by Jon Bentley
with Special Guest Oysters
Don Knuth and Doug McIlroy

programming pearls

A LITERATE PROGRAM

Last month's column introduced Don Knuth's style of "Literate Programming" and his WEB *system for building programs that are works of literature. This column presents a literate program by Knuth (its origins are sketched in last month's column) and, as befits literature, a review. So without further ado, here is Knuth's program, retypeset in* Communications *style.* —*Jon Bentley*

Common Words Section

1. Introduction. The purpose of this program is to solve the following problem posed by Jon Bentley:

> Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

Jon intentionally left the problem somewhat vague, but he stated that "a user should be able to find the 100 most frequent words in a twenty-page technical paper (roughly a 50K byte file) without undue emotional trauma."

Let us agree that a *word* is a sequence of one or more contiguous letters; `"Bentley"` is a word, but `"ain't"` isn't. The sequence of letters should be maximal, in the sense that it cannot be lengthened without including a nonletter. Uppercase letters are considered equivalent to their lowercase counterparts, so that the words `"Bentley"` and `"BENTLEY"` and `"bentley"` are essentially identical.

The given problem still isn't well defined, for the file might contain more than k words, all of the same frequency; or there might not even be as many as k words. Let's be more precise: The most common words are to be printed in order of decreasing frequency, with words of equal frequency listed in alphabetic order. Printing should stop after k words have been output, if more than k words are present.

2. The *input* file is assumed to contain the given text. If it begins with a positive decimal number (preceded by optional blanks), that number will be the value of k; otherwise we shall assume that $k = 100$. Answers will be sent to the *output* file.

```
define default_k = 100 {use this value if k isn't
          otherwise specified}
```

3. Besides solving the given problem, this program is supposed to be an example of the WEB system, for people who know some Pascal but who have never seen WEB before. Here is an outline of the program to be constructed:

```
program common_words (input, output);
  type ⟨Type declarations 17⟩
  var ⟨Global variables 4⟩
  ⟨Procedures for initialization 5⟩
  ⟨Procedures for input and output 9⟩
  ⟨Procedures for data manipulation 20⟩
  begin ⟨The main program 8⟩;
  end.
```

4. The main idea of the WEB approach is to let the program grow in natural stages, with its parts presented in roughly the order that they might have been written by a programmer who isn't especially clairvoyant.

For example, each global variable will be introduced when we first know that it is necessary or desirable; the WEB system will take care of collecting these declarations into the proper place. We already know about one global variable, namely the number that Bentley called k. Let us give it the more descriptive name *max_words_to_print*.

```
⟨Global variables 4⟩ ≡
max_words_to_print: integer;
    {at most this many words will be printed}
```

See also sections 11, 13, 18, 22, 32, and 36.
This code is used in section 3.

5. As we introduce new global variables, we'll often want to give them certain starting values. This will be done by the *initialize* procedure, whose body will consist of various pieces of code to be specified when we think of particular kinds of initialization.

```
⟨Procedures for initialization 5⟩ ≡
procedure initialize;
  var i: integer;   {all-purpose index for initializa-
        tions}
  begin ⟨Set initial values 12⟩
  end;
```

This code is used in section 3.

6. The WEB system, which may be thought of as a preprocessor for Pascal, includes a macro definition facility so that portable programs are easier to write. For example, we have already defined '*default_k*' to be 100. Here are two more examples of WEB macros; they allow us to write, e.g., '*incr*(*count*[*p*])' as a convenient abbreviation for the statement '*count*[*p*] ← *count*[*p*] + 1'.

```
  define incr(#) ≡ # ← # + 1   {increment a vari-
        able}
  define decr(#) ≡ # ← # − 1   {decrement a vari-
        able}
```

7. Some of the procedures we shall be writing come to abrupt conclusions; hence it will be convenient to introduce a '**return**' macro for the operation of jumping to the end of the procedure. A symbolic label '*exit*' will be declared in all such procedures, and '*exit*:' will be placed just before the final **end**. (No other labels or **goto** statements are used in the present program, but the author would find it painful to eliminate these particular ones.)

```
  define exit = 30  {the end of a procedure}
  define return ≡ goto exit   {quick termination}
  format return ≡ nil   {typeset 'return' in boldface}
```

8. Strategic considerations. What algorithms and data structures should be used for Bentley's problem? Clearly we need to be able to recognize different occurrences of the same word, so some sort of internal dictionary is necessary. There's no obvious way to decide that a particular word of the input cannot possibly be in the final set, until we've gotten very near the end of the file; so we might as well remember every word that appears.

There should be a frequency count associated with each word, and we will eventually want to run through the words in order of decreasing frequency. But there's no need to keep these counts in order as we read through the input, since the order matters only at the end.

Therefore it makes sense to structure our program as follows:

```
⟨The main program 8⟩ ≡
  initialize;
  ⟨Establish the value of max_words_to_print 10⟩;
  ⟨Input the text, maintaining a dictionary with
    frequency counts 34⟩;
  ⟨Sort the dictionary by frequency 39⟩;
  ⟨Output the results 41⟩
```

This code is used in section 3.

9. Basic input routines. Let's switch to a bottom-up approach now, by writing some of the procedures that we know will be necessary sooner or later. Then we'll have some confidence that our program is taking shape, even though we haven't decided yet how to handle the searching or the sorting. It will be nice to get the messy details of Pascal input out of the way and off our minds.

Here's a function that reads an optional positive integer, returning zero if none is present at the beginning of the current line.

```
⟨Procedures for input and output 9⟩ ≡
function read_int: integer;
  var n: integer;   {the accumulated value}
  begin n ← 0;
  if ¬eof then
    begin while (¬eoln) ∧ (input↑ = '␣') do
      get(input);
    while (input↑ ≥ '0') ∧ (input↑ ≤ '9') do
      begin n ← 10*n + ord(input↑) − ord('0');
      get(input);
      end;
    end;
  read_int ← n;
  end;
```

See also sections 15, 35, and 40.
This code is used in section 3.

