

mispellings

speller.c

1. calls `load` on the dictionary file
 - ▣ dictionary contains valid words, one per line
2. calls `check` on each word in the text file and prints all misspelled words
3. calls `size` to determine number of words in dictionary
4. calls `unload` to free up memory

TODO

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

TODO

- load
- check
- size
- unload

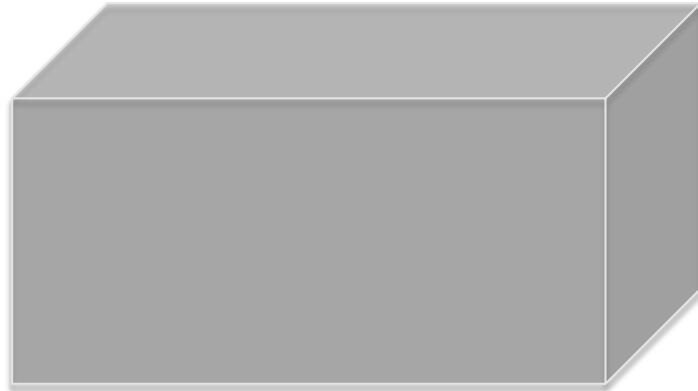
load

- for each word in the dictionary text file, store it in the dictionary's data structure
 - linked lists
 - hash tables
 - tries

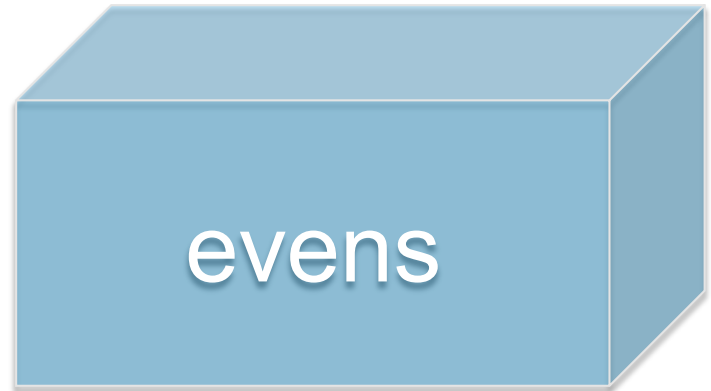
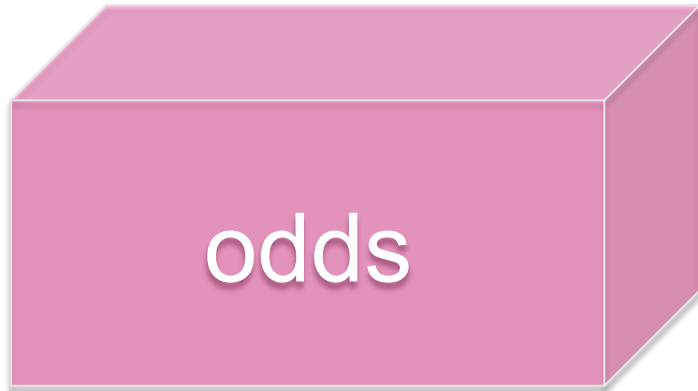
hash tables

- an array of buckets
- hash function
 - ▣ returns the bucket that a given key belongs to

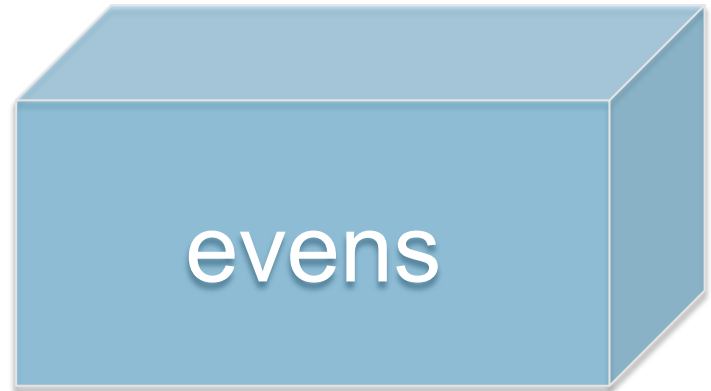
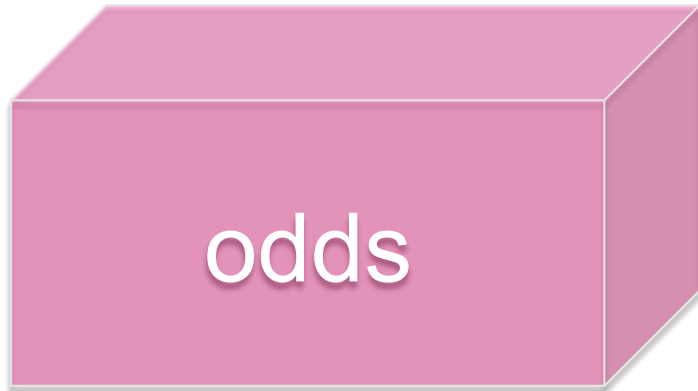
hash tables



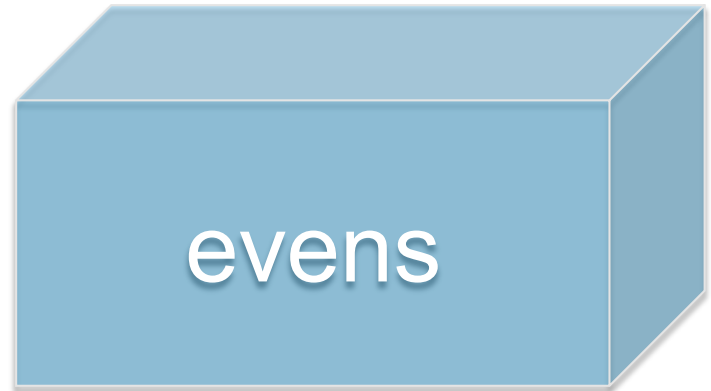
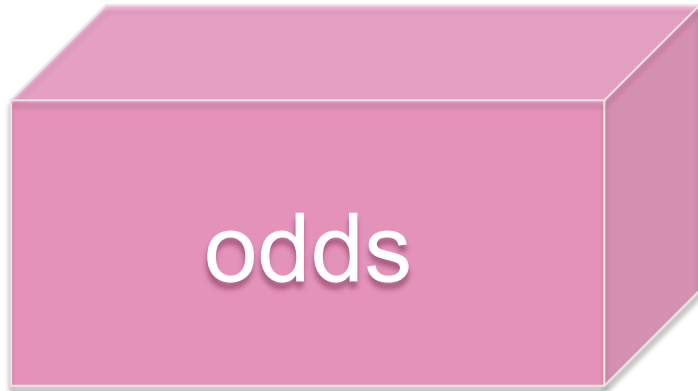
hash tables



