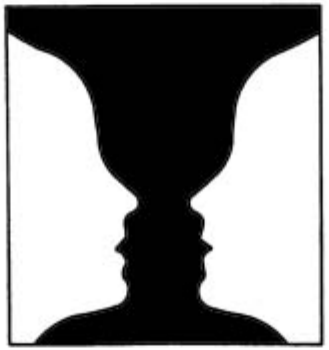
CSE403 ● Software engineering ● sp12

Week 3				
Monday	Tuesday	Wednesday	Thursday	Friday
• Design • Reading II due	• Group meetings • SRS due	• Design	• UML	• Design • Progress report due



Design is not just what it looks like and feels like. Design is how it works. –Steve Jobs

Apologies in advance: some slides with too much text (read off-line) and a really bad joke

What does this 1-line shell script do?

```
tr -cs A-Za-z '\n' | tr A-Z a-z | sort }
            uniq -c | sort -rn | sed ${1}q
```

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

Jon Bentley, Don Knuth, and Doug McIlroy. 1986. Programming pearls: a literate program.*Commun. ACM* 29, 6 (June 1986), 471-483. DOI=10.1145/5948.315654

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}).

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.

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."

frequency; or there
words. Let's be more
words are to be prin
quency, with words
phabetic order. Prin
have been output, if

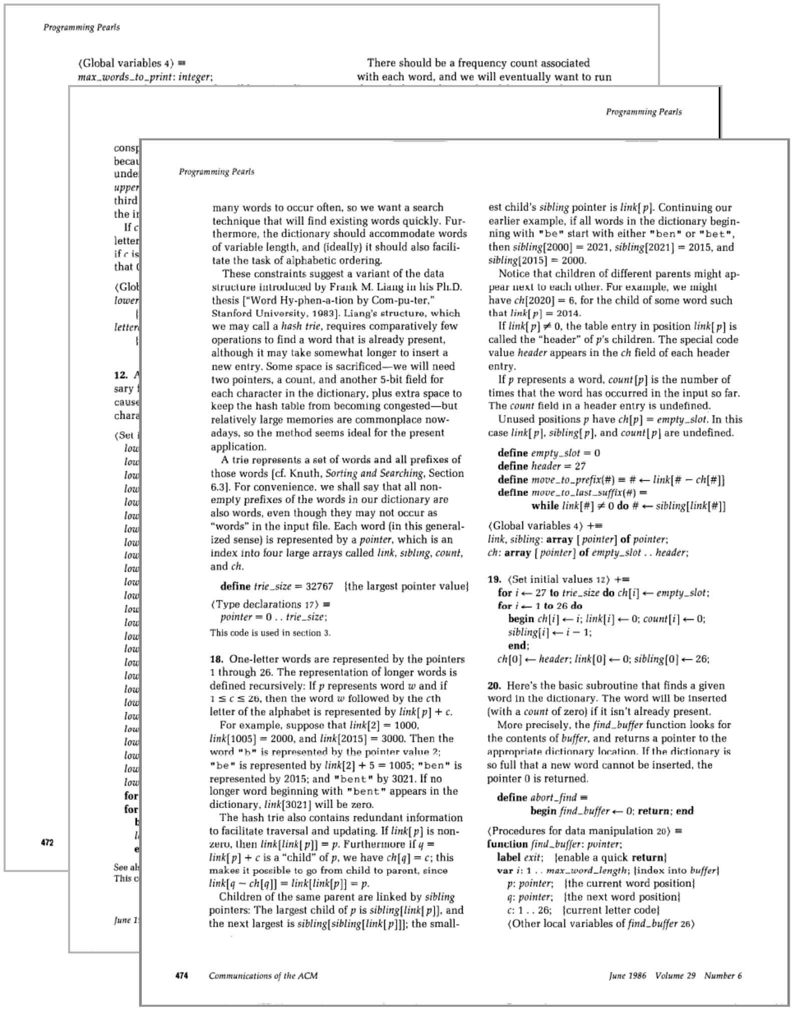
2. The *input* file is a
text. If it begins wit
(preceded by option
the value of *k*; other
k = 100. Answers w

define *d*... 100 {use this value if *k* isn't
...pecified}

...e given problem, this program is
...example of the WEB system, for
...ow some Pascal but who have never
...fore. Here is an outline of the program
...ed:

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.

