pset 5: Mispellings

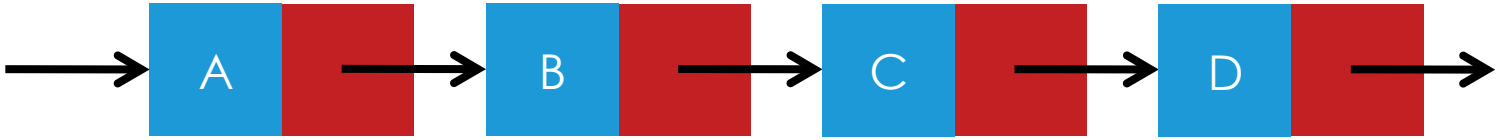
Zamyla Chan | zamyla@cs50.net

Toolbox

- distro code
- pset spec
- pointers
- linked lists
- hash tables
- tries
- valgrind
- pen and paper

linked lists

- Consist of nodes
- Each node has a value, as well as a pointer to the next node
- Important:
  - Where's the first node?
  - Last node points to NULL



linked lists

```
typedef struct node
{
    int value;
    struct node* next
}
node;

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

traversing linked lists

- Have a node pointer (ie, **`node*`**) to act as your cursor; start at first element
  - Let's call this `cursor`
- Loop until **`cursor`** is **`NULL`**
- How to access the value at the cursor?
  - `(*cursor).value`
  - **`cursor->value`**
- Update `cursor`
  - `cursor = (*cursor).next`
  - `cursor =` **`cursor->next`**

traversing linked lists

```
node* cursor = head;

while (cursor != NULL)
{
    // do something

    cursor = cursor->next;
}
```

freeing linked lists

- Remember to `free()` every element in the list once you're finished with the list
- Be careful! Avoid memory leaks

freeing linked lists

```
node* cursor = head;

while (cursor != NULL)
{
    node* temp = cursor;
    cursor = cursor->next;
    free(temp);
}
```

inserting into linked lists

- Tradeoff: sorted or unsorted?

Mispellings

0. A Section of Questions
1. Mispellings

Distribution Code

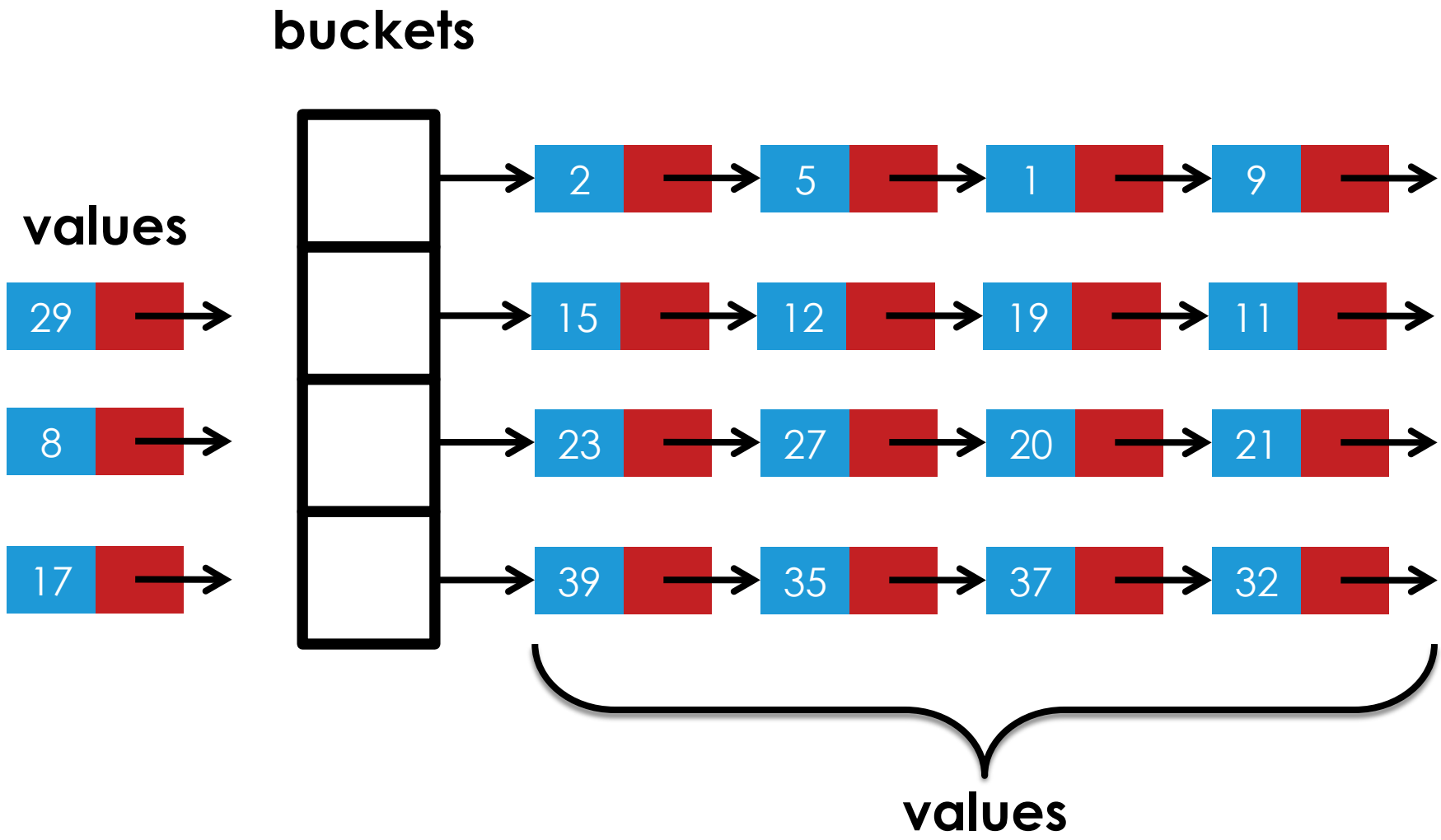
- `dictionary.h`
- `speller.c`
- `Makefile`

TODO: speller.c

- `load`
  - loads the dictionary
- `size`
  - returns the number of words in the dictionary
- `check`
  - checks if a given word is in the dictionary
- `unload`
  - frees the dictionary from memory

Hash Table Method

hash tables


buckets
values
29
8
17
2
5
1
9
15
12
19
11
23
27
20
21
39
35
37
32
values

hash tables

- an array of buckets
- each bucket is a linked list
- hash function
  - returns the bucket that a given key belongs to

hash tables

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

node* hashtable[HASHTABLE_SIZE];
```