10. We invoke *read_int* only once.

```
⟨Establish the value of max_words_to_print 10⟩ ≡
  max_words_to_print ← read_int;
  if max_words_to_print = 0 then
    max_words_to_print ← default_k
```

This code is used in section 8.

11. To find words in the *input* file, we want a quick way to distinguish letters from nonletters. Pascal has conspired to make this problem somewhat tricky, because it leaves many details of the character set undefined. We shall define two arrays, *lowercase* and *uppercase*, that specify the letters of the alphabet. A third array, *lettercode*, maps arbitrary characters into the integers 0 . . 26.

If c is a value of type *char* that represents the kth letter of the alphabet, then $lettercode[ord(c)] = k$; but if c is a nonletter, $lettercode[ord(c)] = 0$. We assume that $0 \le ord(c) \le 255$ whenever c is of type *char*.

```
⟨Global variables 4⟩ +≡
lowercase, uppercase: array [1 . . 26] of char;
    {the letters}
lettercode; array [0 . . 255] of 0 . . 26;
    {the input conversion table}
```

12. A somewhat tedious set of assignments is necessary for the definition of *lowercase* and *uppercase*, because letters need not be consecutive in Pascal's character set.

```
⟨Set initial values 12⟩ ≡
  lowercase[1] ← 'a'; uppercase[1] ← 'A';
  lowercase[2] ← 'b'; uppercase[2] ← 'B';
  lowercase[3] ← 'c'; uppercase[3] ← 'C';
  lowercase[4] ← 'd'; uppercase[4] ← 'D';
  lowercase[5] ← 'e'; uppercase[5] ← 'E';
  lowercase[6] ← 'f'; uppercase[6] ← 'F';
  lowercase[7] ← 'g'; uppercase[7] ← 'G';
  lowercase[8] ← 'h'; uppercase[8] ← 'H';
  lowercase[9] ← 'i'; uppercase[9] ← 'I';
  lowercase[10] ← 'j'; uppercase[10] ← 'J';
  lowercase[11] ← 'k'; uppercase[11] ← 'K';
  lowercase[12] ← 'l'; uppercase[12] ← 'L';
  lowercase[13] ← 'm'; uppercase[13] ← 'M';
  lowercase[14] ← 'n'; uppercase[14] ← 'N';
  lowercase[15] ← 'o'; uppercase[15] ← 'O';
  lowercase[16] ← 'p'; uppercase[16] ← 'P';
  lowercase[17] ← 'q'; uppercase[17] ← 'Q';
  lowercase[18] ← 'r'; uppercase[18] ← 'R';
  lowercase[19] ← 's'; uppercase[19] ← 'S';
  lowercase[20] ← 't'; uppercase[20] ← 'T';
  lowercase[21] ← 'u'; uppercase[21] ← 'U';
  lowercase[22] ← 'v'; uppercase[22] ← 'V';
  lowercase[23] ← 'w'; uppercase[23] ← 'W';
  lowercase[24] ← 'x'; uppercase[24] ← 'X';
  lowercase[25] ← 'y'; uppercase[25] ← 'Y';
  lowercase[26] ← 'z'; uppercase[26] ← 'Z';
  for i ← 0 to 255 do lettercode [i] ← 0;
  for i ← 1 to 26 do
    begin lettercode[ord(lowercase[i])] ← i;
    lettercode[ord(uppercase[i])] ← i;
    end;
```

See also sections 14, 19, 23, and 33.
This code is used in section 5.

13. Each new word found in the input will be placed into a *buffer* array. We shall assume that no words are more than 60 letters long; if a longer word appears, it will be truncated to 60 characters, and a warning message will be printed at the end of the run.

```
  define max_word_length = 60
      {words shouldn't be longer than this}
⟨Global variables 4⟩ +≡
buffer: array [1 . . max_word_length] of 1 . . 26;
    {the current word}
word_length: 0 . . max_word_length;
    {the number of active letters currently in buffer}
word_truncated: boolean;
    {was some word longer than max_word_length?}
```

14. ⟨Set initial values 12⟩ +≡

```
  word_truncated ← false;
```

15. We're ready now for the main input routine, which puts the next word into the buffer. If no more words remain, *word_length* is set to zero; otherwise *word_length* is set to the length of the new word.

```
⟨Procedures for input and output 9⟩ +≡
procedure get_word;
  label exit;   {enable a quick return}
  begin word_length ← 0;
    if ¬eof then
    begin while lettercode[ord(input ↑)] = 0 do
      if ¬eoln then get(input)
      else begin read_ln;
        if eof then return;
        end;
    ⟨Read a word into buffer 16⟩;
    end;
exit: end;
```

16. At this point $lettercode[ord(input \uparrow)] > 0$, hence $input \uparrow$ contains the first letter of a word.

```
⟨Read a word into buffer 16⟩ ≡
repeat if word_length = max_word_length then
      word_truncated ← true
  else begin incr(word_length);
      buffer[word_length] ← lettercode[ord(input ↑)];
      end;
  get(input);
until lettercode[ord(input ↑)] = 0
```

This code is used in section 15.

17. Dictionary lookup. Given a word in the buffer, we will want to look for it in a dynamic dictionary of all words that have appeared so far. We expect many words to occur often, so we want a search technique that will find existing words quickly. Furthermore, the dictionary should accommodate words of variable length, and (ideally) it should also facilitate the task of alphabetic ordering.

These constraints suggest a variant of the data structure introduced by Frank M. Liang in his Ph.D. thesis ["Word Hy-phen-a-tion by Com-pu-ter," Stanford University, 1983]. Liang's structure, which we may call a *hash trie*, requires comparatively few operations to find a word that is already present, although it may take somewhat longer to insert a new entry. Some space is sacrificed—we will need two pointers, a count, and another 5-bit field for each character in the dictionary, plus extra space to keep the hash table from becoming congested—but relatively large memories are commonplace nowadays, so the method seems ideal for the present application.

