This is CS50

Week 5

Scan your HUID at the back table for attendance
Open code.cs50.io!

https://carterzenke.me/section

Think, Pair, Share

- What are you excited about from this week's lecture?
- What do you want to learn more about?

- What are the key **trade-offs** between data structures we should consider in decisions about which to use?
- What are some of the primary operations we should know how to do on a **linked list**?
- How can we build our very own **hash table**?

Scenario

Imagine you work for a company that has created a **digital assistant** running on a mobile device.

Customer reports indicate

people have trouble

activating the assistant

with its "wake word".

Your team has been asked to ensure the voice assistant can be **awoken with a greater variety of words.**

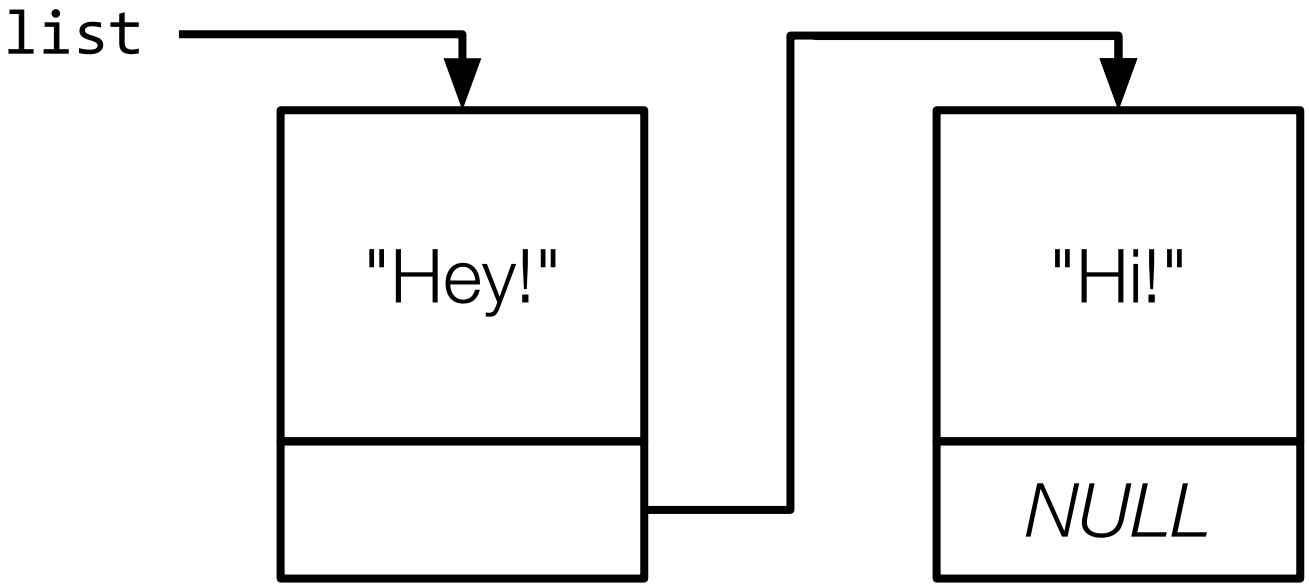
What **data structure** would you propose the team build to store these words?

