This is CS50

Think.
Pair.
Share.

- 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.**

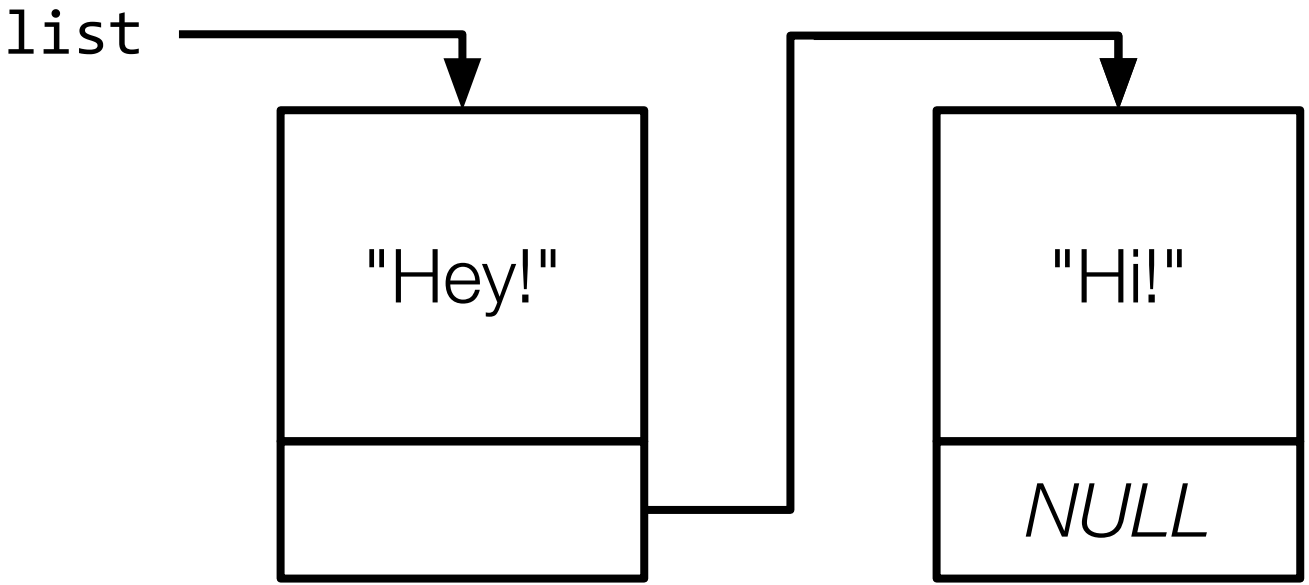
What **data structure** would you propose the team build to store these words?

