This is CS50.

Lab 5

Concept Deep Dive

Week 5 Concepts:

- Linked List
- Hash Tables
- Queue, Stack

Which one is the most confusing?

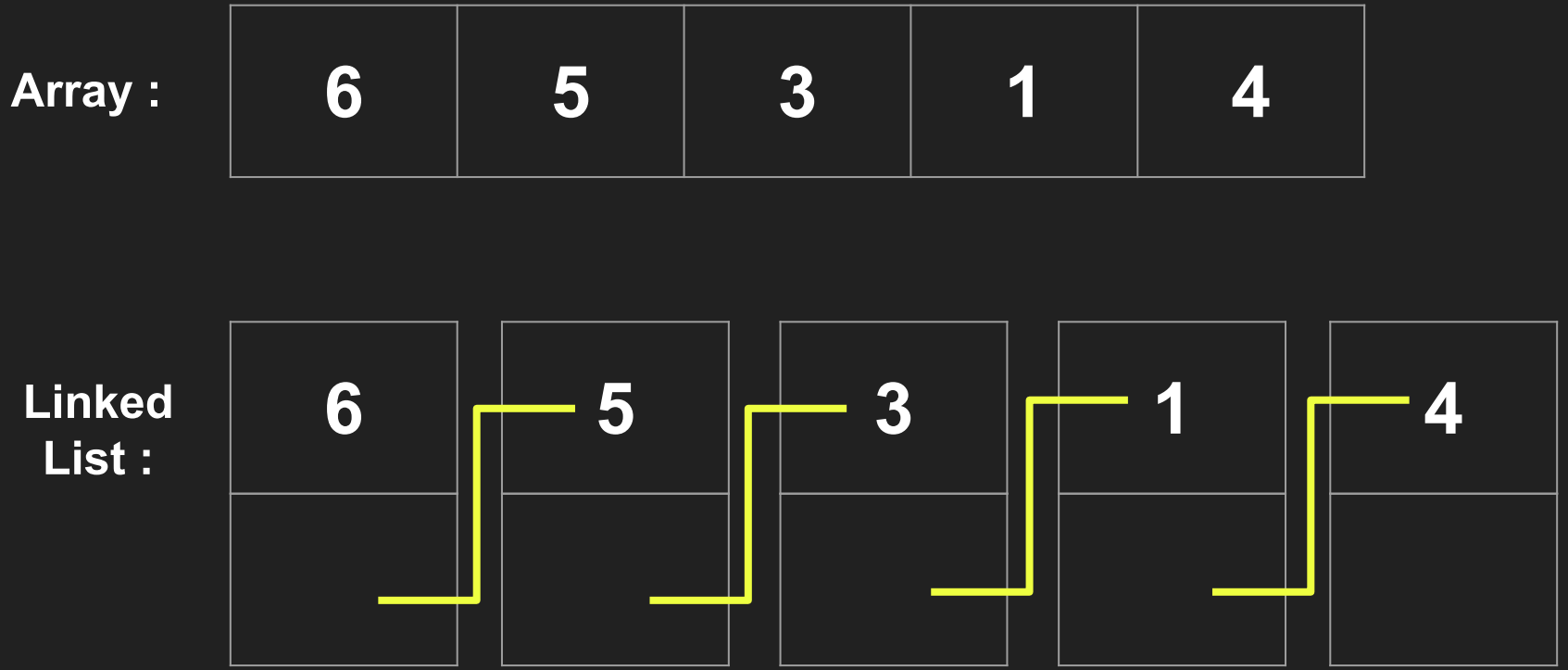
Recall important functions:

- `struct` to …
- `.` to …
- `*` to …
- `→` to …

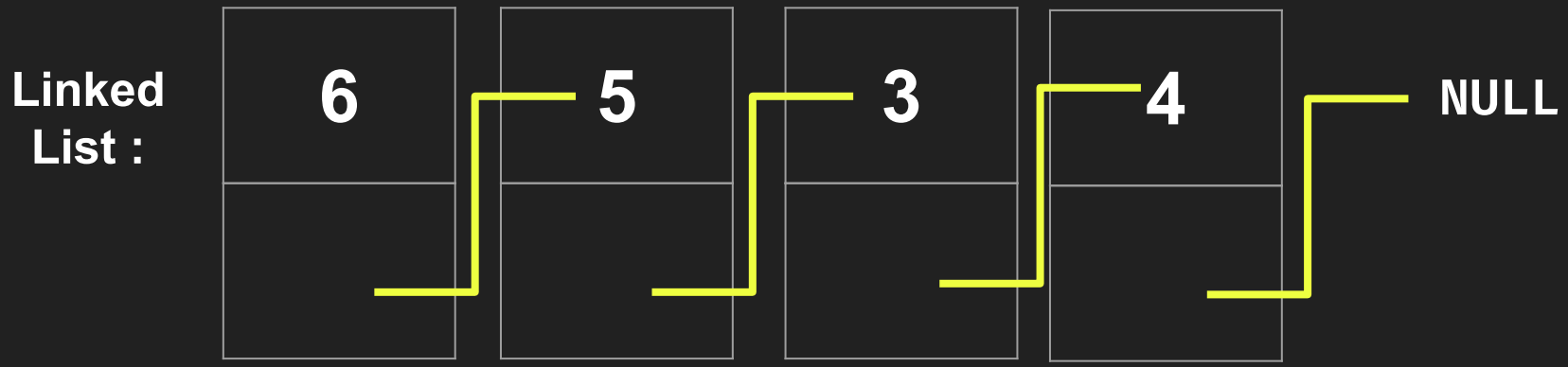
Recall important functions:

- `struct` to create custom data types
- `.` to access fields, or values, in a structure
- `*` to go to an address in memory pointed to by a pointer
- `→` to access fields in a structure pointed to by a pointer

Why Linked Lists?



What is a Linked Lists?



A linked list is a data structure in C that allows us to create a chain of nodes that is dynamically-sized.

How might we “create” a node In Linked Lists?

Chain of Nodes

Hence, the fundamental basis of any linked list is the `node` structure:

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

Chain of Nodes

Hence, the fundamental basis of any linked list is the `node` structure:

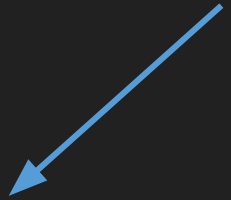
```
typedef struct node
{
    int value;
    struct node *next;
}
node;
```

This can be any value you want. You could have a linked list of strings, integers, floats, etc. Just adjust the type as needed

Chain of Nodes

Hence, the fundamental basis of any linked list is the `node` structure:

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



This stores a pointer (the address) to the next element in the linked list.

Let’s do an example : Creating a new linked list!

Steps:

1. Define a custom data type, typedef struct node, as we did.
2. Dynamically allocate space for a new node
3. Initialize the value field
4. Initialize the next field (Specifically to `NULL`!)

Let’s do an example : Creating a new linked list!

Steps:

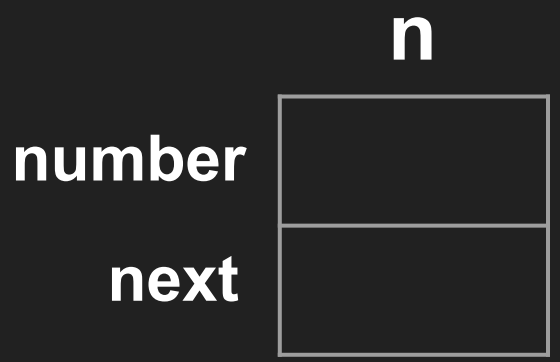
1. Define a custom data type, typedef struct node, as we did.
2. Dynamically allocate space for a new node
3. Initialize the value field
4. Initialize the next field (Specifically to `NULL`!)

Creating a new linked list!

We need to:

2. Dynamically allocate space for a new node.

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

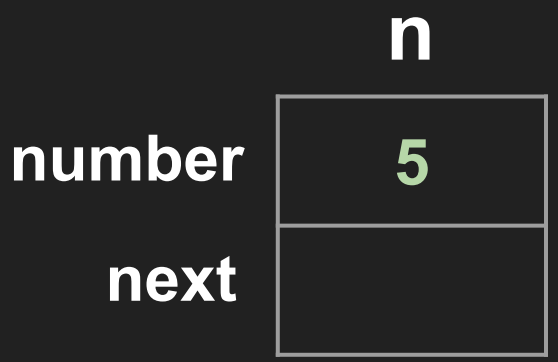


Creating a new linked list!