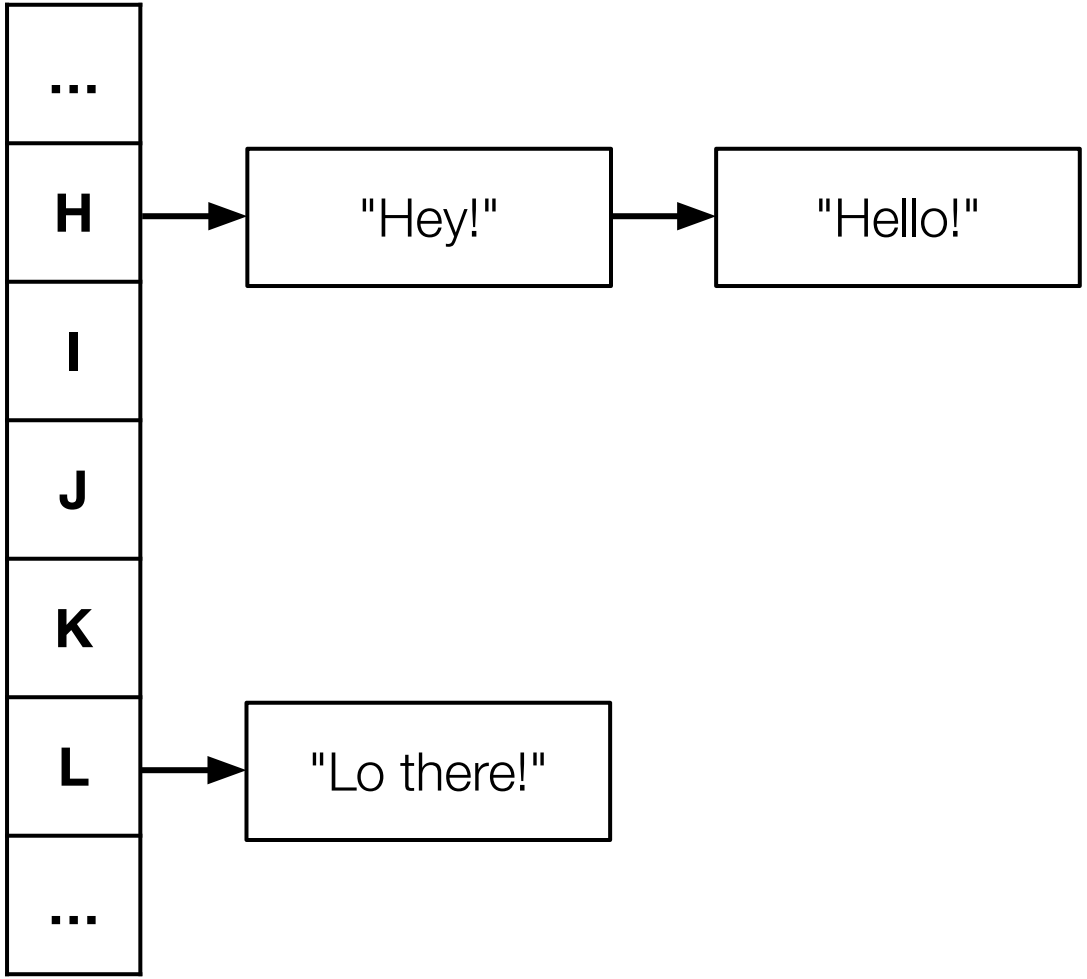
Deletion
Insertion
Search

1. Search
2. Insertion
3. Deletion

1. Insertion
2. Search
3. Deletion

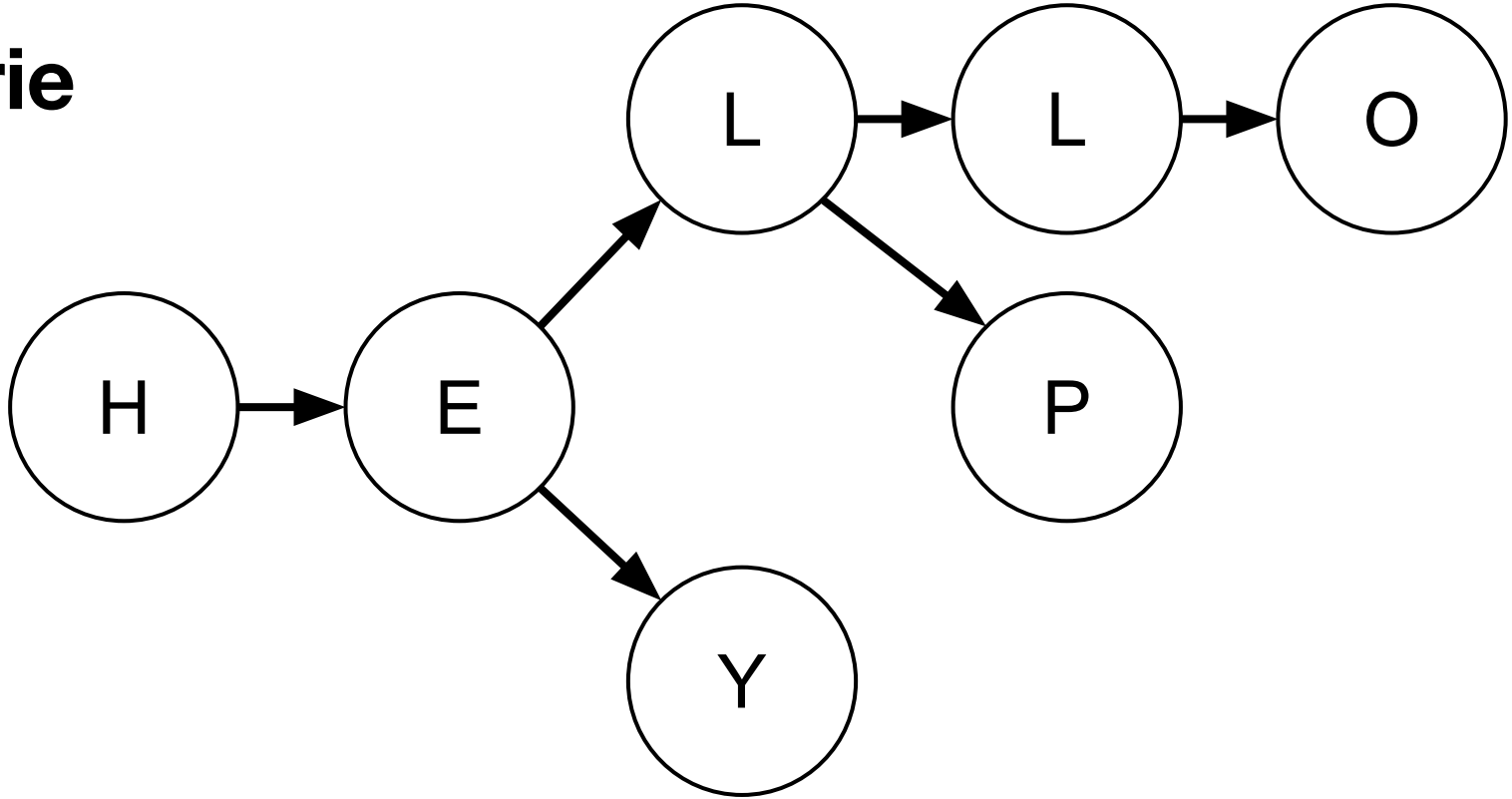
Linked List



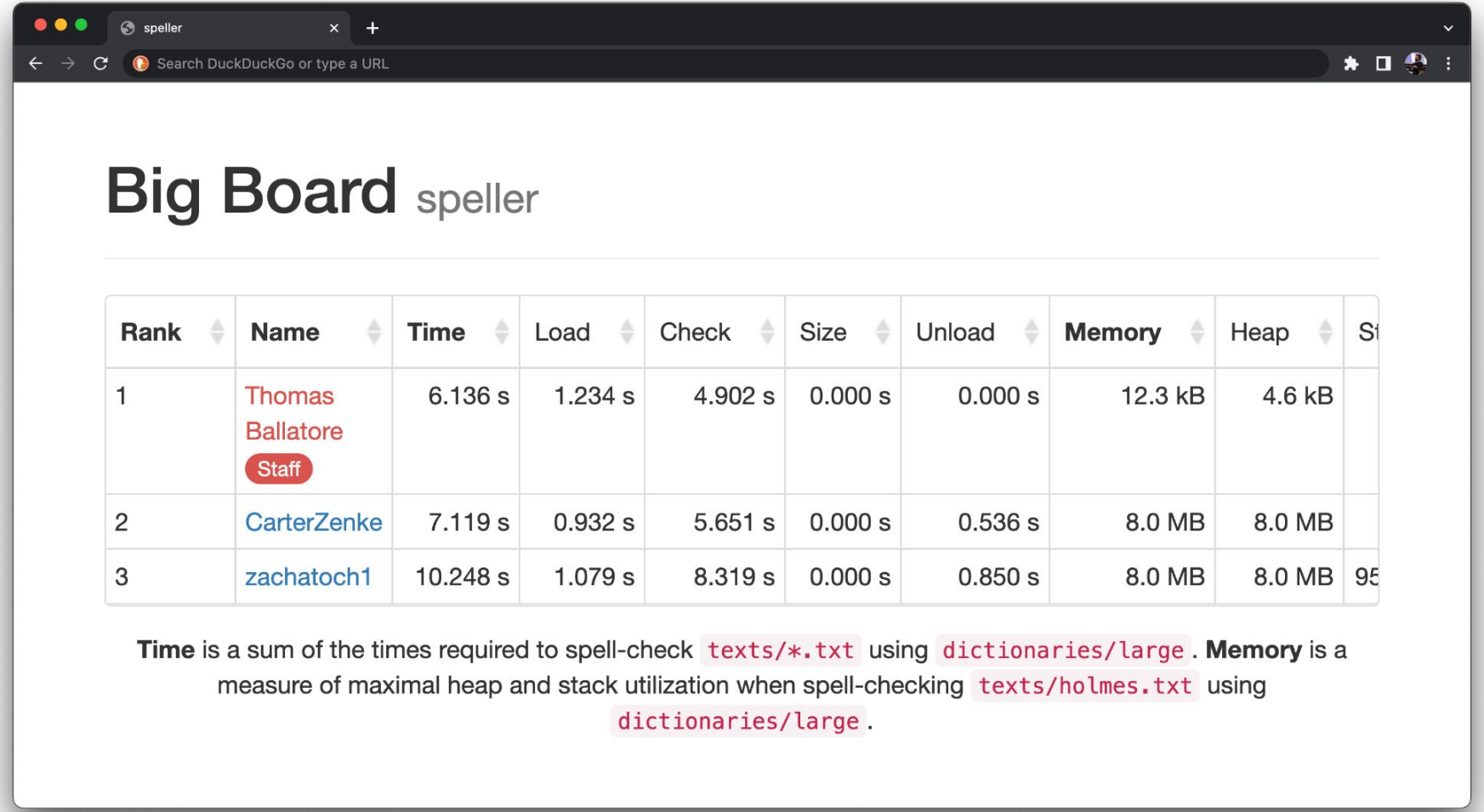


Hash Table

Trie



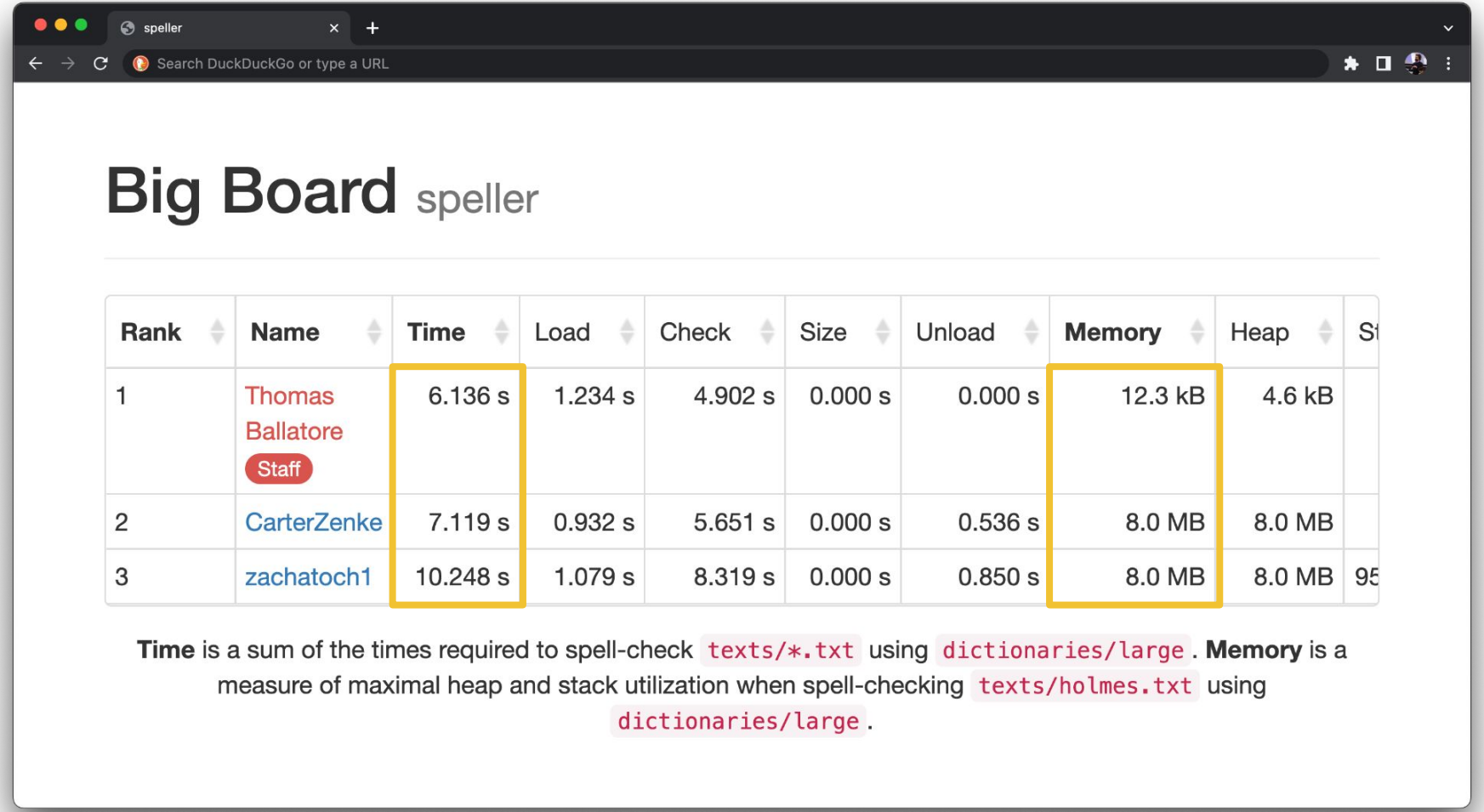
Trade-offs



Big Board speller

Rank	Name	Time	Load	Check	Size	Unload	Memory	Heap	St
1	Thomas Ballatore Staff	6.136 s	1.234 s	4.902 s	0.000 s	0.000 s	12.3 kB	4.6 kB	
2	CarterZenke	7.119 s	0.932 s	5.651 s	0.000 s	0.536 s	8.0 MB	8.0 MB	
3	zachatoch1	10.248 s	1.079 s	8.319 s	0.000 s	0.850 s	8.0 MB	8.0 MB	95

Time is a sum of the times required to spell-check `texts/*.txt` using `dictionaries/large`. **Memory** is a measure of maximal heap and stack utilization when spell-checking `texts/holmes.txt` using `dictionaries/large`.



Big Board speller

Rank	Name	Time	Load	Check	Size	Unload	Memory	Heap	St
1	Thomas Ballatore Staff	6.136 s	1.234 s	4.902 s	0.000 s	0.000 s	12.3 kB	4.6 kB	
2	CarterZenke	7.119 s	0.932 s	5.651 s	0.000 s	0.536 s	8.0 MB	8.0 MB	
3	zachatoch1	10.248 s	1.079 s	8.319 s	0.000 s	0.850 s	8.0 MB	8.0 MB	95

Time is a sum of the times required to spell-check `texts/*.txt` using `dictionaries/large`. **Memory** is a measure of maximal heap and stack utilization when spell-checking `texts/holmes.txt` using `dictionaries/large`.

Embedded EthiCS

About me

William Cochran
Postdoctoral Fellow in Philosophy
Embedded EthiCS program @ Harvard

The Embedded EthiCS course modules teach students to...

identify ethical and social issues

reason through ethical and social issues

communicate their reasoned position

design ethically and socially responsible systems

1. Run time (speed)
2. Memory usage (space)
3. Time to implementation

Privacy &
Security

Privacy ~ the ability to control information about oneself

Security ~ the ability to protect information from unauthorized access

In groups of 2-4, imagine that you're a team that has been tasked with developing each of the following (A-D). Ask yourselves: which of these merits taking the extra steps to ensure privacy/security and which does not? Be prepared to give reasons for your answers.

A. A mobile app for a 'smart' water filter that helps users track their weekly water consumption.

B. A spell checker that stores a dictionary of words for fast lookup and to suggest corrections for misspelled words.

C. A database containing the average high school GPA of each Harvard freshman class from 2018-2022, as part of a study on the impact of COVID on academic performance among high schoolers.

D. A voice assistant app that is having trouble with its wake word recognizing people with non-English accents, and you've been asked to gather and store voice recordings of users to help address the error.