hash functions

- deterministic
  - The same value needs to map to the same bucket every time
- returns an index
  - Number of buckets is fixed
  - Index < the number of buckets
- ideally, should be well-balanced
  - Each bucket has approximately the same number of keys
  - Tradeoff: # of collisions, size of hash table

`load`

- Iterate over each word in the dictionary
  - File I/O functions
- Each word should become a new node
  - `malloc` a `node*` for each new word
  - `node* new_node = malloc(...);`
- Read string from file
  - `fscanf(file, "%s", new_node->word)`
- Hash the word: insert into hash table

`load`: Hashing the word

- `new_node->word` contains the word from the dictionary
- calling our hash function on `new_node->word` will give us the index of a bucket in the hash table
- insert into the linked list
- remember
  - `hashtable[i]` is a `node*`
  - last node should point to `NULL`

`size`

Returns the number of words in the dictionary

check

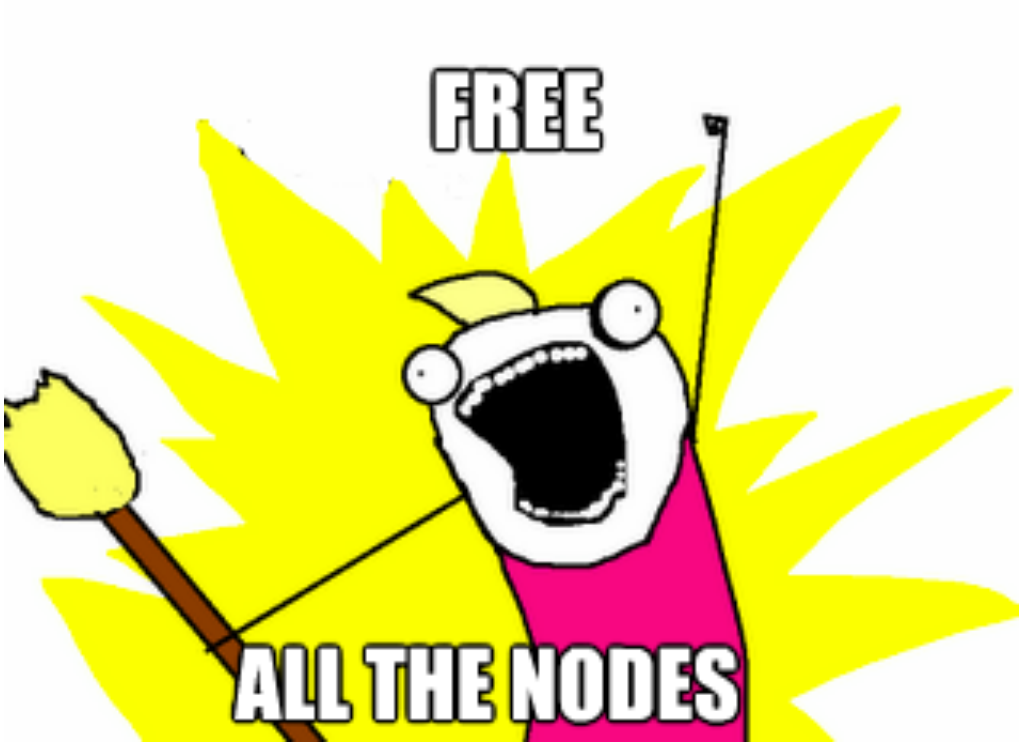
