second performance: 1,603,240w/s
0459 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0460 Size of all TEXTual Files: 415,982,896
0461 word count: 35,271,297 of them 22,202,980 distinct
0462 Number Of Trees(GREATER THE BETTER): 3551744
0463
0464 wow: 1,603,240w/s vs 1,356,588w/s respectively Leprechaun_Intel.exe vs Leprechaun_Microsoft.exe, i.e. 18% betterment, no joke!
0465
0466 Alchemical search for best PRIME-PAIR revision uses next line:
0467 \$!ot = FNV1A_Hash_4_OCTETS(wrd, wrdlen>2)<<2; //13+++
0468 This revision uses next lines:
0469 if (wrdlen<19) // 4x4+3=19 i.e. last contains 7 clashes
0470 \$!ot = FNV1A_Hash_Granularity(wrd, wrdlen>2, 2)<<2; //13++++
0471 else // 2x8+4=20 i.e. first contains 6 clashes
0472 \$!ot = FNV1A_Hash_Granularity(wrd, wrdlen>3, 3)<<2; //13++++
0473
0474 ! 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.
0475
0476 Leprechaun.exe: FNV1_32_PRIME: //3551961: 428904191
0477
0478 Above Binary-Search-Tree with MaxPEAK = 60 has NODES = 60 and LEAFs = 1
0479 words per second performance: 966,298w/s
0480 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_en-WORDS.lst
0481 Size of all TEXTual Files: 146,973,879
0482 word count: 12,561,874 of them 12,561,874 distinct
0483 Number Of Trees(GREATER THE BETTER): 2786383
0484
0485 Above Binary-Search-Tree with MaxPEAK = 39 has NODES = 71 and LEAFs = 16
0486 words per second performance: 1,410,851w/s
0487 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0488 Size of all TEXTual Files: 415,982,896
0489 word count: 35,271,297 of them 22,202,980 distinct
0490 Number Of Trees(GREATER THE BETTER): 3551503
0491
0492 Leprechaun.exe: FNV1_32_PRIME: //3552103: 588411137
0493
0494 Above Binary-Search-Tree with MaxPEAK = 6 has NODES = 6 and LEAFs = 1
0495 Size of all TEXTual Files: 4,067,439
0496 word count: 358,798 of them 351,116 distinct
0497 Number Of Trees(GREATER THE BETTER): 310622
0498 Total Number Of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 310,622
0499
0500 Above Binary-Search-Tree with MaxPEAK = 60 has NODES = 60 and LEAFs = 1
0501 Size of all TEXTual Files: 146,973,879
0502 word count: 12,561,874 of them 12,561,874 distinct
0503 Number Of Trees(GREATER THE BETTER): 2786485
0504 Total Number Of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,485
0505
0506 Above Binary-Search-Tree with MaxPEAK = 39 has NODES = 62 and LEAFs = 15
0507 Size of all TEXTual Files: 415,982,896
0508 word count: 35,271,297 of them 22,202,980 distinct
0509 Number Of Trees(GREATER THE BETTER): 3551956
0510 Total Number Of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,131
0511
0512 Leprechaun.exe: FNV1_32_PRIME: //3552039: 602173697 !!!GOODEST so far!!!
0513
0514 Above Binary-Search-Tree with MaxPEAK = 6 has NODES = 6 and LEAFs = 1
0515 Size of all TEXTual Files: 4,067,439
0516 word count: 358,798 of them 351,116 distinct
0517 Number Of Trees(GREATER THE BETTER): 310948
0518 Total Number Of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 310,948
0519

```

0520 Above Binary-Search-Tree with MaxPEAK = 63 has NODES = 63 and LEAFs = 1
0521 Size of all TEXTUAL Files: 146,973,879
0522 Word count: 12,561,874 of them 12,561,874 distinct
0523 Number Of Trees(GREATER THE BETTER): 2786806
0524 Total Number Of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 2,786,806
0525
0526 Above Binary-Search-Tree with MaxPEAK = 36 has NODES = 52 and LEAFs = 9
0527 Input File with a list of TEXTUAL Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
0528 Size of all TEXTUAL Files: 415,982,896
0529 Word count: 35,271,297 of them 22,202,980 distinct
0530 Number Of Trees(GREATER THE BETTER): 3552296
0531 Total Number Of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,899
0532
0533 Between 1 and 602392027 at step 100 following FNV1_32_PRIMES(for FNV1_32_INIT=2166136261) give(FNV1A_Hash_4_OCTETS) dispersion:
0534 3550022: 423779327
0535 3550028: 513793537
0536 3550053: 434840321
0537 3550067: 437062229
0538 3550080: 420344321
0539 3550090: 304777471
0540 3550097: 496547839
0541 3550129: 390809599
0542 3550132: 175757909
0543 3550163: 353712127
0544 3550231: 334434817
0545 3550237: 272789761
0546 3550247: 590341121
0547 3550255: 358814207
0548 3550277: 437182721
0549 3550326: 521795327
0550 3550347: 311867393
0551 3550447: 456137729
0552 3550458: 418208767
0553 3550516: 602048767
0554 3550525: 513597697
0555 3550526: 347283199
0556 3550528: 598773503
0557 3550592: 598139137
0558 3550598: 242448127
0559 3550611: 571481087
0560 3550628: 457012993
0561 3550664: 482822143
0562 3550666: 249098753
0563 3550687: 201887489
0564 3550702: 489976063
0565 3550710: 272961023
0566 3550733: 172783361
0567 3550734: 431562497
0568 3550929: 204312319
0569 3550984: 562853633
0570 3550991: 551362303
0571 3551359: 332820737
0572 3551484: 354126079
0573 3551514: 407138561
0574 3551523: 442058753
0575 3551701: 449230849
0576 3551736: 107712257
0577 3551961: 428904191
0578 3552039: 602173697
0579 3552103: 588411137
0580 */
0581
0582 // Windows: ~~~~~
0583 // _CRTIMP size_t __cdecl fread(void *, size_t, size_t, FILE *);
0584 // _CRTIMP size_t __cdecl fwrite(const void *, size_t, size_t, FILE *);
0585 // _CRTIMP int __cdecl fgetpos(FILE *, fpos_t *);
0586 // _CRTIMP int __cdecl fsetpos(FILE *, const fpos_t *);
0587
0588 // _CRTIMP __int64 __cdecl _lseeki64(int, __int64, int);
0589 // _CRTIMP __int64 __cdecl _telli64(int);
0590 // _CRTIMP __int64 __cdecl _filelengthi64(int);
0591 // above 3 are in 'io.h'
0592
0593 // _CRTIMP int __cdecl fseek(FILE *, long, int);
0594 // _CRTIMP long __cdecl ftell(FILE *);
0595 // _CRTIMP int __cdecl fclose(FILE *);
0596
0597 // #ifndef _SIZE_T_DEFINED
0598 // #ifdef _WIN64
0599 // typedef unsigned __int64 size_t;
0600 // #else
0601 // typedef _w64 unsigned int size_t;
0602 // #endif
0603 // #define _SIZE_T_DEFINED
0604 // #endif
0605
0606 // typedef __int64 fpos_t;
0607

```

```

0608 // Linux: ~~~~~
0609 // size_t fread (void *data, size_t size, size_t count, FILE *stream)
0610 // size_t fwrite (const void *data, size_t size, size_t count, FILE *stream)
0611 // int fgetpos (FILE *stream, fpos_t *position)
0612 // int fsetpos (FILE *stream, const fpos_t *position)
0613
0614 // FILE * fopen64 (const char *filename, const char *opentype)
0615 // int fseeko64 (FILE *stream, off64_t offset, int whence)
0616 // off64_t ftello64 (FILE *stream)
0617 // int fclose (FILE *stream)
0618
0619 // off_t lseek (int filesd, off_t offset, int whence)
0620 // above 1 is in 'unistd.h'
0621
0622
0623 // ===== MUST work both for windows and Linux =====
0624 #define _WIN32_ENVIRONMENT_
0625 // #define _POSIX_ENVIRONMENT_
0626
0627
0628 #ifndef NULL
0629 #ifdef __cplusplus
0630 #define NULL 0
0631 #else
0632 #define NULL ((void*)0)
0633 #endif
0634 #endif
0635
0636 // To do #1: Put this 31 in MAXWl: 'int MAXWl = 31;'
0637 // To do #2: No need of flushing unsorted words to file: make backup[] array
0638 // instead of writing. And mostly sort 26 times!
0639 // HEAVY BUG in r.7: unsigned long Hi1(unsigned long n)
0640 // is NOT identical with
0641 // unsigned long GRMBLhi1[32]; // 00 not used, only 01..31
0642 // BECAUSE DUMBEST DUMB Array GRMBLhi1 expects 'int' not
0643 // 'unsigned long' !!!
0644
0645 #include <stdio.h>
0646 #include <ctype.h>
0647 #include <time.h>
0648 #if defined(_WIN32_ENVIRONMENT_)
0649 #include <io.h> // needed for windows' 'lseeki64' and 'telli64'
0650 // Above line must be commented in order to compile with Intel C compiler: an error "can't find io.h" occurs.
0651 #else
0652 #endif /* defined(_WIN32_ENVIRONMENT_) */
0653
0654 typedef unsigned char char_t;
0655 typedef char_t *string;
0656
0657 clock_t clocks1, clocks2;
0658 int bozan;
0659
0660 typedef unsigned char u_int8_t; //FNV only
0661 typedef unsigned long u_int32_t; //FNV only
0662 typedef unsigned long long u_int64_t; //FNV only
0663
0664 // SINHA fragment[
0665
0666 #define swapKAZE(a, b) { t = *(a); *(a) = *(b); *(b) = t; }
0667
0668 static void InsertSortKAZE(string *a, int n, int d) //void insort(unsigned char **a, int n, int d)
0669 { string *pi, *pj, s, t; //unsigned char **pi, **pj, *s, *t;
0670 for (pi = a + 1; --n > 0; pi++)
0671 for (pj = pi; pj > a; pj--) {
0672 /* Inline strcmp: break if *(pj-1) <= *pj */
0673 for (s=(pj-1)+d, t=*pj+d; *s==*t && *s!=0; s++, t++)
0674 ;
0675 if (*s <= *t)
0676 break;
0677 swapKAZE(pj, pj-1);
0678 }
0679 }
0680
0681 //int cmpit(unsigned char **h1, unsigned char **h2)
0682 //{
0683 // return( strcmp(*h1, *h2) );
0684 //}
0685
0686 //int scmp( unsigned char *s1, unsigned char *s2 )
0687 //{
0688 // while( *s1 != '\0' && *s1 == *s2 )
0689 // {
0690 // s1++;
0691 // s2++;
0692 // }
0693 // return( *s1-*s2 );
0694 //}
0695

```

```

0696 //static void simplesort(string a[], int n, int b)
0697 //{
0698 //  int i, j;
0699 //  string tmp;
0700 //  for (i = 1; i < n; i++)
0701 //    for (j = i; j > 0 && strcmp(a[j-1]+b, a[j]+b) > 0; j--)
0702 //      { tmp = a[j]; a[j] = a[j-1]; a[j-1] = tmp; }
0703 //}
0704 //}
0705
0706 // SINHA fragment]
0707
0708 // mkqsort.c BEGIN *****
0709 /*
0710 Multikey quicksort, a radix sort algorithm for arrays of character
0711 strings by Bentley and Sedgwick.
0712
0713 J. Bentley and R. Sedgwick. Fast algorithms for sorting and
0714 searching strings. In Proceedings of 8th Annual ACM-SIAM Symposium
0715 on Discrete Algorithms, 1997.
0716
0717 http://www.CS.Princeton.EDU/~rs/strings/index.html
0718
0719 The code presented in this file has been tested with care but is
0720 not guaranteed for any purpose. The writer does not offer any
0721 warranties nor does he accept any liabilities with respect to
0722 the code.
0723
0724 Ranjan Sinha, 1 jan 2003.
0725
0726 School of Computer Science and Information Technology,
0727 RMIT University, Melbourne, Australia
0728 rsinha@cs.rmit.edu.au
0729
0730 */
0731
0732 //include "sortstring.h"
0733
0734 /* MULTIKEY QUICKSORT */
0735
0736 #ifndef min
0737 #define min(a, b) ((a)<=(b) ? (a) : (b))
0738 #endif
0739
0740
0741 // ----- BTREE [
0742 #define false -1
0743 #define true 0
0744
0745 struct nodeBTREE {
0746 int data;
0747 struct nodeBTREE* left;
0748 struct nodeBTREE* right;
0749 };
0750
0751 // ----- BTREE ]
0752
0753
0754 /* ssort2 -- Faster Version of Multikey Quicksort */
0755
0756 void vecswap2(unsigned char **a, unsigned char **b, int n)
0757 { while (n-- > 0) {
0758   unsigned char *t = *a;
0759   *a++ = *b;
0760   *b++ = t;
0761 }
0762 }
0763
0764 #define swap2(a, b) { t = *(a); *(a) = *(b); *(b) = t; }
0765 #define ptr2char(i) (*(i) + depth)
0766
0767 unsigned char **med3func(unsigned char **a, unsigned char **b, unsigned char **c, int depth)
0768 { int va, vb, vc;
0769   if ((va=ptr2char(a)) == (vb=ptr2char(b)))
0770     return a;
0771   if ((vc=ptr2char(c)) == va || vc == vb)
0772     return c;
0773   return va < vb ?
0774     (vb < vc ? b : (va < vc ? c : a)) :
0775     (vb > vc ? b : (va < vc ? a : c));
0776 }
0777 #define med3(a, b, c) med3func(a, b, c, depth)
0778
0779 void inssort(unsigned char **a, int n, int d)
0780 { unsigned char **pi, **pj, **s, **t;
0781   for (pi = a + 1; --n > 0; pi++)
0782     for (pj = pi; pj > a; pj--) {
0783       /* In-line strcmp: break if *(pj-1) <= *pj */

```

```

0784         for (s=(pj-1)+d, t=*pj+d; *s==*t && *s!=0; s++, t++)
0785           ;
0786         if (*s <= *t)
0787           break;
0788         swap2(pj, pj-1);
0789       }
0790     }
0791
0792 void mkqsort(unsigned char **a, int n, int depth)
0793 { int d, r, partval;
0794   unsigned char **pa, **pb, **pc, **pd, **pl, **pm, **pn, **t;
0795   if (n < 20) {
0796     inssort(a, n, depth);
0797     return;
0798   }
0799   pl = a;
0800   pm = a + (n/2);
0801   pn = a + (n-1);
0802   if (n > 30) { /* On big arrays, pseudomedian of 9 */
0803     d = (n/8);
0804     pl = med3(pl, pl+d, pl+2*d);
0805     pm = med3(pm-d, pm, pm+d);
0806     pn = med3(pn-2*d, pn-d, pn);
0807   }
0808   pm = med3(pl, pm, pn);
0809   swap2(a, pm);
0810   partval = ptr2char(a);
0811   pa = pb = a + 1;
0812   pc = pd = a + n-1;
0813   for (;;) {
0814     while (pb <= pc && (r = ptr2char(pb)-partval) <= 0) {
0815       if (r == 0) { swap2(pa, pb); pa++; }
0816       pb++;
0817     }
0818     while (pb <= pc && (r = ptr2char(pc)-partval) >= 0) {
0819       if (r == 0) { swap2(pc, pd); pd--; }
0820       pc--;
0821     }
0822     if (pb > pc) break;
0823     swap2(pb, pc);
0824     pb++;
0825     pc--;
0826   }
0827   pn = a + n;
0828   r = min(pa-a, pb-pa); vecswap2(a, pb-r, r);
0829   r = min(pd-pc, pn-pd-1); vecswap2(pb, pn-r, r);
0830   if ((r = pb-pa) > 1)
0831     mkqsort(a, r, depth);
0832   if (ptr2char(a+r) != 0)
0833     mkqsort(a+r, pa-a+pn-pd-1, depth+1);
0834   if ((r = pd-pc) > 1)
0835     mkqsort(a+n-r, r, depth);
0836 }
0837
0838 void mkqsort_main(unsigned char **a, int n) { mkqsort(a, n, 0); }
0839 // mkqsort.c END *****
0840
0841 // why Sinha uses int instead of long?!!
0842 static int readlines(char *file_name, string **lines)
0843 {
0844   int nlines = 0;
0845   size_t size;
0846   FILE *in_file;
0847   string basep, cur, next;
0848   string *ASBackup;
0849
0850   if (!(in_file = fopen(file_name, "rb"))) {
0851     printf("Leprechaun: Can't open file %s \n", file_name);
0852     exit(-1);
0853   }
0854   fseek(in_file, 0, SEEK_END);
0855   size = ftell(in_file);
0856   fseek(in_file, 0, SEEK_SET);
0857   if (!(basep = (string) malloc(size*sizeof(char_t)))) return -1;
0858   printf("Allocated memory for words in MB: %lu\n", ((size*sizeof(char_t))>>20)+1);
0859   if (fread(basep, 1, size, in_file) < size) {
0860     printf("Leprechaun: Can't read file %s \n", file_name);
0861     exit(-1);
0862   }
0863   fclose(in_file);
0864
0865 // GET nlines:
0866 cur = basep;
0867 while (cur < basep + size) {
0868   next = cur;
0869   while ((next < basep + size) && (*next != '\n')) {next++;}
0870   *--next = '\0'; // This is ala DOS i.e. Windows
0871   // 1310 not 10(\n=10)

```

```

0872     cur = next + 2;
0873     nlines++;
0874 }
0875
0876 // printf("%lu\n", (unsigned long)*lines); -> backup = *lines = 0
0877 ASbackup = (string *)malloc( nlines*sizeof(string) ); // sizeof(string) is 4
0878 if( ASbackup == NULL )
0879 { puts( "Leprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
0880 printf( "Allocated memory for pointers-to-words in MB: %lu\n", ((nlines*sizeof(string))>>20)+1 );
0881 *lines = ASbackup;
0882 //printf("%lu\n", (unsigned long)*lines); -> backup = *lines = ASbackup = 6946888
0883
0884 // Upload nlines times:
0885 nlines = 0;
0886 cur = basep;
0887 while (cur < basep + size) {
0888     next = cur;
0889     while ((next < basep + size) && (*next != '\n')) {next++;}
0890     *--next = '\0'; // This is ala DOS i.e. Windows
0891     // 1310 not 10(\n=10)
0892     ASbackup[nlines] = cur;
0893     cur = next + 2;
0894     nlines++;
0895 }
0896 return nlines;
0897 }
0898
0899 void x64toakAZE ( /* stdcall is faster and smaller... Might as well use it for the helper. */
0900     unsigned long long val,
0901     char *buf,
0902     unsigned radix,
0903     int is_neg
0904 )
0905 {
0906     char *p; // pointer to traverse string */
0907     char *firstdig; // pointer to first digit */
0908     char temp; // temp char */
0909     unsigned digval; // value of digit */
0910
0911     p = buf;
0912
0913     if ( is_neg )
0914     {
0915         *p++ = '-'; // negative, so output '-' and negate */
0916         val = (unsigned long long)(-(long long)val);
0917     }
0918
0919     firstdig = p; // save pointer to first digit */
0920
0921     do {
0922         digval = (unsigned) (val % radix);
0923         val /= radix; // get next digit */
0924
0925         /* convert to ascii and store */
0926         if (digval > 9)
0927             *p++ = (char) (digval - 10 + 'a'); /* a letter */
0928         else
0929             *p++ = (char) (digval + '0'); /* a digit */
0930     } while (val > 0);
0931
0932     /* We now have the digit of the number in the buffer, but in reverse
0933     order. Thus we reverse them now. */
0934
0935     *p-- = '\0'; // terminate string; p points to last digit */
0936
0937     do {
0938         temp = *p;
0939         *p = *firstdig;
0940         *firstdig = temp; // swap *p and *firstdig */
0941         --p;
0942         ++firstdig; // advance to next two digits */
0943     } while (firstdig < p); /* repeat until halfway */
0944 }
0945
0946 /* Actual functions just call conversion helper with neg flag set correctly,
0947 and return pointer to buffer. */
0948
0949 char * _i64toakAZE (
0950     long long val,
0951     char *buf,
0952     int radix
0953 )
0954 {
0955     x64toakAZE((unsigned long long)val, buf, radix, (radix == 10 && val < 0));
0956     return buf;
0957 }
0958
0959 char * _ui64toakAZE (

```

```

0960     unsigned long long val,
0961     char *buf,
0962     int radix
0963 )
0964 {
0965     x64toakAZE(val, buf, radix, 0);
0966     return buf;
0967 }
0968
0969 char * _ui64toakAZEzerocomma (
0970     unsigned long long val,
0971     char *buf,
0972     int radix
0973 )
0974 {
0975     char *p;
0976     char temp;
0977     int txpman;
0978     int pxnman;
0979     x64toakAZE(val, buf, radix, 0);
0980     p = buf;
0981     do {
0982     } while (*++p != '\0');
0983     p--; // p points to last digit
0984     // buf points to first digit
0985     buf[26] = 0;
0986     txpman = 1;
0987     pxnman = 0;
0988     do
0989     { if (buf <= p)
0990     { temp = *p;
0991       buf[26-txpman] = temp; pxnman++;
0992       p--;
0993       if (pxnman % 3 == 0)
0994       { txpman++;
0995         buf[26-txpman] = (char) ('.','.');
0996       }
0997     }
0998     else
0999     { buf[26-txpman] = (char) ('0'); pxnman++;
1000     if (pxnman % 3 == 0)
1001     { txpman++;
1002       buf[26-txpman] = (char) ('.','.');
1003     }
1004     }
1005     txpman++;
1006     } while (txpman <= 26);
1007     return buf;
1008 }
1009
1010 char * _ui64toakAZEcomma (
1011     unsigned long long val,
1012     char *buf,
1013     int radix
1014 )
1015 {
1016     char *p;
1017     char temp;
1018     int txpman;
1019     int pxnman;
1020     x64toakAZE(val, buf, radix, 0);
1021     p = buf;
1022     do {
1023     } while (*++p != '\0');
1024     p--; // p points to last digit
1025     // buf points to first digit
1026     buf[26] = 0;
1027     txpman = 1;
1028     pxnman = 0;
1029     while (buf <= p)
1030     { temp = *p;
1031       buf[26-txpman] = temp; pxnman++;
1032       p--;
1033       if (pxnman % 3 == 0 && buf <= p)
1034       { txpman++;
1035         buf[26-txpman] = (char) ('.','.');
1036       }
1037     }
1038     txpman++;
1039     return buf+26-(txpman-1);
1040 }
1041
1042 unsigned char KuxHash(char *str)
1043 { unsigned char h = 0;
1044   int max31 = 0;
1045   //while (*str)
1046   while (str[max31])
1047   { h = h ^ str[max31++];