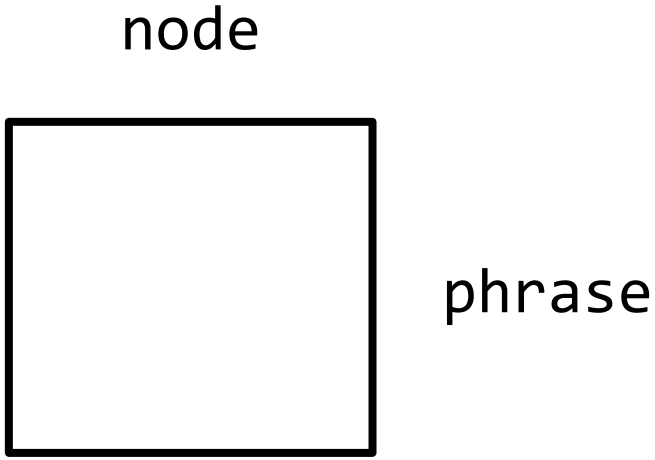
Nodes

```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```

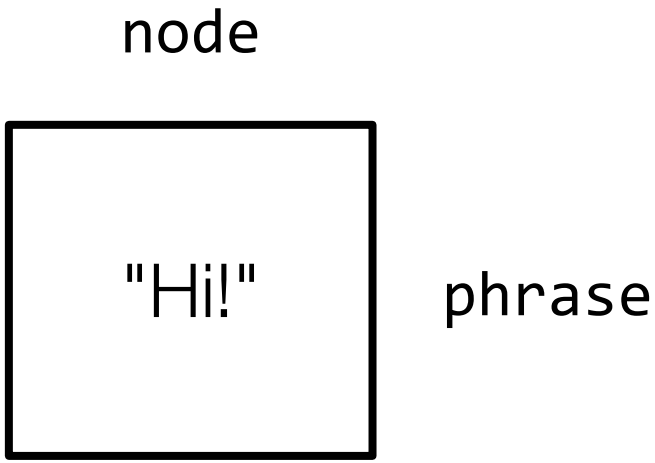
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```

node

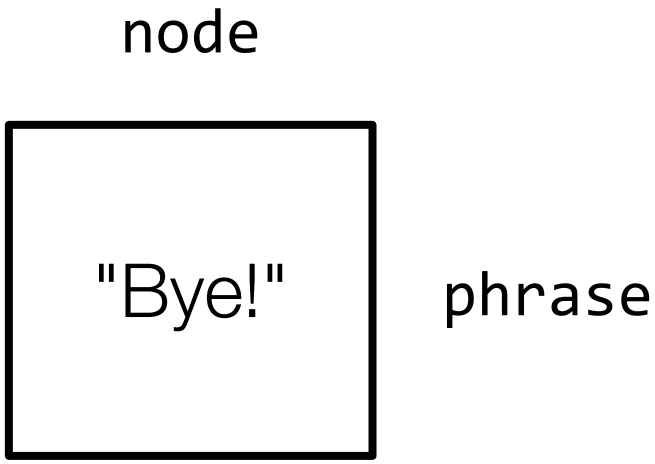
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



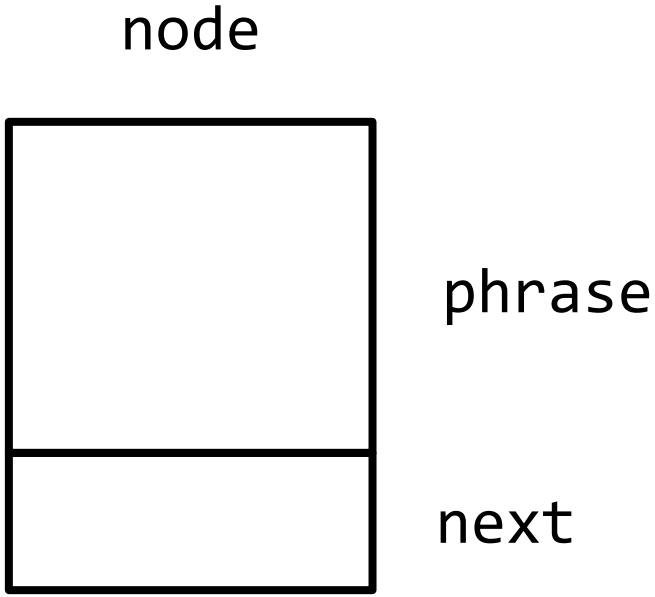
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



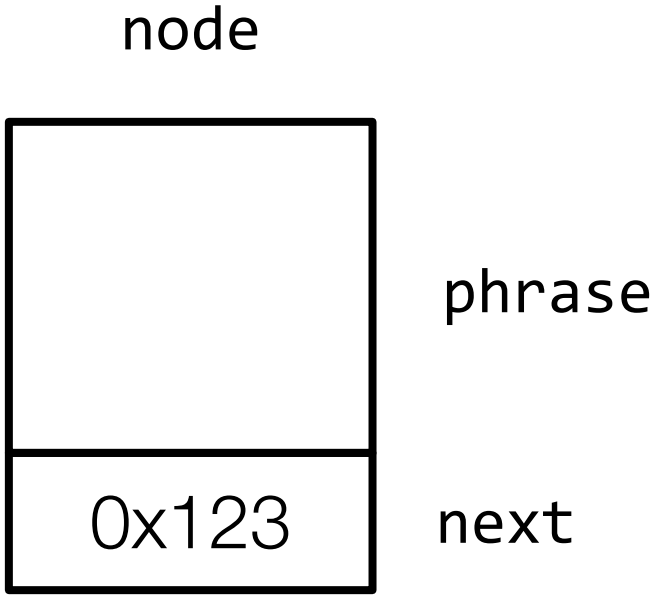
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



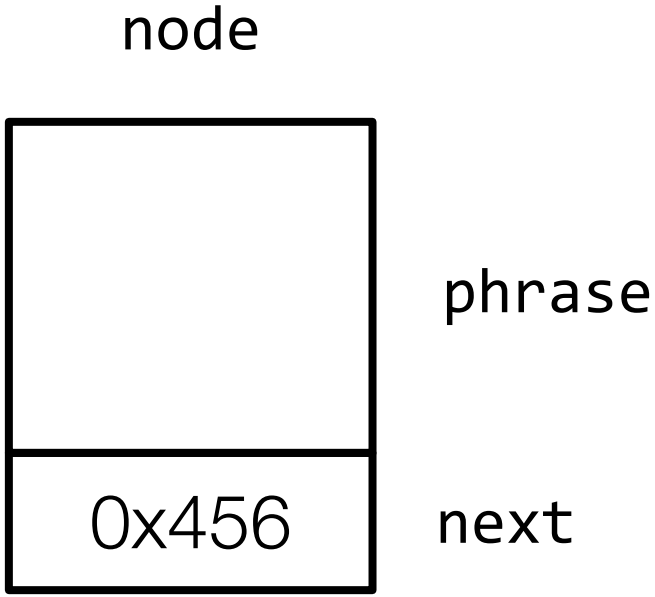
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



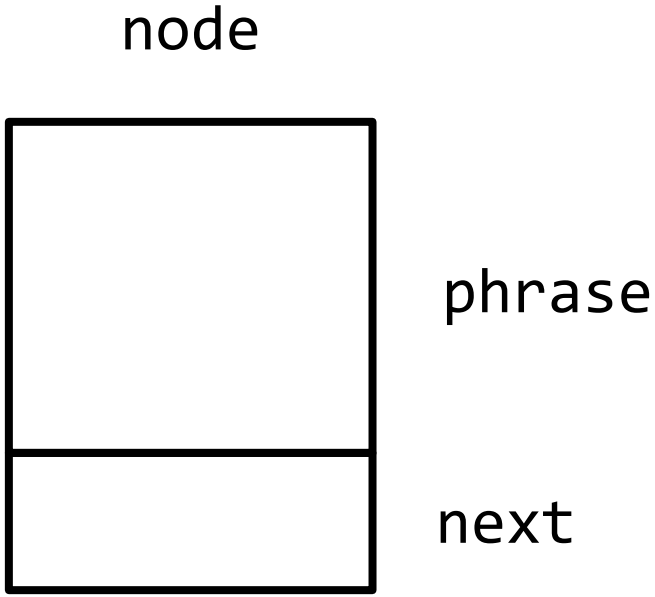
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```

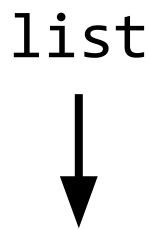


Creating a Linked List

Creating a Linked List

Download and open **list.c**.