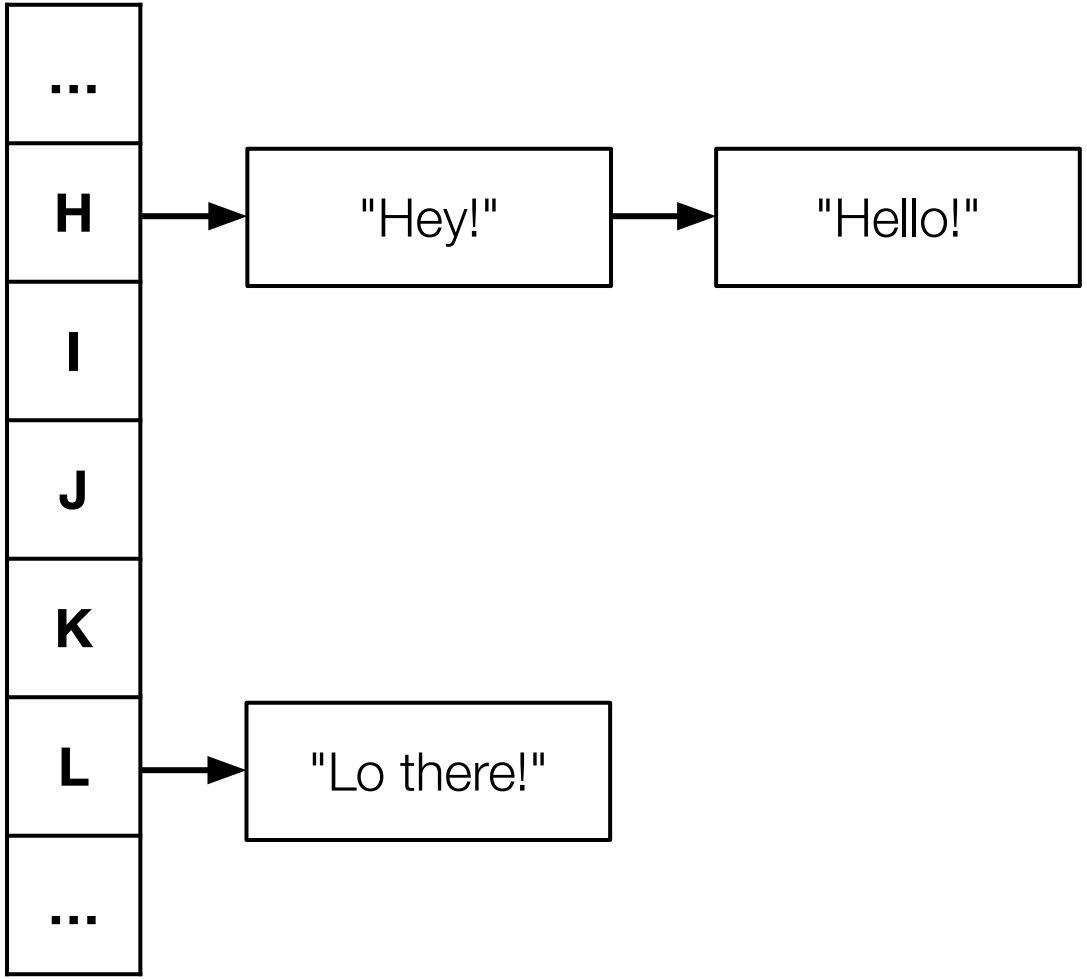
Deletion
Insertion
Search

1. Search
2. Insertion
3. Deletion

1. Insertion
2. Search
3. Deletion

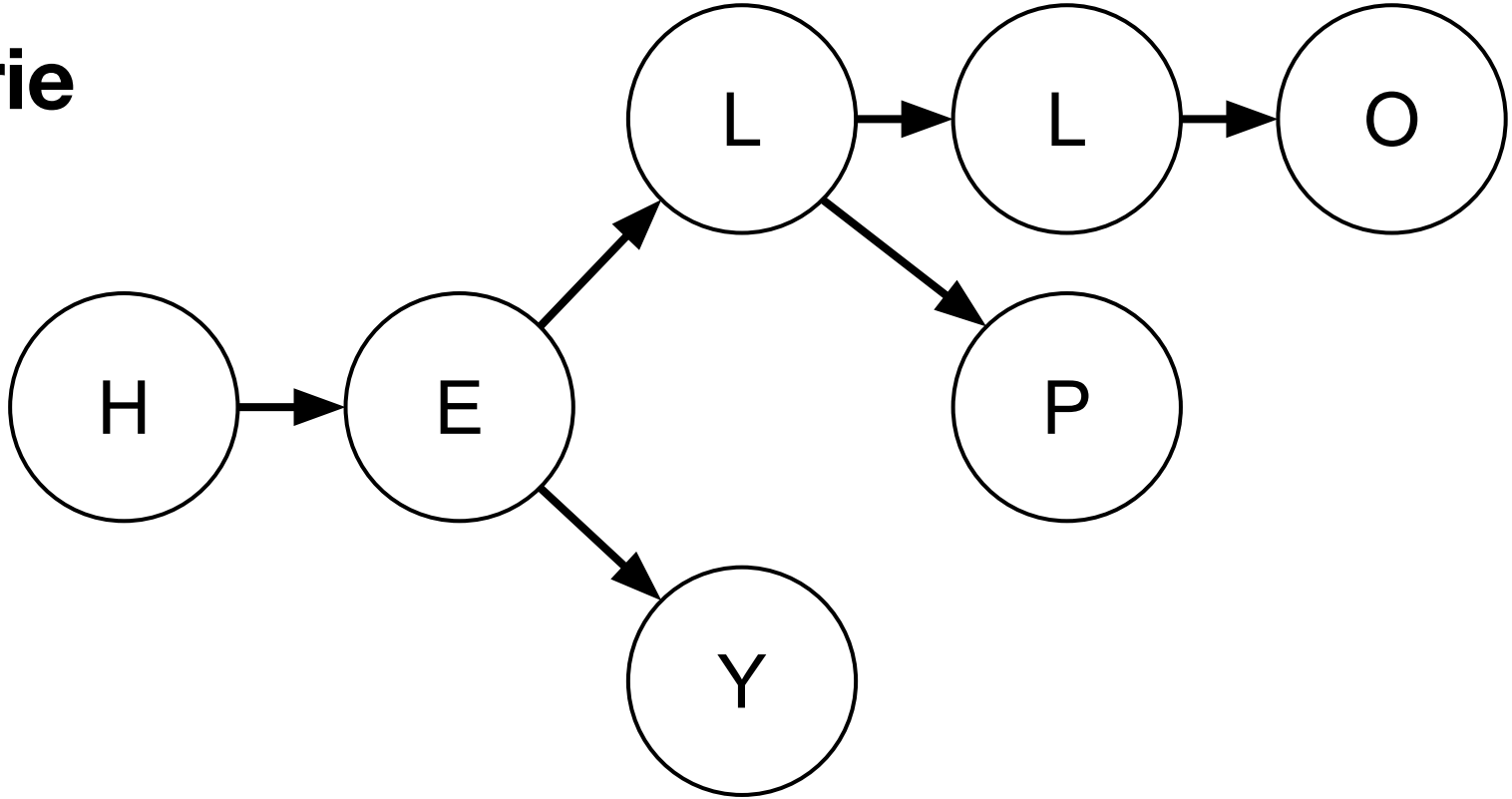
Linked List



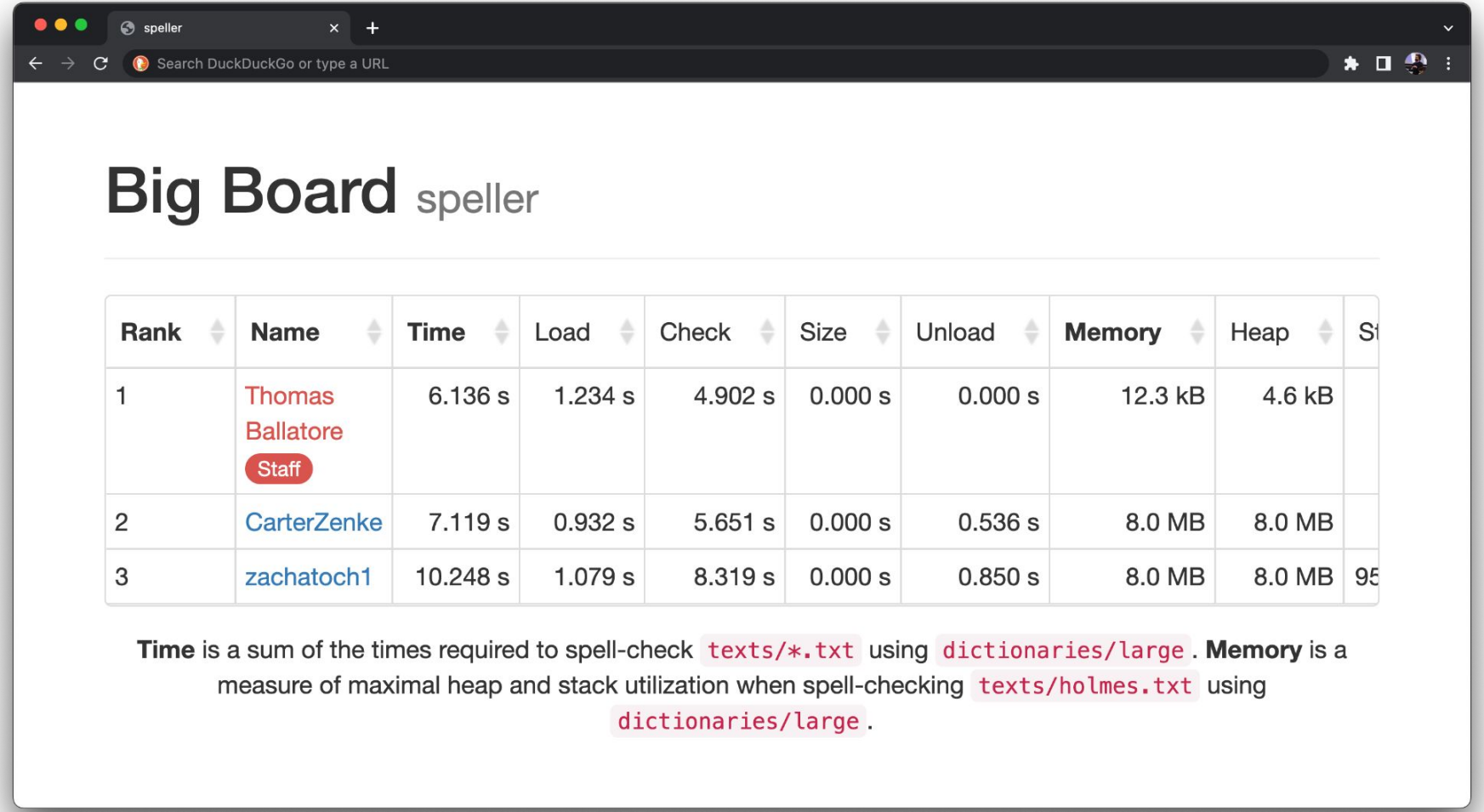


Hash Table

Trie



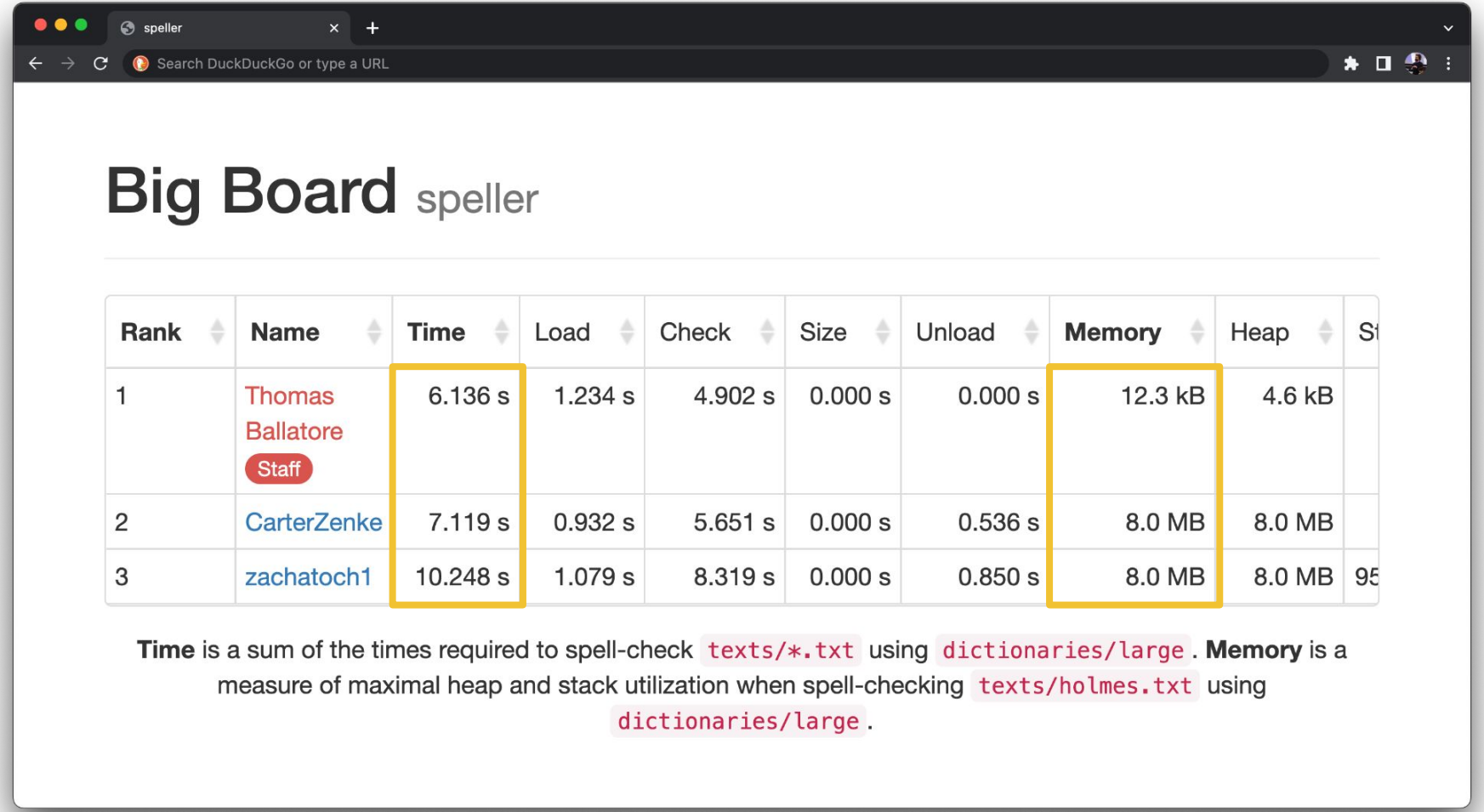
Trade-offs



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`.

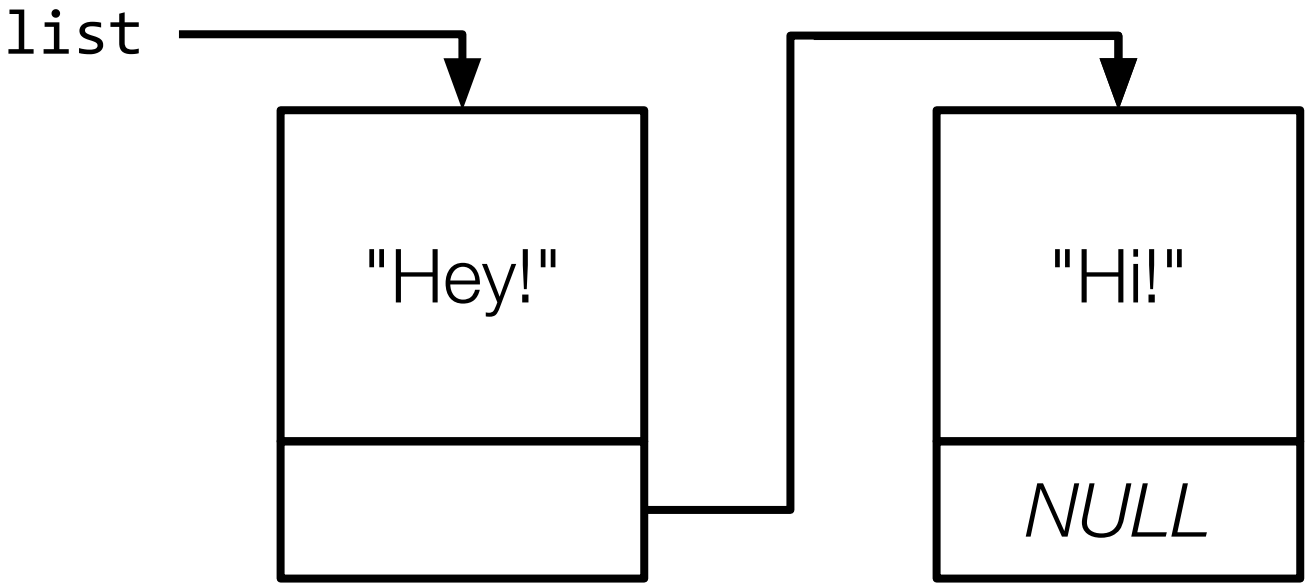


Big Board speller

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`.

Linked List



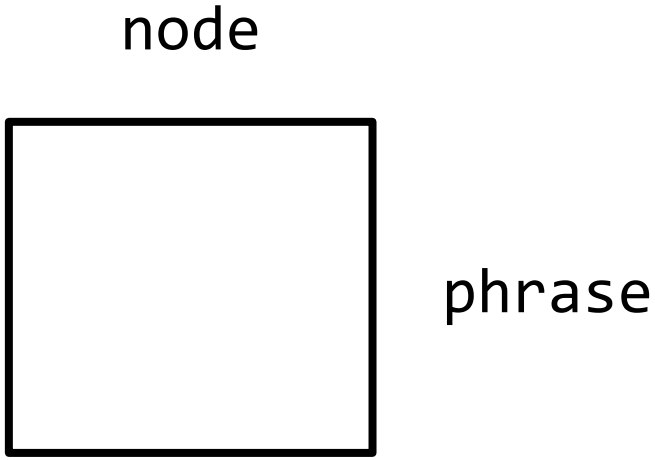
Nodes

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

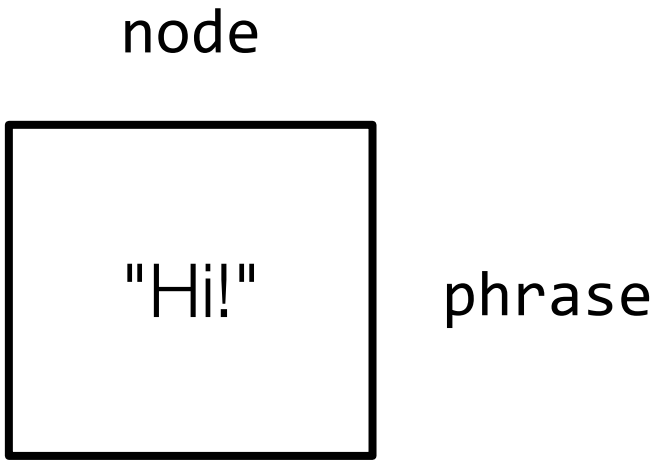
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```

node

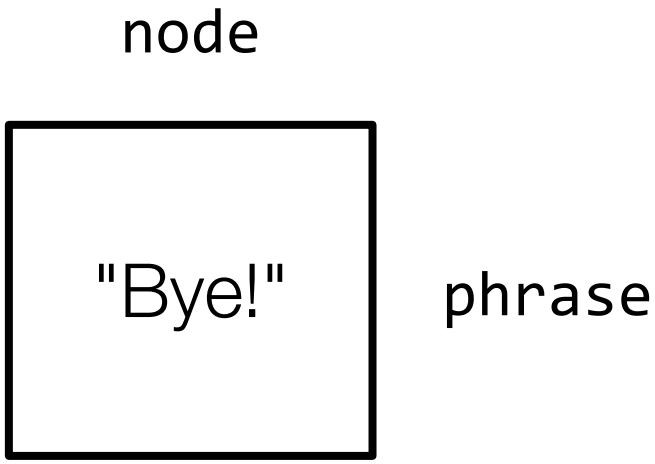
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