```

```

1048 //h = h ^ *str++; // I am not sure 'str' is returned changed after return?!
1049 }
1050 return h; // 00..255 i.e. 2^8=256
1051 }
1052
1053 int KuxHash2(char *str)
1054 { int h = 0;
1055 unsigned long h2 = 0; // must be long: 31*'z'=31*122
1056 int max31 = 0;
1057 while (str[max31])
1058 { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1059 //h2 = h2 + str[max31++]; // [113s]
1060 h2 = h2 + max31 * str[max31++];
1061 }
1062 h<<4; // 00..15 i.e. 2^4=16
1063 //h = h | (str[0] ^ str[max31-1]); // [111s] a..z: each XOR each gives 00..31
1064 h = h | (h2%((1<<4)-1));
1065 return h; // 00..4095 i.e. 2^12=4096
1066 }
1067
1068 //OSHO test - Attempts to Find/Put a WORD into linked list count: 32,011,937
1069 int KuxHash3(char *str)
1070 { int h = 0;
1071 unsigned long h2 = 0; // must be long: 31*'z'=31*122
1072 int max31 = 0;
1073 while (str[max31])
1074 { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1075 //h2 = h2 + str[max31++]; // [113s]
1076 h2 = h2 + str[max31++] * (max31+1);
1077 }
1078 // Result is: 7bits in 'h' and 32bits in 'h2'.
1079
1080 //printf("%s:\n ",str);
1081 //printf("%d ",h);
1082 h<<6; // 00..15 i.e. 00-05+7bits=13bits
1083 //printf("%d ",h);
1084 //printf("%d ",h2);
1085 //h = h | (str[0] ^ str[max31-1]); // [111s] a..z: each XOR each gives 00..31
1086 h = h | (h2%((1<<6)-1)); // 64-1=63=9*7; 61 is prime
1087 //printf("%d\n",h);
1088 return h; // 00..8191 i.e. 2^13=8192
1089 }
1090
1091 //OSHO test - Attempts to Find/Put a WORD into a linked list count: 31,927,285
1092 int KuxHash3plus(char *str)
1093 { int h = 0;
1094 unsigned long h2 = 0; // must be long: 31*'z'=31*122
1095 int max31 = 0;
1096 while (str[max31])
1097 { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1098 //h2 = h2 + str[max31++]; // [113s]
1099 h2 = h2 + str[max31++] * (max31+1);
1100 }
1101 // Result is: 7bits in 'h' and 32bits in 'h2'.
1102
1103 //printf("%s:\n ",str);
1104 //printf("%d ",h);
1105 // a in ASCII is 097 = 0110 0001
1106 // z in ASCII is 122 = 0111 1010
1107 // Above two lines show that bits 8-7-6 are always 0-1-1 so need for low 5 bits.
1108 //h<<8; // 00..15 i.e. 5bits + 00-07bits=13bits
1109 //printf("%d ",h);
1110 //printf("%d ",h2);
1111 //h = h | (str[0] ^ str[max31-1]); // [111s] a..z: each XOR each gives 00..31
1112 h = ((h<<8) | (h2%251)) && 8191; // 251 prime
1113 //printf("%d\n",h);
1114 return h; // 00..8191 i.e. 2^13=8192
1115 }
1116
1117 /*
1118 PUBLIC _KuxHash3plus
1119 ; Function compile flags: /Ogty
1120 _TEXT SEGMENT
1121 _str$ = 8 ; size = 4
1122 _KuxHash3plus PROC NEAR
1123 ; Line 511
1124 mov ecx, DWORD PTR _str$[esp-4]
1125 mov d1, BYTE PTR [ecx]
1126 push esi
1127 xor esi, esi
1128 xor eax, eax
1129 test d1, d1
1130 je SHORT $L1561
1131 push ebx
1132 push edi
1133 mov edi, 1
1134 sub edi, ecx
1135 npad 8

```

```

1136 $L1560:
1137 ; Line 512
1138 movsx     edx, BYTE PTR [ecx]
1139 ; Line 514
1140 lea ebx, DWORD PTR [edi+ecx]
1141 imul ebx, edx
1142 xor esi, edx
1143 mov d1, BYTE PTR [ecx+1]
1144 add eax, ebx
1145 inc ecx
1146 test d1, d1
1147 jne SHORT $L1560
1148 pop edi
1149 pop ebx
1150 $L1561:
1151 ; Line 527
1152 xor edx, edx
1153 mov ecx, 251 ; 000000fbh
1154 div ecx
1155 shl esi, 8
1156 mov eax, edx
1157 ; Line 529
1158 or eax, esi
1159 and eax, 8191 ; 00001ffffh
1160 pop esi
1161 ; Line 530
1162 ret 0
1163 _KuxHash3plus ENDP
1164 _TEXT ENDS
1165 */
1166
1167 //OSHO test - Attempts to Find/Put a WORD into a linked list count: 32,021,975
1168 int KuxHash4(char *str)
1169 {
1170 int h2 = 0;
1171 for (; *str != 0; str++) {
1172 //h2 = (127*h2 + *str) % (8192-1); // 2^13-1 = 8191 is Mersenne prime
1173 h2 = ((h2<<7) + *str) % (8192-1); // 2^13-1 = 8191 is Mersenne prime
1174 }
1175
1176 return h2; // 00..8191 i.e. 2^13=8192
1177 }
1178
1179 /*
1180 int hash(char *v, int M)
1181 { int h = 0, a = 127;
1182 for (; *v != 0; v++)
1183 h = (a*h + *v) % M;
1184 return h;
1185 }
1186
1187 int hashU(char *v, int M)
1188 { int h, a = 31415, b = 27183;
1189 for (h = 0; *v != 0; v++, a = a*b % (M-1))
1190 h = (a*h + *v) % M;
1191 return (h < 0) ? (h + M) : h;
1192 }
1193 */
1194
1195 // Kaze: My appreciation of FNV is far beyond C code optimization, it is alchemical, and why not, magical.
1196
1197 /*
1198 FNV hash history
1199 The basis of the FNV hash algorithm was taken from an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee
1200 by Glenn Fowler and Phong Vo back in 1991. In a subsequent ballot round: Landon Curt Noll improved on their algorithm.
1201 Some people tried this hash and found that it worked rather well. In an Email message to Landon, they named it
1202 the "Fowler/Noll/Vo" or FNV hash.
1203 FNV hashes are designed to be fast while maintaining a low collision rate. The FNV speed allows one to quickly hash
1204 lots of data while maintaining a reasonable collision rate. The high dispersion of the FNV hashes makes them well suited
1205 for hashing nearly identical strings such as URLs, hostnames, filenames, text, IP addresses, etc.
1206 */
1207
1208 /* NOTE: u_int64_t is a 64 bit unsigned type */
1209 /* NOTE: u_int32_t is a 32 bit unsigned type */
1210 /* NOTE: u_int16_t is a 16 bit unsigned type */
1211 /* NOTE: u_int8_t is a 8 bit unsigned type */
1212
1213 //typedef unsigned char u_int8_t; //FNV only
1214 //typedef unsigned long u_int32_t; //FNV only
1215 //typedef unsigned long long u_int64_t; //FNV only
1216
1217 // 32 bit FNV_prime = 2^24 + 2^8 + 0x93 = 16777619
1218 // 64 bit FNV_prime = 2^40 + 2^8 + 0xb3 = 1099511628211
1219
1220 // 32 bit offset_basis = 2166136261
1221 // 64 bit offset_basis = 14695981039346656037
1222
1223 #define FNV1_64_INIT ((u_int64_t)14695981039346656037)

```



```

1224 #define FNV1_64_PRIME ((u_int64_t)1099511628211)
1225 #define FNV1_32_INIT ((u_int32_t)2166136261)
1226 #define FNV1_32_PRIME ((u_int32_t)602173697)
1227 // FNV1A_Hash_4_OCTETS gives dispersion as follows:
1228 //3549448: 1607
1229 //3549669: 171072511
1230 //3550710: 272961023
1231 //3550733: 172783361
1232 //3550734: 431562497
1233 //3550929: 204312319
1234 //3550984: 562853633
1235 //3550991: 551362303
1236 //3551359: 332820737
1237 //3551484: 354126079
1238 //3551514: 407138561
1239 //3551523: 442058753
1240 //3551701: 449230849
1241 //3551736: 107712257
1242 //3551961: 428904191
1243 //3552039: 602173697
1244 //3552103: 588411137
1245
1246 #define FNV_64A_OP(hash, octet) \
1247     (((u_int64_t)(hash) ^ (u_int8_t)(octet)) * FNV1_64_PRIME)
1248
1249 #define FNV_64A_OP64(hash, octet) \
1250     (((u_int64_t)(hash) ^ (u_int64_t)(octet)) * FNV1_64_PRIME)
1251
1252 #define FNV_32A_OP_GENERIC(hash, octet) \
1253     (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * 16777619)
1254
1255 #define FNV_32A_OP(hash, octet) \
1256     (((u_int32_t)(hash) ^ (u_int8_t)(octet)) * FNV1_32_PRIME)
1257
1258 #define FNV_32A_OP_MULless_core(hash, octet) \
1259     ((u_int32_t)(hash) ^ (u_int8_t)(octet))
1260
1261 #define FNV_32A_OP_MULless(hash, octet) \
1262     ((FNV_32A_OP_MULless_core(hash, octet)<<5) - FNV_32A_OP_MULless_core(hash, octet))
1263
1264 #define FNV_32A_OP32(hash, octet) \
1265     (((u_int32_t)(hash) ^ (u_int32_t)(octet)) * FNV1_32_PRIME)
1266
1267 #define FNV_32A_OP64(hash, octet) \
1268     (((u_int64_t)(hash) ^ (u_int64_t)(octet)) * FNV1_32_PRIME)
1269
1270 #define FNV_32A_OP32_MULless_core(hash, octet) \
1271     ((u_int32_t)(hash) ^ (u_int32_t)(octet))
1272
1273 #define FNV_32A_OP32_MULless(hash, octet) \
1274     ((FNV_32A_OP32_MULless_core(hash, octet)<<5) - FNV_32A_OP32_MULless_core(hash, octet))
1275
1276
1277 // Invoking: FNV1A_Hash_4_OCTETS_31(wrd, wrdlen>2) // = 0,1,2,3,4,5,6,7 [1..31]
1278 int FNV1A_Hash_4_OCTETS_31(char *str, int wrdlen_QUADRUPLETS)
1279 {
1280     u_int32_t hash;
1281     char *p;
1282
1283     hash = FNV1_32_INIT;
1284     p=str;
1285
1286     // The goal of stage #1: to reduce number of 'imul's in fact to reduce loops.
1287
1288     // Stage #1:
1289     for (; wrdlen_QUADRUPLETS != 0; --wrdlen_QUADRUPLETS) {
1290         hash = FNV_32A_OP32_MULless(hash, (unsigned long)*(long *p)); // mov edi, DWORD PTR [eax]
1291         p=p+4; // add eax, 4
1292     }
1293
1294     // Stage #2:
1295     for (; *p; ++p) {
1296         hash = FNV_32A_OP_MULless(hash, *p); // mov d1, BYTE PTR [ecx]
1297     }
1298
1299     //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1300     return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1301 }
1302
1303 // Invoking: FNV1A_Hash_4_OCTETS(wrd, wrdlen>2) // = 0,1,2,3,4,5,6,7 [1..31]
1304 int FNV1A_Hash_4_OCTETS(char *str, int wrdlen_QUADRUPLETS)
1305 {
1306     u_int32_t hash;
1307     char *p;
1308
1309     hash = FNV1_32_INIT;
1310     p=str;

```

```

1312
1313 // The goal of stage #1: to reduce number of 'imul's.
1314
1315 // Stage #1:
1316 for (; wrdlen_QUADRUPLETS != 0; --wrdlen_QUADRUPLETS) {
1317     hash = FNV_32A_OP32(hash, (unsigned long)*(long *p)); // mov edi, DWORD PTR [eax]
1318     p=p+4; // add eax, 4
1319 }
1320
1321 // Stage #2:
1322 for (; *p; ++p) {
1323     hash = FNV_32A_OP(hash, *p); // mov d1, BYTE PTR [ecx]
1324 }
1325
1326 //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1327     return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1328 }
1329
1330 /*
1331 Results for 'FNV1A_Hash_8_OCTETS':
1332 Bytes per second performance: 23,110,160B/s
1333 words per second performance: 1,959,516W/s
1334 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
1335 Size of all TEXTual Files: 415,982,896
1336 word count: 35,271,297 of them 22,202,980 distinct
1337 Number of Files: 8
1338 Number of Lines: 35271297
1339 Allocated memory in MB: 1950
1340 Number of Trees(GREATER THE BETTER): 3419429
1341 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 51%
1342 Number of Hash Collisions(Distinct WORDS - Number of Trees): 18783551
1343 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '1,119'
1344 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 268,085,505
1345 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 7,690,615
1346 Perfectly-Balanced-Binary-Search-Tree for MaxNODES = 2,622 must have PEAK = 12 = rounding down of integer (1+1b(2,622))
1347 Binary-Search-Tree(1st out of 1) with MaxNODES = 2,622 has PEAK = 592 and LEAFs = 689
1348 Binary-Search-Tree(1st out of 1) with MaxPEAK = '1,119' has NODES = 1,537 and LEAFs = 287
1349 Binary-Search-Tree(1st out of 1) with MaxLEAFs = 731 has NODES = 2,517 and PEAK = 448
1350 */
1351 // Invoking: FNV1A_Hash_8_OCTETS(wrd, wrdlen>3) // = 0,1,2,3 [1..31]
1352 int FNV1A_Hash_8_OCTETS(char *str, int wrdlen_OCTETS)
1353 {
1354     u_int32_t hash;
1355     char *p;
1356
1357     hash = FNV1_32_INIT;
1358     p=str;
1359
1360     // The goal of stage #1: to reduce number of 'imul's.
1361
1362     // Stage #1:
1363     for (; wrdlen_OCTETS != 0; --wrdlen_OCTETS) {
1364         hash = FNV_32A_OP64(hash, (unsigned long long)*(long *p)); // mov edi, DWORD PTR [eax]
1365         p=p+8; // add eax, 4
1366     }
1367
1368     // Stage #2:
1369     for (; *p; ++p) {
1370         hash = FNV_32A_OP(hash, *p); // mov d1, BYTE PTR [ecx]
1371     }
1372
1373     //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1374     return ((hash>>16) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1375 }
1376
1377 // Invoking: FNV1A_Hash_Granularity(wrd, wrdlen>0|2|3, 0|2|3)
1378 int FNV1A_Hash_Granularity(char *str, int wrdlen_granulated, int Granularity) // wrdlen>0=wrden
1379 {
1380     u_int32_t hash;
1381     u_int64_t hash64;
1382     char *p;
1383
1384     hash = FNV1_32_INIT;
1385     p=str;
1386
1387     // The goal of stage #1: to reduce number of 'imul's and mainly: the number of loops.
1388
1389     // Stage #1:
1390     if (Granularity == 2) {
1391         for (; wrdlen_granulated != 0; --wrdlen_granulated) {
1392             hash = FNV_32A_OP32(hash, (u_int32_t)*(u_int32_t *p));
1393             p=p+4; // (1<<Granularity): 1<<0=1, 1<<2=4, 1<<3=8
1394         }
1395     }
1396     if (Granularity == 3) {
1397         hash64 = FNV1_64_INIT;
1398         for (; wrdlen_granulated != 0; --wrdlen_granulated) {

```

```

1400 hash64 = FNV_64A_OP64(hash64, (u_int64_t)*(u_int64_t *)p);
1401 p=p+8; // (1<<Granularity): 1<<0=1, 1<<2=4, 1<<3=8
1402 }
1403 for (; *p; ++p) {
1404     hash64 = FNV_64A_OP(hash64, (u_int8_t)*(u_int8_t *)p);
1405 }
1406
1407 //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1408 return (hash64>>51) ^ hash64 & 8191; // 00..8191 i.e. 2^13=8192
1409 // probably better shifting is not by 16 bits but ...
1410 //hash64>>16: 3,544,160 just bad
1411 //hash64>>33: 3,547,854
1412 //hash64>>34: 3,547,266
1413 //hash64>>35: 3,547,453
1414 //hash64>>36: 3,547,242
1415 //hash64>>40: 3,548,263
1416 //hash64>>44: 3,548,242
1417 //hash64>>45: 3,549,056
1418 //hash64>>46: 3,549,207
1419 //hash64>>47: 3,549,094
1420 //hash64>>50: 3,549,392
1421 //hash64>>51: 3,549,395 i.e. maximum shift: the 13 most significant bits i.e. (64-13); closest to 3,549,448
1422
1423 // Above results are obtained for following set:
1424 //if (wrklen<=19) // 4x4+3=19 i.e. last contains 7 clashes
1425 //    Slot = FNV1A_Hash_Granularity(wrd, wrklen>>2, 2)<<2; //13++++
1426 //else // 2x8+4=20 i.e. first contains 6 clashes
1427 //    Slot = FNV1A_Hash_Granularity(wrd, wrklen>>3, 3)<<2; //13++++
1428 // }
1429
1430 //if (Granularity != 3) {
1431 // Stage #2:
1432 for (; *p; ++p) {
1433     hash = FNV_32A_OP(hash, (u_int8_t)*(u_int8_t *)p);
1434 }
1435
1436 //return ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1437 return (hash>>16) ^ hash & 8191; // 00..8191 i.e. 2^13=8192
1438 // }
1439 }
1440
1441 // char *string; // the string to 64 bit FNV-1a hash */
1442 // u_int64_t hash; // will hold the final value of the hash */
1443 // char *p;
1444 //
1445 // hash = FNV1_64_INIT;
1446 // for (p=string; *p; ++p) {
1447 //     hash = FNV_64A_OP(hash, *p);
1448 // }
1449 // }
1450
1451
1452 // If you need an x-bit hash where x is not a power of 2,
1453 // 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.
1454 // By xor-folding we mean shift the excess high order bits down and xor them with the lower x-bits.
1455 // For tiny x < 16 bit values, we recommend using a 32 bit FNV-1 hash as follows:
1456
1457 // /* NOTE: for 0 < x < 16 ONLY!!! */
1458 // #define TINY_MASK(x) (((u_int32_t)1<<(x))-1)
1459 // #define FNV1_32_INIT ((u_int32_t)2166136261)
1460 // u_int32_t hash;
1461 // void *data;
1462 // size_t data_len;
1463 //
1464 // hash = fnv_32_buf(data, data_len, FNV1_32_INIT);
1465 // hash = (((hash>>x) ^ hash) & TINY_MASK(x));
1466
1467
1468 int FNV1A_Hash_SHIFTless_XORless(char *str)
1469 {
1470     u_int32_t hash; // will hold the final value of the hash */
1471     char *p;
1472
1473     hash = FNV1_32_INIT;
1474     for (p=string; *p; ++p) {
1475         hash = FNV_32A_OP(hash, *p);
1476     }
1477     //hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1478     return hash & 8191; // 00..8191 i.e. 2^13=8192
1479 }
1480
1481 /*
1482 _FNV1A_Hash_SHIFTless_XORless PROC NEAR
1483 ; Line 721
1484 mov edx, DWORD PTR _str$[esp-4]
1485 mov cl, BYTE PTR [edx]
1487 test cl, cl

```

```

1488 mov eax, -2128831035 ; 811c9dc5H
1489 je SHORT $L1582
1490 npad l
1491 $L1580:
1492 ; Line 722
1493 movzx ecx, cl
1494 xor ecx, eax
1495 imul ecx, 16777619 ; 01000193H
1496 inc edx
1497 mov eax, ecx
1498 mov cl, BYTE PTR [edx]
1499 test cl, cl
1500 jne SHORT $L1580
1501 $L1582:
1502 ; Line 726
1503 and eax, 8191 ; 00001ffffH
1504 ; Line 727
1505 ret 0
1506 _FNV1A_Hash_SHIFTless_XORless ENDP
1507 */
1508
1509
1510 int FNV1A_Hash(char *str)
1511 {
1512     u_int32_t hash; // will hold the final value of the hash */
1513     char *p;
1514
1515     hash = FNV1_32_INIT;
1516     for (p=string; *p; ++p) {
1517         hash = FNV_32A_OP(hash, *p);
1518     }
1519     //hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1520
1521     return ((hash>>13) ^ hash) & 8191; // 00..8191 i.e. 2^13=8192
1522 }
1523
1524 /*
1525 _FNV1A_Hash PROC NEAR
1526 ; Line 722
1527 mov edx, DWORD PTR _str$[esp-4]
1528 mov al, BYTE PTR [edx]
1529 test al, al
1530 mov ecx, -2128831035 ; 811c9dc5H
1531 je SHORT $L1582
1532 npad l
1533 $L1580:
1534 ; Line 723
1535 movzx eax, al
1536 xor eax, ecx
1537 imul eax, 16777619 ; 01000193H
1538 inc edx
1539 mov ecx, eax
1540 mov al, BYTE PTR [edx]
1541 test al, al
1542 jne SHORT $L1580
1543 $L1582:
1544 ; Line 727
1545 mov eax, ecx
1546 shr eax, 13 ; 0000000dH
1547 xor eax, ecx
1548 and eax, 8191 ; 00001ffffH
1549 ; Line 728
1550 ret 0
1551 _FNV1A_Hash ENDP
1552 */
1553
1554 /*
1555 Wayne Diamond implemented 32-bit FNV algorithm in PowerBASIC inline x86 assembly:
1556
1557
1558 FUNCTION FNV32(BYVAL dOffset AS DWORD, BYVAL dwLen AS DWORD, BYVAL offset_basis AS DWORD) AS DWORD
1559 #REGISTER NONE
1560 ! mov esi, dOffset ;esi = ptr to buffer
1561 ! mov ecx, dwLen ;ecx = length of buffer (counter)
1562 ! mov eax, offset_basis ;set to 2166136261 for FNV-1
1563 ! mov edi, &h01000193 ;FNV_32_PRIME = 16777619
1564 ! xor ebx, ebx ;ebx = 0
1565 nextbyte:
1566 ! mul edi ;eax = eax * FNV_32_PRIME
1567 ! mov bl, [esi] ;bl = byte from esi
1568 ! xor eax, ebx ;al = al xor bl
1569 ! inc esi ;esi = esi + 1 (buffer pos)
1570 ! dec ecx ;ecx = ecx - 1 (counter)
1571 ! jnz nextbyte ;if ecx is 0, jmp to NextByte
1572 ! mov FUNCTION, eax ;else, function = eax
1573 END FUNCTION
1574
1575 Wayne said:

