programming pearls

by Jon Bentley

A SPELLING CHECKER

Spelling mistakes irritate readers. And for most writers, checking spelling is a boring and error-prone job. Fortunately, the problem is ideally suited for computers: dull, repetitive work that requires fast reading and a good memory.

In this column we'll study the design of the Unix[1] spelling checker `spell`. It's a beautiful and useful program, with a history rich in important lessons about program development.

A Simple Program

Steve Johnson wrote the first version of `spell` in an afternoon in 1975. His straightforward approach is shown in Figure 1: isolate the words in a document, sort them, and then compare the sorted list with the dictionary. The output is a list of all words in the document that aren't in the dictionary.

Kernighan and Plauger reconstruct Johnson's program on page 133 of their *Software Tools in Pascal*. Their pipeline of programs is paraphrased in Program 1. The vertical bar connects the output of one program with the input to the next program. The input to the first program is `filename`, and the output of the last program is the list of (potentially) misspelled words.

The first program in the pipeline, `prepare`, deals with the fact that many computerized documents contain formatting commands. To print a word in a **boldface** font, for instance, one might type `@b(boldface)` or `\fBboldface\fR`. A spelling checker must ignore such commands; the poor user who wades through misspellings like *b* and *fboldface* is too exhausted to notice real errors. `prepare` copies its input to its output, with formatting commands removed.

`translit` transliterates its input to its output, substituting certain characters. Its first invocation in the pipeline changes upper case letters to lower case. The next invocation removes all nonalphabetic characters by mapping them into the newline character, @n. The result is a file that contains the words of the document in the order they appear, with at most one word per line.

The next program `sorts` the words into alphabetic order, and `unique` removes multiple occurrences. The result is a sorted list of the distinct words in the document. `common`, with the cryptic `-2` option, uses a standard merge algorithm to report all lines in its (sorted) input that are not in the (sorted) named file, and the output is the desired list of spelling errors.

In an afternoon Johnson assembled five existing programs and an online dictionary to make a new tool. His program was far from perfect, but it demonstrated the feasibility of a spelling checker and gained a loyal following of users. Changes to the program over the next several months were minor modifications to this structure; a complete redesign would wait for several years.

The Design Space

Before moving on to the next version of `spell`, let's survey the options available to the designer. We'll first examine the program's external appearance (the problem specification seen by the user), and then turn to its internal structure.

When hunting for bogus words, it can be helpful to know their source: bad spelling or bad typing. Poor spellers often write *oftun*, good spellers occasionally write *occaisionally*, and we all make *typnig* mistakes. The design of a program may reflect the errors its users make most frequently.

There are two mistakes a spelling program can make, and both compromise its usefulness. Failing to flag an illegal word is an obvious problem. And if a program reports too many valid words as errors, the user may be unwilling to search through the mud to find the pay dirt. All spelling programs make mistakes—whether to report too few or too many suspicious words is a design choice.

[1] Unix is a trademark of AT&T Bell Laboratories.

FIGURE 1. A Simple Spelling Checker

```
prepare filename |     # remove formatting commands
 translit A-Z a-z |    # map upper to lower case
  translit ^a-z @n |   # remove punctuation
   sort |              # put words in alphabetical order
    unique |           # remove duplicate words
     common -2 dict    # report words not in dictionary
```

PROGRAM 1. Code for the Simple Checker

The program should check words for spelling, but what exactly is a word? Johnson's checker recognized the importance of removing formatting commands. It ignored distinctions between upper and lower case, and therefore correctly looked up *The* as *the*. Some case problems are more subtle: *DEC* is the name of a computer company and *Dec* is a month, but *dec* is an error (unless, of course, it is an abbreviation for Decimal). Other subtleties about words include numbers, hyphens, and apostrophes (consider *VAX-11/780*[2] and *his's*). A prototype can be sloppy about these fine points, but a production program should be more careful.