Knuth’s version: literate programming



Programming Pearls

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;

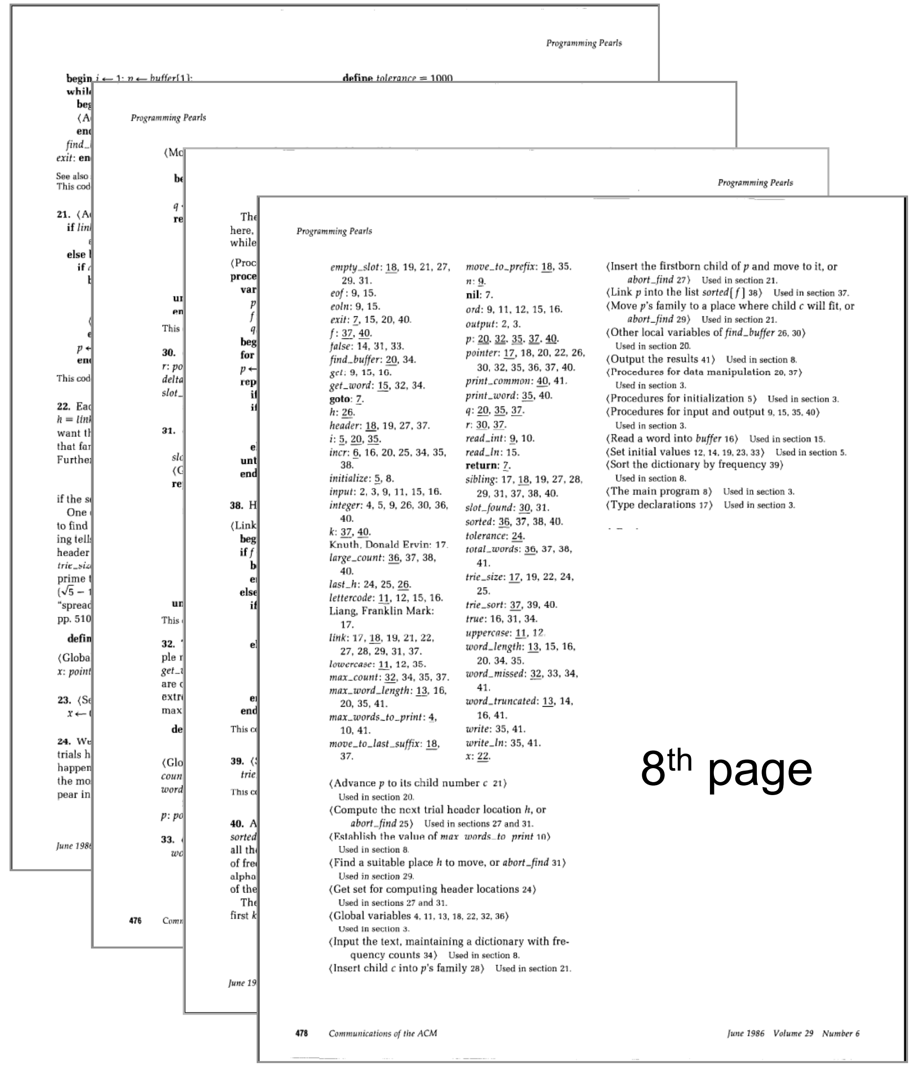
20. Here's the basic subroutine that finds a given word in the dictionary. The word will be inserted (with a *count* of zero) if it isn't already present.

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⟩

474 Communications of the ACM June 1986 Volume 29 Number 6



Programming Pearls

empty_slot: 18, 19, 21, 27, 29, 31.
eof: 9, 15.
eoln: 9, 15.
exit: 7, 15, 20, 40.
f: 37, 40.
false: 14, 31, 33.
find_buffer: 20, 34.
get: 9, 15, 16.
get_word: 15, 32, 34.
goto: 7.
h: 26.
header: 18, 19, 27, 37.
i: 5, 20, 35.
incr: 6, 16, 20, 25, 34, 35, 38.
initialize: 5, 8.
input: 2, 3, 9, 11, 15, 16.
integer: 4, 5, 9, 26, 30, 36, 40.
k: 37, 40.
Knuth, Donald Ervin: 17.
large_count: 36, 37, 38, 40.
last_h: 24, 25, 26.
lettercode: 11, 12, 15, 16.
Liang, Franklin Mark: 17.
link: 17, 18, 19, 21, 22, 27, 28, 29, 31, 37.
lowercase: 11, 12, 35.
max_count: 32, 34, 35, 37.
max_word_length: 13, 16, 20, 35, 41.
max_words_to_print: 4, 10, 41.
move_to_last_suffix: 18, 37.
move_to_prefix: 18, 35.
n: 9.
nil: 7.
ord: 9, 11, 12, 15, 16.
output: 2, 3.
p: 20, 32, 35, 37, 40.
pointer: 17, 18, 20, 22, 26, 30, 32, 35, 36, 37, 40.
print_common: 40, 41.
print_word: 35, 40.
q: 20, 35, 37.
r: 30, 37.
read_int: 9, 10.
read_ln: 15.
return: 7.
sibling: 17, 18, 19, 27, 28, 29, 31, 37, 38, 40.
slot_found: 30, 31.
sorted: 36, 37, 38, 40.
tolerance: 24.
total_words: 36, 37, 38, 41.
trie_size: 17, 19, 22, 24, 25.
trie_sort: 37, 39, 40.
true: 16, 31, 34.
uppercase: 11, 12.
word_length: 13, 15, 16, 20, 34, 35.
word_missed: 32, 33, 34, 41.
word_truncated: 13, 14, 16, 41.
write: 35, 41.
write_ln: 35, 41.
x: 22.