```

```

1576
1577 ''Just thought I should let you know that I've ported the 32-bit FNV algorithm over to inline assembly.
1578 It's actually in PowerBASIC (www.powerbasic.com) format - a compiler I use, but the main function is all assembly.
1579 It could be optimized further in terms of saving a couple of clock cycles,
1580 but it's fairly optimized already - only 6 instructions in the main loop, plus 5 setup instructions,
1581 and compiles to just 33 bytes.''
1582
1583 M.S.Schulte sent us these 32-bit FNV-1 and FNV-1a x86 assembler implementations (written in flat assembler),
1584 half of which were optimized for speed, the other half were optimized for size:
1585
1586 small_fnv32: ;FNV1 32bit (size: 31 bytes)
1587 ; Intel Core 2 Duo E6600: 354.20 mb/s
1588 push esi
1589 push edi
1590 mov esi, [esp + 0ch] ;buffer
1591 mov ecx, [esp + 10h] ;length
1592 mov eax, [esp + 14h] ;basis
1593 mov edi, 01000193h ;fnv_32_prime
1594 next:
1595 mul edi
1596 xor al, [esi]
1597 inc esi
1598 loop snext
1599 pop edi
1600 pop esi
1601 ret 0ch
1602
1603 small_fnv32a: ;FNV1a 32bit (size: 31 bytes)
1604 ; Intel Core 2 Duo E6600: 327.68 mb/s
1605 push esi
1606 push edi
1607 mov esi, [esp + 0ch] ;buffer
1608 mov ecx, [esp + 10h] ;length
1609 mov eax, [esp + 14h] ;basis
1610 mov edi, 01000193h ;fnv_32_prime
1611 nexta:
1612 xor al, [esi]
1613 mul edi
1614 inc esi
1615 loop nexta
1616 pop edi
1617 pop esi
1618 ret 0ch
1619
1620 fast_fnv32: ;FNV1 32bit (size: 36 bytes)
1621 ; Intel Core 2 Duo E6600: 565.12 mb/s
1622 push ebx
1623 push esi
1624 push edi
1625 mov esi, [esp + 10h] ;buffer
1626 mov ecx, [esp + 14h] ;length
1627 mov eax, [esp + 18h] ;basis
1628 mov edi, 01000193h ;fnv_32_prime
1629 xor ebx, ebx
1630 next:
1631 mul edi
1632 mov bl, [esi]
1633 xor eax, ebx
1634 inc esi
1635 dec ecx
1636 jnz next
1637 pop edi
1638 pop esi
1639 pop ebx
1640 ret 0ch
1641
1642 fast_fnv32a: ;FNV1a 32bit (size: 36 bytes)
1643 ; Intel Core 2 Duo E6600: 574.95 mb/s
1644 push ebx
1645 push esi
1646 push edi
1647 mov esi, [esp + 10h] ;buffer
1648 mov ecx, [esp + 14h] ;length
1649 mov eax, [esp + 18h] ;basis
1650 mov edi, 01000193h ;fnv_32_prime
1651 xor ebx, ebx
1652 nexta:
1653 mov bl, [esi]
1654 xor eax, ebx
1655 mul edi
1656 inc esi
1657 dec ecx
1658 jnz nexta
1659 pop edi
1660 pop esi
1661 pop ebx
1662 ret 0ch
1663 */

```

```

1664
1665 //Number of Trees(GREATER THE BETTER): 352537
1666 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1667 //Number of Hash Collisions(Distinct WORDS - Number of Trees): 18677243
1668 int Hash17_unrolled(const char *key, int wrdlen)
1669 {
1670 int hash = 1;
1671 int i;
1672 for(i = 0; i < (wrdlen & -2); i += 2) {
1673 hash = (17) * hash + (key[i] - ' ');
1674 hash = (17) * hash + (key[i+1] - ' ');
1675 }
1676 if(wrdlen & 1)
1677 hash = (17) * hash + (key[wrdlen-1] - ' ');
1678 return ( hash ^ (hash >> 16) ) & 8191;
1679 }
1680
1681 //hash = 1:
1682 //Number of Trees(GREATER THE BETTER): 3556516
1683 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1684 //Number of Hash Collisions(Distinct WORDS - Number of Trees): 18646464
1685 //hash = 13:
1686 //Number of Trees(GREATER THE BETTER): 3556755
1687 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1688 //Number of Hash Collisions(Distinct WORDS - Number of Trees): 18646225
1689 //hash = 11:
1690 //Number of Trees(GREATER THE BETTER): 3557011
1691 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1692 //Number of Hash Collisions(Distinct WORDS - Number of Trees): 18645969
1693 //hash = 7:
1694 //Number of Trees(GREATER THE BETTER): 3557181
1695 //Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1696 //Number of Hash Collisions(Distinct WORDS - Number of Trees): 18645799
1697 int Alfalfa(const char *key, int wrdlen)
1698 {
1699 int hash = 7;
1700 int i;
1701 for(i = 0; i < (wrdlen & -2); i += 2) {
1702 hash = (17+9) * ((17+9) * hash + (key[i])) + (key[i+1]);
1703 }
1704 if(wrdlen & 1)
1705 hash = (17+9) * hash + (key[wrdlen-1]);
1706 return ( hash ^ (hash >> 16) ) & 8191;
1707 }
1708
1709 /*
1710 [FNV1A 'shift-less-&-xor-less' hash used in Leprechaun r.13+++:]
1711
1712 int FNV1A_Hash_SHIFTless_XORless(char *str)
1713 {
1714 u_int32_t hash;
1715 char *p;
1716
1717 hash = FNV1_32_INIT;
1718 for (p=str; *p; ++p) {
1719 hash = FNV_32A_OP(hash, *p);
1720 }
1721 //hash = ((hash>>13) ^ hash) & 8191; // (((u_int32_t)1<<(x))-1) where x=13
1722
1723 return hash & 8191; // 00..8191 i.e. 2^13=8192
1724 }
1725
1726 words per second performance: 837,458w/s
1727 Input File with a list of TEXTual Files: wikipedia-en-html.tar.wrd.lst
1728 word count: 12,561,874 of them 12,561,874 distinct
1729 Number of Trees(GREATER THE BETTER): 2772875
1730 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 41%
1731 Number of Hash Collisions(Distinct WORDS - Number of Trees): 9788999
1732
1733 words per second performance: 1,007,751w/s
1734 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
1735 word count: 35,271,297 of them 22,202,980 distinct
1736 Number of Trees(GREATER THE BETTER): 3537061
1737 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1738 Number of Hash Collisions(Distinct WORDS - Number of Trees): 18665919
1739
1740 [My '2in1' hash used in Leprechaun r.13+++:]
1741
1742 int KuxHash3plus(char *str)
1743 { int h = 0;
1744 unsigned long h2 = 0; // must be long: 31*'z'=31*122
1745 int max31 = 0;
1746 while (str[max31])
1747 { h = h ^ str[max31]; // 00..255 i.e. 2^8=256
1748 //h2 = h2 + str[max31+]; // [113s]
1749 h2 = h2 + str[max31+] * (max31+1);
1750 }
1751 // Result is: 7bits in 'h' and 32bits in 'h2'.

```

```

1752
1753 //printf("%s:\n",str);
1754 //printf("%d",h);
1755 // a in ASCII is 097 = 0110 0001
1756 // z in ASCII is 122 = 0111 1010
1757 // Above two lines show that bits 8-7-6 are always 0-1-1 so need for low 5 bits.
1758 //h<8; // 00..15 i.e. 5bits + 00-07bits=13bits
1759 //printf("%d",h);
1760 //printf("%d",h2);
1761 //h = h|(str[0] ^ str[max31-1]); // [111s] a..z: each XOR each gives 00..31
1762 h = ((h<8)|(h2%251))&8191; // 251 prime
1763 //printf("%d\n",h);
1764 return h; // 00..8191 i.e. 2^13=8192
1765 }
1766
1767 words per second performance: 785,117w/s
1768 Input File with a list of TEXTual Files: wikipedia-en.html.tar.wrd.lst
1769 word count: 12,561,874 of them 12,561,874 distinct
1770 Number of Trees(GREATER THE BETTER): 2663566
1771 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 40%
1772 Number of Hash Collisions(Distinct WORDS - Number of Trees): 9898308
1773
1774 words per second performance: 979,758w/s
1775 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
1776 word count: 35,271,297 of them 22,202,980 distinct
1777 Number of Trees(GREATER THE BETTER): 3410463
1778 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 51%
1779 Number of Hash Collisions(Distinct WORDS - Number of Trees): 18792517
1780
1781 [Last standing for English(en)-wikipedia's wordlist:]
1782 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%
1783 chongo's hash has better distribution than kaze's hash by ((9898308 - 9788999)/9788999)*100% = 1.1%
1784
1785 [Last standing for LATIN(de,en,es,fr,it,nl,pt,ro)-wikipedia's wordlist:]
1786 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%
1787 chongo's hash has better distribution than kaze's hash by ((18792517 - 18665919)/18665919)*100% = 0.6%
1788
1789 Bottomline is:
1790 your hash thrash, my hash for trash, he-he.
1791 Thanks a lot, again, Mr. Noll.
1792
1793 Yummy little file: http://www.isthe.com/chongo/src/fnv/fnv-5.0.2.tar.gz
1794 */
1795
1796 /*
1797 // Paul Larson (http://research.microsoft.com/~PALARSON/)
1798 UINT HashLarson(const CHAR *key, SIZE_T len) {
1799     UINT hash = 0;
1800     for(UINT i = 0; i < len; ++i)
1801         hash = 101 * hash + key[i];
1802     return hash ^ (hash >> 16);
1803 }
1804
1805 // Kernighan & Ritchie, "The C programming Language", 3rd edition.
1806 UINT HashKernighanRitchie(const CHAR *key, SIZE_T len) {
1807     UINT hash = 0;
1808     for(UINT i = 0; i < len; ++i)
1809         hash = 31 * hash + key[i];
1810     return hash;
1811 }
1812
1813 // A hash function with multiplier 65599 (from Red Dragon book)
1814 UINT Hash65599(const CHAR *key, SIZE_T len) {
1815     UINT hash = 0;
1816     for(UINT i = 0; i < len; ++i)
1817         hash = 65599 * hash + key[i];
1818     return hash ^ (hash >> 16);
1819 }
1820
1821 // FNV hash, http://isthe.com/chongo/tech/comp/fnv/
1822 UINT HashFNV1a(const CHAR *key, SIZE_T len) {
1823     UINT hash = 2166136261;
1824     for(UINT i = 0; i < len; ++i)
1825         hash = 16777619 * (hash ^ key[i]);
1826     return hash ^ (hash >> 16);
1827 }
1828
1829 // Ramakrishna hash
1830 UINT HashRamakrishna(const CHAR *key, SIZE_T len) {
1831     UINT h = 0;
1832     for(UINT i = 0; i < len; ++i) {
1833         h ^= (h << 5) + (h >> 2) + key[i];
1834     }
1835     return h;
1836 }
1837 */
1838
1839 /*

```

```

1840 Results for 'Hash_Alfa':
1841 Bytes per second performance: 19,808,709B/s
1842 words per second performance: 1,679,585w/s
1843 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
1844 Size of all TEXTual Files: 415,982,896
1845 word count: 35,271,297 of them 22,202,980 distinct
1846 Number of Files: 8
1847 Number of Lines: 35271297
1848 Allocated memory in MB: 1950
1849 Number of Trees(GREATER THE BETTER): 3549079
1850 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1851 Number of Hash Collisions(Distinct WORDS - Number of Trees): 18653901
1852 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '37'
1853 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 117,063,824
1854 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,279
1855 Perfectly-Balanced-Binary-Search-Tree for MaxNODES = 84 must have PEAK = 7 = rounding down of integer (1+lb(84))
1856 Binary-Search-Tree(1st out of 2) with MaxNODES = 84 has PEAK = 20 and LEAFs = 24
1857 Binary-Search-Tree(1st out of 3) with MaxPEAK = '37' has NODES = 67 and LEAFs = 17
1858 Binary-Search-Tree(1st out of 1) with MaxLEAFs = 28 has NODES = 78 and PEAK = 22
1859 */
1860 UINT HashAlfa(const char *key, unsigned int wrdlen)
1861 {
1862     UINT hash = 7;
1863     unsigned int i;
1864     for (i = 0; i < (wrdlen & -2); i += 2) {
1865         hash = (53) * ((53) * hash + (key[i])) + (key[i+1]);
1866     }
1867     if (wrdlen & 1)
1868         hash = (53) * hash + (key[wrdlen-1]);
1869     return ((hash>>16) ^ hash) & 8191;
1870 }
1871
1872 /*
1873 Results for 'HashAlfa_HALF':
1874 Bytes per second performance: 19,808,709B/s
1875 words per second performance: 1,679,585w/s
1876 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
1877 Size of all TEXTual Files: 415,982,896
1878 word count: 35,271,297 of them 22,202,980 distinct
1879 Number of Files: 8
1880 Number of Lines: 35271297
1881 Allocated memory in MB: 1950
1882 Number of Trees(GREATER THE BETTER): 3550665
1883 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1884 Number of Hash Collisions(Distinct WORDS - Number of Trees): 18652315
1885 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '39'
1886 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 117,053,918
1887 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,072,259
1888 Perfectly-Balanced-Binary-Search-Tree for MaxNODES = 87 must have PEAK = 7 = rounding down of integer (1+lb(87))
1889 Binary-Search-Tree(1st out of 1) with MaxNODES = 87 has PEAK = 21 and LEAFs = 27
1890 Binary-Search-Tree(1st out of 2) with MaxPEAK = '39' has NODES = 65 and LEAFs = 18
1891 Binary-Search-Tree(1st out of 4) with MaxLEAFs = 27 has NODES = 77 and PEAK = 23
1892 */
1893 UINT HashAlfa_HALF(const char *key, unsigned int wrdlen)
1894 {
1895     UINT hash = 12;
1896     UINT hashBUFFER;
1897     unsigned int i,j;
1898     for(i = 0; i < (wrdlen & -4); i += 4) {
1899         //hash = (( (hash<<5)-hash) + key[i] )<<5) - (( (hash<<5)-hash) + key[i] ) + (key[i+1]);
1900         hashBUFFER = ((hash<<5)-hash) + key[i];
1901         hash = (( hashBUFFER <<5) - ( hashBUFFER ) + (key[i+1]));
1902         //hash = (( (hash<<5)-hash) + key[i+2] )<<5) - (( (hash<<5)-hash) + key[i+2] ) + (key[i+3]);
1903         hashBUFFER = ((hash<<5)-hash) + key[i+2];
1904         hash = (( hashBUFFER <<5) - ( hashBUFFER ) + (key[i+3]));
1905     }
1906     for(j = 0; j < (wrdlen & 3); j += 1) {
1907         hash = ((hash<<5)-hash) + key[i+j];
1908     }
1909     return ((hash>>16) ^ hash) & 8191;
1910 }
1911
1912 /*
1913 Results for 'HashFNV1A_unrolled_Final':
1914 Bytes per second performance: 19,808,709B/s
1915 words per second performance: 1,679,585w/s
1916 Input File with a list of TEXTual Files: Leprechaun_vs_Wikipedia_LATIN-WORDS.lst
1917 Size of all TEXTual Files: 415,982,896
1918 word count: 35,271,297 of them 22,202,980 distinct
1919 Number of Files: 8
1920 Number of Lines: 35271297
1921 Allocated memory in MB: 1950
1922 Number of Trees(GREATER THE BETTER): 3445337
1923 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 52%
1924 Number of Hash Collisions(Distinct WORDS - Number of Trees): 18737643
1925 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '43'
1926 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 118,349,998
1927 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 7,997,033

```

```

1928 Perfectly-Balanced-Binary-Search-Tree for MaxNODES = 89 must have PEAK = 7 = rounding down of integer (1+lb(89))
1929 Binary-Search-Tree(1st out of 1) with MaxNODES = 89 has PEAK = 28 and LEAFs = 28
1930 Binary-Search-Tree(1st out of 1) with MaxPEAK = '43' has NODES = 65 and LEAFs = 11
1931 Binary-Search-Tree(1st out of 2) with MaxLEAFs = 28 has NODES = 78 and PEAK = 24
1932 */
1933 UINT HashFNVA_unrolled_Final(char *str, unsigned int wrdlen)
1934 {
1935 //const UINT PRIME = 31;
1936 unsigned int hash = 2166136261;
1937 char * p = str;
1938
1939 /*
1940 // Reduce the number of multiplications by unrolling the loop
1941 for (SIZE_T ndwords = wrdlen / sizeof(DWORD); ndwords; --ndwords) {
1942 //hash = (hash ^ *(DWORD*)p) * PRIME;
1943 hash = ((hash ^ *(DWORD*)p)<<5) - (hash ^ *(DWORD*)p);
1944
1945 p += sizeof(DWORD);
1946 }
1947 */
1948 for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
1949 hash = ((hash ^ *(unsigned int*)p)<<5) - (hash ^ *(unsigned int*)p);
1950 }
1951
1952 // Process the remaining bytes
1953 /*
1954 for (SIZE_T i = 0; i < (wrdlen & (sizeof(DWORD) - 1)); i++) {
1955 //hash = (hash ^ *p++) * PRIME;
1956 hash = ((hash ^ *p)<<5) - (hash ^ *p);
1957 p++;
1958 }
1959 */
1960 if (wrdlen & -2) {
1961 hash = ((hash ^ (*(unsigned int*)&0xFFFF)<<5) - (hash ^ (*(unsigned int*)&0xFFFF)));
1962 p++;p++;
1963 }
1964 if (wrdlen & 1)
1965 hash = ((hash ^ *p)<<5) - (hash ^ *p);
1966
1967 return ((hash>>16) ^ hash) & 8191;
1968 }
1969
1970 /*
1971 Results for 'Sixtinsensitive':
1972 Bytes per second performance: 19,808,709B/s
1973 words per second performance: 1,679,585W/s
1974 Input File with a list of TEXTUAL Files: Leprechaun_vs_wikipedia_LATIN-WORDS.lst
1975 Size of all TEXTUAL Files: 415,982,896
1976 word count: 35,271,297 of them 22,202,980 distinct
1977 Number of Files: 8
1978 Number of Lines: 35271297
1979 Allocated memory in MB: 1950
1980 Number of Trees(GREATER THE BETTER): 3531949
1981 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
1982 Number of Hash Collisions(Distinct WORDS - Number of Trees): 18671031
1983 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '38'
1984 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 118,959,016
1985 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,047,983
1986 Perfectly-Balanced-Binary-Search-Tree for MaxNODES = 98 must have PEAK = 7 = rounding down of integer (1+lb(98))
1987 Binary-Search-Tree(1st out of 1) with MaxNODES = 98 has PEAK = 36 and LEAFs = 30
1988 Binary-Search-Tree(1st out of 1) with MaxPEAK = '38' has NODES = 54 and LEAFs = 11
1989 Binary-Search-Tree(1st out of 2) with MaxLEAFs = 30 has NODES = 98 and PEAK = 36
1990 */
1991 // Tuned for lowercase-and-uppercase letters i.e. 26 ASCII symbols 65-90 and 97-122 decimal.
1992 UINT Sixtinsensitive(const char *str, unsigned int wrdlen)
1993 {
1994 UINT hash = 2166136261;
1995 UINT hashBUFFER_EAX, hashBUFFER_BH, hashBUFFER_BL;
1996 const char * p = str;
1997
1998 // 0x41 = 065 'A' 010 [0 0001]
1999 // 0x5A = 090 'Z' 010 [1 1010]
2000 // 0x61 = 097 'a' 011 [0 0001]
2001 // 0x7A = 122 'z' 011 [1 1010]
2002
2003 // Reduce the number of multiplications by unrolling the loop
2004 for(; wrdlen >= 6; wrdlen -= 6, p += 6) {
2005 //hashBUFFER_EAX = (*(DWORD*)(p+0)&0xFFFF);
2006 hashBUFFER_EAX = (*(DWORD*)(p+0)&0x1F1F1F1F);
2007 hashBUFFER_BL = (*(p+4)&0x1F);
2008 hashBUFFER_BH = (*(p+5)&0x1F);
2009 //6bytes-in-4bytes or 48bits-to-30bits
2010 // Two times next:
2011 //3bytes-in-2bytes or 24bits-to-15bits
2012 //EAX BL BH
2013 //[[5bit][3bit][5bit][3bit][5bit][3bit][5bit][3bit]
2014 // 5th[0..15] 13th[0..15]
2015 // BL lower 3 BL higher 2bits