A trie represents a set of words and all prefixes of those words [cf. Knuth, *Sorting and Searching*, Section 6.3]. For convenience, we shall say that all nonempty prefixes of the words in our dictionary are also words, even though they may not occur as "words" in the input file. Each word (in this generalized sense) is represented by a *pointer*, which is an index into four large arrays called *link, sibling, count,* and *ch*.

```
define trie_size = 32767   {the largest pointer value}
⟨Type declarations 17⟩ ≡
  pointer = 0 . . trie_size;
```

This code is used in section 3.

18. One-letter words are represented by the pointers 1 through 26. The representation of longer words is defined recursively: If p represents word w and if $1 \le c \le 26$, then the word w followed by the cth letter of the alphabet is represented by $link[p] + c$.

For example, suppose that $link[2] = 1000$, $link[1005] = 2000$, and $link[2015] = 3000$. Then the word "b" is represented by the pointer value 2; "be" is represented by $link[2] + 5 = 1005$; "ben" is represented by 2015; and "bent" by 3021. If no longer word beginning with "bent" appears in the dictionary, $link[3021]$ will be zero.

The hash trie also contains redundant information to facilitate traversal and updating. If $link[p]$ is nonzero, then $link[link[p]] = p$. Furthermore if $q = link[p] + c$ is a "child" of p, we have $ch[q] = c$; this makes it possible to go from child to parent, since $link[q - ch[q]] = link[link[p]] = p$.

Children of the same parent are linked by *sibling* pointers: The largest child of p is $sibling[link[p]]$, and the next largest is $sibling[sibling[link[p]]]$; the smallest child's *sibling* pointer is $link[p]$. Continuing our earlier example, if all words in the dictionary beginning with "be" start with either "ben" or "bet", then $sibling[2000] = 2021$, $sibling[2021] = 2015$, and $sibling[2015] = 2000$.

Notice that children of different parents might appear next to each other. For example, we might have $ch[2020] = 6$, for the child of some word such that $link[p] = 2014$.

If $link[p] \ne 0$, the table entry in position $link[p]$ is called the "header" of p's children. The special code value *header* appears in the *ch* field of each header entry.

If p represents a word, $count[p]$ is the number of times that the word has occurred in the input so far. The *count* field in a header entry is undefined.

Unused positions p have $ch[p] = empty\_slot$. In this case $link[p]$, $sibling[p]$, and $count[p]$ are undefined.

```
define empty_slot = 0
define header = 27
define move_to_prefix(#) ≡ # ← link[# − ch[#]]
define move_to_last_suffix(#) ≡
      while link[#] ≠ 0 do # ← sibling[link[#]]
⟨Global variables 4⟩ +≡
link, sibling: array [pointer] of pointer;
ch: array [pointer] of empty_slot . . header;
```

19.
```
⟨Set initial values 12⟩ +≡
  for i ← 27 to trie_size do ch[i] ← empty_slot;
  for i ← 1 to 26 do
    begin ch[i] ← i; link[i] ← 0; count[i] ← 0;
    sibling[i] ← i − 1;
    end;
  ch[0] ← header; link[0] ← 0; sibling[0] ← 26;
```

20. Here's the basic subroutine that finds a given word in the dictionary. The word will be inserted (with a *count* of zero) if it isn't already present.

More precisely, the *find_buffer* function looks for the contents of *buffer*, and returns a pointer to the appropriate dictionary location. If the dictionary is so full that a new word cannot be inserted, the pointer 0 is returned.

```
define abort_find ≡
      begin find_buffer ← 0; return; end
⟨Procedures for data manipulation 20⟩ ≡
function find_buffer: pointer;
  label exit;   {enable a quick return}
  var i: 1 . . max_word_length; {index into buffer}
    p: pointer;   {the current word position}
    q: pointer;   {the next word position}
    c: 1 . . 26;   {current letter code}
    ⟨Other local variables of find_buffer 26⟩
  begin $i \leftarrow 1$; $p \leftarrow buffer[1]$;
  while $i < word\_length$ do
    begin $incr(i)$; $c \leftarrow buffer[i]$;
    ⟨Advance $p$ to its child number $c$ 21⟩;
    end;
  $find\_buffer \leftarrow p$;
exit: end;
```

See also section 37.
This code is used in section 3.

21. ⟨Advance p to its child number c 21⟩ ≡

```
  if $link[p] = 0$ then ⟨Insert the firstborn child of $p$
      and move to it, or abort_find 27⟩
  else begin $q \leftarrow link[p] + c$;
    if $ch[q] \neq c$ then
      begin if $ch[q] \neq empty\_slot$ then
        ⟨Move $p$'s family to a place where child $c$
          will fit, or abort_find 29⟩;
      ⟨Insert child $c$ into $p$'s family 28⟩;
      end;
    $p \leftarrow q$;
    end
```

This code is used in section 20.

22. Each "family" in the trie has a header location $h = link[p]$ such that child c is in location $h + c$. We want these values to be spread out in the trie, so that families don't often interfere with each other. Furthermore we will need to have

$$26 < h \leq trie\_size - 26$$

if the search algorithm is going to work properly.

One of the main tasks of the insertion algorithm is to find a place for a new header. The theory of hashing tells us that it is advantageous to put the nth header near the location $x_n = \alpha n \bmod t$, where $t = trie\_size - 52$ and where α is an integer relatively prime to t such that α/t is approximately equal to $(\sqrt{5} - 1)/2 \approx .61803$. [These locations x_n are about as "spread out" as you can get; see *Sorting and Searching*, pp. 510–511.]

```
  define $alpha = 20219$   {$\approx .61803 trie\_size$}
```

⟨Global variables 4⟩ +≡
x: *pointer*; {$\alpha n \bmod (trie\_size - 52)$}

23. ⟨Set initial values 12⟩ +≡
```
  $x \leftarrow 0$;
```

24. We will give up trying to find a vacancy if 1000 trials have been made without success. This will happen only if the table is quite full, at which time the most common words will probably already appear in the dictionary.

```
  define $tolerance = 1000$
```

⟨Get set for computing header locations 24⟩ ≡

```
  if $x < trie\_size - 52 - alpha$ then $x \leftarrow x + alpha$
  else $x \leftarrow x + alpha - trie\_size + 52$;
  $h \leftarrow x + 27$;  {now $26 < h \leq trie\_size - 26$}
  if $h \leq trie\_size - 26 - tolerance$ then
    $last\_h \leftarrow h + tolerance$
  else $last\_h \leftarrow h + tolerance - trie\_size + 52$;
```

This code is used in sections 27 and 31.