And what about the dictionary itself? A program should recognize some words not in a regular dictionary, such as *IBM* and *VLSI* (but not *vlsi*). But bigger isn't always better—the dictionary may know that a *cere* is near a bird's bill, but in one of my files, it is more likely a misspelling of *care*. An affix is a prefix like *pre-* or a suffix like *-ly*; most dictionaries leave affix analysis to the reader. Although a spelling checker may in good conscience report *antidisestablishmentarianism* as a mistake, I would be miffed to wade through a long list of misspellings like *cats* and *replay* when *cat* and *play* were in the dictionary. Johnson modified his program to handle the common *-s* and *-ed* endings, but a production spelling checker must do a more thorough affix analysis.

Perhaps the most debated problem in specifying a spelling program is what it should do when it finds an error. Johnson's simple checker produces a list of misspelled words. At the other extreme, interactive spelling *correctors* show the user a misspelled word in its context and ask whether to leave the word unchanged, change this occurrence to a suggested word, change this and all future occurrences to the suggested word, edit this word and change all future occurrences to the edited version, and so on and so on.

Some people say they couldn't live without a fancy spelling corrector—poor spellers seem to find its advice especially valuable. My personal taste, as a fairly good speller, runs toward a simple checker. I now routinely use `spell` on all documents; I rarely used the fancy corrector on a previous system because it took several minutes for me to relearn its command language each time I tried it. Its advice was often more irritating (or amusing) than helpful—it once suggested that J. W. Tukey's last name be corrected to "Turkey." Additionally, a corrector is usually more difficult to build and to maintain than a checker.

Turning from specification to implementation, there are two canonical structures for spelling programs. Johnson's batch checker used the structure on the left of Figure 2; the online program on the right looks up

[2] VAX is a Trademark of Digital Equipment Corporation.

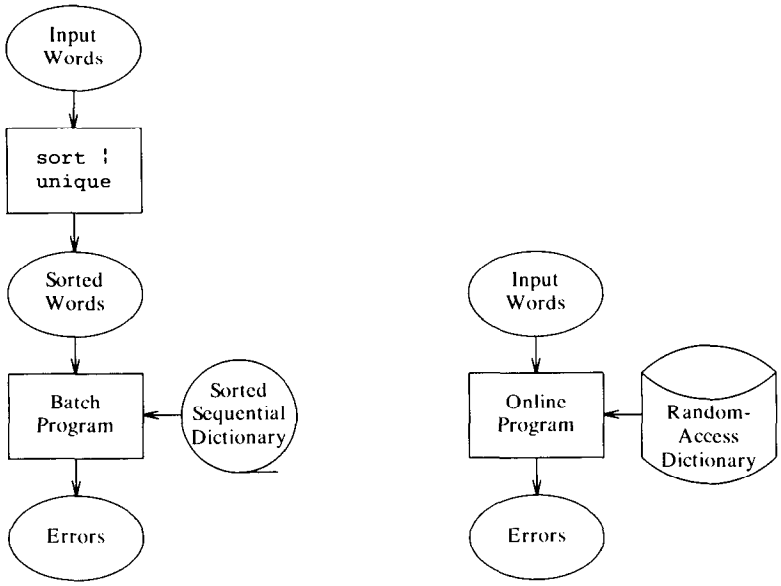


FIGURE 2. Two Structures for a Spelling Program

each word as it is encountered. A spelling checker may use either structure, but an interactive corrector is usually restricted to be online. Similarly, a random-access dictionary may be used by either structure, but a sequential dictionary is only suitable for a batch program.

There is a tradeoff between the sophistication of affix analysis and dictionary size. A simple program that does no affix analysis requires a huge dictionary, including *test, tests, tested, tester, testing, retest, pretest,* etc. Sophisticated affix schemes store the stem *test* and an encoding of its valid affixes.

There are many possible ways to store the dictionary itself. If it fits in main memory, it could be represented by a hash table, a binary search tree, or perhaps a "trie" that exploits the fact that a word is a sequence of characters. If the dictionary must reside on disk, then a B-tree or disk hash table is more suitable.