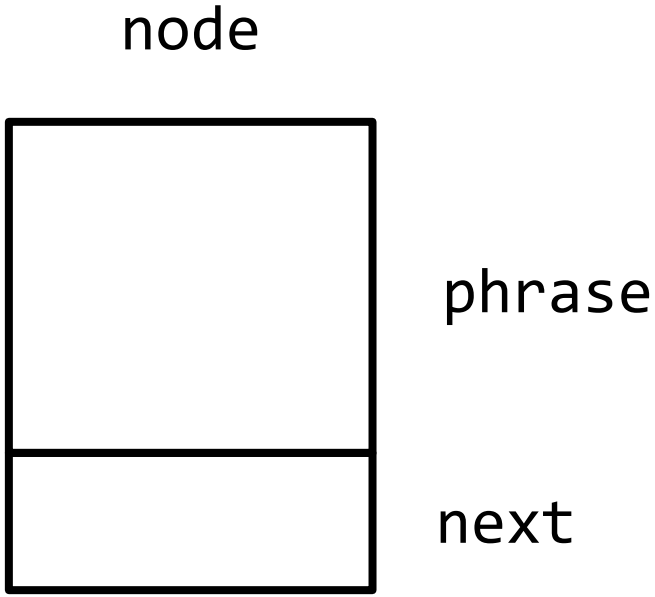
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



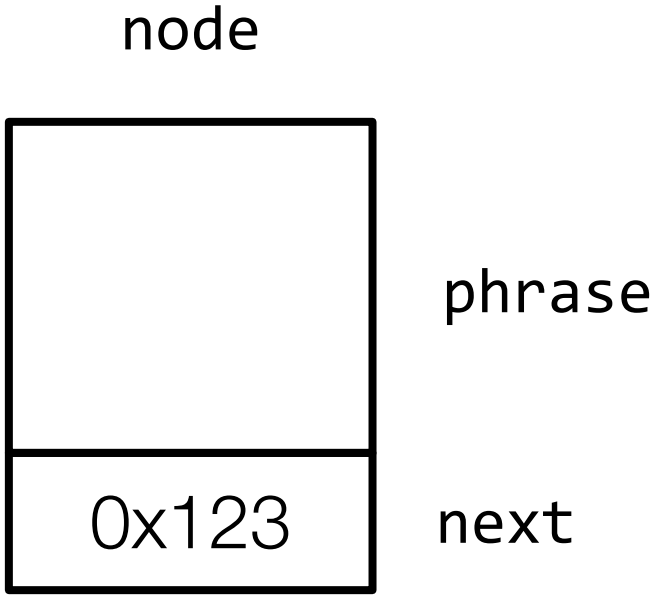
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



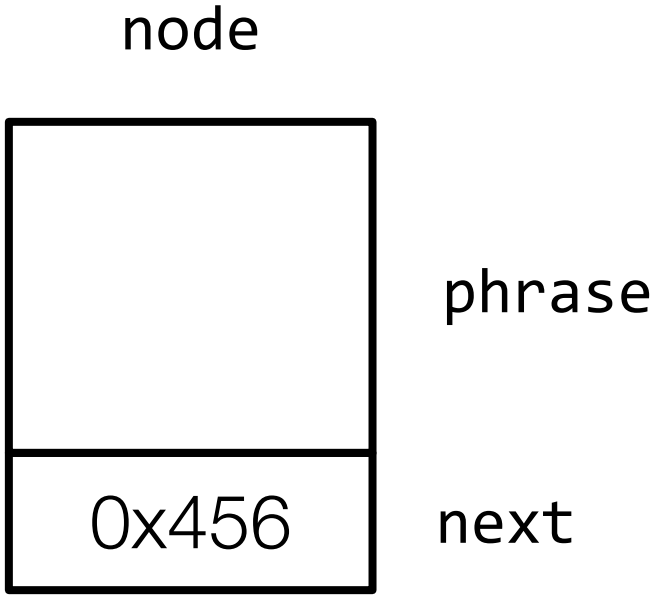
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



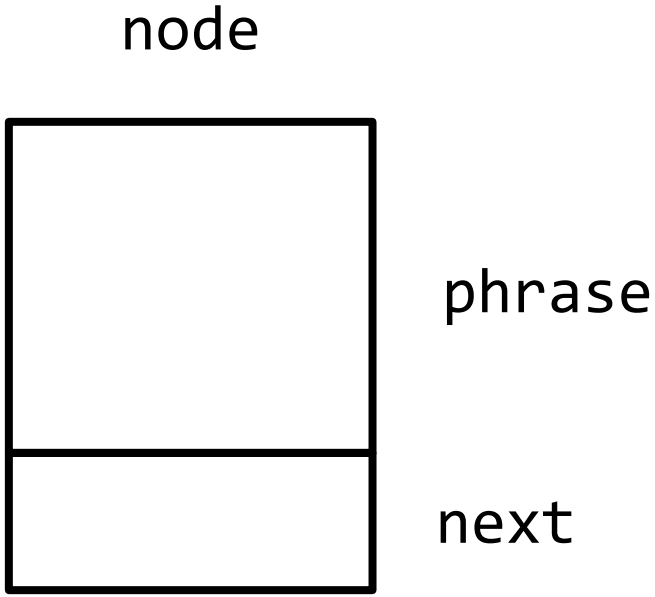
```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



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

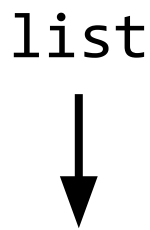


```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```



Creating a Linked List

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

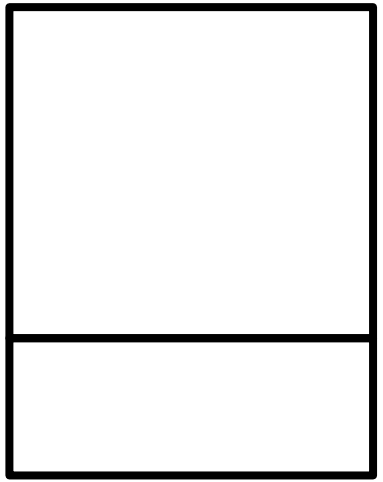


```
node *n = malloc(sizeof(node));
```