```

```

2016 // OR or XOR no difference
2017 hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BL&0x07)<<5); // BL lower 3bits of 5bits
2018 hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BL&0x18)<<(2+8)); // BL higher 2bits of 5bits
2019 hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BH&0x07)<<(5+16)); // BH lower 3bits of 5bits
2020 hashBUFFER_EAX = hashBUFFER_EAX ^ ((hashBUFFER_BH&0x18)<<(2+8+16)); // BH higher 2bits of 5bits
2021 //hash = (hash ^ hashBUFFER_EAX)*1607; //What a mess: <<7 becomes imul but <<5 not!?!
2022 hash = (hash ^ hashBUFFER_EAX)<<5) - (hash ^ hashBUFFER_EAX);
2023 //1607: [2118599]
2024 // 127: [2121081]
2025 // 31: [2139242]
2026 // 17: [2150803]
2027 // 7: [2166336]
2028 // 5: [2183044]
2029 // 8191: [2200477]
2030 // 3: [2205095]
2031 // 257: [2206188]
2032 }
2033 // Post-Variant #1:
2034 for(; wrdlen; wrdlen--, p++) {
2035 hash = ((hash ^ (*p&0x1F)<<5) - (hash ^ (*p&0x1F)));
2036 }
2037 /*
2038 // Post-Variant #2:
2039 for(; wrdlen >= 2; wrdlen -= 2, p += 2) {
2040 hash = ((hash ^ (*(DWORD*)p&0xFFFF)<<5) - (hash ^ (*(DWORD*)p&0xFFFF)));
2041 }
2042 if (wrdlen & 1)
2043 hash = ((hash ^ *p)<<5) - (hash ^ *p);
2044 */
2045 /*
2046 // Post-Variant #3:
2047 for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
2048 hash = ((hash ^ *(DWORD*)p)<<5) - (hash ^ *(DWORD*)p);
2049 }
2050 if (wrdlen & -2) {
2051 hash = ((hash ^ (*(DWORD*)p&0xFFFF)<<5) - (hash ^ (*(DWORD*)p&0xFFFF)));
2052 p++;p++;
2053 }
2054 if (wrdlen & 1)
2055 hash = ((hash ^ *p)<<5) - (hash ^ *p);
2056 */
2057 return ((hash>>16) ^ hash) & 8191;
2058 }
2059
2060 /*
2061 #define FNVL_32_INIT ((UINT)2166136261)
2062 #define FNVL_32_PRIME ((UINT)1709)
2063
2064 #define FNVL_32A_OP(hash, octet) \
2065 (((UINT)(hash) ^ (unsigned char)(octet)) * FNVL_32_PRIME)
2066
2067 #define FNVL_32A_OP32(hash, octet) \
2068 (((UINT)(hash) ^ (UINT)(octet)) * FNVL_32_PRIME)
2069
2070 UINT FNVL1A_Hash_WHIZ(const char *str, SIZE_T wrdlen)
2071 {
2072
2073 UINT hash32;
2074 const char *p;
2075
2076 hash32 = FNVL_32_INIT;
2077 p=str;
2078
2079 for(; wrdlen >= 4; wrdlen -= 4, p += 4) {
2080 hash32 = FNVL_32A_OP32(hash32, (UINT)*(UINT *)p);
2081 }
2082 if (wrdlen & -2) {
2083 hash32 = FNVL_32A_OP32(hash32, *(UINT*)&0xFFFF);
2084 p++;p++;
2085 }
2086 if (wrdlen & 1)
2087 hash32 = FNVL_32A_OP(hash32, *p);
2088
2089 return hash32 ^ (hash32 >> 16);
2090 }
2091 */
2092
2093 /*
2094 Results for 'FNVL1A_Hash_Jester':
2095 Bytes per second performance: 19,808,709B/s
2096 words per second performance: 1,679,585W/s
2097 Input File with a list of TEXTUAL Files: Leprechaun_vs_wikipedia_LATIN-WORDS.lst
2098 Size of all TEXTUAL Files: 415,982,896
2099 word count: 35,271,297 of them 22,202,980 distinct
2100 Number of Files: 8
2101 Number of Lines: 35271297
2102 Allocated memory in MB: 1950
2103 Number of Trees(GREATER THE BETTER): 3537352

```

```

2104 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2105 Number Of Hash Collisions(Distinct WORDS - Number Of Trees): 18665628
2106 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '37'
2107 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 117,243,563
2108 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,063,361
2109 Perfectly-Balanced-Binary-Search-Tree for MaxNODES = 87 must have PEAK = 7 = rounding down of integer (1+lb(87))
2110 Binary-Search-Tree(1st out of 2) with MaxNODES = 87 has PEAK = 27 and LEAFs = 23
2111 Binary-Search-Tree(1st out of 1) with MaxPEAK = '37' has NODES = 66 and LEAFs = 18
2112 Binary-Search-Tree(1st out of 3) with MaxLEAFs = 27 has NODES = 84 and PEAK = 27
2113 */
2114 UINT FNV1A_Hash_Jester(const char *str, unsigned int wrdlen)
2115 {
2116     const UINT PRIME = 709607;
2117     UINT hash32 = 2166136261;
2118     const char *p = str;
2119
2120     // Idea comes from Igor Pavlov's 7zCRC, thanks.
2121     /*
2122     for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
2123         hash32 = (hash32 ^ *p) * PRIME;
2124     }
2125     */
2126     for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
2127         hash32 = (hash32 ^ *(DWORD*)p) * PRIME;
2128         hash32 = (hash32 ^ *(DWORD*)(p+4)) * PRIME;
2129     }
2130     // Cases: 0,1,2,3,4,5,6,7
2131     if (wrdlen & sizeof(DWORD)) {
2132         hash32 = (hash32 ^ *(DWORD*)p) * PRIME;
2133         p += sizeof(DWORD);
2134     }
2135     if (wrdlen & sizeof(WORD)) {
2136         hash32 = (hash32 ^ *(WORD*)p) * PRIME;
2137         p += sizeof(WORD);
2138     }
2139     if (wrdlen & 1)
2140         hash32 = (hash32 ^ *p) * PRIME;
2141
2142     return (hash32 ^ (hash32 >> 16)) & 8191;
2143 }
2144
2145 /*
2146 Results for 'FNV1A_Hash_Jesteress':
2147 Bytes per second performance: 19,808,709B/s
2148 words per second performance: 1,679,585w/s
2149 Input File with a list of TEXTUAL Files: Leprechaun_vs_wikipedia_LATIN-WORDS.lst
2150 Size of all TEXTUAL Files: 415,982,896
2151 word count: 35,271,297 of them 22,202,980 distinct
2152 Number of Files: 8
2153 Number of Lines: 35271297
2154 Allocated memory in MB: 1950
2155 Number of Trees(GREATER THE BETTER): 3537293
2156 Forest population(Hash Function Quality regarding Collisions i.e. Hash Table Utilization): 53%
2157 Number Of Hash Collisions(Distinct WORDS - Number Of Trees): 18665687
2158 Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: '40'
2159 Total Attempts to Find/Put WORDS into Binary-Search-Trees: 117,526,680
2160 Total Number of LEAFs in Binary-Search-Trees(GREATER THE BETTER): 8,051,512
2161 Perfectly-Balanced-Binary-Search-Tree for MaxNODES = 89 must have PEAK = 7 = rounding down of integer (1+lb(89))
2162 Binary-Search-Tree(1st out of 1) with MaxNODES = 89 has PEAK = 25 and LEAFs = 23
2163 Binary-Search-Tree(1st out of 1) with MaxPEAK = '40' has NODES = 49 and LEAFs = 8
2164 Binary-Search-Tree(1st out of 1) with MaxLEAFs = 28 has NODES = 72 and PEAK = 21
2165 */
2166 #define ROL(x, n) (((x) << (n)) | ((x) >> (32-(n))))
2167 UINT FNV1A_Hash_Jesteress(const char *str, unsigned int wrdlen)
2168 {
2169     const UINT PRIME = 709607;
2170     UINT hash32 = 2166136261;
2171     const char *p = str;
2172
2173     // Idea comes from Igor Pavlov's 7zCRC, thanks.
2174     /*
2175     for(; wrdlen && ((unsigned)(ptrdiff_t)p&3); wrdlen -= 1, p++) {
2176         hash32 = (hash32 ^ *p) * PRIME;
2177     }
2178     */
2179     for(; wrdlen >= 2*sizeof(DWORD); wrdlen -= 2*sizeof(DWORD), p += 2*sizeof(DWORD)) {
2180         hash32 = (hash32 ^ (ROL(*(DWORD*)p,5)^*(DWORD*)(p+4))) * PRIME;
2181     }
2182     // Cases: 0,1,2,3,4,5,6,7
2183     if (wrdlen & sizeof(DWORD)) {
2184         hash32 = (hash32 ^ *(DWORD*)p) * PRIME;
2185         p += sizeof(DWORD);
2186     }
2187     if (wrdlen & sizeof(WORD)) {
2188         hash32 = (hash32 ^ *(WORD*)p) * PRIME;
2189         p += sizeof(WORD);
2190     }
2191     if (wrdlen & 1)

```

```

2192     hash32 = (hash32 ^ *p) * PRIME;
2193
2194     return (hash32 ^ (hash32 >> 16)) & 8191;
2195 }
2196
2197
2198 /*
2199 UINT NextPowerOfTwo(UINT x) {
2200     // Henry Warren, "Hacker's Delight", ch. 3.2
2201     x--;
2202     x |= (x >> 1);
2203     x |= (x >> 2);
2204     x |= (x >> 4);
2205     x |= (x >> 8);
2206     x |= (x >> 16);
2207     return x + 1;
2208 }
2209
2210 UINT NextLog2(UINT x) {
2211     // Henry Warren, "Hacker's Delight", ch. 5.3
2212     if(x <= 1) return x;
2213     x--;
2214     UINT n = 0;
2215     UINT y;
2216     y = x >> 16; if(y) {n += 16; x = y;}
2217     y = x >> 8; if(y) {n += 8; x = y;}
2218     y = x >> 4; if(y) {n += 4; x = y;}
2219     y = x >> 2; if(y) {n += 2; x = y;}
2220     y = x >> 1; if(y) return n + 2;
2221     return n + x;
2222 }
2223 */
2224
2225 // 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.
2226 /*
2227 * Returns the floor form of binary logarithm for a 32 bit integer.
2228 * -1 is returned if n is 0.
2229 */
2230 int floorLog2(unsigned int n) {
2231     int pos = 0;
2232     if (n >= 1<<16) {n >>= 16; pos += 16;}
2233     if (n >= 1<<8) {n >>= 8; pos += 8;}
2234     if (n >= 1<<4) {n >>= 4; pos += 4;}
2235     if (n >= 1<<2) {n >>= 2; pos += 2;}
2236     if (n >= 1<<1) {pos += 1;}
2237     return ((n == 0) ? (-1) : pos);
2238 }
2239
2240
2241 int main( argc, argv )
2242 int argc; char *argv[];
2243 {
2244     int nlines;
2245     string *backup = NULL;
2246
2247     FILE *fp_in, *fp_out, *fp_outLOG, *fp_inLINE;
2248     int LetterOffset;
2249     unsigned long long FilesLEN;
2250     unsigned long long WORDcount;
2251     unsigned long long WORDcountAttemptsToPut;
2252     int Thunderwith;
2253     unsigned long NumberOfFiles, WORDcountDistinct;
2254     unsigned long long NumberOfLines; // rev. 12+
2255     unsigned long WHOLELetter_Buffersize;
2256     unsigned long memory_size, LetterBuffer, j, k, LINE10len, wrdlen;
2257     unsigned long k_FIX;
2258     unsigned long long i; // rev. 12+
2259     //unsigned long size_in, size_out, size_inLINE;
2260     unsigned long size_in; // rev. 12+
2261     #if defined(_WIN32_ENVIRONMENT_)
2262     unsigned long long size_inLINESIXFOUR;
2263     #else
2264     size_t size_inLINESIXFOUR;
2265     #endif /* defined(_WIN32_ENVIRONMENT_) */
2266
2267     //unsigned long t1, t2, t3;
2268     time_t t1, t2, t3;
2269
2270     const int NumberOfSLOTS = 4096*2; // Since r.12+ in rev.12 it was 4096
2271     unsigned long StackPtr;
2272     unsigned long BSTstack [65536*3]; // BST in worst case could become a LL.
2273     unsigned long NumberOfTrees=0, NumberOfHashCollisions=0;
2274     unsigned long iBSTwithMAXpeak, jBSTwithMAXpeak;
2275     unsigned int PEAKiBST;
2276     unsigned long BSTSTotalLEAFs=0; //?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur below where 'break'
    is?!

```

```

2277 unsigned long BSTwithMAXnode=0, BSTcurrentNode=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2278 unsigned long BSTcurrentNodeMAXQUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
BSTcurrent occur below where 'break' is ?!
2279 unsigned long BSTwithMAXnodePEAK=1, BSTwithMAXnodeLEAF=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
occur below where 'break' is ?!
2280 unsigned long BSTwithMAXpeak=0, BSTcurrentPeak=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2281 unsigned long BSTcurrentPeakMAX=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2282 unsigned long BSTcurrentPeakMAXQUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
BSTcurrent occur below where 'break' is ?!
2283 unsigned long BSTwithMAXpeakNODE=1, BSTwithMAXpeakLEAF=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
occur below where 'break' is ?!
2284 unsigned long BSTwithMAXleaf=0, BSTcurrentLeaf=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent occur
below where 'break' is ?!
2285 unsigned long BSTcurrentLeafMAXQUANTITY=0; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of
BSTcurrent occur below where 'break' is ?!
2286 unsigned long BSTwithMAXleafNODE=1, BSTwithMAXleafPEAK=1; // ?! MADHOUSE: if BSTcurrent is not zeroed here then INSANE values of BSTcurrent
occur below where 'break' is ?!
2287
2288 char *pointerFlush, *pointerFlushUNALIGN, *BufStart, *Flushing;
2289 unsigned long PseudoLinkedPointer, PseudoLinkedPointerNEW, PseudoLinkedPointerROOT, PseudoLinkedPointerNEWold;
2290 unsigned long PseudoLinkedPointerNEWleft, PseudoLinkedPointerNEWright;
2291 unsigned long PseudoLinkedPointerNEWmiddle;
2292 char *bufend[ 806 ]; // 'a'=0, ... 'z'=25 - 26 letters x 31 lengths
2293 long bufNumberOfWords[ 806 ]; // 'a'=0, ... 'z'=25 - 26 letters x 31 lengths
2294 // long bufNoWps[ 806 ][ 8192 ]; // ?! crashes below when an attempt to use it occur
2295 char wrd[32]; // 0..30, 31 = 0
2296 char wrdup[32]; // 0..30, 31 = 0
2297 char wrdupold[32]; // 0..30, 31 = 0
2298 char LINE10[257]; // 000..255, 256 = 0
2299 char ZEROS[4]; // 0..3, 0 = 0, 1 = 0, 2 = 0, 3 = 0
2300 char CrdLfA[2]; // 0..1, 0 = 13, 1 = 10
2301 char workbyte;
2302 char workk[1024*128];
2303 long workKoffset = -1;
2304 int FoundInLinkedList, Slot;
2305 unsigned long OffsetsInBuffer[31]; // 00..30
2306 unsigned long MAXusedBuffer[32]; // 00 not used, only 01..31
2307 unsigned long GRMBLhill[32]; // 00..31
2308 unsigned long GRMBLFoolAgain[32]; // 00..31
2309 int MeInitchka;
2310 unsigned long MAXusedBufferABS = 0;
2311 unsigned long Utiliza1 = 0;
2312 unsigned long Utiliza2 = 0;
2313 unsigned long TotalWlchars = 0;
2314
2315 /* minimum signed 64 bit value */
2316 #define _I64_MIN (-9223372036854775807i64 - 1)
2317 /* maximum signed 64 bit value */
2318 #define _I64_MAX 9223372036854775807i64
2319 /* maximum unsigned 64 bit value */
2320 #define _UI64_MAX 0xffffffffffffffffui64
2321
2322 /* minimum signed 128 bit value */
2323 #define _I128_MIN (-170141183460469231731687303715884105727i128 - 1)
2324 /* maximum signed 128 bit value */
2325 #define _I128_MAX 170141183460469231731687303715884105727i128
2326 /* maximum unsigned 128 bit value */
2327 #define _UI128_MAX 0xffffffffffffffffffffffffffffffffui128
2328
2329 char l1Toadigits[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('0')+6(,)
2330 // below duplicates are needed because of one_line_invoiking need different buffers.
2331 char l1Toadigits2[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('0')+6(,)
2332 char l1Toadigits3[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('0')+6(,)
2333 char l1Toadigits4[27]; // 9,223,372,036,854,775,807: 1(sign or carry)+19(digits)+1('0')+6(,)
2334 unsigned long HEADOfFetFromStartBUKVA = 0;
2335 unsigned long TAILOfFetFromStartBUKVA = 0;
2336 int BStorBtree = 0;
2337 int SplitOccurred;
2338 int PoffsetInLEAF;
2339 char *Auberge[4] = {"\0", "\0", "\0", "\0"};
2340 int hashAlfa[4], iAlfa[4];
2341 int PLE_words=0; // Quadruple!
2342 char wrd1st[32]; // 0..30, 31 = 0
2343 char wrd2nd[32]; // 0..30, 31 = 0
2344 char wrd3rd[32]; // 0..30, 31 = 0
2345 char wrd4th[32]; // 0..30, 31 = 0
2346 char *DelimiterUnderscore = "_\0";
2347 int PLE_words_INITFlag = 0;
2348
2349 // INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT INIT
2350 puts( "Leprechaun(Fast Greedy word-ripper), rev. 13_7pluses_FIX quadrupleton_r1, written by Svalqatchx." );
2351 puts( "Leprechaun: 'oh, well, didn't you hear? Bigger is good, but jumbo is dear.' );
2352 puts( "Kaze: Let's see what a 3-way hash + 6,602,752 Binary-Search-Trees can give us, );
2353 puts( " also the performance of a 3-way hash + 6,602,752 B-Trees of order 3." );
2354 //puts( "Note1: Compiled with Microsoft C v. 13.10.3077: 'cl /ox /cLeprechaun.c.' );

```

```

2355 //puts( "Note2: This WORDLISTER makes as output pseudo(unsorted)_wordlist_CRLF_file." );
2356 if( argc != 3 && argc != 4 && argc != 5 && argc != 6 ) // +1 for program name
2357 {
2358 puts( "" );
2359 puts( "The Little Monster' short notes." );
2360 puts( "Note1: I wish to thank to R.N. Horspool, Ranjan Sinha, Dmitry Shkarin." );
2361 puts( "Michael Abrash, J. Bentley, R. Sedgewick, Igor Pavlov, Lasse Reinhold." );
2362 puts( "Landon Noll, Peter Kankowski for sharing their knowledge to public." );
2363 puts( "Note2: Run it without parameters to get usage and short notes." );
2364 puts( "Note3: This simple amateurish(more over I am not versed well neither in C nor );
2365 puts( "in mathematics nor in english language, but I am persistent in INDEXING );
2366 puts( "GBS of english TEXTS tool is written in ANSI C(at least its source is );
2367 puts( "compleable for CL(windows) and GCC(Linux), and its purpose is to );
2368 puts( "create a wordList for a group of files(given via filelist)." );
2369 puts( "Its name comes(according to Heritage Dictionary) from 'low corpus' or );
2370 puts( "little body", in fact from amazing movie saga 'Leprechaun 1-2-3-4-5-6" );
2371 puts( "starring by warwick Davis." );
2372 puts( "Note4: Only words up to 31 chars are proceeded - the reason is 'ODT(the' );
2373 puts( "longest word in Heritage Dictionary 3rd edition) or );
2374 puts( "dichlorodiphenyltrichloroethane." );
2375 puts( "Note5: Cursor hiding in C - mission impossible for me." );
2376 puts( "Note6: By default(third parameter is 1023) allocated memory is 393MB." );
2377 puts( "Due to 'malloc()' limitation under WINDOVS, maximum value of third );
2378 puts( "parameter is 5174 which is 1988MB allocated block." );
2379 puts( "Note7: File Leprechaun.LOG is a log, where new statistics are appended." );
2380 puts( "Note8: Revision 12+ can handle files larger than 4GB." );
2381 puts( "Note9: Revision 12++ has a buffered 'fread()' - therefore I/O READ-BURST SPEED );
2382 puts( "is the first(worst) bottleneck, as a result r.12++ is much-much faster );
2383 puts( "the second(worse) bottleneck: the linked lists - the b-trees );
2384 puts( "might be the answer; the third(bad) bottleneck: the amateurish author." );
2385 puts( "NoteA: Revision 12+++ has an improved(2 bits were used doltishly) main hash );
2386 puts( "function - therefore less collisions, for example: );
2387 puts( "for file 'wikipedia-de.html.tar' 42,291,855,360 bytes with );
2388 puts( "5,750,179,678 words of them 7,375,373 distinct attempts to Find/Put );
2389 puts( "a WORD into a linked list are 6,117,675,470(r.12++) and 5,845,989,790 );
2390 puts( "(r.12+++); also two 'if' sections were moved because they were executed );
2391 puts( "unnecessarily many times." );
2392 puts( "NoteB: Revision 13 uses BSTs instead of LLs, that is Linked-Lists were );
2393 puts( "replaced by Binary-Search-Trees, as a result for 22,202,980 distinct );
2394 puts( "words(out of 35,271,297) r.12+++ needs 225,548,268 total attempts to );
2395 puts( "Find/Put WORDS into linked lists where r.13 needs 121,674,042 total );
2396 puts( "attempts to Find/Put WORDS into Binary-Search-Trees. But this is a );
2397 puts( "significant boost in performance only for wordlists of million words." );
2398 puts( "NoteC: Revision 13+ gives only more statistics. Future revisions could lessen );
2399 puts( "number of attempts to Find/Put WORDS into Binary-Search-Trees );
2400 puts( "furthermore by making them at some point Perfectly-Balanced. But );
2401 puts( "for huge amount(multi-(m)billion) of distinct words the b-tree family );
2402 puts( "must come in, until then this is the leprechaunish niche." );
2403 puts( "NoteD: Revision 13++ has a little fix(2 unnecessary ZEROings, when a new word );
2404 puts( "is inserted, were deleted) and a fixed bug(13+ adds stupidly the );
2405 puts( "highest BST to the wordlist). Also B-Tree of order 3 is added as a );
2406 puts( "searching method. Main goal of B-Tree is to reduce number of );
2407 puts( "comparisons but at nasty cost: a precious time wasted to construct it );
2408 puts( "and twice more memory, i.e. one step forward two backward: this tree is );
2409 puts( "more effective than BST in cases of 2++ billion/million );
2410 puts( "different/distinct words." );
2411 puts( "The improvement which comes from using B-Tree of order 3 is about 200% );
2412 puts( "much more pleasing than I expected, for wikipedia-en.html.tar.wrd with );
2413 puts( "12,561,874 distinct words Total Attempts to Find/Put WORDS into: );
2414 puts( "Binary-Search-Trees was 61,895,043 while for );
2415 puts( "B-trees order 3 was 19,295,791." );
2416 puts( "NoteE: Revision 13+++ has a faster(not heavily tested yet) and with );
2417 puts( "better(0.6% to 1.1%) dispersion Fowler/Noll/Vo hash." );
2418 puts( "so called FNV1a hash. Revision 13++++ boosting: Leprechaun_Intel.exe );
2419 puts( "gives 1,256,187w/s for wikipedia-en.html.tar.wrd with FNV1_32_PRIME );
2420 puts( "107712257 with 3,551,736 dispersion for 'FNV1A_Hash_Granularity'." );
2421 puts( "NoteF: For old r.12+ a USB connected HDD crippled test: );
2422 puts( "for 'H:\>Leprechaun.exe static.wikipedia.org_downloads_2008-06_en.lst" );
2423 puts( "wikipedia-en.html.tar.wrd 5400" );
2424 puts( "where 223,674,511,360 wikipedia-en.html.tar );
2425 puts( "on laptop Toshiba Pentium T3400 2166 MHz with );
2426 puts( "Motherboard Name: Toshiba Satellite L305" );
2427 puts( "CPU Type: Mobile DualCore Intel Pentium, 2166 MHz (13 x 167)" );
2428 puts( "CPU Alias: Merom-1M" );
2429 puts( "L1 Code Cache: 32 KB per core" );
2430 puts( "L1 Data Cache: 32 KB per core" );
2431 puts( "L2 Cache: 1 MB (On-Die, ECC, ASC, Full-Speed)" );
2432 puts( "Bus Type: Dual DDR2 SDRAM" );
2433 puts( "Bus width: 128-bit" );
2434 puts( "Real Clock: 333 MHz (DDR)" );
2435 puts( "Effective Clock: 666 MHz" );
2436 puts( "EVEREST v5.00.1650 Memory Copy: 3725MB/s with timings 5-5-5-13" );
2437 puts( "result is logged to 'Leprechaun.Log'" );
2438 puts( "Bytes per second performance: 20,658,958/s" );
2439 puts( "words per second performance: 2,860,880w/s" );
2440 puts( "Input File with a list of TEXTUAL Files:" );
2441 puts( "static.wikipedia.org_downloads_2008-06_en.lst" );
2442 puts( "Size of all TEXTUAL Files: 223,674,511,360" );