The best implementation of the dictionary depends on several factors. A simple data structure will ease development and maintenance, but a back-of-the-envelope calculation shows that performance may be crucial in this application. Suppose that a disk-based scheme looks up a word in two disk reads of fifty milliseconds each. The program processes ten words per second, so a document of 4,000 words (the average size of these columns) requires six long minutes. The program we'll soon study does the job in half a minute.[3] A spelling corrector referenced under Further Reading sometimes makes as many as 200 disk accesses to correct a single misspelled word, which translates to ten seconds and an annoying wait in a real-time program.

The outline in Figure 3 summarizes our considerations of the design space. It is intended more as a programmer's sketch than a formal taxonomy. Problem 4 mentions approaches to spelling programs that are outside this space.

REQUIREMENTS — The Customer's View
Typical user
    Source of errors: bad spelling or sloppy typing
    Response to errors: good spellers need only notification of their mistakes, bad spellers appreciate more assistance
Development resources
    How much programmer time is available?
Application resources
    Time and space requirements of the final code

SPECIFICATION — The User's View
Word definition
    Fine points include formatting commands, upper vs. lower case distinction, and embedded numbers and punctuation
The word list
    Explicit words: stored in the list
    Implicit words: present by affix analysis
Response to errors
    Checkers report errors, correctors fix them

IMPLEMENTATION — The Programmer's View
Program structure
    Batch programs sort the words to remove duplicates
    Online programs check each word as it occurs
Word list implementation
    Tradeoffs between affix analysis, dictionary size, and quality of answers
Dictionary specification
    Are words accessed in sorted order or random order?
Dictionary implementation
    Primary memory: hashing, search trees, search tries
    Secondary memory: B-trees, hashing
    Combinations represent common words in primary memory and rarer words on secondary memory

FIGURE 3. The Design Space of Spelling Programs

A Subtle Program

In this section we'll study the `spell` program that Doug McIlroy wrote in 1978. Its user interface is the same as Johnson's: typing `spell filename` produces a list of the misspelled words in the file. The two advantages of this program over Johnson's are a superior word list and reduced run time. I'm an enthusiastic user: the program is simple to use, and it quickly reports all my misspellings and very few words that aren't in error. My dictionary defines a pearl as something "very choice or precious"; this program qualifies.

The first problem McIlroy faced was assembling the word list; to appreciate some of the subtleties of the task, see the sidebar on page 460. He started by intersecting an unabridged dictionary (for validity) with the million-word Brown University corpus (for currency). That was a reasonable beginning, but there was much work left to do.

McIlroy's approach is illustrated in his quest for proper nouns, which are omitted from most dictionaries. First came people: the 1,000 most common last names in a large telephone directory, a list of boys' and girls' names, famous names (like Dijkstra and Nixon), and mythological names from an index to Bulfinch. After observing "misspellings" like *Xerox* and *Texaco*, he added the companies on the Fortune 500 list. Publishing companies are rampant in bibliographies, so they're in. Next came geography: the nations and their capitals, the states and theirs, the hundred largest cities in the United States and the world, and don't forget oceans, planets, and stars.

He also added common names of animals and plants, and terms from chemistry, anatomy, and (for local consumption) computing. But he was careful not to add too much: he kept out valid words that tend to be real-life misspellings (like the geological term *cwm*) and included only one of several alternative spellings (hence *traveling* but not *travelling*).

McIlroy's trick was to examine `spell`'s output on real runs; for some time, it automatically mailed a copy of the output to him. When he spotted a problem, he would apply the broadest possible solution. The result is a fine list of 75,000 words; it includes most of the words I use in my files, yet still finds my spelling errors.

[3] Anecdotal evidence that performance matters: After writing this paragraph I ran McIlroy's `spell` on the two-thousand-word draft, and twenty seconds later the error list reported the words *monagraph, textfile,* and *filename*. I fixed the spelling error, and changed several occurrences of *textfile* to *filename* for consistency. That was high return on a twenty second investment; I probably wouldn't have run the program if it cost three minutes.