hash tables

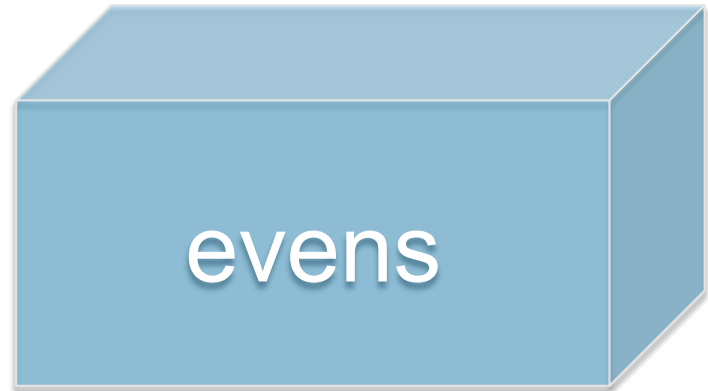
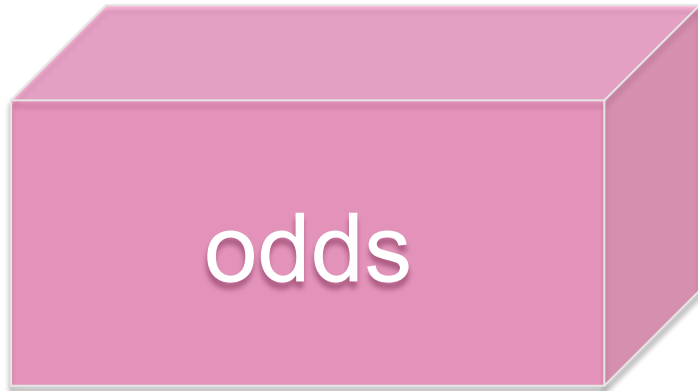


hash tables

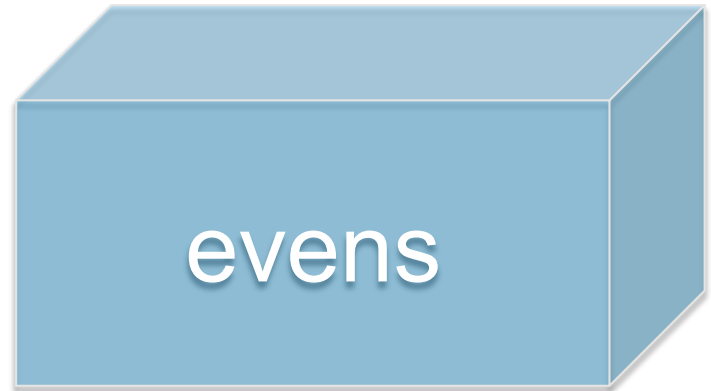
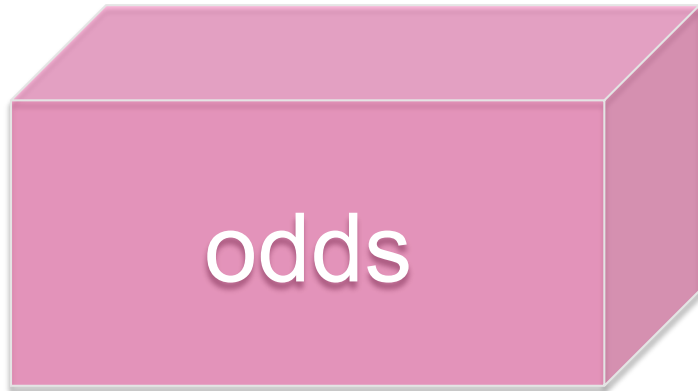
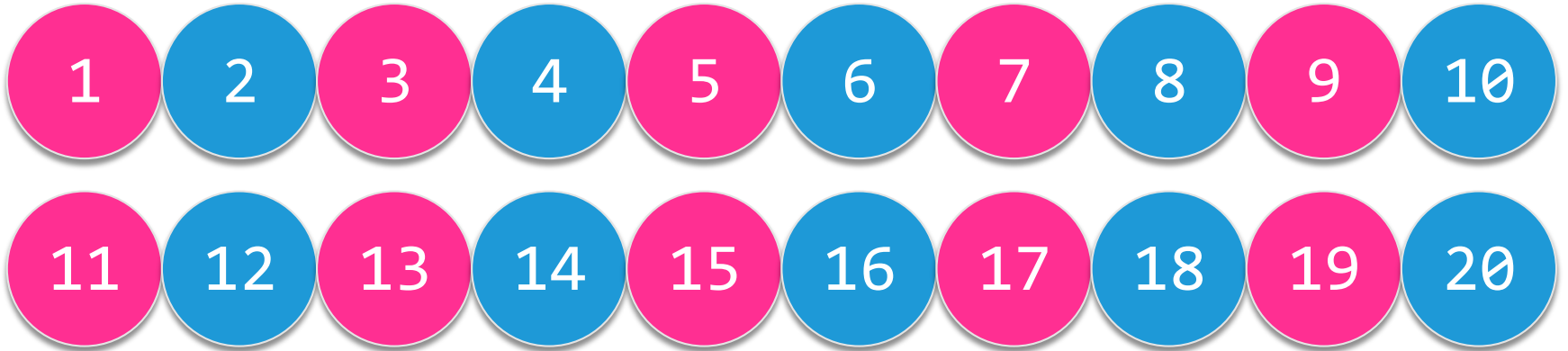


