October 7, 1998 at 08:54

1. Introduction. This program is a simple filter that sorts and outputs all lines of input that do not appear in a given set of sorted files. It is called `wordtest` because each line of input is considered to be a 'word' and each of the sorted files is considered to be a 'dictionary'. Words are output when they don't appear in any given dictionary.

The character set and alphabetic order are flexible. Every 8-bit character is mapped into an integer called its *ord*. A character is called a *null* if its ord is zero; such characters are discarded from the input. A character is called a *break* if its ord is negative; such characters break the input into so-called words. Otherwise a character's ord is positive, and the character is called a *letter*. One letter precedes another in alphabetic order if and only if it has a smaller ord. Two letters are considered identical, for purposes of sorting, if their ords are the same.

The null character `'\n'` must have ord 0; thus, it must remain null. Otherwise the ord mapping is arbitrary. If the user doesn't specify any special mapping, the default ord table simply maps every 8-bit character code into itself, considering characters to be unsigned char values in the range 0–255, except that ASCII codes `a-z` are mapped into the corresponding codes for `A-Z`, and newline is a break character. Optional command-line arguments, described below, can change this default mapping to any other desired scheme.

A word is any nonempty sequence of letters that is immediately preceded and followed by break characters, when nulls are ignored. Technically speaking, we pretend that a break character is present at the beginning of a file but not at the end; thus, all letters following the final break character of a file are ignored, if any such letters are present. Two words are *equivalent* to each other if their letters have the same sequence of ord values. If two or more words of the input are equivalent, only the first will be output, and it will be output only if it is not equivalent to any word in the given dictionary files. Words in each dictionary are assumed to be in lexicographic order and to contain no nulls. Words in the output file will satisfy these conditions; therefore `wordtest` can be used to generate and update the dictionaries it needs. Notice that if no dictionaries are given, `wordtest` will act as a sorting routine that simply discards nulls and duplicate lines.

2. The UNIX command line '`wordtest [options] [dictionaries]`' is interpreted by executing option commands from left to right and then by regarding any remaining arguments as the names of dictionary files.

Most of the option commands are designed to specify the *ord* table. Initially $ord[c] = c$ for each unsigned char code c. The command

`-b`*string*

makes every character in the string a break character. If the string is empty, `-b` makes every nonnull character a break (i.e., it sets $ord[c] = -1$ for $1 \le c \le 255$). The command

`-n`*string*

makes every character in the string a null character. If the string is empty, `-n` makes every character null. The command

`-a`*string*

sets the ord of the kth element of the string equal to $\delta + k$, where δ is an offset value (normally zero). The command

`-d`*offset*

sets the value of δ; the offset should be a decimal integer between 0 and 255.

There is also an option that has no effect on the *ord* table:

`-m`*length*

defines the length of the longest word. If any word of a file has more than this many characters, a break is artificially inserted so that a word of this maximum length is obtained. The default value is 50. The maximum legal value is 1000.

If the given options do not specify at least one break character, `wordtest` applies the option commands

```
-b"\
" -d64 -a"abcdefghijklmnopqrstuvwxyz"
```

which generate the default mapping mentioned above (unless other ords were changed).

The program is designed to run fastest when there are at most two dictionary files (usually one large system dictionary and another personalized one), although it places no limit on the actual number of dictionaries that can be mentioned on the command line. Users who want to specify a multitude of dictionaries should ask themselves why they wouldn't prefer to merge their dictionaries together first (using `wordtest`).

#define `MAX_LENGTH_DEFAULT` 50
#define `MAX_LENGTH_LIMIT` 1000

3. The general organization of `wordtest` is typical of applications written in C, and its approach is quite simple. If any errors are detected, an indication of the error is sent to the *stderr* file and a nonzero value is returned.

```
#include <stdio.h>
  ⟨ Typedefs 4 ⟩
  int main(argc, argv)
      int argc;    /* the number of command-line arguments */
      char *argv[];    /* the arguments themselves */
  {
    ⟨ Local variables 5 ⟩;
    ⟨ Scan the command line arguments 6 ⟩;
    ⟨ Sort the input into memory 17 ⟩;
    ⟨ Output all input words that aren't in dictionaries 19 ⟩;
    return 0;
  }
```

4. ⟨ Typedefs 4 ⟩ ≡

```
  typedef unsigned char byte;    /* our bytes will range from 0 to 255 */
```

See also sections 9 and 20.