`spell`'s affix analysis is both necessary and convenient. It's necessary because there is no such thing as a word list for English; a spelling checker must either guess at the derivation of words like *misrepresented* or report as errors a lot of valid English words. Affix analysis has the convenient side effect of reducing the size of the dictionary.

The goal of affix analysis is to reduce *misrepresented* down to *sent*, stripping off *mis-*, *re-*, *pre-*, and *-ed*.[4] `spell`'s tables contain 40 prefix rules and 30 suffix rules. A "stop list" of 1,300 exceptions halts good but incorrect guesses like reducing *entend* (a misspelling of *intend*) to *en* + *-tend*. This analysis reduces the 75,000 word list to 30,000 words.

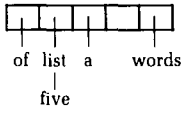
McIlroy's program is the same as Johnson's up to the point of looking up words in the dictionary (the `common` program in the previous pipeline). The new program loops on each word, stripping affixes and looking up the result until it either finds a match or no affixes remain (and the word is declared to be an error). Because affix processing may destroy the sorted order in which the words arrive, the dictionary is accessed in random order.

Back-of-the-envelope analysis showed the importance of keeping the dictionary in main memory. This was particularly hard for McIlroy, whose machine had only a 64-kilobyte address space. The abstract of his paper summarizes his space squeezing: "Stripping prefixes and suffixes reduces the list below one third of its original size, hashing discards 60 percent of the bits that remain, and data compression halves it once again." Thus a list of 75,000 English words (and roughly as many inflected forms) was represented in 26,000 16-bit computer words.

McIlroy used hashing to represent 30,000 English words in 27 bits each (we'll see later why 27 is magic). We'll study a progression of schemes illustrated on the toy word list

a list of five words

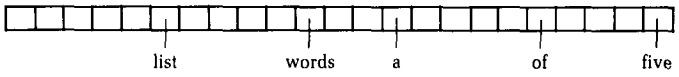
The first hashing method uses an N-element hash table roughly the size of the list and a hash function that maps a string into an integer in the range $1 \ldots N$. The I^{th} entry of the table points to a linked list that contains all strings that hash to I. If null lists are represented by empty cells and the hash function yields $H(a) = 3$, $H(list) = 2$, etc., then a five-element table might look like



To look up the word W we perform a sequential search in the list pointed to by the $H(W)^{th}$ cell.

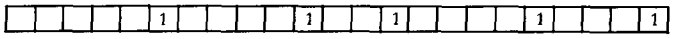
[4] Even though *represent* doesn't mean "to present again" and *present* doesn't mean "sent beforehand", `spell` uses coincidences to reduce dictionary size.

The next scheme uses a much larger table. Choosing $N = 23$ makes it likely that most lists contain just one element. In this example, $H(a) = 14$ and $H(list) = 6$.



The `spell` program uses $N = 2^{27}$ (roughly 134 million), and all but a few of the nonempty lists contain just a single element.

The next step is daring: instead of a linked list of words, McIlroy stores just a single bit in each table entry. This reduces space dramatically, but introduces errors. This picture uses the same hash function as the previous example, and represents zero bits by empty cells.



To look up word W, the program accesses the $H(W)^{th}$ bit in the table. If that bit is zero, then the program correctly reports that word W is not in the table. If the bit is one, then the program assumes W is in the table. Sometimes a bad word just happens to hash to a valid bit, but the probability of such an error is just $30{,}000/2^{27}$, or roughly $1/4{,}000$. On the average, therefore, one out of every 4,000 bad words will sneak by as valid. McIlroy observed that typical rough drafts rarely contain more than 20 errors, so this defect hampers at most one run of `spell` out of every hundred—that's why he chose 27.

Representing the hash table by a string of $N = 2^{27}$ bits would consume over sixteen million bytes. The program therefore represents just the one bits; in the above example,

6 11 14 19 23

The word W is declared to be in the table if $H(W)$ is present. The obvious representation of those values uses 30,000 27-bit words, but McIlroy's machine had only 32,000 16-bit words in its address space. He therefore sorted the list and used a variable-length code to represent the *differences* between successive hash values. Assuming a fictitious starting value of zero, the above list is encoded as

6 5 3 5 4

`spell` represents the differences in an average of 13.6 bits each. That left a few hundred extra words to be used to speed up the sequential search in the compressed list. The result is a 64-kilobyte dictionary that has fast access time and almost never makes mistakes.

