Software Construction

word_frequency0.c

```
 Written in C for "speed" but slow on large inputs:



 % gcc -O3 -o word_frequency0  word_frequency0.c
 % time word_frequency0 <WarAndPeace.txt >/dev/null
 real    0m52.726s
 user    0m52.643s
 sys     0m0.020s
```

```
 Profiling with gprof revels get function is problem



 gcc -p -g word_frequency0.c -o word_frequency0_profile
 head -10000 WarAndPeace.txt|word_frequency0_profile >/dev/null
 % gprof word_frequency0_profile
   %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 88.90      0.79     0.79    88335     0.01     0.01  get
  7.88      0.86     0.07     7531     0.01     0.01  put
  2.25      0.88     0.02    80805     0.00     0.00  get_word
  1.13      0.89     0.01        1    10.02   823.90  read_words
  0.00      0.89     0.00        2     0.00     0.00  size
  0.00      0.89     0.00        1     0.00     0.00  create_map
  0.00      0.89     0.00        1     0.00     0.00  keys
  0.00      0.89     0.00        1     0.00     0.00  sort_words
....
```

```c
#include <stdlib.h>
#include "time.h"
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*
 * returns the next word from the streeam
 * a word is a non-zero length sequence of
 * alphabetic characters
 *
 * NULL is returned if there are no more words to be read
 */
char *
get_word(FILE *stream) {
    int i, c;
    char *w;
    static char *buffer = NULL;
    static int buffer_length = 0;

    if (buffer == NULL) {
        buffer_length = 32;
        buffer = malloc(buffer_length*sizeof (char));
        if (buffer == NULL) {
            fprintf(stderr, "out of memory\n");
            exit(1);
        }
    }

    i = 0;
    while ((c = fgetc(stream)) != EOF) {
        if (!isalpha(c) && i == 0)
            continue;
        if (!isalpha(c))
            break;
        if (i >= buffer_length) {
            buffer_length += 16;
            buffer = realloc(buffer, buffer_length*sizeof (char));
            if (buffer == NULL) {
                fprintf(stderr, "out of memory\n");
                exit(1);
            }
        }
        buffer[i++] = c;
    }

    if (i == 0)
        return NULL;

    buffer[i] = '\0';

    w = malloc(strlen(buffer) + 1);
    if (w == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    strcpy(w, buffer);
    return w;
}

typedef struct map  map;

struct map {
       int                size;
       struct map_node   *list;
};
```

```
struct map_node {
        char            *key;
        void            *value;
        struct map_node   *next;
};


map *
create_map() {
    struct map *m;
    if ((m = malloc(sizeof *m)) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    m->size = 0;
    m->list = NULL;
    return m;
}

void *get(map *m, char *key) {
    struct map_node *v;
    for (v = m->list; v != NULL; v = v->next) {
        if (strcmp(key, v->key) == 0) {
            return v->value;
        }
    }
    return NULL;
}

void
put(map *m, char *key, void *value) {
    struct map_node *v;
    for (v = m->list; v != NULL; v = v->next) {
        if (strcmp(key, v->key) == 0) {
            v->value = value;
            return;
        }
    }

    if ((v = malloc(sizeof *v)) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    v->key = key;
    v->value = value;
    v->next = m->list;
    m->list = v;
    m->size++;
}

int
size(map *m) {
    return m->size;
}

char **keys(map *m) {
    struct map_node *v;
    int  i, n_keys = size(m);
    char **key_array;

    if ((key_array = malloc(n_keys*sizeof (char **))) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    for (v = m->list, i=0; v != NULL; v = v->next,i++)
        key_array[i] = v->key;
    return key_array;
}

static void
free_map_nodes(struct map_node  *list) {
    if (list == NULL)
        return;
    free_map_nodes(list->next);
    free(list);
}

void free_map(map *m) {
    free_map_nodes(m->list);
    free(m);
}
/*
 * One word_count struct is malloc'ed for each
 * distinct word read
 */
struct word_count {
    int count;
};

/*
 * read the words from a stream
 * associate a word_count struct with
 * each new word
 *
 * increment the count field each time the
 * word is seen
 */
map *
read_words(FILE *stream) {
    char *word;
    struct word_count *w;
    map *m;

    m = create_map();
    while (1) {
        word = get_word(stdin);
        if (word == NULL)
            return m;
        w = get(m, word);
        if (w != NULL) {
            w->count++;
            free(word);
            continue;
        }
        if ((w = malloc(sizeof *w)) == NULL) {
            fprintf(stderr, "Out of memory\n");
            exit(1);
        }
        w->count = 1;
        put(m, word, w);
    }
}

void
sort_words(char **sequence, int length) {
    int i, j;
    char *pivotValue;
    char *temp;

    if (length <= 1)
        return;

    /* start from left and right ends */

    i = 0;
    j = length - 1;

    /* use middle value as pivot */

    pivotValue = sequence[length/2];
    while (i < j) {
        /* Find two out-of-place elements */

        while (strcmp(sequence[i], pivotValue) < 0)
            i++;
        while (strcmp(sequence[j], pivotValue) > 0)
           j--;
        /* and swap them over */

        if (i <= j) {
            temp = sequence[i];
            sequence[i] = sequence[j];
            sequence[j] = temp;
            i++;
            j--;
        }
    }
    sort_words(sequence, j + 1);
    sort_words(sequence+i, length - i);
}
int
main(int argc, char *argv[]) {
    int i, n_unique_words;
    char **key_array;
    map *m;

    m = read_words(stdin);

    key_array = (char **)keys(m);

    n_unique_words = size(m);

    sort_words(key_array, n_unique_words);

    for (i = 0; i < n_unique_words; i++) {
        struct word_count *w;
        w = (struct word_count *)get(m, key_array[i]);
        printf("%5d %s\n", w->count, key_array[i]);
    }

    return 0;
}
```

