

# CS50 Section

Week 5

**Attendance Sign In:** [tinyurl.com/y5u2x4m6](https://tinyurl.com/y5u2x4m6)

# Agenda

- Week 4: Remaining Pset / General Questions
- Week 5
  - Linked Lists
  - Hash Tables
  - Trees
  - Stacks + Queues

# Week 4 Debrief

# Week 4 Review - What Questions do You Have?

- Hexadecimal
- Pointers
- Dynamic Memory Management:
  - Malloc
  - Free
- File I/O

# Week 5

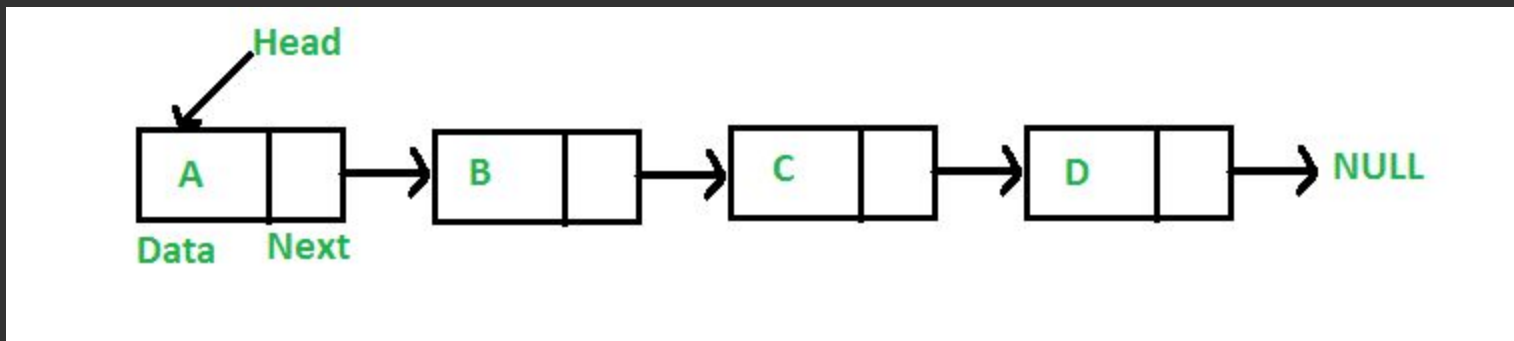
# Today We'll Cover

- Linked Lists
- Hash Tables
- Trees
- Stacks + Queues

# Linked Lists

# LINKED LISTS

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





# LINKED LISTS

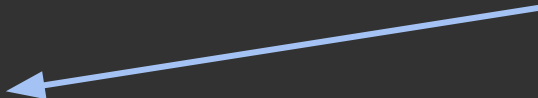
The fundamental basis of any linked list is the **node** structure:

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

# LINKED LISTS

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

# LINKED LISTS

What are the advantages of a linked list?

# LINKED LISTS

What are the advantages of a linked list?

- **They're dynamically-sized, so we can add/remove elements (arrays are fixed!)**
- **As a corollary, insertion/deletion is really easy**
- **It efficiently uses memory: we just unlink nodes when we no longer need them**

# LINKED LISTS

What are the disadvantages of a linked list?

# LINKED LISTS

What are the disadvantages of a linked list?

- **They take extra memory to store: an array just stores the elements, but linked lists need the node structure**
- **You have to traverse multiple elements to access a single node (you can't just go directly to a specific index like you can with arrays)**

# Working with Linked Lists

Today we'll try:

1. Allocating a new node
2. Adding a new node to the head of a linked list
3. Printing all nodes
4. Freeing all nodes

# Exercise:

## Working with Linked Lists

<http://bit.ly/2ntgcZH>



# Part 1

## Allocating a New Node

# Part 1

## Allocating a New Node

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

# Part 2

## Add new node to start of linked list

# Part 2

## Add new node to start of linked list

```
n->next = list;  
list = n;
```

# Part 3

## Print all nodes

# Part 3

## Print all nodes

```
for (node *ptr = list; ptr != NULL; ptr = ptr->next)
{
    printf("%i\n", ptr->number);
}
```

# Part 4

## Free all nodes

# Part 4

## Free all nodes

```
node *ptr = list;
while (ptr != NULL)
{
    node *tmp = ptr;
    ptr = ptr->next;
    free(tmp);
}
```



# Exercise Solution:

## Working with Linked Lists

<http://bit.ly/31Z8nki>

# Bonus

## Adding to Ends of Linked Lists

```
if (list == NULL)
{
    list = n;
}
else
{
    node *ptr = list;
    while (ptr->next != NULL)
    {
        ptr = ptr->next;
    }
    ptr->next = n;
}
```

# Hash Tables

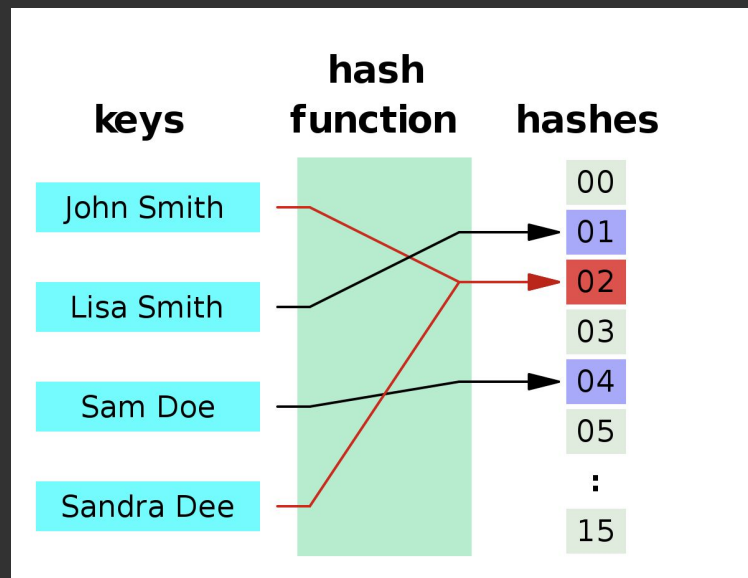
# HASH TABLES

What is a **hash function**?

# 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.”



# HASH TABLES

A **hash table** is a data structure which uses a hash function and an array to determine where in the array to store elements

You run your elements through the hash function, get back a hash value, and then use that as index in your array to store the value

# HASH TABLES

A really simple example of this:

```
#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:

```
#include <string.h>

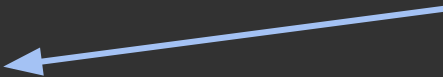
int hash(char *word);

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?

```
#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?

```
#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.**

# HASH TABLES

What do we do when we get a collision?

# HASH TABLES

What do we do when we get a collision?

We can solve it in two ways:

1. Linear probing
2. Chaining

# HASH TABLES

## Linear Probing:

- If we have a collision, try to place the data instead in the next consecutive element of the array, trying each consecutive element until we find a vacancy.
- That way, if we don't find it right where we expect it, it's hopefully nearby.