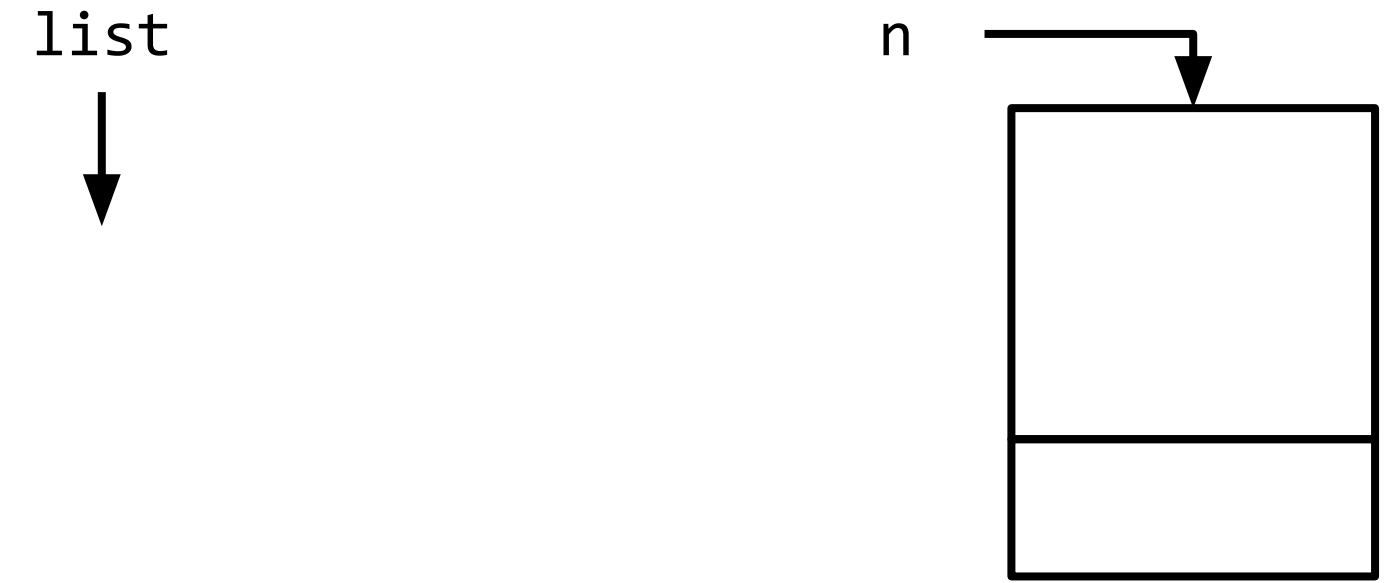
list

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

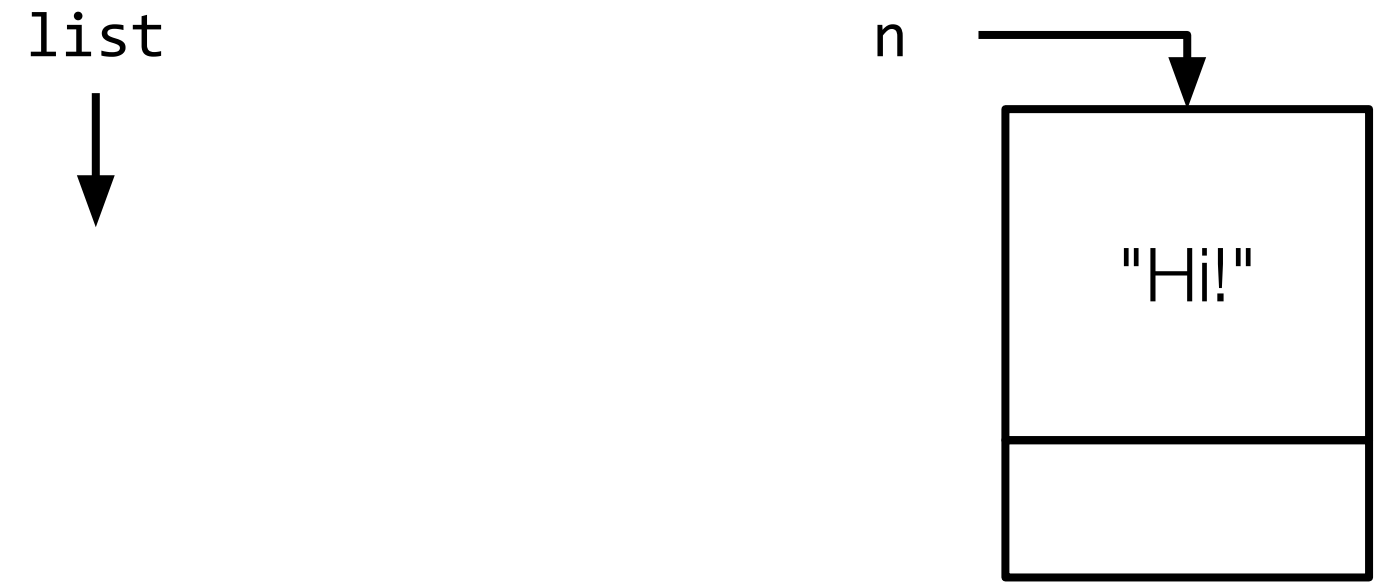
list



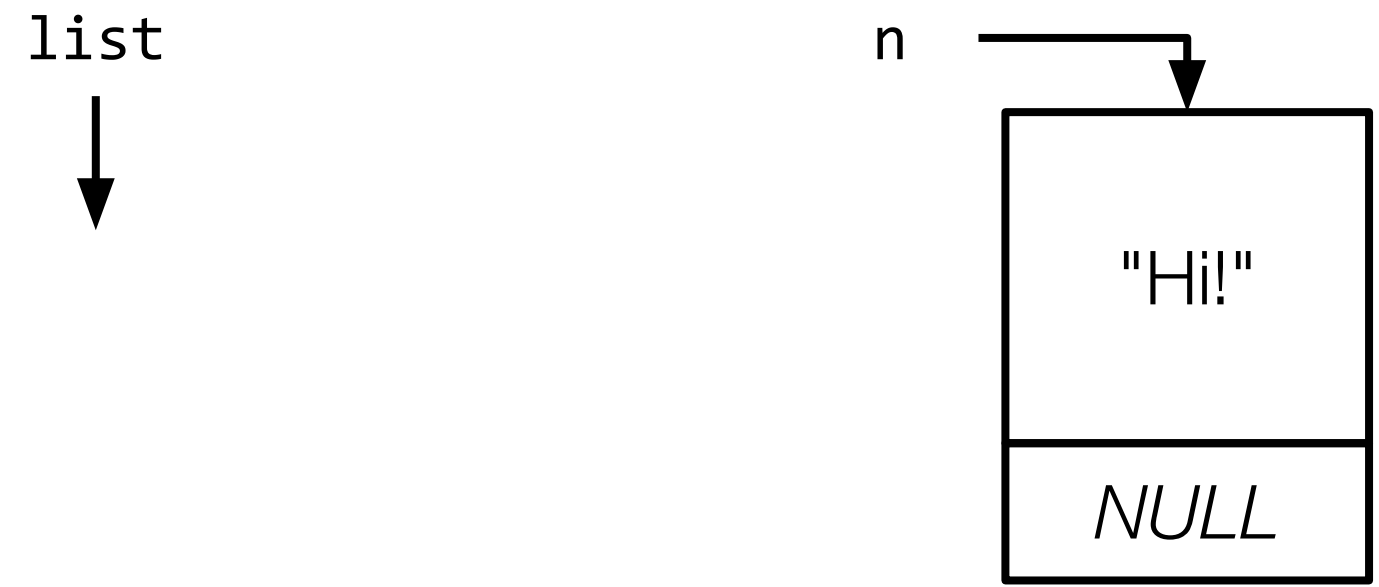
```
node *n = malloc(sizeof(node));
```



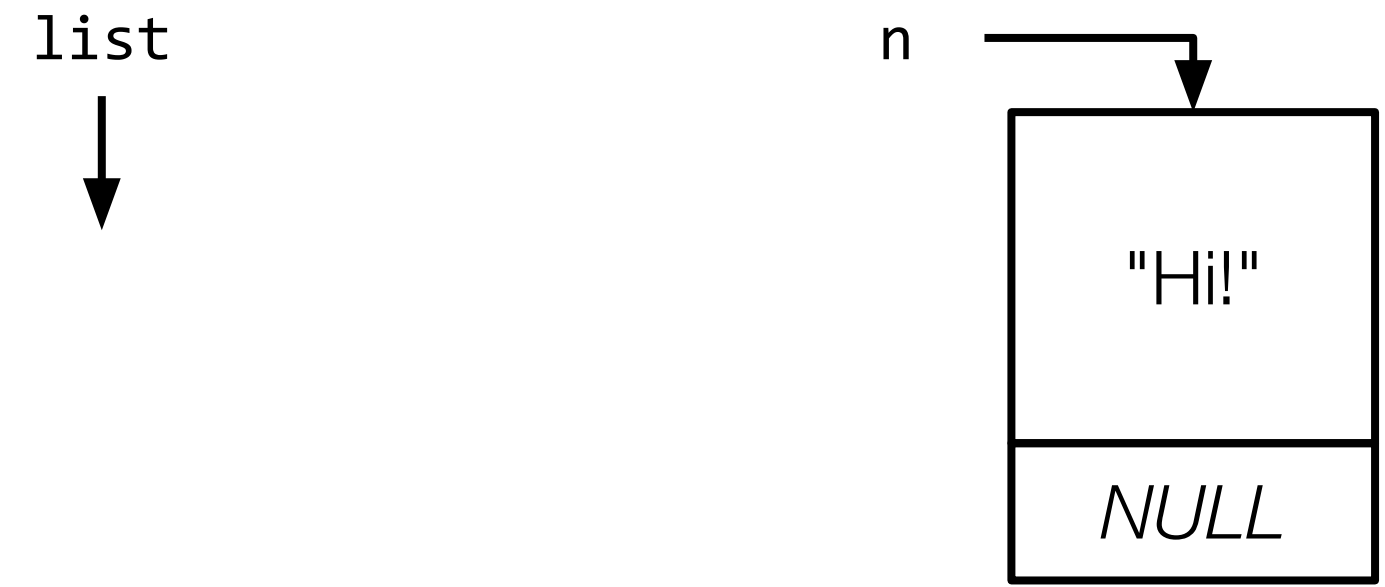
```
node *n = malloc(sizeof(node));
n->phrase = "Hi!";
```



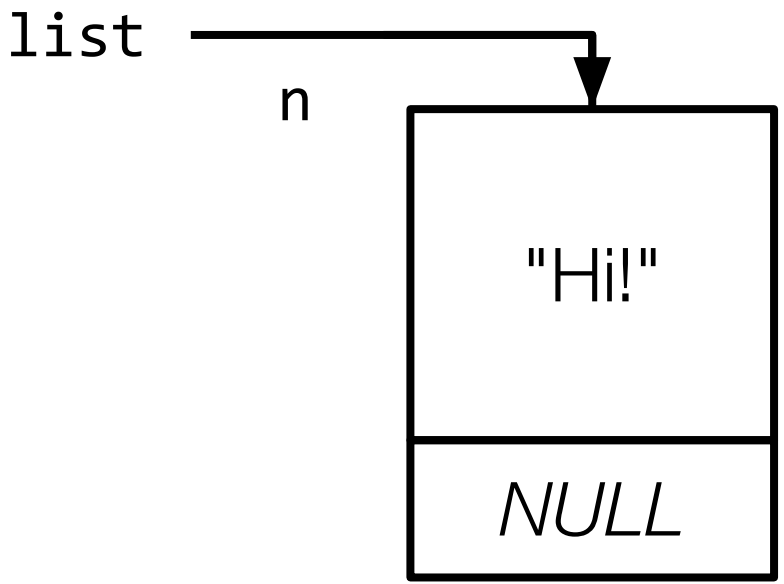
```
node *n = malloc(sizeof(node));
n->phrase = "Hi!";
n->next = NULL;
```



```
list = n;
```