⟨Advance *p* to its child number *c* 21⟩ Used in section 20.
⟨Compute the next trial header location *h*, or *abort_find* 25⟩ Used in sections 27 and 31.
⟨Establish the value of *max_words_to_print* 10⟩ Used in section 8.
⟨Find a suitable place *h* to move, or *abort_find* 31⟩ Used in section 29.
⟨Get set for computing header locations 24⟩ Used in sections 27 and 31.
⟨Global variables 4, 11, 13, 18, 22, 32, 36⟩ Used in section 3.
⟨Input the text, maintaining a dictionary with frequency counts 34⟩ Used in section 8.
⟨Insert child *c* into *p*'s family 28⟩ Used in section 21.
⟨Insert the firstborn child of *p* and move to it, or *abort_find* 27⟩ Used in section 21.
⟨Link *p* into the list *sorted*[*f*] 38⟩ Used in section 37.
⟨Move *p*'s family to a place where child *c* will fit, or *abort_find* 29⟩ Used in section 21.
⟨Other local variables of *find_buffer* 26, 30⟩ Used in section 20.
⟨Output the results 41⟩ Used in section 8.
⟨Procedures for data manipulation 20, 37⟩ Used in section 3.
⟨Procedures for initialization 5⟩ Used in section 3.
⟨Procedures for input and output 9, 15, 35, 40⟩ Used in section 3.
⟨Read a word into *buffer* 16⟩ Used in section 15.
⟨Set initial values 12, 14, 19, 23, 33⟩ Used in section 5.
⟨Sort the dictionary by frequency 39⟩ Used in section 8.
⟨The main program 8⟩ Used in section 3.
⟨Type declarations 17⟩ Used in section 3.

478 Communications of the ACM June 1986 Volume 29 Number 6

8^{th} page

McIroy's exposition

Engineering: "Design under Constraints"

- A wise engineering solution would produce—or better, exploit—reusable parts.

Reuse

- Very few people can obtain the virtuoso services of Knuth …but old UNIX hands know instinctively how to solve this one in a jiffy.

Resources

- … 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.

Reuse++

- 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.

Process

Challenger Disaster: Feynman

The usual way that such engines are designed … may be called the component system, or bottom-up design. First it is necessary to thoroughly understand the properties and limitations of the materials to be used (for turbine blades, for example), and tests are begun in experimental rigs to determine those. With this knowledge larger component parts … are designed and tested individually. As deficiencies and design errors are noted they are corrected and verified with further testing. … Finally one works up to the final design of the entire engine, to the necessary specifications. There is a good chance, by this time that the engine will generally succeed, or that any failures are easily isolated and analyzed because the failure modes, limitations of materials, etc., are so well understood. …

Roughly, build bottom-up with components with known properties

The Space Shuttle Main Engine was handled in a different manner, top down, we might say. The engine was designed and put together all at once with relatively little detailed preliminary study of the material and components. Then when troubles are found in the bearings, turbine blades, coolant pipes, etc., it is more expensive and difficult to discover the causes and make changes. For example, cracks have been found in the turbine blades of the high pressure oxygen turbopump. Are they caused by flaws in the material, the effect of the oxygen atmosphere on the properties of the material, the thermal stresses of startup or shutdown, … or mainly at some resonance at certain speeds, etc.? … Using the completed engine as a test bed to resolve such questions is extremely expensive. One does not wish to lose an entire engine in order to find out where and how failure occurs. Yet, an accurate knowledge of this information is essential to acquire a confidence in the engine reliability in use. …

A further disadvantage of the top-down method is that, if an understanding of a fault is obtained, a simple fix, such as a new shape for the turbine housing, may be impossible to implement without a redesign of the entire engine.

