Quiz 0 Review Session

October 11th, 2015

Topics (non-exhaustive)

- Binary. ASCII. Algorithms. Pseudocode. Source code. Compiler. Object code. Scratch. Statements. Boolean expressions. Conditions. Loops. Variables. Functions. Arrays. Threads. Events.
- Linux. C. Compiling. Libraries. Types. Standard output.
- Casting. Imprecision. Switches. Scope. Strings. Arrays. Cryptography.
- Command-line arguments. Searching. Sorting. Bubble sort. Selection sort. Insertion sort. *O*. Ω.Θ. Recursion. Merge Sort.
- Stack. Debugging. File I/O. Hexadecimal. Strings. Pointers. Dynamic memory allocation.
- Heap. Buffer overflow. Linked lists.
- Hash tables. Tries. Trees. Stacks. Queues.

Official Word

cdn.cs50.net/2015/fall/quizzes/0/harvard.html

Tips for Quiz 0

- practice coding on paper (e.g., `strlen`, `atoi`)
- be familiar with your problem sets!
- do previous quizzes under time constraint
- creating your reference sheet is a great way to study

Data Types and Sizes

- `char` : 1 byte
- `int` : 4 bytes
- `long long` : 8 bytes
- `float` : 4 bytes
- `double` : 8 bytes
- `<type>*` : 8 bytes

Binary

conversion:

binary to decimal

$101010_2 =$

decimal to binary

$50_{10} =$

addition:

$$
\begin{array}{r}
0\ 1\ 0\ 0\ 1 \\
+\ 1\ 0\ 0\ 1\ 1 \\
\hline
\end{array}
$$

Binary

conversion:

binary to decimal

101010_2 = 42

decimal to binary

50_{10} =

addition:

$$
\begin{array}{cccccc}
 & 0 & 1 & 0 & 0 & 1 \\
+ & 1 & 0 & 0 & 1 & 1 \\
\hline
\end{array}
$$

Binary

conversion:

binary to decimal

101010_2 = 42

decimal to binary

50_{10} = 110010

addition:

$$
\begin{array}{cccccc}
 & 0 & 1 & 0 & 0 & 1 \\
+ & 1 & 0 & 0 & 1 & 1 \\
\hline
\end{array}
$$

Binary

conversion:

binary to decimal

$101010_2 = 42$

decimal to binary

$50_{10} = 110010$

addition:

		1	1	
0	1	0	0	1
+ 1	0	0	1	1
1	1	1	0	0

Hexadecimal

conversion:

binary to hexadecimal

$11111111_2 =$

hexadecimal to binary

0x5A =

Hexadecimal

conversion:

binary to hexadecimal

11111111_2 = 0xFF

hexadecimal to binary

0x5A =

Hexadecimal

conversion:

binary to hexadecimal

11111111_2 = 0xFF

hexadecimal to binary

0x5A = 01011010

Bitwise Operators

Allow us to manipulate individual bits

& AND

- gives 1 if *both* arguments are 1

| OR

- gives 1 if *at least 1* argument is 1

^ XOR

- gives 1 if *exactly 1* argument is 1

~ NOT

- flips the given bit

<< left shift

>> right shift

- shifts a bit the given number of places in the given direction

Bitwise Operators

0 & 1 =

1 & 1 =

0 | 1 =

1 | 1 =

0 ^ 1 =

1 ^ 1 =

~0 =

~1 =

```
int x = 8;
int y = x << 3;
y =
```

Bitwise Operators

0 & 1 = 0

1 & 1 = 1

0 | 1 =

1 | 1 =

0 ^ 1 =

1 ^ 1 =

~0 =

~1 =

```
int x = 8;
int y = x << 3;
y =
```

Bitwise Operators

0 & 1 = 0

1 & 1 = 1

0 | 1 = 1

1 | 1 = 1

0 ^ 1 =

1 ^ 1 =

~0 =

~1 =

```
int x = 8;
int y = x << 3;
y =
```

Bitwise Operators

0 & 1 = 0

1 & 1 = 1

0 | 1 = 1

1 | 1 = 1

0 ^ 1 = 1

1 ^ 1 = 0

~0 =

~1 =

```
int x = 8;
int y = x << 3;
y =
```

Bitwise Operators

0 & 1 = 0

1 & 1 = 1

0 | 1 = 1

1 | 1 = 1

0 ^ 1 = 1

1 ^ 1 = 0

~0 = 1

~1 = 0

```
int x = 8;
int y = x << 3;
y =
```

Bitwise Operators

0 & 1 = 0

1 & 1 = 1

0 | 1 = 1

1 | 1 = 1

0 ^ 1 = 1

1 ^ 1 = 0

~0 = 1

~1 = 0

```
int x = 8;
int y = x << 3;
y = 64
```

ASCII - Math

Because characters are fundamentally just numbers, we can do math with chars!

```
int A = 65;
int B = 'A' + 1;
char C = 'D' - 1;
char D = 68;
printf("%c %c %c %c", A, B, C, D);
```

What will this print out?

