Important: Before reading GB_WORDS, please read or at least skim the programs for GB_GRAPH and GB_IO.

1. Introduction. This GraphBase module provides two external subroutines:

words, a routine that creates a graph based on five-letter words;
find_word, a routine that looks for a given vertex in such a graph.

Examples of the use of these routines can be found in two demo programs, WORD_COMPONENTS and LADDERS.

⟨ `gb_words.h` 1 ⟩ ≡
```
extern Graph *words();
extern Vertex *find_word();
```
See also section 26.

2. The subroutine call *words*$(n, wt\_vector, wt\_threshold, seed)$ constructs a graph based on the five-letter words in `words.dat`. Each vertex of the graph corresponds to a single five-letter word. Two words are adjacent in the graph if they are the same except in one letter position. For example, '`words`' is adjacent to other words such as '`cords`', '`wards`', '`woods`', '`worms`', and '`wordy`'.

The constructed graph has at most n vertices; indeed, it has exactly n vertices if there are enough qualifying words. A word qualifies if its "weight" is *wt_threshold* or more, when weights are computed from a table pointed to by *wt_vector* according to rules described below. (If parameter *wt_vector* is Λ, i.e., `NULL`, default weights are used.) The fourth parameter, *seed*, is the seed of a random number generator.

All words of `words.dat` will be sorted by weight. The first vertex of the graph will be the word of largest weight, the second vertex will have second-largest weight, and so on. Words of equal weight will appear in pseudo-random order, as determined by the value of *seed* in a system-independent fashion. The first n words in order of decreasing weight are chosen to be vertices of the graph. However, if fewer than n words have weight $\geq$ *wt_threshold*, the graph will contain only the words that qualify. In such cases the graph will have fewer than n vertices—possibly none at all.

Exception: The special case $n = 0$ is equivalent to the case when n has been set to the highest possible value. It causes all qualifying words to appear.

3. Every word in `words.dat` has been classified as 'common' (`*`), 'advanced' (`+`), or 'unusual' (␣). Each word has also been assigned seven frequency counts c_1, ..., c_7, separated by commas; these counts show how often the word has occurred in different publication contexts:

c_1 times in the American Heritage Intermediate Corpus of elementary school material;
c_2 times in the Brown Corpus of reading material from America;
c_3 times in the Lancaster-Oslo/Bergen Corpus of reading material from Britain;
c_4 times in the Melbourne-Surrey Corpus of newspaper material from Australia;
c_5 times in the Revised Standard Version of the Bible;
c_6 times in *The TEXbook* and *The METAFONTbook* by D. E. Knuth;
c_7 times in *Concrete Mathematics* by Graham, Knuth, and Patashnik.

For example, one of the entries in `words.dat` is

```
happy*774,92,121,2,26,8,1
```

indicating a common word with $c_1 = 774$, ..., $c_7 = 1$.

Parameter *wt_vector* points to an array of nine integers $(a, b, w_1, \ldots, w_7)$. The weight of each word is computed from these nine numbers by using the formula

$$c_1 w_1 + \cdots + c_7 w_7 + \begin{cases} a, & \text{if the word is 'common';} \\ b, & \text{if the word is 'advanced';} \\ 0, & \text{if the word is 'unusual'.} \end{cases}$$

The components of *wt_vector* must be chosen so that

$$\max\bigl(|a|, |b|\bigr) + C_1|w_1| + \cdots + C_7|w_7| < 2^{30},$$

where C_j is the maximum value of c_j in the file; this restriction ensures that the *words* procedure will produce the same results on all computer systems.

4. The maximum frequency counts actually present are $C_1 = 15194$, $C_2 = 3560$, $C_3 = 4467$, $C_4 = 460$, $C_5 = 6976$, $C_6 = 756$, and $C_7 = 362$; these can be found in the entries for the common words '`shall`', '`there`', '`which`', and '`would`'.

The default weights are $a = 100$, $b = 10$, $c_1 = 4$, $c_2 = c_3 = 2$, $c_4 = c_5 = c_6 = c_7 = 1$.

File `words.dat` contains 5757 words, of which 3300 are 'common', 1194 are 'advanced', and 1263 are 'unusual'. Included among the unusual words are 891 having $c_1 = \cdots = c_7 = 0$; such words will always have weight zero, regardless of the weight vector parameter.

