1. Introduction. This short GraphBase module provides a simple utility routine called *gb_linksort*, which is used in many of the other programs.

```
#include <stdio.h>      /* the NULL pointer (Λ) is defined here */
#include "gb_flip.h"      /* we need to use the random number generator */
  ⟨ Preprocessor definitions ⟩
  ⟨ Declarations 2 ⟩
  ⟨ The gb_linksort routine 5 ⟩
```

2. Most of the graphs obtained from GraphBase data are parameterized, so that different effects can be obtained easily from the same underlying body of information. In many cases the desired graph is determined by selecting the "heaviest" vertices according to some notion of "weight," and/or by taking a random sample of vertices. For example, the GraphBase routine *words*(n, *wt_vector*, *wt_threshold*, *seed*) creates a graph based on the n most common five-letter words of English, where common-ness is determined by a given weight vector. When several words have equal weight, we want to choose between them at random. In particular, this means that we can obtain a completely random choice of words if the weight vector assigns the same weight to each word.

The *gb_linksort* routine is a convenient tool for this purpose. It takes a given linked list of nodes and shuffles their link fields so that the nodes can be read in decreasing order of weight, and so that equal-weight nodes appear in random order. *Note: The random number generator of* GB_FLIP *must be initialized before gb_linksort is called.*

The nodes sorted by *gb_linksort* can be records of any structure type, provided only that the first field is '**long** *key*' and the second field is '**struct** *this_struct_type* **link*'. Further fields are not examined. The **node** type defined in this section is the simplest possible example of such a structure.

Sorting is done by means of the *key* fields, which must each contain nonnegative integers less than 2^{31}.

After sorting is complete, the data will appear in 128 linked lists: *gb_sorted*[127], *gb_sorted*[126], ..., *gb_sorted*[0]. To examine the nodes in decreasing order of weight, one can read through these lists with a routine such as

```
{
  int j;
  node *p;
  for (j = 127; j ≥ 0; j−−)
    for (p = (node *) gb_sorted[j]; p; p = p→link)
      look_at(p);
}
```

All nodes whose keys are in the range $j \cdot 2^{24} \le key < (j+1) \cdot 2^{24}$ will appear in list *gb_sorted*[j]. Therefore the results will all be found in the single list *gb_sorted*[0], if all the keys are strictly less than 2^{24}.

format *node* *int*

```
⟨ Declarations 2 ⟩ ≡
  typedef struct node_struct {
    long key;      /* a numeric quantity, assumed nonnegative */
    struct node_struct *link;      /* the next node on a list */
  } node;      /* applications of gb_linksort may have other fields after link */
```

See also section 4.

This code is used in section 1.

3. In the header file, *gb_sorted* is declared to be an array of pointers to **char**, since nodes may have different types in different applications. User programs should cast *gb_sorted* to the appropriate type as in the example above.

⟨ `gb_sort.h` 3 ⟩ ≡
```
extern void gb_linksort();    /* procedure to sort a linked list */
extern char *gb_sorted[];    /* the results of gb_linksort */
```

4. Six passes of a radix sort, using radix 256, will accomplish the desired objective rather quickly. (See, for example, Algorithm 5.2.5R in *Sorting and Searching*.) The first two passes use random numbers instead of looking at the key fields, thereby effectively extending the keys so that nodes with equal keys will appear in reasonably random order.

We move the nodes back and forth between two arrays of lists: the external array *gb_sorted* and a private array called *alt_sorted*.

⟨ Declarations 2 ⟩ +≡
```
node *gb_sorted[256];    /* external bank of lists, for even-numbered passes */
static node *alt_sorted[256];    /* internal bank of lists, for odd-numbered passes */
```

5. So here we go with six passes over the data.

⟨ The *gb_linksort* routine 5 ⟩ ≡
```
void gb_linksort(l)
    node *l;
{ register long k;    /* index to destination list */
  register node **pp;    /* current place in list of pointers */
  register node *p, *q;    /* pointers for list manipulation */
  ⟨ Partition the given list into 256 random sublists alt_sorted 6 ⟩;
  ⟨ Partition the alt_sorted lists into 256 random sublists gb_sorted 7 ⟩;
  ⟨ Partition the gb_sorted lists into alt_sorted by low-order byte 8 ⟩;
  ⟨ Partition the alt_sorted lists into gb_sorted by second-lowest byte 9 ⟩;
  ⟨ Partition the gb_sorted lists into alt_sorted by second-highest byte 10 ⟩;
  ⟨ Partition the alt_sorted lists into gb_sorted by high-order byte 11 ⟩;
}
```
This code is used in section 1.