ASCII - Math

Because characters are fundamentally just numbers, we can do math with chars!

```
int A = 65;
int B = 'A' + 1;
char C = 'D' - 1;
char D = 68;
printf("%c %c %c %c", A, B, C, D);
```

What will this print out? A B C D

Scope

Determines the region where a variable exists. Within this area, we can access or change the variable

- Global
  - Entire program has access to it
  - Exist for the duration of the program
- Local
  - Confined to a region
  - Examples: Within specific functions, if statements, for loops

Prototypes

When we define a function after we plan to use it, we must include a prototype!

```
<return type> function_name(arguments);
```

```c
#include <stdio.h>

int cube(int input);

int main(void)
{
    int x = 2;
    printf("x is %d\n", x);
    x = cube(x);
    printf("x is %d\n", x);
}

int cube(int input)
{
    return input * input * input;
}
```

← prototype

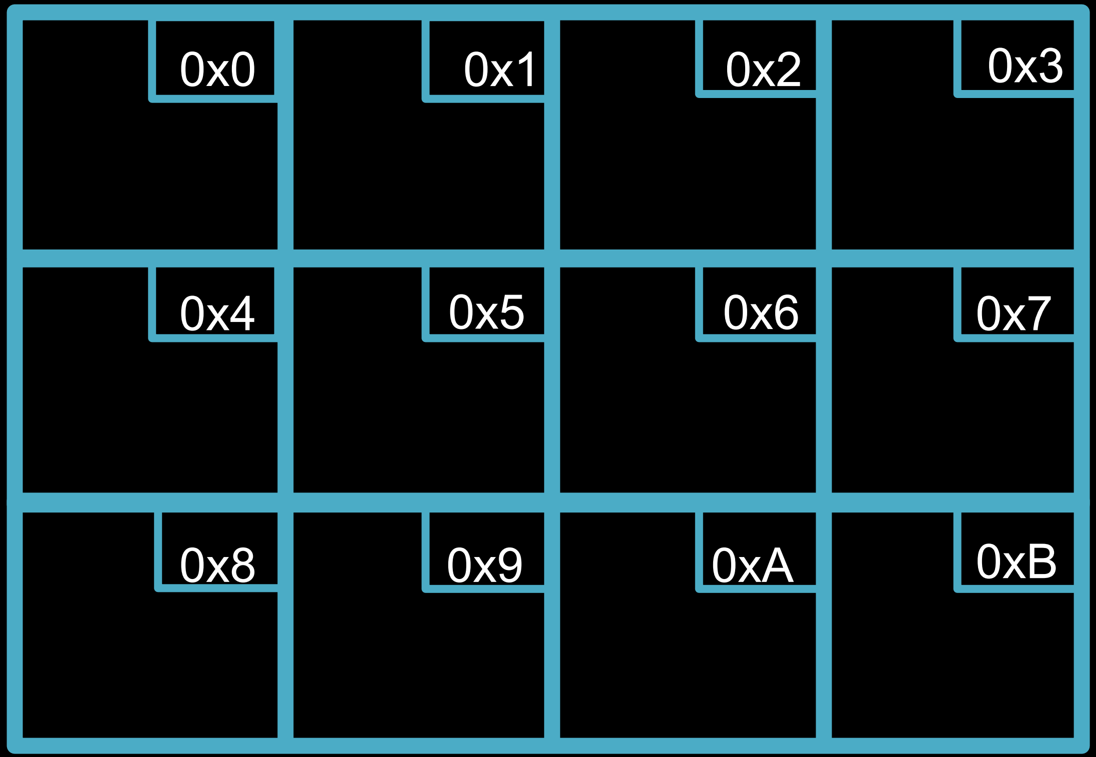
Floating-Point Imprecision

infinitely many real numbers (even between 0 and 1!) but **finitely** many bits to represent real numbers

⇒ imprecision

Pointers

Memory



Creating Pointers

<type>* <variable name>

Examples:

```
int* x;
char* y;
float* z;
```

Referencing and Dereferencing

Referencing (i.e., address of):
&<variable name>

Dereferencing:
*<pointer name>

Under the hood...

```
int x = 5;

int* ptr = &x;

int copy = *ptr;
```

Variable	Address	Value
x	0x04	5
ptr		
copy		

Under the hood...

```
int x = 5;

int* ptr = &x;

int copy = *ptr;
```

Variable	Address	Value
x	0x04	5
ptr	0x08	0x04
copy		

Under the hood...

```
int x = 5;

int* ptr = &x;

int copy = *ptr;
```

Variable	Address	Value
x	0x04	5
ptr	0x08	0x04
copy	0x10	5

Buggy

```
#include <stdio.h>

void to_five(int a)
{
  3:
    a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(x);
  5:
    printf("%d\n", x);
}
```

	x	a
1:		
2:		
3:		
4:		
5:		

Buggy

```
#include <stdio.h>

void to_five(int a)
{
  3:
    a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(x);
  5:
    printf("%d\n", x);
}
```

	x	a
1:	N/A	N/A
2:		
3:		
4:		
5:		

Buggy

```c
#include <stdio.h>

void to_five(int a)
{
  3:
    a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(x);
  5:
    printf("%d\n", x);
}
```

	x	a
1:	N/A	N/A
2:	3	N/A
3:		
4:		
5:		

Buggy

```
#include <stdio.h>

void to_five(int a)
{
  3:
    a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(x);
  5:
    printf("%d\n", x);
}
```

	x	a
1:	N/A	N/A
2:	3	N/A
3:	3	3
4:		
5:		

Buggy

```
#include <stdio.h>

void to_five(int a)
{
  3:
    a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(x);
  5:
    printf("%d\n", x);
}
```

	x	a
1:	N/A	N/A
2:	3	N/A
3:	3	3
4:	3	5
5:		

Buggy

```c
#include <stdio.h>

void to_five(int a)
{
  3:
    a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(x);
  5:
    printf("%d\n", x);
}
```

	x	a
1:	N/A	N/A
2:	3	N/A
3:	3	3
4:	3	5
5:	3	N/A

Fixed

Assume &x == 0x12

```
#include <stdio.h>

void to_five(int* a)
{
  3:
    *a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(&x);
  5:
    printf("%d\n", x);
}
```

	x	a	*a
1:			
2:			
3:			
4:			
5:			