⟨ Private variables 4 ⟩ ≡

```
static long max_c[] = {15194, 3560, 4467, 460, 6976, 756, 362};      /* maximum counts C_j */
static long default_wt_vector[] = {100, 10, 4, 2, 2, 1, 1, 1, 1};    /* use this if wt_vector = Λ */
```

See also sections 17 and 25.

This code is used in section 7.

5. Examples: If you call *words*$(2000, \Lambda, 0, 0)$, you get a graph with 2000 of the most common five-letter words of English, using the default weights. The GraphBase programs are designed to be system-independent, so that identical graphs will be obtained by everybody who asks for *words*$(2000, \Lambda, 0, 0)$. Equivalent experiments on algorithms for graph manipulation can therefore be performed by researchers in different parts of the world.

The subroutine call *words*$(2000, \Lambda, 0, s)$ will produce slightly different graphs when the random seed s varies, because some words have equal weight. However, the graph for any particular value of s will be the same on all computers. The seed value can be any integer in the range $0 \le s < 2^{31}$.

Suppose you call *words*$(0, w, 1, 0)$, with w defined by the C declaration

$$\textbf{long } w[9] = \{1\};$$

this means that $a = 1$ and $b = w_1 = \cdots = w_7 = 0$. Therefore you'll get a graph containing only the 3300 'common' words. Similarly, it's possible to obtain only the $3300 + 1194 = 4494$ non-'unusual' words, by specifying the weight vector

$$\textbf{long } w[9] = \{1, 1\};$$

this makes $a = b = 1$ and $w_1 = \cdots = w_7 = 0$. In both of these examples, the qualifying words all have weight 1, so the vertices of the graph will appear in pseudo-random order.

If w points to an array of nine 0's, the call *words*$(n, w, 0, s)$ gives a random sample of n words, depending on s in a system-independent fashion.

If the entries of the weight vector are all nonnegative, and if the weight threshold is zero, every word of `words.dat` will qualify. Thus you will obtain a graph with $\min(n, 5757)$ vertices.

If w points to an array with *negative* weights, the call *words*$(n, w, -^{\#}\texttt{7fffffff}, 0)$ selects n of the *least* common words in `words.dat`.

6. If the *words* routine encounters a problem, it returns Λ, after putting a code number into the external variable *panic_code*. This code number identifies the type of failure. Otherwise *words* returns a pointer to the newly created graph, which will be represented with the data structures explained in GB_GRAPH. (The external variable *panic_code* is itself defined in GB_GRAPH.)

```
#define panic(c) { gb_free(node_blocks);
                   panic_code = c; gb_trouble_code = 0; return Λ; }
```

7. Now let's get going on the program. The C file `gb_words.c` begins as follows:

```
#include "gb_io.h"     /* we will use the GB_IO routines for input */
#include "gb_flip.h"     /* we will use the GB_FLIP routines for random numbers */
#include "gb_graph.h"     /* we will use the GB_GRAPH data structures */
#include "gb_sort.h"     /* and gb_linksort for sorting */
  ⟨ Preprocessor definitions ⟩
  ⟨ Type declarations 15 ⟩
  ⟨ Private variables 4 ⟩
  ⟨ Private functions 10 ⟩
  Graph *words(n, wt_vector, wt_threshold, seed)
      unsigned long n;     /* maximum number of vertices desired */
      long wt_vector[];     /* pointer to array of weights */
      long wt_threshold;     /* minimum qualifying weight */
      long seed;     /* random number seed */
  { ⟨ Local variables 8 ⟩
    gb_init_rand(seed);
    ⟨ Check that wt_vector is valid 9 ⟩;
    ⟨ Input the qualifying words to a linked list, computing their weights 18 ⟩;
    ⟨ Sort and output the words, determining adjacencies 22 ⟩;
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);     /* oops, we ran out of memory somewhere back there */
    }
    return new_graph;
  }
```

8. ⟨ Local variables 8 ⟩ ≡

```
  Graph *new_graph;     /* the graph constructed by words */
```

See also sections 14, 16, and 24.

This code is used in section 7.