We've already considered two aspects of `spell`'s performance: it produces useful output and it fits in a 64-kilobyte address space. It's also fast, I mentioned earlier that it can check a 5,000-word document in 30 seconds of VAX-11/750 CPU time; that translates into

Why Spelling is Hard

The recipe for elephant stew begins, "First, catch an elephant." If your recipe for building a spelling program begins, "First, find a valid word list for English," you may find it easier to prepare a delicious dish of elephant stew. After reading a draft of this column, Vic Vyssotsky wrote the following note, which helped me appreciate the problem.

"Spelling is one of the best examples I've seen of the need for prototyping: build something small, try it, see how useful it is in practice, then modify and extend. As you point out, it would be nearly impossible to guess a priori what features a spelling checker should have in detail in order to be most useful.

"This is related to the fact that we're dealing with English. French, for instance, has an academy to define root words and a more systematic set of derivations. In French it is a great deal easier than in English to determine whether *glotchification* is a word and is correctly spelled. So it would be much easier to build a spelling checker for French than for English. But it would also be much less useful for French, because anybody who writes much in French knows how to spell correctly (and how to determine word boundaries, and how to decide whether a particular neologism is plausible).

"Language has the challenging property of changing as we speak: *fribble* is a word, and it's in Webster's, but my daughters would be astonished at the dictionary's definition; *glotch* is a word, and everybody knows what it means[5], but it's not in Webster's; *glout* is clearly not a word, Webster's notwithstanding. These days I encounter *cwm* more often than *cum* (because climbing has become a popular sport, and children are no longer forced to learn Latin), but 30 years ago it was the other way around. And what's the correct spelling of *thru*? A generation ago, grade school teachers knew the answer, even if I didn't, but the California Department of Highways changed it for all of us.

"It seems to me that this malleability of English is the deep fundamental reason why a spelling corrector won't work. My fifth grade English teacher was a spelling corrector, and in your draft column she would surely correct *newline* and *online* as well as *filename*, none of which need correcting. I still remember the firmness of the putdown she administered when I suggested adding *abaft* to her canonical list of all-the-prepositions-in-the-English-language. Unfortunately for correctors, whether human or electronic, the English language (and its spelling) rests on agreement among its users, and not on decisions made by an academy of experts.

"That's what makes a spelling checker such an interesting undertaking, and such a good example of program design issues."

[5] I didn't—a glotch is a large, disorderly aggregate.

just 20 minutes for checking a 400-page book. And if I want to check the spelling of a single word, I type, for instance,

```
spell
necessary
^d
```

and in about four seconds I know that it is a valid word—the small dictionary can be read from disk quickly. The paper cited under Further Reading surveys many spelling programs, but none are in the same performance class as McIlroy's.

Principles

Here's the story in a nutshell. With a good idea, some powerful tools, and a free afternoon, Steve Johnson built a useful six-line spelling checker. Assured that the project was worthy of substantial effort, a few years later Doug McIlroy spent several months engineering a great program. The tale has several morals.

Prototypes. Before you build a fancy program, let potential users experiment with a simple prototype on many real inputs. Johnson used a trivial word list to build a slow checker; users wanted a better word list and faster program, but there was no demand for a corrector. Prototypes can help estimate parameters of the final product: experience with Johnson's program gave McIlroy insight into the typical number of total, distinct, and misspelled words in a document.

Separation of Concerns. A well-built system is divided into independent components, each of which does one thing well. The five different programs in Johnson's pipeline solve five different problems; any one could be enhanced without adversely affecting the others. Affix analysis and dictionary representation are largely independent in McIlroy's program; one needn't learn much about one to work on the other. And for my money, separation of concerns speaks against spelling correctors; errors should be found by a simple checker, fixing them is the job of a text editor, and help about the correct spelling of a word should come from a "suggester" (see Problem 5).

