pset6: Mispellings

Tommy MacWilliam

tmacwilliam@cs50.net

October 23, 2011

Today's Music

- Epic Music
  - Don't Touch This (Busta Rhymes feat. Travis Barker)
  - Lux Aeterna (Clint Mansell)
  - Tapp (3OH!3)
  - 300 Violin Orchestra (Jorge Quintero)
  - Beaumont (3OH!3)

Today

- `speller.c`
- linked lists
- hash tables
- tries

speller.c

- calls `load()` on dictionary file
  - dictionary contains valid words, one per line
- iterates through words in file to spellcheck, calls `check()` on each word
- calls `size()` to determine number of words in dictionary
- calls `unload()` to free up memory

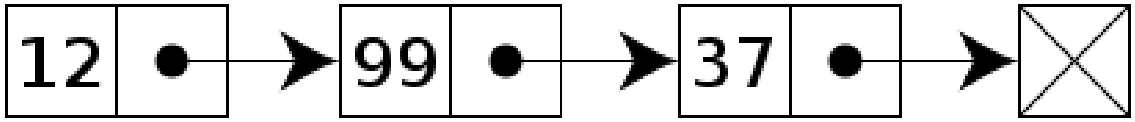
dictionary.c

- we must implement `load()`, `check()`, `size()`, and `unload()`
- high-level overview:
  - given a list of correctly-spelled words in a dictionary file, load them all into memory
  - for each word in some text, spell-check each word
  - if word from text is found in memory, it must be spelled correctly
  - if word from text is not found in memory, it cannot be spelled correctly

speller.c

- example time!

Linked Lists



- each node contains a value and a pointer to the next node
  - need to maintain a pointer to the first node
  - last node points to `NULL`

Creating Linked Lists


```
typedef struct node {
    char *word;
    struct node *next;
} node;
node *node1 = malloc(sizeof(node));
node *node2 = malloc(sizeof(node));
node1->word = "this";
node2->word = "is";
node1->next = node2;
node2->next = NULL;
```

Traversing Linked Lists

- create pointer to iterate through list, starting at first element
- loop until iterator is `NULL` (aka no more elements)
- at every point in loop, iterator will point at an element in the linked list
  - can access any element of the element
- to go to next element, simply move iterator to `next`

Traversing Linked Lists


```
// assuming first points to the first element
node *iterator = first;
while (iterator != NULL)
{
    printf("%s\n", iterator->word);
    iterator = iterator->next;
}
```