Fixed

Assume &x == 0x12

```
#include <stdio.h>

void to_five(int* a)
{
  3:
    *a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(&x);
  5:
    printf("%d\n", x);
}
```

	x	a	*a
1:	N/A	N/A	N/A
2:			
3:			
4:			
5:			

Fixed

Assume &x == 0x12

```
#include <stdio.h>

void to_five(int* a)
{
  3:
    *a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(&x);
  5:
    printf("%d\n", x);
}
```

	x	a	*a
1:	N/A	N/A	N/A
2:	3	N/A	N/A
3:			
4:			
5:			

Fixed

Assume &x == 0x12

```
#include <stdio.h>

void to_five(int* a)
{
  3:
    *a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(&x);
  5:
    printf("%d\n", x);
}
```

	x	a	*a
1:	N/A	N/A	N/A
2:	3	N/A	N/A
3:	3	0x12	3
4:			
5:			

Fixed

Assume &x == 0x12

```
#include <stdio.h>

void to_five(int* a)
{
  3:
    *a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(&x);
  5:
    printf("%d\n", x);
}
```

	x	a	*a
1:	N/A	N/A	N/A
2:	3	N/A	N/A
3:	3	0x12	3
4:	5	0x12	5
5:			

Fixed

Assume &x == 0x12

```
#include <stdio.h>

void to_five(int* a)
{
  3:
    *a = 5;
  4:
}

int main(void)
{
  1:
    int x = 3;
  2:
    to_five(&x);
  5:
    printf("%d\n", x);
}
```

	x	a	*a
1:	N/A	N/A	N/A
2:	3	N/A	N/A
3:	3	0x12	3
4:	5	0x12	5
5:	5	N/A	N/A

Pointer Arithmetic

Adding/subtracting i adjusts the pointer by i * sizeof(<type of the pointer>) bytes

Assume &x == 0x04	x	y
int x = 5;	5	
int* y = &x;		
y += 1;		

Pointer Arithmetic

Adding/subtracting `i` adjusts the pointer by `i * sizeof(<type of the pointer>)` bytes

Assume &x == 0x04	x	y
int x = 5;	5	
int* y = &x;	5	0x04
y += 1;		

Pointer Arithmetic

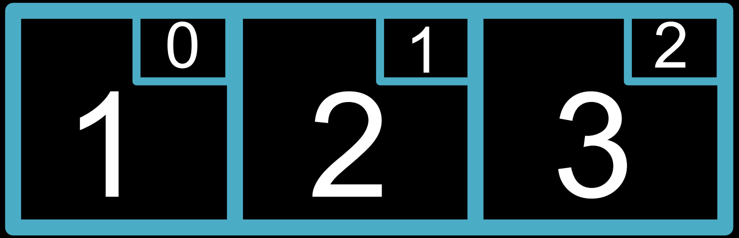
Adding/subtracting `i` adjusts the pointer by `i * sizeof(<type of the pointer>)` bytes

Assume &x == 0x04

	x	y
int x = 5;	5	
int* y = &x;	5	0x04
y += 1;	5	0x08

Pointers and Arrays

```
int array[3];

*array = 1;
*(array + 1) = 2;
*(array + 2) = 3;
```

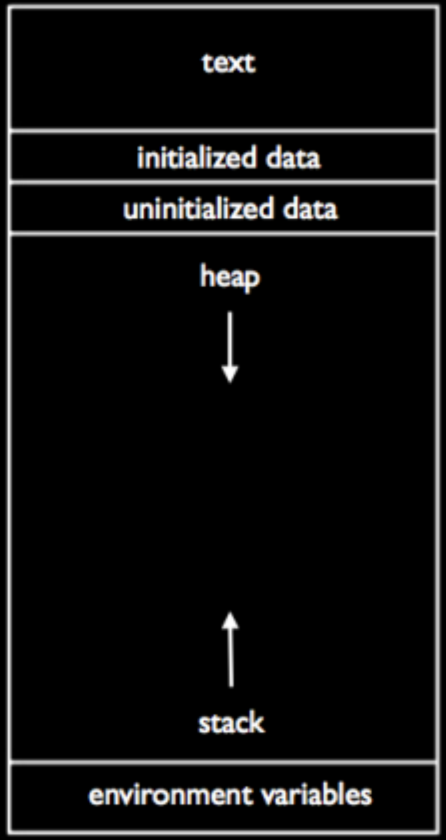


Memory

- **stack**: block of memory set aside when a program starts running
  - each function gets its own stack frame
  - **stack overflow**: when the stack runs out of space, results in a program crash

- **heap**: region of unused memory that can be dynamically allocated using malloc (and realloc, etc.)

- don’t forget to **free** dynamically allocated memory to prevent **memory leaks**

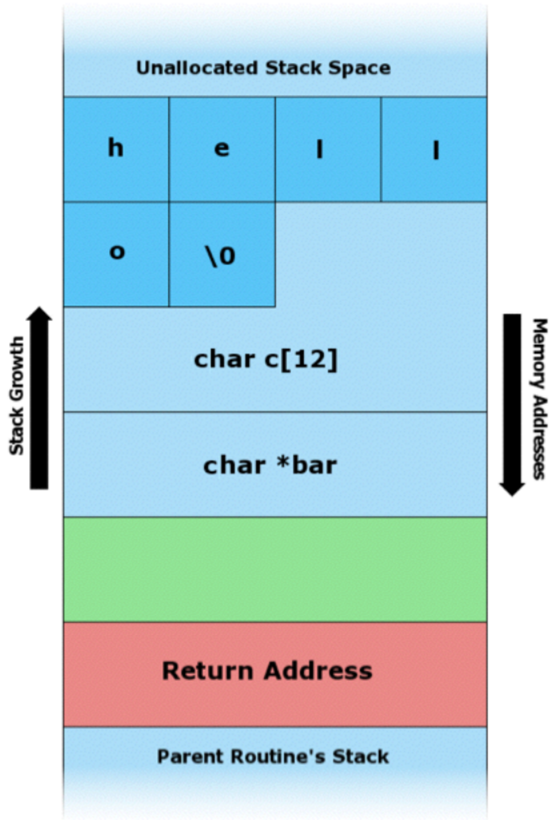


Allocating Memory

```
void* malloc(<size in bytes>);

    int* ptr = malloc(sizeof(int) * 10);
    …
    free(ptr);
```