25. ⟨Compute the next trial header location h, or *abort_find* 25⟩ ≡

```
  if $h = last\_h$ then abort_find;
  if $h = trie\_size - 26$ then $h \leftarrow 27$
  else $incr(h)$
```

This code is used in sections 27 and 31.

26. ⟨Other local variables of *find_buffer* 26⟩ ≡
h: *pointer*; {trial header location}
last_h: *integer*; {the final one to try}

See also section 30.
This code is used in section 20.

27. ⟨Insert the firstborn child of p and move to it, or *abort_find* 27⟩ ≡

```
  begin ⟨Get set for computing header locations 24⟩;
  repeat ⟨Compute the next trial header location $h$,
          or abort_find 25⟩;
  until $(ch[h] = empty\_slot) \wedge$
      $(ch[h + c] = empty\_slot)$;
  $link[p] \leftarrow h$; $link[h] \leftarrow p$; $p \leftarrow h + c$;
  $ch[h] \leftarrow header$; $ch[p] \leftarrow c$;
  $sibling[h] \leftarrow p$; $sibling[p] \leftarrow h$; $count[p] \leftarrow 0$;
  $link[p] \leftarrow 0$;
  end
```

This code is used in section 21.

28. The decreasing order of *sibling* pointers is preserved here. We assume that $q = link[p] + c$.

⟨Insert child c into p's family 28⟩ ≡

```
  begin $h \leftarrow link[p]$;
  while $sibling[h] > q$ do $h \leftarrow sibling[h]$;
  $sibling[q] \leftarrow sibling[h]$; $sibling[h] \leftarrow q$;
  $ch[q] \leftarrow c$; $count[q] \leftarrow 0$; $link[q] \leftarrow 0$;
  end
```

This code is used in section 21.

29. There's one complicated case, which we have left for last. Fortunately this step doesn't need to be done very often in practice, and the families that need to be moved are generally small.

```
⟨Move p's family to a place where child c will fit,
        or abort_find 29⟩ ≡
    begin ⟨Find a suitable place h to move,
            or abort_find 31⟩;
    q ← h + c; r ← link[p]; delta ← h − r;
    repeat sibling[r + delta] ← sibling[r] + delta;
      ch[r + delta] ← ch[r]; ch[r] ← empty_slot;
      count[r + delta] ← count[r];
      link[r + delta] ← link[r];
      if link[r] ≠ 0 then link[link[r]] ← r + delta;
      r ← sibling[r];
    until ch[r] = empty_slot;
    end
```

This code is used in section 21.

30. ⟨Other local variables of *find_buffer* 26⟩ +≡

```
r: pointer;   {family member to be moved}
delta: integer;   {amount of motion}
slot_found: boolean;   {have we found a new homestead?}
```

31. ⟨Find a suitable place *h* to move, or *abort_find* 31⟩ ≡

```
  slot_found ← false;
  ⟨Get set for computing header locations 24⟩;
  repeat ⟨Compute the next trial header location h,
             or abort_find 25⟩;
    if ch[h + c] = empty_slot then
      begin r ← link[p]; delta ← h − r;
      while (ch[r + delta] = empty_slot) ∧
            (sibling[r] ≠ link[p]) do r ← sibling[r];
      if ch[r + delta] = empty_slot then
        slot_found ← true;
      end;
  until slot_found
```

This code is used in section 29.

32. The frequency counts. It is, of course, a simple matter to combine dictionary lookup with the *get_word* routine, so that all the word frequencies are counted. We may have to drop a few words in extreme cases (when the dictionary is full or the maximum count has been reached).

```
  define max_count = 32767
      {counts won't go higher than this}
```

⟨Global variables 4⟩ +≡

```
count: array [pointer] of 0 . . max_count;
word_missed: boolean;   {did the dictionary get too full?}
p: pointer;   {location of the current word}
```

33. ⟨Set initial values 12⟩ +≡

```
  word_missed ← false;
```

34. ⟨Input the text, maintaining a dictionary with frequency counts 34⟩ ≡

```
  get_word;
  while word_length ≠ 0 do
    begin p ← find_buffer;
    if p = 0 then word_missed ← true
    else if count[p] < max_count then incr(count[p]);
    get_word;
    end
```

This code is used in section 8.

35. While we have the dictionary structure in mind, let's write a routine that prints the word corresponding to a given pointer, together with the corresponding frequency count.

For obvious reasons, we put the word into the buffer backwards during this process.

⟨Procedures for input and output 9⟩ +≡

```
procedure print_word(p: pointer);
  var q: pointer;   {runs through ancestors of p}
    i: 1 . . max_word_length;   {index into buffer}
  begin word_length ← 0; q ← p; write('␣');
  repeat incr(word_length);
    buffer[word_length] ← ch[q];
    move_to_prefix(q);
  until q = 0;
  for i ← word_length downto 1 do
    write(lowercase[buffer[i]]);
  if count[p] < max_count then
    write_ln('␣', count[p] : 1)
  else write_ln('␣', max_count : 1, '␣or␣more');
  end;
```

36. Sorting a trie. Almost all of the frequency counts will be small, in typical situations, so we needn't use a general-purpose sorting method. It suffices to keep a few linked lists for the words with small frequencies, with one other list to hold everything else.

```
  define large_count = 200   {all counts less than this
                 will appear in separate lists}
```

⟨Global variables 4⟩ +≡

```
sorted: array [1 . . large_count] of pointer;   {list heads}
total_words: integer;   {the number of words sorted}
```

37. If we walk through the trie in reverse alphabetical order, it is a simple matter to change the sibling links so that the words of frequency *f* are pointed to by *sorted*[*f*], *sibling*[*sorted*[*f*]], . . . in alphabetical order. When *f* = *large_count*, the words must also be linked in decreasing order of their *count* fields.