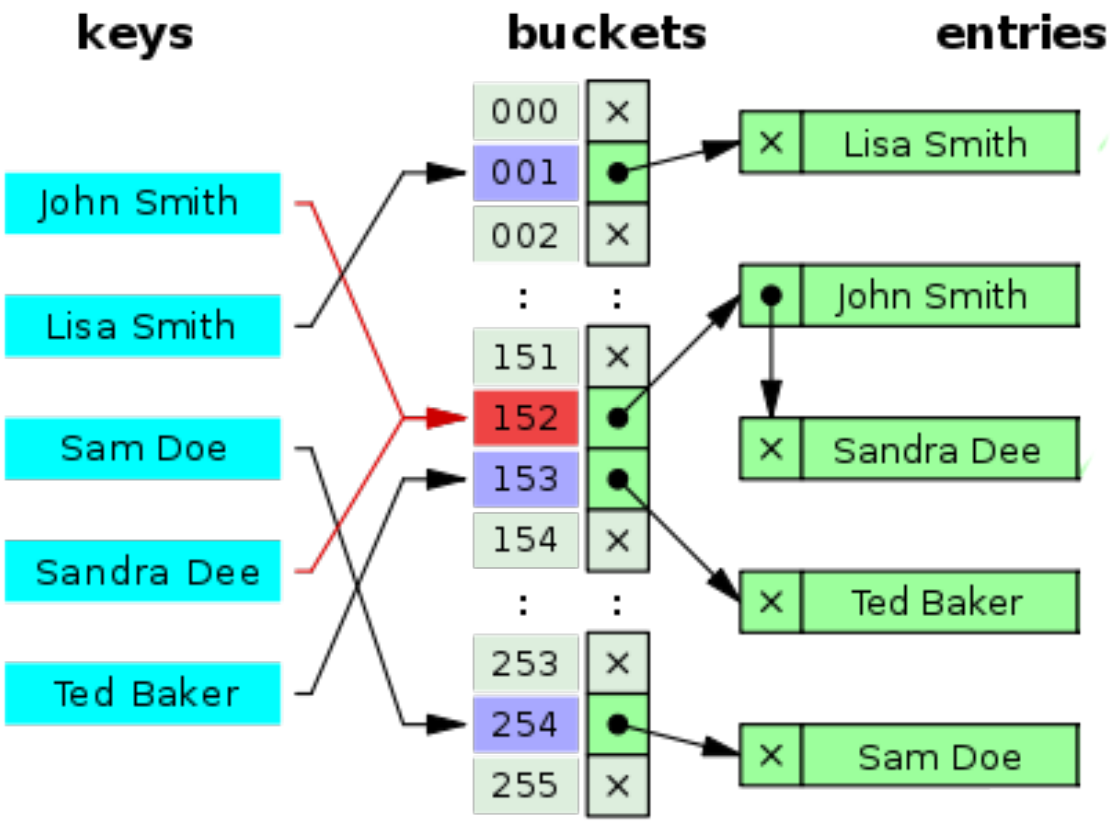
Freeing Linked Lists

- need to explicitly `free()` each element in the list
  - but, once you `free()`, you can't access `next` any more
  - determine `next` node, `free()` the current node, then move on to next node

Freeing Linked Lists


```
// assuming first points to the first element
node *iterator = first;
while (iterator != NULL)
{
    node *n = iterator;
    iterator = iterator->next;
    free(n);
}
```

Hash Tables



(image courtesy Wikipedia)

Hash Tables

- fixed number of buckets (aka an array)
- hash function maps each value to a bucket
  - must be deterministic: same value must map to same bucket every time

Hash Function

- outputs a bucket number for each input
- since each input is a word, need to convert a word to an integer
- also make sure integer is a valid bucket number
  - can't be larger than the number of buckets, which doesn't change

Best Hash Function Ever


```
int hash(char *name)
{
    if (strcmp(name, "John Smith") == 0)
        return 152;
    else if (strcmp(name, "Lisa Smith") == 0)
        return 1;
    else if (strcmp(name, "Sam Doe") == 0)
        return 254;
    else if (strcmp(name, "Sandra Dee") == 0)
        return 152;
    else
        return 153;
}
```

Still Not a Great Hash Function


```
// many words still have same hash value!
int hash(char *word)
{
    int result = 0;
    int n = strlen(word);
    for (int i = 0; i < n; i++)
    {
        result += word[i]
    }
    return result % HASHTABLE_SIZE;
}
```

Collisions

(image courtesy knowyourmeme.com)

Collisions

- what if two values map to the same bucket?
  - can't just wipe out the other value!
- hash table contains pointers to the start of linked lists instead of words
  - need to traverse every element of the linked list to look for word
  - still MUCH faster than linear searching entire dictionary for every word

Structure


```
typedef struct node {
    char word[LENGTH + 1];
    struct node *next;
} node;

node *hashtable[HASHTABLE_SIZE];

// hashtable[i] is a pointer to the
// start of a linked list of all words
// that hash to i
```

Reading the Dictionary

- goal: load every word in the dictionary into memory somehow
- need to iterate over each word in dictionary text file
  - iterate over text file with `while (!feof(fp))`
  - each word must be individually inserted into the hash table

Creating Nodes

- `malloc` a new `node* n` for each word
- use `fscanf` to read string from file
  - `fscanf(fp, "%s", n->word);`
  - reads one word from dictionary at a time

Hashing

- now, `n->word` contains the word from the dictionary
  - now we can hash `n->word`, since our hash function converts strings to integers
- result of hash function gives bucket in hash table for node
  - remember, hash table contains pointers to the start of linked lists

Inserting Nodes

- `hashtable[index] == NULL`
  - no linked list exists yet
  - make `hashtable[index]` point to `n`
  - make `n->next` point to `NULL` because it is the last element in the linked list

Inserting Nodes

- `hashtable[index] != NULL`
  - linked list exists already, so add to the beginning of it
    - adding to the end is much slower!
  - make `n->next` point to what is already there
  - make `hashtable[index]` point to `n`

Size

- `size()` returns the number of words in the dictionary
  - aka the sum of the number of nodes in your hash table
- just keep a counter as you're loading words!