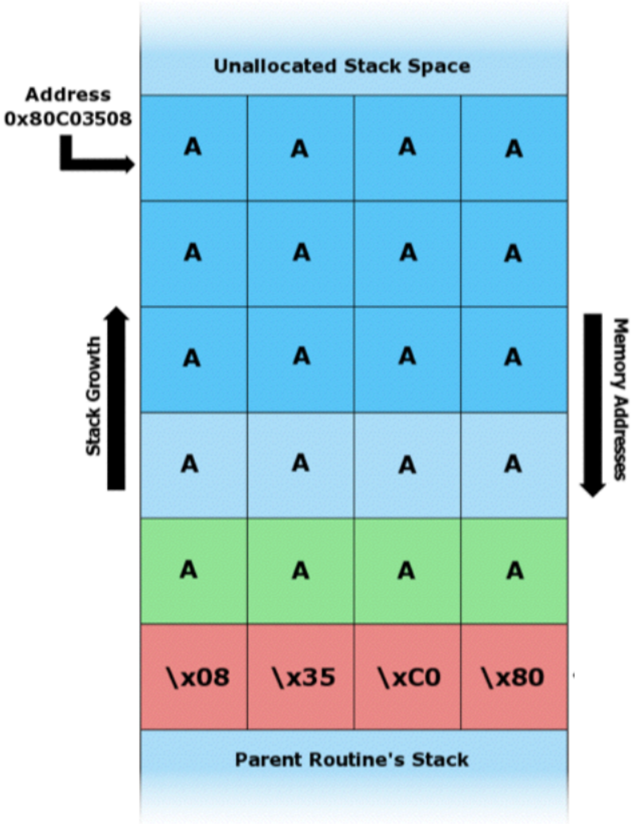
Don't forget to check for NULL!

Buffer Overflow



adapted from http://en.wikipedia.org/wiki/Stack_buffer_overflow

Buffer Overflow



adapted from http://en.wikipedia.org/wiki/Stack_buffer_overflow

Common Error Messages

- segmentation fault
  - when a program attempts to access memory that it is not allowed to access
  - check for `NULL`!
- implicit declaration of function
  - when a program is defined after the `main` function, and no **prototype** is present above
  - when a program is missing a necessary `#include`
- undeclared identifier
  - when a variable has not been declared

Recursion

- a programming concept whereby a function calls itself
- don’t forget to include a base case!
- pros:
  - can lead to more concise, elegant code
  - some algorithms lend themselves to recursion
    - e.g., merge sort

Search and Sort Run Times

	linear search	binary search	bubble sort	selection sort	insertion sort	merge sort
O	n	$\log(n)$	n^2	n^2	n^2	$n \log(n)$
Ω	1	1	n	n^2	n	$n \log(n)$
Θ				n^2		$n \log(n)$

O **upper bound (in the worst case)**

Ω **lower bound (in the best case)**

Θ **identical upper and lower bound**

Structs

Allow us to create our own data type or container to hold data of different types

```
typedef struct
{
    int id;
    string name;
}
student;
```

Creating and Accessing Structs

- Declare using the struct name as the variable type
- Access using the . operator

```
int main(void)
{
    student student_1;
    student_1.id = 1;
    student_1.name = "Daven";
}
```

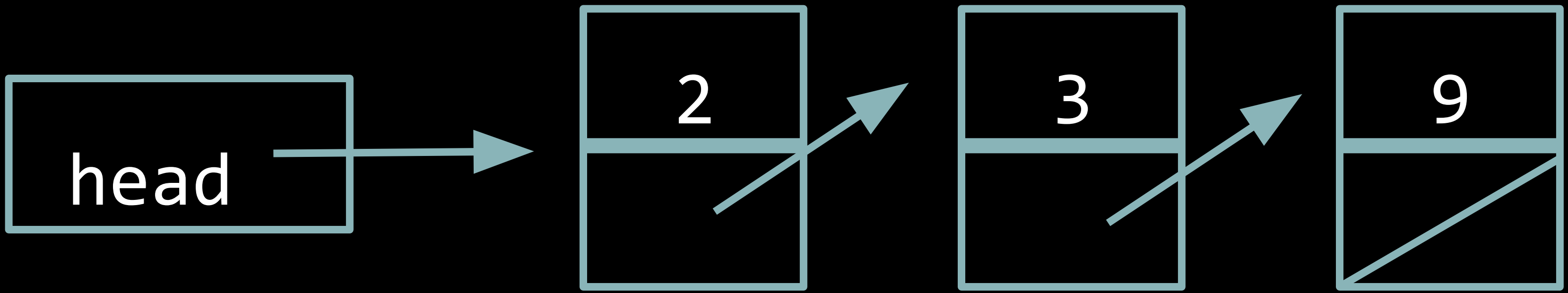
Creating and Accessing Structs

- If we have a pointer to a struct we can use -> notation

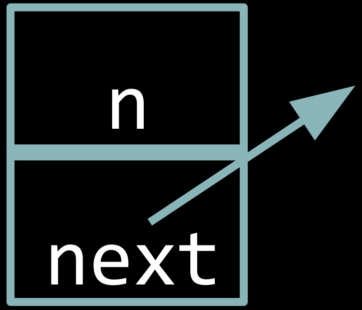
```c
int main(void)
{
    student student_1;
    student* ptr = &student_1;

    ptr->name = "Rob";
    (*ptr).name = "Rob";
}
```