```

```

2443 puts( " Word count: 30,974,750,142 of them 12,561,874 distinct" );
2444 puts( " Number of Files: 1" );
2445 puts( " Number of Lines: 2088618575" );
2446 puts( " Allocated memory in MB: 1920" );
2447 puts( " Words with length 01 occupy 0,033KB of 0,349KB given i.e. 09% utilization" );
2448 puts( " Words with length 02 occupy 0,033KB of 0,349KB given i.e. 09% utilization" );
2449 puts( " Words with length 03 occupy 0,037KB of 0,697KB given i.e. 05% utilization" );
2450 puts( " Words with length 04 occupy 0,151KB of 0,871KB given i.e. 17% utilization" );
2451 puts( " Words with length 05 occupy 0,744KB of 1,568KB given i.e. 47% utilization" );
2452 puts( " Words with length 06 occupy 1,470KB of 3,136KB given i.e. 46% utilization" );
2453 puts( " Words with length 07 occupy 2,605KB of 5,923KB given i.e. 43% utilization" );
2454 puts( " Words with length 08 occupy 3,296KB of 6,968KB given i.e. 47% utilization" );
2455 puts( " Words with length 09 occupy 3,714KB of 6,968KB given i.e. 53% utilization" );
2456 puts( " Words with length 10 occupy 3,483KB of 6,968KB given i.e. 49% utilization" );
2457 puts( " Words with length 11 occupy 3,235KB of 5,923KB given i.e. 54% utilization" );
2458 puts( " Words with length 12 occupy 2,691KB of 4,181KB given i.e. 64% utilization" );
2459 puts( " Words with length 13 occupy 2,230KB of 3,484KB given i.e. 64% utilization" );
2460 puts( " Words with length 14 occupy 1,718KB of 3,484KB given i.e. 49% utilization" );
2461 puts( " Words with length 15 occupy 1,357KB of 2,613KB given i.e. 51% utilization" );
2462 puts( " Words with length 16 occupy 1,063KB of 2,613KB given i.e. 40% utilization" );
2463 puts( " Words with length 17 occupy 0,814KB of 1,742KB given i.e. 46% utilization" );
2464 puts( " Words with length 18 occupy 0,617KB of 1,742KB given i.e. 35% utilization" );
2465 puts( " Words with length 19 occupy 0,485KB of 1,742KB given i.e. 27% utilization" );
2466 puts( " Words with length 20 occupy 0,402KB of 1,742KB given i.e. 23% utilization" );
2467 puts( " Words with length 21 occupy 0,327KB of 1,742KB given i.e. 18% utilization" );
2468 puts( " Words with length 22 occupy 0,274KB of 1,742KB given i.e. 15% utilization" );
2469 puts( " Words with length 23 occupy 0,224KB of 1,394KB given i.e. 16% utilization" );
2470 puts( " Words with length 24 occupy 0,190KB of 1,394KB given i.e. 13% utilization" );
2471 puts( " Words with length 25 occupy 0,162KB of 1,394KB given i.e. 11% utilization" );
2472 puts( " Words with length 26 occupy 0,136KB of 1,220KB given i.e. 11% utilization" );
2473 puts( " Words with length 27 occupy 0,119KB of 1,046KB given i.e. 11% utilization" );
2474 puts( " Words with length 28 occupy 0,107KB of 0,871KB given i.e. 12% utilization" );
2475 puts( " Words with length 29 occupy 0,091KB of 0,697KB given i.e. 13% utilization" );
2476 puts( " Words with length 30 occupy 0,080KB of 0,523KB given i.e. 15% utilization" );
2477 puts( " Words with length 31 occupy 0,076KB of 0,523KB given i.e. 14% utilization" );
2478 puts( " Total pseudo(including hash table) memory utilization: 42%" );
2479 puts( " Total real(wordlist's words vs allocated block) memory utilization: 60/1000" );
2480 puts( " Used value for third parameter in KB: 5400" );
2481 puts( " Use next time as third parameter: 3475-" );
2482 puts( " Time for making unsorted wordlist: 10827 second(s)" );
2483 puts( " Time for sorting unsorted wordlist: 10 second(s)" );
2484 puts( "NoteG: 2011-Mar-07: Fixed a small command line parsing bug." );
2485 puts( "" );
2486 puts( "Usage: Leprechaun InFile OutFile [bufferSize] [sortMethod] [treeMethod]" );
2487 puts( " <InFile>: Input file with files for Leprechauning, in WINDOWS console" );
2488 puts( " you can create it by 'E:\KAZEHOME>dir *.txt/s/b>Leprechaun.lst'" );
2489 puts( " <OutFile>: Output WORDLIST(sorted since r.9, CRLF) file" );
2490 puts( " <BufferSize>: Optional Dynamic RAM buffer in KB, default(and minimum)" );
2491 puts( " in the same time is 1023, i.e. omit or specify greater one" );
2492 puts( " <SortMethod>: Optional Sort Method, default is 'd,'" );
2493 puts( " A - InsertionSort" );
2494 puts( " B - InsertionX26Sort" );
2495 puts( " C - MultikeyQuickSortSort by J. Bentley, R. Sedgewick" );
2496 puts( " D - MultikeyQuickSortX26Sort by J. Bentley, R. Sedgewick" );
2497 puts( " <TreeMethod>: Optional Tree Method, default is 'X,'" );
2498 puts( " X - Binary-Search-Trees" );
2499 puts( " Y - B-Trees of order 3" );
2500 puts( "" );
2501 puts( "Have a nice Leprechauning." );
2502 puts( "For contacts: sanmayce@sanmayce.com" );
2503 puts( "Sanmayce Svalyatchx 'kaze', 2005 Feb 07(rev. 13_7pluses quadrupleton_r1: 2011 Jan 26)." );
2504 return( 1 );
2505 }
2506
2507 GRMBLhll[0]=0;
2508 GRMBLhll[1]=1;
2509 GRMBLhll[2]=1;
2510 GRMBLhll[3]=1;
2511 GRMBLhll[4]=1;
2512 GRMBLhll[5]=1;
2513 GRMBLhll[6]=1;
2514 GRMBLhll[7]=1;
2515 GRMBLhll[8]=1;
2516 GRMBLhll[9]=1;
2517 GRMBLhll[10]=1;
2518 GRMBLhll[11]=1;
2519 GRMBLhll[12]=15;
2520 GRMBLhll[13]=15;
2521 GRMBLhll[14]=15;
2522 GRMBLhll[15]=20;
2523 GRMBLhll[16]=30;
2524 GRMBLhll[17]=40;
2525 GRMBLhll[18]=50;
2526 GRMBLhll[19]=50;
2527 GRMBLhll[20]=50;
2528 GRMBLhll[21]=40;
2529 GRMBLhll[22]=40;
2530 GRMBLhll[23]=40;

```

```

2531 GRMBLhll[24]=30;
2532 GRMBLhll[25]=20;
2533 GRMBLhll[26]=20;
2534 GRMBLhll[27]=20;
2535 GRMBLhll[28]=20;
2536 GRMBLhll[29]=20;
2537 GRMBLhll[30]=10;
2538 GRMBLhll[31]=10;
2539
2540 if( ( fp_in = fopen( argv[1], "rb" ) ) == NULL )
2541 { printf( "Leprechaun: Can't open file %s \n", argv[1] ); return( 1 ); }
2542
2543 fseek( fp_in, 0L, SEEK_END );
2544 size_in = fte1l( fp_in );
2545 fseek( fp_in, 0L, SEEK_SET );
2546 printf( "Size of input file with files for Leprechauning: %lu\n", size_in );
2547
2548 if( ( fp_outLOG = fopen( "Leprechaun.LOG", "a+" ) ) == NULL )
2549 { printf( "Leprechaun: Can't open file Leprechaun.LOG.\n" ); return( 1 ); }
2550
2551 // argc is 4|5|6 due to eventual missing BufferSize
2552 if( argc == 4 ) // not 6 due to eventual missing BufferSize and SortMethod
2553 k_FIX = 3;
2554 if( argc == 5 ) // not 6 due to eventual missing BufferSize or SortMethod
2555 k_FIX = 4;
2556 if( argc == 6 )
2557 k_FIX = 5;
2558 if (*argv[k_FIX] == 'y' || *argv[k_FIX] == 'y') BStorBtree = 1;
2559
2560 if( argc == 4 || argc == 5 || argc == 6 ) Thunderwith = atoi( argv[3] );
2561 else Thunderwith = 527; // for r.12: 527=17*31 this is minimum because of 4096*1*4=16KB+ needed for each buffer!
2562 // for r.12a: 1023=33*31 this is minimum because of 4096*2*4=32KB+ needed for each buffer!
2563 if( Thunderwith < 1023 ) { Thunderwith = 1023; }
2564 LetterBuffer = Thunderwith * 1024;
2565 WHOLELetter_BufferSize = 0;
2566 for( i = 1; i <= 31; i++ )
2567 { OffsetsInBuffer[i-1] = 0;
2568 for( j = 1; j <= i; j++ )
2569 { OffsetsInBuffer[i-1] = OffsetsInBuffer[i-1] + (GRMBLhll[(int)(j-1)] * LetterBuffer)/31;
2570 }
2571 WHOLELetter_BufferSize = WHOLELetter_BufferSize + (GRMBLhll[(int)i] * LetterBuffer)/31; // Make soon 32 in order to shift >>5
2572 GRMBLPoolAgain[(int)i] = (GRMBLhll[(int)i] * LetterBuffer)/31;
2573 }
2574 memory_size = 26 * WHOLELetter_BufferSize + 1 + 64;
2575 printf( "Allocating memory %uMB ...", (memory_size>>20)+1 );
2576 pointerFlushUNALIGN = (char *)malloc( memory_size );
2577 if( pointerFlushUNALIGN == NULL )
2578 { puts( "\nLeprechaun: Needed memory allocation denied!\n" ); return( 1 ); }
2579 pointerFlush = pointerFlushUNALIGN + 64 - (((size_t)pointerFlushUNALIGN) % 64); // 13_6+
2580 //offset=64-int((long)data&63);
2581 printf( "OK\n" );
2582 fprintf( fp_outLOG, "Leprechaun report:\n" );
2583
2584 // Check once for ever whether allocated memory is ZEROed! Answer: YES
2585 //for( i = 0; i < memory_size; i++ )
2586 // if (*(char *)pointerFlush+i)!=0 printf("NON-ZERO encountered, so 'NO'.");
2587
2588 for( i = 0; i < 26; i++ )
2589 { for( k = 1; k <= 31; k++ )
2590 { bufend[i*31+k-1] = pointerFlush + i * WHOLELetter_BufferSize + OffsetsInBuffer[k-1]; // i*31+k-1 must be 0..805
2591 if (i==25) { MAXusedBuffer[k] = (unsigned long)bufend[i*31+k-1]; }
2592 for( j = 0; j < (NumberOfSLOTS+1)*4; j++ ) // ? memset(bufend[i],0,(NumberOfSLOTS+1)*4);
2593 { *bufend[i*31+k-1]++ = 0;
2594 //++bufend[i*31+k-1];
2595 }
2596 if (i==25) { MAXusedBuffer[k] = (unsigned long)bufend[i*31+k-1]-MAXusedBuffer[k]; }
2597 bufNumberOfWords[i*31+k-1]=0;
2598 //for( j = 0; j < NumberOfSLOTS; j++ )
2599 //bufNoWps[i*31+k-1][j]=0;
2600 }
2601 }
2602
2603 // PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM PROGRAM
2604 (void) time(&t1);
2605 MeInitchka = 0;
2606 WORDcount = 0; // Total word count i.e. for all files!
2607 WORDcountDistinct = 0;
2608 NumberOfFiles = 0;
2609 NumberOfLines = 0;
2610 FilesLEN = 0;
2611 LINE10len = 0;
2612
2613 for( k = 0; k < size_in; k++ )
2614 {
2615 fread( &workbyte, 1, 1, fp_in );
2616 if( workbyte != 10 )
2617 { if( workbyte != 13 ) // NON UNIX
2618 { if( LINE10len < 255 ) { LINE10[ LINE10len ] = workbyte; }

```



```

2619         LINE10len++;
2620     }
2621     else
2622     {
2623     }
2624 }
2625 else
2626     { if( 1 <= LINE10len && LINE10len <= 255 )
2627       { LINE10[ LINE10len ] = 0;
2628 if( ( fp_inLINE = fopen( LINE10, "rb" ) ) == NULL )
2629   { printf( "Leprechaun: Can't open file %s\n", LINE10 ); return( 1 ); }
2630
2631 //fseek( fp_inLINE, 0L, SEEK_END ); //Rev. 12
2632 //size_inLINE = ftell( fp_inLINE ); //Rev. 12
2633 //fseek( fp_inLINE, 0L, SEEK_SET ); //Rev. 12
2634
2635 #if defined(_WIN32_ENVIRONMENT_)
2636 // 64bit:
2637 _lseeki64( fileno(fp_inLINE), 0L, SEEK_END );
2638 size_inLINESIXFOUR = _telli64( fileno(fp_inLINE) );
2639 _lseeki64( fileno(fp_inLINE), 0L, SEEK_SET );
2640 #else
2641 // 64bit:
2642 fseeko( fp_inLINE, 0L, SEEK_END );
2643 size_inLINESIXFOUR = ftello( fp_inLINE );
2644 fseeko( fp_inLINE, 0L, SEEK_SET );
2645 #endif /* defined(_WIN32_ENVIRONMENT_) */
2646
2647 printf( "Size of Input TEXTual file: %s\n", _ui64toaKAZEComma(size_inLINESIXFOUR, l1ToAdigits, 10) );
2648 FilesLEN = FilesLEN + size_inLINESIXFOUR;
2649 NumberOfFiles++;
2650
2651 //~~~~~
2652 wrdlen = 0;
2653 for( i = 0; i < size_inLINESIXFOUR; i++)
2654 {
2655     //~~~~~ Buffering fread [
2656     if (workkoffset == -1) {
2657         if (i + 1024*128 < size_inLINESIXFOUR) {
2658             fread( &workk[0], 1, 1024*128, fp_inLINE );
2659             workkoffset = 0;
2660             workbyte = workk[workkoffset];
2661         } else
2662             fread( &workbyte, 1, 1, fp_inLINE );
2663     } else {
2664         workkoffset++;
2665         workbyte = workk[workkoffset];
2666         if (workkoffset == 1024*128 - 1) workkoffset = -1;
2667     }
2668     //~~~~~ Buffering fread ]
2669
2670     if( isalpha( workbyte ) )
2671     {
2672         if( wrdlen < 31 )
2673             { wrd[ wrdlen ] = tolower( workbyte ); }
2674         wrdlen++;
2675     }
2676
2677     if ( workbyte < 'A' ) // Most characters are under alphabet - only one if
2678     {
2679
2680 ElStupido:
2681         // This fragment is MIRRORed: #1 copy [
2682         if (workbyte == 10) {NumberOfLines++;}
2683
2684 // Quadruple! [
2685 // Sliding window for 'wrđ': The incoming string 'a lot of things must' becomes 'a_lot_of_things' and 'lot_of_things_must':
2686
2687 // ain_t_that_a
2688 // didn_t_feel_a
2689 // i_didn_t_feel
2690 // t_feel_a_thing
2691 // t_that_a_cake
2692
2693 // 316
2694 // 00:17:55,859 --> 00:17:58,447
2695 // Ain't that a cake ? I didn't feel a thing !
2696
2697         if ( PLE_words_INITflag == 0 && ( PLE_words != 0 ) || ( PLE_words == 0 && wrdlen != 0 ) )
2698         if ( workbyte == '.' || workbyte == '!' || workbyte == '?' || workbyte == ':' || workbyte == ';' || workbyte == ',' )
2699         || workbyte == '\t' ) {
2700             PLE_words_INITflag = 1;
2701         }
2702 // Quadruple! ]
2703
2704         if ( 1 <= wrdlen && wrdlen <= 31 )
2705         {
2706             wrd[ wrdlen ] = 0;

```

```

2706         // OTKACHAM: 1<<17-1 gives 65536 i.e. '-' have had high priority than '<<'
2707         //Next line gives error due to mix of '&' and 'double'
2708         if ((++WORDcount & ((1<<20)-1)) == 0)
2709         { _ui64toaKAZEComma(WORDcount, l1ToAdigits, 10);
2710         //printf( "word count: %s(%lu/128 done)\r", l1ToAdigits, ((long long)i*100) / size_inLINESIXFOUR );
2711 //++Melnitckha;
2712 //Melnitckha = Melnitckha & 4;
2713 //if (Melnitckha == 0){ printf( "\n"; word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEComma(WORDcount, l1ToAdigits, 10),
2714 //ui64toaKAZEComma((unsigned long long)WORDcountDistinct, l1ToAdigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
2715 //if (Melnitckha == 1){ printf( "\n"; word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEComma(WORDcount, l1ToAdigits, 10),
2716 //ui64toaKAZEComma((unsigned long long)WORDcountDistinct, l1ToAdigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
2717 //if (Melnitckha == 2){ printf( "\n"; word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEComma(WORDcount, l1ToAdigits, 10),
2718 //ui64toaKAZEComma((unsigned long long)WORDcountDistinct, l1ToAdigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
2719 //if (Melnitckha == 3){ printf( "\n"; word count: %s of them %s distinct; Done: %lu/64\r", _ui64toaKAZEComma(WORDcount, l1ToAdigits, 10),
2720 //ui64toaKAZEComma((unsigned long long)WORDcountDistinct, l1ToAdigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR ); }
2721 Melnitckha = Melnitckha & 3; // 0 1 2 3: 00 01 10 11
2722 printf( "\n"; word count: %s of them %s distinct; Done: %lu/64\r", Auberge[Melnitckha+], _ui64toaKAZEComma(WORDcount, l1ToAdigits, 10),
2723 //ui64toaKAZEComma((unsigned long long)WORDcountDistinct, l1ToAdigits2, 10), ((long long)i<<6) / size_inLINESIXFOUR );
2724 }
2725
2726 // Quadruple! [
2727 PLE_words++;
2728 if (PLE_words == 1)
2729     strcpy( wrd1st, wrd );
2730 else if (PLE_words == 2)
2731     strcpy( wrd2nd, wrd );
2732 else if (PLE_words == 3)
2733     strcpy( wrd3rd, wrd );
2734 else if (PLE_words == 4) {
2735     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+1+1+1; // '_''_'
2736 //wrdlen = strlen(wrd);
2737 if ( wrdlen <= 31 ) {
2738     strcpy(wrd, wrd1st);
2739     strcat(wrd, DelimiterUnderscore);
2740     strcat(wrd, wrd2nd);
2741     strcat(wrd, DelimiterUnderscore);
2742     strcat(wrd, wrd3rd);
2743     strcat(wrd, DelimiterUnderscore);
2744     strcat(wrd, wrd4th);
2745 }
2746 }
2747 else {
2748     PLE_words = 4;
2749     strcpy( wrd1st, wrd2nd );
2750     strcpy( wrd2nd, wrd3rd );
2751     strcpy( wrd3rd, wrd4th );
2752     strcpy( wrd4th, wrd );
2753     wrdlen = strlen(wrd1st)+strlen(wrd2nd)+strlen(wrd3rd)+strlen(wrd4th)+1+1+1; // '_''_'
2754 //wrdlen = strlen(wrd);
2755 if ( wrdlen <= 31 ) {
2756     strcpy(wrd, wrd1st);
2757     strcat(wrd, DelimiterUnderscore);
2758     strcat(wrd, wrd2nd);
2759     strcat(wrd, DelimiterUnderscore);
2760     strcat(wrd, wrd3rd);
2761     strcat(wrd, DelimiterUnderscore);
2762     strcat(wrd, wrd4th);
2763 }
2764 }
2765 // Quadruple! ]
2766
2767     if ( ( PLE_words == 4 ) && ( 12 <= wrdlen ) && ( wrdlen <= 31 ) ) {
2768         LetterOffset = (int)( wrd[0] - 'a' ) * 31 + (wrdlen-1); // 0..805
2769         //BufStart = pointerflush + LetterOffset * LetterBuffer; // OLD
2770
2771         BufStart = pointerflush + (int)( wrd[0] - 'a' ) * WHOLELetter_BufferSize + OffsetsInBuffer[wrdlen-1];
2772         // Above line and Below line are equal
2773         //BufStart = pointerflush + (LetterOffset / 31) * WHOLELetter_BufferSize + OffsetsInBuffer[LetterOffset % 31];
2774
2775         //Slot = KuxHash3plus(wrd)<<2; //13++
2776         //Slot = FNWIA_Hash_SHIFTless_XORless(wrd)<<2; //13+++
2777         //Slot = FNWIA_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13++++
2778         //Slot = FNWIA_Hash_4_OCTETS_31(wrd, wrdlen>>2)<<2; //13++++
2779
2780 //
2781 if (wrdlen<19) // 4x4+3=19
2782     Slot = FNWIA_Hash_4_OCTETS(wrd, wrdlen>>2)<<2; //13++++
2783 else
2784     // 2x8+4=20 i.e. First contains 5 clashes
2785     Slot = FNWIA_Hash_8_OCTETS(wrd, wrdlen>>3)<<2; //13++++
2786 //
2787 if (wrdlen<19) // 4x4+3=19 i.e. last contains 7 clashes
2788     Slot = FNWIA_Hash_Granularity(wrd, wrdlen>>2, 2)<<2; //13++++
2789 //else
2790     // 2x8+4=20 i.e. first contains 6 clashes
2791     Slot = FNWIA_Hash_Granularity(wrd, wrdlen>>3, 3)<<2; //13++++
2792 //
2793 //Slot = FNWIA_Hash_8_OCTETS(wrd, wrdlen>>3)<<2; //13_7p
2794 //Slot = HashFNWIA_unrolled_Final(wrd, wrdlen)<<2; //13_7p
2795 //Slot = Hash1fa1fa_HALF(wrd, wrdlen)<<2; //13_7p