Check

- goal: given some word, check if it is in the dictionary
- if word exists in our hash table, it must be spelled correctly
  - if it does not exist in our hash table, it cannot be spelled correctly

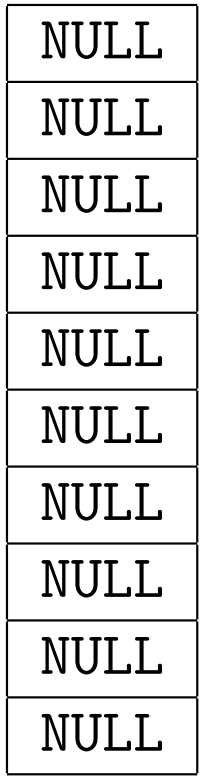
Check

- don't need to search entire hash table for word
  - only need to search linked list starting at `hash(word);`
- linked list to traverse starts at `hashtable[hash(word)];`

Check

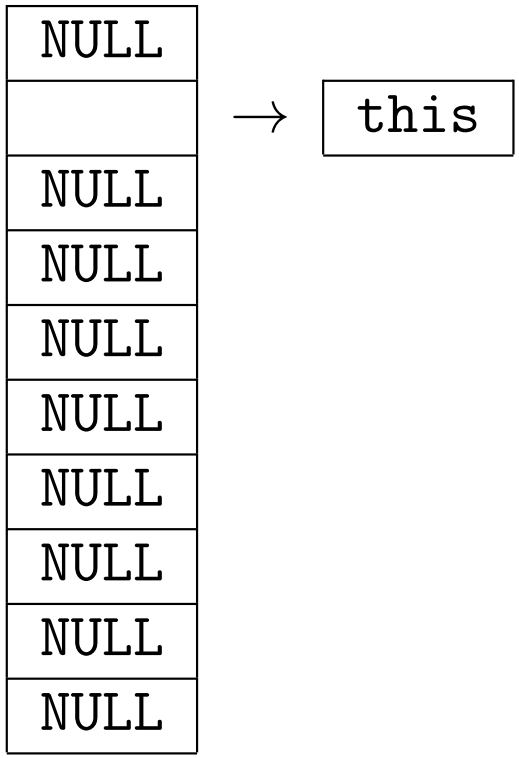
- we already know how to traverse a linked list!
- at each node in linked list, compare `word` to input
  - `strcmp(string1, string2)`: returns `0` if `string1` and `string2` are equal
  - still, spell-checker needs to be case-insensitive!
- if strings are equal, word is spelled correctly
- if end of linked list is reached, word is not spelled correctly