We need to:

3. Initialize the value field.

```
node *n = malloc(sizeof(node));
n->number = 5;
```



Creating a new linked list!

We need to:

4. Initialize the next field (specifically, to NULL).

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

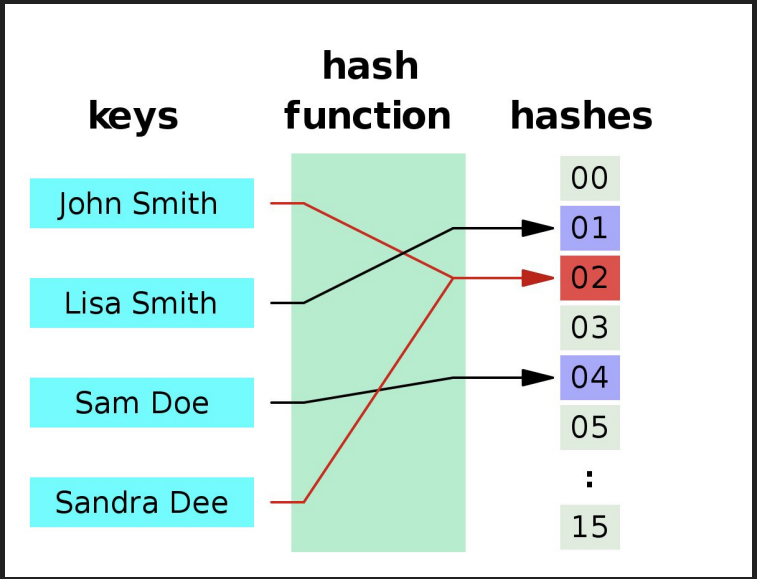
Any questions for me?

HASH TABLES

What is a **hash function**?

“A hash function is any function that can be used to map data of arbitrary size to data of a fixed size.”

But why and how?



HASH TABLES

A really simple example of this:

```c
#include <string.h>

int hash(char *word);

int main(void) {
   // Creating an array for the hash table
   char *hashTable[26];

   // Adding an element to the hash table
   char *firstWord = "Hello";
   strcpy(hashTable[hash(firstWord)], firstWord);
}

// Hash Function
int hash(char *word) {
   return  (int) word[0] - 'A';
}
```

HASH TABLES

A really simple example of this:

```c
#include <string.h>

int hash(char *word);

int main(void) {
   // Creating an array for the hash table
   char *hashTable[26];

   // Adding an element to the hash table
   char *firstWord = "Hello";
   strcpy(hashTable[hash(firstWord)], firstWord);
}

// Hash Function
int hash(char *word) {
   return  (int) word[0] - 'A';
}
```

The hash function literally just returns the “alphabetic index” for whatever the first character of the word you pass it (only works for capital letters)

HASH TABLES

Why might this be a bad hash table?

```c
#include <string.h>

int hash(char *word);

int main(void) {
   // Creating an array for the hash table
   char *hashTable[26];

   // Adding an element to the hash table
   char *firstWord = "Hello";
   strcpy(hashTable[hash(firstWord)], firstWord);
}

// Hash Function
int hash(char *word) {
   return  (int) word[0] - 'A';
}
```

HASH TABLES

Why might this be a bad hash table?

```c
#include <string.h>

int hash(char *word);

int main(void) {
   // Creating an array for the hash table
   char *hashTable[26];

   // Adding an element to the hash table
   char *firstWord = "Hello";
   strcpy(hashTable[hash(firstWord)], firstWord);
}

// Hash Function
int hash(char *word) {
   return  (int) word[0] - 'A';
}
```

The array it uses is small! Words will quickly fill up every slot and the slots are uneven (‘Z’ will be referenced a lot less than ‘T’).

HASH TABLES

What makes for a good hash function?

HASH TABLES

What makes for a good hash function?

- **Use only the data being hashed.**
- **Use all of the data being hashed.**
- **Be deterministic (same result every time given same input; no randomness!).**
- **Uniformly distribute data.**
- **Generate very different hash codes for very similar data.**

There’s a whole lots of research being done about good hash functions

Any questions for me?

Stacks and Queue

Remember David’s Closet?

That’s a stack!

Stack: LIFO

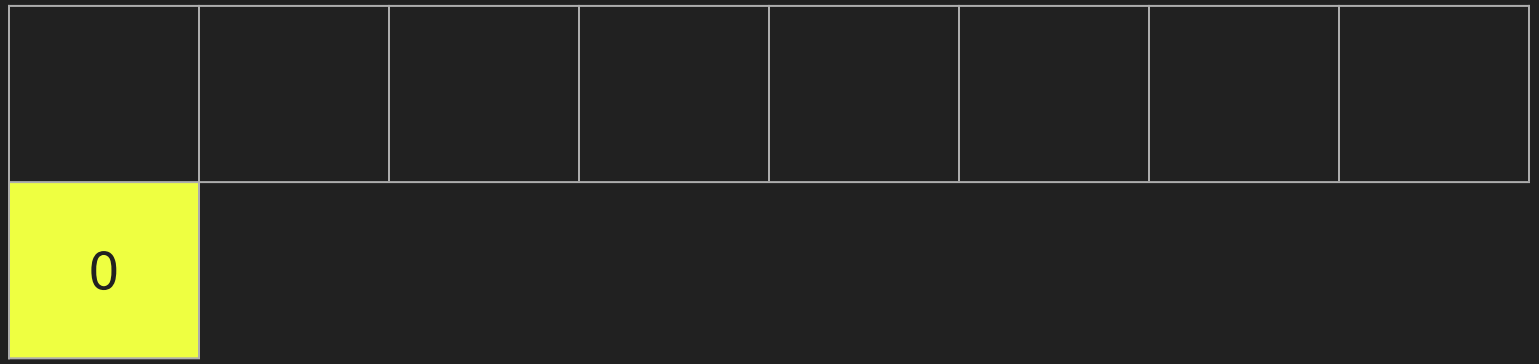
- Stack implemented as an array:

```
typedef struct stack
{
    int array[CAPACITY];
    int top;
}
stack;
```

Stack: push

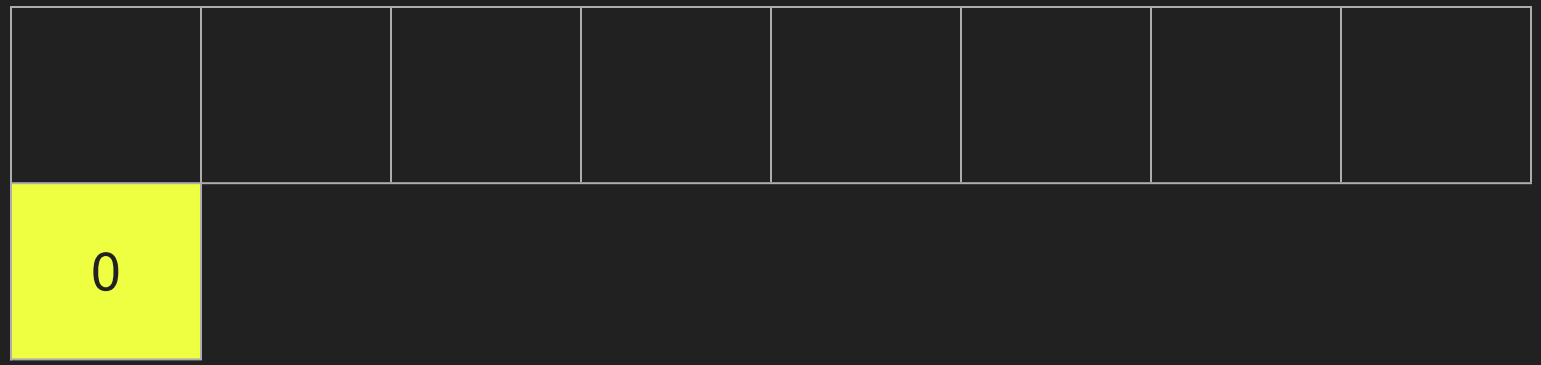
- Stack implemented as an array:

```
stack s;
s.top = 0;
```