case-insensitivity

assume strings with only alphabetical characters and/or apostrophes

check

- If the word exists, it can be found in the hash table
- Hash the word to find what bucket it would be in: `hashtable[hash(word)]`
- Traverse the linked list
  - `strcmp`
  - if the end of the linked list of reached, word is not in dictionary

unload



`unload`

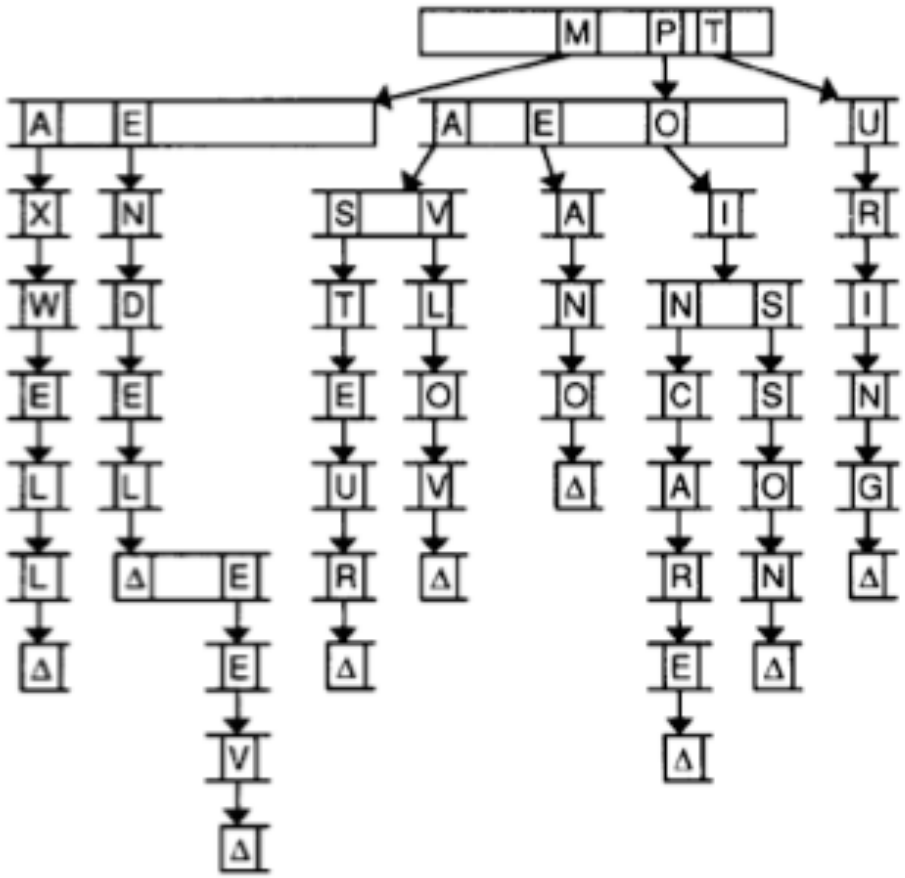
- `free()` anything you've `malloc`'d

for each element in hashtable
  for each node in the linked list
    free node

- `valgrind`

Tries Method

tries


M P T
A E
A E O
U
X N
S V
A
I
R
W D
T L
N
N S
I
E E
E O
O
C S
N
L L
U V
Δ
A O
G
L Δ E
R Δ
R N Δ
Δ
E Δ
E Δ
V
Δ

tries

- Every node contains an array of `node*`s
  - One for every letter in the alphabet + `'\''`
  - Each element in the array points to another `node`
    - if that `node` is `NULL`, then that letter isn't the next letter of any word in that sequence
- Every node indicates whether it's the last character of a word