```
node *list = NULL;
```

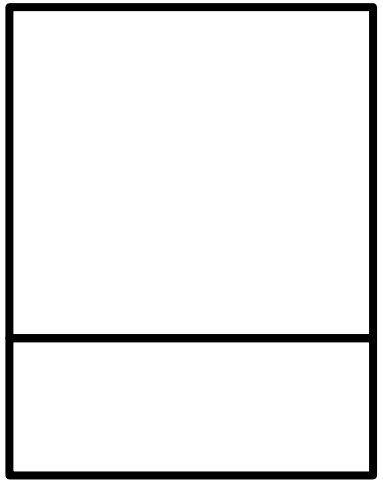


```
node *n = malloc(sizeof(node));
```

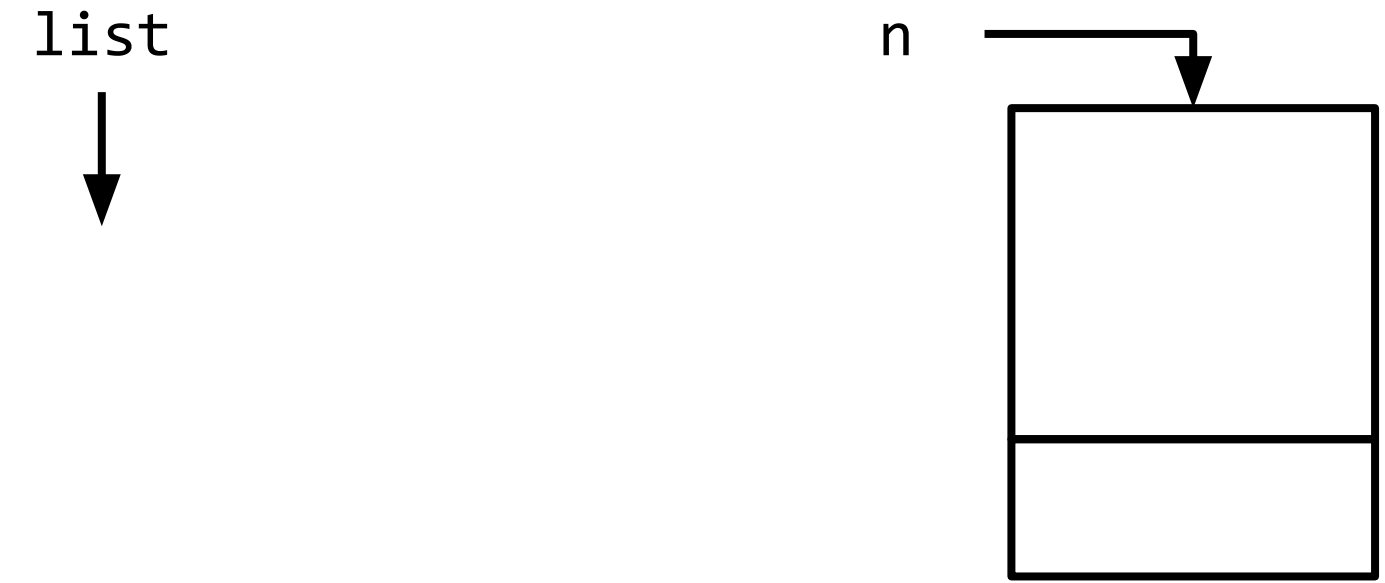
list

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

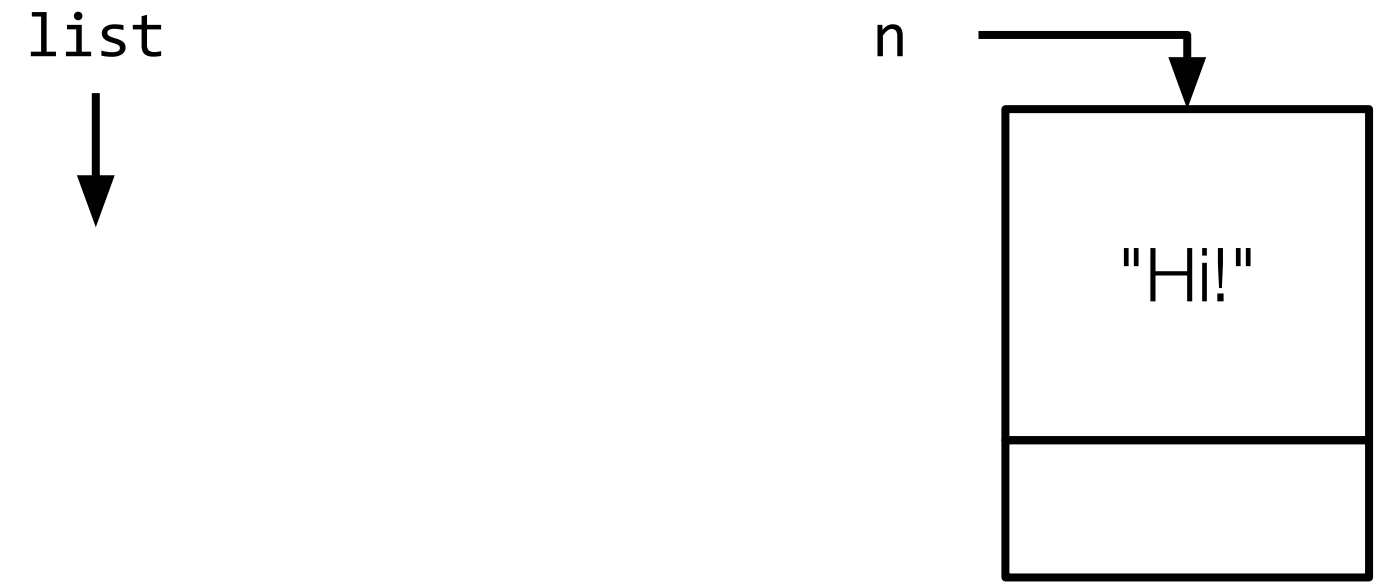
list



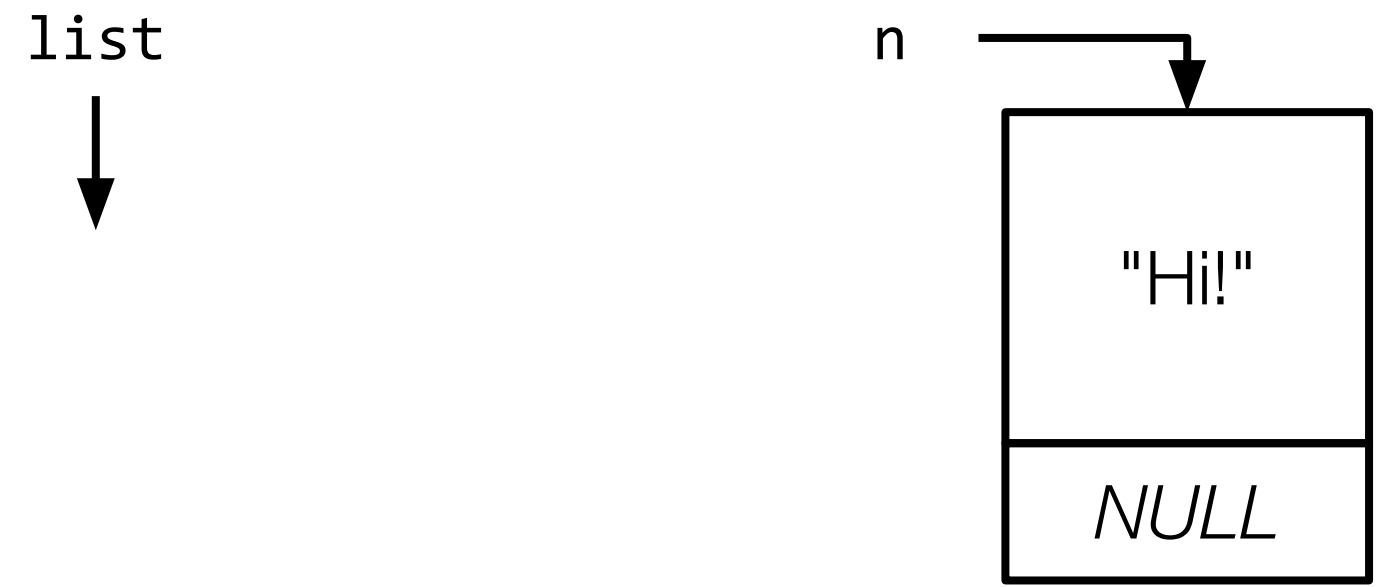
```
node *n = malloc(sizeof(node));
```



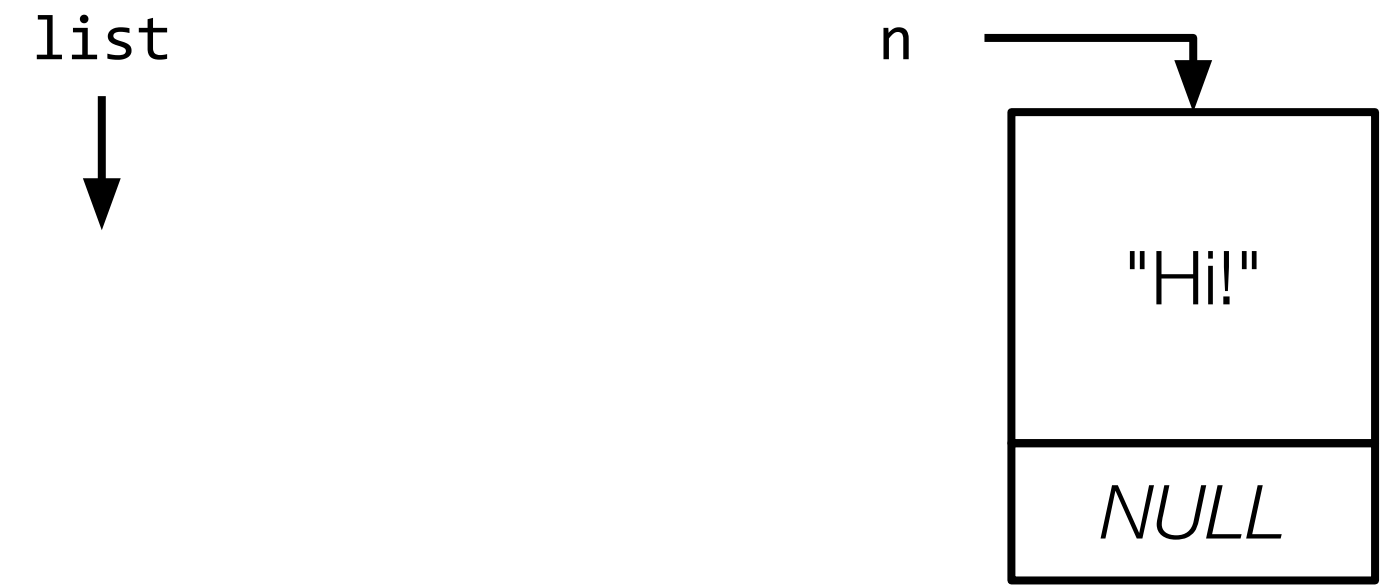
```
node *n = malloc(sizeof(node));
n->phrase = "Hi!";
```



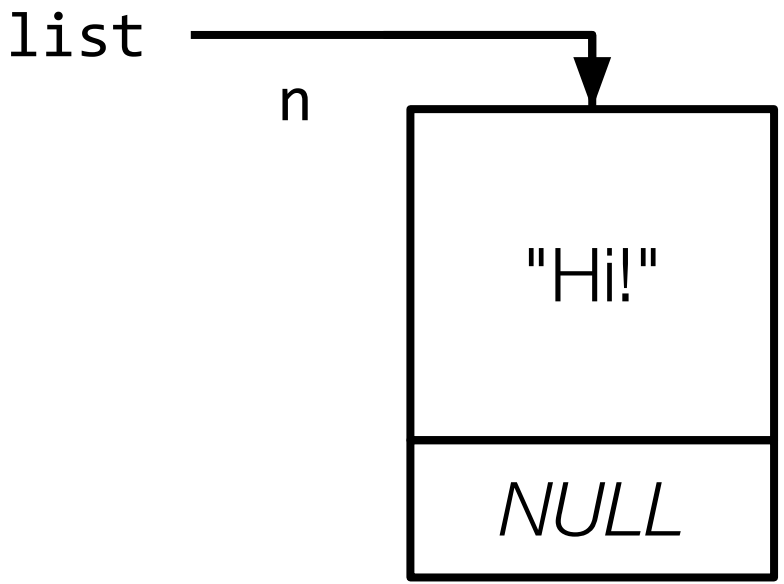
```
node *n = malloc(sizeof(node));
n->phrase = "Hi!";
n->next = NULL;
```