Example

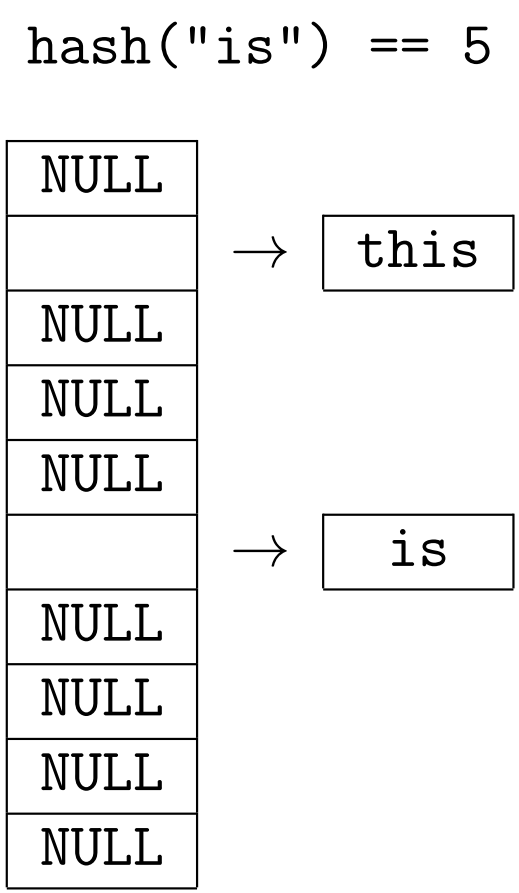


Load


```
hash("this") == 1
```

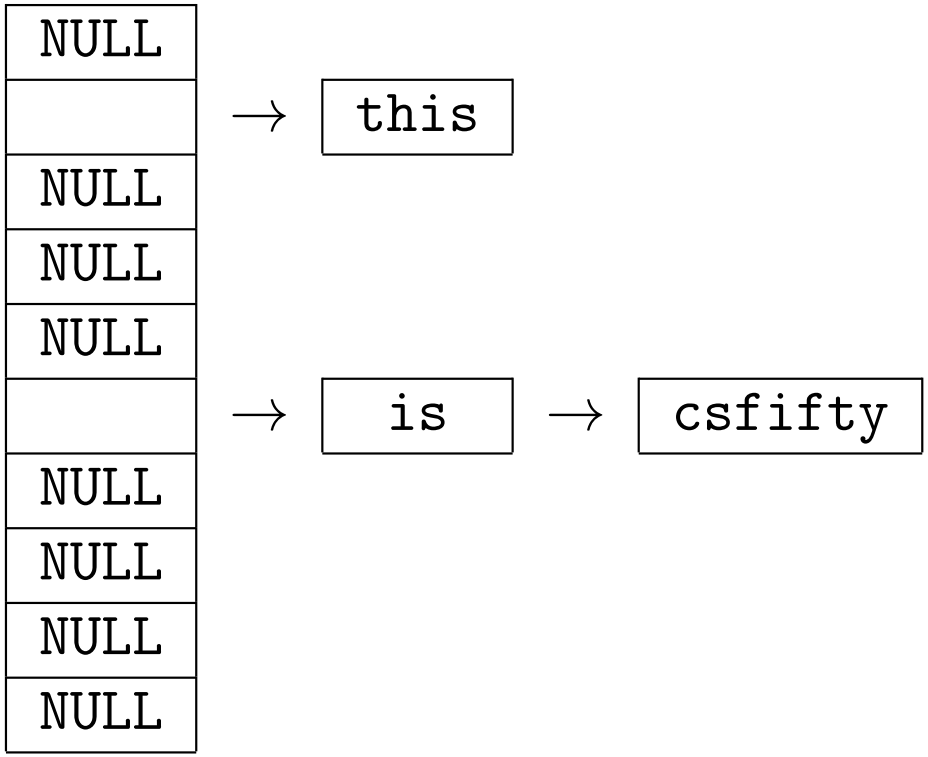


Load



Load

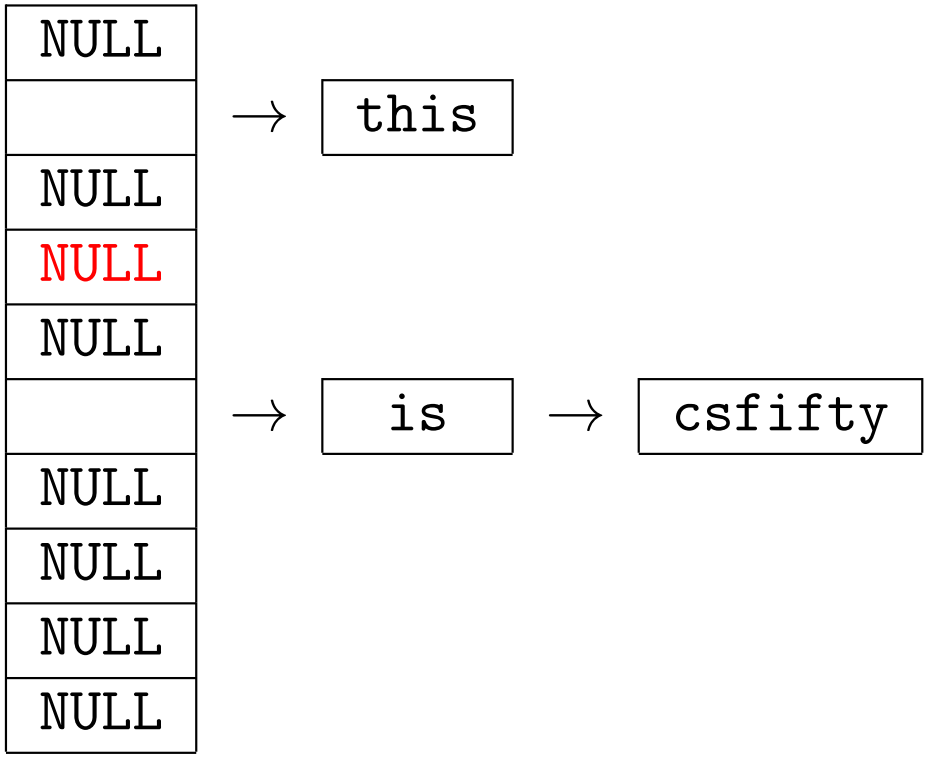

```
hash("csfifty") == 5
```



Check


```
check("isn't");
```