The point?

- Software design – like all engineering design – has a set of dimensions and criteria to consider
  - Correctness, cost, performance, robustness, usability, understandability, modifiability, …
  - Some of these properties come directly from parts of the software system
  - Others are more properties of the overall system, sometimes called *emergent* properties
- These are constraints that, in part, distinguish software engineering from the theoretical foundations of computation – that work is critical, and software engineering augments it with constraints
- Underlying all effective software design – indeed, computational thinking – is the notion of *abstraction*

Continuous & iterative

Continuous • iterative

- High-level ("architectural") design
  - What pieces?
  - How connected?
- Low-level design
  - Should I use a hash table or binary search tree?
- Very low-level design
  - Variable naming, which language constructs, etc.
  - Boolean Zen
  - About 1000 design decisions at various levels are made in producing a single page of code

A few key criteria for software design

- Accommodating change – taking advantage of software's "soft"-ness
  - Agile Manifesto; "Software that does not change becomes useless over time" [Belady & Lehman]; …
- Generality vs. performance
  - In math, a more general theorem is always better than a less general one
  - In software, a less general solution may consume enough fewer resources to dominate a more general solution – but don't forget #1
- Complexity – physical properties constrain physical design, but fewer constraints are imposed by software as a material

Abstraction

Kramer, CACM 2007

- "…[remove] detail to simplify and focus attention based on the definitions:
  - The act of withdrawing or removing something, and;
  - The act or process of leaving out of consideration one or more properties of a complex object so as to attend to others.
- "…the process of generalization to identify the common core or essence based on the definitions:
  - The process of formulating general concepts by abstracting common properties of instances, and;
  - A general concept formed by extracting common features from specific examples."

Computational thinking

Wing, CACM 2006

- "Computational thinking is using abstraction and decomposition when attacking a large complex task or designing a large complex system. It is separation of concerns. It is choosing an appropriate representation for a problem or modeling the relevant aspects of a problem to make it tractable. It is using invariants to describe a system's behavior succinctly and declaratively. It is having the confidence we can safely use, modify, and influence a large complex system without understanding its every detail. It is modularizing something in anticipation of multiple users or prefetching and caching in anticipation of future use.
- "Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction…"

Mechanisms for abstraction?

Two minutes

- Methods
- Classes

In small groups list other software abstraction mechanisms

Decomposition and composition

- The technique of mastering complexity has been known since ancient times: *Divide et impera*
  —Dijkstra, 1965
  - …strategy of gaining and maintaining power by breaking up larger concentrations of power into chunks that individually have less power than the one implementing the strategy. —Wikipedia, today
- Divide and conquer. Separate your concerns. Yes. But sometimes the conquered tribes must be reunited under the conquering ruler, and the separated concerns must be combined to serve a single purpose." —M. Jackson, 1995

Benefits of decomposition

- Decrease size of tasks
- Support independent testing and analysis
- Separate work assignments
- Ease understanding

- In principle, can significantly reduce paths to consider by introducing an interface
  - Consider the Knuth and McIlory examples
  - Many more…

Alan Perlis quotations: abstraction

- If you have a procedure with 10 parameters, you probably missed some.
- One man's constant is another man's variable.
- Simplicity does not precede complexity, but follows it.
  - Our designs are so complex there is no hope of getting them right first time by pure thought. To expect to is arrogant. —Brooks

Anticipating change & design

- It is generally believed that to accommodate change one must anticipate possible changes
  - Counterpoint: Extreme Programming
- By anticipating (and perhaps prioritizing) changes, one defines additional criteria for guiding the design activity – *what abstractions one should choose*
- It is not possible to anticipate all changes

Anticipating vs. not anticipating: comments?

Extra credit

KWIC: “hello world” of module design

The KWIC index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a list of all circular shifts of all lines in alphabetical order.

Another script…

```
awk '{print $0
for (i = length($0); i > 0; i--)
  if (substr($0,i,1) == " ")
      print substr($0,i+1) "\t" substr($0,1,i-1)
}' $1 | sort -f | awk '
BEGIN {FS = "\t"; WID = 30}
{printf("%" WID "s      %s\n",
        substr($2,length($2)-WID+1),substr($1,1,WID))
}'
```

- Why not always the best?
- What might change?
- More in lecture and in Reading III

CSE403 ● Software engineering ● sp12

Week 3				
Monday	Tuesday	Wednesday	Thursday	Friday
• Design • Reading II due	• Group meetings • SRS due	• Design	• UML	• Design • Progress report due