This code is used in section 3.

5. ⟨ Local variables 5 ⟩ ≡

```
  int targc;    /* temporary modifications to argc */
  byte **targv;    /* pointer to the current argument of interest */
  unsigned delta;    /* the offset used in the -a and -d options */
  unsigned max_length = MAX_LENGTH_DEFAULT;    /* longest allowable word */
  byte breakchar;    /* break character to use in the output */
  int ord[256];    /* table of ord values */
  register int c;    /* an all-purpose index */
  register byte *u, *v;    /* pointer to current string characters */
```

See also sections 12, 16, and 22.

This code is used in section 3.

6. We try to use newline as the output break character, if possible.

```
⟨ Scan the command line arguments 6 ⟩ ≡
  for (c = 0; c < 256; c++) ord[c] = c;
  delta = 0;
  targc = argc − 1; targv = (byte **) argv + 1;
  while (targc ∧ **targv ≡ '-') {
    ⟨ Execute the option command targv 7 ⟩;
    targc−−; targv++;
  }
  if (ord['\n'] < 0) breakchar = '\n';
  else {
    breakchar = '\0';
    for (c = 255; c; c−−)
      if (ord[c] < 0) breakchar = c;
    if (¬breakchar) ⟨ Set up the default ords 8 ⟩;
  }
  ⟨ Allocate data structures for a total of targc files 21 ⟩;
  for ( ; targc; targc−−, targv++) ⟨ Open the dictionary file named *targv 23 ⟩;
```

This code is used in section 3.

7.

```
⟨ Execute the option command targv 7 ⟩ ≡
  switch ((*targv)[1]) {
  case 'a':
    for (c = delta, u = *targv + 2; *u; u++) ord[*u] = ++c; break;
  case 'b':
    if ((*targv)[2])
      for (u = *targv + 2; *u; u++) ord[*u] = −1;
    else
      for (c = 1; c < 256; c++) ord[c] = −1;
    break;
  case 'n':
    if ((*targv)[2])
      for (u = *targv + 2; *u; u++) ord[*u] = 0;
    else
      for (c = 1; c < 256; c++) ord[c] = 0;
    break;
  case 'd':
    if (sscanf((char *) *targv + 2, "%u", &delta) ≡ 1 ∧ delta < 256) break;
    goto print_usage;
  case 'm':
    if (sscanf((char *) *targv + 2, "%u", &max_length) ≡ 1 & max_length ≤ MAX_LENGTH_LIMIT) break;
    goto print_usage;
  default: print_usage:
    fprintf(stderr, "Usage:␣%s␣{-{{a|b|n}string|{d|m}number}}*␣dictionaryname*\n", *argv);
    return −1;
  }
```

This code is used in section 6.

8. ⟨ Set up the default ords 8 ⟩ ≡

```
{
  ord['\n'] = -1;    /* newline is break character */
  breakchar = '\n';
  for (c = 1; c <= 26; c++) ord['a' - 1 + c] = 'A' - 1 + c;
}
```

This code is used in section 6.

9. Treaps. The most interesting part of this program is its sorting algorithm, which is based on the "treap" data structure of Aragon and Seidel [*30th IEEE Symposium on Foundations of Computer Science* (1989), 540–546]. A treap is a binary tree whose nodes have two key fields. The primary key, which in our application is a word from the input, obeys tree-search order: All descendants of the left child of node p have a primary key that is less than the primary key of p, and all descendants of its right child have a primary key that is greater. The secondary key, which in our application is a unique pseudorandom integer attached to each input word, obeys heap order: The secondary key of p's children is greater than p's own secondary key.

A given set of nodes with distinct primary keys and distinct secondary keys can be made into a treap in exactly one way. This unique treap can be obtained, for example, by using ordinary tree insertion with respect to primary keys while inserting nodes in order of their secondary keys. It follows that, if the secondary keys are random, the binary tree will almost always be quite well balanced.