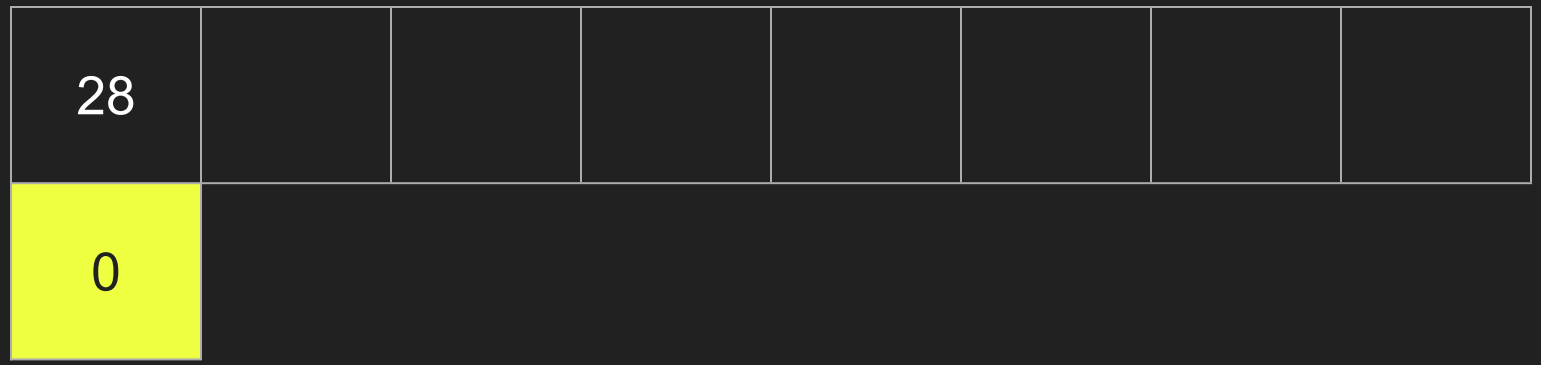
- Stack implemented as an array:

```
push(&s, 28);
```



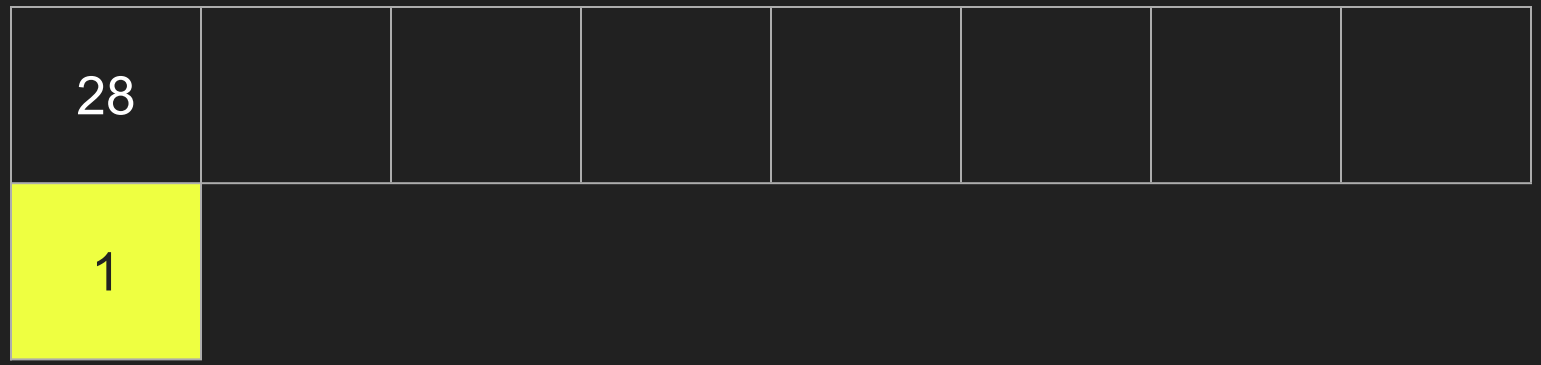
- Stack implemented as an array:

```
push(&s, 28);
```



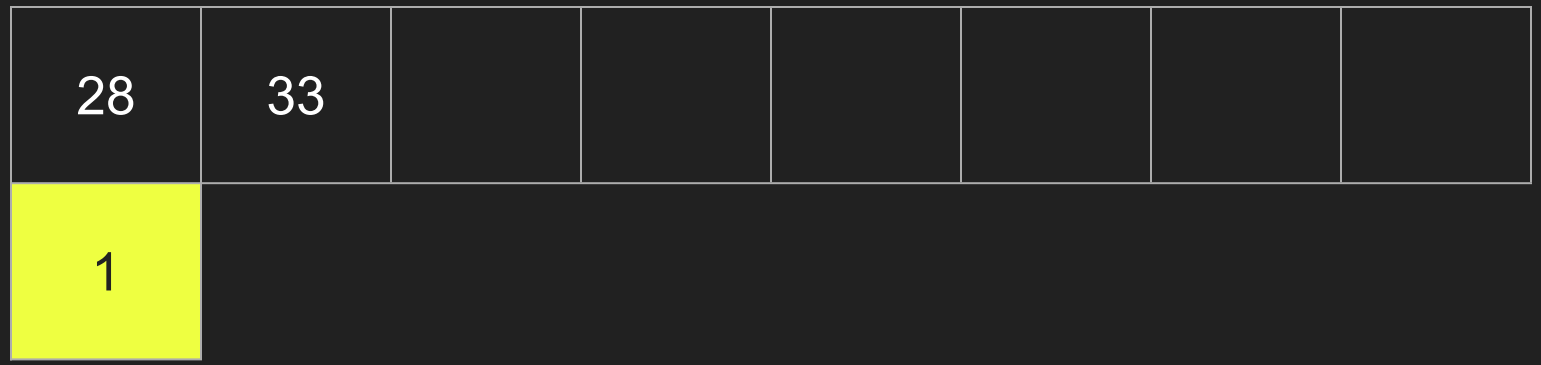
- Stack implemented as an array:

```
push(&s, 33);
```



- Stack implemented as an array:

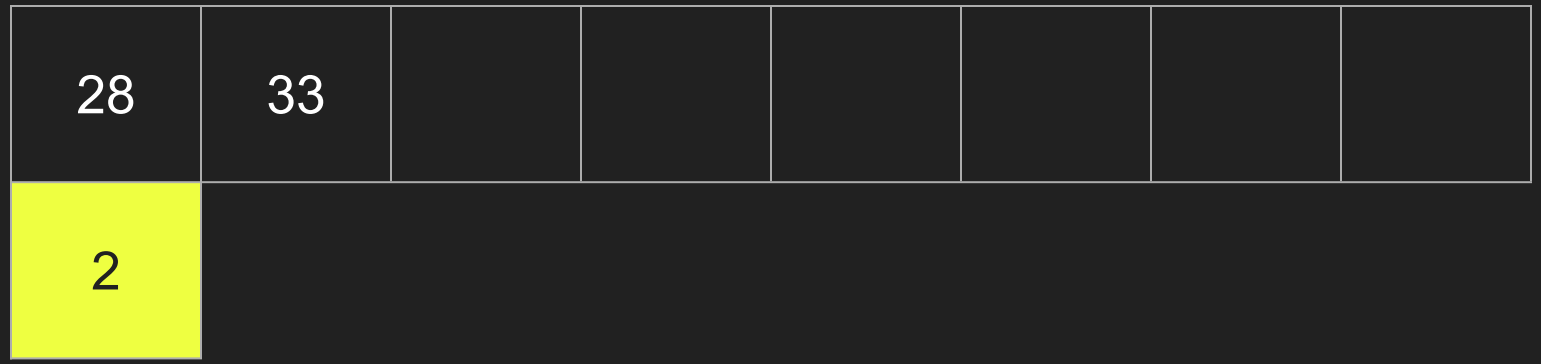
```
push(&s, 33);
```



Stack: pop

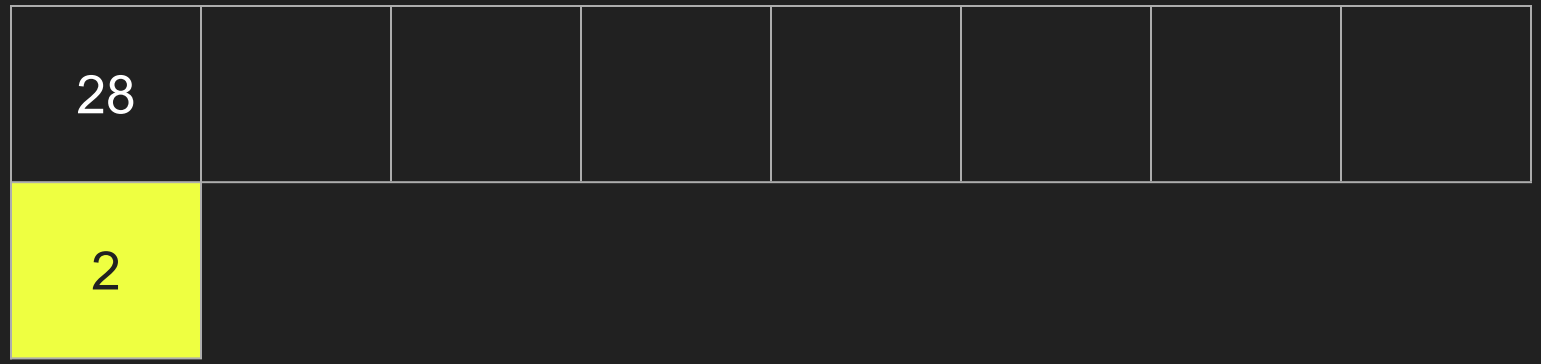
- Stack implemented as an array:

```
int x = pop(&s);
```



- Stack implemented as an array:

```
int x = pop(&s); // x gets 33
```



What will you get if I call pop again?

28!

Any questions for me?

Queues

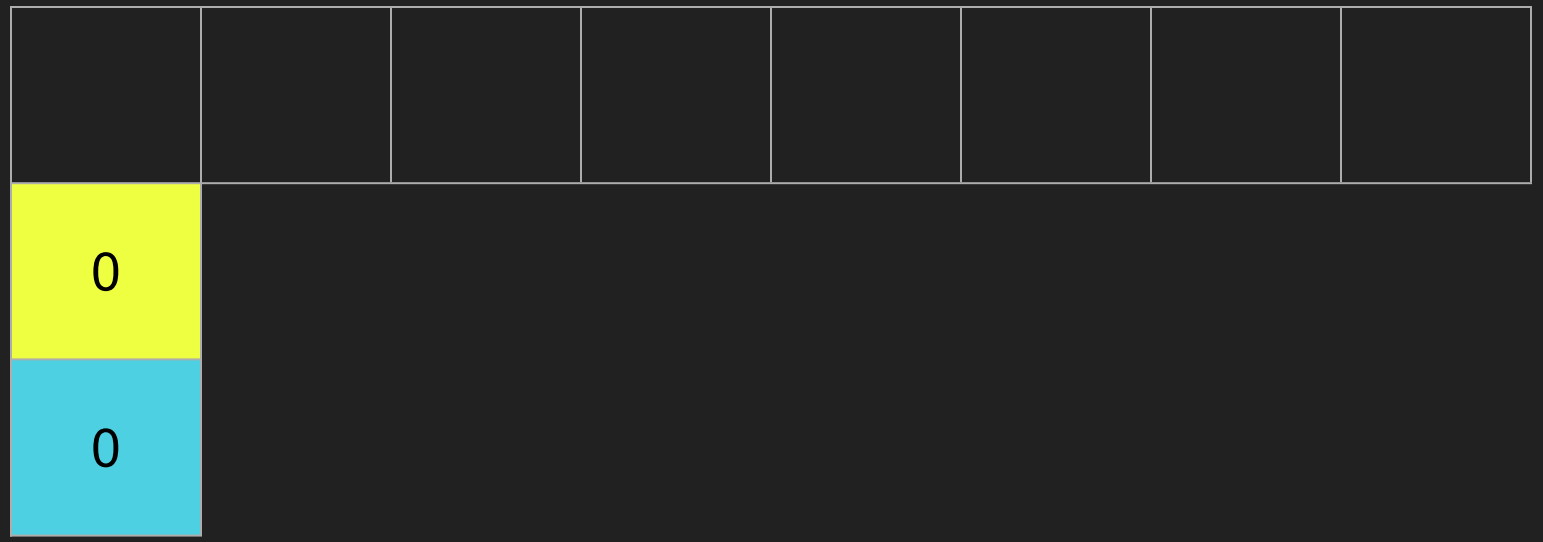
Queues: FIFO

- Queue implemented as an array:

```
typedef struct queue
{
    int array[CAPACITY];
    int front;
    int size;
}
queue;
```

- Queue implemented as an array:

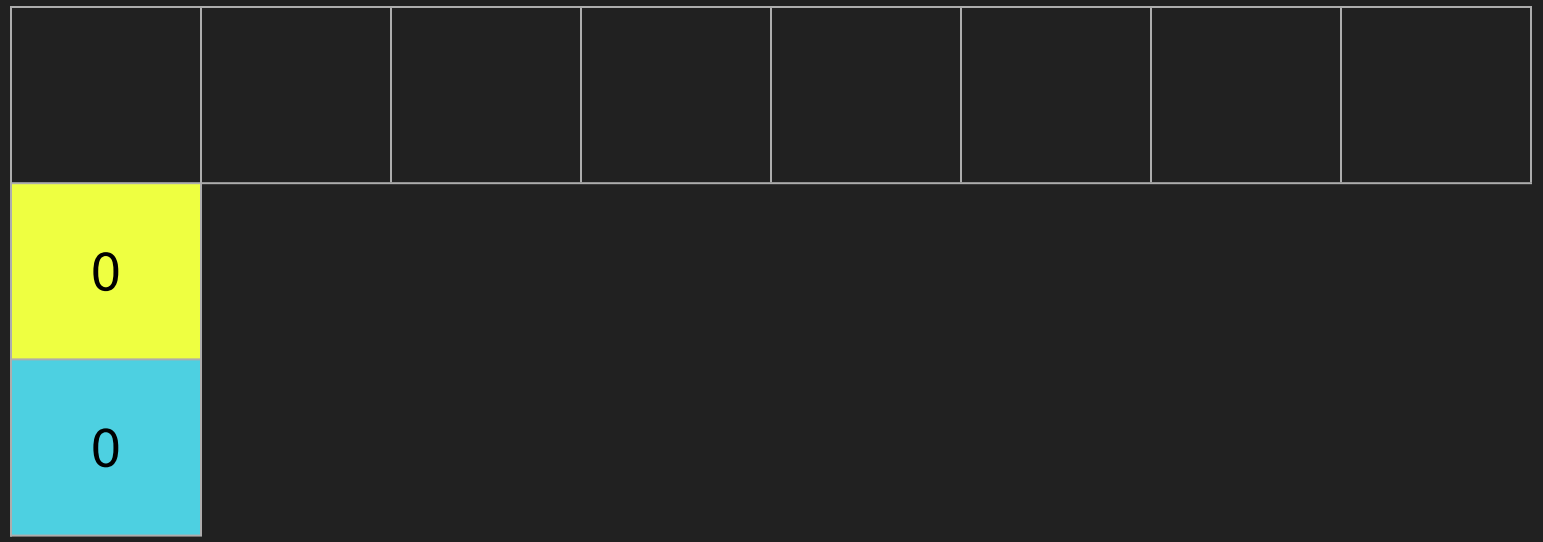
```
queue q;
q.front = 0;
q.size = 0;
```



