This is CS50.
CS Concentration Advising

Tuesday March 10
11:30am-12:30pm
Maxwell Dworkin Ground Floor
cs50.brianyu.me
Week 5

• Data Structures
• Linked Lists
• Trees
• Hash Tables
• Tries
What questions do you have?
Questions

• Why do we use malloc
• Tries
• Manipulate pointers in linked lists
• Coming up with hash functions
Today

Linked Lists
Hash Tables
Trees and Tries
Part One

Linked Lists
Arrays
Arrays

- Fixed size
- Contiguous in memory
Linked Lists

• Any size
• Not contiguous in memory
typedef struct node
{
    int number;
    struct node *next;
}
node;
node *list = malloc(sizeof(node));
list->number = 10;
list->next = NULL;
Linked List

- 0x500
- list

- 0x500
  - number: 10
  - next: 0x600

- 0x600
  - number: 20
  - next: 0x700

- 0x700
  - number: 30
  - next: NULL
Insertion
Insertion

list

number

10

next

number

20

next

number

30

next

number

40

next

number

25

next
Insertion

list

number 10
next

number 20
next

number 30
next

number 40
next

number 25
next
Insertion

list

number

10

next

number

20

next

number

30

next

number

40

next

number

25

next
Insertion

list

number
10
next

number
20
next

number
30
next

number
40
next

number
25
next
Insertion

list

10
next

20
next

30
next

40
next

25
next
Insertion

list

number 10
next

number 20
next

number 30
next

number 40
next

number 25
next
node *blue = malloc(sizeof(node));
red->next = blue;
node *green = malloc(sizeof(node));
red->next = green;
green->next = blue
red->next = green
free(red);
free(green);
free(blue);
Insertion

list

number 10
next

number 20
next

number 30
next

number 40
next
Insertion

list

number
10
next

number
20
next

number
30
next

number
40
next

number
5
next
Insertion

list

10
next

20
next

30
next

40
next

5
next
Insertion

list

number
next

10
next

number
next

20
next

number
next

30
next

number
next

40
next

number
next

5
node *list = NULL;
node *list = NULL;
node *list = NULL;
node *list = NULL;
node *n = malloc(sizeof(node));
node *list = NULL;
node *n = malloc(sizeof(node));
node *list = NULL;
node *n = malloc(sizeof(node));
node *list = NULL;
node *n = malloc(sizeof(node));
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
n->number = 50;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
n->number = 50;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;

list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
```c
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
list = n;
```
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
list = n;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
list = n;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
node *list = NULL;

node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;

list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
n->next = list;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
n->next = list;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
n->next = list;
list = n;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;

list = n;

n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
n->next = list;
list = n;
node *list = NULL;
node *n = malloc(sizeof(node));
n->number = 28;
n->next = NULL;
list = n;
n = malloc(sizeof(node));
n->number = 50;
n->next = NULL;
n->next = list;
list = n;
Exercise

Download distribution code at
wget https://cdn.cs50.net/2020/spring/classes/5/list.c

Update list.c to allocate memory for a new node, for each number entered by the user.

Be sure to set the number of the node, but no need to link the nodes together yet!
Exercise

Update `list.c` to add each new node to the beginning of the linked list.
Exercise

Update `list.c` to print out all of the nodes, on on each line.
Exercise

Update *list.c* to free all of the nodes.
Exercise

Update `list.c` to add new nodes to the end of the linked list, instead of the beginning.
PART TWO

Hash Tables
Hash Tables
apple -> ace -> air

book -> bat

cat

dog -> dark -> desk

eagle
Hash Table

- Array of linked lists
- Use a **hash function** to take an input, and pick a corresponding linked list
function hash(char *s) 
{
    return s[0] - 'A';
}

Hash Function

• Deterministic: always maps same input to the same output
• Minimize collisions: fewer collisions means shorter linked lists
Linked List

node *list;
Hash Table

declare table as a node array

define table as a node array of 50 elements

node *table[50];
PART THREE
Trees and Tries
Trees
Binary Trees
typedef struct node
{
    int number;
    struct node *left;
    struct node *right;
} node;
Binary Search Trees
Tries
typedef struct node
{
    bool word;
    struct node *children[26];
}
node;
https://www.cs.usfca.edu/~galles/visualization/Trie.html
Stacks, Queues
Queues

- Enqueue
- Dequeue
Queues

• First In, First Out
Queues
Queues

10
Queues

10  20
Queues

10   20   30
Queues

10  20  30
Queues

10  20  30  40
Queues

20  30  40
Queues

20  30  40  50
Queues

30  40  50
Queues

40  50
Queues

50
Queues
Stacks

- Push
- Pop
Stacks

• First In, Last Out
Stacks
Stacks
Stacks

10  20
Stacks

10  20  30
Stacks

10  20  30  40
Stacks

10  20  30
Stacks

10  20  30  50
Stacks

10  20  30
Stacks

10  20
Stacks

10
Stacks
Problem Set 5
Problem Set 5

• Speller
<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Time</th>
<th>Load</th>
<th>Check</th>
<th>Size</th>
<th>Unload</th>
<th>Memory</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CS50 Staff Solution</td>
<td>7.445 s</td>
<td>0.825 s</td>
<td>6.165 s</td>
<td>0.000 s</td>
<td>0.455 s</td>
<td>8.0 MB</td>
<td>8.0 MB</td>
<td>2.9 kB</td>
</tr>
</tbody>
</table>

*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`. 
This is CS50.