CS50 Section. Week 7. 10/20/15.

Tuesdays 7:00-8:30pm, Science Center 309A
https://github.com/hathix/cs50-section

Neel Mehta
neelmehta@college.harvard.edu
(215) 990-6434

Get these handouts at https://github.com/hathix/cs50-section/tree/master/handouts.

pset5

USE HASHTABLES, NOT TRIES! I tried using a trie last year (pun intended) and it was really painful. Hashtables are way easier and tend to run faster.

Choosing a good hash function

You'll need a hash function for `load` and `check` -- something to assign words to buckets in your hashtable. Some specifications:

- It must have a header like this: `unsigned int hash(char* word)`
- It must return an index that's less than the number of buckets, `BUCKETS` (mod `%` by `BUCKETS` to ensure this)

Don't try writing your own hash function... just adapt one off the internet. Some places you can look:

- My compilation of hash functions from around the internet: https://github.com/hathix/cs50-section/blob/master/code/7/sample-hash-functions
- Random Stack Overflow questions like http://stackoverflow.com/q/7666509 (search for "string hash functions for c")

Random advice gathered from past TFs (might be a little advanced)

- Use `#define BUCKETS` for the number of buckets in your hashtable
- Check for `NULL` when you're using functions that might fail (e.g. `fopen`, `malloc`)
- Always `fclose` your files
- Run valgrind with `show-reachable=yes`
- Use `mmap` and `calloc` when allocating memory (better than plain ol' `malloc`)
- Avoid using `strlen` when possible because it's slow; if you're using a loop try to just compare each character to `\0`
- Eliminate function calls when possible because they take up lots of time

Linked Lists

Linked lists are a data structure that let you keep as many elements as you want in a list. Each element (or *node*) just points to the next node.

Solutions to all these challenges at https://github.com/hathix/cs50-section/blob/master/code/7/linked-soln.c

Create

Define a **node struct** to represent each item in the linked list:

```
typedef struct node
{
    // the value to store in this node
    int n;

    // the link to the next node in the list
    struct node* next;
}
node;
```

Then, to **create the first element** in the linked list:

```
node* head = malloc(sizeof(node));

// error checking: quit if malloc fails
if (head == NULL)
{
    exit(1);
}

// initialize fields in linked list
head->n = 50;
head->next = NULL;
```

Iterate (length)

Here's a good formula for **iterating through a linked list**:

```
// `ptr` iterates over every node starting at `head`
for (node* ptr = head; ptr != NULL; ptr = ptr->next)
{
    // do something
}
```

Challenge

```
/*
 * Returns the length of the linked list that starts at `head`.
 */
int length()
{
    // **YOUR CODE HERE**










}
```

Insert

Challenge

```
/**
 * Adds an element containing `i` to the front of the linked list that
 * starts at `head`.
 */
void prepend(int i)
{
    // **YOUR CODE HERE**














}
```

Search (contains)

Challenge

```
/**\
 * Returns true if the linked list that starts at `head` contains `needle`,
 * false otherwise.
 */
bool contains(int needle)
{
    // **YOUR CODE HERE**










}
```

Hashtables

Hashtables are just organized collections of linked lists, so once you understand linked lists, hashtables are actually pretty easy. You just assign every node to a linked list (a "bucket") and do the usual iterate/insert/search operations on the linked list.

Create

```
// the maximum length of a word in our hashtable
#define WORD_LENGTH 20

// number of buckets in our hash table
#define NUM_BUCKETS 100

/**
 * Each node will store an int and a pointer to the next node with the
 * same hash value (or NULL, if no such node follows).
 */
typedef struct node
{
    // what's the +1 for here?
    char word[WORD_LENGTH + 1];
    struct node* next;
}
node;


// a hashtable is an array of pointers: each pointer is the head of a linked
// list (aka bucket), and you have NUM_BUCKETS buckets stored in an array
node* hashtable[NUM_BUCKETS];
```

Hashing

To figure out what bucket a string belongs in, run it through a **hash function.** Hash functions turn strings into integers, where the integer indicates the index of the bucket. A good hash function will minimize "collisions" in a bucket (when ≥1 string falls into the same bucket) by trying to make each string correspond to a unique integer.

There are a ton of good pre-made hash functions; don't try to make your own. Check out a few I've compiled at https://github.com/hathix/cs50-section/blob/master/code/7/sample-hash-functions.

```
char* word = "chumbucket";
int bucket_index = hash_function(word);
```

Insert

Inserting into a hashtable is easy: figure out what bucket the node belongs in (use the hash function) and `prepend` the node to that linked list.

Lookup

Looking up is also easy: figure out what bucket the node belongs in (use the hash function) and `search` through that linked list's contents.