hash tables

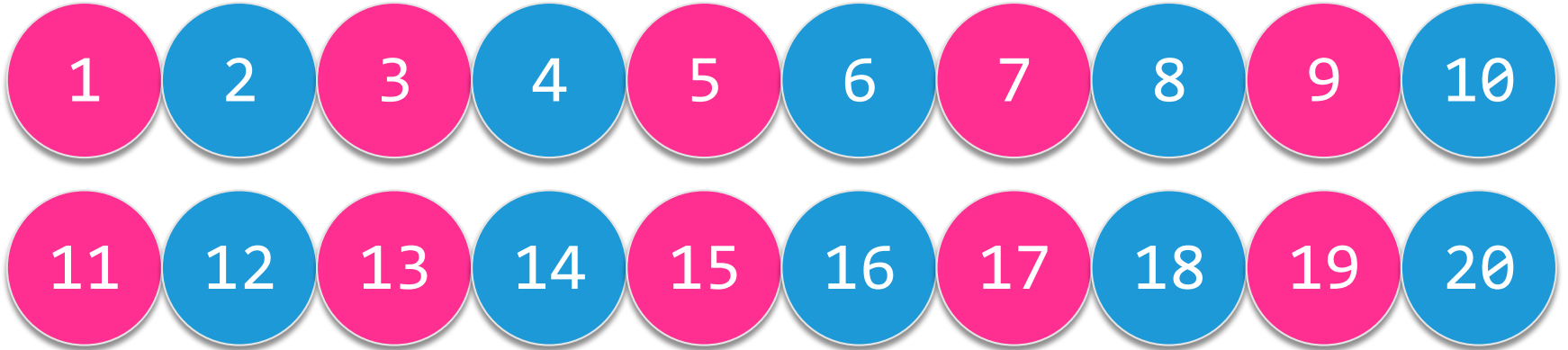
- hash table: 2 buckets
- hash function: `if (n % 2 == 1), odd box`
`else, even box`