word_frequency1.c

```
word_frequency0.c  with linked list replaced by binary tree - much faster:


% gcc -O3 word_frequency1.c -o word_frequency1
% time word_frequency1 <WarAndPeace.txt >/dev/null
real    0m0.277s
user    0m0.268s
sys     0m0.008s
```

```c
#include <stdlib.h>
#include "time.h"
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*
 * returns the next word from the streeam
 * a word is a non-zero length sequence of
 * alphabetic characters
 *
 * NULL is returned if there are no more words to be read
 */
char *
get_word(FILE *stream) {
    int i, c;
    char *w;
    static char *buffer = NULL;
    static int buffer_length = 0;

    if (buffer == NULL) {
        buffer_length = 32;
        buffer = malloc(buffer_length*sizeof (char));
        if (buffer == NULL) {
            fprintf(stderr, "out of memory\n");
            exit(1);
        }
    }

    i = 0;
    while ((c = fgetc(stream)) != EOF) {
        if (!isalpha(c) && i == 0)
            continue;
        if (!isalpha(c))
            break;
        if (i >= buffer_length) {
            buffer_length += 16;
            buffer = realloc(buffer, buffer_length*sizeof (char));
            if (buffer == NULL) {
                fprintf(stderr, "out of memory\n");
                exit(1);
            }
        }
        buffer[i++] = c;
    }

    if (i == 0)
        return NULL;

    buffer[i] = '\0';

    w = malloc(strlen(buffer) + 1);
    if (w == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    strcpy(w, buffer);
    return w;
}

typedef struct map  map;

struct map {
       int              size;
       struct map_tnode *tree;
};
```

```c
struct map_tnode {
        char            *key;
        void            *value;
        struct map_tnode *smaller;
        struct map_tnode *larger;
};

map *
create_map() {
    struct map *m;
    if ((m = malloc(sizeof *m)) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    m->size = 0;
    m->tree = NULL;
    return m;
}

/*
 * Return the value associated with key in map m.
 */
static void *
get_tree(struct map_tnode *t, char *key) {
    int compare;
    if (t == NULL)
        return NULL;
    compare = strcmp(key, t->key);
    if (compare == 0)
        return t->value;
    else if (compare < 0)
        return get_tree(t->smaller, key);
    else
        return get_tree(t->larger, key);
}

void *get(map *m, char *key) {
    return  get_tree(m->tree, key);
}

/*
 * Return the value associated with key in map m.
 */
struct map_tnode *
put_tree(struct map_tnode *t, char *key, void *value, map *m) {
    int compare;
    if (t == NULL) {
        if ((t = malloc(sizeof *t)) == NULL) {
            fprintf(stderr, "Out of memory\n");
            exit(1);
        }
        t->key = key;
        t->value = value;
        t->smaller = NULL;
        t->larger = NULL;
        m->size++;
        return t;
    }

    compare = strcmp(key, t->key);
    if (compare == 0) {
        t->value = value;
    } else if (compare < 0)
        t->smaller = put_tree(t->smaller, key, value, m);
    else
        t->larger = put_tree(t->larger, key, value, m);
    return t;
}

void
put(map *m, char *key, void *value) {
    m->tree = put_tree(m->tree, key, value, m);
}

int
size(map *m) {
    return m->size;
}

static int
tree_to_array(struct map_tnode *t, char **key_array, int index) {
    if (t == NULL)
        return index;
    index = tree_to_array(t->smaller, key_array, index);
    key_array[index] = t->key;
    return tree_to_array(t->larger, key_array, index + 1);
}

char **keys(map *m) {
    char **key_array;

    if ((key_array = malloc(size(m)*sizeof (char **))) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    tree_to_array(m->tree, key_array, 0);
    return key_array;
}

static void
free_tnodes(struct map_tnode  *t) {
    if (t == NULL)
        return;
    free_tnodes(t->smaller);
    free_tnodes(t->larger);
    free(t);
}

void free_map(map *m) {
    free_tnodes(m->tree);
    free(m);
}

/*
 * One word_count struct is malloc'ed for each
 * distinct word read
 */
struct word_count {
    int count;
};

/*
 * read the words from a stream
 * associate a word_count struct with
 * each new word
 *
 * increment the count field each time the
 * word is seen
 */
map *
read_words(FILE *stream) {
    char *word;
    struct word_count *w;
    map *m;

    m = create_map();
    while (1) {
        word = get_word(stdin);
        if (word == NULL)
            return m;
        w = get(m, word);
        if (w != NULL) {
            w->count++;
            free(word);
            continue;
        }
        if ((w = malloc(sizeof *w)) == NULL) {
            fprintf(stderr, "Out of memory\n");
            exit(1);
        }
        w->count = 1;
        put(m, word, w);
    }
}

void
sort_words(char **sequence, int length) {
    int i, j;
    char *pivotValue;
    char *temp;

    if (length <= 1)
        return;

    /* start from left and right ends */

    i = 0;
    j = length - 1;

    /* use middle value as pivot */

    pivotValue = sequence[length/2];
    while (i < j) {

        /* Find two out-of-place elements */

        while (strcmp(sequence[i], pivotValue) < 0)
            i++;
        while (strcmp(sequence[j], pivotValue) > 0)
           j--;
        /* and swap them over */

        if (i <= j) {
            temp = sequence[i];
            sequence[i] = sequence[j];
            sequence[j] = temp;
            i++;
            j--;
        }
    }
    sort_words(sequence, j + 1);
    sort_words(sequence+i, length - i);
}
int
main(int argc, char *argv[]) {
    int i, n_unique_words;
    char **key_array;
    map *m;

    m = read_words(stdin);

    key_array = (char **)keys(m);

    n_unique_words = size(m);

    sort_words(key_array, n_unique_words);

    for (i = 0; i < n_unique_words; i++) {
        struct word_count *w;
        w = (struct word_count *)get(m, key_array[i]);
        printf("%5d %s\n", w->count, key_array[i]);
    }
    return 0;
}
```