Can lead to **clustering**—statistically more likely to form “clusters” with other collisions reducing effectiveness of the hash table; Also, # of elements is capped at the size of our array

# HASH TABLES

## Chaining:

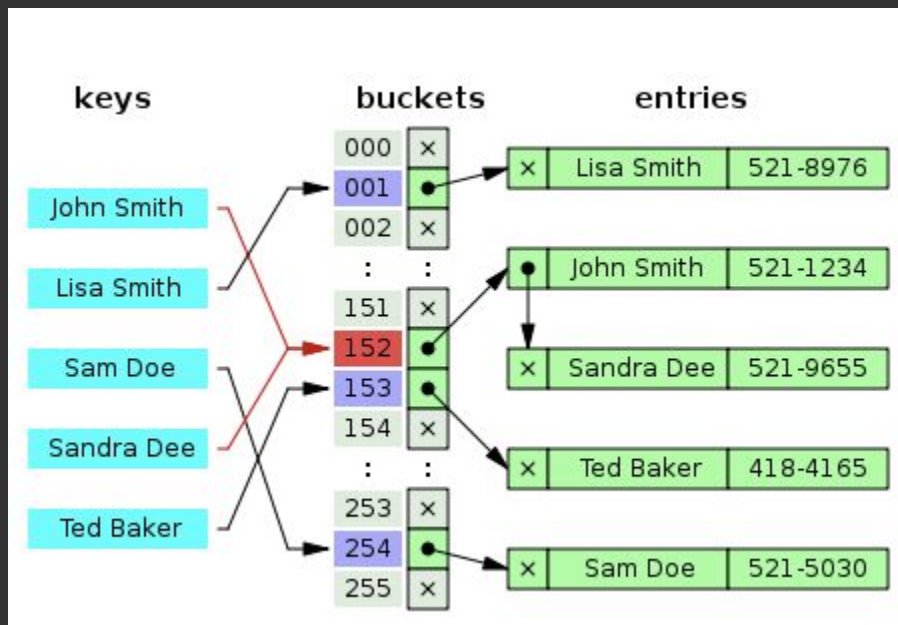
- Each element of the array is now a pointer to the head of a linked list.
- If we have a collision, create a new node and add it to the chain at that location.
- That way, if it's in the chain at the hash code location, it's in the data structure.

Not subject to the clustering problem from before and our array can now store as many elements as you have memory for



# HASH TABLES

## Chaining:



# KEY-VALUE PAIRS

Many of the data structures we've discussed in CS50 thus far conform to the *key-value pair* pattern:

You have a **key** which is unique or mostly unique for each **value** that it represents

# KEY-VALUE PAIRS

Examples:

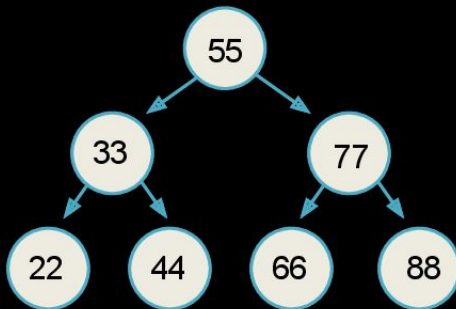
- Arrays - The index is the key and the data at that location is the value
- Hash Tables - The hash code of the data is the key and the slot in the array at that index is the value

# Trees

# TREES

- A **tree** is a hierarchical data type in computer science with *many* applications

## Binary Search Tree



# TRIES

- One type of tree is a **trie**, an ordered tree optimized for searching (usually strings)
- Tries follow the key-value pattern too, but *implicitly*
  - There is no explicit key defined for each element; Instead, the position of the element in the tree is its key
  - The data located at that position is the value

# TRIES

A simple trie to store years (keys) and the universities founded during those years (values):

```
typedef struct trie
{
    char university[20];
    struct trie *paths[10];
}
node;
```

# TRIES



# TRIES

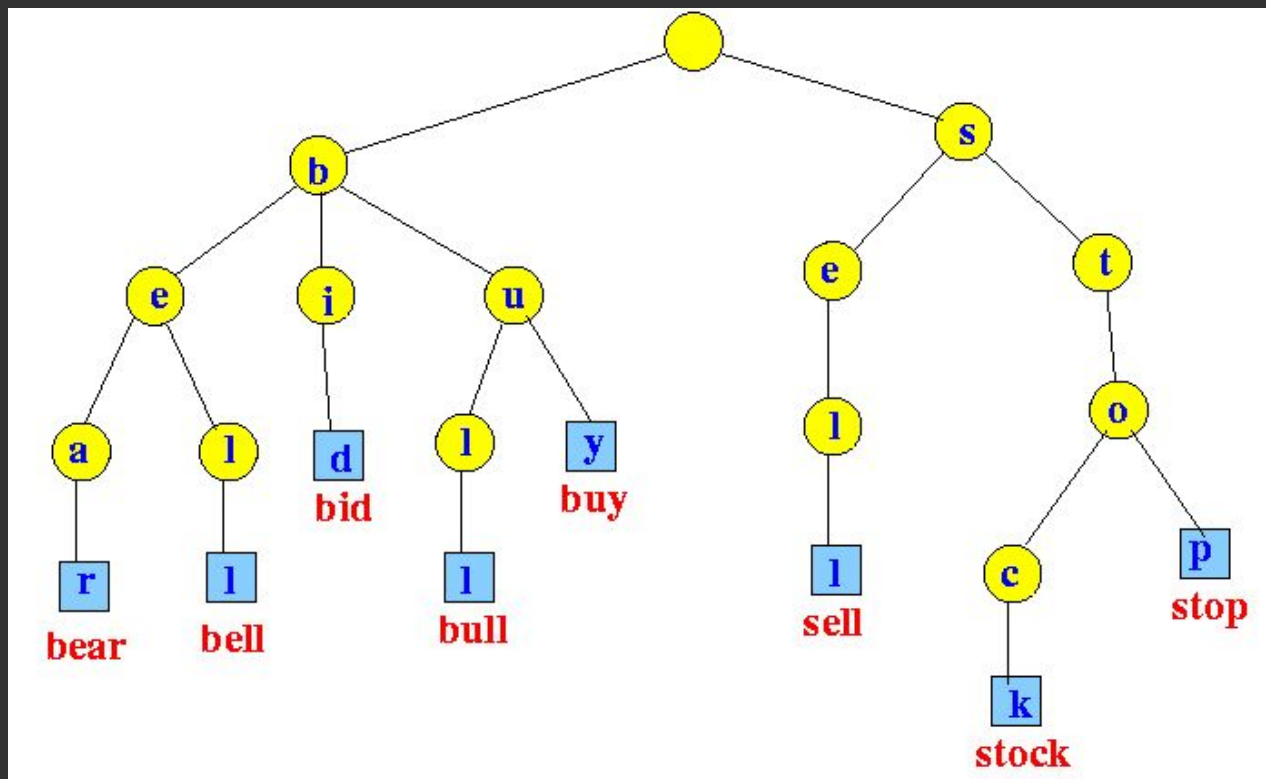
To search for an element in the trie, use successive digits to navigate from the root, and if you make it to the end without hitting a dead end (a `NULL` pointer in this case), simply read the data at your current location.

# TRIES

# TRIES

It's more common to represent words in a trie as the values (guaranteed to be unique as opposed to the example from before—two universities could have been founded in the same year)

# TRIES



# TRIES

What are the advantages and disadvantages of a trie?

# TRIES

What are the advantages and disadvantages of a trie?