hash tables



hash tables



1 - 5

6 - 10

11 - 15

16 - 20

hash tables



1 - 5

6 - 10

11 - 15

16 - 20

hash tables

- a hash table is an array of buckets
- each bucket is a linked list

a hash table is
an array of linked lists

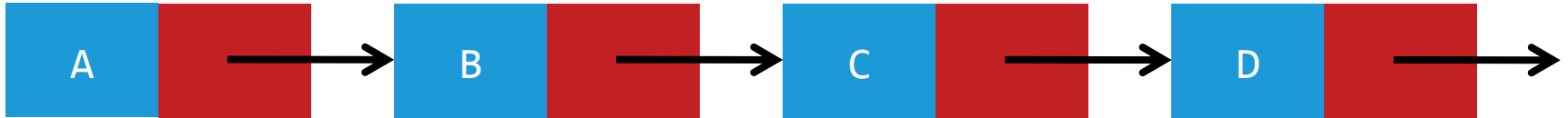
nodes

- each node has a value, as well as a pointer to the next node



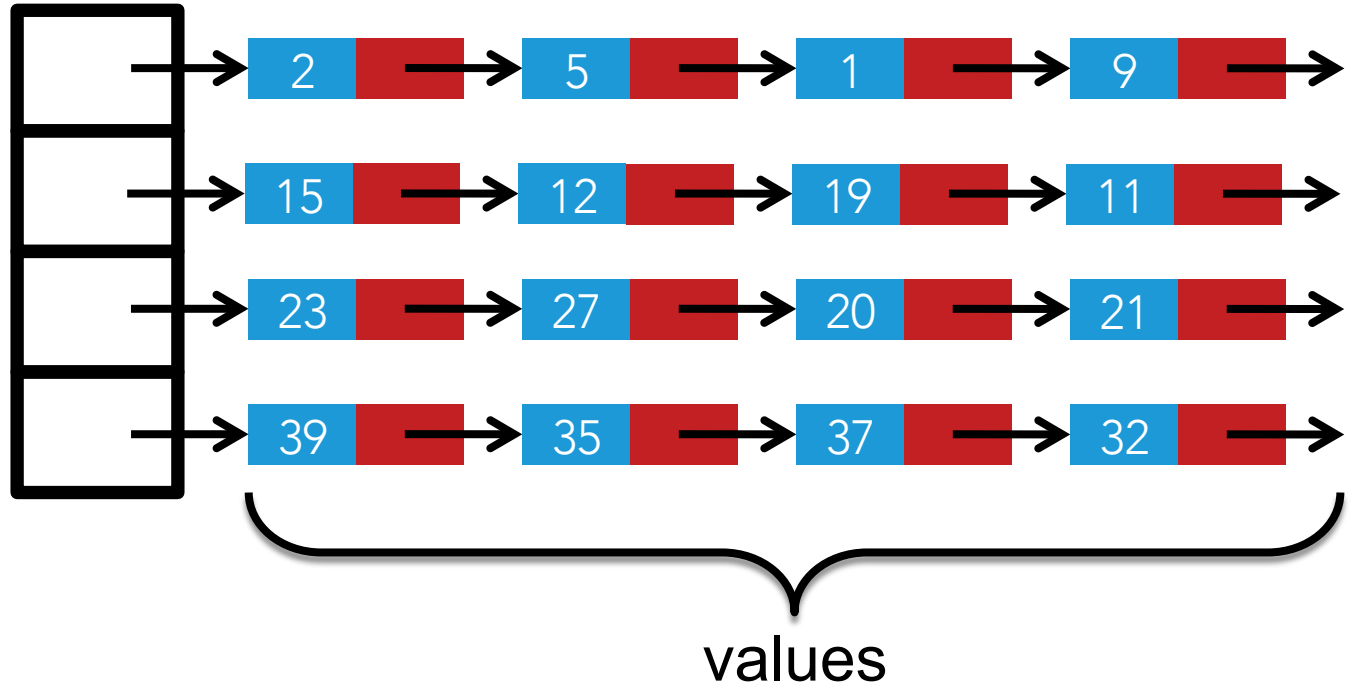
linked lists

- important:
 - ▣ don't lose any links!
 - ▣ last node points to NULL

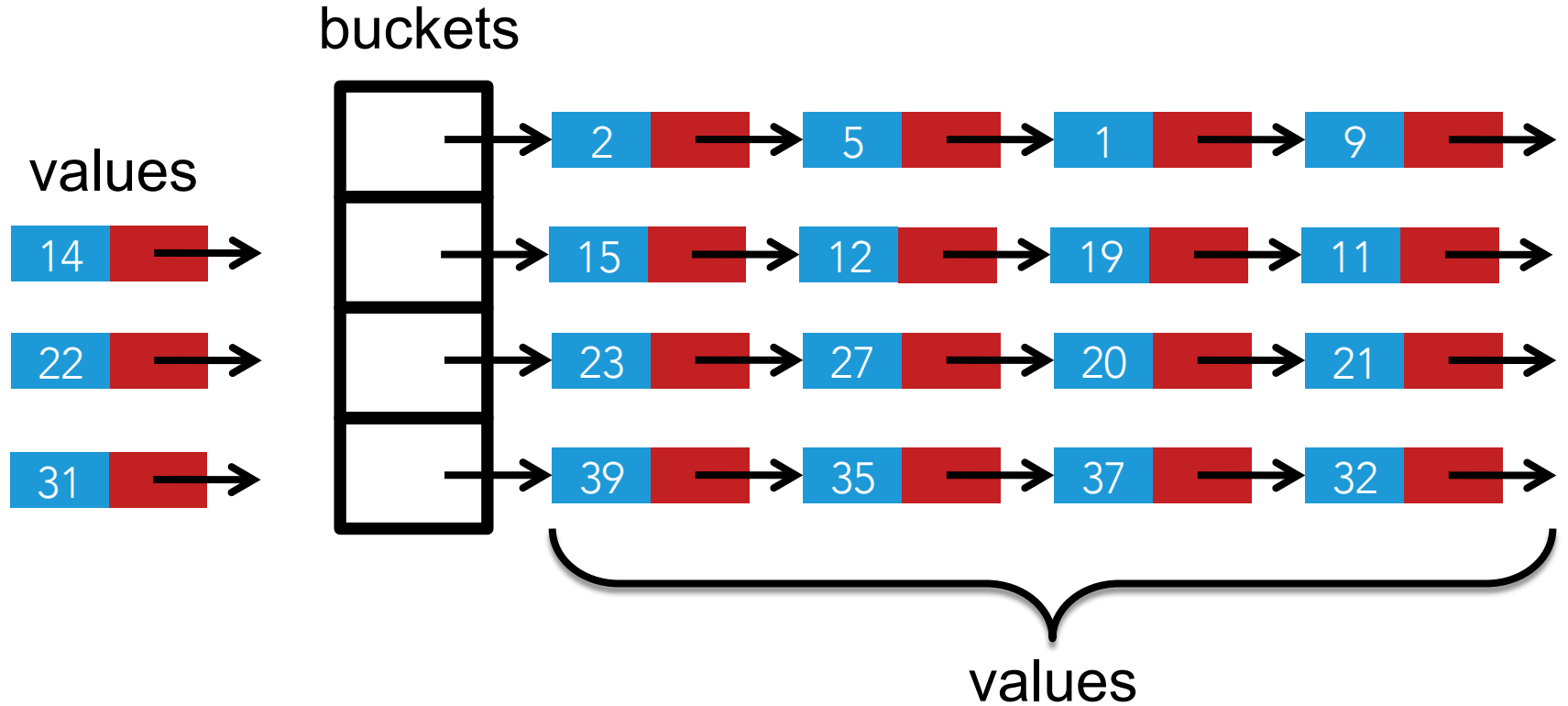


hash tables

buckets



hash tables



linked lists

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

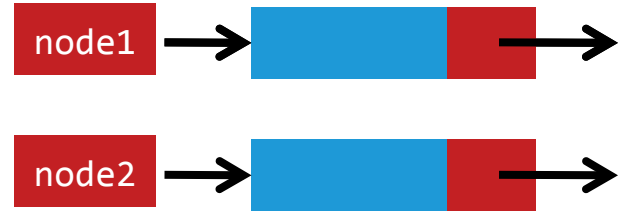
node *node1 = malloc(sizeof(node));
```



linked lists

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

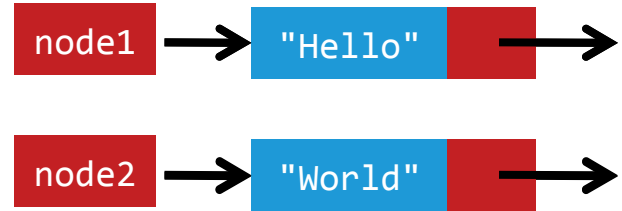
node *node1 = malloc(sizeof(node));
node *node2 = malloc(sizeof(node));
```



linked lists

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

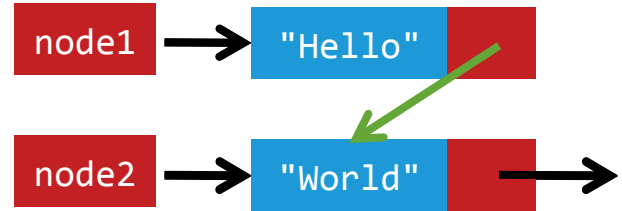
node *node1 = malloc(sizeof(node));
node *node2 = malloc(sizeof(node));
node1->word = "Hello";
node2->word = "World";
```



linked lists

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

node *node1 = malloc(sizeof(node));
node *node2 = malloc(sizeof(node));
node1->word = "Hello";
node2->word = "World";
node1->next = node2;
```



linked lists

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

```
node *node1 = malloc(sizeof(node));
node *node2 = malloc(sizeof(node));
node1->word = "Hello";
node2->word = "World";
node1->next = node2;
```



a hash table is
an array of linked lists

each element of array is a node *

hash table

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

node *hashtable[50];
```

a hash table is
an array of linked lists

each element of array is a node *

make a new word

- scan dictionary word by word