Simplicity. The best design is usually the simplest. Johnson's problem definition is trivial to specify, and his program is just a few lines of code. Even McIlroy's subtle data structure yields simplicity: his single structure does a job that many programs use several data structures to do more slowly.

Software Engineering. Although it didn't require any project management, the `spell` program represents engineering of the first rank. Johnson and McIlroy using the old engineering tools of prototyping, simplicity, sep-

arating concerns, careful problem definition, and back-of-the-envelope calculations. They built with standard components: the design uses off-the-shelf filters for removing formatting commands, sorting, and removing duplicates. When they couldn't use existing software tools, they used proven techniques: McIlroy's word list data structure combines hashing, approximation algorithms, and data compression. The final program is steeped in skillful design decisions: McIlroy traded small chunks of run time, space, accuracy, and problem definition to yield an effective tool.

Problems

1. Gather data on documents and dictionaries such as the distribution of word lengths and the frequency of all possible letters and digrams (letter pairs). For dictionaries, evaluate the compression of simple affix analysis (what percent of words are covered by *-s*, *-ed*, and *-ly*?). For documents, count the number of total, distinct, and misspelled words. What are other useful statistics?
2. Use back-of-the-envelope calculations to evaluate various designs for a spelling checker (for instance, should a fast filter be used to weed out the one hundred most common English words? Is it worth sorting the words in a batch program with a fast online dictionary?). Characterize the run time of the spelling program on your system.
3. Investigate other data structures for representing a random-access dictionary; consider especially structures that don't always give the right answer. Analyze space requirements and run time (both for reading the dictionary from disk and for accessing a word).
4. Investigate spelling checkers that don't use a complete dictionary. Possible approaches include finding near misses (such as the words *programmer* and *programer* in one document) and checking for common letter pairs and triples (see Problem 1).
5. Design a spelling suggester that a bad speller might use with a checker. Given the input *occurrance*, it should suggest that you mean *occurrence*.
6. Programs for checking spelling, playing word games, and making crossword puzzles require different dictionaries. Give words that might be in one of the dictionaries but not in the others. What other dictionaries might be required by various programs?
7. Discuss the design of spelling checkers for languages other than English. In connexion with this problem, how would you build a program to cheque spelling of the British flavour?
8. Discuss the specifications and implementation of other programs that might prove useful to writers who store their documents on computers.

Solutions for March's Problems

1. To modify *SiftDown* to have precondition *Heap*($L+1$, U) and postcondition *Heap*(L, U), change the first line of code from `I:=1` to `I:=L`. One can build a heap in $O(N)$ time with the code

```
for I := N-1 downto 1 do
        /* Inv: Heap(I+1,N) */
        SiftDown(I,N)
        /* Heap(I,N) */
```

   Because *Heap*(L,N) is true for all integers $L > N/2$, the bound $N-1$ in the `for` loop can be changed to floor($N/2$).
2. The BASIC program in Figure 4 is a reasonably effi-

```
1000 ' HEAPSORT X(1..N)
1010 IF N<=1 THEN RETURN
1020 U=N
1030 FOR L=INT(N/2) TO 1 STEP -1: GOSUB 1100: NEXT L
1040 L=1
1050 FOR U=N-1 TO 1 STEP -1
1060   T=X(1): X(1)=X(U+1): X(U+1)=T
1070   GOSUB 1100
1080 NEXT U
1090 RETURN

1100 ' SIFTDOWN: PRE MAXHEAP(L-1,U), POST MAXHEAP(L,U)
1110 I=L: T=X(I)
1120 ' LOOP INV: MAXHEAP(L,U) EXCEPT BETWEEN I AND ITS CHILDREN
1130   C=2*I
1140   IF C>U THEN GOTO 1190
1150   IF C<U THEN IF X(C+1)>X(C) THEN C=C+1
1160   IF T>=X(C) THEN GOTO 1190
1170   X(I)=X(C): I=C
1180   GOTO 1120
1190 X(I)=T: RETURN
```

FIGURE 4. Heapsort in BASIC

Further Reading

The details of McIlroy's `spell` program are described in his paper "Development of a spelling list," which appeared in *IEEE Transactions on Communications COM-30*, 1 (January 1982, pp. 91–99). It is fascinating and delightful reading, and a must for any serious student of programming.