- They are very fast because they have a constant lookup time (based off of the size of the word)
- If constructed off unique keys, collisions are impossible
- They need a lot of memory—Each node must point to all different possibilities at that node (e.g. each letter of the alphabet at *every level*)
- A very good hash table might have even faster lookup depending on how you write your hash function

# Stacks

- Stacks are an abstract data type used primarily to organize data. They are most commonly implemented as either arrays or linked lists.
- Regardless of the underlying implementation, when data is added to the stack it sits “on top”, and so if an element needs to be removed, the most recently added element is the only one that can be.
- Last in, first out (LIFO).



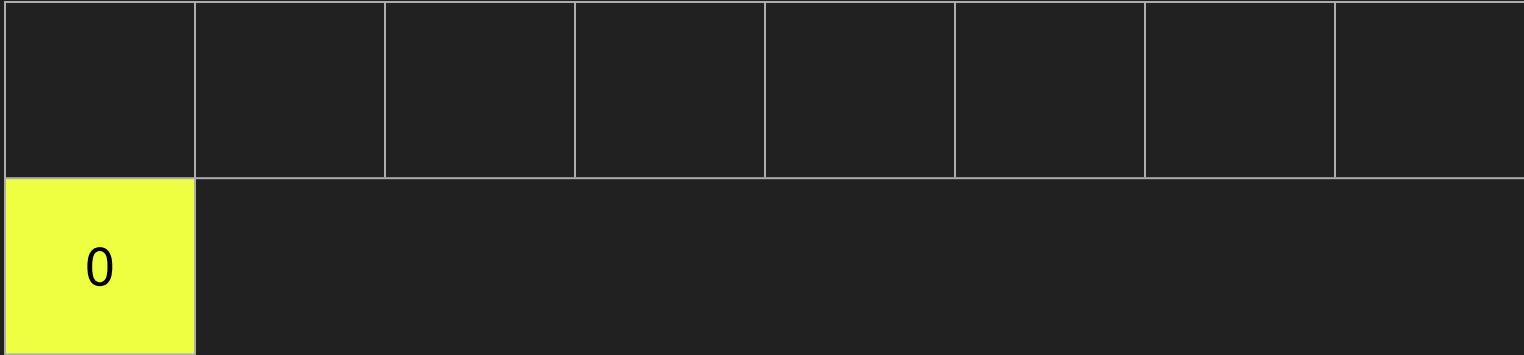
- Only two operations may be performed on stacks:
  - Push: Add a new element to the top of the stack.
  - Pop: Remove the most recently added element from the top of the stack.

- Stack implemented as an array:

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

- Stack implemented as an array:

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

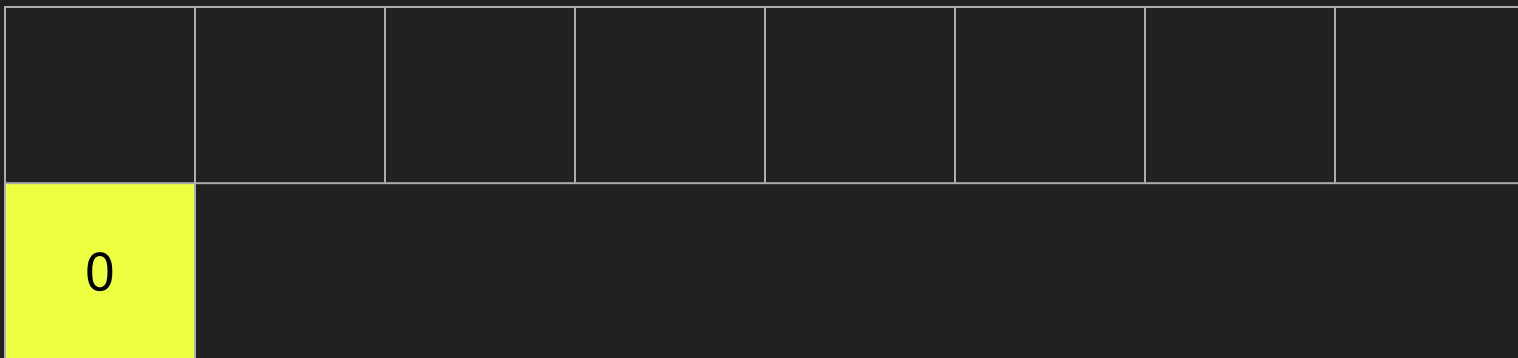


- Push:

- Accept a pointer to the stack (so that you can actually modify the contents notwithstanding push being a separate function).
- Accept data to be added to the stack.
- Add that data to the stack at the top of the stack.
- Change the location of the top of the stack (so that the next piece of data can be properly inserted there.)

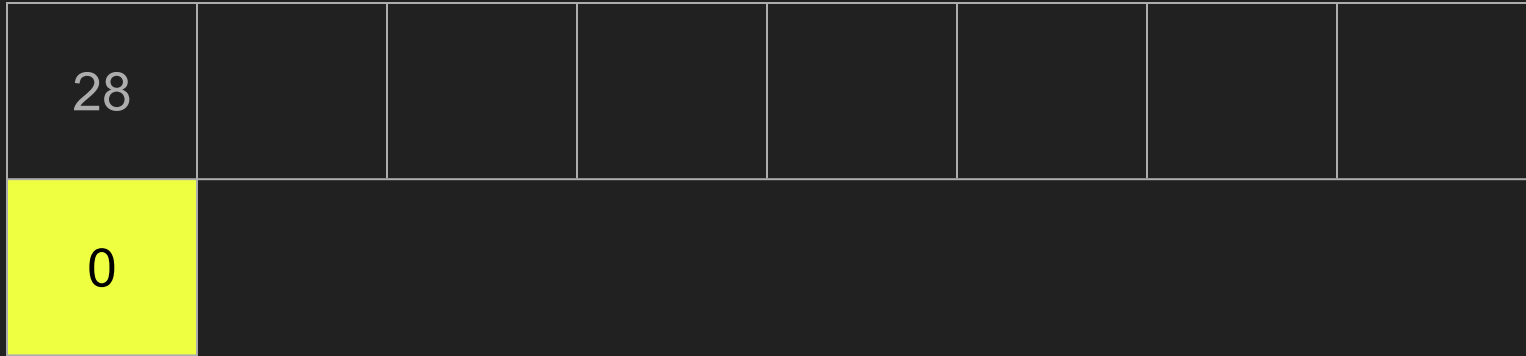
- Stack implemented as an array:

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



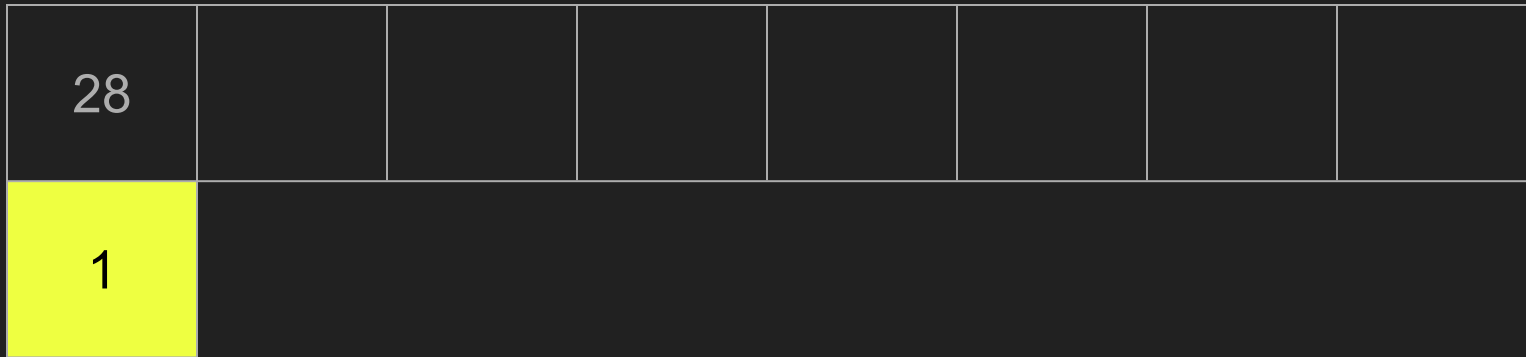
- Stack implemented as an array:

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



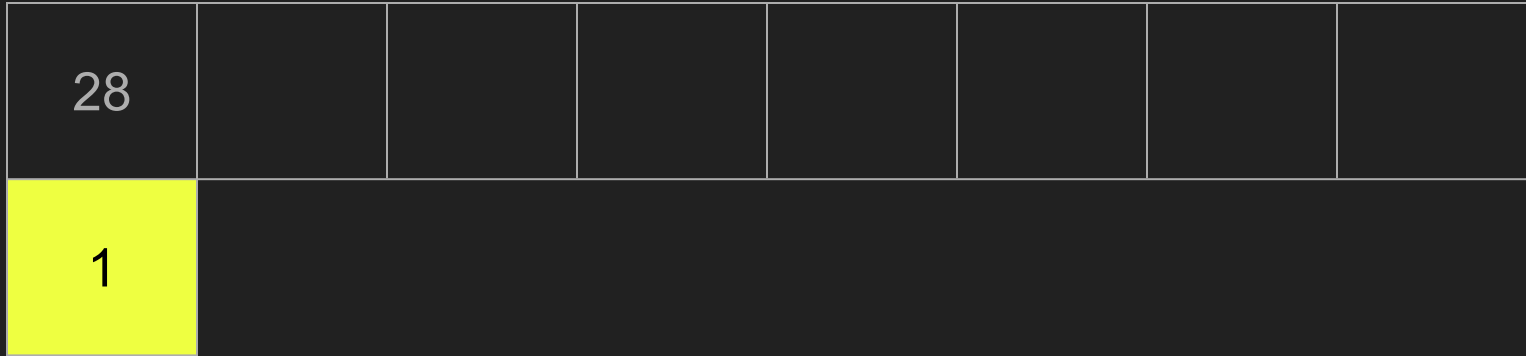
- Stack implemented as an array:

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



- Stack implemented as an array:

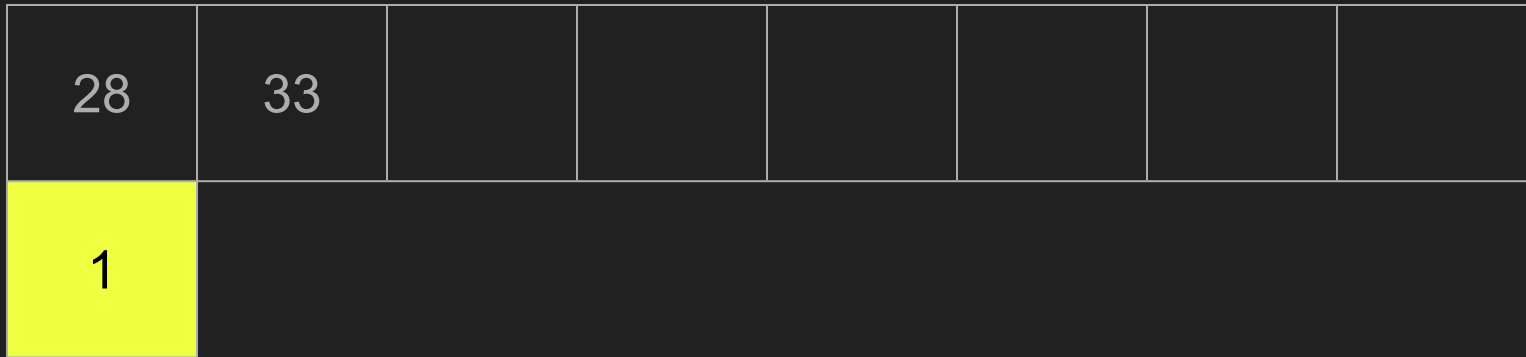
```
push(&s, 33);
```





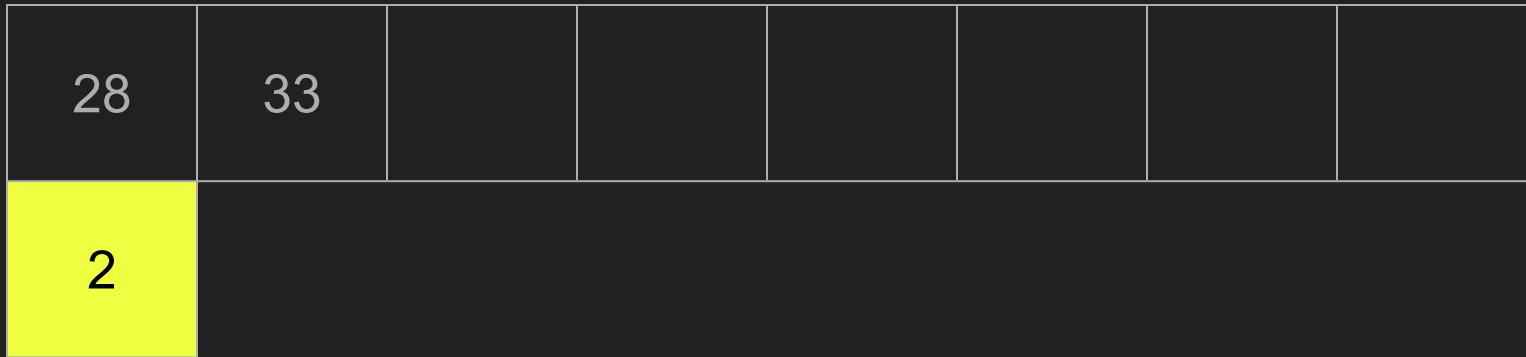
- Stack implemented as an array:

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



- Stack implemented as an array:

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

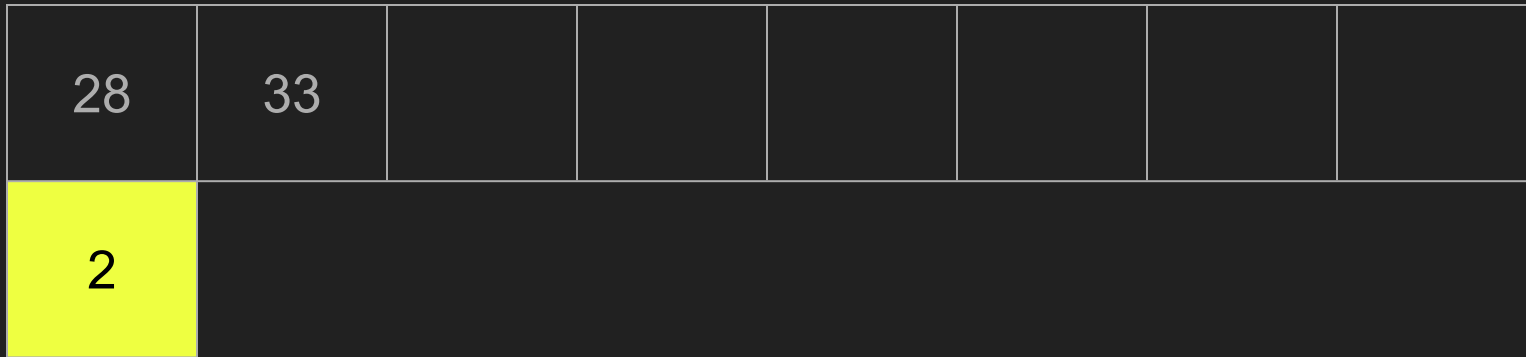


- Pop:

- Accept a pointer to the stack (so that you can actually modify the contents notwithstanding pop being a separate function).
- Change the location of the top of the stack.
- Return the value that was removed from the stack.