```

```

2789 //Slot = Hash_Alfa(wrd, wrdlen)<<2; //13_7p
2790 // Slot = Sixtinsensitive(wrd, wrdlen)<<2; //13_7p
2791 Slot = FNVA_Hash_Jesteress(wrd, wrdlen)<<2; //13_7p
2792 // Slot = FNVA_Hash_Jester(wrd, wrdlen)<<2; //13_7p
2793
2794 /*
2795 hashAlfa = 7;
2796 for(iAlfa = 0; iAlfa < (wrdlen & -2); iAlfa += 2) {
2797 hashAlfa = (17+9) * ((17+9) * hashAlfa + (wrd[iAlfa])) + (wrd[iAlfa+1]);
2798 }
2799 if(wrdlen & 1)
2800 hashAlfa = (17+9) * hashAlfa + (wrd[wrdlen-1]);
2801
2802 Slot = ((hashAlfa ^ (hashAlfa >> 16)) & 8191)<<2; //13_7p
2803 */
2804
2805 memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
2806 //; Line 917
2807 // mov     edx, DWORD PTR [eax+ebp]
2808 // add     esp, 4
2809 // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
2810 //PseudoLinkedPointer = (unsigned long)*(long *) (BufStart+Slot);
2811 //; Line 919
2812 // mov     edx, DWORD PTR [eax+ebp]
2813 // add     esp, 4
2814 // while (count--) {
2815 // *(char *)dst = *(char *)src;
2816 // dst = (char *)dst + 1;
2817 // src = (char *)src + 1;
2818 // }
2819 if (BStorBtree != 1)
2820 {
2821 // @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ BST fragment [
2822 // if (PseudoLinkedPointer == 0) // means EMPTY-SLOT
2823 // {
2824 // if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 < GRMBLH11[(int)wrdlen] * LetterBuffer/31 ) // OLD slower
2825 // if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 + 4 < GRMBLFOOAgain[(int)wrdlen] ) // +4 more for BST instead
of LL
2826 {
2827 memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
2828 //; Line 932
2829 // mov     DWORD PTR [eax+ebp], esi
2830 // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
2831 //*(long *) (BufStart+Slot) = *(long *) &bufend[LetterOffset];
2832 //; Line 936
2833 // mov     DWORD PTR [eax+ebp], esi
2834
2835 // Below 3 lines are commented due to experiment below malloc which shows that allocated memory is ZEROed.
2836 //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not: Means PseudoLinkedPointerL =
0
2837 //; Line 940
2838 // mov     ecx, DWORD PTR [ebp+32768]
2839 // ?! DANGEROUS: above and below lines are(must:long must be 4bytes) identical
2840 //*(long *) (bufend[LetterOffset] + 4) = *(long *) &BufStart[NumberOfSLOTS*4];
2841 //; Line 944
2842 // mov     ecx, DWORD PTR [ebp+32768]
2843 //bufend[LetterOffset] = bufend[LetterOffset] + 4;
2844 //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not: Means PseudoLinkedPointerR =
0
2845 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4; // + 4 due to above commenting
2846 memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
2847 //bufNowps[LetterOffset][Slot]++; // ?! crashes
2848 bufend[LetterOffset] = bufend[LetterOffset] + wrdlen;
2849 if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
2850 }
else
2851 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
2852 printf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
2853 printf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, l1toADigits, 10) );
2854 printf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, l1toADigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, l1toADigits2, 10) );
2855 printf( fp_outLOG, "Number of Files: %lu\n", NumberOfFiles );
2856 printf( fp_outLOG, "Number of Lines: %lu\n", NumberOfLines );
2857 printf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
2858 printf( fp_outLOG, "Total Attempts to Find/Put WORDS into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, l1toADigits,
10) );
2859 for( k = 1; k < 32; k++)
2860 { printf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s% utilization\n", _ui64toaKAZEzerocomma(k, l1toADigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, l1toADigits2, 10)+(26-5), _ui64toaKAZEzerocomma(((GRMBLH11[(int)k] *
LetterBuffer/31)>>10)+1, l1toADigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/(GRMBLH11[(int)k] *
LetterBuffer/31), l1toADigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
2861 }
2862 printf( fp_outLOG, "used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
2863 printf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
2864 return( 1 );
2865 }
2866 }
2867 }

```

```

2868 else // means USED-SLOT
2869 { FoundInLinkedList = 0;
2870 while (PseudoLinkedPointer != 0 && FoundInLinkedList == 0)
2871 {
2872 if (memcmp(PseudoLinkedPointer+4+4, wrd, wrdlen) == 0)
2873 while ( --count && *(char *)buf1 == *(char *)buf2 ) {
2874 buf1 = (char *)buf1 + 1;
2875 buf2 = (char *)buf2 + 1;
2876 }
2877 return( *((unsigned char *)buf1) - *((unsigned char *)buf2) );
2878 }
2879 FoundInLinkedList = 1;
2880 }
else // i.e < or >
2881 {
2882 if (memcmp(PseudoLinkedPointer+4+4, wrd, wrdlen) > 0)
2883 memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
2884 else
2885 {
2886 PseudoLinkedPointer = PseudoLinkedPointer + 4;
2887 memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
2888 }
2889 }
if (PseudoLinkedPointerNEW == 0)
2890 {
2891 //if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 < GRMBLH11[(int)wrdlen] * LetterBuffer/31 ) // OLD slower
2892 if( (unsigned long)(bufend[LetterOffset] - BufStart) + wrdlen + 4 + 4 < GRMBLFOOAgain[(int)wrdlen] )
2893 { memcpy( PseudoLinkedPointer, &bufend[LetterOffset], 4 );
2894 // Below 3 lines are commented due to experiment below malloc which shows that allocated memory is ZEROed.
2895 //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not
2896 //bufend[LetterOffset] = bufend[LetterOffset] + 4;
2897 //memcpy( bufend[LetterOffset], &BufStart[NumberOfSLOTS*4], 4 ); // means next exists not
2898 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4; // + 4 due to above commenting
2899 memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
2900 //bufNowps[LetterOffset][Slot]++; // ?! crashes
2901 bufend[LetterOffset] = bufend[LetterOffset] + wrdlen;
2902 if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
2903 }
else
2904 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
2905 printf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
2906 printf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, l1toADigits, 10) );
2907 printf( fp_outLOG, "Word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, l1toADigits, 10), _ui64toaKAZEcomma((unsigned long
long)WORDcountDistinct, l1toADigits2, 10) );
2908 printf( fp_outLOG, "Number of Files: %lu\n", NumberOfFiles );
2909 printf( fp_outLOG, "Number of Lines: %lu\n", NumberOfLines );
2910 printf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
2911 printf( fp_outLOG, "Total Attempts to Find/Put WORDS into Binary-Search-Trees: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, l1toADigits,
10) );
2912 for( k = 1; k < 32; k++)
2913 { printf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s% utilization\n", _ui64toaKAZEzerocomma(k, l1toADigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, l1toADigits2, 10)+(26-5), _ui64toaKAZEzerocomma(((GRMBLH11[(int)k] *
LetterBuffer/31)>>10)+1, l1toADigits3, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/(GRMBLH11[(int)k] *
LetterBuffer/31), l1toADigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
2914 }
2915 printf( fp_outLOG, "used value for third parameter in KB: %lu\n", (unsigned long)Thunderwith );
2916 printf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
2917 return( 1 );
2918 }
}
PseudoLinkedPointer = PseudoLinkedPointerNEW;
WORDcountAttemptsToPut++;
} // while
}
2927 // @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ BST fragment [
2928 // else
2929 // {
2930 // ##### B-tree order 3 fragment [
2931 //
2932 // LEAF structure: [LeftPointer][MiddlePointer][RightPointer][LeftWord][RightWord]
2933 // 4bytes 4bytes 4bytes wrdlen wrdlen * * <- if *(char *)==0 means the word cell is empty
2934 //
2935 // ALL B-tree order 3 fragment consists of 3 sub-fragments:
2936 // [1] Search 2] if Search failed Trasrascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search 3] Insert Iterative
2937 // [1] Search [ _____1407 line in C - see below: whole Search in assembler_____
2938 // if (PseudoLinkedPointer == 0) // means EMPTY-SLOT
2939 {
2940 if( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFOOAgain[(int)wrdlen] ) // +4 more for BST
instead of LL; + more: (see LEAF)
2941 {
2942 memcpy( BufStart+Slot, &bufend[LetterOffset], 4 );
2943 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
2944 memcpy( bufend[LetterOffset], wrd, wrdlen ); WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
2945 bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
2946 if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);}
2947 }
}

```

```

2948     }
2949     else
2950     { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n" );
2951     fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
2952     fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, 11ToADigits, 10) );
2953     fprintf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORdcount, 11ToADigits, 10), _ui64toaKAZEcomma((unsigned long)
long)WORdcountDistinct, 11ToADigits, 10) );
2954     fprintf( fp_outLOG, "Number Of Files: %lu\n", NumberOfFiles );
2955     fprintf( fp_outLOG, "Number Of Lines: %lu\n", NumberOfLines );
2956     fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>20)+1 );
2957     fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORdcountAttemptsToPut, 11ToADigits, 10)
);
2958     for( k = 1; k < 32; k++)
2959     { fprintf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s% utilization\n", _ui64toaKAZEzerocomma(k, 11ToADigits, 10)+(26-
2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, 11ToADigits, 10)+(26-5), _ui64toaKAZEzerocomma(((GRMBLh1l1[(int)k] *
LetterBuffer)/31)>>10)+1, 11ToADigits, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/((GRMBLh1l1[(int)k] *
LetterBuffer)/31), 11ToADigits, 10)+(26-2), "%0" ); // 26 are all 26-DESIREd=24
2960     }
2961     fprintf( fp_outLOG, "used value for third parameter in KB: %lu\n", (unsigned long)thunderwith );
2962     fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n");
2963     return( 1 );
2964     }
2965     FoundInLinkedList = 1;
2966     }
2967     else // means USED-SLOT
2968     { FoundInLinkedList = 0;
2969     while (PseudoLinkedPointer != 0 && FoundInLinkedList == 0)
2970     {
2971     // ***** 'P W P' section [
2972     // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
2973     // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wrDlen) != 0 )
2974     // here ALWAYS LW exists: no need for existence check - line below
2975     // if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
2976     if (memcmp(PseudoLinkedPointer+4+4+4, wrd, wrDlen) > 0) // go LP
2977     { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 0, 4 ); //LP
2978     PseudoLinkedPointer = PseudoLinkedPointerNEW;
2979     }
2980     else if (memcmp(PseudoLinkedPointer+4+4+4, wrd, wrDlen) < 0) // go RP or MP
2981     { // RW existence check - line below:
2982     if ( *(char *) (PseudoLinkedPointer+4+4+4+wrDlen) != 0 ) // RW exists
2983     { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
2984     // ++++++
2985     // ***** 'P W P' section 2 [
2986     // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
2987     // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wrDlen) != 0 )
2988     // here ALWAYS RW exists: no need for existence check - line below
2989     // if ( *(char *) (PseudoLinkedPointer+4+4+4+wrDlen) != 0 )
2990     if (memcmp(PseudoLinkedPointer+4+4+4+wrDlen, wrd, wrDlen) > 0) // go MP
2991     { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
2992     PseudoLinkedPointer = PseudoLinkedPointerNEW;
2993     }
2994     else if (memcmp(PseudoLinkedPointer+4+4+4+wrDlen, wrd, wrDlen) < 0) // go RP
2995     { // No ?w after RW - go RP
2996     memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
2997     PseudoLinkedPointer = PseudoLinkedPointerNEW;
2998     }
2999     else FoundInLinkedList = 1; // wrd is RW
3000     WORdcountAttemptsToPut++;
3001     // ***** 'P W P' section 2 ]
3002     // ++++++
3003     }
3004     else // RW empty - go MP
3005     { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
3006     PseudoLinkedPointer = PseudoLinkedPointerNEW;
3007     }
3008     }
3009     else FoundInLinkedList = 1; // wrd is LW
3010     WORdcountAttemptsToPut++;
3011     // ***** 'P W P' section ]
3012     // while
3013     WORdcountAttemptsToPut--; // - 1 due to BST way of counting i.e. direct hash hit is not counted only successors
3014     }
3015     // ] Search ] _____ 1484 line in C - see below: whole Search in assembler _____
3016
3017     /*
3018     ; Line 1397
3019     jmp $L2139
3020     $L2042:
3021     ; Line 1408
3022     test edx, edx
3023     jne SHORT $L2110
3024     ; Line 1410
3025     mov ecx, DWORD PTR _bufend$[esp+esi*4+892340]
3026     mov edi, DWORD PTR _GRMBLFoolAgain$[esp+ebx*4+892340]
3027     lea edx, DWORD PTR _bufend$[esp+esi*4+892340]
3028     lea esi, DWORD PTR [ebx+ebx+12]
3029     sub esi, ebp
3030     add esi, ecx

```

```

3031     cmp esi, edi
3032     mov DWORD PTR tv4122[esp+892340], edx
3033     jae $L2113
3034     ; Line 1412
3035     mov DWORD PTR [eax+ebp], ecx
3036     ; Line 1413
3037     lea eax, DWORD PTR [ecx+12]
3038     ; Line 1436
3039     jmp $L2749
3040     $L2110:
3041     ; Line 1437
3042     mov DWORD PTR _FoundInLinkedList$[esp+892340], 0
3043     npad11
3044     $L2141:
3045     ; Line 1438
3046     mov eax, DWORD PTR _FoundInLinkedList$[esp+892340]
3047     test eax, eax
3048     jne $L2142
3049     ; Line 1445
3050     lea ebp, DWORD PTR [edx+12]
3051     mov ecx, ebx
3052     lea edi, DWORD PTR _wrd$[esp+892340]
3053     mov esi, ebp
3054     xor eax, eax
3055     repe cmpsb
3056     je SHORT $L2682
3057     sbb eax, eax
3058     sbb eax, -1
3059     $L2682:
3060     test eax, eax
3061     jle SHORT $L2143
3062     ; Line 1447
3063     mov edx, DWORD PTR [edx]
3064     ; Line 1449
3065     jmp $L2153
3066     $L2143:
3067     mov ecx, ebx
3068     lea edi, DWORD PTR _wrd$[esp+892340]
3069     mov esi, ebp
3070     xor eax, eax
3071     repe cmpsb
3072     je SHORT $L2640
3073     sbb eax, eax
3074     sbb eax, -1
3075     $L2640:
3076     test eax, eax
3077     jge SHORT $L2145
3078     ; Line 1451
3079     mov cl, BYTE PTR [edx+ebx+12]
3080     test cl, cl
3081     lea eax, DWORD PTR [edx+ebx+12]
3082     je SHORT $L2147
3083     ; Line 1459
3084     mov ecx, ebx
3085     lea edi, DWORD PTR _wrd$[esp+892340]
3086     mov esi, eax
3087     xor ebp, ebp
3088     repe cmpsb
3089     je SHORT $L2695
3090     sbb ebp, ebp
3091     sbb ebp, -1
3092     $L2695:
3093     test ebp, ebp
3094     jle SHORT $L2148
3095     ; Line 1461
3096     mov edx, DWORD PTR [edx+4]
3097     ; Line 1463
3098     jmp SHORT $L2151
3099     $L2148:
3100     mov esi, eax
3101     mov ecx, ebx
3102     lea edi, DWORD PTR _wrd$[esp+892340]
3103     xor eax, eax
3104     repe cmpsb
3105     je SHORT $L2642
3106     sbb eax, eax
3107     sbb eax, -1
3108     $L2642:
3109     test eax, eax
3110     jge SHORT $L2150
3111     ; Line 1466
3112     mov edx, DWORD PTR [edx+8]
3113     ; Line 1468
3114     jmp SHORT $L2151
3115     $L2150:
3116     mov DWORD PTR _FoundInLinkedList$[esp+892340], 1
3117     $L2151:
3118     ; Line 1469

```

```

3119 mov ecx, DWORD PTR _WORDcountAttemptsToPut$[esp+892340]
3120 mov eax, DWORD PTR _WORDcountAttemptsToPut$[esp+892344]
3121 add ecx, 1
3122 adc eax, 0
3123 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892340], ecx
3124 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892344], eax
3125 ; Line 1473
3126 jmp SHORT $L2153
3127 $L2147:
3128 ; Line 1475
3129 mov edx, DWORD PTR [edx+4]
3130 ; Line 1478
3131 jmp SHORT $L2153
3132 $L2145:
3133 mov DWORD PTR _FoundInLinkedList$[esp+892340], 1
3134 $L2153:
3135 ; Line 1479
3136 mov esi, DWORD PTR _WORDcountAttemptsToPut$[esp+892340]
3137 mov ecx, DWORD PTR _WORDcountAttemptsToPut$[esp+892344]
3138 add esi, 1
3139 adc ecx, 0
3140 test edx, edx
3141 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892340], esi
3142 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892344], ecx
3143 jne $L2141
3144 $L2142:
3145 ; Line 1482
3146 mov edx, DWORD PTR _WORDcountAttemptsToPut$[esp+892340]
3147 mov ecx, DWORD PTR _WORDcountAttemptsToPut$[esp+892344]
3148 or eax, -1
3149 add edx, eax
3150 adc ecx, eax
3151 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892344], ecx
3152 mov DWORD PTR _WORDcountAttemptsToPut$[esp+892340], edx
3153 $L2139:
3154 */
3155
3156 if (FoundInLinkedList == 0)
3157 {
3158 // 2] if Search failed Trasirascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search [
3159 // 'TracingSearch' is the same as 'Search' except that adds the trail in my simulated stack,
3160 // the goal is not to waste time in 'Search' by dealing with no needed trail in case of not 'Insert'.
3161 // Simulated stack contains pairs of 'Address of ParentLEAF' + 'Offset of ParentPointer in ParentLEAF i.e. 0 for LP, 4 for MP, 8 for RP'.
3162 // 'Offset ...' saves unnecessary comparisons of NEWWORD which after splitting goes up.
3163 memcpy( &PseudoLinkedPointerNEW, BufStart+Slot, 4 );
3164 StackPtr = 0;
3165 while (PseudoLinkedPointer != 0)
3166 {
3167 // ***** 'P W P' section [
3168 // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
3169 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
3170 // here ALWAYS LW exists: no need for existence check - line below
3171 // if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
3172 if ( memcmp(PseudoLinkedPointer+4+4+4, wrd, wrdlen) > 0 ) // go LP
3173 memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 0, 4 ); //LP
3174 if (StackPtr > 65536*3-1-1) { printf( "\nLeprechaun: Failure! 'b-tree order 3' simulated stack overflow!\n" ); return( 13 );}
3175 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
3176 BSTstack[StackPtr] = 0; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
3177 PseudoLinkedPointer = PseudoLinkedPointerNEW;
3178 }
3179 else if ( memcmp(PseudoLinkedPointer+4+4+4, wrd, wrdlen) < 0 ) // go RP or MP
3180 { // RW existence check - line below:
3181 if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 ) // RW exists
3182 { // Here all 'P W P' section is repeated; the way of handling case when dynamic number of words in leaf
3183 // ++++++
3184 // ***** 'P W P' section 2 [
3185 // LW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4) != 0 )
3186 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
3187 // here ALWAYS RW exists: no need for existence check - line below
3188 // if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
3189 if ( memcmp(PseudoLinkedPointer+4+4+4+wordlen, wrd, wrdlen) > 0 ) // go MP
3190 memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
3191 if (StackPtr > 65536*3-1-1) { printf( "\nLeprechaun: Failure! 'b-tree order 3' simulated stack overflow!\n" ); return( 13 );}
3192 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
3193 BSTstack[StackPtr] = 4; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
3194 PseudoLinkedPointer = PseudoLinkedPointerNEW;
3195 }
3196 else if ( memcmp(PseudoLinkedPointer+4+4+4+wordlen, wrd, wrdlen) < 0 ) // go RP
3197 { // No 'P' after 'W' - go RP
3198 memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4 + 4, 4 ); //RP
3199 if (StackPtr > 65536*3-1-1) { printf( "\nLeprechaun: Failure! 'b-tree order 3' simulated stack overflow!\n" ); return( 13 );}
3200 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
3201 BSTstack[StackPtr] = 8; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
3202 PseudoLinkedPointer = PseudoLinkedPointerNEW;
3203 }
3204 else FoundInLinkedList = 1; // wrd is RW
3205 // ***** 'P W P' section 2 ]
3206 // ++++++