The restructuring operations are slightly subtle here, because we are modifying the *sibling* pointers while traversing the trie.

```
⟨Procedures for data manipulation 20⟩ +≡
procedure trie_sort;
  var k: 1 . . large_count;   {index to sorted}
    p: pointer;   {current position in the trie}
    f: 0 . . max_count;   {current frequency count}
    q, r: pointer;   {list manipulation variables}
  begin total_words ← 0;
  for k ← 1 to large_count do sorted[k] ← 0;
  p ← sibling[0]; move_to_last_suffix(p);
  repeat f ← count[p]; q ← sibling[p];
    if f ≠ 0 then ⟨Link p into the list sorted[f] 38⟩;
    if ch[q] ≠ header then
      begin p ← q; move_to_last_suffix(p);
      end
    else p ← link[q];   {move to prefix}
  until p = 0;
  end;
```

38. Here we use the fact that *count*[0] = 0.

```
⟨Link p into the list sorted[f] 38⟩ ≡
  begin incr(total_words);
  if f < large_count then   {easy case}
    begin sibling[p] ← sorted[f]; sorted[f] ← p;
    end
  else begin r ← sorted[large_count];
    if count[p] ≥ count[r] then
      begin sibling[p] ← r; sorted[large_count] ← p;
      end
    else begin while count[p] < count[sibling[r]] do
        r ← sibling[r];
      sibling[p] ← sibling[r]; sibling[r] ← p;
      end;
    end;
  end
```

This code is used in section 37.

39. ⟨Sort the dictionary by frequency 39⟩ ≡
trie_sort

This code is used in section 8.

40. After *trie_sort* has done its thing, the linked lists *sorted*[*large_count*], . . . , *sorted*[1] collectively contain all the words of the input file, in decreasing order of frequency. Words of equal frequency appear in alphabetic order. These lists are linked by means of the *sibling* array.

Therefore the following procedure will print the first *k* words, as required in Bentley's problem.

```
⟨Procedures for input and output 9⟩ +≡
procedure print_common(k : integer);
  label exit;   {enable a quick return}
  var f: 1 . . large_count + 1;   {current frequency}
    p: pointer;   {current or next word}
begin f ← large_count + 1; p ← 0;
repeat while p = 0 do
      begin if f = 1 then return;
      decr(f); p ← sorted[f];
      end;
    print_word(p); decr(k); p ← sibling[p];
  until k = 0;
exit: end;
```

41. The endgame. We have recorded *total_words* different words. Furthermore the variables *word_missed* and *word_truncated* tell whether or not any storage limitations were exceeded. So the remaining task is simple:

```
⟨Output the results 41⟩ ≡
  if total_words = 0 then write_ln('There␣are',
        '␣no␣words␣in␣the␣input!')
  else begin if total_words < max_words_to_print then
          {we will print all words}
      write_ln('Words␣of␣the␣input␣file,',
        '␣ordered␣by␣frequency:')
    else if max_words_to_print = 1 then
        write('The␣most␣common␣word',
          '␣and␣its␣frequency:')
      else write_ln('The␣', max_words_to_print : 1,
          '␣most␣common␣words,',
          '␣and␣their␣frequencies:');
    print_common(max_words_to_print);
    if word_truncated then
      write_ln('(At␣least␣one␣word␣had␣to␣be',
        '␣shortened␣to␣', max_word_length : 1,
        '␣letters.)');
    if word_missed then
    write_ln('(Some␣input␣data␣was␣skipped,',
        '␣due␣to␣memory␣limitations.)');
    end
```

This code is used in section 8.

42. Index. Here is a list of all uses of all identifiers, underlined at the point of definition.

A Review

My dictionary defines criticism as "the art of evaluating or analyzing with knowledge and propriety, especially works of art or literature." Knuth's program deserves criticism on two counts. He was the one, after all, who put forth the analogy of programming as literature, so what is more deserved than a little criticism? This program also merits criticism by its intrinsic interest; although Knuth set out only to display WEB, *he has produced a program that is fascinating in its own right. Doug McIlroy of Bell Labs was kind enough to provide this review.—J.B.*

I found Don Knuth's program convincing as a demonstration of WEB and fascinating for its data structure, but I disagree with it on engineering grounds. The problem is to print the *K* most common words in an input file (and the number of their occurrences) in decreasing frequency. Knuth's solution is to tally in an associative data structure each word as it is read from the file. The data structure is a trie, with 26-way (for technical reasons actually 27-way) fanout at each letter. To avoid wasting space all the (sparse) 26-element arrays are cleverly interleaved in one common arena, with hashing used to assign homes. Homes may move underfoot as new words cause old arrays to collide. The final sorting is done by distributing counts less than 200 into buckets and insertion-sorting larger counts into a list.

The presentation is engaging and clear. In WEB one deliberately writes a paper, not just comments, along with code. This of course helps readers. I am sure that it also helps writers: reflecting upon design choices sufficiently to make them explainable must help clarify and refine one's thinking. Moreover, because an explanation in WEB is intimately combined with the hard reality of implementation, it is qualitatively different from, and far more useful than, an ordinary "specification" or "design" document. It can't gloss over the tough places.

Perhaps the greatest strength of WEB is that it allows almost any sensible order of presentation. Even if you did not intend to include any documentation, and even if you had an ordinary cross-referencer at your disposal, it would make sense to program in WEB simply to circumvent the unnatural order forced by the syntax of Pascal. Knuth's exercise amply demonstrates the virtue of doing so.