6. ⟨ Partition the given list into 256 random sublists *alt_sorted* 6 ⟩ ≡
```
for (pp = alt_sorted + 255; pp ≥ alt_sorted; pp−−) *pp = Λ;    /* empty all the destination lists */
for (p = l; p; p = q) {
  k = gb_next_rand() ≫ 23;    /* extract the eight most significant bits */
  q = p→link;
  p→link = alt_sorted[k];
  alt_sorted[k] = p;
}
```
This code is used in section 5.

7. ⟨ Partition the *alt_sorted* lists into 256 random sublists *gb_sorted* 7 ⟩ ≡

$\textbf{for}\ (pp = gb\_sorted + 255;\ pp \ge gb\_sorted;\ pp{-}{-})\ {*}pp = \Lambda;$ /* empty all the destination lists */
$\textbf{for}\ (pp = alt\_sorted + 255;\ pp \ge alt\_sorted;\ pp{-}{-})$
  $\textbf{for}\ (p = {*}pp;\ p;\ p = q)\ \{$
    $k = gb\_next\_rand(\,) \gg 23;$ /* extract the eight most significant bits */
    $q = p\text{-}{>}link;$
    $p\text{-}{>}link = gb\_sorted[k];$
    $gb\_sorted[k] = p;$
  $\}$

This code is used in section 5.

8. ⟨ Partition the *gb_sorted* lists into *alt_sorted* by low-order byte 8 ⟩ ≡

$\textbf{for}\ (pp = alt\_sorted + 255;\ pp \ge alt\_sorted;\ pp{-}{-})\ {*}pp = \Lambda;$ /* empty all the destination lists */
$\textbf{for}\ (pp = gb\_sorted + 255;\ pp \ge gb\_sorted;\ pp{-}{-})$
  $\textbf{for}\ (p = {*}pp;\ p;\ p = q)\ \{$
    $k = p\text{-}{>}key\ \&\ \texttt{\#ff};$ /* extract the eight least significant bits */
    $q = p\text{-}{>}link;$
    $p\text{-}{>}link = alt\_sorted[k];$
    $alt\_sorted[k] = p;$
  $\}$

This code is used in section 5.

9. Here we must read from *alt_sorted* from 0 to 255, not from 255 to 0, to get the desired final order. (Each pass reverses the order of the lists; it's tricky, but it works.)

⟨ Partition the *alt_sorted* lists into *gb_sorted* by second-lowest byte 9 ⟩ ≡

$\textbf{for}\ (pp = gb\_sorted + 255;\ pp \ge gb\_sorted;\ pp{-}{-})\ {*}pp = \Lambda;$ /* empty all the destination lists */
$\textbf{for}\ (pp = alt\_sorted;\ pp < alt\_sorted + 256;\ pp{+}{+})$
  $\textbf{for}\ (p = {*}pp;\ p;\ p = q)\ \{$
    $k = (p\text{-}{>}key \gg 8)\ \&\ \texttt{\#ff};$ /* extract the next eight bits */
    $q = p\text{-}{>}link;$
    $p\text{-}{>}link = gb\_sorted[k];$
    $gb\_sorted[k] = p;$
  $\}$

This code is used in section 5.

10. ⟨ Partition the *gb_sorted* lists into *alt_sorted* by second-highest byte 10 ⟩ ≡

$\textbf{for}\ (pp = alt\_sorted + 255;\ pp \ge alt\_sorted;\ pp{-}{-})\ {*}pp = \Lambda;$ /* empty all the destination lists */
$\textbf{for}\ (pp = gb\_sorted + 255;\ pp \ge gb\_sorted;\ pp{-}{-})$
  $\textbf{for}\ (p = {*}pp;\ p;\ p = q)\ \{$
    $k = (p\text{-}{>}key \gg 16)\ \&\ \texttt{\#ff};$ /* extract the next eight bits */
    $q = p\text{-}{>}link;$
    $p\text{-}{>}link = alt\_sorted[k];$
    $alt\_sorted[k] = p;$
  $\}$

This code is used in section 5.

11. The most significant bits will lie between 0 and 127, because we assumed that the keys are nonnegative and less than 2^{31}. (A similar routine would be able to sort signed integers, or unsigned long integers, but the C code would not then be portable.)

⟨ Partition the *alt_sorted* lists into *gb_sorted* by high-order byte 11 ⟩ ≡

```
for (pp = gb_sorted + 255; pp ≥ gb_sorted; pp−−) *pp = Λ;   /* empty all the destination lists */
for (pp = alt_sorted; pp < alt_sorted + 256; pp++)
  for (p = *pp; p; p = q) {
    k = (p→key ≫ 24) & #ff;   /* extract the most significant bits */
    q = p→link;
    p→link = gb_sorted[k];
    gb_sorted[k] = p;
  }
```

This code is used in section 5.

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

May 19, 2018 at 02:29

GB_SORT

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