Queues: Enqueue

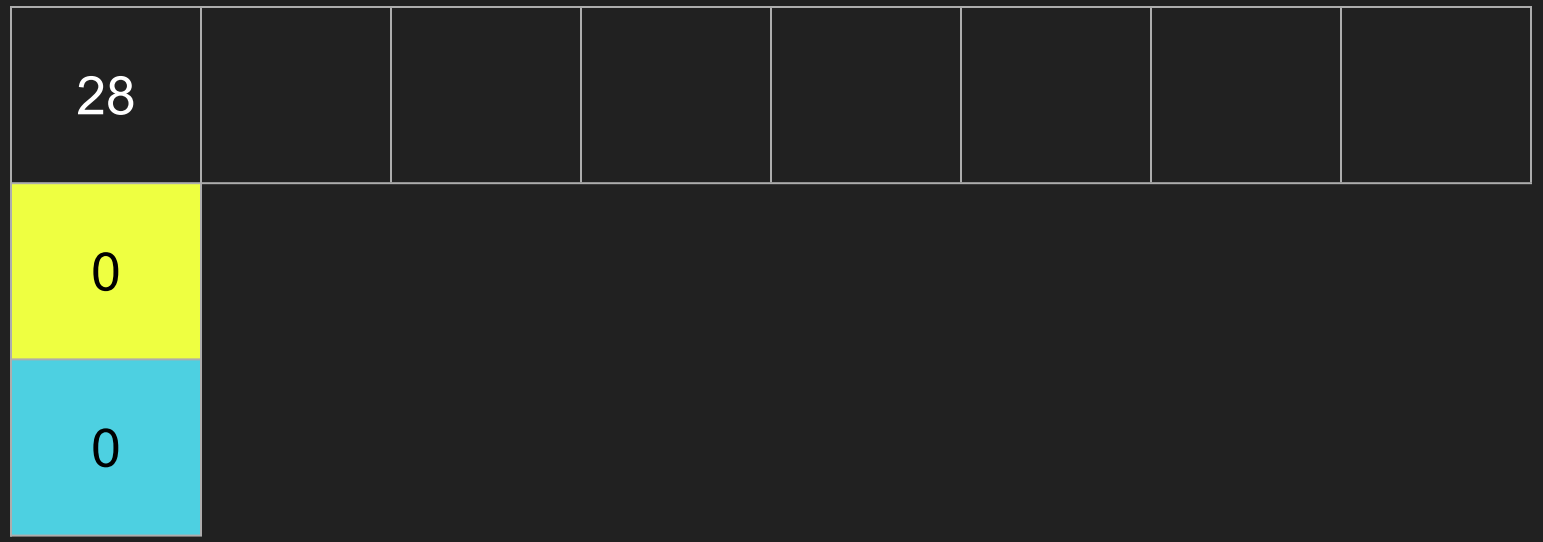
- Queue implemented as an array:

```
enqueue(&q, 28);
```



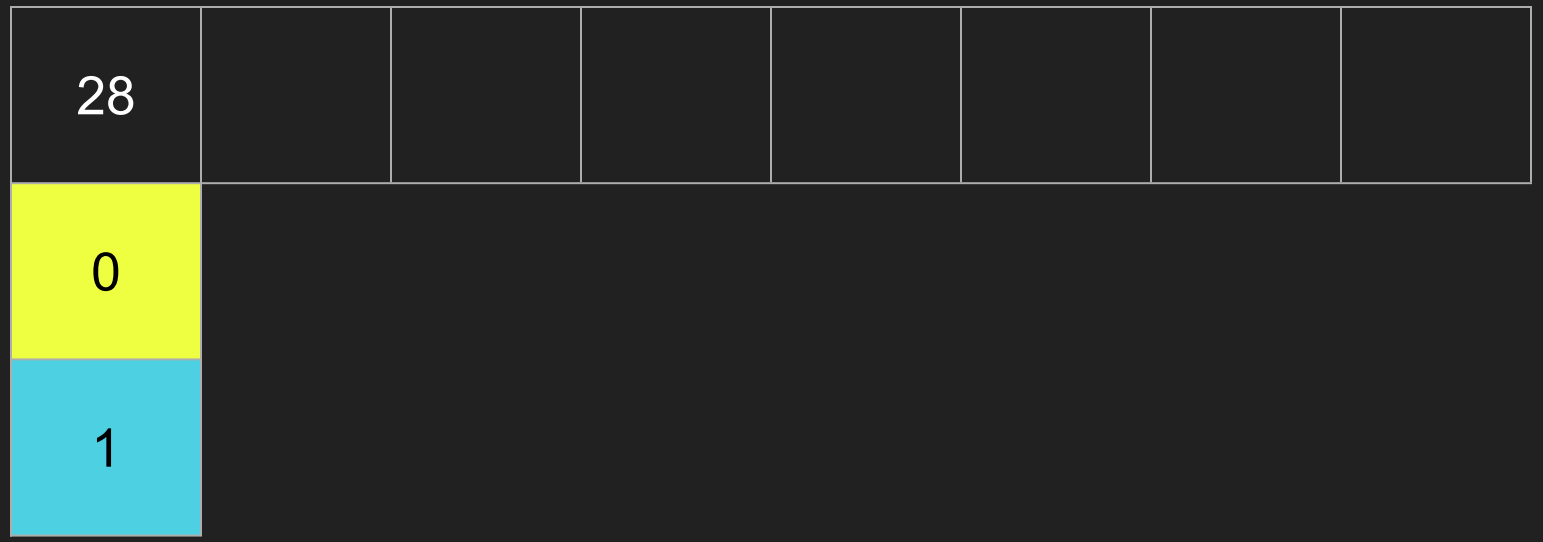
- Queue implemented as an array:

```
enqueue(&q, 28);
```



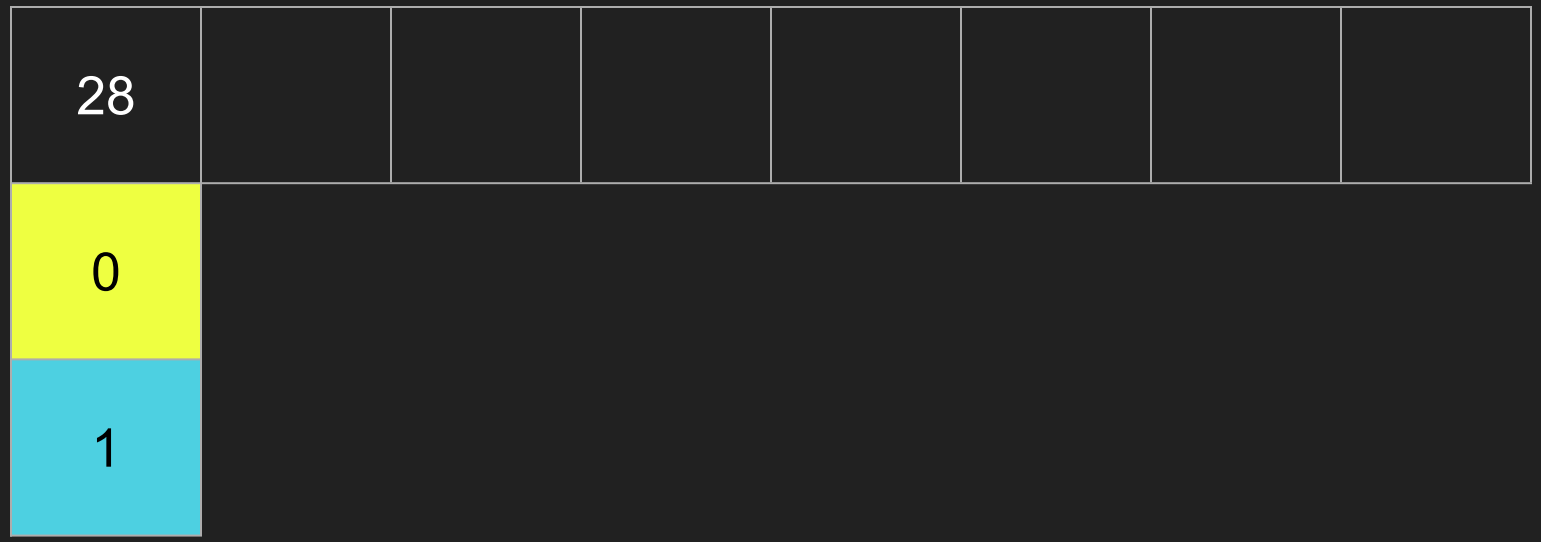
- Queue implemented as an array:

```
enqueue(&q, 28);
```