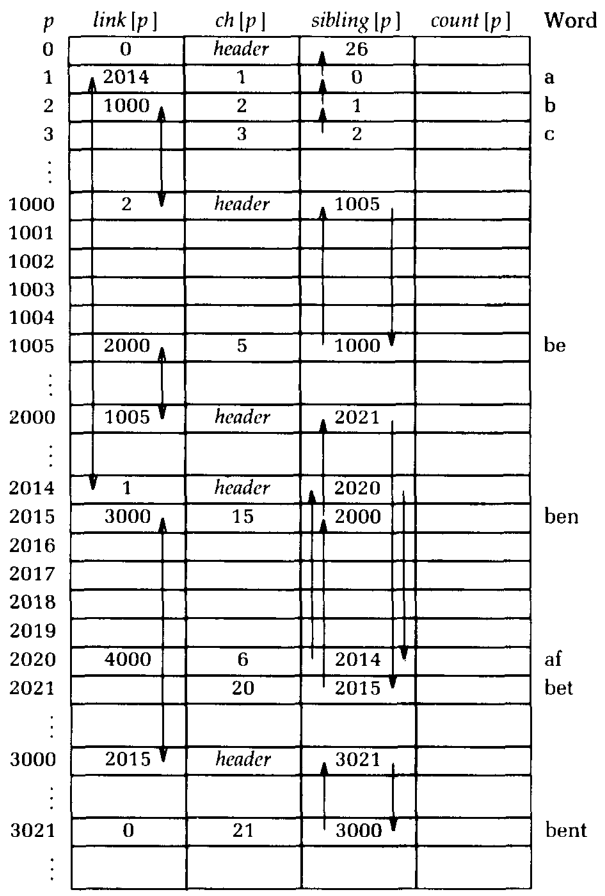
Mere use of WEB, though, won't assure the best organization. In the present instance the central idea of interleaving sparsely populated arrays is not mentioned until far into the paper. Upon first reading that, with hash tries, "some space is sacrificed," I snorted to myself that some understatement had been made of the wastage. Only much later was I disabused of my misunderstanding. I suspect that this oversight in presentation was a result of documenting on the fly. With this sole exception, the paper eloquently attests that the discipline of simultaneously writing and describing a program pays off handsomely.

A few matters of style: First, the program is studded with instances of an obviously important constant, which variously take the guise of "26", "27", and "52." Though it is unobjectionable to have such a familiar number occur undocumented in a program about words, it is impossible to predict all its disguised forms. Just how might one confidently change it to handle, say, Norwegian or, more mundanely, alphanumeric "words"? A more obscure example is afforded by a constant *alpha*, calculated as the golden ratio times another constant, *trie_size*. Signaled only by a comment deep inside the program, this relationship would surely be missed in any quick attempt to change the table size. WEB, unlike Pascal, admits dependent constants. They should have been used.

Second, small assignment statements are grouped several to the line with no particularly clear rationale. This convention saves space; but the groupings impose a false and distracting phrasing, like "poetry" produced by randomly breaking prose into lines.

Third, a picture would help the verbal explanation of the complex data structure. Indeed, pictures in listings are another strong reason to combine programming with typesetting; see Figure 1.

Like any other scheme of commentary, WEB can't guarantee that the documentation agrees perfectly



p	*link*[*p*]	*ch*[*p*]	*sibling*[*p*]	*count*[*p*]	Word
0	0	*header*	26		
1	2014	1	0		a
2	1000	2	1		b
3		3	2		c
⋮					
1000	2	*header*	1005		
1001					
1002					
1003					
1004					
1005	2000	5	1000		be
⋮					
2000	1005	*header*	2021		
⋮					
2014	1	*header*	2020		
2015	3000	15	2000		ben
2016					
2017					
2018					
2019					
2020	4000	6	2014		af
2021		20	2015		bet
⋮					
3000	2015	*header*	3021		
⋮					
3021	0	21	3000		bent
⋮					

FIGURE 1. A Picture to Accompany Knuth's §18[1]

with the program. For example, the procedure *read_int* expects an integer possibly preceded by blanks, although a literal reading of the description would require the integer to appear at the exact beginning of the text. This, however, was the only such case I noticed. The accuracy of the documentation compliments both the author and his method.

The WEB description led me pleasantly through the program and contained excellent cross references to help me navigate in my own ways. It provided important guideposts to the logical structure of the program, which are utterly unavailable in straight Pascal. For no matter how conscientiously commented or stylishly expressed programs in Pascal may be, the language forces an organization convenient for translating, not for writing or reading. In the WEB presentation variables were introduced

[1] I typeset this picture in an hour or two of my time using the PIC language; Knuth could have used TeX features to include such a figure in his WEB program. The figure contains a slight mistake for consistency with an off-by-one slip in §18 of Knuth's program: he assumed that "N" was the 15th letter of the alphabet, while it is really the 14th. Knuth insisted that we publish his program as he wrote it, warts and all, since it shows that WEB does not totally eliminate mistakes.—J.B.

where needed, rather than where permitted, and procedures were presented top-down as well as bottom-up according to pedagogical convenience rather than syntactic convention. I was able to skim the dull parts and concentrate on the significant ones, learning painlessly about a new data structure. Although I could have learned about hash tries without the program, it was truly helpful to have it there, if only to taste the practical complexity of the idea. Along the way I even learned some math: the equidistribution (mod 1) of multiples of the golden mean. Don Knuth's ideas and practice mix into a whole greater than the parts.