We will compute secondary keys as unsigned long integers, assigning the key $(cn) \bmod 2^{32}$ to the nth node, where c is an odd number. This will guarantee that the secondary keys are distinct. By choosing c close to $2^{32}/\phi$, where ϕ is the golden ratio $(1+\sqrt{5})/2$, we also spread the values out in a fashion that is unlikely to match any existing order in the data.

```
#define PHICLONE 2654435769   /* ≈ 2^32/φ */
```

```
⟨ Typedefs 4 ⟩ +≡
  typedef struct node_struct {
    struct node_struct *left, *right;   /* children */
    byte *keyword;   /* primary key */
    unsigned long rank;   /* secondary key */
  } node;   /* node of a treap */
```

10. We want to be able to compare two strings rapidly with respect to lexicographic order, as defined by the *ord* table. This can be done if one string is delimited by '\0' as usual, while the other is delimited by a break character. Then we are sure to have an unequal comparison, and the inner loop is fast.

Here is a routine that checks to see if a word is already present in the treap. The word is assumed to be in *buffer*, terminated by *breakchar*. The words in the treap are terminated by nulls. The treap is accessed by means of *root*, a pointer to its root node.

```
⟨ Search for buffer in the treap; goto found if it's there 10 ⟩ ≡
  { register node *p = root;
    while (p) {
      for (u = buffer, v = p→keyword; ord[*u] ≡ ord[*v]; u++, v++) ;
      if (*v ≡ '\0' ∧ *u ≡ breakchar) goto found;
      if (ord[*u] < ord[*v]) p = p→left;
      else p = p→right;
    }
  }
```

This code is used in section 17.

11. We don't need to insert nodes into the treap as often as we need to look words up, so we don't mind repeating the comparisons already made when we discover that insertion is necessary. (Actually a more comprehensive study of this tradeoff ought to be done. But not today; I am trying here to keep the program short and sweet.)

The insertion algorithm proceeds just as the lookup algorithm until we come to a node whose rank is larger than the rank of the node to be inserted. We insert the new node in its place, then split the old node and its descendants into two subtrees that will become the left and right subtrees of the new node.

```
⟨ Insert the buffer word into the treap 11 ⟩ ≡
  { register node *p, **q, **qq, *r;
    current_rank += PHICLONE;    /* unsigned addition mod 2^32 */
    p = root; q = &root;
    while (p) {
      if (p→rank > current_rank) break;    /* end of the first phase */
      for (u = buffer, v = p→keyword; ord[*u] ≡ ord[*v]; u++, v++) ;
      if (ord[*u] < ord[*v]) q = &(p→left), p = *q;
      else q = &(p→right), p = *q;
    }
    ⟨ Set r to the address of a new node, and move buffer into it 14 ⟩;
    r→rank = current_rank;
    *q = r;    /* link the new node into the tree */
    ⟨ Split subtree p and attach it below node r 13 ⟩;
  }
```

This code is used in section 17.

12. ⟨ Local variables 5 ⟩ +≡

```
  unsigned long current_rank = 0;    /* pseudorandom number */
```

13. At this point p may already be empty. If not, we can hook its parts together easily. (A formal proof is a bit tricky, but the computer doesn't slow down like people do when they get to a conceptually harder part of an algorithm.)

```
⟨ Split subtree p and attach it below node r 13 ⟩ ≡
  q = &(r→left); qq = &(r→right);    /* slots to fill in as we split the subtree */
  while (p) {
    for (u = buffer, v = p→keyword; ord[*u] ≡ ord[*v]; u++, v++) ;
    if (ord[*u] < ord[*v]) {
      *qq = p;
      qq = &(p→left);
      p = *qq;
    }
    else {
      *q = p;
      q = &(p→right);
      p = *q;
    }
  }
  *q = *qq = Λ;
```

This code is used in section 11.

14. We allocate node memory dynamically, in blocks of 100 nodes at a time. We also allocate string memory dynamically, 1000 characters at once (in addition to space for the current string). The variable l will be set to the length of the word in *buffer*.

```
#define NODES_PER_BLOCK 100
#define CHARS_PER_BLOCK 1000
#define out_of_mem(x)
        { fprintf(stderr, "%s: Memory exhausted!\n", *argv);
          return x; }
```

⟨ Set r to the address of a new node, and move *buffer* into it 14 ⟩ ≡

```
  if (next_node ≡ bad_node) {
    next_node = (node *) calloc(NODES_PER_BLOCK, sizeof(node));
    if (next_node ≡ Λ) out_of_mem(−2);
    bad_node = next_node + NODES_PER_BLOCK;
  }
  r = next_node++;
  ⟨ Move buffer to a new place in the string memory, and make r→keyword point to it 15 ⟩;
```

This code is used in section 11.