} equivalent

Linked Lists

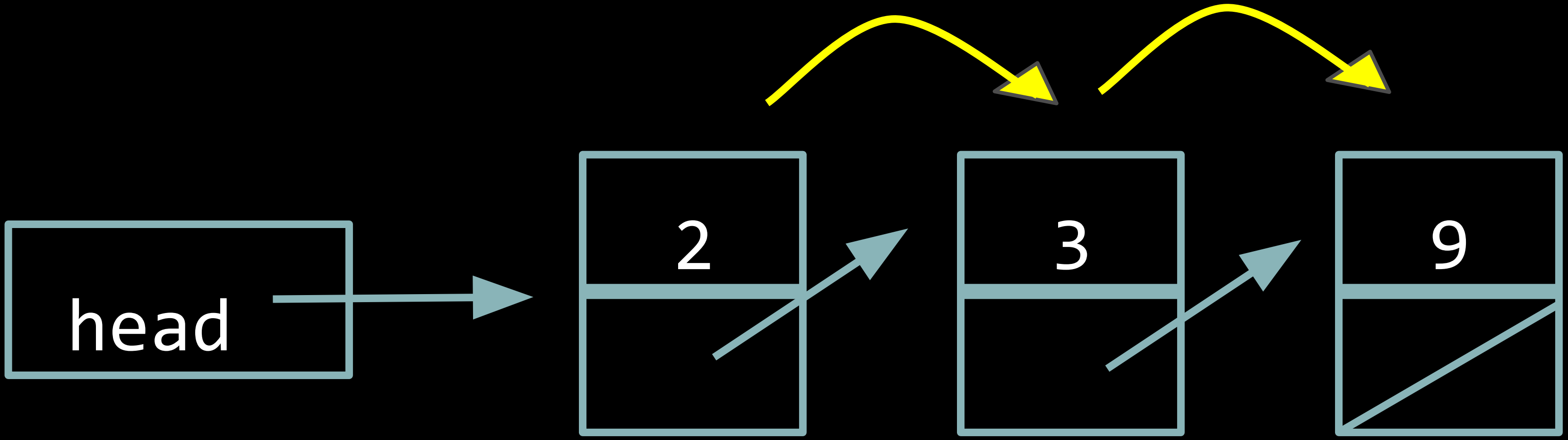


Nodes



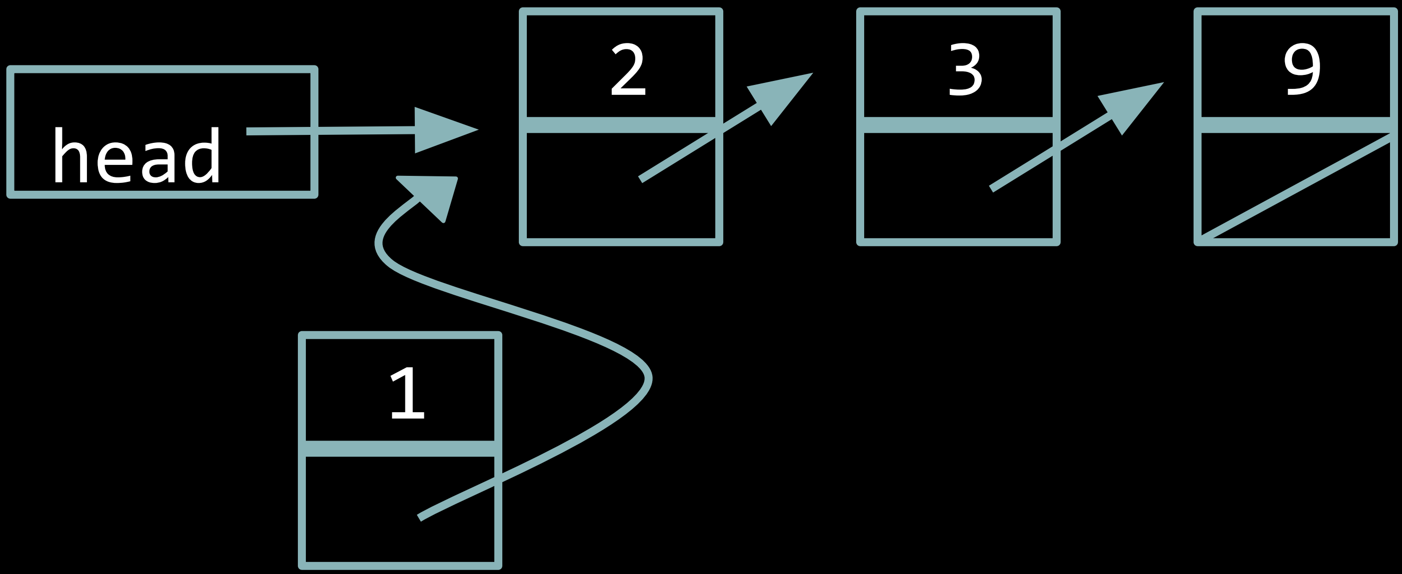

```
typedef struct node
{
    int n;
    struct node* next;
}
node;
```