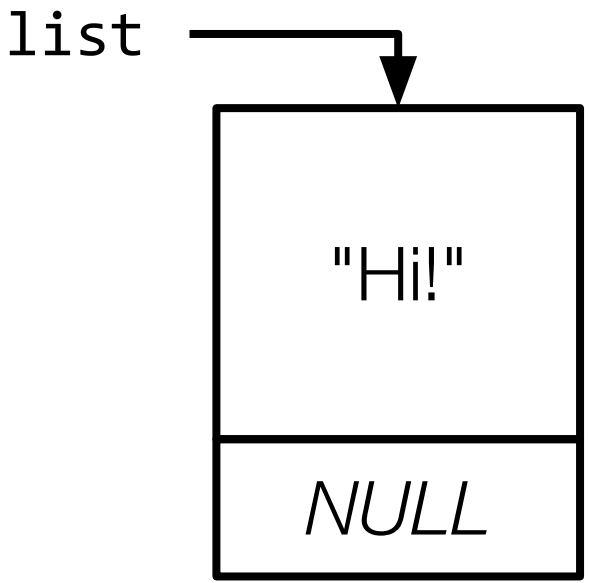


```
list = n;
```

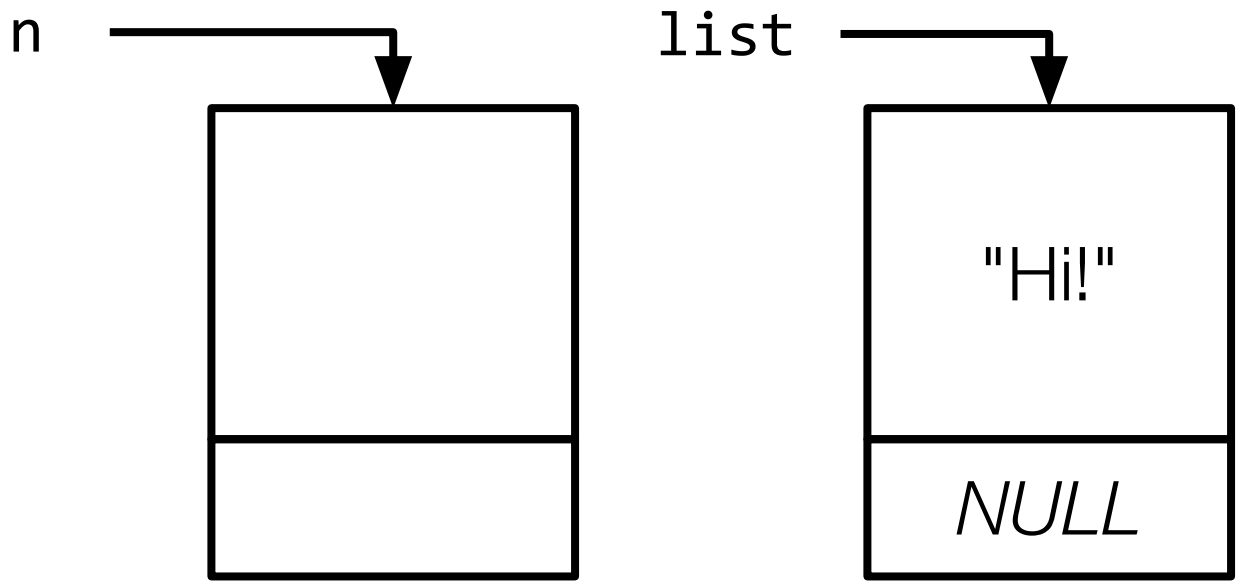


Inserting Nodes

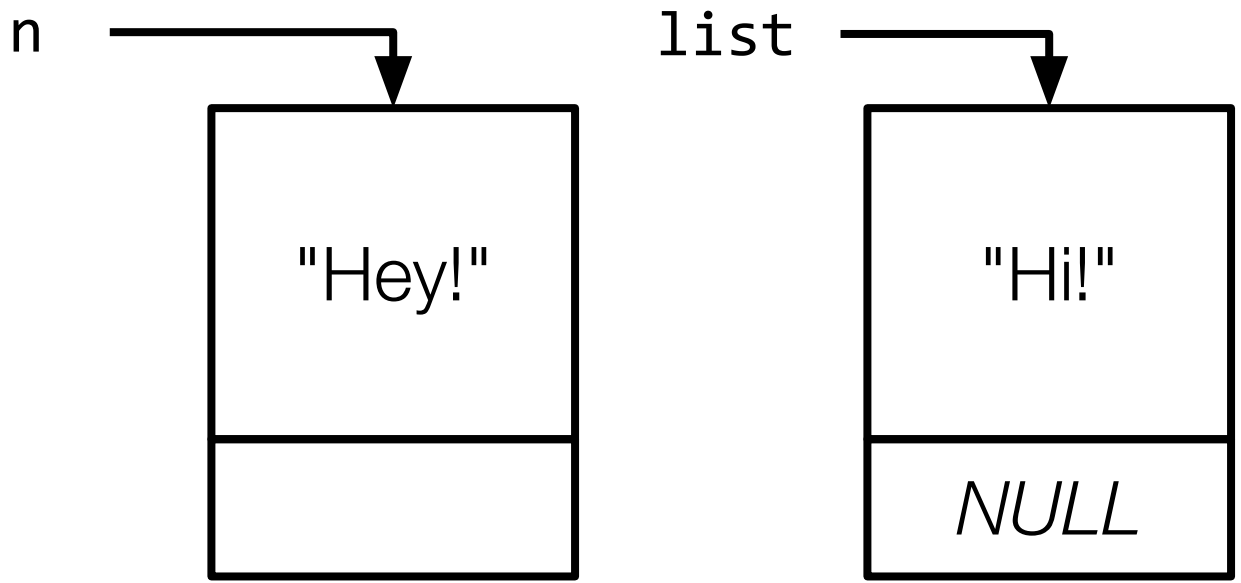
```
n = malloc(sizeof(node));
```



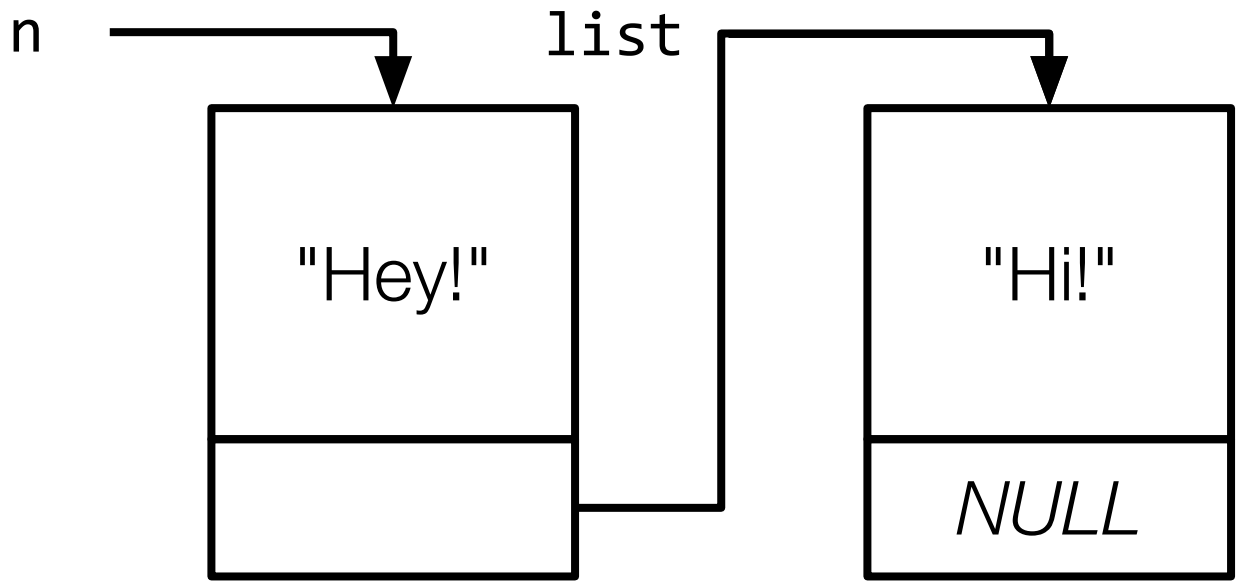
```
n = malloc(sizeof(node));
```



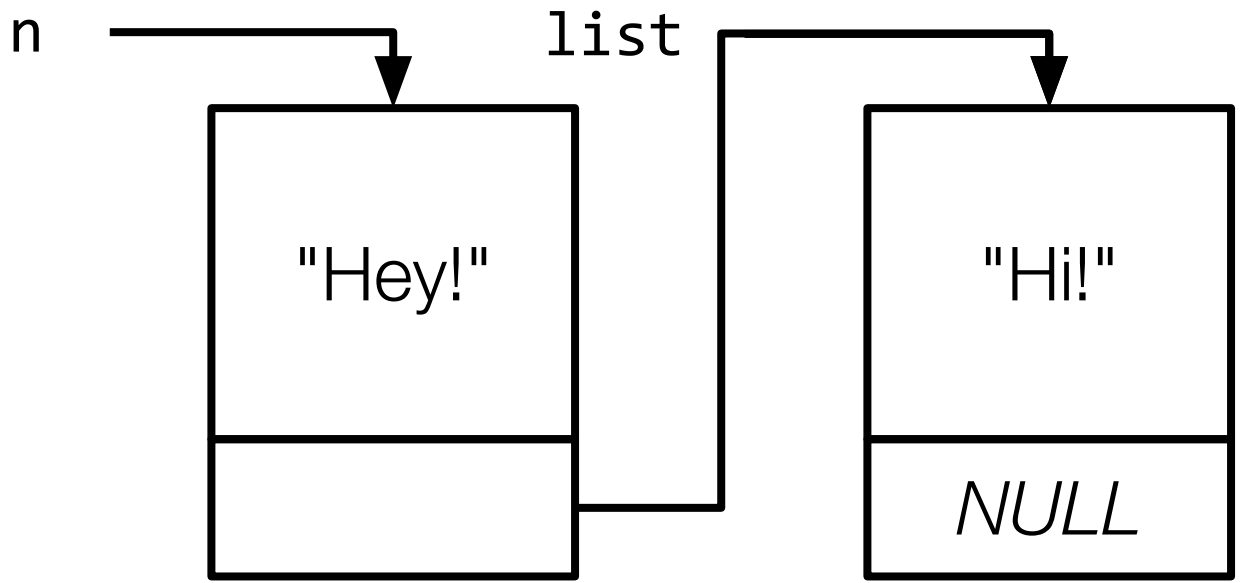
```
n = malloc(sizeof(node));
n->phrase = "Hey!";
```



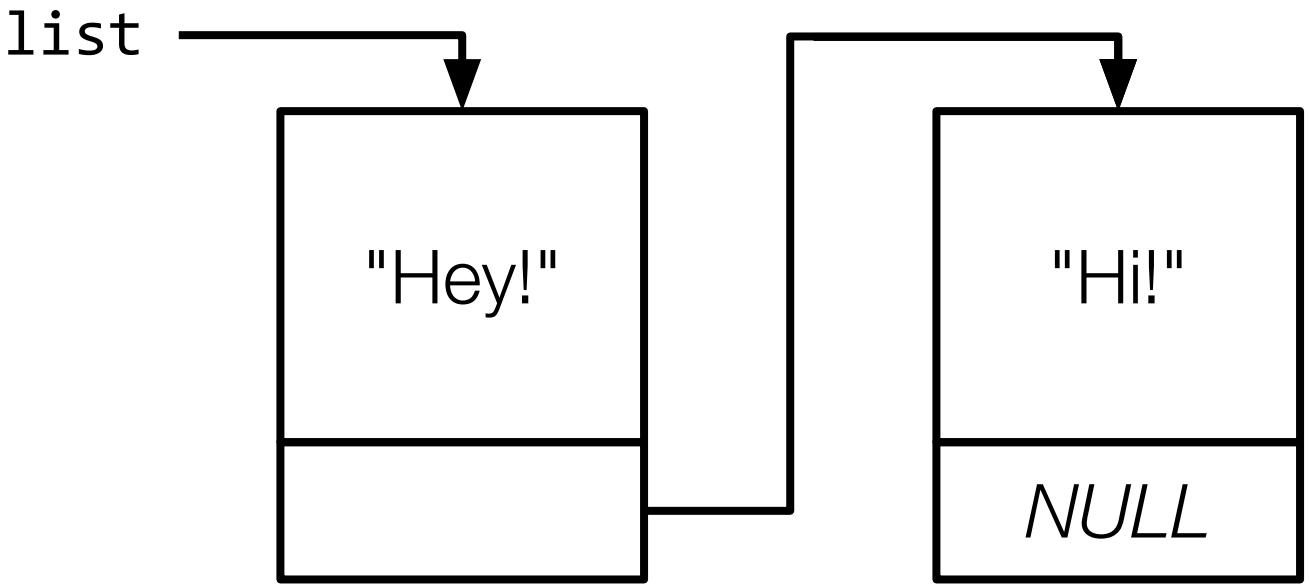
```
n = malloc(sizeof(node));
n->phrase = "Hey!";
n->next = list;
```



```
list = n;
```



```
list = n;
```

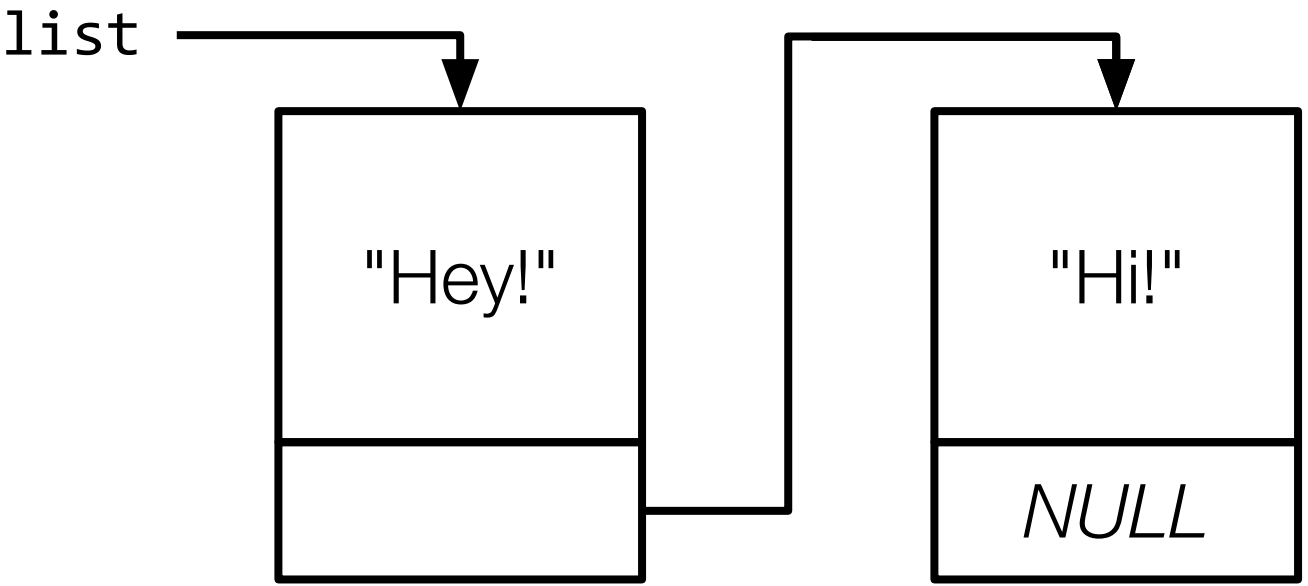


Inserting into a Linked List

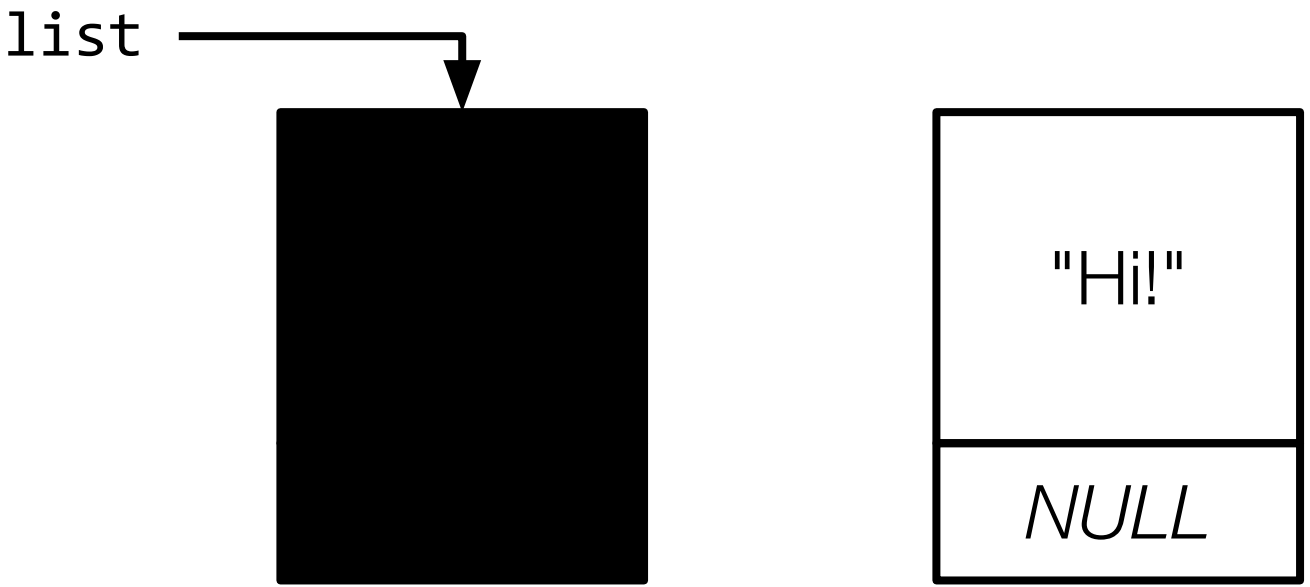
Download and open **list.c**.