9. Validating the weights. The first job that *words* needs to tackle is comparatively trivial: We want to verify the condition

$$\max\bigl(|a|,|b|\bigr)+C_1|w_1|+\cdots+C_7|w_7|<2^{30}. \qquad (*)$$

This proves to be an interesting exercise in "portable C programming," because we don't want to risk integer overflow. Our approach is to do the calculation first in floating point arithmetic, thereby ruling out cases that are clearly unacceptable. Once that test is passed, we can safely test the condition with ordinary integer arithmetic. Floating point arithmetic is system dependent, but we use it carefully so that system-independent results are obtained.

```
⟨ Check that wt_vector is valid 9 ⟩ ≡
  if (¬wt_vector) wt_vector = default_wt_vector;
  else { register double flacc;
    register long *p, *q;
    register long acc;
    ⟨ Use floating point arithmetic to check that wt_vector isn't totally off base 11 ⟩;
    ⟨ Use integer arithmetic to check that wt_vector is truly OK 12 ⟩;
  }
```

This code is used in section 7.

10. The floating-point calculations are facilitated by a routine that converts an integer to its absolute value, expressed as a **double**:

```
⟨ Private functions 10 ⟩ ≡
  static double flabs(x)
      long x;
  { if (x ≥ 0) return (double) x;
    return −((double) x);
  }
```

See also section 13.

This code is used in section 7.

11. Although floating point arithmetic is system dependent, we can certainly assume that at least 16 bits of precision are used. This implies that the difference between $flabs(x)$ and $|x|$ must be less than 2^{14}. Also, if x and y are nonnegative values less than 2^{31}, the difference between their floating-point sum and their true sum must be less than 2^{14}.

The floating point calculations in the following test will never reject a valid weight vector. For if condition $(*)$ holds, the floating-point value of $\max(flabs(a),flabs(b))+C_1*flabs(w_1)+\cdots+C_7*flabs(w_7)$ will be less than $2^{30}+(8+C_1+\cdots+C_7)2^{14}$, which is less than $2^{30}+2^{29}$.

```
⟨ Use floating point arithmetic to check that wt_vector isn't totally off base 11 ⟩ ≡
  p = wt_vector;
  flacc = flabs(*p++);
  if (flacc < flabs(*p)) flacc = flabs(*p);      /* now flacc = max(|a|, |b|) */
  for (q = &max_c[0]; q < &max_c[7]; q++) flacc += *q * flabs(*++p);
  if (flacc ≥ (double) #60000000)      /* this constant is 6 × 2^28 = 2^30 + 2^29 */
    panic(very_bad_specs);      /* whoa; the weight vector is way too big */
```

This code is used in section 9.

12. Conversely, if the floating point test just made is passed, the true value of the sum will be less than $2^{30}+2^{29}+2^{29}=2^{31}$; hence integer overflow will never occur when we make the following more refined test:

```
⟨ Use integer arithmetic to check that wt_vector is truly OK 12 ⟩ ≡
  p = wt_vector;
  acc = iabs(*p++);
  if (acc < iabs(*p)) acc = iabs(*p);      /* now acc = max(|a|, |b|) */
  for (q = &max_c[0]; q < &max_c[7]; q++) acc += *q * iabs(*++p);
  if (acc ≥ #40000000) panic(bad_specs);      /* the weight vector is a bit too big */
```

This code is used in section 9.

13.

```
⟨ Private functions 10 ⟩ +≡
  static long iabs(x)
        long x;
  { if (x ≥ 0) return (long) x;
    return −((long) x);
  }
```

14. The input phase. Now we're ready to read `words.dat`.

```
⟨ Local variables 8 ⟩ +≡
  register long wt;     /* the weight of the current word */
  char word[5];     /* the current five-letter word */
  long nn = 0;     /* the number of qualifying words found so far */
```

15. As we read the words, we will form a linked list of nodes containing each qualifying word and its weight, using the memory management routines of GB_GRAPH to allocate space for 111 nodes at a time. These nodes should be returned to available memory later, so we will keep them in a separate area under local control.

The nodes start out with *key* and *link* fields, as required by the *gb_linksort* routine, which we'll use to sort by weight. The sort key must be nonnegative; we obtain it by adding 2^{30} to the weight.

```
#define nodes_per_block 111
⟨ Type declarations 15 ⟩ ≡
  typedef struct node_struct {
    long key;     /* the sort key (weight plus 2^30) */
    struct node_struct *link;     /* links the nodes together */
    char wd[5];     /* five-letter word (which typically consumes eight bytes, too bad) */
  } node;
```

See also section 23.

This code is used in section 7.