tries

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

node* root;
```

`load`

Put each word in the dictionary

`load`

- For every dictionary word, iterate through the trie
- Each element in `children` corresponds to a different letter
- Check the value at `children[i]`
  - if `NULL`, `malloc` a new node, have `children[i]` point to it
  - if not `NULL`, move to new node and continue
- If at end of word, set `is_word` to `true`

`size`

Returns the number of words in the dictionary

check

case-insensitivity

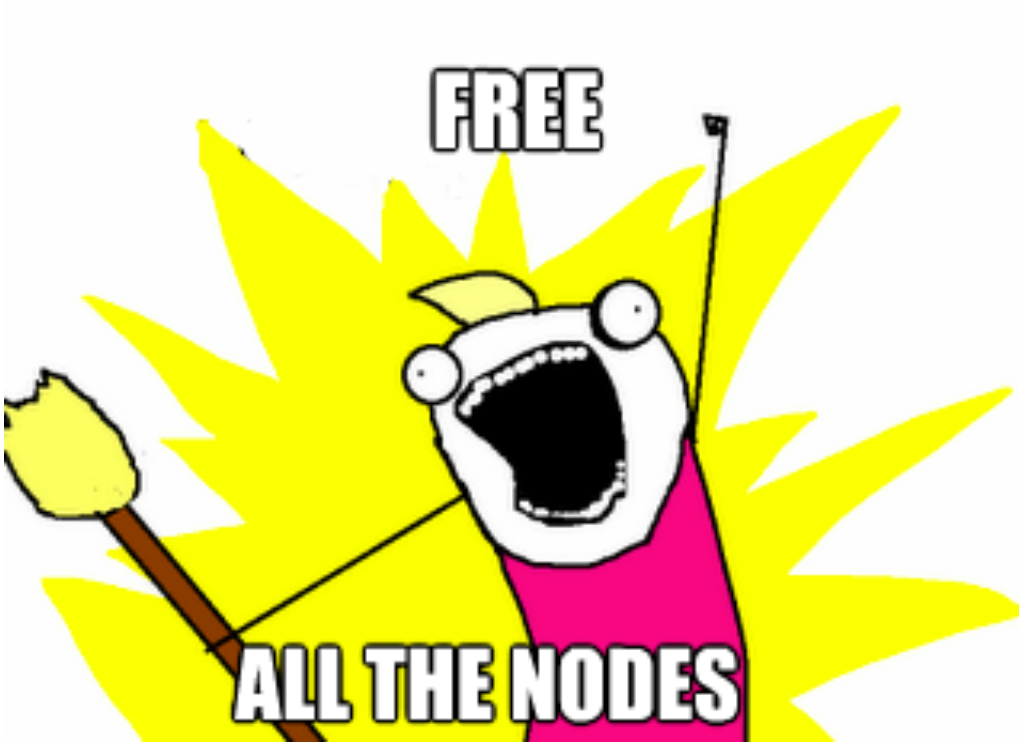
assume strings with only alphabetical characters and/or apostrophes

check

- Travel downwards in trie

for each letter in input word
  go to corresponding element in `children`
    if NULL, word is misspelled
    if not NULL, move to next letter
  once at end of input word
    check if is_word is true

unload



`unload`

- Unload from bottom to top
- Travel to lowest possible node
  - free all pointers in `children`
  - backtrack upwards, freeing all elements in each `children` array until you hit root node
- Recursion!

Valgrind

```
valgrind -v --leak-check=full ./speller
  ~cs50/pset5/texts/austinpowers.txt
```

The Big Board!

```
~cs50/pset5/challenge ~/Dropbox/pset5
```

Tips

- Pass in a smaller dictionary
  - usage: `./speller [dictionary] text`
  - default: `~cs50/pset5/dictionaries/`**`large`**
  - `~cs50/pset5/dictionaries/`**`small`**
  - make your own!
- Pen and paper!

this was walkthrough 5