```
list = n;
```

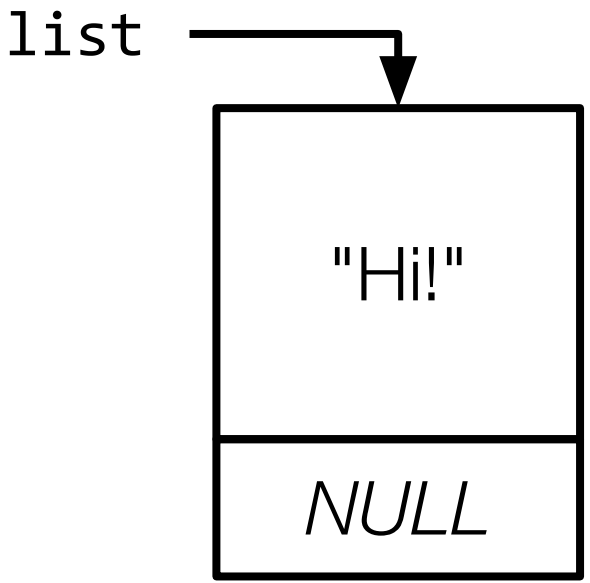


```
list = n;
```

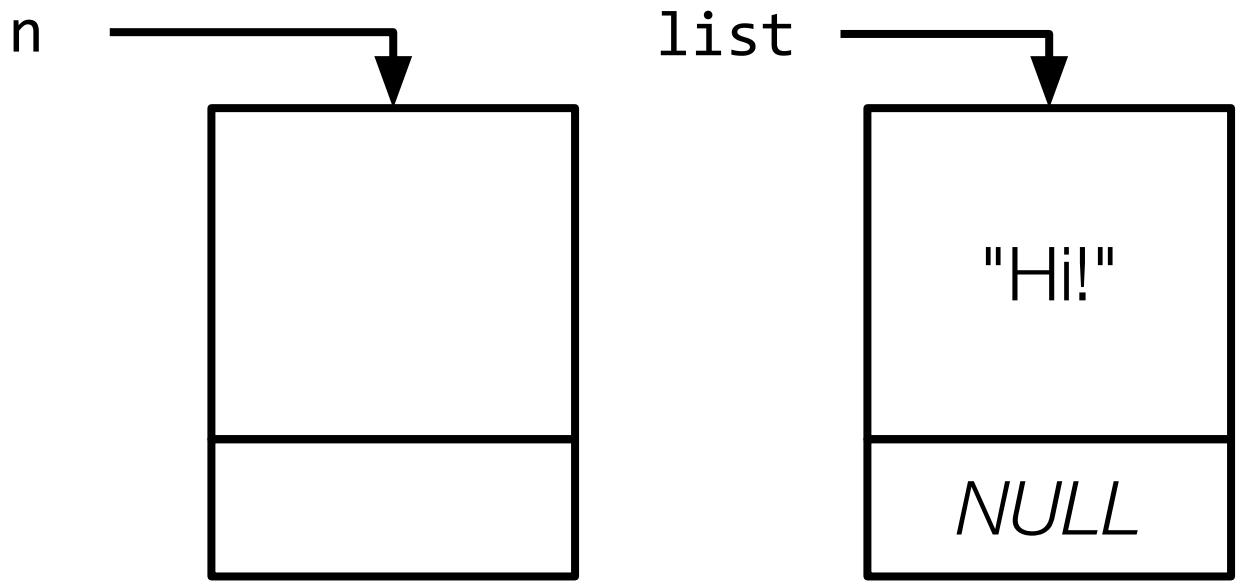


Inserting Nodes

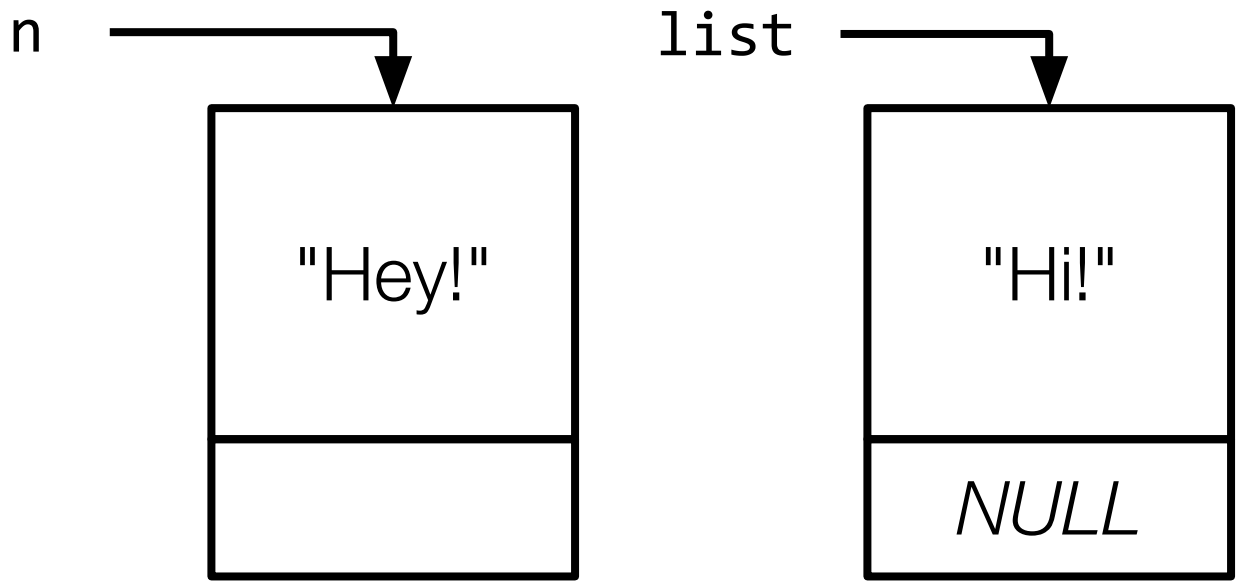
```
n = malloc(sizeof(node));
```



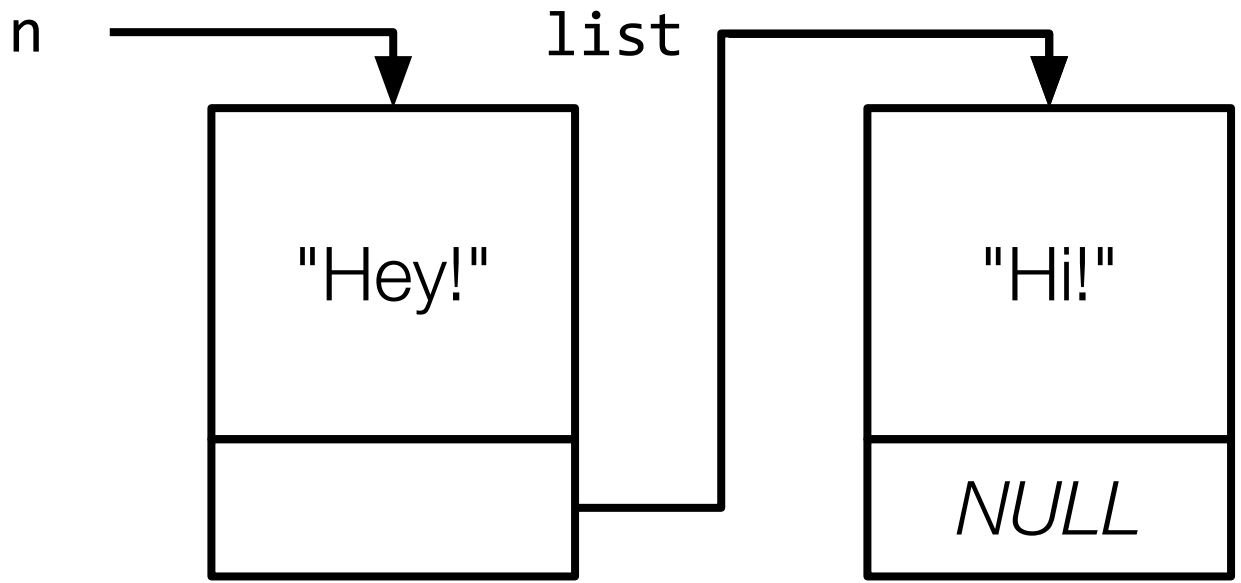
```
n = malloc(sizeof(node));
```



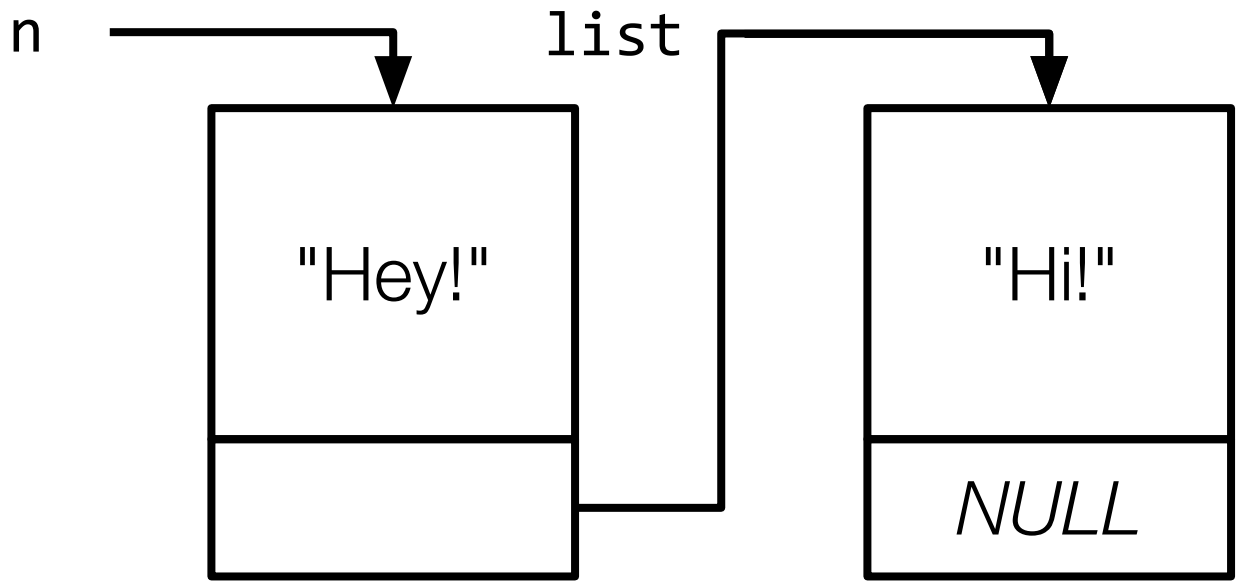
```
n = malloc(sizeof(node));
n->phrase = "Hey!";
```



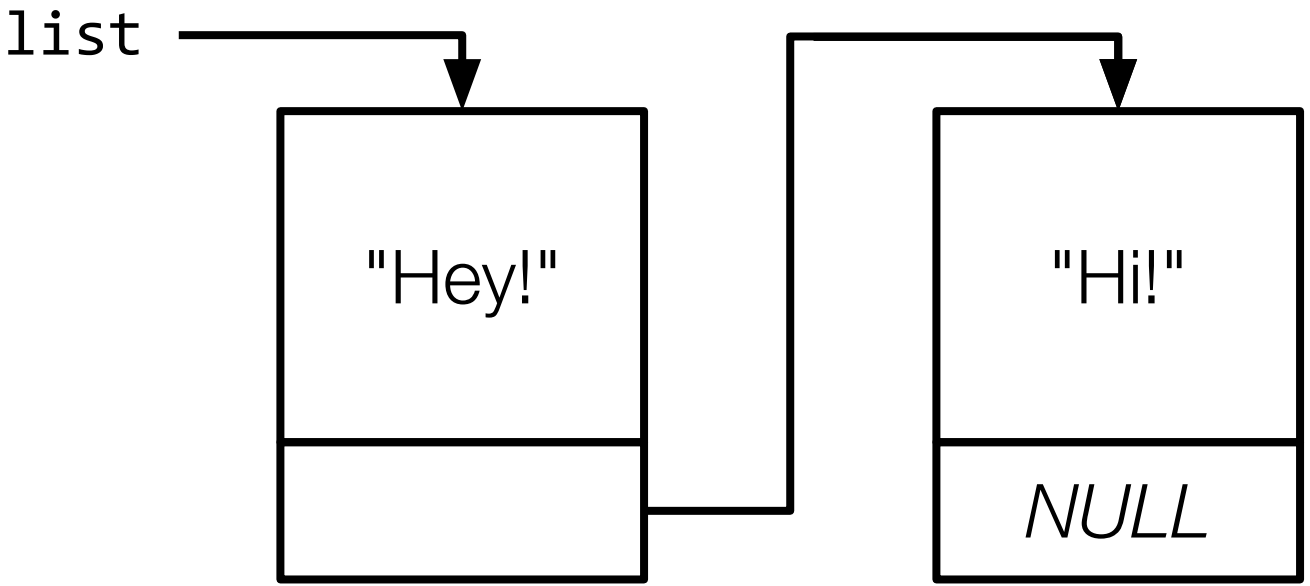
```
n = malloc(sizeof(node));
n->phrase = "Hey!";
n->next = list;
```