16.
```
⟨ Local variables 8 ⟩ +≡
  node *next_node;     /* the next node available for allocation */
  node *bad_node;     /* if next_node = bad_node, the node isn't really there */
  node *stack_ptr;     /* the most recently created node */
  node *cur_node;     /* current node being created or examined */
```

17.
```
⟨ Private variables 4 ⟩ +≡
  static Area node_blocks;     /* the memory area for blocks of nodes */
```

18.
```
⟨ Input the qualifying words to a linked list, computing their weights 18 ⟩ ≡
  next_node = bad_node = stack_ptr = Λ;
  if (gb_open("words.dat") ≠ 0) panic(early_data_fault);
      /* couldn't open "words.dat" using GraphBase conventions; io_errors tells why */
  do ⟨ Read one word, and put it on the stack if it qualifies 19 ⟩ while (¬gb_eof( ));
  if (gb_close( ) ≠ 0) panic(late_data_fault);     /* something's wrong with "words.dat"; see io_errors */
```

This code is used in section 7.

19.
```
⟨ Read one word, and put it on the stack if it qualifies 19 ⟩ ≡
  { register long j;     /* position in word */
    for (j = 0; j < 5; j++) word[j] = gb_char( );
    ⟨ Compute the weight wt 21 ⟩;
    if (wt ≥ wt_threshold) {     /* it qualifies */
      ⟨ Install word and wt in a new node 20 ⟩;
      nn++;
    }
    gb_newline( );
  }
```

This code is used in section 18.

20. #**define** *copy5*(y, x)
{ $*(y) = *(x)$; $*((y)+1) = *((x)+1)$; $*((y)+2) = *((x)+2)$;
$*((y)+3) = *((x)+3)$; $*((y)+4) = *((x)+4)$; }

⟨ Install *word* and *wt* in a new node 20 ⟩ ≡
```
if (next_node ≡ bad_node) {
  cur_node = gb_typed_alloc(nodes_per_block, node, node_blocks);
  if (cur_node ≡ Λ) panic(no_room + 1);      /* out of memory already */
  next_node = cur_node + 1;
  bad_node = cur_node + nodes_per_block;
} else cur_node = next_node++;
cur_node→key = wt + #40000000;
cur_node→link = stack_ptr;
copy5(cur_node→wd, word);
stack_ptr = cur_node;
```
This code is used in section 19.

21. Recall that *gb_number*( ) returns 0, without giving an error, if no digit is present in the current position of the file being read. This implies that the `words.dat` file need not include zero counts explicitly. Furthermore, we can arrange things so that trailing zero counts are unnecessary; commas can be omitted if all counts following them on the current line are zero.

⟨ Compute the weight *wt* 21 ⟩ ≡
```
{ register long *p, *q;      /* pointers to C_j and w_j */
  register long c;      /* current count */
  switch (gb_char( )) {
  case '*': wt = wt_vector[0]; break;      /* 'common' word */
  case '+': wt = wt_vector[1]; break;      /* 'advanced' word */
  case '␣': case '\n': wt = 0; break;      /* 'unusual' word */
  default: panic(syntax_error);      /* unknown type of word */
  }
  p = &max_c[0];
  q = &wt_vector[2];
  do {
    if (p ≡ &max_c[7]) panic(syntax_error + 1);      /* too many counts */
    c = gb_number(10);
    if (c > *p++) panic(syntax_error + 2);      /* count too large */
    wt += c * *q++;
  } while (gb_char( ) ≡ ',');
}
```
This code is used in section 19.

22. The output phase. Once the input phase has examined all of `words.dat`, we are left with a stack of *nn* nodes containing the qualifying words, starting at *stack_ptr*.

The next step is to call *gb_linksort*, which takes the qualifying words and distributes them into the 128 lists *gb_sorted*[*j*], for $0 \le j < 128$. We can then access the words in order of decreasing weight by reading through these lists, starting with *gb_sorted*[127] and ending with *gb_sorted*[0]. (See the documentation of *gb_linksort* in the GB_SORT module.)