Although WEB circumvents Pascal's rigid rules of order, it makes no attempt to remedy other defects of Pascal (and rightly so, for the idea of WEB transcends the particulars of one language). Knuth tiptoes around the tarpits of Pascal I/O—as I do myself. To avoid multiple input files, he expects a numerical parameter to be tacked on to the beginning of otherwise pure text. Besides violating the spirit of Bentley's specification, where the file was clearly distinguished from the parameter, this clumsy convention could not conceivably happen in any real data. Worse still, how is the parameter, which Knuth chose to make optional, to be distinguished from the text proper? Finally, by overtaxing Pascal's higher-level I/O capabilities, the convention compels Knuth to write a special, but utterly mundane, read routine.

Knuth's purpose was to illustrate WEB. Nevertheless, it is instructive to consider the program at face value as a solution to a problem. A first engineering question to ask is: how often is one likely to have to do this exact task? Not at all often, I contend. It is plausible, though, that similar, but not identical, problems might arise. A wise engineering solution would produce—or better, exploit—reusable parts.

If the application were so big as to need the efficiency of a sophisticated solution, the question of size should be addressed before plunging in. Bentley's original statement suggested middling size input, perhaps 10,000 words. But a major piece of engineering built for the ages, as Knuth's program is, should have a large factor of safety. Would it, for example, work on the Bible? A quick check of a concordance reveals that the Bible contains some 15,000 distinct words, with typically 3 unshared letters each (supposing a trie solution, which squeezes out common prefixes). At 4 integers per trie node, that makes 180,000 machine integers. Allowing for gaps in the hash trie, we may reasonably round up to half a million. Knuth provided for 128K integers; the prospects for scaling the trie store are not impossible.

Still, unless the program were run in a multi-megabyte memory, it would likely have to ignore some late-arriving words, and not necessarily the least frequent ones, either: the word "Jesus" doesn't appear until three-fourths of the way through the Bible.

Very few people can obtain the virtuoso services of Knuth (or afford the equivalent person-weeks of lesser personnel) to attack nonce problems such as Bentley's from the ground up. But old Unix® hands know instinctively how to solve this one in a jiffy. (So do users of SNOBOL and other programmers who have associative tables[2] readily at hand—for almost any small problem, there's some language that makes it a snap.) The following shell script[3] was written on the spot and worked on the first try. It took 30 seconds to handle a 10,000-word file on a VAX-11/750®.

```
(1) tr -cs A-Za-z'
    ' |
(2) tr A-Z a-z |
(3) sort |
(4) uniq -c |
(5) sort -rn |
(6) sed ${1}q
```

If you are not a Unix adept, you may need a little explanation, but not much, to understand this pipeline of processes. The plan is easy:

1. Make one-word lines by transliterating the complement (−c) of the alphabet into newlines (note the quoted newline), and squeezing out (−s) multiple newlines.
2. Transliterate upper case to lower case.
3. Sort to bring identical words together.
4. Replace each run of duplicate words with a single representative and include a count (−c).
5. Sort in reverse (−r) numeric (−n) order.
6. Pass through a stream editor; quit (q) after printing the number of lines designated by the script's first parameter (${1}).

The utilities employed in this trivial solution are Unix staples. They grew up over the years as people noticed useful steps that tended to recur in real problems. Every one was written first for a particular need, but untangled from the specific application.

Unix is a trademark of AT&T Bell Laboratories. VAX is a trademark of Digital Equipment Corporation.

[2] *The June 1985 column describes associative arrays as they are implemented in the AWK language; page 572 contains a 6-line AWK program to count how many times each word occurs in a file.—J.B.*

[3] This shell script is similar to a prototype spelling checker described in the May 1985 column. (That column also described a production-quality spelling checker designed and implemented by one-and-the-same Doug McIlroy.) This shell script runs on a descendant of the "seventh edition" UNIX system; trivial syntactic changes would adapt it for System V.—J.B.

With time they accreted a few optional parameters to handle variant, but closely related, tasks. *Sort*, for example, did not at first admit reverse or numeric ordering, but these options were eventually identified as worth adding.

As an aside on programming methodology, let us compare the two approaches. At a sufficiently abstract level both may be described in the same terms: partition the words of a document into equivalence classes by spelling and extract certain information about the classes. Of two familiar strategies for constructing equivalence classes, tabulation and sorting, Knuth used the former, and I chose the latter. In fact, the choice seems to be made preconsciously by most people. Everybody has an instinctive idea how to solve this problem, but the instinct is very much a product of culture: in a small poll of programming luminaries, all (and only) the people with Unix experience suggested sorting as a quick-and-easy technique.

The tabulation method, which gets right to the equivalence classes, deals more directly with the data of interest than does the sorting method, which keeps the members much longer. The sorting method, being more concerned with process than with data, is less natural and, in this instance, potentially less efficient. Yet in many practical circumstances it is a clear winner. The causes are not far to seek: we have succeeded in capturing generic processes in a directly usable way far better than we have data structures. One may hope that the new crop of more data-centered languages will narrow the gap. But for now, good old process-centered thinking still has a place in the sun.

Program transformations between the two approaches are interesting to contemplate, but only one direction seems reasonable: sorting to tabulation. The reverse transformation is harder because the elaboration of the tabulation method obscures the basic pattern. In the context of standard software tools, sorting is the more primitive, less irrevocably committed method from which piecewise refinements more easily flow.

To return to Knuth's paper: everything there—even input conversion and sorting—is programmed monolithically and from scratch. In particular the isolation of words, the handling of punctuation, and the treatment of case distinctions are built in. Even if data-filtering programs for these exact purposes were not at hand, these operations would well be implemented separately: for separation of concerns, for easier development, for piecewise debugging, and for potential reuse. The small gain in efficiency from integrating them is not likely to warrant the resulting loss of flexibility. And the worst possible eventuality—being forced to combine programs—is not severe.

The simple pipeline given above will suffice to get answers right now, not next week or next month. It could well be enough to finish the job. But even for a production project, say for the Library of Congress, it would make a handsome down payment, useful for testing the value of the answers and for smoking out follow-on questions.