Find the first TODO.

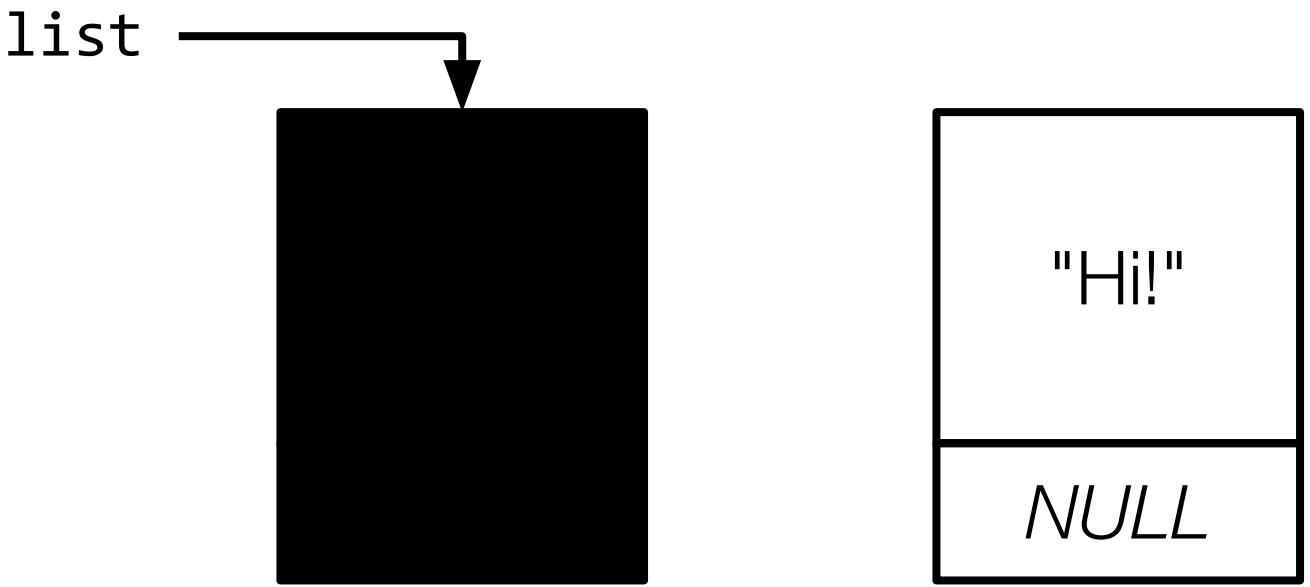
Starting below that TODO, implement code to add a node to the linked list. Ensure that **list** always points to the head of the linked list. Also ensure your new node contains a phrase.