Search

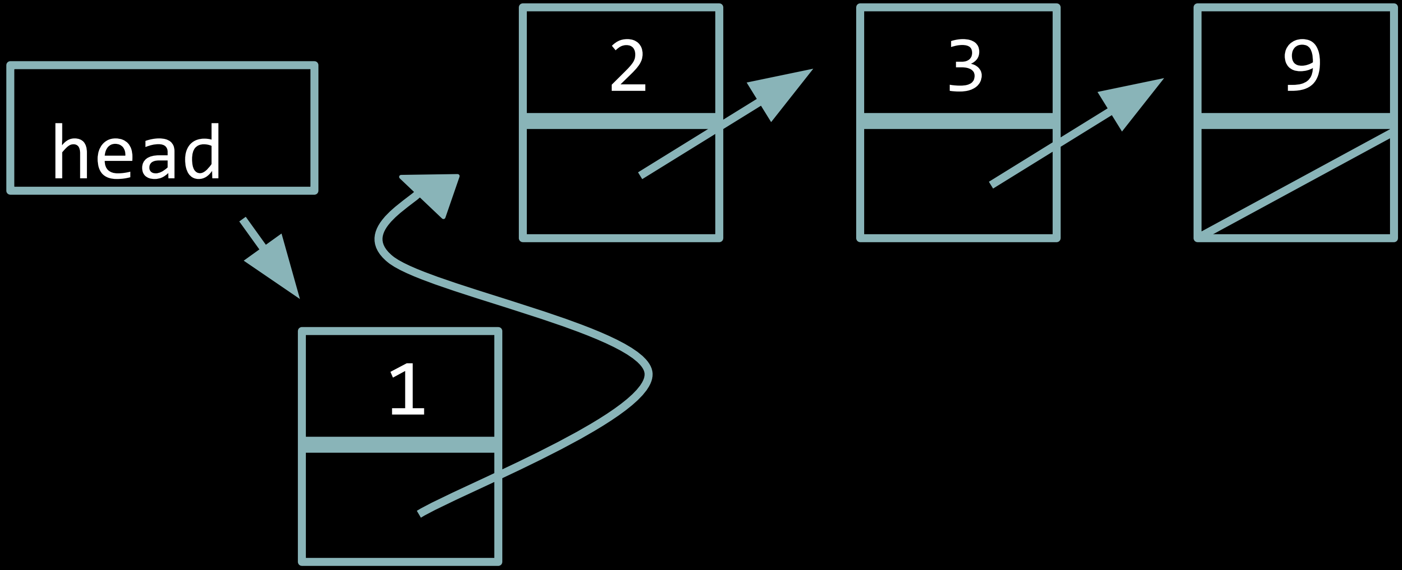


```
bool search(int n, node* list)
{
    // points at current node
    node* ptr = list;

    // traverse the list until the end
    while (ptr != NULL)
    {
        // check if we found value
        if (ptr->n == n)
        {
            return true;
        }

        // move on to next element
        ptr = ptr->next;
    }
    return false;
}
```

Insertion



Insertion



```c
bool insert(int n)
{
    // create new node
    node* new = malloc(sizeof(node));

    // check for NULL
    if (new == NULL)
    {
        return false;
    }

    // initialize new node
    new->n = n;
    new->next = NULL;

    // insert new node at head
    new->next = head;
    head = new;

    return true;
}
```

Stacks

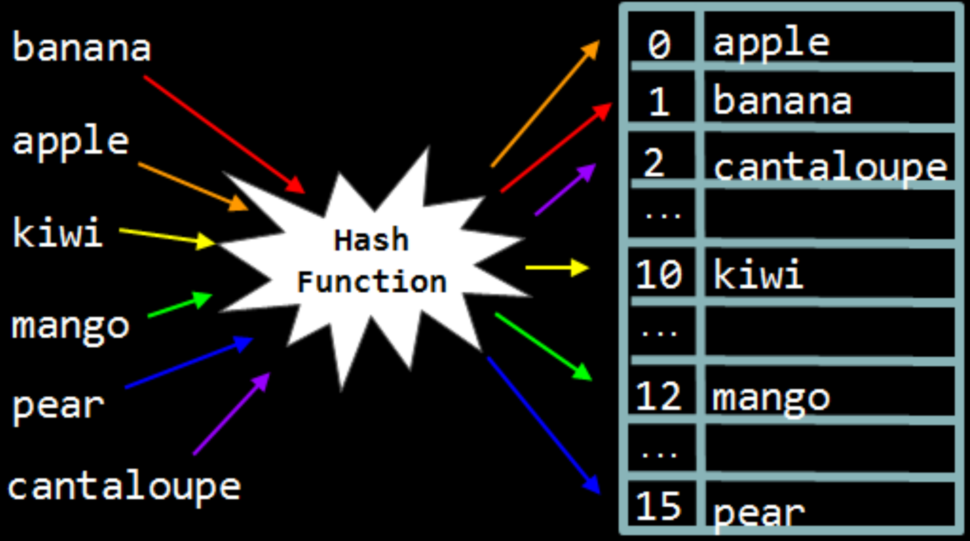
- first-in, last-out (FILO)
- elements are successively pushed down as other items are added
- elements are **pushed** on and **popped** off
- keep track of both the **size** and **capacity**
    - you need not keep track of capacity if you use a linked list rather than an array