The output phase therefore has the following general outline:

```
⟨ Sort and output the words, determining adjacencies 22 ⟩ ≡
  gb_linksort(stack_ptr);
  ⟨ Allocate storage for the new graph; adjust n if it is zero or too large 27 ⟩;
  if (gb_trouble_code ≡ 0 ∧ n) {
    register long j;    /* runs through sorted lists */
    register node *p;    /* the current node being output */
    nn = n;
    for (j = 127; j ≥ 0; j−−)
      for (p = (node *) gb_sorted[j]; p; p = p→link) {
        ⟨ Add the word p→wd to the graph 28 ⟩;
        if (−−nn ≡ 0) goto done;
      }
  }
done: gb_free(node_blocks);
```

This code is used in section 7.

23. The only slightly unusual data structure needed is a set of five hash tables, one for each of the strings of four letters obtained by suppressing a single letter of a five-letter word. For example, a word like '`words`' will lead to entries for '`␣ords`', '`w␣rds`', '`wo␣ds`', '`wor␣s`', and '`word␣`', one in each of the hash tables.

```
#define hash_prime 6997    /* a prime number larger than the total number of words */
⟨ Type declarations 15 ⟩ +≡
  typedef Vertex *hash_table[hash_prime];
```

24.
```
⟨ Local variables 8 ⟩ +≡
  Vertex *cur_vertex;    /* the current vertex being created or examined */
  char *next_string;    /* where we'll store the next five-letter word */
```

25.
```
⟨ Private variables 4 ⟩ +≡
  static hash_table *htab;    /* five dynamically allocated hash tables */
```

26. The weight of each word will be stored in the utility field *u.I* of its **Vertex** record. The position in which adjacent words differ will be stored in utility field *a.I* of the **Arc** records between them.

```
#define weight u.I    /* weighted frequencies */
#define loc a.I    /* index of difference (0, 1, 2, 3, or 4) */
⟨ gb_words.h 1 ⟩ +≡
#define weight u.I    /* repeat the definitions in the header file */
#define loc a.I
```

27. ⟨ Allocate storage for the new graph; adjust n if it is zero or too large 27 ⟩ ≡

```
if (n ≡ 0 ∨ nn < n) n = nn;
new_graph = gb_new_graph(n);
if (new_graph ≡ Λ) panic(no_room);      /* out of memory before we're even started */
if (wt_vector ≡ default_wt_vector)
  sprintf(new_graph→id, "words(%lu,0,%ld,%ld)", n, wt_threshold, seed);
else sprintf(new_graph→id, "words(%lu,{%ld,%ld,%ld,%ld,%ld,%ld,%ld,%ld,%ld},%ld,%ld)", n,
      wt_vector[0], wt_vector[1], wt_vector[2], wt_vector[3], wt_vector[4], wt_vector[5], wt_vector[6],
      wt_vector[7], wt_vector[8], wt_threshold, seed);
strcpy(new_graph→util_types, "IZZZZZIZZZZZZZ");
cur_vertex = new_graph→vertices;
next_string = gb_typed_alloc(6 * n, char, new_graph→data);
htab = gb_typed_alloc(5, hash_table, new_graph→aux_data);
```

This code is used in section 22.

28. ⟨ Add the word p→wd to the graph 28 ⟩ ≡

```
{ register char *q;      /* the new word */
  q = cur_vertex→name = next_string;
  next_string += 6;
  copy5(q, p→wd);
  cur_vertex→weight = p→key - #40000000;
  ⟨ Add edges for all previous words r that nearly match q 29 ⟩;
  cur_vertex++;
}
```

This code is used in section 22.