15. ⟨ Move *buffer* to a new place in the string memory, and make *r→keyword* point to it 15 ⟩ ≡

```
  if (next_string + l + 1 ≥ bad_string) { int block_size = CHARS_PER_BLOCK + l + 1;
    next_string = (byte *) malloc(block_size);
    if (next_string ≡ Λ) out_of_mem(−3);
    bad_string = next_string + block_size;
  }
  r→keyword = next_string;
  for (u = buffer, v = next_string; ord[*u] > 0; u++, v++) *v = *u;
  *v = '\0';
  next_string = v + 1;
```

This code is used in section 14.

16. We had better define the variables we've been assuming in these storage allocation routines.

⟨ Local variables 5 ⟩ +≡

```
  node *next_node = Λ, *bad_node = Λ;
  byte *next_string = Λ, *bad_string = Λ;
  node *root = Λ;
  byte *buffer;
  int l;    /* length of current string in buffer */
```

17. The mechanisms for sorting the input words are now all in place. We merely need to invoke them at the right times.

```
⟨ Sort the input into memory 17 ⟩ ≡
  buffer = (byte *) malloc(max_length + 1);
  if (buffer ≡ Λ) out_of_mem(−5);
  while (1) {
    ⟨ Set buffer to the next word from stdin; goto done if file ends 18 ⟩;
    if (l) {
      ⟨ Search for buffer in the treap; goto found if it's there 10 ⟩;
      ⟨ Insert the buffer word into the treap 11 ⟩;
    found: ;
    }
  }
done: ;
```

This code is used in section 3.

18. ⟨ Set *buffer* to the next word from *stdin*; **goto** *done* if file ends 18 ⟩ ≡

```
  u = buffer; l = 0;
  while (l < max_length) {
    c = getchar();
    if (c ≡ EOF) {
      if (ferror(stdin)) {
        fprintf(stderr, "%s:␣File␣read␣error␣on␣standard␣input!\n", *argv);
        return −6;
      }
      goto done;      /* end of file; the current word, if any, is discarded */
    }
    if (ord[c] ≤ 0) {
      if (ord[c] < 0) break;
    }
    else {
      *u++ = (byte) c;
      l++;
    }
  }
  *u = breakchar;
```

This code is used in section 17.

19. At the end we want to traverse the treap in symmetric order, so that we see its words in alphabetic order. We might as well destroy the treap structure as we do this. During this phase, *root* will point to a stack of nodes that remain to be visited (followed by traversal of their right subtrees).

⟨ Output all input words that aren't in dictionaries 19 ⟩ ≡

```
if (root ≠ Λ) { register node *p, *q;
    p = root;
    root = Λ;
    while (1) {
      while (p→left ≠ Λ) {
        q = p→left;
        p→left = root;    /* left links are now used for the stack */
        root = p;
        p = q;
      }
    visit: ⟨ Output p→keyword, if it's not in the dictionaries 25 ⟩;
      if (p→right ≡ Λ) {
        if (root ≡ Λ) break;    /* the stack is empty, we're done */
        p = root;
        root = root→left;    /* pop the stack */
        goto visit;
      }
      else p = p→right;
    }
  }
```

This code is used in section 3.

20. The dictionaries. So now all we have to do is provide a mechanism for reading the words in the dictionaries. The dictionaries are sorted, and by now the input words have been sorted too. So we need only scan through the dictionaries once; we'll try to zoom through as quickly as possible.

First we need data structures. There will be an array of pointers to filenodes, for all dictionary files currently open. Each filenode will contain a buffer of size BUFSIZ + 1 for raw input bytes not yet scanned, as well as a buffer of size MAX_LENGTH_LIMIT + 1 for the current word being considered.

```
⟨ Typedefs 4 ⟩ +≡
  typedef struct filenode_struct {
    struct filenode_struct *link;   /* pointer to next open file */
    FILE *dfile;   /* dictionary file */
    byte buf[BUFSIZ + 1], curword[MAX_LENGTH_LIMIT + 1];
    byte *pos;   /* current position in buf */
    byte *limit;   /* end of input bytes in buf */
    byte *endword;   /* the first break character in curword */
  } filenode;
```

21. ⟨ Allocate data structures for a total of *targc* files 21 ⟩ ≡

```
  if (targc) {
    curfile = (filenode *) calloc(targc, sizeof(filenode));
    if (curfile ≡ Λ) out_of_mem(−7);
    for (f = curfile; f < curfile + targc − 1; f++) f→link = f + 1;
    f→link = curfile;   /* circular linking */
  }
  else curfile = Λ;
```

This code is used in section 6.