```
while (fscanf(file, "%s", word) != EOF)
{
    ...
}
```

make a new word

- malloc a node * for each new word

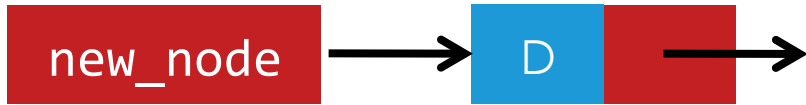
```
node *new_node = malloc(sizeof(node));
if (new_node == NULL)
{
    unload();
    return false;
}
```

make a new word

- copy word into node

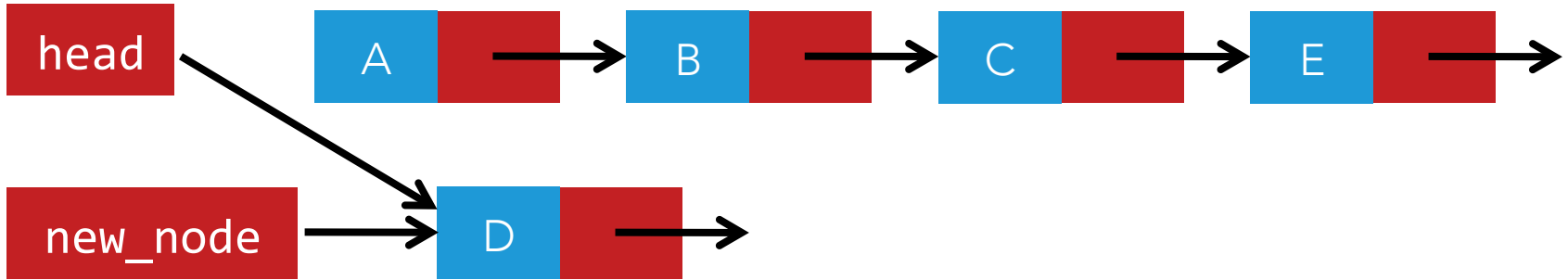
```
strcpy(new_node->word, word);
```

insert into a linked list: incorrect



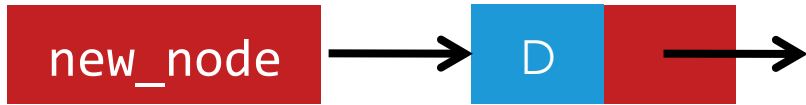
```
head = new_node;
```

insert into a linked list: incorrect



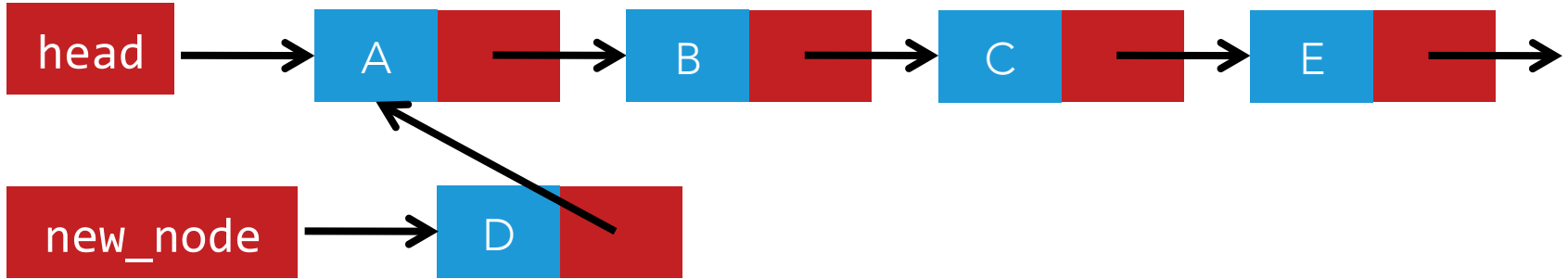
```
head = new_node;
```


insert into a linked list: correct



```
new_node->next = head;
```

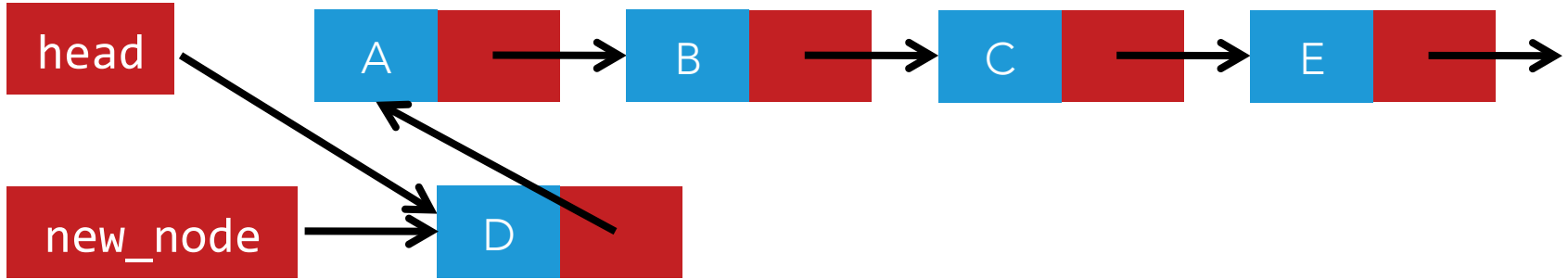
insert into a linked list: correct



```
new_node->next = head;
```

```
head = new_node;
```

insert into a linked list: correct



```
new_node->next = head;
```

```
head = new_node;
```

hash function

- takes a string
- returns an index
 - ▣ $\text{index} < \text{the number of buckets}$
- deterministic
 - ▣ the same value needs to map to the same bucket every time

hash the word

- `new_node->word` has the word from the dictionary
- hashing `new_node->word` will give us the index of a bucket in the hash table
- insert into the linked list

a hash table is
an array of linked lists

each element of array is a node *

tries

- every node contains an array of node pointers
 - ▣ 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;
```

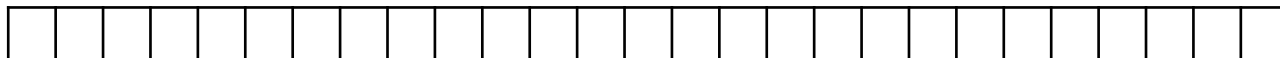

load

- 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

"fox"

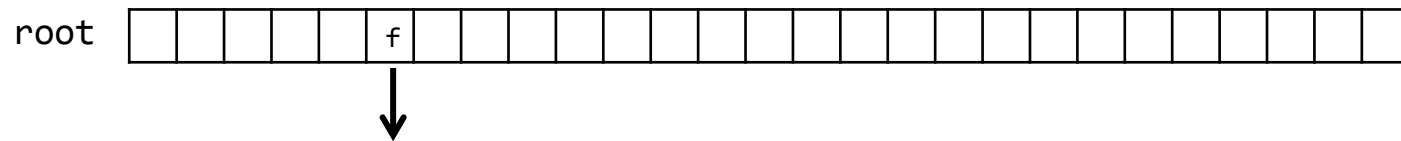
f: root->children[5]

root



"fox"

f: root->children[5]

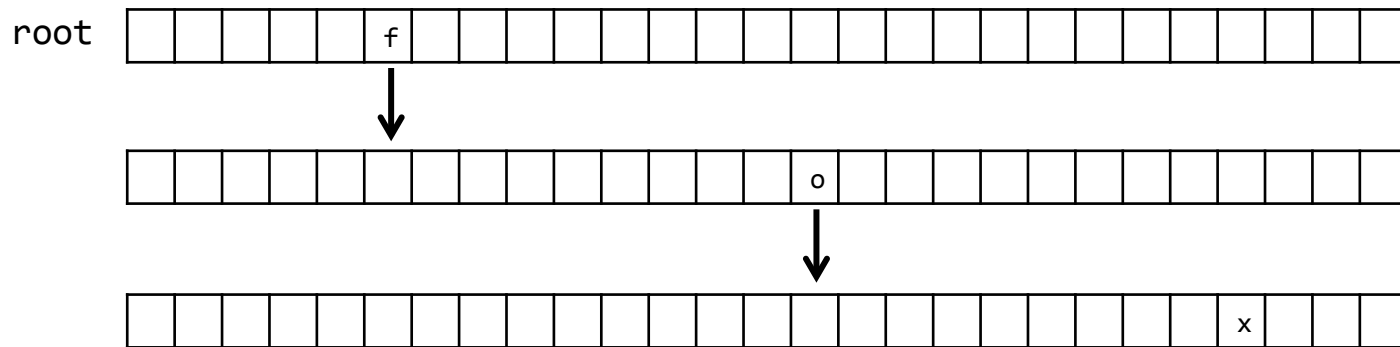


"fox"

f: root->children[5]

o: root->children[5]->children[14]

x: root->children[5]->children[14]->children[23]