"Computer programs for detecting and correcting spelling errors" by James L. Peterson appeared in *Communications of the ACM 23*, 12 (December 1980, pp. 676–687). The first part of the paper surveys the spelling problem and various implementations of checkers and correctors. He then describes a spelling corrector that he designed and implemented; the complete Pascal program is published in a Springer-Verlag monograph. The paper's 44 references are an excellent introduction to the literature of spelling.

cient implementation of Heapsort; the invocation `GOSUB 1000` sorts the array X[1 . . *N*]. It uses the linear-time heap-building algorithm of Problem 1 and moves the *Swap* assignments to and from the temporary variable *T* out of the *SiftDown* loop. The *SiftUp* procedure can be made faster by moving code out of loops and by placing a sentinel element of "negative infinity" in *X*[0] to remove the test `if I=1`.

3. Heaps replace a $O(N)$ step by a $O(\log N)$ step in all the problems.
   - **a.** The iterative step in constructing a Huffman code selects the two smallest nodes in the set and merges them into a new node; this is implemented by two *ExtractMins* followed by an *Insert*. If the input frequencies are presented in sorted order, then the Huffman code can be computed in linear time; the details are left as an exercise.
   - **b.** A simple algorithm to sum floating point numbers might lost accuracy by adding very small numbers to large numbers. A superior algorithm always adds the two smallest numbers in the set, and is isomorphic to the Huffman code algorithm mentioned above.
   - **c.** A 1,000-element heap represents the 1,000 largest numbers seen so far; the problem is trivial if the elements are sorted.
   - **d.** A heap can be used to merge sorted files by representing the next element in each file; the iterative step selects the smallest element from the heap and inserts its successor into the heap. The next element to be output from *N* files can be chosen in $O(\log N)$ time.
4. Johnson places a heap-like structure over the sequence of bins; each node in the heap tells the amount of space left in the least full bin among its descendants. When deciding where to place a new weight, the search goes left if it can (i.e., the least full bin to the left has enough space to hold it) and right if it must; that requires time proportional to the heap's depth of $O(\log N)$. After the weight is inserted, the path is traversed up to fix the weights in the heap.
5. The common implementation of a sequential file on disk has block *I* point to block $I + 1$. McCreight observed that if node *I* also points to node $2I$, then an arbitrary node *N* can be found in at most $1 + \log_2 N$ accesses. The following recursive program prints the access path.

```
func Path(N)
        pre   integer N>=0
        post  Path to N is printed
if N=0 then
        print "Start at 0"
else if even(N) then
        Path(N/2)
        print "Double to ", N
else
        Path(N-1)
        print "Increment to ", N
```

Notice that it is structurally isomorphic to this program for computing X^N in $O(\log N)$ steps.

```
func Exp(X,N)
        pre   integer N>=0
        post  result = X**N
if N=0 then
        return 1
else if even(N) then
        return square(Exp(X,N/2))
else
        return X*Exp(X,N-1)
```

6. The modified binary search begins with $I = 1$, and at each iteration sets *I* to either $2I$ or $2I + 1$. *X*[1] contains the median element, *X*[2] contains the first quartile, *X*[3] the third quartile, and so on. S. R. Mahaney of Bell Labs and J. I. Munro of the University of Waterloo have found a routine to put an *N*-element sorted array into "Heapsearch" order in $O(N)$ time and $O(1)$ extra space. As a precursor to their method, consider copying a sorted array *A* of size $2^K - 1$ into a "Heapsearch" array *B*: the odd elements of *A* go, in order, into the last half of the elements of *B*, and so on recursively.

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