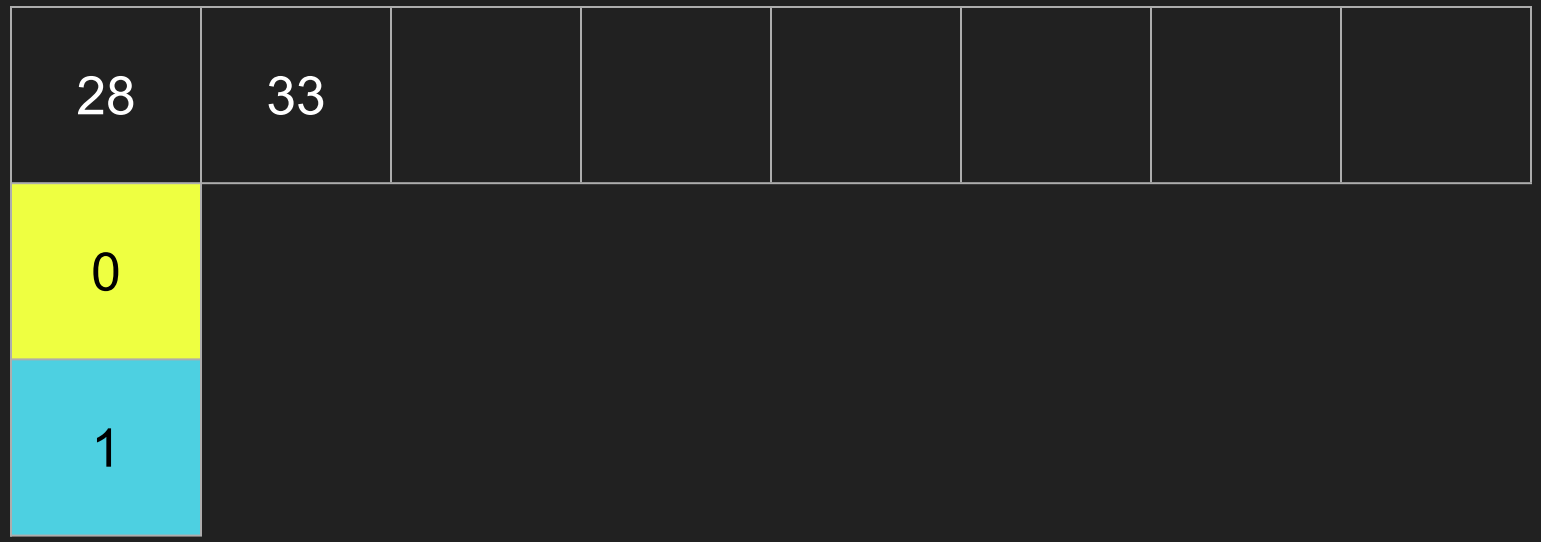
- Queue implemented as an array:

```
enqueue(&q, 33);
```



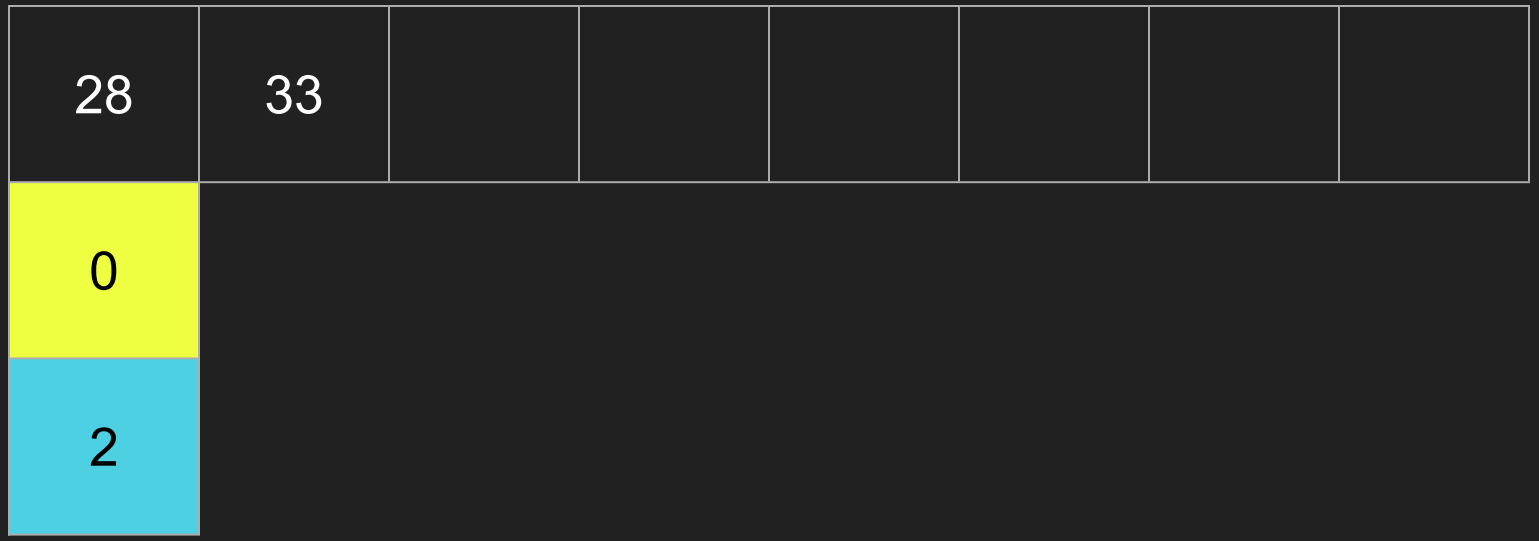
- Queue implemented as an array:

```
enqueue(&q, 33);
```



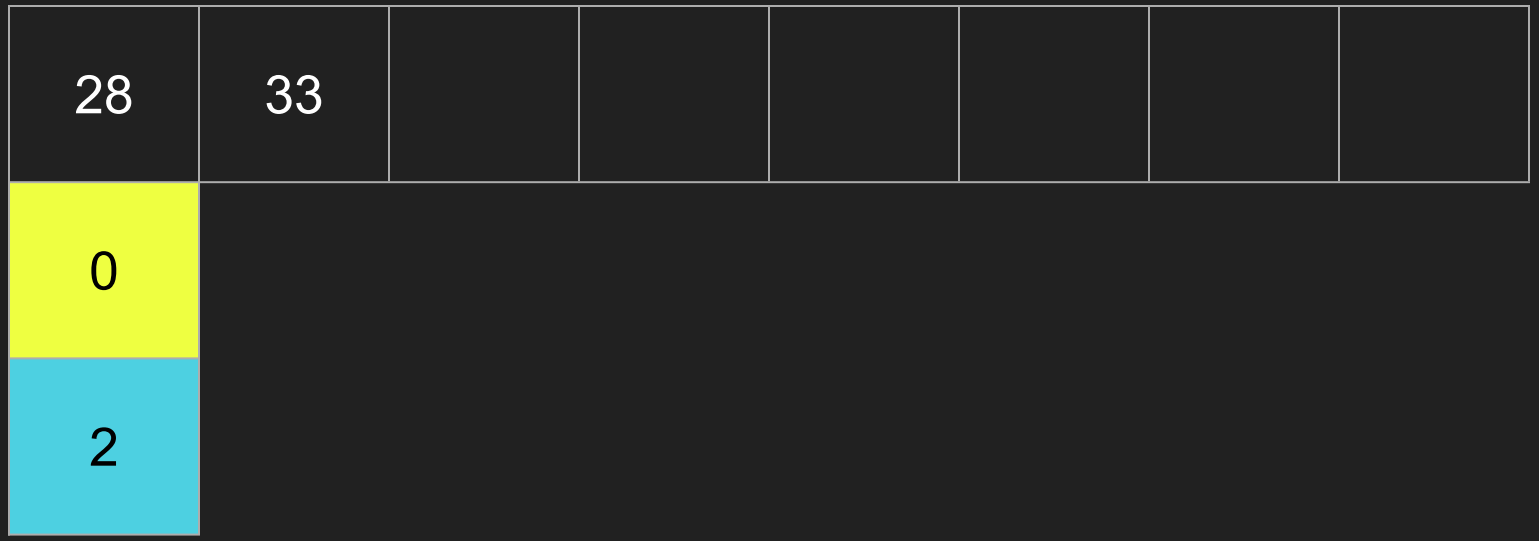
- Queue implemented as an array:

```
enqueue(&q, 33);
```