29. The length of each edge in a *words* graph is set to 1; the calling routine can change it later if desired.

```
#define mtch(i) (*(q+i) ≡ *(r+i))
#define match(a,b,c,d) (mtch(a) ∧ mtch(b) ∧ mtch(c) ∧ mtch(d))
#define store_loc_of_diff(k) cur_vertex→arcs→loc = (cur_vertex→arcs − 1)→loc = k
#define ch(q) ((long) *(q))
#define hdown(k) h ≡ htab[k] ? h = htab[k+1] − 1 : h−−
⟨ Add edges for all previous words r that nearly match q 29 ⟩ ≡
  { register char *r;      /* previous word possibly adjacent to q */
    register Vertex **h;      /* hash address for linear probing */
    register long raw_hash;      /* five-letter hash code before remaindering */
    raw_hash = (((((((ch(q) ≪ 5) + ch(q+1)) ≪ 5) + ch(q+2)) ≪ 5) + ch(q+3)) ≪ 5) + ch(q+4);
    for (h = htab[0] + (raw_hash − (ch(q) ≪ 20)) % hash_prime; *h; hdown(0)) {
      r = (*h)→name;
      if (match(1,2,3,4)) gb_new_edge(cur_vertex, *h, 1L), store_loc_of_diff(0);
    }
    *h = cur_vertex;
    for (h = htab[1] + (raw_hash − (ch(q+1) ≪ 15)) % hash_prime; *h; hdown(1)) {
      r = (*h)→name;
      if (match(0,2,3,4)) gb_new_edge(cur_vertex, *h, 1L), store_loc_of_diff(1);
    }
    *h = cur_vertex;
    for (h = htab[2] + (raw_hash − (ch(q+2) ≪ 10)) % hash_prime; *h; hdown(2)) {
      r = (*h)→name;
      if (match(0,1,3,4)) gb_new_edge(cur_vertex, *h, 1L), store_loc_of_diff(2);
    }
    *h = cur_vertex;
    for (h = htab[3] + (raw_hash − (ch(q+3) ≪ 5)) % hash_prime; *h; hdown(3)) {
      r = (*h)→name;
      if (match(0,1,2,4)) gb_new_edge(cur_vertex, *h, 1L), store_loc_of_diff(3);
    }
    *h = cur_vertex;
    for (h = htab[4] + (raw_hash − ch(q+4)) % hash_prime; *h; hdown(4)) {
      r = (*h)→name;
      if (match(0,1,2,3)) gb_new_edge(cur_vertex, *h, 1L), store_loc_of_diff(4);
    }
    *h = cur_vertex;
  }
```

This code is used in section 28.

30. Finding a word. After *words* has created a graph g, the user can remove the hash tables by calling *gb_free*(*g*¬*aux_data*). But if the hash tables have not been removed, another procedure can be used to find vertices that match or nearly match a given word.

The subroutine call *find_word*(q, f) will return a pointer to a vertex that matches a given five-letter word q, if that word is in the graph; otherwise, it returns Λ (i.e., `NULL`), after calling $f(v)$ for each vertex v whose word matches q in all but one letter position.

```
Vertex *find_word(q, f)
    char *q;
    void (*f)( );    /* *f should take one argument, of type Vertex *, or f should be Λ */
{ register char *r;    /* previous word possibly adjacent to q */
  register Vertex **h;    /* hash address for linear probing */
  register long raw_hash;    /* five-letter hash code before remaindering */
  raw_hash = (((((((ch(q) ≪ 5) + ch(q + 1)) ≪ 5) + ch(q + 2)) ≪ 5) + ch(q + 3)) ≪ 5) + ch(q + 4);
  for (h = htab[0] + (raw_hash − (ch(q) ≪ 20)) % hash_prime; *h; hdown(0)) {
    r = (*h)¬name;
    if (mtch(0) ∧ match(1, 2, 3, 4)) return *h;
  }
  ⟨ Invoke f on every vertex that is adjacent to word q 31 ⟩;
  return Λ;
}
```

31. ⟨ Invoke f on every vertex that is adjacent to word q 31 ⟩ ≡

```
if (f) {
  for (h = htab[0] + (raw_hash − (ch(q) ≪ 20)) % hash_prime; *h; hdown(0)) {
    r = (*h)¬name;
    if (match(1, 2, 3, 4)) (*f)(*h);
  }
  for (h = htab[1] + (raw_hash − (ch(q + 1) ≪ 15)) % hash_prime; *h; hdown(1)) {
    r = (*h)¬name;
    if (match(0, 2, 3, 4)) (*f)(*h);
  }
  for (h = htab[2] + (raw_hash − (ch(q + 2) ≪ 10)) % hash_prime; *h; hdown(2)) {
    r = (*h)¬name;
    if (match(0, 1, 3, 4)) (*f)(*h);
  }
  for (h = htab[3] + (raw_hash − (ch(q + 3) ≪ 5)) % hash_prime; *h; hdown(3)) {
    r = (*h)¬name;
    if (match(0, 1, 2, 4)) (*f)(*h);
  }
  for (h = htab[4] + (raw_hash − ch(q + 4)) % hash_prime; *h; hdown(4)) {
    r = (*h)¬name;
    if (match(0, 1, 2, 3)) (*f)(*h);
  }
}
```

This code is used in section 30.

32. Index. Here is a list that shows where the identifiers of this program are defined and used.

August 4, 2025 at 10:32

GB_WORDS

Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.