word_frequency.pl

```
while ($line = <>) {
    $line =~ tr/A-Z/a-z/;
    foreach $word ($line =~ /[a-z]+/g) {
        $count{$word}++;
    }
}

@words = keys %count;

@sorted_words = sort {$count{$a} <=> $count{$b}} @words;

foreach $word (@sorted_words) {
    printf "%8d %s\n", $count{$word}, $word;
}
```

word_frequency.py

```
import fileinput,re, collections

count = collections.defaultdict(int)
for line in fileinput.input():
    for word in re.findall(r'\w+', line.lower()):
        count[word] += 1

words = count.keys()

sorted_words = sorted(words,  key=lambda w: count[w])

for word in sorted_words:
    print("%8d %s" % (count[word], word))
```

word_frequency.sh

```
tr -c a-zA-Z ' '|
tr ' ' '\n'|
tr A-Z a-z|
egrep -v '^$'|
sort|
uniq -c
```

fib0.c

```
#include <stdio.h>
#include <stdlib.h>

int fib(int n) {
    if (n < 3) return 1;
    return fib(n-1) + fib(n-2);
}

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        int n = atoi(argv[i]);
        printf("fib(%d) = %d\n", n, fib(n));
    }
    return 0;
}
```

fib0.pl

```
sub fib {
    my ($n) = @_;
    return 1 if $n < 3;
    return fib($n-1) + fib($n-2);
}
printf "fib(%d) = %d\n", $_, fib($_) foreach @ARGV;
```

fib1.pl

```perl
sub fib {
    my ($n) = @_;
    return 1 if $n < 3;
    return $fib{$n} || ($fib{$n} = fib($n-1) + fib($n-2));
}
printf "fib(%d) = %d\n", $_, fib($_) foreach @ARGV;
```

fib2.pl

```perl
use Memoize;
sub fib($);
memoize('fib');
printf "fib(%d) = %d\n", $_, fib($_) foreach @ARGV;
sub fib($) {
    my ($n) = @_;
    return 1 if $n < 3;
    return fib($n-1) + fib($n-2);
}
```