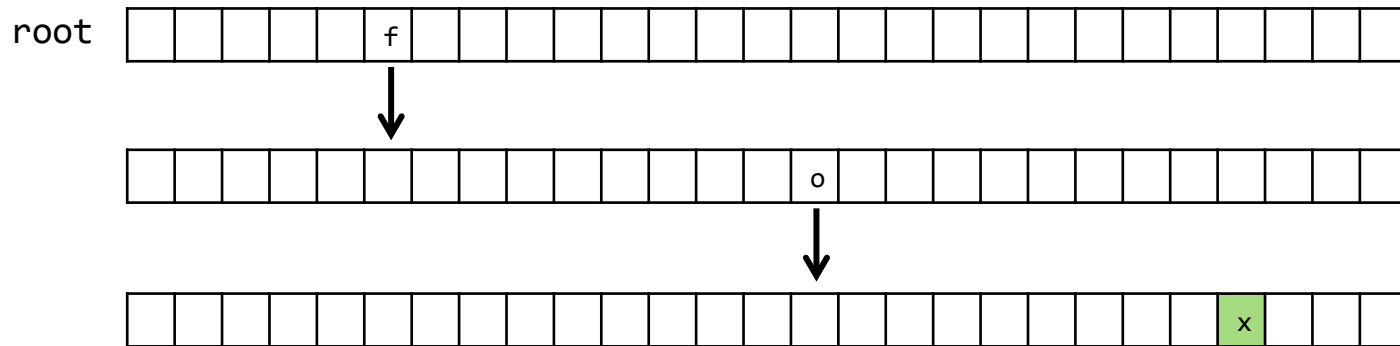


"fox"

f: root->children[5]

o: root->children[5]->children[14]

x: root->children[5]->children[14]->children[23]

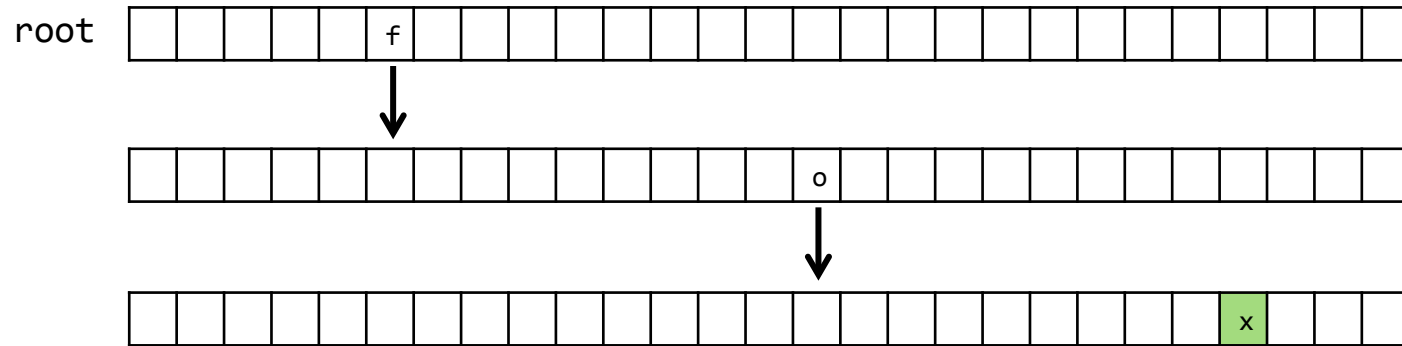


"foo"

f: root->children[5]

o: root->children[5]->children[14]

o: root->children[5]->children[14]->children[14]

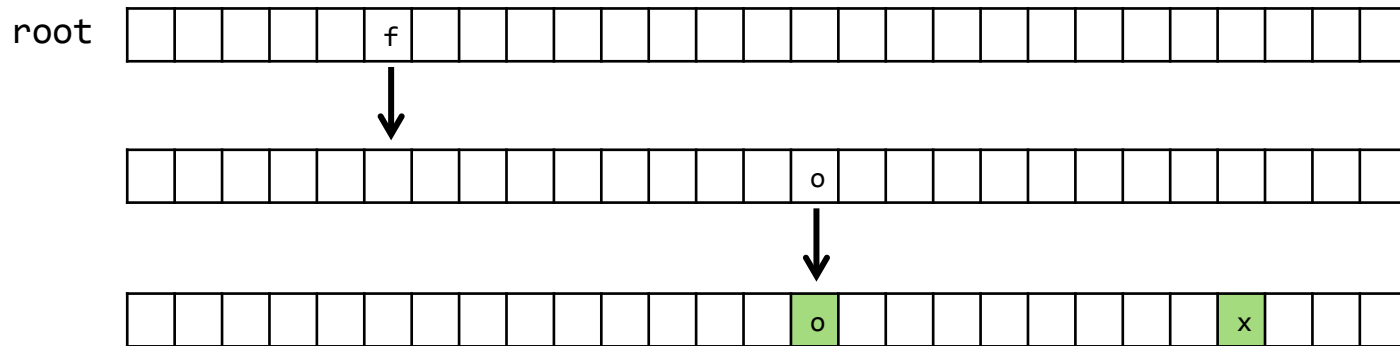


"foo"

f: root->children[5]

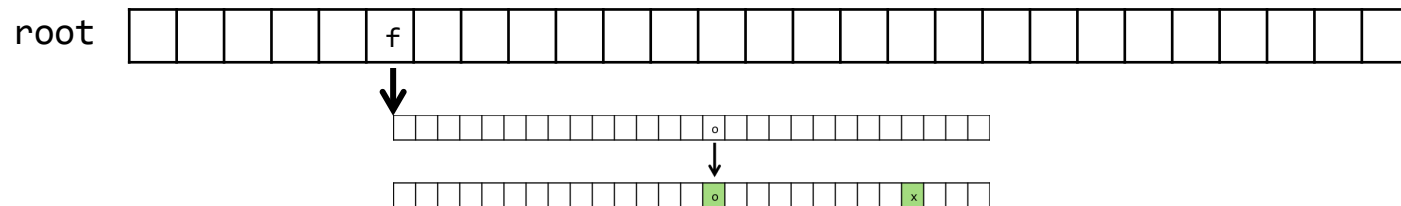
o: root->children[5]->children[14]

o: root->children[5]->children[14]->children[14]