```
list = n;
```



```
list = n;
```

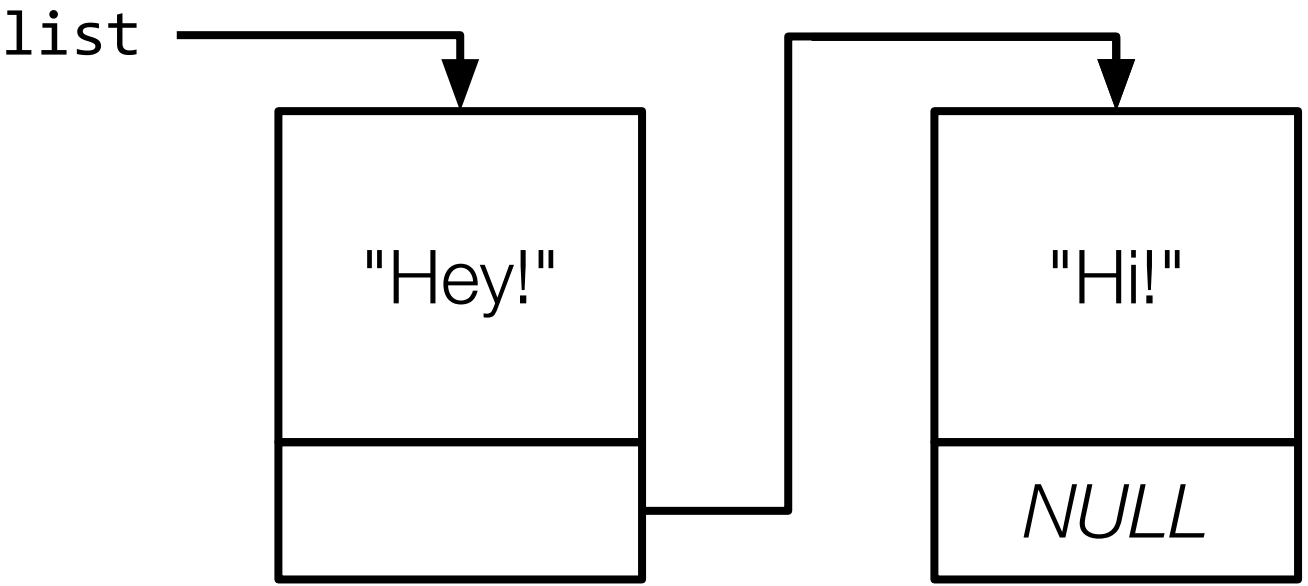


Inserting into a Linked List

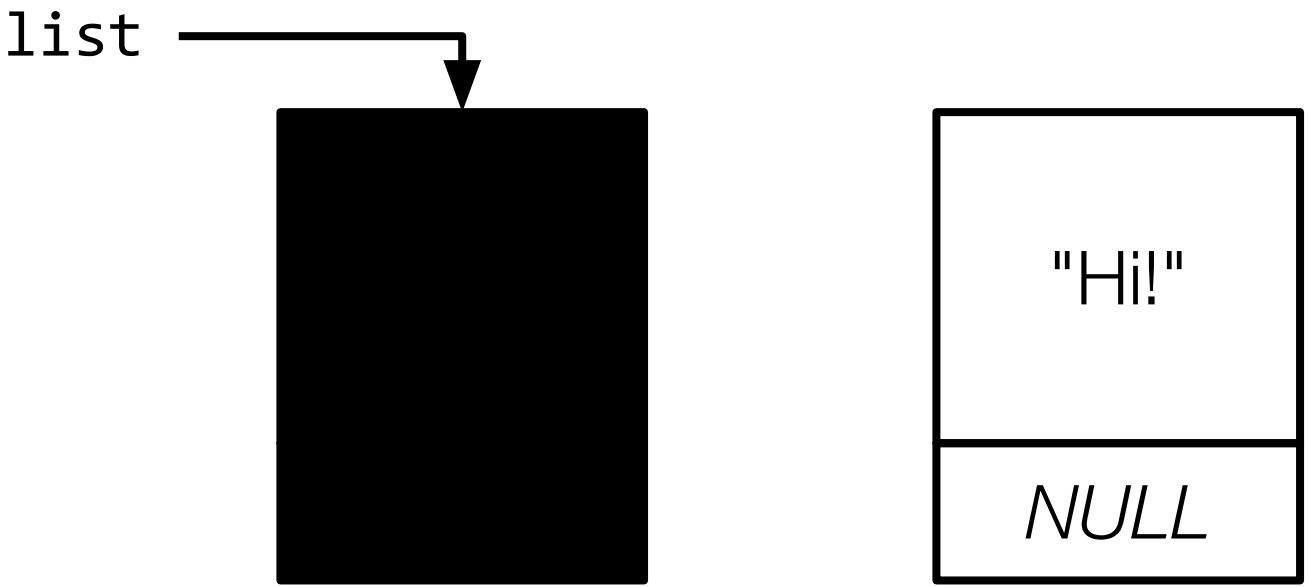
Download and open **list.c**.

Find the first TODO.

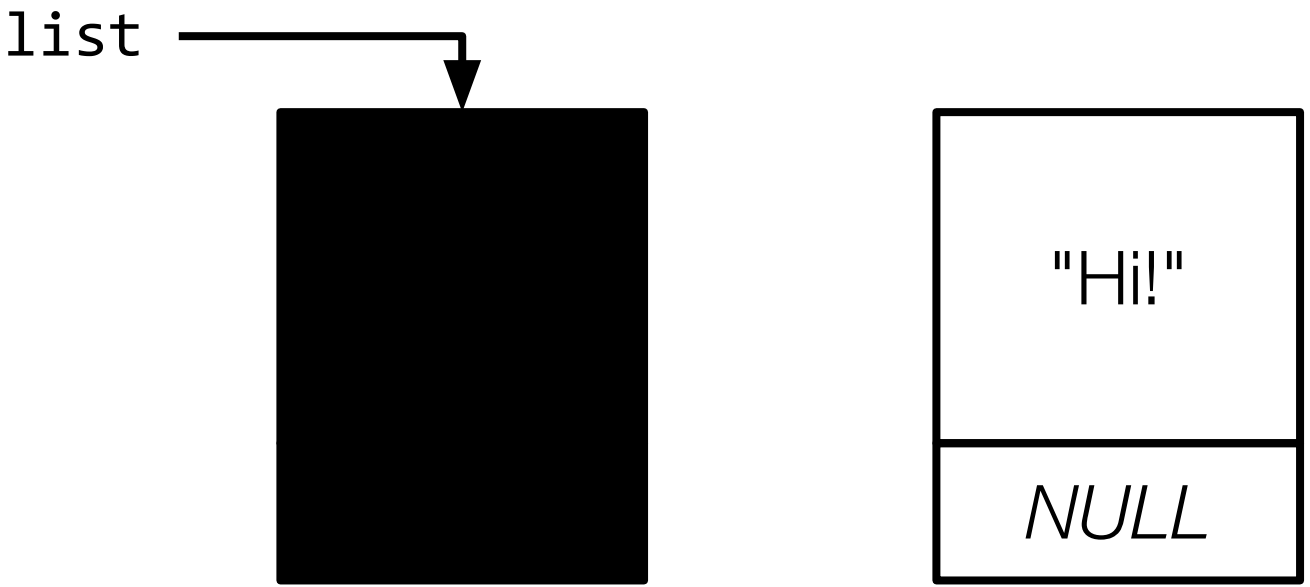
Starting below that TODO, implement code to add a node to the linked list. Ensure that **list** always points to the head of the linked list. Also ensure your new node contains a phrase.

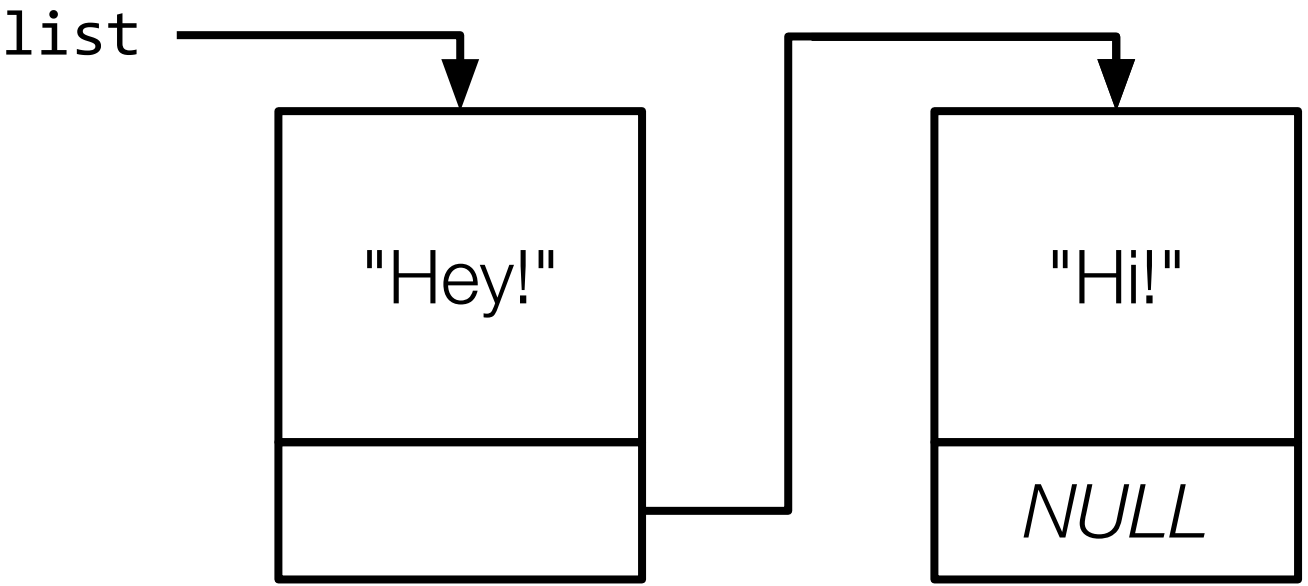

list
"Hey!"
"Hi!"
NULL