- Queue implemented as an array:

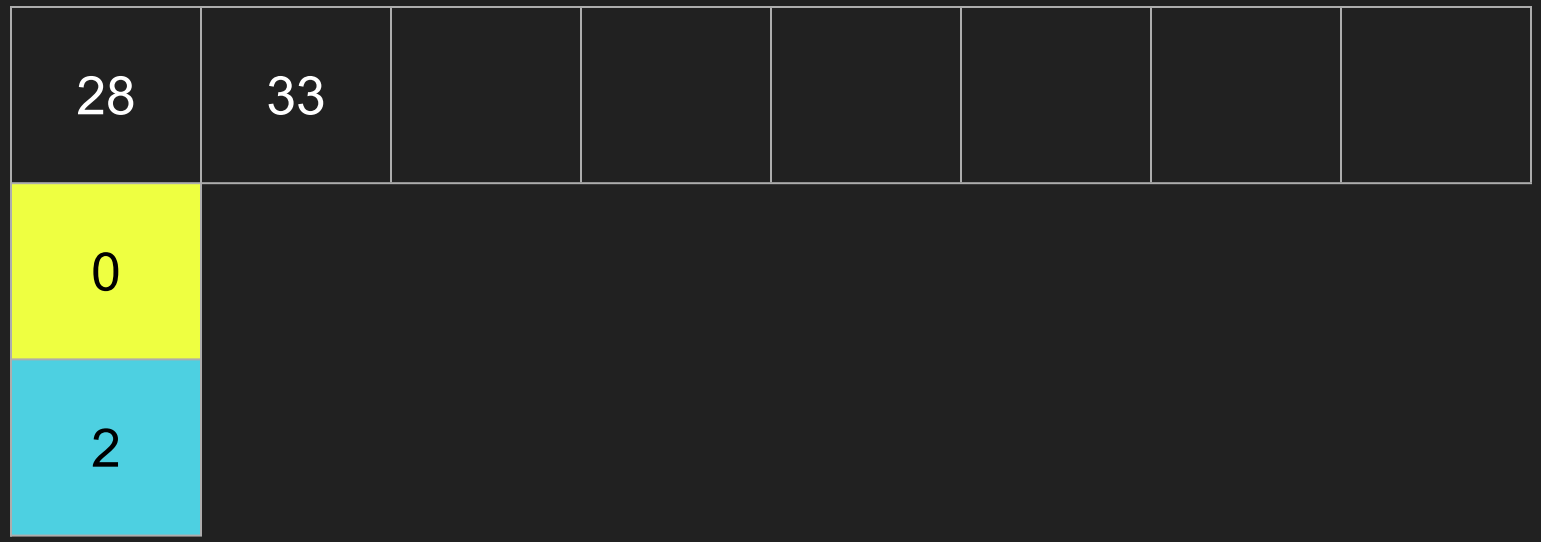
```
enqueue(&q, 19);
```



Queues: Dequeue

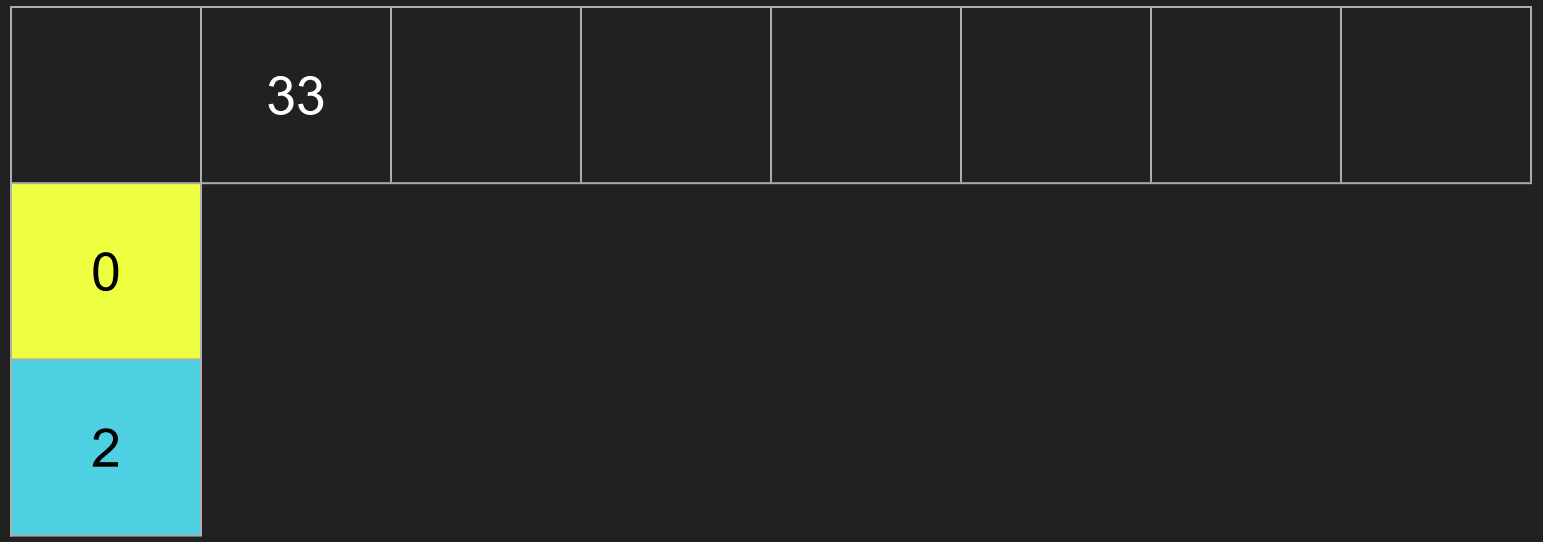
- Queue implemented as an array:

```
int x = dequeue(&q);
```



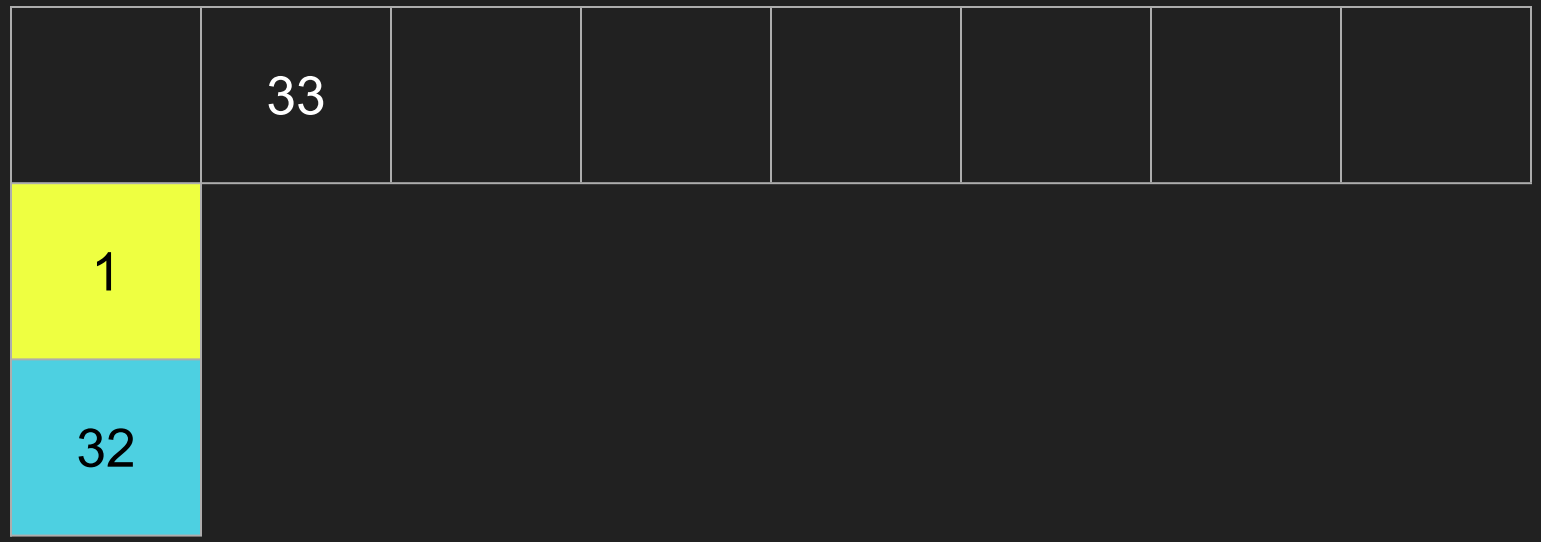
- Queue implemented as an array:

```
int x = dequeue(&q); // x gets 28
```



- Queue implemented as an array:

```
int x = dequeue(&q); // x gets 28
```



What will you get if I call dequeue again?