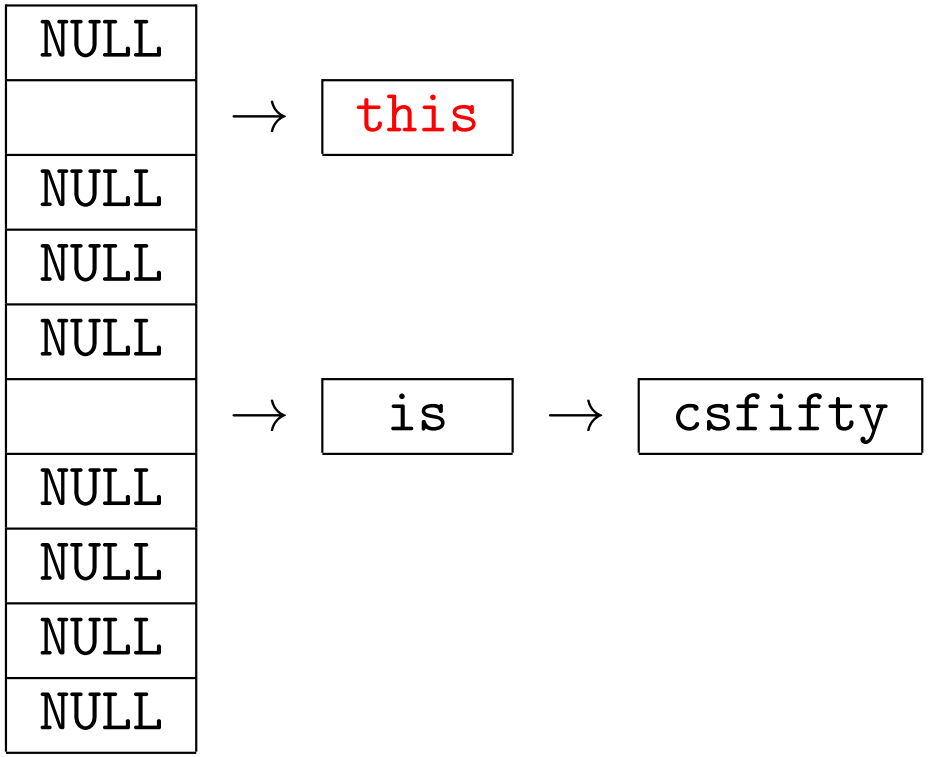
```
hash("isn't") == 3
```



Check


```
check("this");
```

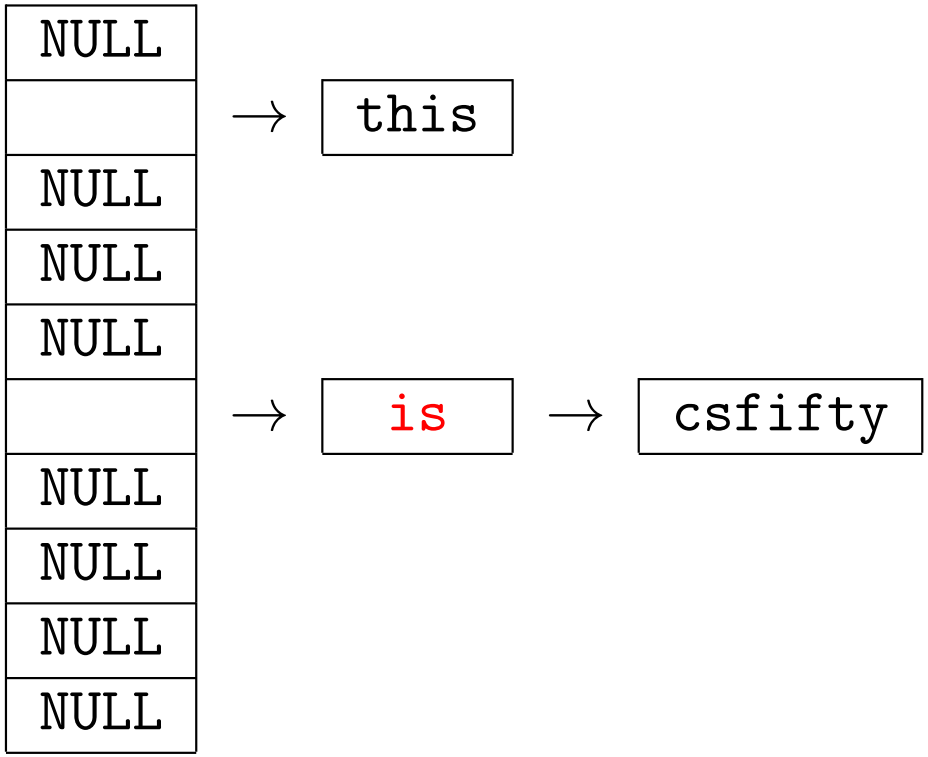
```
hash("this") == 1
```



Check


```
check("csfifty");
```

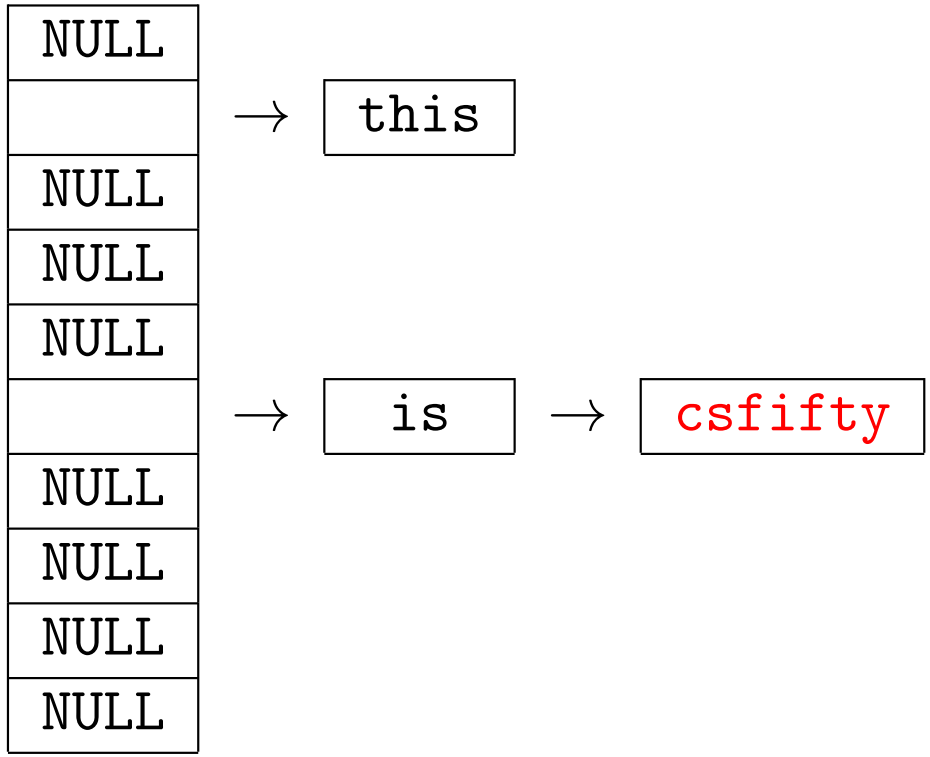
```
hash("csfifty") == 5
```



Check


```
check("csfifty");
```

```
hash("csfifty") == 5
```



Unload

- goal: `free()` entire hash table from memory
- array allocated with `node *array[LENGTH]` does not need to be freed
- anything `malloc`'d must be freed