```
free(list);
```

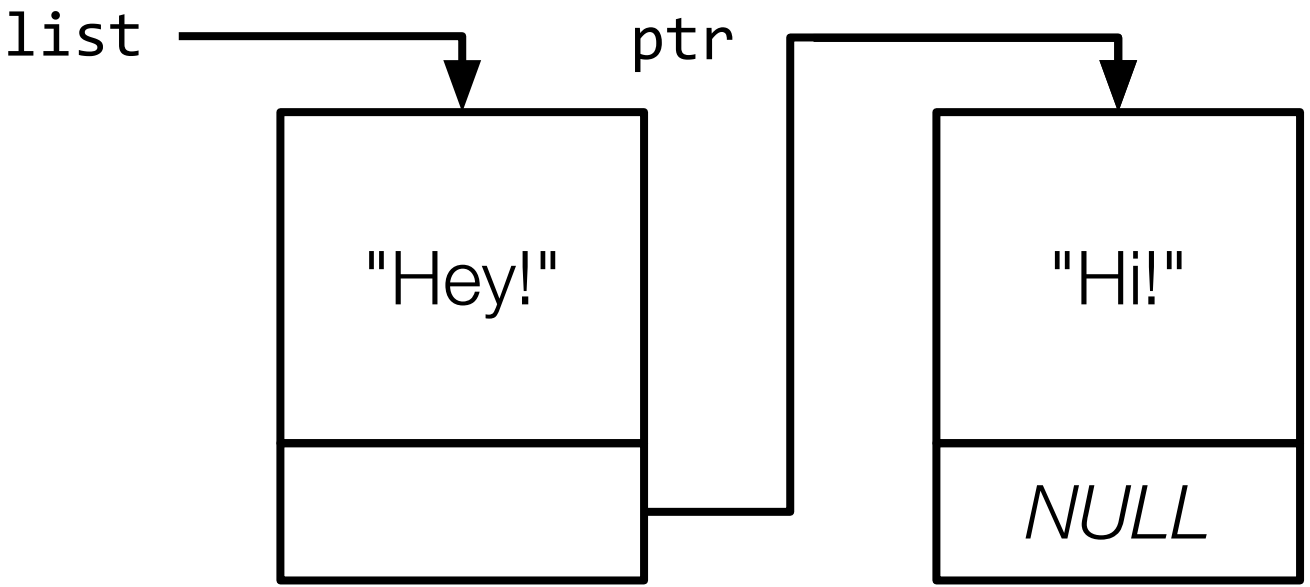


```
free(list);
```

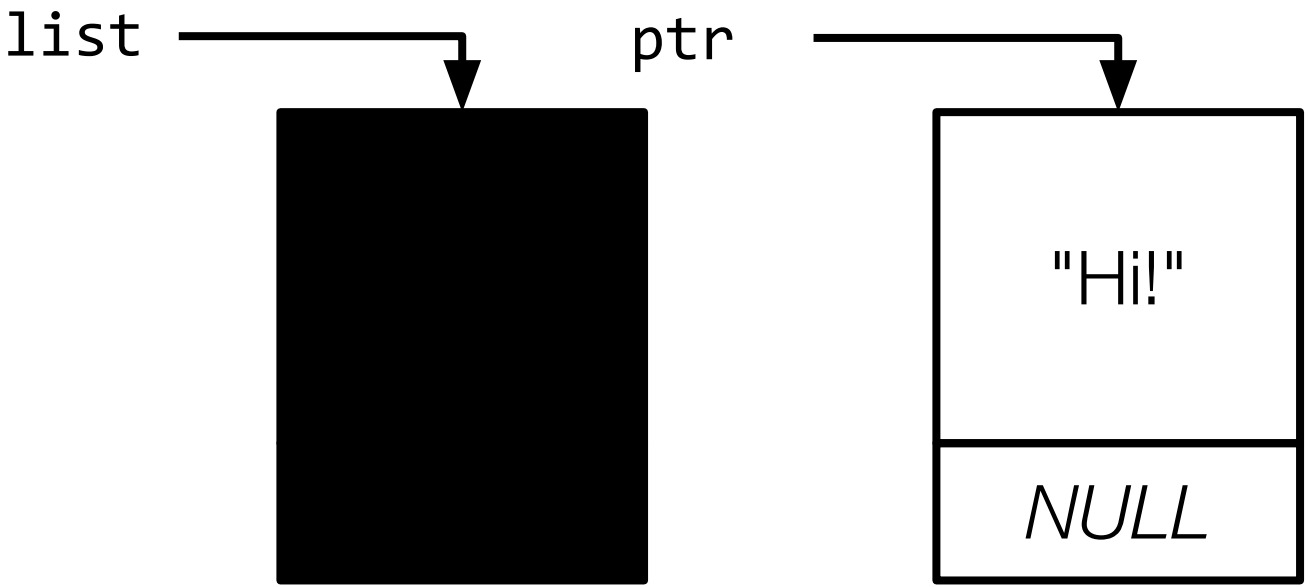



list
"Hey!"
"Hi!"
NULL

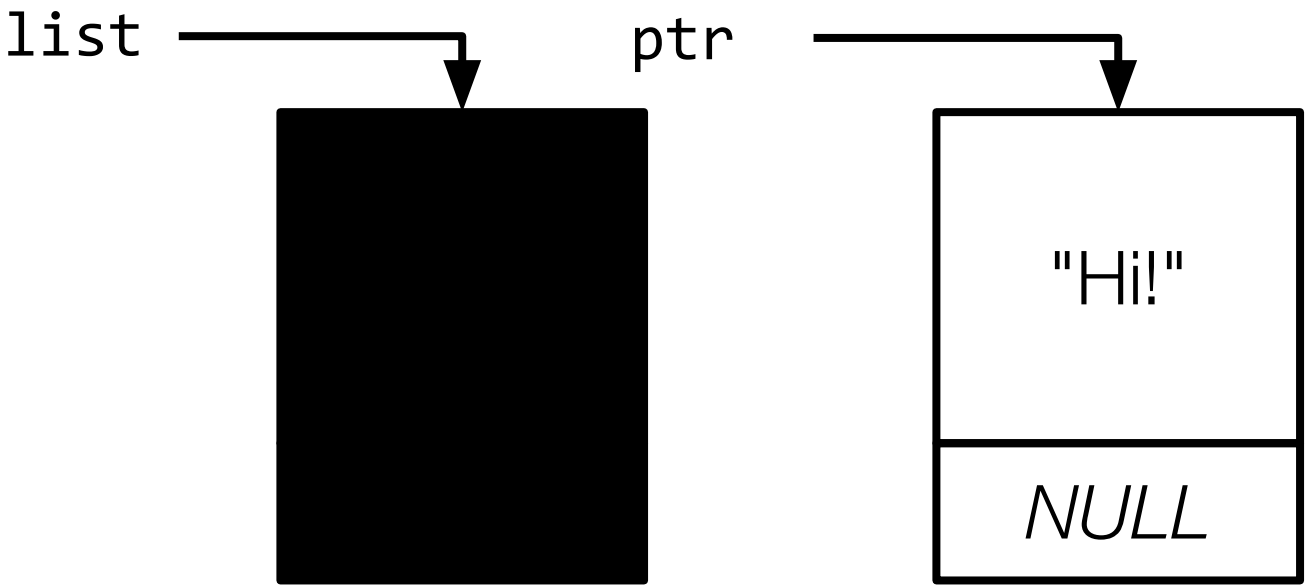
```
node *ptr = list->next;
```



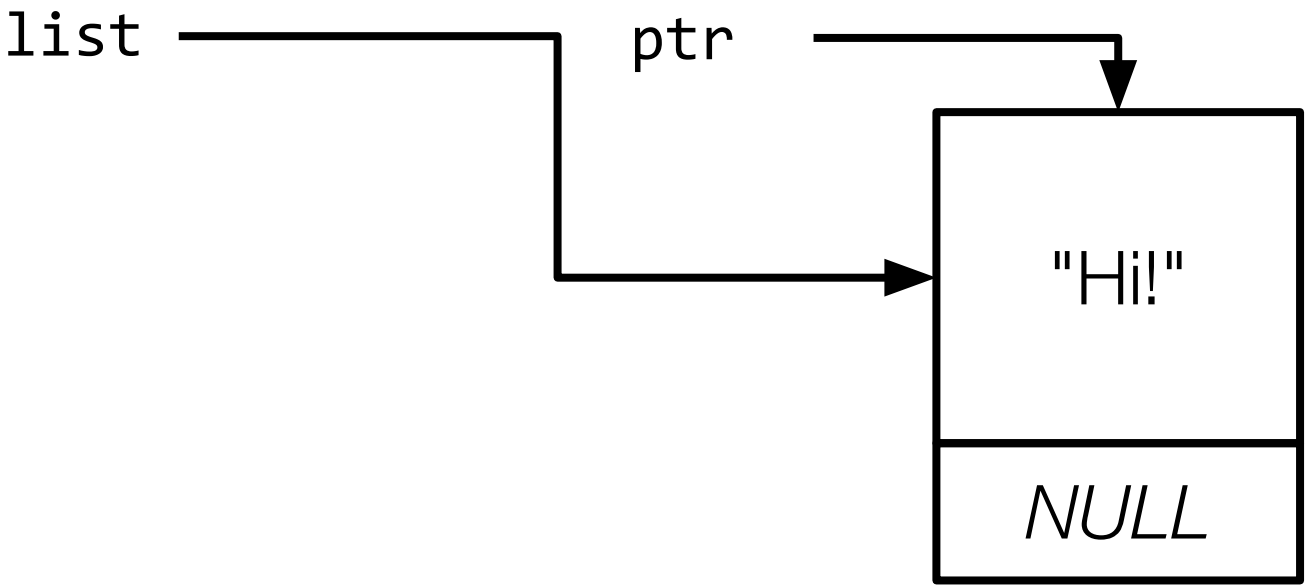
```
free(list);
```



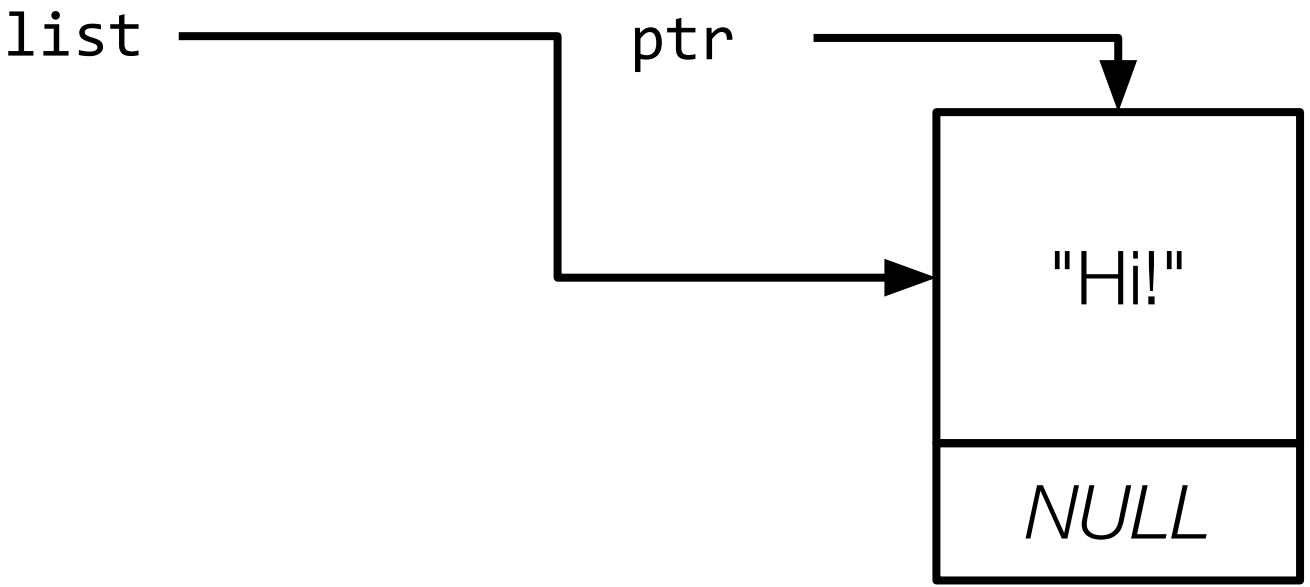
```
list = ptr;
```



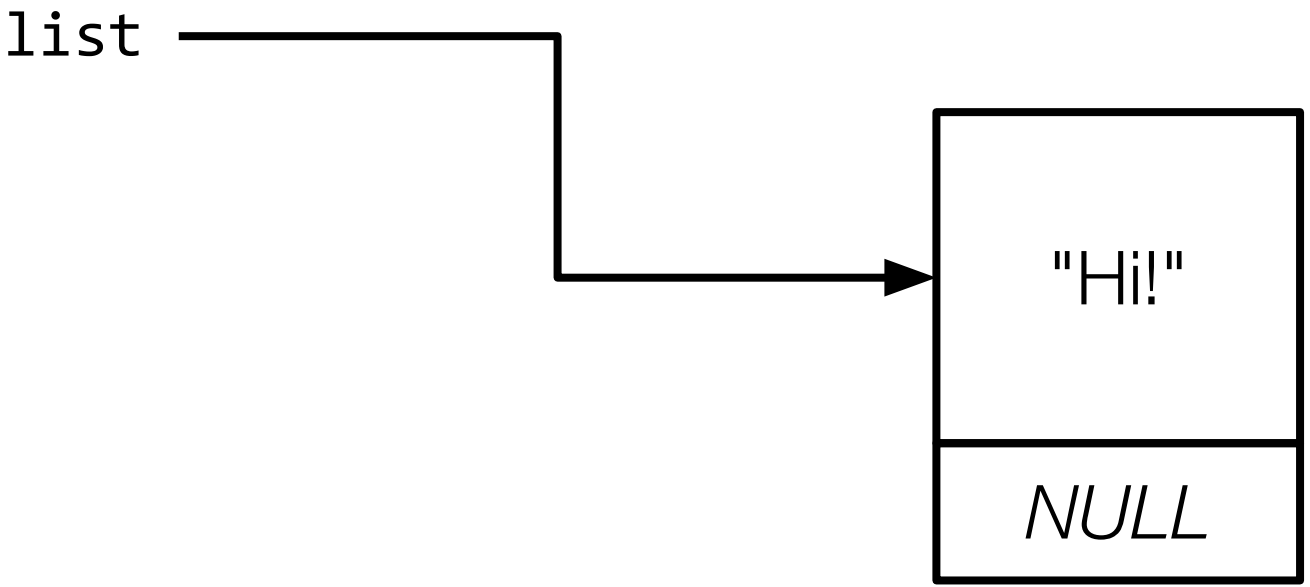
```
list = ptr;
```



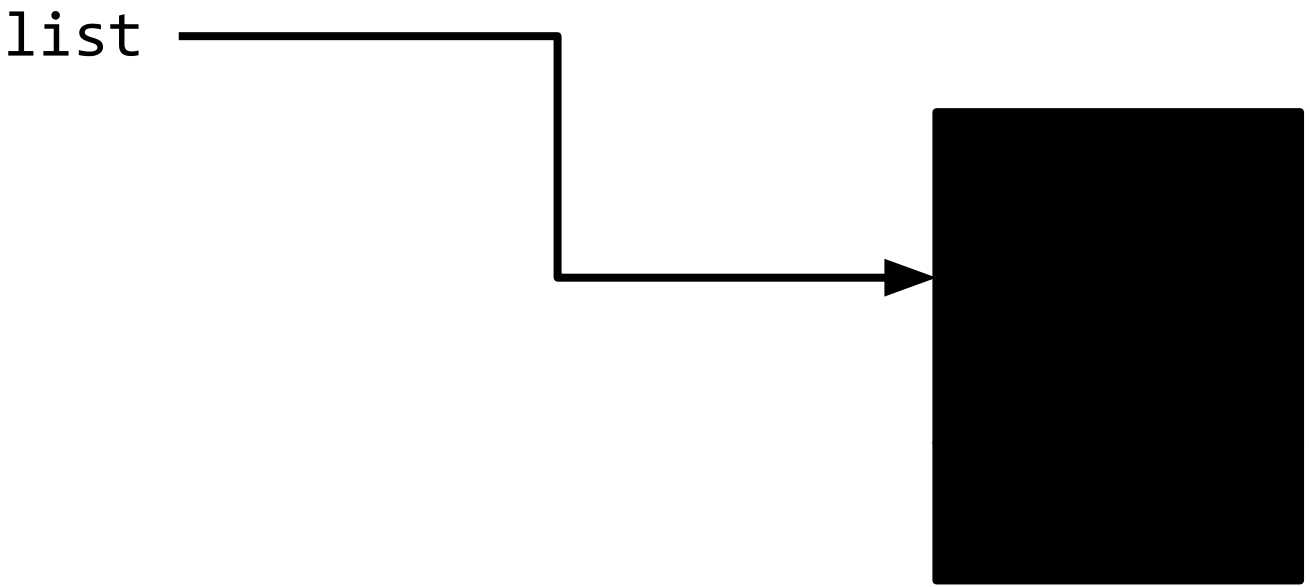
```
ptr = list->next;
```



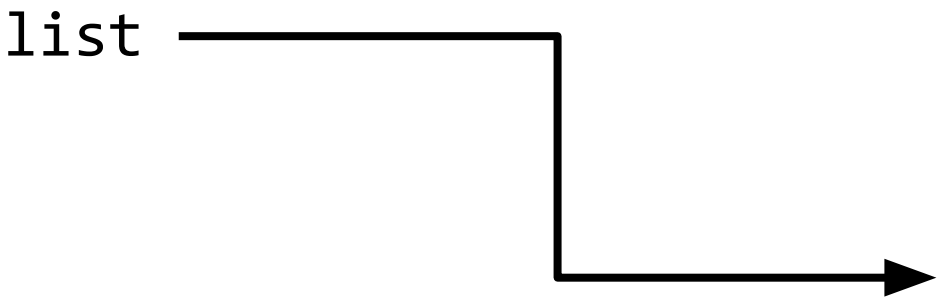
```
ptr = list->next;
```