33!

Any questions for me?

STACKS AND QUEUES

No time to discuss these in section, but check out these resources:

- https://www.geeksforgeeks.org/stack-data-structure-introduction-program/
- https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/
- https://www.hackerearth.com/practice/notes/stacks-and-queues/

Any questions for me?

Lab

Task:

- The create_family function
- The free_family function

(1) `create_family` function

- First, you should allocate memory for a new person. Recall that you can use `malloc` to allocate memory, and `sizeof(person)` to get the number of bytes to allocate.

(1) `create_family` function

- Next, we've included a condition to check if `generations > 1`.
  - If `generations > 1`, then there are more generations that still need to be allocated. We've already created two new `parents, parent0` and `parent1`, by recursively calling `create_family.` Your `create_family` function should then set the parent pointers of the new person you created. Finally, assign both `alleles` for the new person by randomly choosing one allele from each parent.
- Otherwise (`if generations == 1`), then there will be no parent data for this person. Both `parents` of your new person should be set to `NULL`, and each `allele` should be generated randomly.
-

(1) `create_family` function

- Finally, your function should return a pointer for the `person` that was allocated.

Hint: `create_family` function

- You might find the `rand()` function useful for randomly assigning alleles. This function returns an integer between 0 and RAND_MAX, or 32767.
  - In particular, to generate a pseudorandom number that is either 0 or 1, you can use the expression `rand() % 2`.
- Remember, to allocate memory for a particular person, we can use `malloc(n)`, which takes a size as argument and will allocate n bytes of memory.
- Remember, to access a variable via a pointer, we can use arrow notation.
  - For example, if `p` is a pointer to a person, then a pointer to this person's first parent can be accessed by `p→parents[0]`.

(2) `free_family` function

- The `free_family` function should accept as input a pointer to a `person`, free memory for that person, and then recursively free memory for all of their ancestors.
  - Since this is a recursive function, you should first handle the base case. If the input to the function is `NULL`, then there's nothing to free, so your function can return immediately.
  - Otherwise, you should recursively `free` both of the person's parents before `free`ing the child.

Any questions for me?

Let’s do lab!

```c
person *create_family(int generations)
{
    // Allocate memory for new person
    person *new_person = malloc(sizeof(person));

    if (new_person == NULL)
    {
        printf("Memory error while creating new person.\n");
        return 1;
    }

    // If there are still generations left to create
    if (generations > 1)
    {
        // Create two new parents for current person by recursively calling create_family
        person *parent0 = create_family(generations - 1);
        person *parent1 = create_family(generations - 1);

        // Set parent pointers for current person
        new_person->parents[0] = parent0;
        new_person->parents[1] = parent1;

        // Randomly assign current person's alleles based on the alleles of their parents
        new_person->alleles[0] = parent0->alleles[rand() % 2];
        new_person->alleles[1] = parent1->alleles[rand() % 2];
    }

    // If there are no generations left to create
```

```c
    // If there are no generations left to create
    else
    {
        // Set parent pointers to NULL
        new_person->parents[0] = NULL;
        new_person->parents[1] = NULL;

        // Randomly assign alleles
        new_person->alleles[0] = random_allele();
        new_person->alleles[1] = random_allele();
    }

    // Return newly created person
    return new_person;
}
```

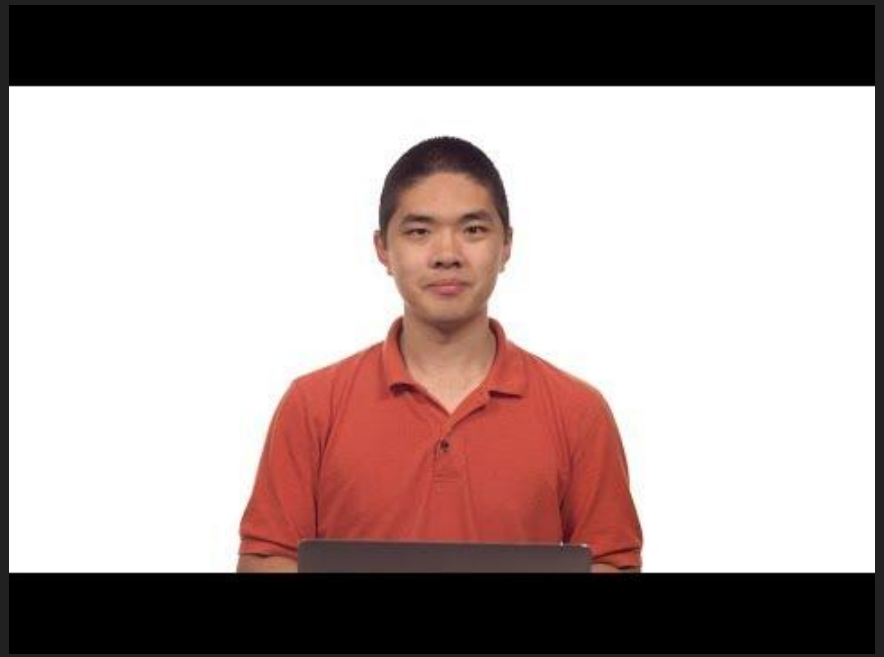
```c
/// Free `p` and all ancestors of `p`.
void free_family(person *p)
{
    // Handle base case
    if (p == NULL)
    {
        return;
    }

    // Free parents recursively
    free_family(p->parents[0]);
    free_family(p->parents[1]);

    // Free child
    free(p);
}
```

Any questions for me?

Problem Set Tips

Problem Set: Speller

Prompt walkthrough, watch Bryan's Video



Five functions to code:

- `load`
- `hash`
- `size`
- `check`
- `unload`

Overview: `load`

Complete the `load` function.

- **Loads hash table into memory. Returns true if successful else false.**
- **More information at Bryan's video!**

Pseudocode

```
// initialize dictionary size (for count function later)

// open the dictionary

// check whether you have successfully open the dictionary

// iterate through the file, one word at a time

    // create memory for new hash node (malloc)
    // check whether malloc is successful or not

    // read the next string from the source dictionary into new node (fscanf==EOF)
        // if we have reached the end, free node
        // break
    // hash the string to obtain a hash value
    // insert into the hash table at the location
    // increase dictionary count

// Close the dictionary file, then return true.
```

Overview: hash

- You can use any hash function you like or create one!
- As long as when you pass in a word, it return an integer so it can be hashed!!

Overview: `check`

1) Complete the `check` function.
    - Returns true if word is in hashtable else false.
- More information in Bryan's walkthrough!

Pseudocode

```
// Hash the word received to get the corresponding hash element

// set the pointer to the hashcode (node *sth = table[hashcode])

(The above will return a list)

// while loop through the above pointer

    // case sensitive comparison (use strcasecmp!)

        // return true

    // advance the pointer

// return false (not found)
```

Overview: `size`

1) Complete the `size` function.
    - Return the size of the dictionary (hash table)
    - Tips: keep the size in your `load` function!

Pseudocode

```
// return dictionary_size
```

Overview: `unload`

1) Complete the `unload` function.
    - will call free on any nodes using malloc and will return true if can be done
    - iterate over the length lists and free them
    - Don't free up memory before using it so use a variable to keep track (make up a tmp)
    - make the cursor equal to the next element and free the tmp, but not necessarily the cursor

Pseudocode

```
// loop through every element in our hash table
    // if the head == NULL,
        //do nothing
    // free the next element in the chain
    // free the one

// return true
```

Tutorials, OHs, More 1-1 too!! :)

Feedback form:

tinyurl.com/zad-feedback

Thank you!

See you next week!