The Problem

Given n >= 1 dictionary files and m >= 1 words, report for each word which dictionary its in

```
input: ./whichdictionary a b -- elbow
-> hello          not in dictionary
-> elbow          a:1
```

a	b
elbow knee sphincter	sameer

Solution A: Make array of words

This would use n arrays (1 per dictionary).

Solution B: Make one array for all words

A single array can hold all words using pointers to refer to relevant pieces of information. Each element in the array will hold three pieces of information:

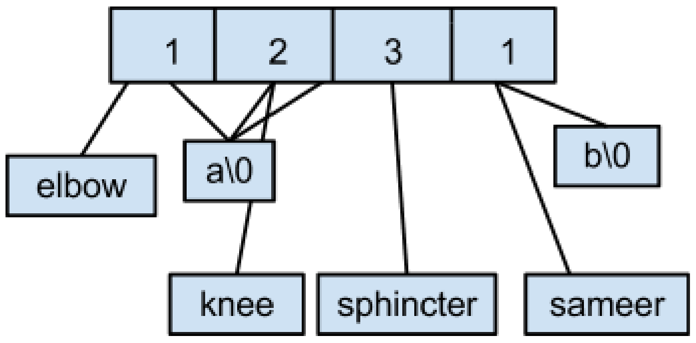
1) line number and
2) which dictionary it is in
3) the word itself

For the example above, the information in each element of the array would be:

elbow 1 a	knee 2 a	sphincter 3 a	sameer 1 b	

This can be accomplished by pointers. Strings for *a* and *b* referring to the dictionary can be shared by various elements, thus saving memory.

```
struct dict_word {
    char *word;
    char *file;
    int line;
}
```



Digression: How is memory allocated?

contiguous range of address - all addresses are next to each other
e.g. starts at 5, and goes to 6, 7, 8, 9

non-contiguous range of address - every string is located at a non-contiguous place
e.g. stored at 5, 17, 39

Sample Program

`whichdictionary.c` uses three pieces of info
- the word
- n words
- capacity

Dictionary_read_file iterates through the dictionary, finds words that match, then returns the pointer to an element of an array.

Digression: What to do when you don't know what a program does?

You can go to Piazza like so: https://piazza.com/class#fall2012/cs61/122.

Or you can use `man name_of_function`.

For example, `man strcmp` shows you that it compares two strings, returns an integer that determines if it is greater than, equal to, or less than each other. **Important:** to use `strcmp` you must include a library file!

Back to the Problem - The Dictionary has issues!

When printing out dictionary words, all words were "nine".
Upon printing dictionary pointers, all addresses were the same, "0xbfabf18c".
This is because we stored the pointer to the buf (buffer) variable, we exposed a pointer that points to invalid memory.
Nine was stored in the stack, which has automatic storage duration.
Meanwhile the array had dynamic storage duration.
Thus the stack no longer has the memory of the word!

A test
Changing a letter in a word that no longer exists in the stack exposes pointer to invalid memory allows injection of random functions, like Hello Kitty vampire.

Fix
Point the stack to heap, not other way around! Thus, malloc space for each word in the dictionary.

To store strings in the heap, we write:
`malloc(sizeof(char)*(len+1));`
This will give us enough space in the heap for our string and its terminating character.

Some pset work
Counting the mallocs
- int total_alloc as a global, increase it every time malloc'd
- int active_alloc as a global, increase it every time malloc'd, decrement active on free

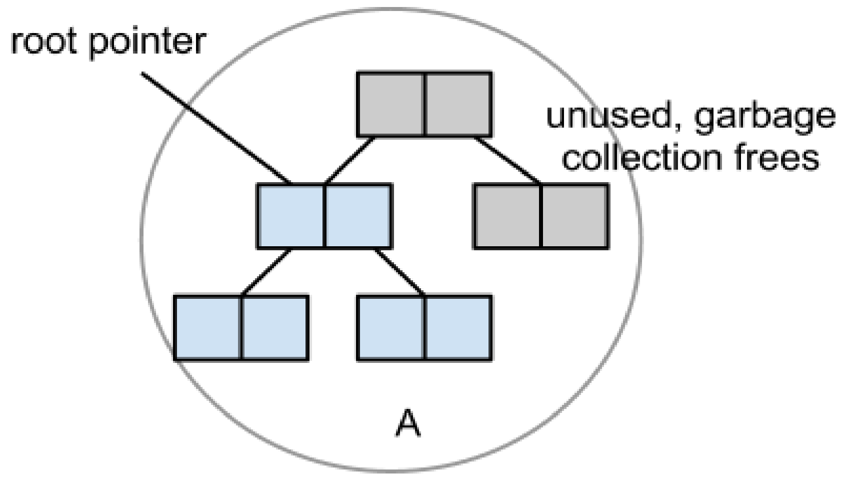
The count reveals that there is a memory leak whichdictionary.c, caused by us not freeing every word in the dictionary.
Solution: write a loop that iterates through the dictionary and frees every word.

The power of dynamic memory allocation
C
- explicit dynamic memory allocation (user calls malloc, free)

Jaca, Ruby, Perl, Lisp, APL
- implicit dynamic memory allocation with garbage collection
- will find leftover malloc'd and automatically frees
- figures out what pointers are in use and frees the ones that aren't

Garbage Collection
Given a set of allocations A and one or more root pointers R as a subset of A, tries to find the smallest set A' with R as a subset of A' as a subset of A, so that all pointers in A' point within A'.

Example: Imagine dynamically allocated nodes on a tree called Set A. A root pointer stored elsewhere points to an element in this tree. The root pointer could point to the root node, or it could point to a node on a branch.



If the root pointer only points to a branch, then the other branches are not accessible by the program. Garbage collection will identify the top and the right branch as inaccessible, freeing those nodes.

Goal: Garbage collection finds nodes that are not used and frees them.

How can we write a Garbage Collection in C to help us in our dictionary problem?

We need A, a set of roots. We need B, a set of **chasing pointers** that is contained within the allocation. The roots are all the pointers in static and automatic storage, or everything on the stack.

How do we make sure a pointer works? We can chase pointers by pulling four bytes at a time that represent a pointer, ask “does this look like a pointer?” and if so, treat it like a pointer.

Side note: A conservative garbage collector preserves all the pointers but also preserves some other things that aren’t pointers.

Writing a garbage collector

A: The roots.
What range of addresses are needed for roots?
- bottom - address of stack in main
- top - address of the function declare a local variable in the function, and the variable’s function will be the address of that function

B: Chase pointers
Now, to find all pointers from that range:
- extract pointer-sized regions from that range
- Some voodoo: `void *p = *(void **) ((char *) ptr + i);`

Generally, we’ve seen enough pointers that we can say:
bf0 - starts stack pointer

08 - starts heap pointer

To find pointers in a block of memory
- repeatedly read 4 blocks of memory looking for a value that looks like a pointer - shift 1 byte at a time
- when finding pointer, save that block of memory

Side Notes

longest word: an invented fish dish
smashing the stack - hackers use this method to take advantage of memory faults to insert executable