"dog"

d: root->children[3]

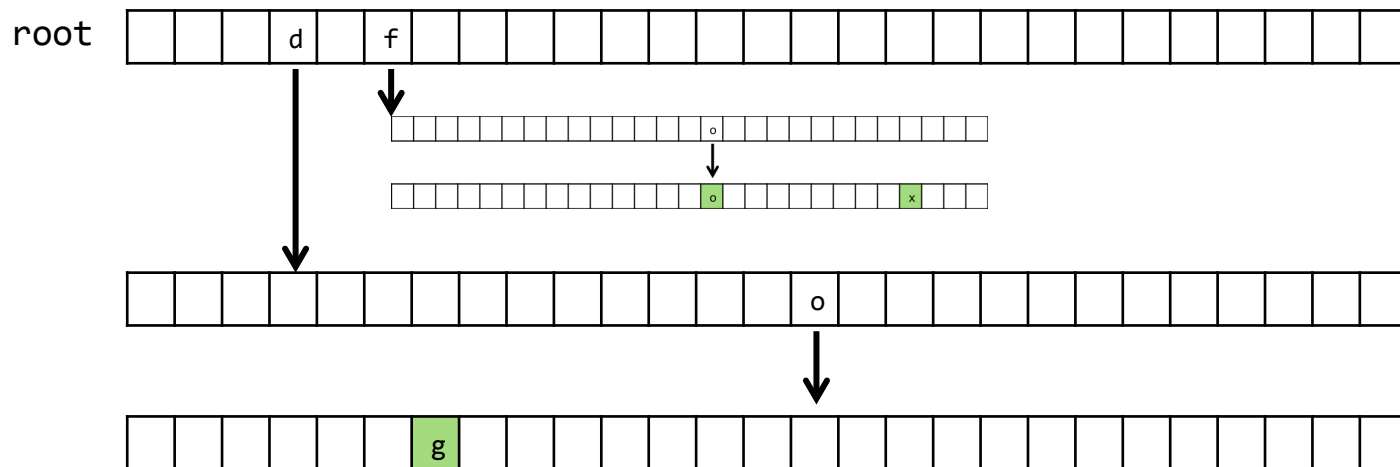


"dog"

d: root->children[3]

o: root->children[3]->children[14]

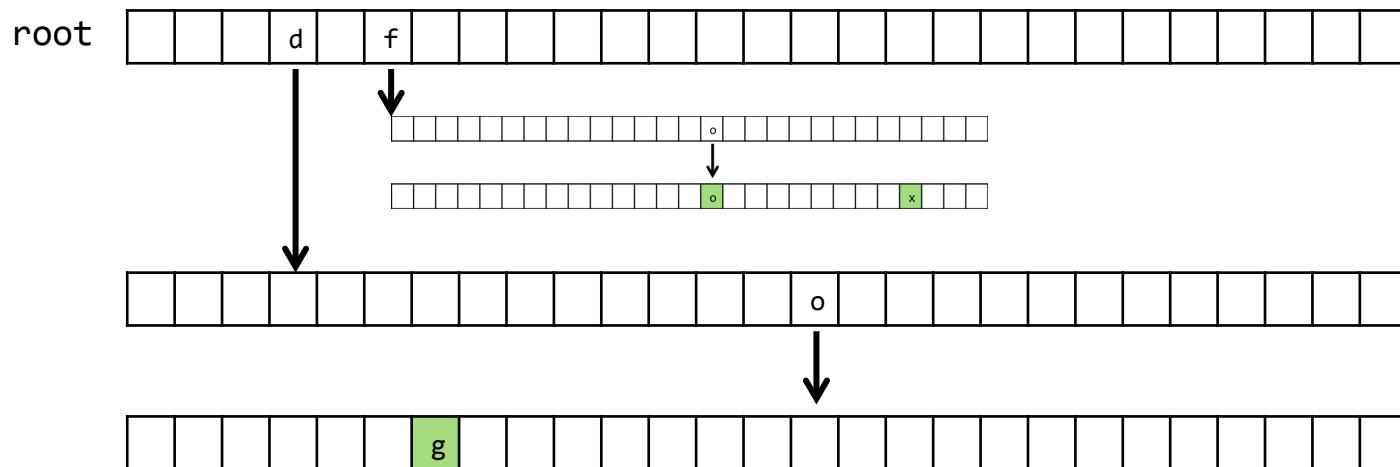
g: root->children[3]->children[6]



"do"

d: root->children[3]

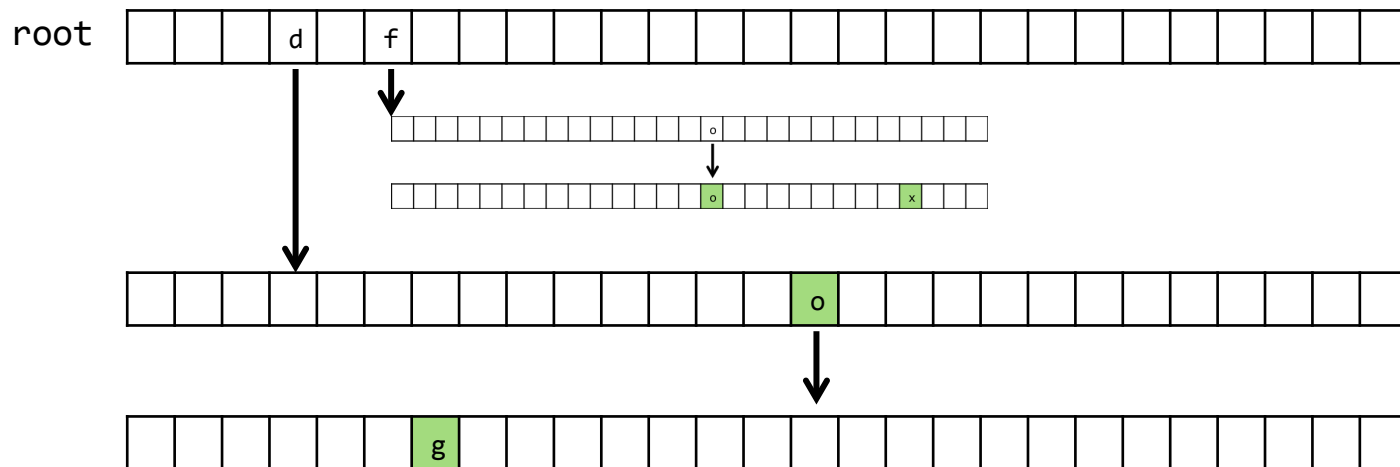
o: root->children[3]->children[14]