Queues

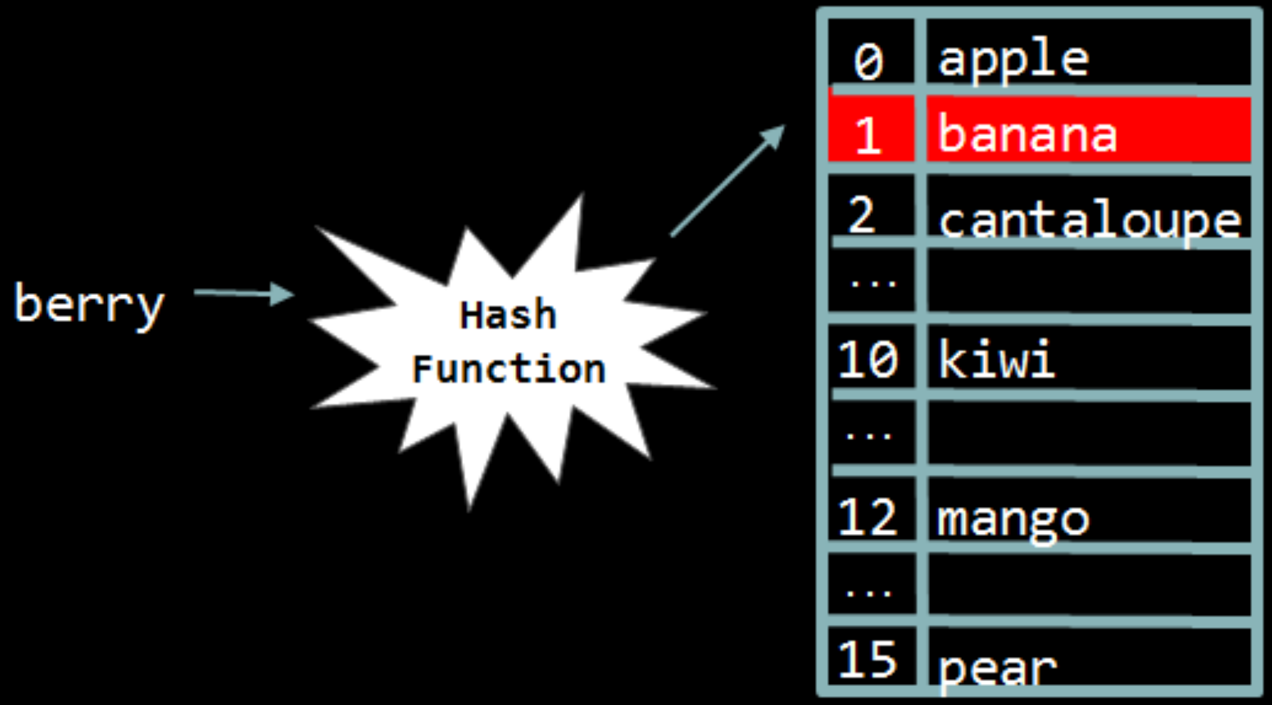
- first-in, first-out (FIFO)
- picture a line!
- elements are **enqueued** and **dequeued**
- keep track of the **size**, **capacity**, and **head**
  - you need not keep track of capacity if you use a linked list rather than an array

Hash Table

- data structure where the position of each element is decided by a hash function



Collisions


berry
Hash Function
0
apple
1
banana
2
cantaloupe
...
10
kiwi
...
12
mango
...
15
pear

Linear Probing

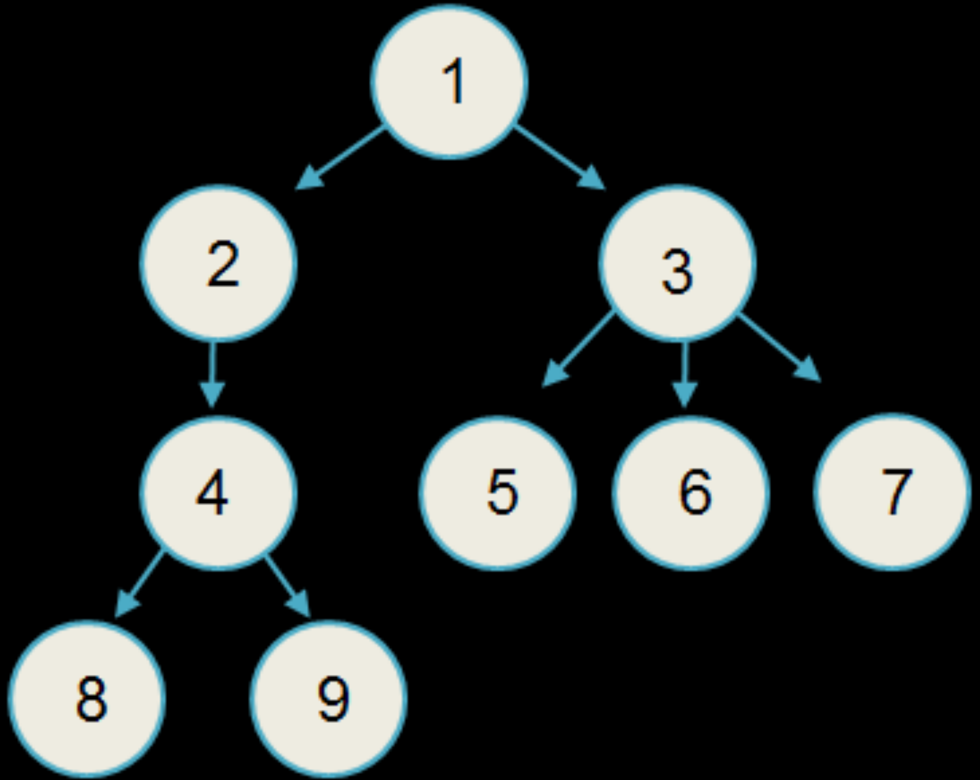
Separate Chaining

Trees and Tries

- **tree**: a data structure in which data is organized hierarchically
  - e.g., binary search tree