```
free(list);
```



```
list = ptr;
```



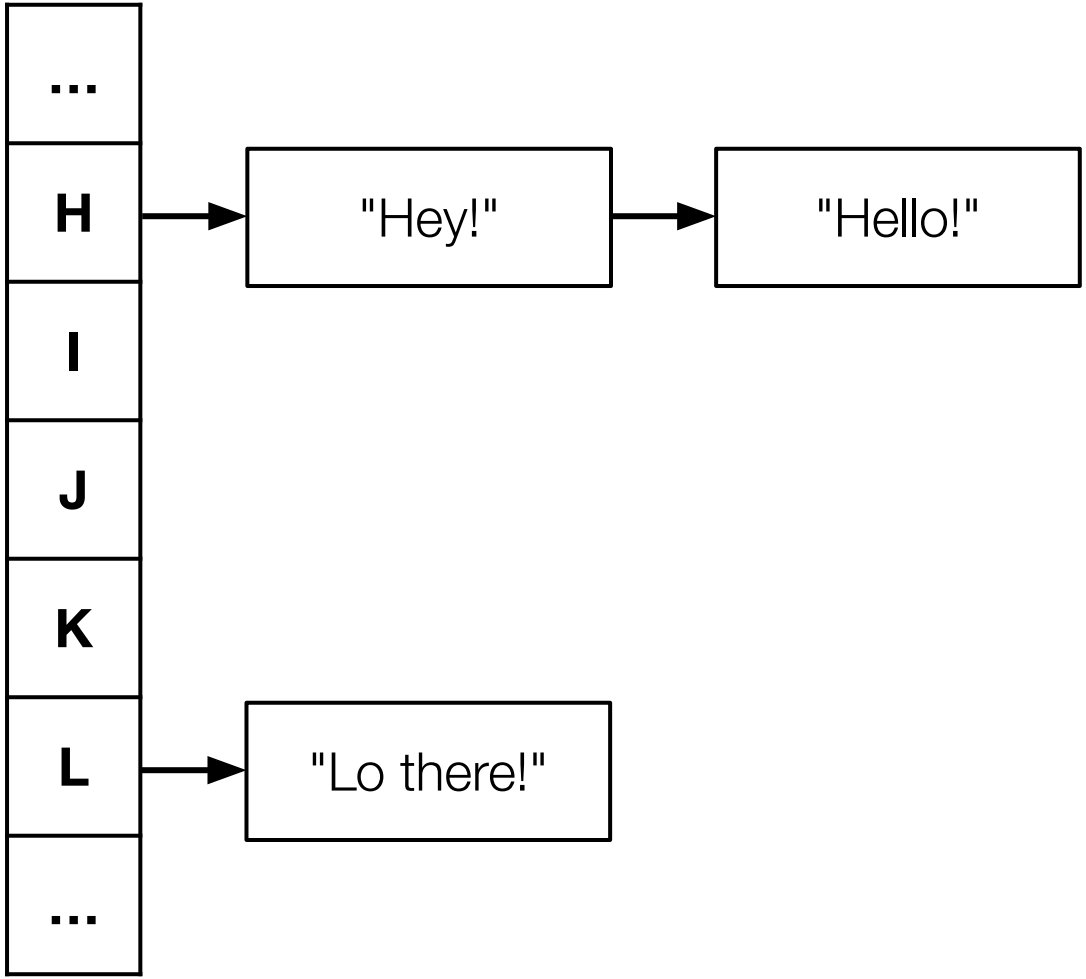
Unloading a Linked List

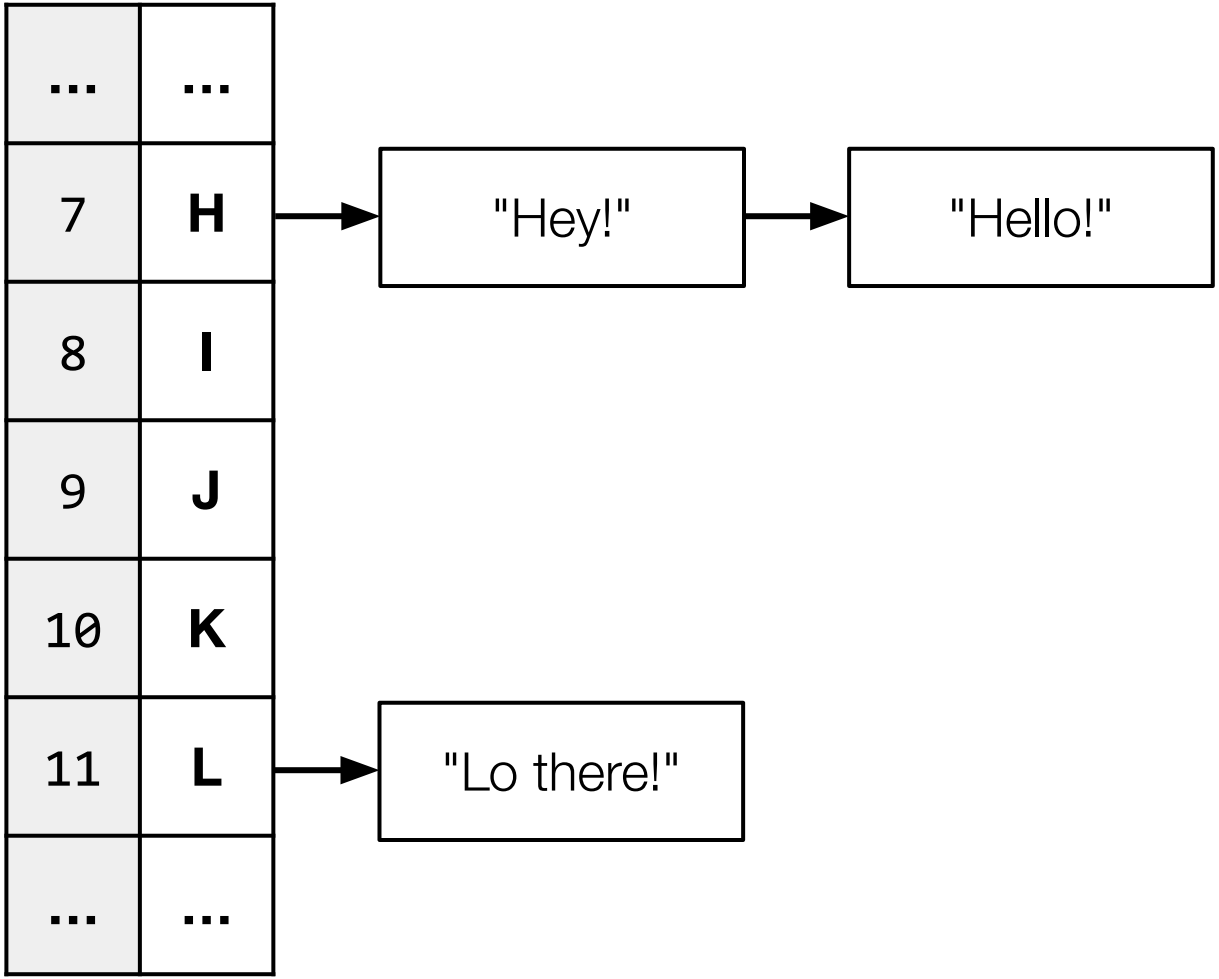
Open the same **list.c** file.

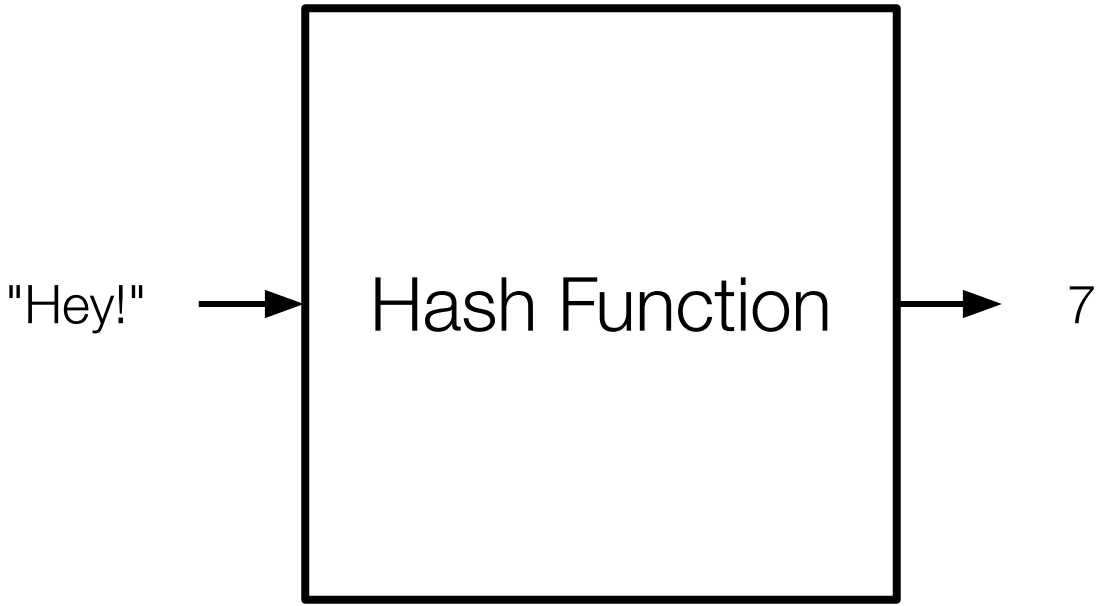
Find the **unload** function below **main**.

Implement **unload** such that all nodes in the linked list are **free**'d when the function is called. Return **true** when successful.


"Hey!"
"Hello!"
"Lo there!"


...
H
I
J
K
L
...
"Hey!"
"Hello!"
"Lo there!"


...
...
7
H
"Hey!"
"Hello!"
8
I
9
J
10
K
11
L
"Lo there!"
...
...


"Hey!"
Hash Function
7

Speller

A good hash function…

Always gives you the same value for the same input

Produces an even distribution across buckets

Uses all buckets

This was CS50