"do"

d: root->children[3]

o: root->children[3]->children[14]



TODO:

- load
- check
- size
- unload

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
- which bucket would the word be in?
 - ▣ `hashtable[hash(word)]`

a hash table is
an array of linked lists

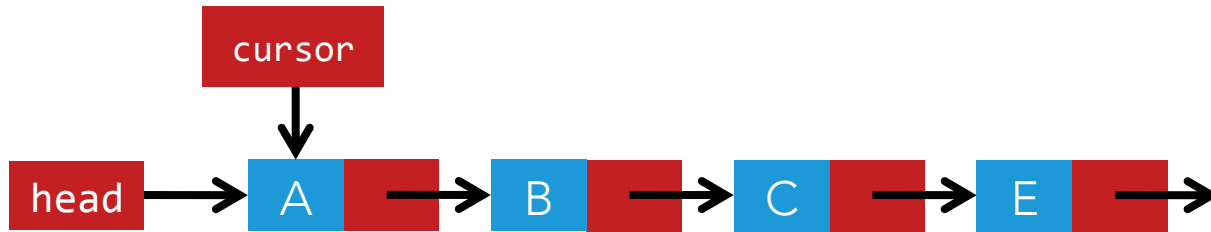
each element of array is a node *

check

- if the word exists, it can be found in the hash table
- which bucket would the word be in?
 - ▣ `hashtable[hash(word)]`
- search in that linked list
 - ▣ `strcmp`

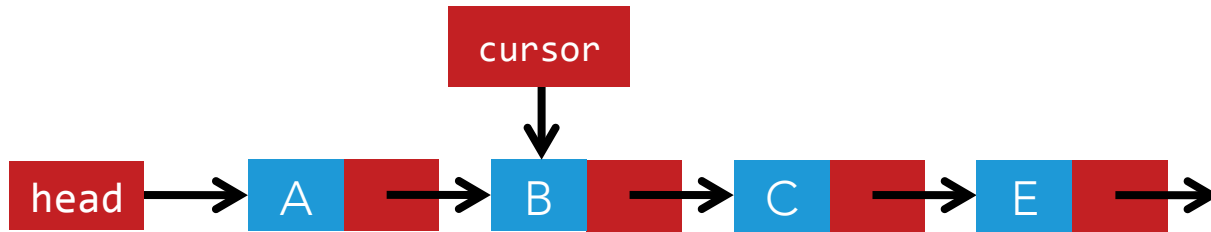
traversing linked lists

```
node *cursor = head;  
while (cursor != NULL)  
{  
    // do something  
    cursor = cursor->next;  
}
```



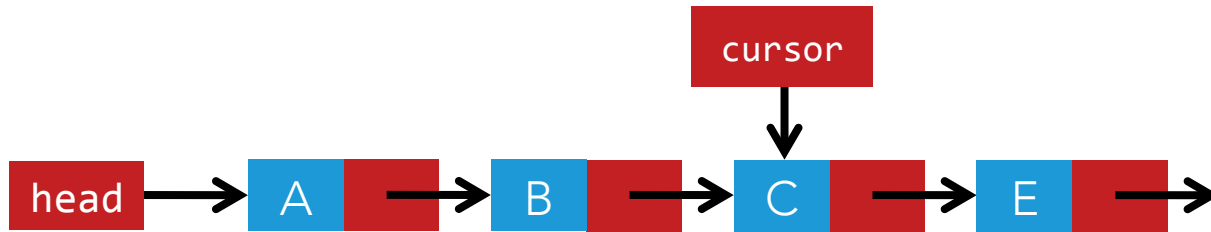
traversing linked lists

```
node *cursor = head;  
while (cursor != NULL)  
{  
    // do something  
    cursor = cursor->next;  
}
```



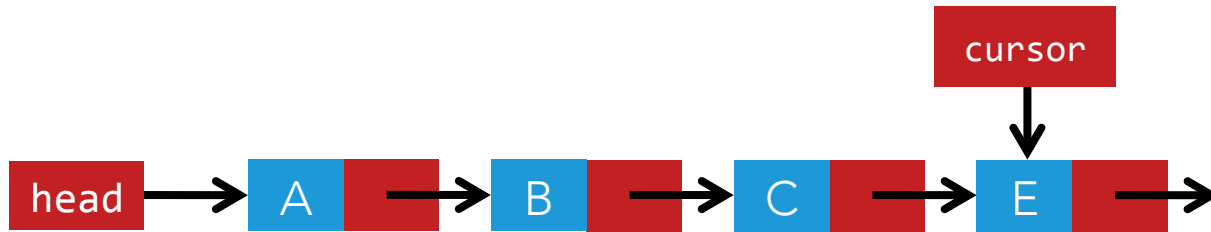
traversing linked lists

```
node *cursor = head;  
while (cursor != NULL)  
{  
    // do something  
    cursor = cursor->next;  
}
```



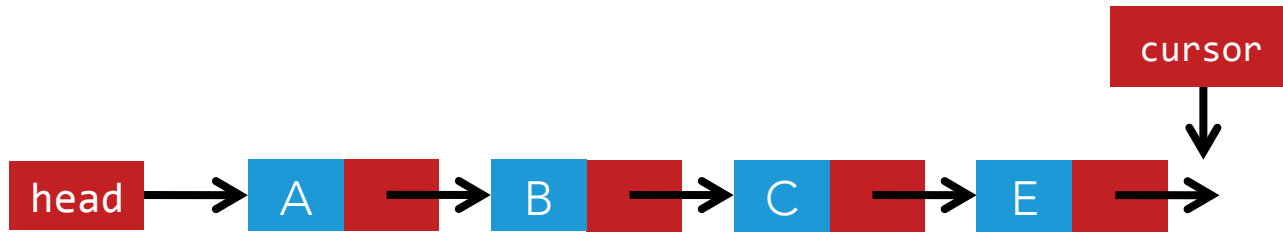
traversing linked lists

```
node *cursor = head;
while (cursor != NULL)
{
    // do something
    cursor = cursor->next;
}
```



traversing linked lists

```
node *cursor = head;  
while (cursor != NULL)  
{  
    // do something  
    cursor = cursor->next;  
}
```



traversing a 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

TODO

- load
- check
- size**
- unload