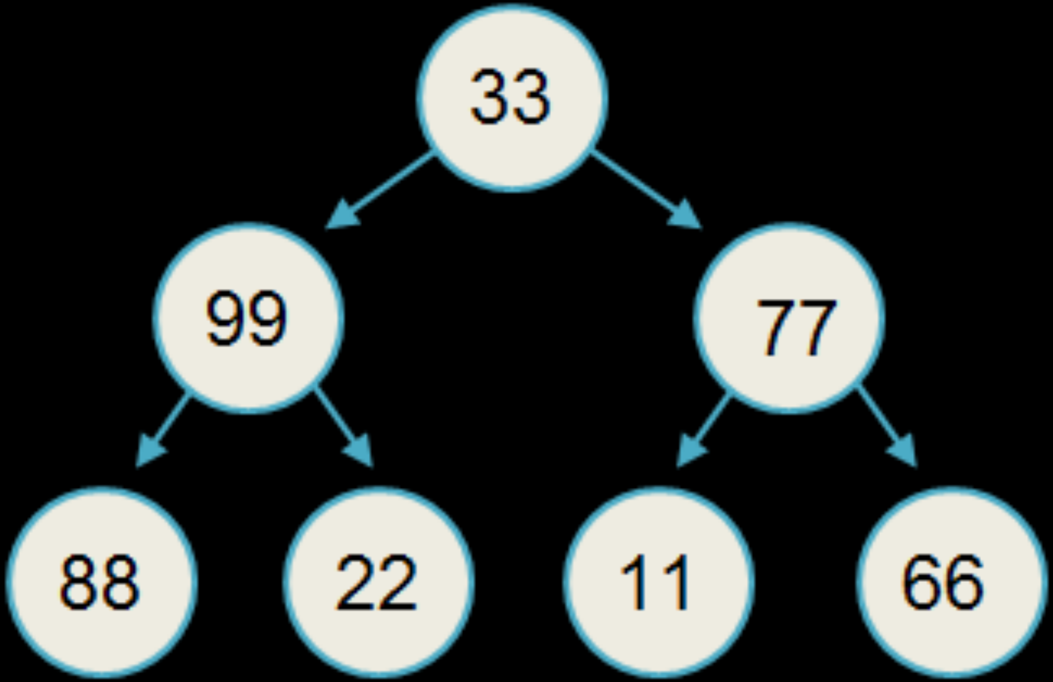
- **trie**: special kind of tree that behaves like a multi-level hash table

Trees



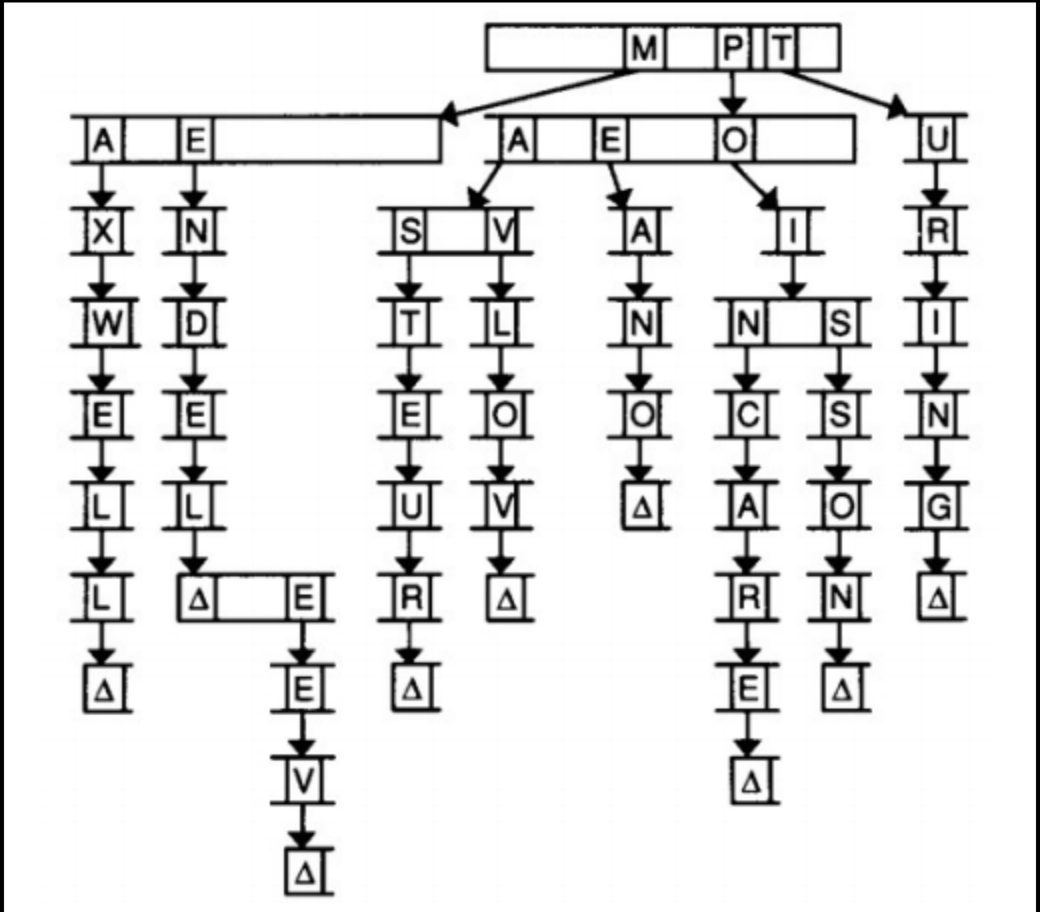
Binary Trees

(Note: not a binary search tree!)



https://study.cs50.net

Tries



https://study.cs50.net

Tries

- *pro:* provide constant time lookup (in theory)
- *con:* use large amounts of memory!

Questions?

And finally...

RELAX AND SLEEP!

(you'll do great! =D)