If we do have to get fancier, what should we do next? We first notice that all the time goes into sorting. It might make sense to look into the possibility of modifying the sort utility to cast out duplicates as it goes (Unix *sort* already can) and to keep counts. A quick experiment shows that this would throw away 85 percent of a 10,000-word document, even more of a larger file. The second sort would become trivial. Perhaps half the time of the first would be saved, too. Thus the idea promises an easy 4-to-1 speedup overall—provided the sort routine is easy to modify. If it isn't, the next best thing to try is to program the tallying using some kind of associative memory—just as Knuth did. Hash tables come to mind as easy to get right. So do simple tries (with list fanout to save space, except perhaps at the first couple of levels where the density of fanout may justify arrays). And now that Knuth has provided us with the idea and the code, we would also consider hash tries. It remains sensible, though, to use utilities for case transliteration and for the final sort by count. With only 15 percent as much stuff to sort (even less on big jobs) and only one sort instead of two, we can expect an order of magnitude speedup, probably enough relief to dispel further worries about sorting.

Knuth has shown us here how to program intelligibly, but not wisely. I buy the discipline. I do not buy the result. He has fashioned a sort of industrial-strength Faberge egg—intricate, wonderfully worked, refined beyond all ordinary desires, a museum piece from the start.

Principles—*J.B.*

Literate Programming. Last month's column sketched the mechanics of literate programming. This month's column provides a large example—by far the most substantial pearl described in detail in this column. I'm impressed by Knuth's methods and his results; I hope that this small sample has convinced you to explore the Further Reading to see his methods applied to real software.

A New Data Structure. I asked Knuth to provide a textbook solution to a textbook problem; he went far beyond that request by inventing, implementing and lucidly describing a fascinating new data structure—the hash trie.

Further Reading

"Literate Programming" is the title and the topic of Knuth's article in the May 1984 *Computer Journal* (Volume 27, Number 2, pages 97–111). It introduces a literate style of programming with the example of printing the first 1,000 prime numbers. Complete documentation of "The WEB System of Structured Documentation" is available as Stanford Computer Science technical report 980 (September 1983, 206 pages); it contains the WEB source code for TANGLE and WEAVE.

The small programs in this column and last month's hint at the benefits of literate programming; its full power can only be appreciated when you see it applied to substantial programs. Two large WEB programs appear in Knuth's five-volume *Computers and Typesetting*, just published by Addison-Wesley. The source code for TEX is Volume B, entitled *TEX: The Program* (xvi + 594 pages). Volume D is *METAFONT: The Program* (xvi + 560 pages). Volume A is *The TEXbook*, Volume C is *The METAFONTbook*, and Volume E is *Computer Modern Typefaces*.

Criticism of Programs. The role of any critic is to give the reader insight, and McIlroy does that splendidly. He first looks inside this gem, then sets it against a background to help us see it in context. He admires the execution of the solution, but faults the problem on engineering grounds. (That is, of course, my responsibility as problem assigner; Knuth solved the problem he was given on grounds that are important to most engineers—the paychecks provided by their problem assigners.) Book reviews tell you what is in the book; good reviews go beyond that to give you insight into the environment that molded the work. As Knuth has set high standards for future authors of programming literature, McIlroy has shown us how to analyze those works.

Problems

1. Design and implement programs for finding the K most common words. Characterize the trade-offs among code length, problem definition, resource utilization (time and space), and implementation language and system.
2. The problem of the K most common words can be altered in many ways. How do solutions to the original problem handle these new twists? Instead of finding the K most common words, suppose you want to find the single most frequent word, the frequency of all words in decreasing order, or the K least frequent words. Instead of dealing with words, suppose you wish to study the frequency of letters, letter pairs, letter triples, word pairs, or sentences.
3. Quantify the time and space required by Knuth's version of Liang's hash tries; use either experimental or mathematical tools (see Problem 4). Knuth's *dynamic* implementation allows insertions into hash tries; how would you use the data structure in a *static* problem in which the entire set of words was known before any lookups (consider representing an English dictionary)?
4. Both Knuth and McIlroy made assumptions about the distribution of words in English documents. For instance, Knuth assumed that most frequent words tend to occur near the front of the document, and McIlroy pointed out that a few frequent words may not appear until relatively late.
   a. Run experiments to test the various assumptions. For instance, does reducing the memory size of Knuth's program cause it to miss any frequent words?
   b. Gather data on the distribution of words in English documents to help one answer questions like this; can you summarize that statistical data in a probabilistic model of English text?
5. A map of the United States has been split into 25,000 tiny line segments, ordered from north to south, for drawing on a one-way plotting device. Design a program to reconnect the segments into reasonably long connected chains for use on a pen plotter.

Solutions to May's Problems

4. J. S. Vitter's "Faster Methods for Random Sampling" in the July 1984 *Communications* shows how to generate M sorted random integers in $O(M)$ expected time and constant space; those resource bounds are within a constant factor of optimal.
5. WEB provides two kinds of macros: **define** for short strings and the ⟨Do this now⟩ notation for longer pieces of code. Howard Trickey writes that "this facility is qualitatively better than the C preprocessor macros, because the syntax for naming and defining C macros is too awkward for use-once code fragments. Even in languages with the ability to declare procedures 'inline', I think most people would resist using procedures as prolifically as WEB users use modules. Somehow the ability to describe modules with sentences instead of having to make up a name

helps me a lot in using lots of modules. Also, WEB macros can be used before they are defined, and they can be defined in pieces (e.g., ⟨Global variables⟩), and that isn't allowed in any macro language I know."

6. Howard Trickey observes that "the fact that TANGLE produces unreadable code can make it hard to use debuggers and other source-level software. Knuth's rejoinder is that if people like WEB enough they will modify such software to work with the WEB source. In practice, I never had much difficulty debugging TANGLE output run through a filter to break lines at statement boundaries."

7. [D. E. Knuth] To determine whether two input sequences define the same set of integers, insert each into an ordered hash table. The two ordered hash tables are equal if and only if the two sets are equal.

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Ave., Murray Hill, NJ 07974.