```

```

3207 }
3208 else // RW empty - go MP
3209 { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer + 4, 4 ); //MP
3210 if (StackPtr > 65536*3-1-1) { printf( "\nLeprechaun: Failure! 'b-tree order 3' simulated stack overflow!\n" ); return( 13 );}
3211 BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //pt to visited leaf
3212 BSTstack[StackPtr] = 4; ++StackPtr; //LPoffset=0;MPoffset=4;RPoffset=8;
3213 PseudoLinkedPointer = PseudoLinkedPointerNEW;
3214 }
3215 }
3216 else FoundInLinkedList = 1; // wrd is LW
3217 // ***** 'P W P' section ]
3218 // while
3219 // 2] if Search failed Trasirascht(pushing in stack PseudoLinkedPointer(visited LEAFs)) Search ]
3220 // 3] Insert Iterative [
3221 // There are total 4 situations:
3222 // Case #1: Outer NODE(including ROOT) [ ][ ][ ][ ][LW][ ]
3223 // Case #2: Outer NODE(including ROOT) [ ][ ][ ][ ][LW][RW] Split Occurs ---- | 'wrdUP' (wrdlen bytes)
3224 // Case #3: ROOT [ ][MP][ ][ ][LW][ ]
3225 // Case #4: Inner NODE(including ROOT) [LP][MP][RP][LW][RW] Split Occurs --- | &
3226 // | 'PseudoLinkedPointerNEW' (ptr to NEW LEAF)
3227 // ARE GOING UP
3228 // There are total 2 situations for PARENT LEAF: <-----
3229 // Case #3: [LP][MP][ ][ ][LW][ ]
3230 // Case #4: [LP][MP][RP][LW][RW] Split occurs
3231
3232 // ~ First deal alonely with the OUTER NODE(LEAF) where Search stopped i.e Case #1 & Case #2:
3233 PoffsetInLEAF = BSTstack[--StackPtr];
3234 PseudoLinkedPointer = BSTstack[--StackPtr];
3235 // NOTE: ONE LEAF IS FULL ONLY WHEN LAST CELL FOR KEY(here RW) EXISTS!
3236 // RW: existence check if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 )
3237 if ( *(char *) (PseudoLinkedPointer+4+4+4+wordlen) != 0 ) // If LEAF is full: Case #2
3238 { Splitoccured = 1; WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
3239 // Allocate NEW LEAF:
3240 if ( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wordlen + 4 + 4 + 4 < GRMBLFOOLAgain[(int)wordlen] ) // +4 more for BST
instead of LL; + more(see LEAF)
3241 {
3242 memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
3243 bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
3244 bufend[LetterOffset] = bufend[LetterOffset] + 2*wordlen;
3245 if (MAXusedBuffer[wordlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) MAXusedBuffer[wordlen] = (unsigned
long)(bufend[LetterOffset] - BufStart);
3246 }
3247 else
3248 { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)\n" );
3249 printf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
3250 printf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEM, l1ToADigits, 10) );
3251 printf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORDcount, l1ToADigits, 10), _ui64toaKAZEcomma((unsigned long)
WORDcountDistinct, l1ToADigits2, 10) );
3252 printf( fp_outLOG, "Number of Files: %lu\n", NumberOfFiles );
3253 printf( fp_outLOG, "Number of Lines: %lu\n", NumberOfLines );
3254 printf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>20)+1 );
3255 printf( fp_outLOG, "Total Attempts to Find/Put WORDS into b-trees order 3: %s\n", _ui64toaKAZEcomma(WORDcountAttemptsToPut, l1ToADigits, 10)
);
3256 for( k = 1; k < 32; k++)
3257 { printf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s% utilization\n", _ui64toaKAZEzerocoma(k, l1ToADigits, 10)+(26-
2), _ui64toaKAZEzerocoma((MAXusedBuffer[k]>10)+1, l1ToADigits2, 10)+(26-5), _ui64toaKAZEzerocoma(((GRMBLhil1[(int)k] *
LetterBuffer[31]>10)+1, l1ToADigits3, 10)+(26-5), _ui64toaKAZEzerocoma((unsigned long long)(MAXusedBuffer[k]*100)/(GRMBLhil1[(int)k] *
LetterBuffer[31]), l1ToADigits4, 10)+(26-2), "%0" ); // 26 are all 26-DESIRE=24
3258 }
3259 printf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)thunderwith );
3260 printf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)\n\n" );
3261 return( 1 );
3262 }
3263 if (PoffsetInLEAF == 0) // wrd < LW
3264 {
3265 memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
3266 memcpy( PseudoLinkedPointer+4+4+4, wrd, wrdlen ); // wrd go to OLD LEAF
3267 memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wordlen, wrdlen ); // RW go to NEW LEAF
3268 *(char *) (PseudoLinkedPointer+4+4+4+wordlen) = 0; // RW mark unused in OLD LEAF
3269 }
3270 if (PoffsetInLEAF == 4) // LW < wrd < RW
3271 {
3272 memcpy( wrdUP, wrd, wrdlen ); // wrd up
3273 memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wordlen, wrdlen ); // RW go to NEW LEAF
3274 *(char *) (PseudoLinkedPointer+4+4+4+wordlen) = 0; // RW mark unused in OLD LEAF
3275 }
3276 if (PoffsetInLEAF == 8) // wrd > RW
3277 {
3278 memcpy( wrdUP, PseudoLinkedPointer+4+4+4+wordlen, wrdlen ); // RW up
3279 *(char *) (PseudoLinkedPointer+4+4+4+wordlen) = 0; // RW mark unused in OLD LEAF
3280 memcpy( PseudoLinkedPointerNEW+4+4+4, wrd, wrdlen ); // wrd go to NEW LEAF
3281 }
3282 }
3283 else // If LEAF is not full: Case #1
3284 { Splitoccured = 0; WORDcountDistinct++; bufNumberOfWords[LetterOffset]++;
3285 if (PoffsetInLEAF == 0) // wrd < [LW][ ] so [LW][ ] -> [ ][LW] -> [wrd][LW]
3286 {
3287 memcpy( PseudoLinkedPointer+4+4+4+wordlen, PseudoLinkedPointer+4+4+4, wrdlen );

```

```

3288         memcpy( PseudoLinkedPointer+4+4+4, wrd, wrdlen );
3289     }
3290     if (PoffsetInLEAF == 4) // wrd > [LW][ ] so [LW][ ] -> [LW][wrd]
3291     {
3292         memcpy( PseudoLinkedPointer+4+4+4+wrdlen, wrd, wrdlen );
3293     }
3294 }
3295 }
3296 if (SplitOccured != 0)
3297 {
3298     // ~ Second deal with the INNER NODE(S) i.e Case #3 & Case #4:
3299     while (StackPtr != 0 || SplitOccured != 0)
3300     {
3301         // 'PseudoLinkedPointerNEW' is new LEAF to be inserted
3302         // 'wrdUP' is NEW word to be inserted
3303         if (StackPtr != 0)
3304         {
3305             PoffsetInLEAF = BSTstack[--StackPtr];
3306             PseudoLinkedPointer = BSTstack[--StackPtr];
3307             if ( *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) != 0 ) // If LEAF is full: Case #4
3308             { SplitOccured = 1;
3309                 memcpy( wrdUPold, wrdUP, wrdlen ); // LW up
3310                 PseudoLinkedPointerNewOld = PseudoLinkedPointerNEW;
3311                 // ALlocate NEW LEAF:
3312                 if ( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST
3313                 instead of LL; + more(see LEAF)
3314                 {
3315                     memcpy( &PseudoLinkedPointerNEW, &bufend[LetterOffset], 4 );
3316                     bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
3317                     bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
3318                     if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
3319                     long)(bufend[LetterOffset] - BufStart);}
3320                 }
3321             }
3322             else
3323             { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n" );
3324               fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
3325               fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, l1toadigits, 10) );
3326               fprintf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORdcount, l1toadigits, 10), _ui64toaKAZEcomma((unsigned long)
3327               long)WORdcountDistinct, l1toadigits, 10) );
3328               fprintf( fp_outLOG, "Number of Files: %lu\n", NumberOfFiles );
3329               fprintf( fp_outLOG, "Number of Lines: %lu\n", NumberOfLines );
3330               fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>20)+1 );
3331               fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORdcountAttemptsToPut, l1toadigits, 10)
3332               );
3333             }
3334             for( k = 1; k < 32; k++)
3335             { fprintf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s% utilization\n", _ui64toaKAZEzerocomma(k, l1toadigits, 10)+(26-
3336             2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, l1toadigits, 10)+(26-5), _ui64toaKAZEzerocomma(((GRMBLhlll[(int)k] *
3337             LetterBuffer)/31)>>10)+1, l1toadigits, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/(GRMBLhlll[(int)k] *
3338             LetterBuffer)/31), l1toadigits, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
3339         }
3340         fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)thunderwith );
3341         fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n" );
3342         return( 1 );
3343     }
3344     if (PoffsetInLEAF == 0) // wrdUPold < LW
3345     {
3346         memcpy( wrdUP, PseudoLinkedPointer+4+4+4, wrdlen ); // LW up
3347         memcpy( PseudoLinkedPointer+4+4+4, wrdUPold, wrdlen ); // wrdUPold go to OLD LEAF
3348         memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
3349         *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
3350         // [LP][PseudoLinkedPointerNewOld][MP][RP][wrdUPold][LW][RW] -----
3351         // [LP] pair [LW] PseudoLinkedPointerNEW goes up -----
3352         // PseudoLinkedPointer: PseudoLinkedPointerNEW: |
3353         // [LP][PseudoLinkedPointerNewOld][ ] [wrdUPold] [MP][RP][ ] [RW] <-----
3354         // no need to put zero in RP because logic is based on words existence:
3355         memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+4, 4 );
3356         memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
3357         memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNewOld, 4 );
3358     }
3359     if (PoffsetInLEAF == 4) // LW < wrdUPold < RW
3360     {
3361         memcpy( wrdUP, wrdUPold, wrdlen ); // wrdUPold up
3362         memcpy( PseudoLinkedPointerNEW+4+4+4, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW go to NEW LEAF
3363         *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
3364         // [LP][MP][PseudoLinkedPointerNewOld][RP][LW][wrdUPold][RW] -----
3365         // pair [wrdUPold] PseudoLinkedPointerNEW goes up |
3366         // PseudoLinkedPointer: PseudoLinkedPointerNEW: |
3367         // [LP][MP][ ] [LW] (PseudoLinkedPointerNewOld)[RP][ ] [RW] <-----
3368         // no need to put zero in RP because logic is based on words existence:
3369         memcpy( PseudoLinkedPointerNEW+0, &PseudoLinkedPointerNewOld, 4 );
3370         memcpy( PseudoLinkedPointerNEW+4, PseudoLinkedPointer+8, 4 );
3371     }
3372     if (PoffsetInLEAF == 8) // wrdUPold > RW
3373     {
3374         memcpy( wrdUP, PseudoLinkedPointer+4+4+4+wrdlen, wrdlen ); // RW up
3375         *(char *) (PseudoLinkedPointer+4+4+4+wrdlen) = 0; // RW mark unused in OLD LEAF
3376         memcpy( PseudoLinkedPointerNEW+4+4+4, wrdUPold, wrdlen ); // wrdUPold go to NEW LEAF
3377         // [LP][MP][RP][PseudoLinkedPointerNewOld][LW][RW][wrdUPold] -----

```

```

3369         // pair [RW] PseudoLinkedPointerNEW goes up |
3370         // PseudoLinkedPointer: PseudoLinkedPointerNEW: |
3371         // [LP][MP][ ] [LW] (PseudoLinkedPointerNewOld)[ ] [wrdUPold] <---
3372         // no need to put zero in RP because logic is based on words existence:
3373         memcpy( PseudoLinkedPointerNEW+0, PseudoLinkedPointer+8, 4 );
3374         memcpy( PseudoLinkedPointerNEW+4, &PseudoLinkedPointerNewOld, 4 );
3375     }
3376 }
3377 else // If LEAF is not full: Case #3
3378 { SplitOccured = 0;
3379     if (PoffsetInLEAF == 0) // wrdUP < [LW][ ] so [LW][ ] -> [ ] [LW] -> [wrdUP][LW]
3380     {
3381         memcpy( PseudoLinkedPointer+4+4+4+wrdlen, PseudoLinkedPointer+4+4+4, wrdlen );
3382         memcpy( PseudoLinkedPointer+4+4+4, wrdUP, wrdlen );
3383         // [LP][MP][ ] -> [LP][ ] [MP] -> [LP][np][MP]
3384         memcpy( PseudoLinkedPointer+8, PseudoLinkedPointer+4, 4 );
3385         memcpy( PseudoLinkedPointer+4, &PseudoLinkedPointerNEW, 4 );
3386     }
3387     if (PoffsetInLEAF == 4) // wrdUP > [LW][ ] so [LW][ ] -> [LW][wrdUP]
3388     {
3389         memcpy( PseudoLinkedPointer+4+4+4+wrdlen, wrdUP, wrdlen );
3390         // [LP][MP][ ] -> [LP][MP][np]
3391         memcpy( PseudoLinkedPointer+8, &PseudoLinkedPointerNEW, 4 );
3392     }
3393     }
3394     }
3395     }
3396     else // Empty stack means ROOT and more over ROOT is already splitted(Case #4 is off)
3397     {
3398         // If LEAF is not full: Case #3
3399         // THIS IS WHERE A NEW(SECOND) LEAF 'PseudoLinkedPointerROOT' must be allocated:
3400         if ( (unsigned long)(bufend[LetterOffset] - BufStart) + 2*wrdlen + 4 + 4 + 4 < GRMBLFoolAgain[(int)wrdlen] ) // +4 more for BST
3401         instead of LL; + more(see LEAF)
3402         {
3403             memcpy( &PseudoLinkedPointerROOT, &bufend[LetterOffset], 4 );
3404             bufend[LetterOffset] = bufend[LetterOffset] + 4 + 4 + 4; // + 4 due to above commenting
3405             memcpy( bufend[LetterOffset], wrdUP, wrdlen );
3406             bufend[LetterOffset] = bufend[LetterOffset] + 2*wrdlen;
3407             if (MAXusedBuffer[wrdlen] < (unsigned long)(bufend[LetterOffset] - BufStart)) {MAXusedBuffer[wrdlen] = (unsigned
3408             long)(bufend[LetterOffset] - BufStart);}
3409         }
3410         else
3411         { printf( "\nLeprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n" );
3412           fprintf( fp_outLOG, "Input File with a list of TEXTual Files: %s\n", argv[1] );
3413           fprintf( fp_outLOG, "Size of all TEXTual Files: %s\n", _ui64toaKAZEcomma(FilesLEN, l1toadigits, 10) );
3414           fprintf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZEcomma(WORdcount, l1toadigits, 10), _ui64toaKAZEcomma((unsigned long)
3415           long)WORdcountDistinct, l1toadigits, 10) );
3416           fprintf( fp_outLOG, "Number of Files: %lu\n", NumberOfFiles );
3417           fprintf( fp_outLOG, "Number of Lines: %lu\n", NumberOfLines );
3418           fprintf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>20)+1 );
3419           fprintf( fp_outLOG, "Total Attempts to Find/Put WORDs into B-trees order 3: %s\n", _ui64toaKAZEcomma(WORdcountAttemptsToPut, l1toadigits, 10)
3420           );
3421         }
3422         for( k = 1; k < 32; k++)
3423         { fprintf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s% utilization\n", _ui64toaKAZEzerocomma(k, l1toadigits, 10)+(26-
3424         2), _ui64toaKAZEzerocomma((MAXusedBuffer[k]>>10)+1, l1toadigits, 10)+(26-5), _ui64toaKAZEzerocomma(((GRMBLhlll[(int)k] *
3425         LetterBuffer)/31)>>10)+1, l1toadigits, 10)+(26-5), _ui64toaKAZEzerocomma((unsigned long long)(MAXusedBuffer[k]*100)/(GRMBLhlll[(int)k] *
3426         LetterBuffer)/31), l1toadigits, 10)+(26-2), "%0" ); // 26 are all 26-DESIRED=24
3427     }
3428     fprintf( fp_outLOG, "Used value for third parameter in KB: %lu\n", (unsigned long)thunderwith );
3429     fprintf( fp_outLOG, "Leprechaun: Failure! Increment 'Memory for each Letter' parameter(third one)!\n\n" );
3430     return( 1 );
3431 }
3432 }
3433 }
3434 }
3435 }
3436 }
3437 }
3438 }
3439 }
3440 }
3441 }
3442 }
3443 }
3444 }
3445 }
3446 }
3447 }
3448 }
3449 }

```

```

3450         { wrd[ wrdlen ] = workbyte + 32 ; }
3451         wrdlen++;
3452     }
3453     else if( workbyte >= 'a' && workbyte <= 'z' )
3454     {
3455         if( wrdlen < 31 )
3456         { wrd[ wrdlen ] = workbyte; }
3457         wrdlen++;
3458     }
3459     else
3460     {
3461         // This fragment is MIRRORed: #2 copy [
3462         goto ElStupido;
3463         // This fragment is MIRRORed: #2 copy ]
3464     }
3465     } // i 'for'
3466     //-----
3467     ///+Melnitckka;
3468     //Melnitckka = Melnitckka % 4;
3469     //if (Melnitckka == 0){ printf( "\n"; word count: %s of them %s distinct; Done: %1u/64\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10),
3470     //if (Melnitckka == 1){ printf( "\n"; word count: %s of them %s distinct; Done: %1u/64\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10),
3471     //if (Melnitckka == 2){ printf( "\n"; word count: %s of them %s distinct; Done: %1u/64\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10),
3472     //if (Melnitckka == 3){ printf( "\n"; word count: %s of them %s distinct; Done: %1u/64\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10),
3473     //if (Melnitckka == 4){ printf( "\n"; word count: %s of them %s distinct; Done: %1u/64\n", _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10),
3474     printf( "%s; word count: %s of them %s distinct; Done: %1u/64\n", Auberge[Melnitckka++], _ui64toaKAZEcomma(WORDcount, 11ToaDigits, 10),
3475     _ui64toaKAZEcomma( (unsigned long long)WORDcountDistinct, 11ToaDigits, 10), 64 );
3476     LINE10len = 0;
3477     LINE10[ LINE10len ] = 0;
3478     fclose( fp_inLINE );
3479     }
3480     } // k 'for'
3481     } // k 'for'
3482
3483     (void) time(&t3);
3484     if (t3 <= t1) {t3 = t1; t3++;}
3485     printf( "bytes per second performance: %sB/s\n", _ui64toaKAZEcomma(FilesLEN/((int) t3-t1), 11ToaDigits, 10) ); // Rev. 12+
3486     printf( "words per second performance: %sW/s\n", _ui64toaKAZEcomma(WORDcount/((int) t3-t1), 11ToaDigits, 10) ); // Rev. 12+
3487
3488     // FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH FLUSH
3489     printf("Flushing unsorted words... \n");
3490     if( ( fp_out = fopen( argv[2], "wb+" ) ) == NULL )
3491     { printf( "Leprechaun: Can't create file %s \n", argv[2] ); return( 1 ); }
3492     ZEROS[0] = 0; ZEROS[1] = 0; ZEROS[2] = 0; ZEROS[3] = 0;
3493     CRDLFa[0] = 13; CRDLFa[1] = 10;
3494
3495     for( i = 0; i < 806; i++ )
3496     { //BufStart = pointerFlush + i * LetterBuffer; // OLD
3497       BufStart = pointerFlush + ( i / 31 ) * WHOLELetter_Buffersize + OffsetsInBuffer[i % 31];
3498       for( j = 0; j < NumberOfSLOTS; j++ )
3499       {
3500         Slot = j<<2;
3501         memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
3502         while (PseudoLinkedPointer != 0)
3503         { memcpy( &PseudoLinkedPointerNEW, PseudoLinkedPointer, 4 );
3504           memcpy( PseudoLinkedPointer, ZEROS, 4 );
3505           PseudoLinkedPointer = PseudoLinkedPointerNEW;
3506         }
3507       }
3508       // Start of COUPLES [OFFSET: 4byte(ZEROS)] [WORD: up to 31bytes]
3509       //fwrite(BufStart+(NumberOfSLOTS+1)*4, bufend[i] - (BufStart+(NumberOfSLOTS+1)*4), 1, fp_out );
3510       // * Follows STATE OF UGLINESS: */
3511       Flushing = BufStart+(NumberOfSLOTS+1)*4 + 4; // '4' in order to skip first 4 zeros
3512       //in case of current buffer not have been used then NOT entering in this cycle
3513       while(Flushing < bufend[i])
3514       { if (*Flushing != 0) {fwrite(Flushing, 1, 1, fp_out); TotalWlchars++;}
3515         // below 'Flushing-1' works due to skipped first 4 zeros!
3516         if (*(Flushing-1) != 0 && *Flushing == 0) {fwrite(CRDLFa, 2, 1, fp_out);}
3517         //last word must be suffixed with 1310 too
3518         if (Flushing == bufend[i]-1) {fwrite(CRDLFa, 2, 1, fp_out);}
3519         Flushing++;
3520       }
3521     }
3522     for( j = 0; j < NumberOfSLOTS; j++ )
3523     {
3524       Slot = j<<2;
3525       memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
3526       if (PseudoLinkedPointer != 0)
3527       {
3528         NumberOfTrees++;
3529         if (BSTorBtree != 1)
3530         {
3531           // ===== BST traverse [
3532           // DONE JOB:

```

```

3533     // Must be written BST traverse ! with simulated stack i.e. non-recursive.
3534     // ...
3535     // /*
3536     // Given a binary search tree, print out
3537     // its data elements in increasing
3538     // sorted order.
3539     // */
3540     // void printTree(struct node* node) {
3541     // if (node == NULL) return;
3542     // printTree(node->left);
3543     // printf("%d ", node->data);
3544     // printTree(node->right);
3545     // }
3546
3547     // FUTURE JOB:
3548     // I need functions:
3549     // BST_LeafNumber() // greater the better
3550     // BST_NodeNumber() // 'BSTcurrent' below
3551     // BST_Peak() // i.e. Levels, root has height = 1
3552     // BST_PeakIB() // IBBST(Ideal Balanced BST) has 1 + lgNodeNumber height
3553     // I need 'Ideal Balancing BST FRAGMENT' with simulated stack:
3554     // I need 'Ideal Balancing BST FRAGMENT' to be executed when Peak() >= PeakIB()<<1:
3555
3556     // ----- [
3557     BSTcurrentNode = 0; BSTcurrentPeak = 0; BSTcurrentLeaf = 0;
3558     BSTcurrentPeakMAX = 0; // Height of current BST
3559     StackPtr = 0;
3560     while ( 2==2 ) {
3561       while (PseudoLinkedPointer != 0)
3562       {
3563         if (StackPtr > 65536*3-1-3) { printf( "\nLeprechaun: Failure! BST simulated stack overflow, too high BST!\n" ); return( 13 );}
3564         memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
3565         PseudoLinkedPointer = PseudoLinkedPointer + 4;
3566         memcpy( &PseudoLinkedPointerNEWright, PseudoLinkedPointer, 4 );
3567         BSTstack[StackPtr] = PseudoLinkedPointer + 4; ++StackPtr; //ptr to wrd
3568         BSTstack[StackPtr] = PseudoLinkedPointerNEWright; ++StackPtr;
3569         BSTstack[StackPtr] = PseudoLinkedPointerNEWleft; ++StackPtr; //needed for stats not for recursion
3570       }
3571       // BST stats [
3572       if (PseudoLinkedPointerNEWleft == 0 && PseudoLinkedPointerNEWright == 0) {BSTcurrentLeaf++; BSTTotalLEAFs++;}
3573       BSTcurrentPeak++;
3574       if (BSTcurrentPeakMAX < BSTcurrentPeak) BSTcurrentPeakMAX = BSTcurrentPeak;
3575       BSTstack[StackPtr] = BSTcurrentPeak; ++StackPtr; //needed for stats not for recursion
3576       // BST stats ]
3577       PseudoLinkedPointer = PseudoLinkedPointerNEWright; // choose right instead of 'PseudoLinkedPointerNEWleft' because of stats
3578     }
3579     if (StackPtr == 0) break;
3580     BSTcurrentPeak = BSTstack[--StackPtr]; // level of the node(1 is root) needed only for stats(print)
3581     PseudoLinkedPointerNEWleft = BSTstack[--StackPtr]; // left pointer needed only for stats(print)
3582     PseudoLinkedPointerNEWright = BSTstack[--StackPtr]; // right pointer
3583     memcpy( wrd, BSTstack[--StackPtr], i%31+1 );
3584     fwrite(wrd, i%31+1, 1, fp_out); TotalWlchars = TotalWlchars + i%31+1;
3585     fwrite(CRDLFa, 2, 1, fp_out);
3586     BSTcurrentNode++;
3587     PseudoLinkedPointer = PseudoLinkedPointerNEWleft;
3588     // ----- ]
3589     // BST stats [
3590     if (BSTwithMAXnode < BSTcurrentNode) {
3591       BSTwithMAXnode = BSTcurrentNode;
3592       BSTwithMAXnodePEAK = BSTcurrentPeakMAX;
3593       BSTwithMAXnodeLEAF = BSTcurrentLeaf;
3594       BSTcurrentNodeMAXQUANTITY = 0;
3595     }
3596     if (BSTwithMAXnode == BSTcurrentNode) BSTcurrentNodeMAXQUANTITY++;
3597     if (BSTwithMAXpeak < BSTcurrentPeakMAX) {
3598       BSTwithMAXpeak = BSTcurrentPeakMAX;
3599       BSTwithMAXpeakNODE = BSTcurrentNode;
3600       BSTwithMAXpeakLEAF = BSTcurrentLeaf;
3601       BSTcurrentPeakMAXQUANTITY = 0;
3602       iBSTwithMAXpeak=i; jBSTwithMAXpeak=j;
3603     }
3604     if (BSTwithMAXpeak == BSTcurrentPeakMAX) BSTcurrentPeakMAXQUANTITY++;
3605     if (BSTwithMAXleaf < BSTcurrentLeaf) {
3606       BSTwithMAXleaf = BSTcurrentLeaf;
3607       BSTwithMAXleafNODE = BSTcurrentNode;
3608       BSTwithMAXleafPEAK = BSTcurrentPeakMAX;
3609       BSTcurrentLeafMAXQUANTITY = 0;
3610     }
3611     if (BSTwithMAXleaf == BSTcurrentLeaf) BSTcurrentLeafMAXQUANTITY++;
3612     // BST stats ]
3613     // ===== BST traverse ]
3614     } else
3615     {
3616       // ===== B-tree order 3 traverse [
3617       // DONE JOB:
3618       // Must be written B-tree traverse ! with simulated stack i.e. non-recursive.
3619       // ...
3620       StackPtr = 0;

```

```

3620 while ( 2==2 ) {
3621     while (PseudoLinkedPointer != 0)
3622     {
3623         if (StackPtr > 65536*3-1) { printf( "\nLeprechaun: Failure! B-tree simulated stack overflow, too high B-tree!\n" ); return( 13 );}
3624     };
3625     BSTstack[StackPtr] = PseudoLinkedPointer; ++StackPtr; //ptr to Rwrtd
3626     if ( *(char *) (PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) == 0 ) {memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTS*4], 4 );}
3627     memcpy( &PseudoLinkedPointerNEWleft, PseudoLinkedPointer, 4 );
3628     memcpy( &PseudoLinkedPointerNEwmiddle, PseudoLinkedPointer + 4, 4 );
3629     memcpy( &PseudoLinkedPointerNEwright, PseudoLinkedPointer + 4 + 4, 4 );
3630 // Give first from right to left non-zero PTR
3631 if (PseudoLinkedPointerNEwright != 0)
3632 { memcpy( PseudoLinkedPointer + 4 + 4, &BufStart[NumberOfSLOTS*4], 4 );
3633   PseudoLinkedPointer = PseudoLinkedPointerNEwright;
3634 }
3635 else if (PseudoLinkedPointerNEwmiddle != 0)
3636 { memcpy( PseudoLinkedPointer + 4, &BufStart[NumberOfSLOTS*4], 4 );
3637   PseudoLinkedPointer = PseudoLinkedPointerNEwmiddle;
3638 }
3639 else if (PseudoLinkedPointerNEwleft != 0)
3640 { memcpy( PseudoLinkedPointer, &BufStart[NumberOfSLOTS*4], 4 );
3641   PseudoLinkedPointer = PseudoLinkedPointerNEwleft;
3642 }
3643 else
3644 { PseudoLinkedPointer = 0;
3645 }
3646 if (StackPtr == 0) break;
3647 PseudoLinkedPointer = BSTstack[--StackPtr];
3648 memcpy( &PseudoLinkedPointerNEwleft, PseudoLinkedPointer, 4 );
3649 memcpy( &PseudoLinkedPointerNEwmiddle, PseudoLinkedPointer + 4, 4 );
3650 memcpy( &PseudoLinkedPointerNEwright, PseudoLinkedPointer + 4 + 4, 4 );
3651 if (PseudoLinkedPointerNEwleft+PseudoLinkedPointerNEwmiddle+PseudoLinkedPointerNEwright == 0) // One LEAF is PRINTED when LP=0 MP=0
3652 RP=0
3653 {
3654     memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4, i%31+1 );
3655     fwrite( wrd, i%31+1, 1, fp_out ); TotalWLchars = TotalWLchars + i%31+1;
3656     fwrite( CRDLfa, 2, 1, fp_out );
3657     if ( *(char *) (PseudoLinkedPointer + 4 + 4 + 4 + i%31+1) != 0 )
3658     { memcpy( wrd, PseudoLinkedPointer + 4 + 4 + 4 + i%31+1, i%31+1 );
3659       fwrite( wrd, i%31+1, 1, fp_out ); TotalWLchars = TotalWLchars + i%31+1;
3660       fwrite( CRDLfa, 2, 1, fp_out );
3661     }
3662     PseudoLinkedPointer = 0;
3663 }
3664 // $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ B-tree order 3 traverse ]
3665 } // j
3666 // i
3667 } // i
3668 if (BSTorBtree != 1)
3669 {
3670     // ~~~~~ Longest path ~~~~~ [
3671     i=BSTwithMAXpeak; j=BSTwithMAXpeak;
3672     BufStart = pointerflush + (i / 31) * WHOLEletter_Buffersize + OffsetsInBuffer[i % 31];
3673     Slot = j<2;
3674     memcpy( &PseudoLinkedPointer, BufStart+Slot, 4 );
3675     if (PseudoLinkedPointer != 0)
3676     {
3677         // ----- ]
3678         BSTcurrentNode = 0; BSTcurrentPeak = 0; BSTcurrentLeaf = 0;
3679         BSTcurrentPeakMAX = 0; // Height of current BST
3680     }
3681     stackPtr = 0;
3682     // BST print [
3683     printf( fp_outLOG, "A(not always THE) Binary-Search-Tree with the longest path(height, PEAK, number of levels):\n" );
3684     // BST print ]
3685     while ( 2==2 ) {
3686     while (PseudoLinkedPointer != 0)
3687     {
3688     if (StackPtr > 65536*3-1-3) { printf( "\nLeprechaun: Failure! BST simulated stack overflow, too high BST!\n" ); return( 13 );}
3689     memcpy( &PseudoLinkedPointerNEwleft, PseudoLinkedPointer, 4 );
3690     PseudoLinkedPointer = PseudoLinkedPointer + 4;
3691     memcpy( &PseudoLinkedPointerNEwright, PseudoLinkedPointer, 4 );
3692     BSTstack[StackPtr] = PseudoLinkedPointer + 4; ++StackPtr; //ptr to wrd
3693     BSTstack[StackPtr] = PseudoLinkedPointerNEwright; ++StackPtr;
3694     BSTstack[StackPtr] = PseudoLinkedPointerNEwleft; ++StackPtr; //needed for stats not for recursion
3695     // BST stats [
3696     if (PseudoLinkedPointerNEwleft == 0 && PseudoLinkedPointerNEwright == 0) {BSTcurrentLeaf++;} //BSTsTotalLEAFs++; // REMOVED
3697     to avoid mess in TOTAL stats
3698     BSTcurrentPeak++;
3699     if (BSTcurrentPeakMAX < BSTcurrentPeak) BSTcurrentPeakMAX = BSTcurrentPeak;
3700     BSTstack[StackPtr] = BSTcurrentPeak; ++StackPtr; //needed for stats not for recursion
3701     // BST stats ]
3702     PseudoLinkedPointer = PseudoLinkedPointerNEwright; // choose right instead of 'PseudoLinkedPointerNEwleft' because of stats
}
}

```

```

print
3705 }
3706 if (StackPtr == 0) break;
3707 BSTcurrentPeak = BSTstack[--StackPtr]; // level of the node(l is root) needed only for stats(print)
3708 PseudoLinkedPointerNEwleft = BSTstack[--StackPtr]; // left pointer needed only for stats(print)
3709 PseudoLinkedPointerNEwright = BSTstack[--StackPtr]; // right pointer
3710 memcpy( wrd, BSTstack[--StackPtr], i%31+1);
3711 //fwrite(wrd, i%31+1, 1, fp_out); TotalWLchars = TotalWLchars + i%31+1;
3712 //fwrite(CRDLfa, 2, 1, fp_out);
3713 BSTcurrentNode++;
3714 PseudoLinkedPointer = PseudoLinkedPointerNEwleft;
3715 // BST print ]
3716 for ( k = 0; k < BSTcurrentPeak; k++ ) printf( fp_outLOG, "%c", ' ' );
3717 if (PseudoLinkedPointerNEwleft == 0) printf( fp_outLOG, "[l" ); else printf( fp_outLOG, "l" );
3718 for ( k = 0; k < i%31+1; k++ ) printf( fp_outLOG, "%c", *(char *) (wrd+k) );
3719 if (PseudoLinkedPointerNEwright == 0) printf( fp_outLOG, "l" ); else printf( fp_outLOG, "[r" );
3720 if (BSTcurrentPeak == 1) printf( fp_outLOG, " ROOT" );
3721 printf( fp_outLOG, "\n" );
3722 // BST print ]
3723 }
3724 // ----- ]
3725 }
3726 printf( fp_outLOG, "Above Binary-Search-Tree with MaxPEAK = %s has NODES = %s and LEAFS = %s\n", _ui64toaKAZeomma(BSTwithMAXpeak,
11ToaDigits2, 10), _ui64toaKAZeomma(BSTwithMAXpeakNODE, 11ToaDigits3, 10), _ui64toaKAZeomma(BSTwithMAXpeakLEAF, 11ToaDigits, 10));
3727 printf( fp_outLOG, "Legend:\n" );
3728 printf( fp_outLOG, "At left side of the word - '[' means no left successor'\n" );
3729 printf( fp_outLOG, "At left side of the word - '[' means left successor exists'\n" );
3730 printf( fp_outLOG, "At right side of the word - '[' means no right successor'\n" );
3731 printf( fp_outLOG, "At right side of the word - '[' means right successor exists'\n" );
3732 // ~~~~~ Longest path ~~~~~ ]
3733 }
3734 // BST stats [
3735 PEAKibBST=1-ffloorLog2(BSTwithMAXnode);
3736 //PEAKibBST=1;
3737 //while (BSTwithMAXnode>PEAKibBST) PEAKibBST++;
3738 // BST stats ]
3739 }
3740 (void) time(&t2);
3741 if (t2 <= t1) {t2 = t1; t2++;}
3742 printf( "Time for making unsorted wordlist: %d second(s)\n", (int) t2-t1);
3743 printf( fp_outLOG, "bytes per second performance: %sB/s\n", _ui64toaKAZeomma(FilesLEN/((int) t3-t1), 11ToaDigits, 10); // Rev. 12+
3744 printf( fp_outLOG, "words per second performance: %sW/s\n", _ui64toaKAZeomma(WORdcount/((int) t3-t1), 11ToaDigits, 10); // Rev. 12+
3745 printf( fp_outLOG, "Input File with a list of TEXTUAL Files: %s\n", argv[1] );
3746 printf( fp_outLOG, "Size of all TEXTUAL Files: %s\n", _ui64toaKAZeomma(FilesLEN, 11ToaDigits, 10);
3747 printf( fp_outLOG, "word count: %s of them %s distinct\n", _ui64toaKAZeomma(WORdcount, 11ToaDigits, 10), _ui64toaKAZeomma((unsigned long)WORdcountDistinct, 11ToaDigits2, 10);
3748 printf( fp_outLOG, "Number of Files: %lu\n", NumberOfFiles );
3749 printf( fp_outLOG, "Number of Lines: %lu\n", NumberOfLines );
3750 printf( fp_outLOG, "Allocated memory in MB: %lu\n", (unsigned long)(memory_size>>20)+1 );
3751 NumberOfHashCollisions = WORdcountDistinct - NumberOfTrees;
3752 printf( fp_outLOG, "Number of Trees(GREATER THE BETTER): %lu\n", NumberOfTrees );
3753 printf( fp_outLOG, "Forest population(Hash Function Quality regarding collisions i.e. Hash Table Utilization): %lu%\n",
(NumberOfTrees*100)/(26*31*8192), "%0" );
3754 printf( fp_outLOG, "Number of Hash Collisions(Distinct WORDS - Number of Trees): %lu\n", NumberOfHashCollisions );
3755 if (BSTorBtree != 1)
3756 {
3757     printf( fp_outLOG, "Maximum Attempts to Find/Put a WORD into a Binary-Search-Tree: %s\n", _ui64toaKAZeomma(BSTwithMAXpeak, 11ToaDigits, 10);
3758     printf( fp_outLOG, "Total Attempts to Find/Put WORDS into Binary-Search-Trees: %s\n", _ui64toaKAZeomma(WORdcountAttemptsToPut, 11ToaDigits, 10);
3759     printf( fp_outLOG, "Total Number of LEAFS in Binary-Search-Trees(GREATER THE BETTER): %s\n", _ui64toaKAZeomma(BSTsTotalLEAFs, 11ToaDigits, 10);
3760     printf( fp_outLOG, "Perfectly-Balanced-Binary-Search-Tree for MAXNODES = %s must have PEAK = %s = rounding down of integer (1+lb(3s))\n",
_ui64toaKAZeomma(BSTwithMAXnode, 11ToaDigits, 10), _ui64toaKAZeomma(PEAKibBST, 11ToaDigits2, 10), _ui64toaKAZeomma(BSTwithMAXnode, 11ToaDigits3, 10);
3761     printf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MAXNODES = %s has PEAK = %s and LEAFS = %s\n",
_ui64toaKAZeomma(BSTcurrentNodeMAXQUANTITY, 11ToaDigits4, 10), _ui64toaKAZeomma(BSTwithMAXnode, 11ToaDigits2, 10),
_ui64toaKAZeomma(BSTwithMAXnodePEAK, 11ToaDigits3, 10), _ui64toaKAZeomma(BSTwithMAXnodeLEAF, 11ToaDigits, 10);
3762     printf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MaxPEAK = %s' has NODES = %s and LEAFS = %s\n",
_ui64toaKAZeomma(BSTcurrentPeakMAXQUANTITY, 11ToaDigits4, 10), _ui64toaKAZeomma(BSTwithMAXpeak, 11ToaDigits2, 10),
_ui64toaKAZeomma(BSTwithMAXpeakNODE, 11ToaDigits3, 10), _ui64toaKAZeomma(BSTwithMAXpeakLEAF, 11ToaDigits, 10);
3763     printf( fp_outLOG, "Binary-Search-Tree(1st out of %s) with MAXLEAFS = %s has NODES = %s and PEAK = %s\n",
_ui64toaKAZeomma(BSTcurrentLeafMAXQUANTITY, 11ToaDigits4, 10), _ui64toaKAZeomma(BSTwithMAXleaf, 11ToaDigits2, 10),
_ui64toaKAZeomma(BSTwithMAXleafNODE, 11ToaDigits3, 10), _ui64toaKAZeomma(BSTwithMAXleafPEAK, 11ToaDigits, 10);
3764 } else
3765 {
3766     printf( fp_outLOG, "Total Attempts to Find/Put WORDS into B-trees order 3: %s\n", _ui64toaKAZeomma(WORdcountAttemptsToPut, 11ToaDigits, 10);
3767 }
3768 }
3769 }
3770 for( k = 1; k < 32; k++ )
3771 { printf( fp_outLOG, "words with length %s occupy %sKB of %sKB given i.e. %s% utilization\n", _ui64toaKAZezerocomma(k, 11ToaDigits, 10)+(26-
2), _ui64toaKAZezerocomma((MAXusedBuffer[k]>>10)+1, 11ToaDigits2, 10)+(26-5), _ui64toaKAZezerocomma(((GRMBLhi11[(int)k] *
letterbuffer)/31)>>10)+1, 11ToaDigits3, 10)+(26-5), _ui64toaKAZezerocomma((unsigned long long)(MAXusedBuffer[k]*100)/(GRMBLhi11[(int)k] *
letterbuffer)/31), 11ToaDigits4, 10)+(26-2), "%0" ); // 26 are all 26 DESIRED=24
3772 }
3773 if ( MAXusedBufferABS < 31 * ((MAXusedBuffer[k]>>10)+1) / GRMBLhi11[(int)k] ) {MAXusedBufferABS = 1+31 * ((MAXusedBuffer[k]>>10)+1) /
}

```



```

3939 clocks1 = clock();
3940 for (Bozan=0; Bozan < (1<<4); Bozan++) // 16 times, at end >>4
3941 {
3942     for( j = 0; j < nlines; j++)
3943         { //Slot = KuxHash3plus(backup[j]);
3944             //fprintf(fp_out, "Hashcode: %s - ", _ui64toaKAZezerocomma(Slot, 110aDigits, 10)+(26-5));
3945             wrdlen = strlen(backup[j]);
3946             if (wrdlen<=19) // 4x4+3=19 i.e. last contains 7 clashes
3947                 Slot = FNV1A_Hash_Granularity(backup[j], wrdlen>>2, 2); //13+++++
3948             else // 2x8+4=20 i.e. first contains 6 clashes
3949                 Slot = FNV1A_Hash_Granularity(backup[j], wrdlen>>3, 3); //13+++++
3950             } // Conclusion: two functions > 64 bytes lead to horrible slowness, so unite them in one: fit in the cache line.
3951 }
3952 clocks2 = clock();
3953 printf( "Performance of 'FNV1A_Hash_Granularity': %lu words/clock or %lu MB/s\n", (nlines/((long)(clocks2 - clocks1 + 1)>>4) ),
3954         ((TotalMChars>>10)/((long)(clocks2 - clocks1 + 1)>>4) ) );
3955 // 6]
3956 // Hash benchmarking ----- ]
3957 /*
3958
3959 if( ( fp_outLOG = fopen( "Leprechaun.LOG", "a+" ) ) == NULL )
3960     { printf( "Leprechaun: Can't open file Leprechaun.LOG.\n" ); return( 1 ); }
3961 fprintf( fp_outLOG, "Time for sorting unsorted wordlist: %d second(s)\n\n", (int) t3-t2);
3962 printf( "Leprechaun: Done.\n" );
3963     return 0;
3964 }
3965 else
3966     { printf("Leprechaun: Input file too large, wordlist remains unsorted!\n");
3967         return 1;
3968     }
3969 // SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT SORT
3970 }
3971 /*
3972 TO BE DONE: Ideal Balancing BST [
3973
3974 link rotR(link h)
3975 { link x = h->l; h->l = x->r; x->r = h;
3976     return x; }
3977
3978 link rotL(link h)
3979 { link x = h->r; h->r = x->l; x->l = h;
3980     return x; }
3981
3982 link parR(link h, int k)
3983 { int t = h->l->N;
3984     if (t > k)
3985         { h->l = parR(h->l, k); h = rotR(h); }
3986     if (t < k)
3987         { h->r = parR(h->r, k-t-1); h = rotL(h); }
3988     return h;
3989 }
3990
3991 link balancer(link h)
3992 {
3993     if (h->N < 2) return h;
3994     h = parR(h, h->N/2);
3995     h->l = balancer(h->l);
3996     h->r = balancer(h->r);
3997     return h;
3998 }
3999
4000
4001 TO BE DONE: Ideal Balancing BST ]
4002 /*
4003
4004 #include <stdlib.h>
4005 #include "Item.h"
4006 typedef struct STnode* link;
4007 struct STnode { Item item; link l, r; int N };
4008 static link head, z;
4009 link NEW(Item item, link l, link r, int N)
4010 { link x = malloc(sizeof *x);
4011     x->item = item; x->l = l; x->r = r; x->N = N;
4012     return x;
4013 }
4014
4015 void STinit()
4016 { head = (z = NEW(NULLitem, 0, 0, 0)); }
4017 int STcount() { return head->N; }
4018 Item searchR(link h, Key v)
4019 { Key t = key(h->item);
4020     if (h == z) return NULLitem;
4021     if (eq(v, t) return h->item;
4022     if (less(v, t) return searchR(h->l, v);
4023         else return searchR(h->r, v);
4024     }
4025 Item STsearch(Key v)

```

```

4026 { return searchR(head, v); }
4027 link insertR(link h, Item item)
4028 { Key v = key(item), t = key(h->item);
4029     if (h == z) return NEW(item, z, z, 1);
4030     if (less(v, t)
4031         h->l = insertR(h->l, item);
4032     else h->r = insertR(h->r, item);
4033     (h->N)++; return h;
4034 }
4035 void STinsert(Item item)
4036 { head = insertR(head, item); }
4037 /*
4038
4039 int count(link h)
4040 {
4041     if (h == NULL) return 0;
4042     return count(h->l) + count(h->r) + 1;
4043 }
4044
4045 int height(link h)
4046 { int u, v;
4047     if (h == NULL) return -1;
4048     u = height(h->l); v = height(h->r);
4049     if (u > v) return u+1; else return v+1;
4050 }
4051
4052 void printnode(char c, int h)
4053 { int i;
4054     for (i = 0; i < h; i++) printf(" ");
4055     printf("%c\n", c);
4056 }
4057
4058 void show(link x, int h)
4059 {
4060     if (x == NULL) { printnode("x", h); return; }
4061     show(x->r, h-1);
4062     printnode(x->item, h);
4063     show(x->l, h-1);
4064 }
4065
4066 */

```