22. ⟨ Local variables 5 ⟩ +≡

```
  filenode *curfile;   /* current filenode of interest */
  filenode *f;   /* temporary register for filenode list processing */
```

23. ⟨ Open the dictionary file named *targv 23 ⟩ ≡

```
  {
    curfile→dfile = fopen((char *) *targv, "r");
    if (curfile→dfile ≡ Λ) {
      fprintf(stderr, "%s:␣Can't␣open␣dictionary␣file␣%s!\n", *argv, (char *) *targv);
      return −8;
    }
    curfile→pos = curfile→limit = curfile→buf;   /* buf is empty */
    curfile→buf[0] = '\0';
    curfile→endword = curfile→curword;   /* curword is empty too */
    curfile→curword[0] = breakchar;
    curfile = curfile→link;   /* move to next filenode */
  }
```

This code is used in section 6.

24. We will implicitly merge the dictionaries together by using a brute force scheme that works fine when there are only a few of them. Namely, *curfile* will point to a file having the currently smallest current word. To get to the next word of the merge, we advance to the next word in that file, comparing it with the current words of the other files to see if *curfile* should switch to one of them. When we get to the end of a file, its filenode simply leaves the circular list. Eventually the list will be empty, and we will set *curfile* to Λ; we will then have seen all the dictionary words in order.

25. ⟨ Output $p\to keyword$, if it's not in the dictionaries 25 ⟩ ≡
  while ($curfile \neq \Lambda$) {
    for ($u = p\to keyword, v = curfile\to curword$; $ord[*u] \equiv ord[*v]$; $u{+}{+}, v{+}{+}$) ;
    if ($*u \equiv$ '\0' $\wedge *v \equiv breakchar$) **goto** *word_done*;   /* we found it in the dictionary */
    if ($ord[*u] < ord[*v]$) **break**;   /* we didn't find it */
    ⟨ Advance to the next dictionary word 27 ⟩;
  }
  ⟨ Print $p\to keyword$ and *breakchar* on *stdout* 26 ⟩
word_done: ;

This code is used in section 19.

26. ⟨ Print $p\to keyword$ and *breakchar* on *stdout* 26 ⟩ ≡
  for ($u = p\to keyword$; $*u$; $u{+}{+}$) $putchar(*u)$;
  $putchar(breakchar)$;

This code is used in section 25.

27. ⟨ Advance to the next dictionary word 27 ⟩ ≡
  ⟨ Read a new word into $curfile\to curword$, as fast as you can 28 ⟩;
  ⟨ Adjust *curfile*, if necessary, to point to a file with minimal *curword* 30 ⟩;

This code is used in section 25.

28. The dictionaries are supposed to be in order, and they shouldn't contain nulls. But if they fail to meet these criteria, we don't want `wordtest` to crash; it should just run more slowly and/or more peculiarly.

The logic of the code here removes null characters, at the cost of speed. If the dictionary contains words out of order, say $\alpha > \beta$ where α precedes β in the file, the effect will be as if β were not present. (In particular, if the dictionary would happen to have a null word because of a break character inserted by our *max_length* logic, that null word would cause no harm, because a null word is always less than any nonnull word.)

A null character always appears in *curfile→limit*.

```
⟨ Read a new word into curfile→curword, as fast as you can 28 ⟩ ≡
  v = curfile→curword;
  l = max_length;     /* here l represents max characters to put in curword */
  while (1) { register byte *w = curfile→limit;
    u = curfile→pos;
    if (u + l ≥ w)
      while (ord[*u] > 0) *v++ = *u++;     /* this is the inner loop */
    else {
      w = u + l;
      c = *w;
      *w = '\0';     /* temporarily store a null to avoid overlong string */
      while (ord[*u] > 0) *v++ = *u++;     /* this too is the inner loop */
      *w = c;     /* restore the damaged byte */
    }
    if (ord[*u] < 0) {
      curfile→pos = u + 1;     /* good, we found the next break character */
      break;
    }
    l −= u − curfile→pos;
    if (l ≡ 0) {     /* max_length reached */
      curfile→pos = u;
      break;
    }
    if (u ≡ w) {     /* we're at curfile→limit */
      ⟨ Refill curfile→buf; or remove the current file from the circular list and goto update_done, if it has
          ended 29 ⟩;
    }
    else curfile→pos = u + 1;     /* bypass a null character in the dictionary */
  }
  curfile→endword = v;
  *v = breakchar;
update_done: ;
```

This code is used in section 27.