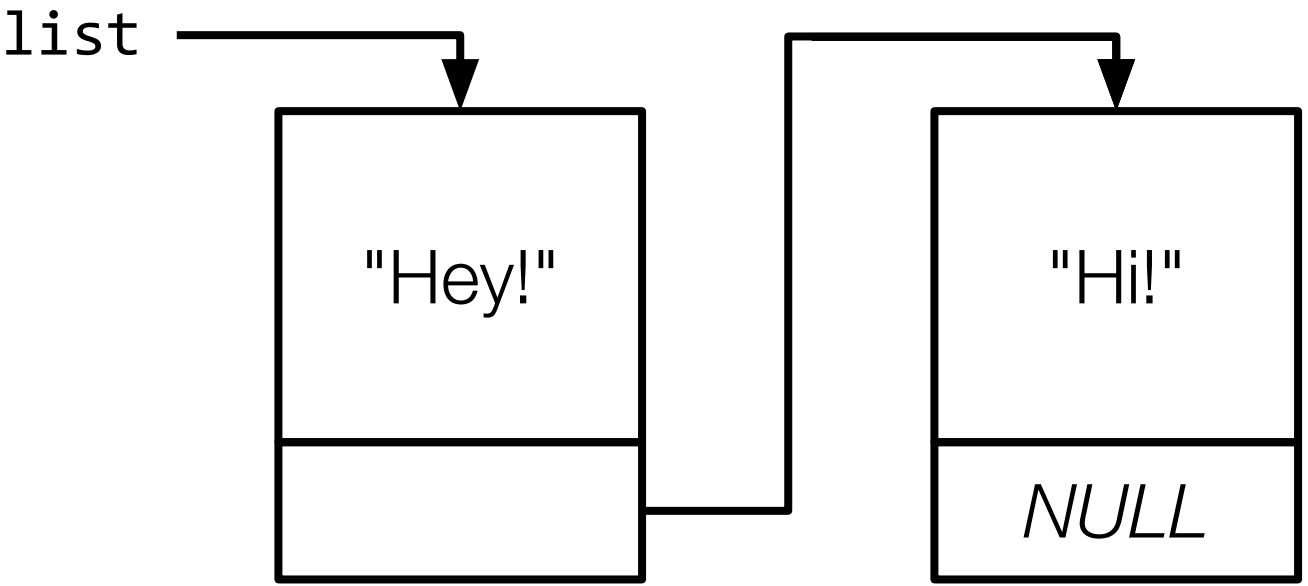

list
"Hey!"
"Hi!"
NULL

```
free(list);
```

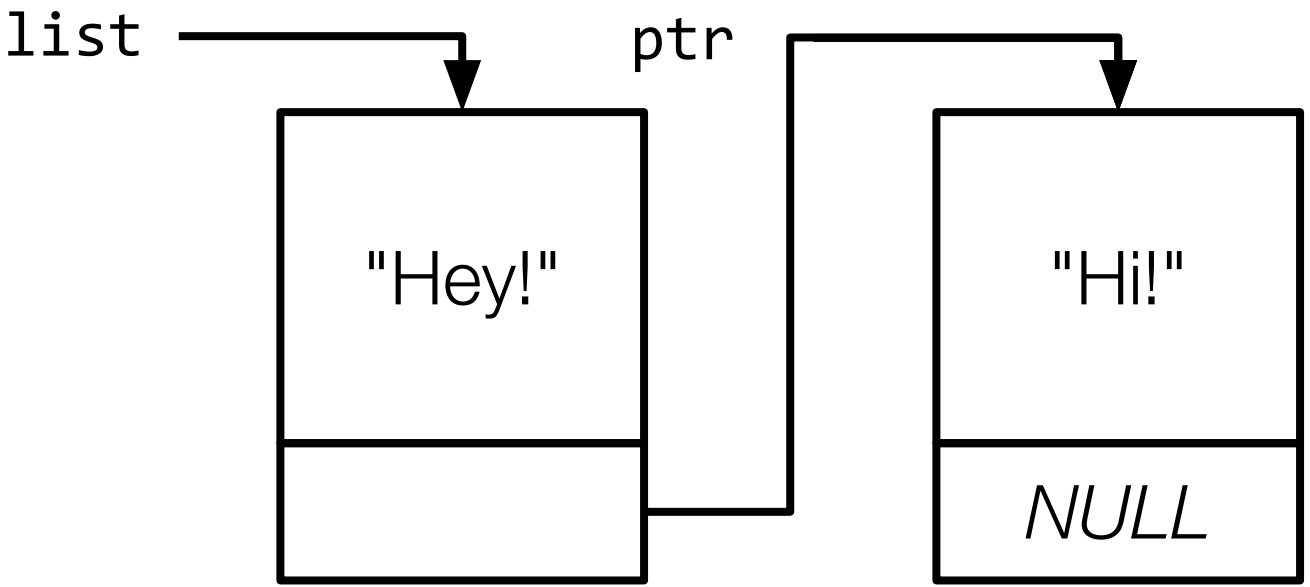


```
free(list);
```

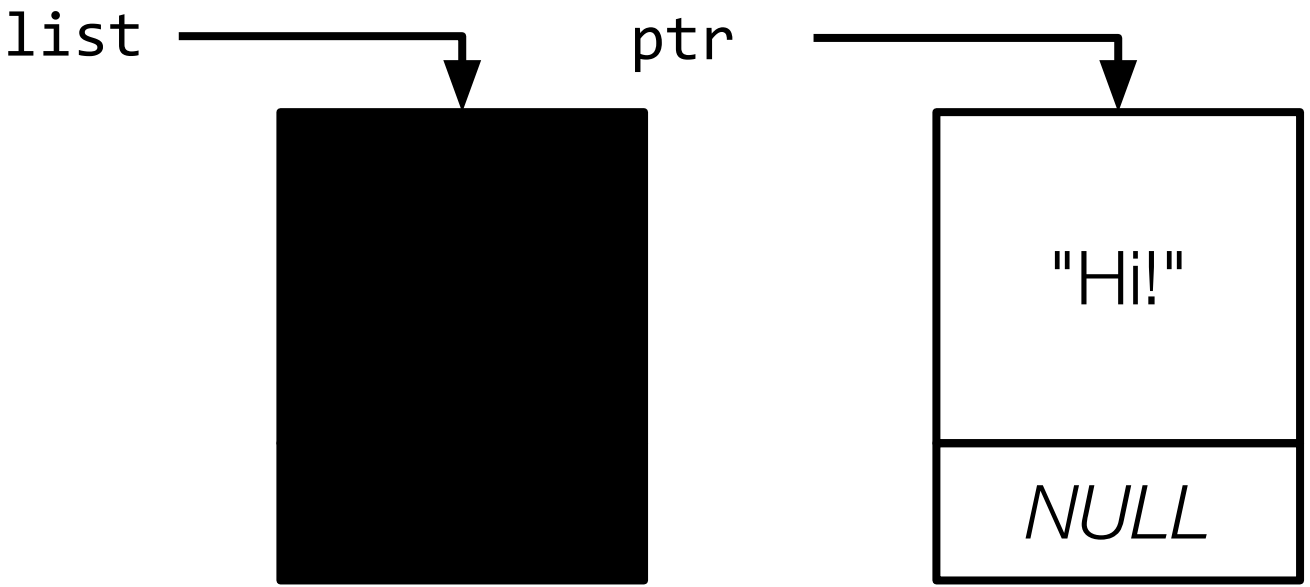



list
"Hey!"
"Hi!"
NULL

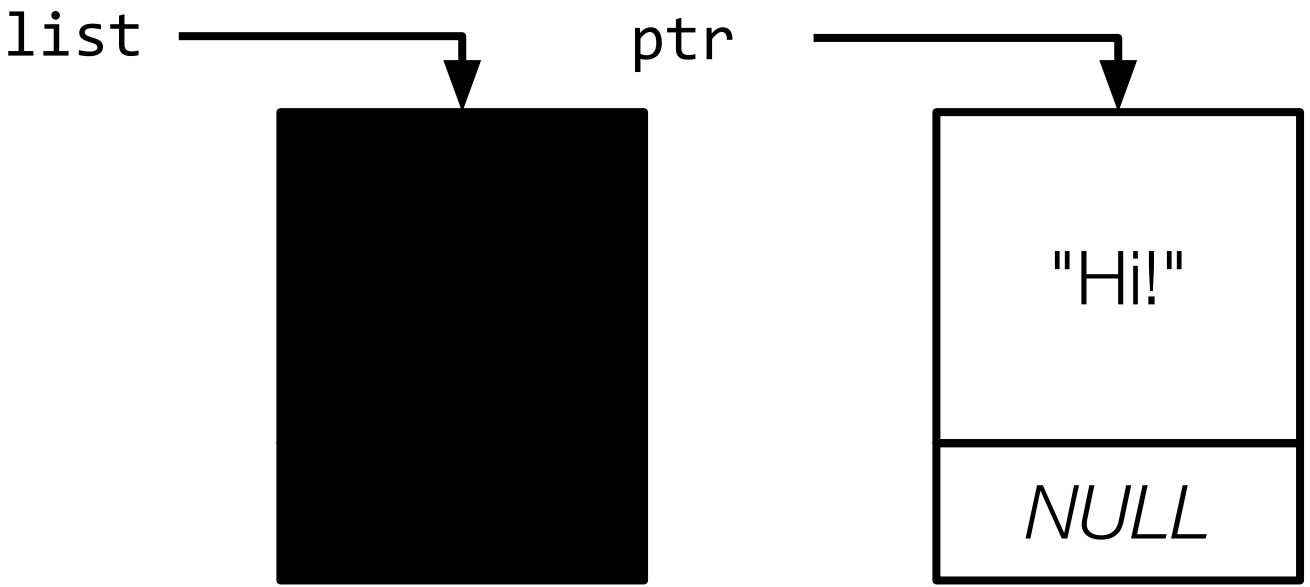
```
node *ptr = list->next;
```



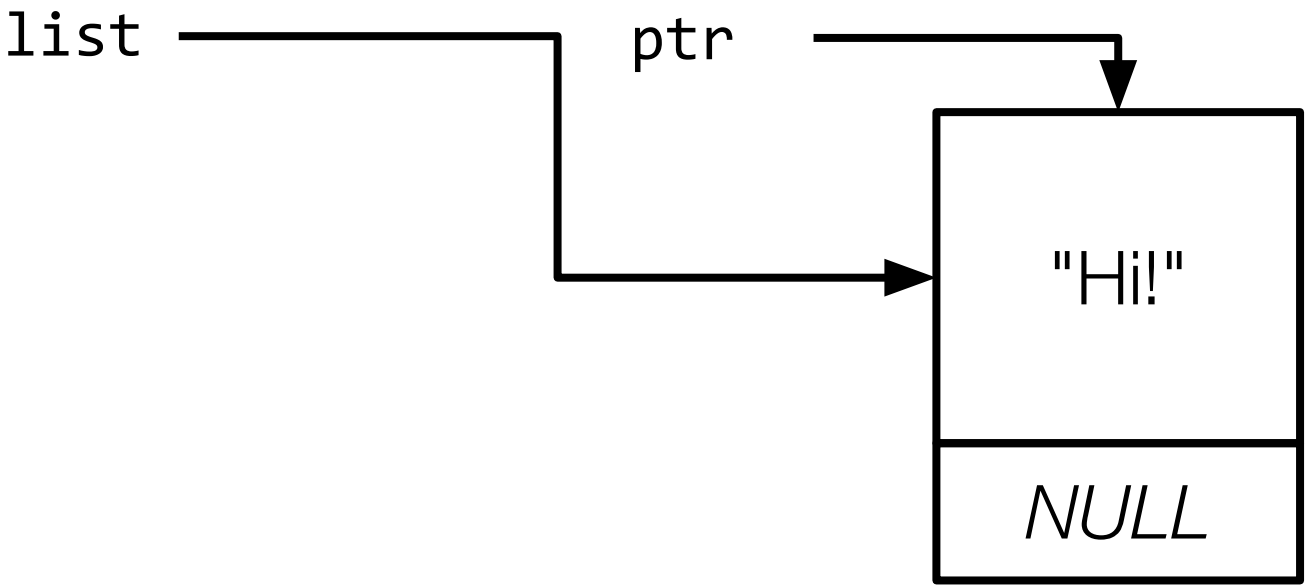
```
free(list);
```



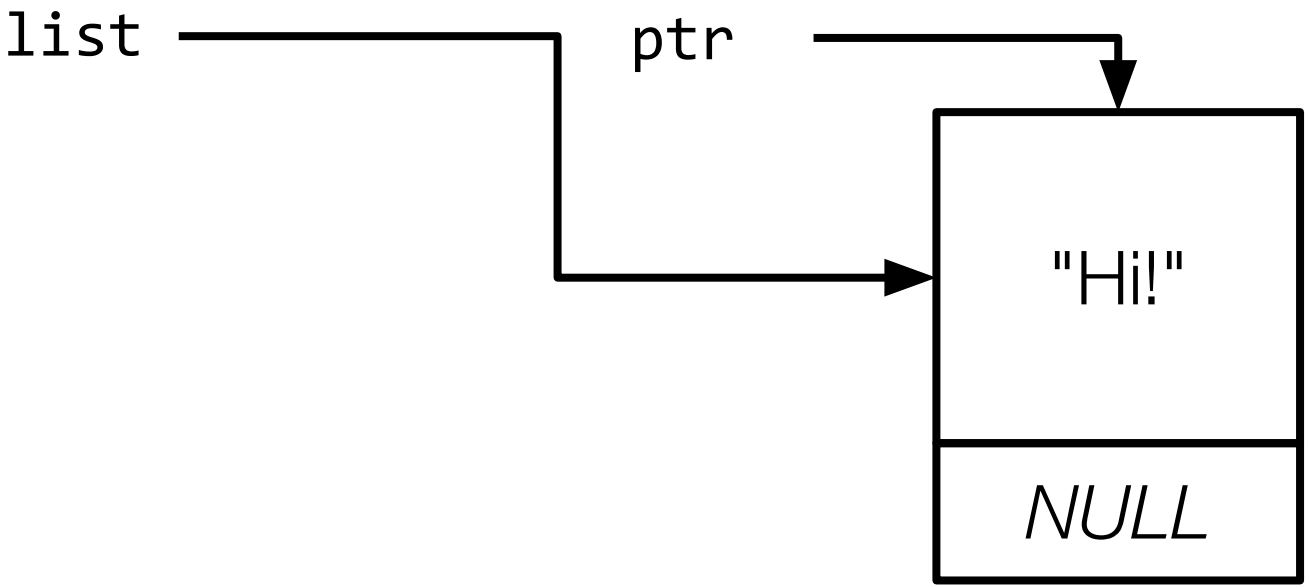
```
list = ptr;
```



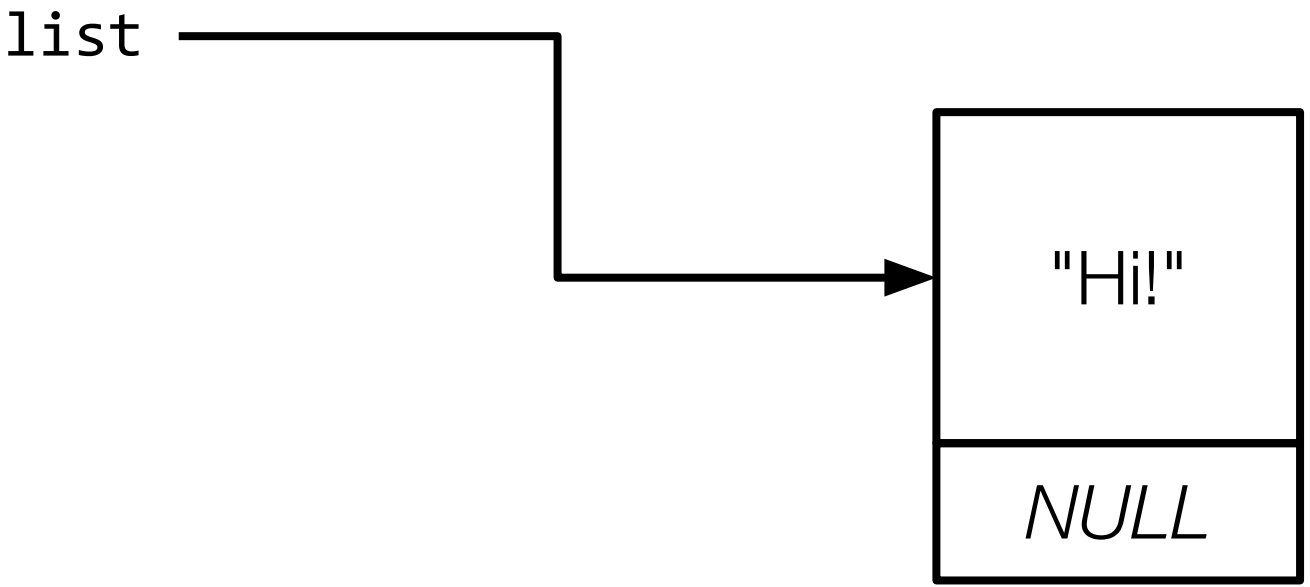
```
list = ptr;
```



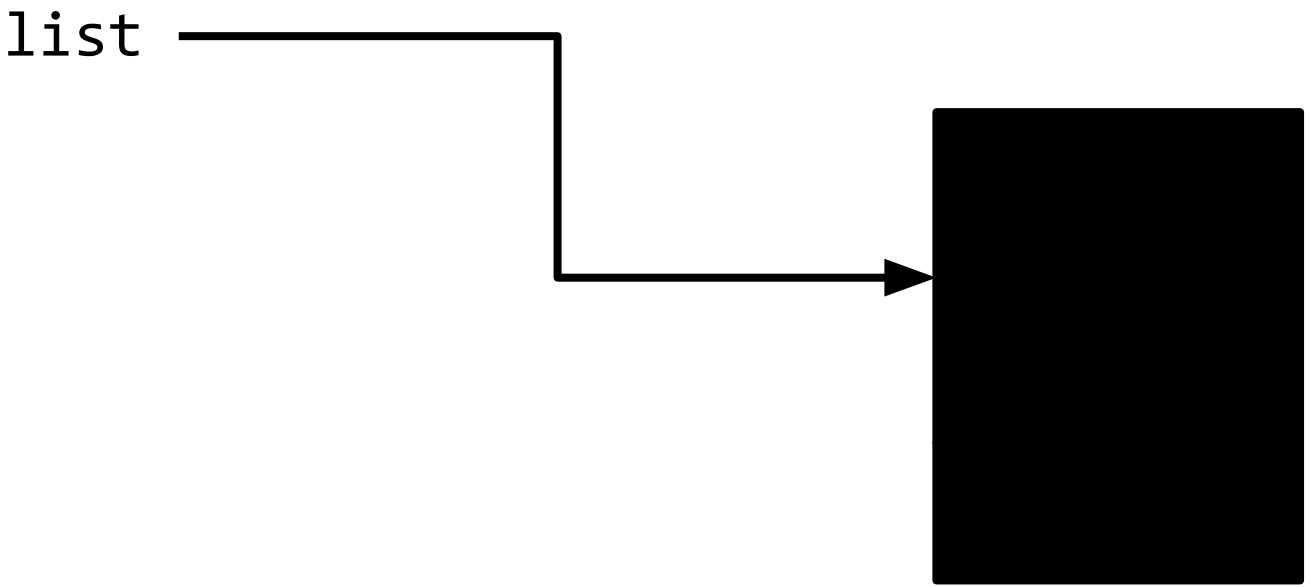
```
ptr = list->next;
```