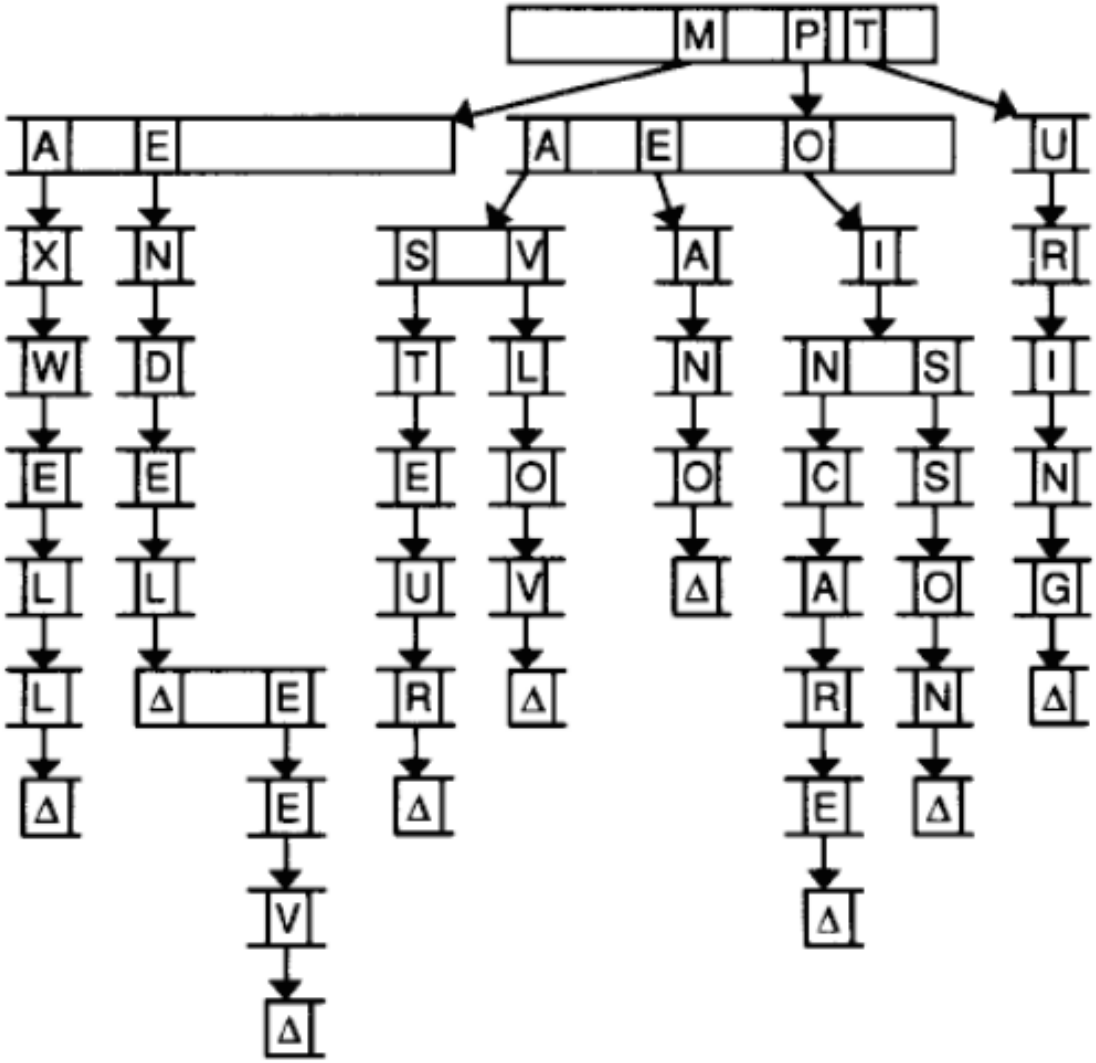
Unload


```
for each element in hashtable
    for each element in linked list
        free element
        move to next element
```

Valgrind

- `valgrind -v --leak-check=full ./speller ~cs50/pset6/texts/austinpowers.txt`
  - example time!

Tries



Tries

- rather than a single word, nodes contain an array with an element for each possible character
- value of element in array points to another node if corresponding letter is the next letter in any word
  - if corresponding letter is not the next letter of any word, that element is `NULL`
- also need to store if current node is the last character of any word

Structure


```
typedef struct node {
    bool is_word;
    struct node *children[27];
} node;
```

load

- iterate through letters in each dictionary word
  - also keep iterator to iterate through trie as you insert letters
- each element in `children` corresponds to a different letter
- look at value for `children` element corresponding to current letter
  - if `NULL`, `malloc` a new `node`, point to it, and move iterator to new node
  - if not `NULL`, simply move iterator to new node
- if letter is `'\n'`, mark node as valid end of word

size

- same thing, keep a counter as you load words!

check

- attempt to travel downwards in trie for each letter in input word
  - for each letter, go to the corresponding element in `children`
  - if `NULL`, word is misspelled
  - if not `NULL`, go to that pointer and move on to next letter
- if at end of word, check if this node marks the end of a word

unload

- unload nodes from bottom to top!
- travel to lowest possible node, then free all pointers in `children`
  - then, backtrack upwards, freeing all elements in each `children` array until you hit root node
- natural recursive implementation