29. ⟨ Refill *curfile→buf*; or remove the current file from the circular list and **goto** *update_done*, if it has ended 29 ⟩ ≡

```
if (ferror(curfile→dfile)) {
  fprintf(stderr, "%s: File read error on dictionary file!\n", *argv);
  return −9;
}
if (feof(curfile→dfile)) {
  f = curfile→link;
  if (f ≡ curfile) curfile = Λ;    /* the last dictionary file has ended */
  else {
    while (f→link ≠ curfile) f = f→link;
    f→link = curfile→link;    /* remove a filenode from the circular list */
    curfile = f;    /* and point to one of the remaining filenodes */
  }
  goto update_done;
}
curfile→limit = curfile→buf + fread(curfile→buf, 1, BUFSIZ, curfile→dfile);
*curfile→limit = '\0';
curfile→pos = curfile→buf;
```

This code is used in section 28.

30. ⟨ Adjust *curfile*, if necessary, to point to a file with minimal *curword* 30 ⟩ ≡

```
if (curfile ≠ Λ) { filenode *sentinel = curfile;
  for (f = curfile→link; f ≠ sentinel; f = f→link)
    ⟨ Change curfile to f if f→curword < curfile→curword 31 ⟩;
}
```

This code is used in section 27.

31. ⟨ Change *curfile* to *f* if *f→curword* < *curfile→curword* 31 ⟩ ≡

```
{
  *f→endword = '\0';
  for (u = f→curword, v = curfile→curword; ord[*u] ≡ ord[*v]; u++, v++) ;
  if (ord[*u] < ord[*v]) curfile = f;
  *f→endword = breakchar;
}
```

This code is used in section 30.

32. Index. Here is a list of the identifiers used by `wordtest`, showing the sections in which they appear, underlined at points of definition.

⟨ Adjust *curfile*, if necessary, to point to a file with minimal *curword* 30 ⟩ Used in section 27.
⟨ Advance to the next dictionary word 27 ⟩ Used in section 25.
⟨ Allocate data structures for a total of *targc* files 21 ⟩ Used in section 6.
⟨ Change *curfile* to f if $f\to curword < curfile\to curword$ 31 ⟩ Used in section 30.
⟨ Execute the option command *targv* 7 ⟩ Used in section 6.
⟨ Insert the *buffer* word into the treap 11 ⟩ Used in section 17.
⟨ Local variables 5, 12, 16, 22 ⟩ Used in section 3.
⟨ Move *buffer* to a new place in the string memory, and make $r\to keyword$ point to it 15 ⟩ Used in section 14.
⟨ Open the dictionary file named $*targv$ 23 ⟩ Used in section 6.
⟨ Output all input words that aren't in dictionaries 19 ⟩ Used in section 3.
⟨ Output $p\to keyword$, if it's not in the dictionaries 25 ⟩ Used in section 19.
⟨ Print $p\to keyword$ and *breakchar* on *stdout* 26 ⟩ Used in section 25.
⟨ Read a new word into $curfile\to curword$, as fast as you can 28 ⟩ Used in section 27.
⟨ Refill $curfile\to buf$; or remove the current file from the circular list and **goto** *update_done*, if it has ended 29 ⟩ Used in section 28.
⟨ Scan the command line arguments 6 ⟩ Used in section 3.
⟨ Search for *buffer* in the treap; **goto** *found* if it's there 10 ⟩ Used in section 17.
⟨ Set up the default ords 8 ⟩ Used in section 6.
⟨ Set *buffer* to the next word from *stdin*; **goto** *done* if file ends 18 ⟩ Used in section 17.
⟨ Set r to the address of a new node, and move *buffer* into it 14 ⟩ Used in section 11.
⟨ Sort the input into memory 17 ⟩ Used in section 3.
⟨ Split subtree p and attach it below node r 13 ⟩ Used in section 11.
⟨ Typedefs 4, 9, 20 ⟩ Used in section 3.

WORDTEST