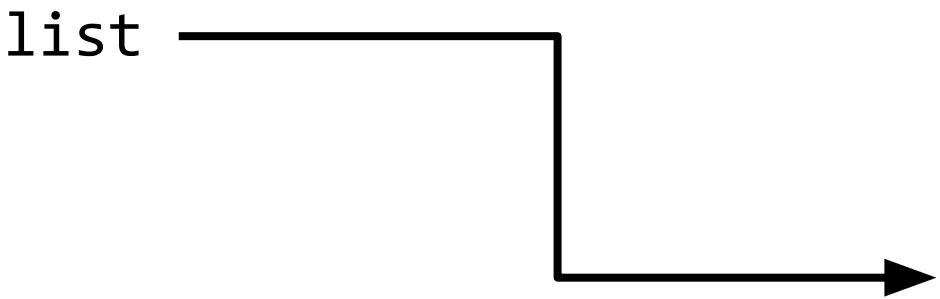
```
ptr = list->next;
```



```
free(list);
```



```
list = ptr;
```



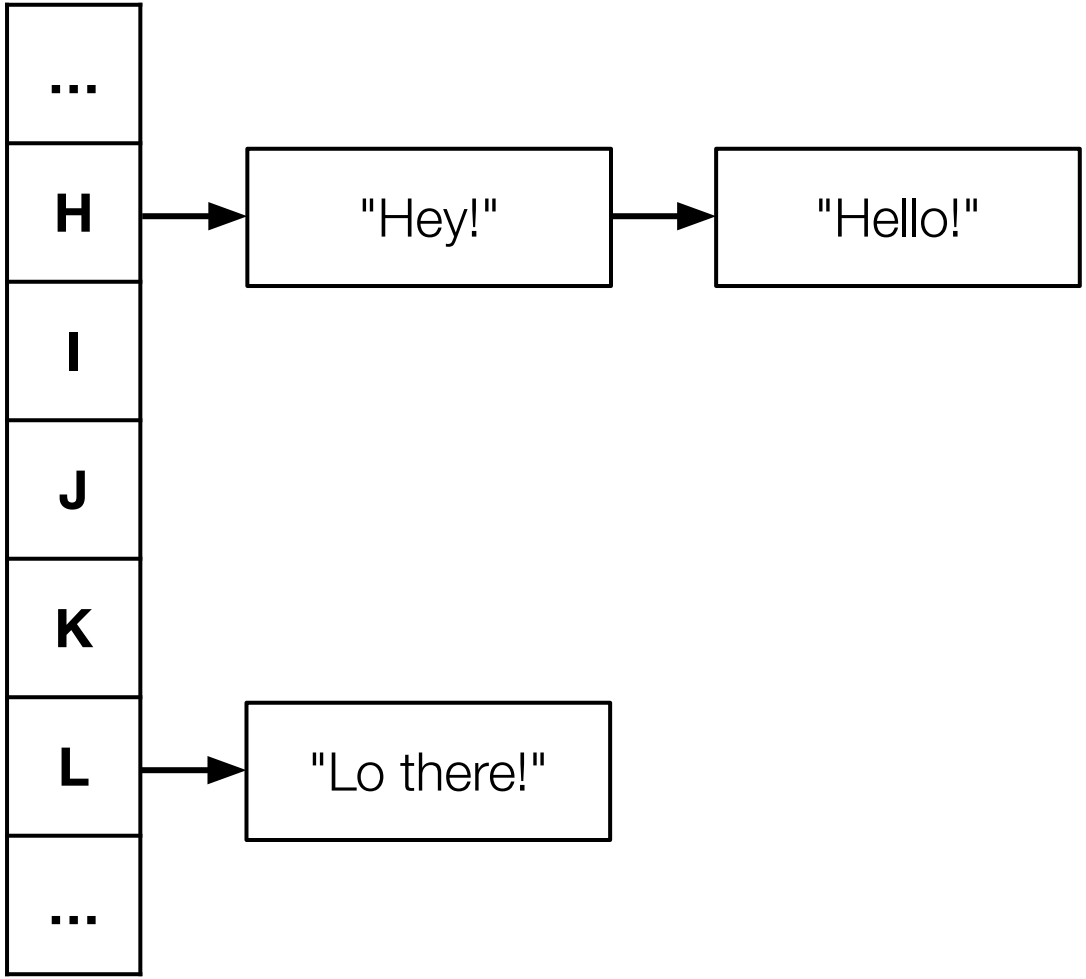
Unloading a Linked List

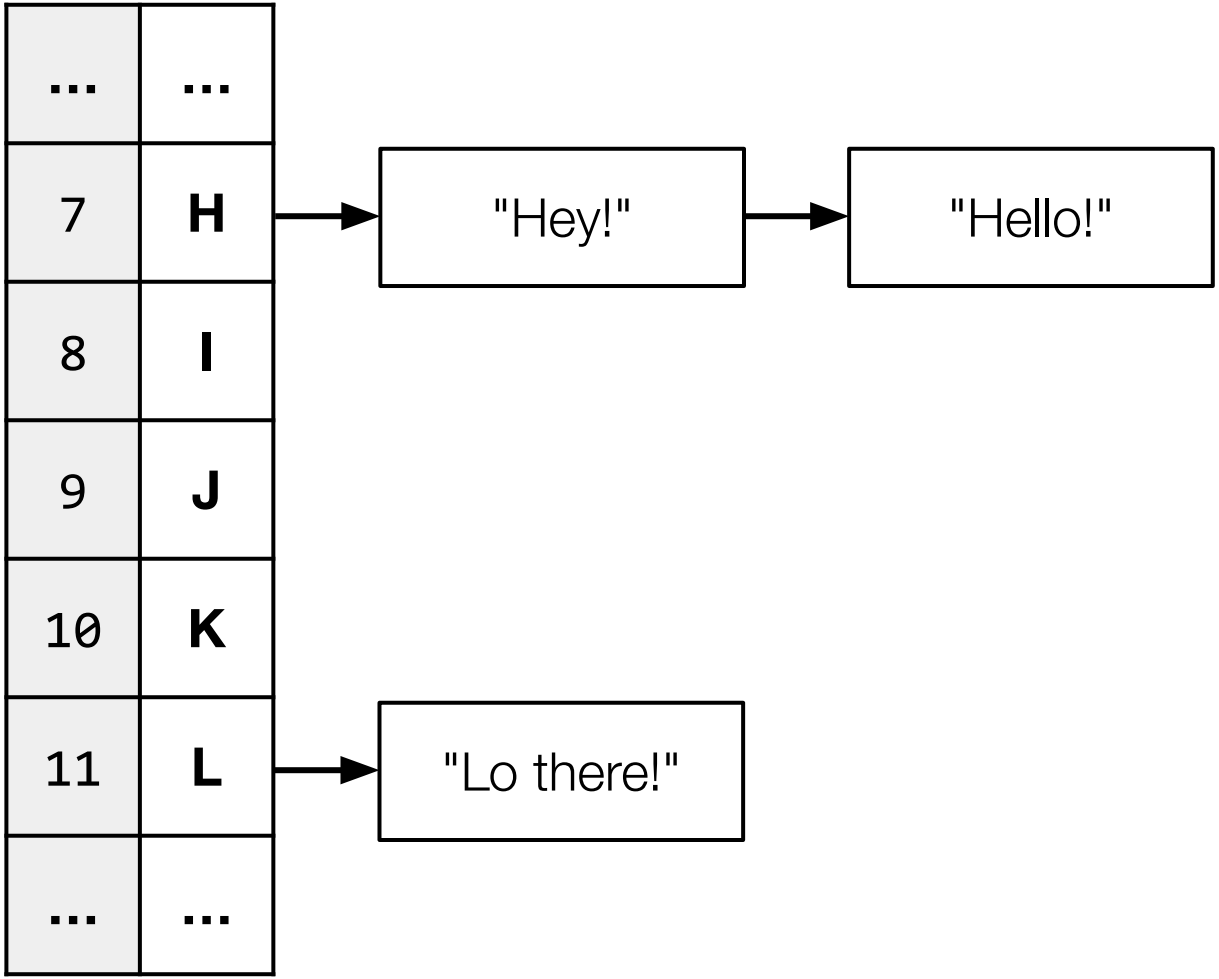
Open the same **list.c** file.

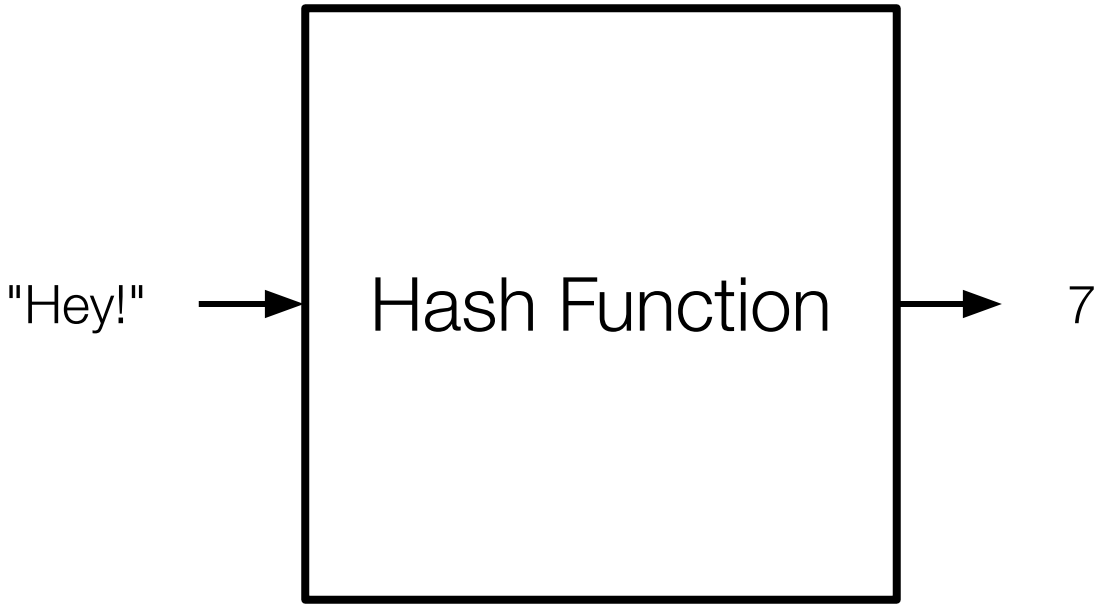
Find the **unload** function below **main**.

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

Hashing

Download and open **table.c**.

Complete **hash** to return a number, 0–25, depending on the first character in the word.

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