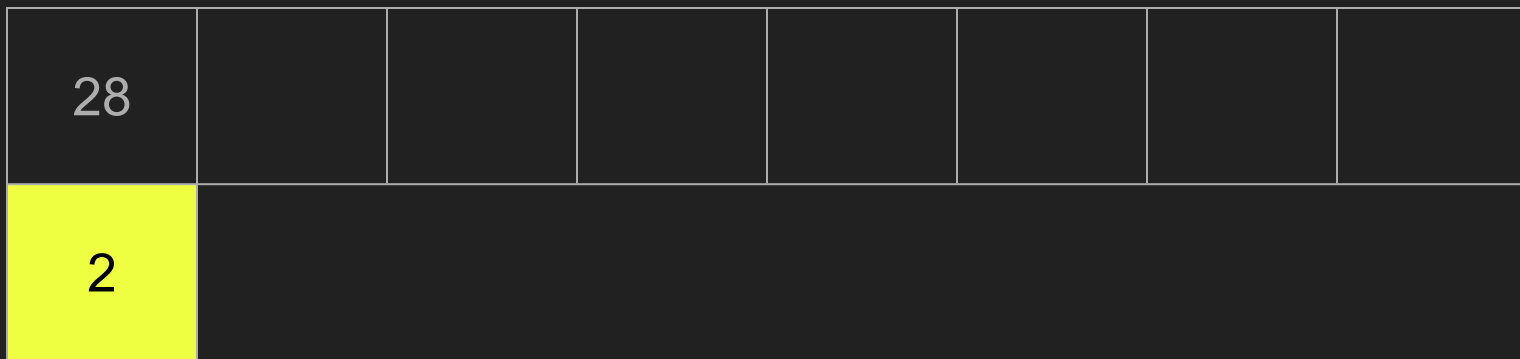
- Stack implemented as an array:

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



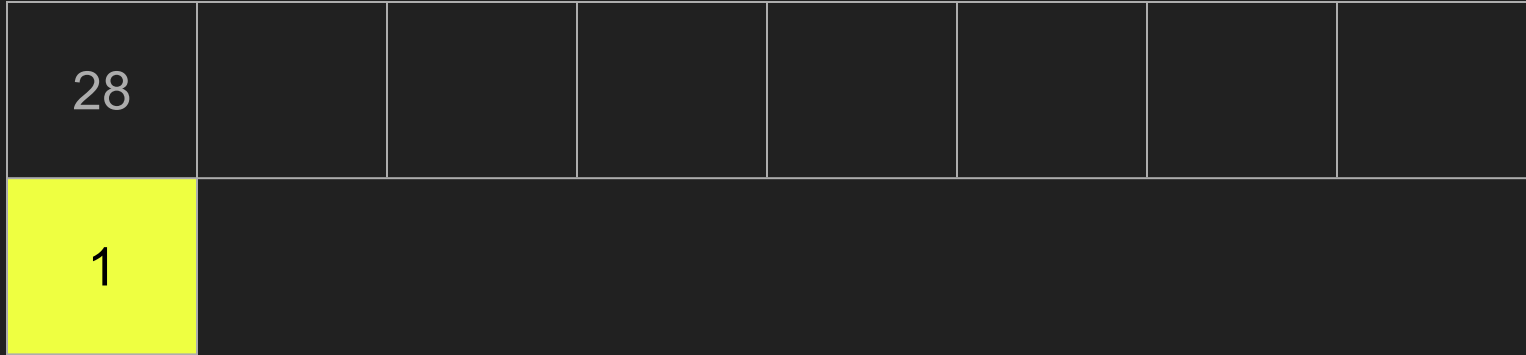
- Stack implemented as an array:

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



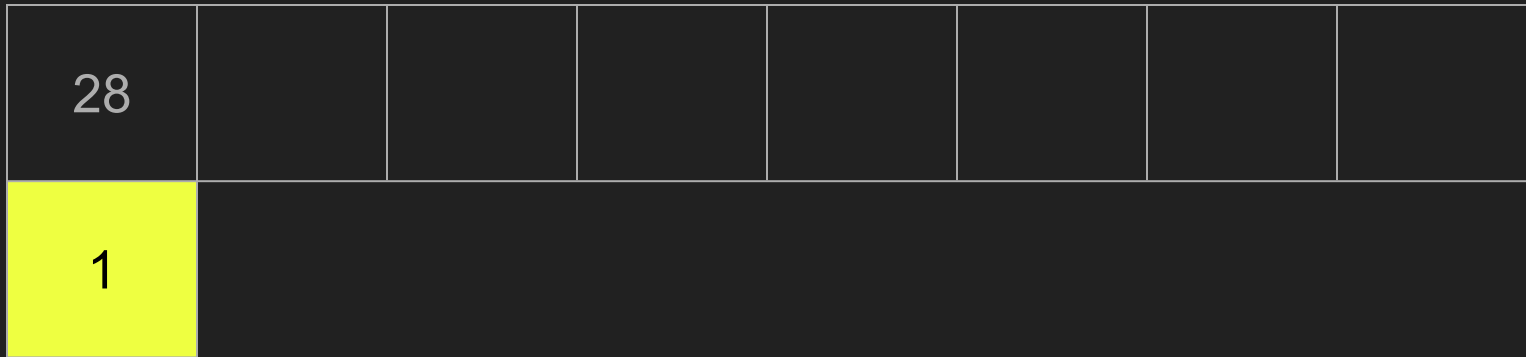
- Stack implemented as an array:

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



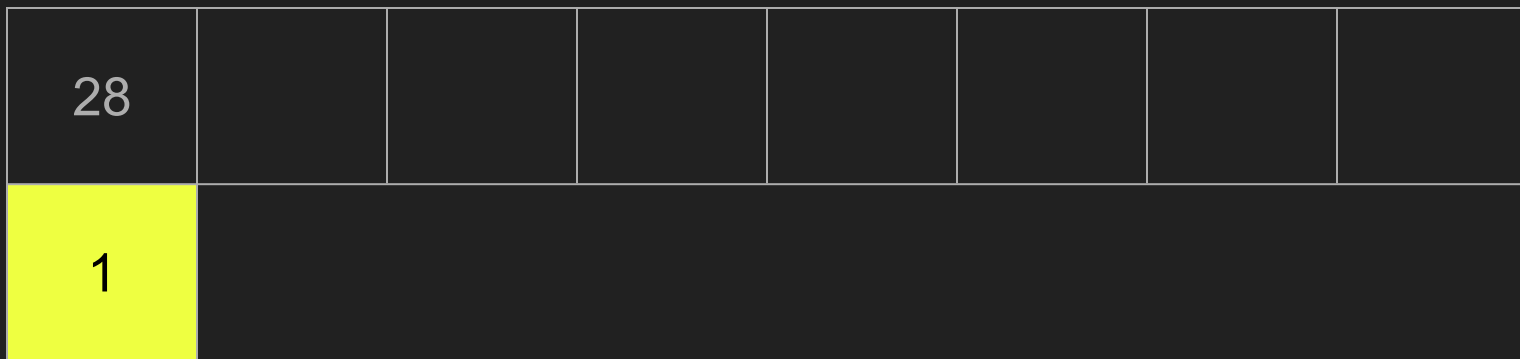
- Stack implemented as an array:

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



- Stack implemented as an array:

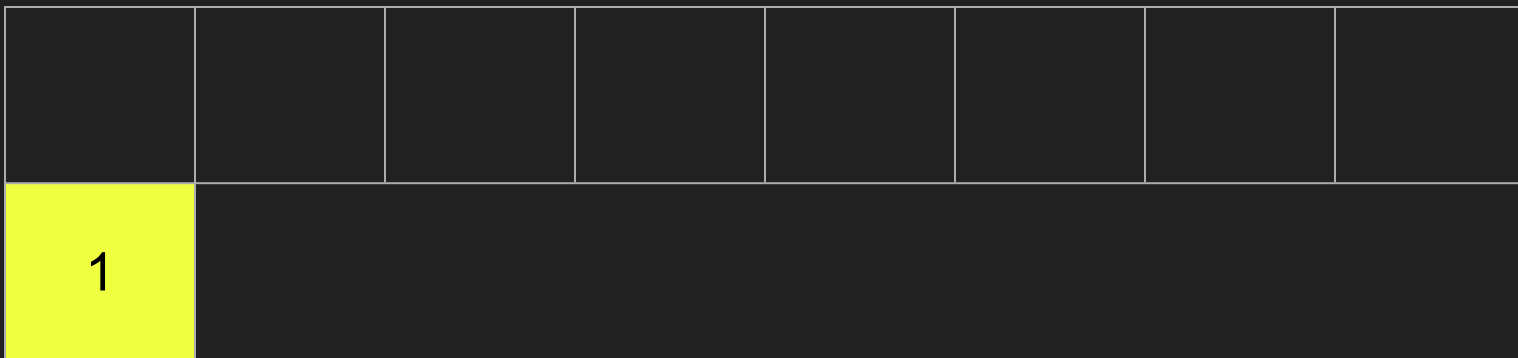
```
int x = pop(&s); // x gets 28
```





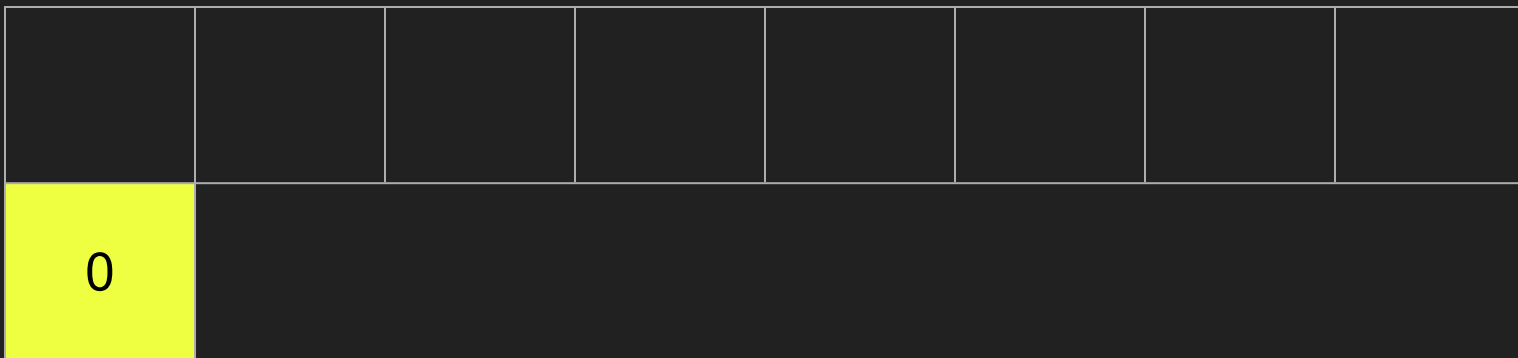
- Stack implemented as an array:

```
int x = pop(&s); // x gets 28
```



- Stack implemented as an array:

```
int x = pop(&s); // x gets 28
```



- Stack implemented as a linked list:

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

- Push:

- Just like maintaining any other linked list.
- Make space for a new list node, populate it.
- Chain that node into the front of the linked list.
- Make the new head of the linked list the node you just added.

- Pop:

- Make the new head of the linked list the second node of the list (if you can).
- Extract the data from the first node.
- Free that node.

# Queues

- Queues are an abstract data type used primarily to organize data. They are most commonly implemented as either arrays or linked lists.
- Regardless of the underlying implementation, when data is added to the queue it is added to the “end”, and if an element needs to be removed, the oldest element (at the “front”) is the only one that can be.
- First in, first out (FIFO).

- Only two operations may be performed on queues:
  - Enqueue: Add a new element to the end of the queue.
  - Dequeue: Remove the oldest element from the front of the queue.

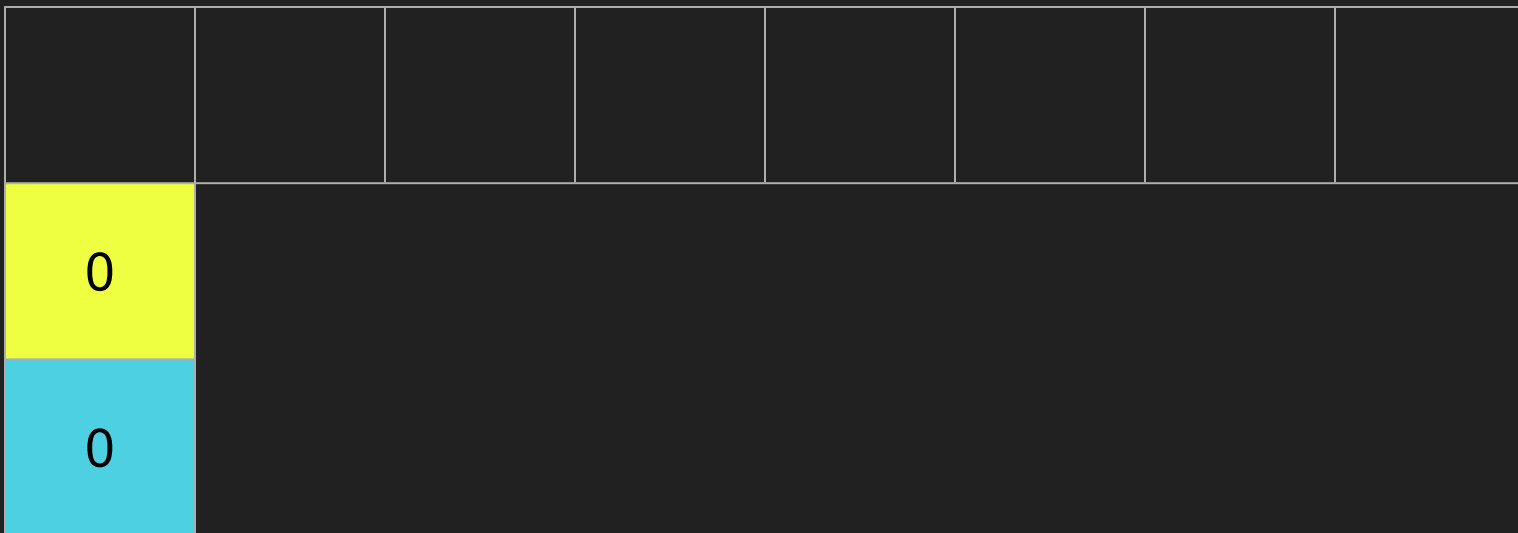
- Queue implemented as an array:

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



- Queue implemented as an array:

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



- Enqueue:

- Accept a pointer to the queue (so that you can actually modify the contents notwithstanding enqueue being a separate function).
- Accept data to be added to the queue.
- Add that data to the queue at the end of the queue.
- Change the size of the queue (so that the next piece of data can be properly inserted there.)

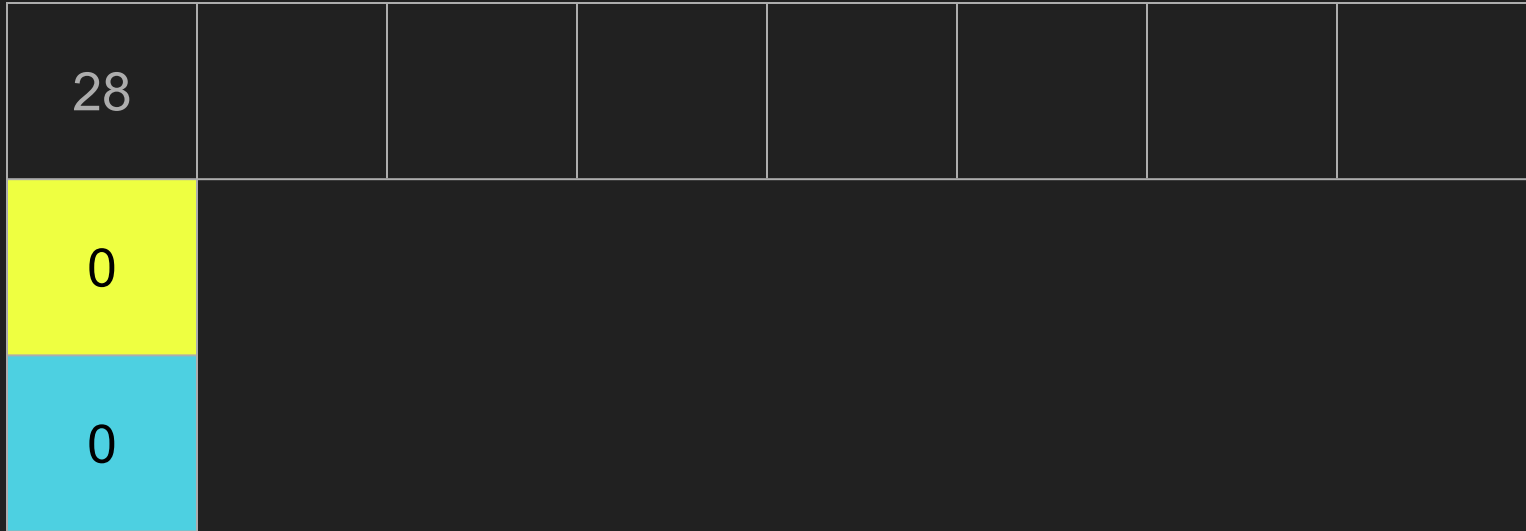
- Queue implemented as an array:

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



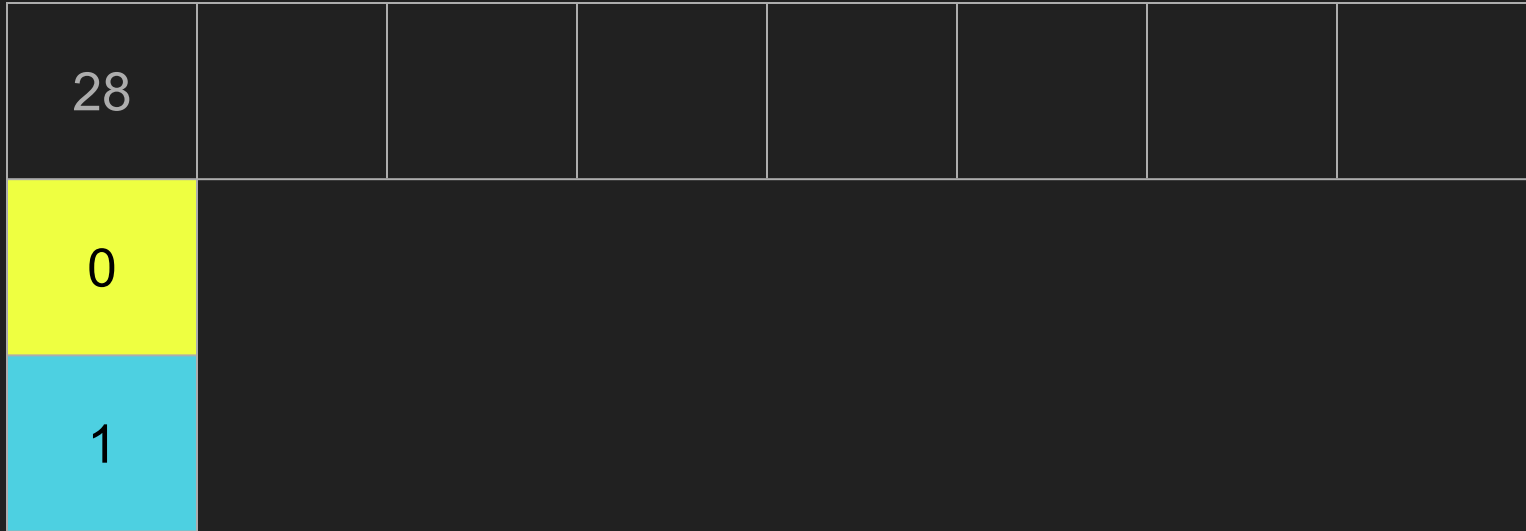
- Queue implemented as an array:

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



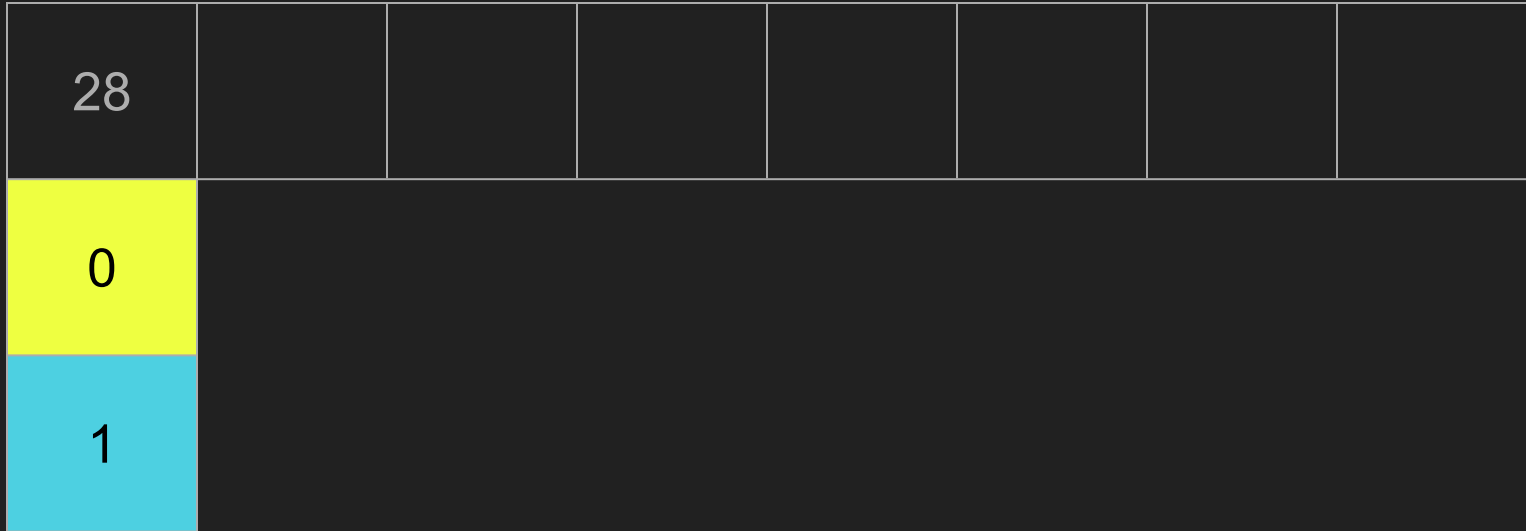
- Queue implemented as an array:

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



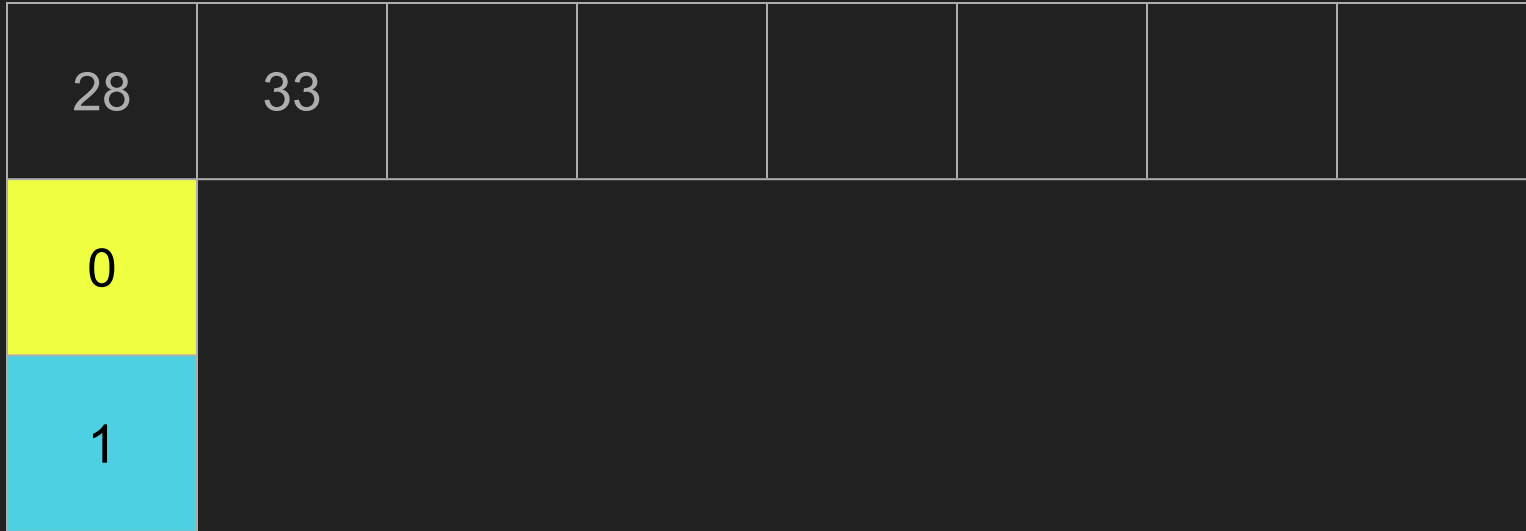
- Queue implemented as an array:

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



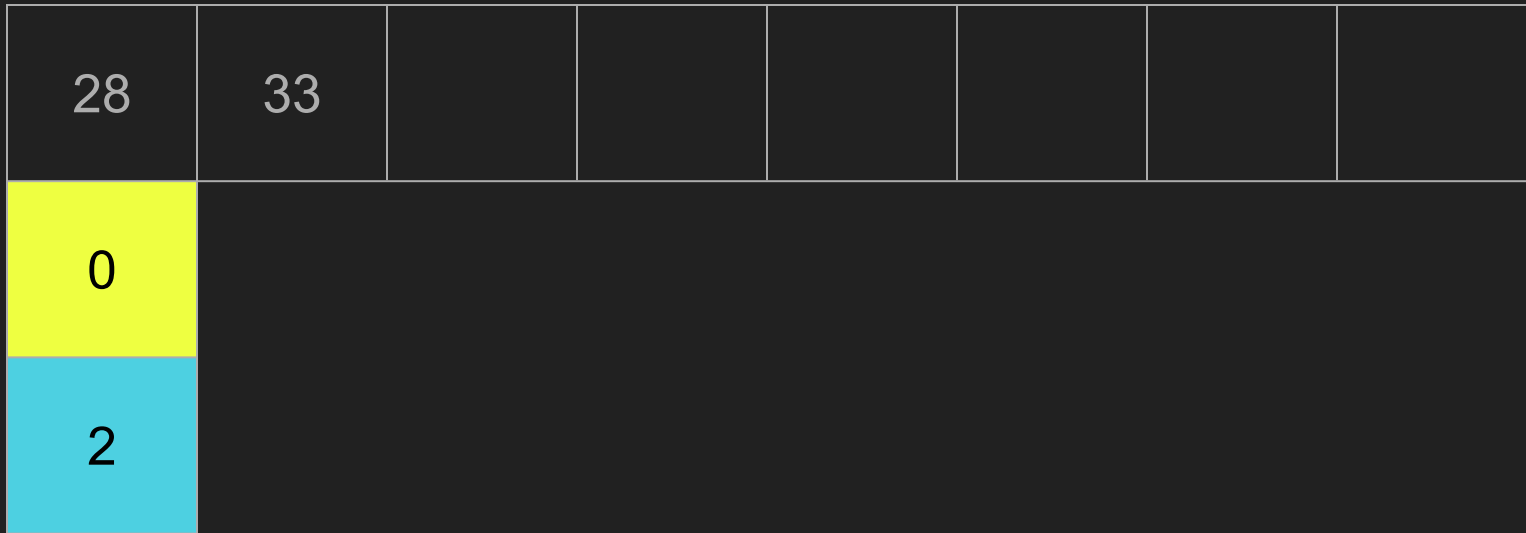
- Queue implemented as an array:

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



- Queue implemented as an array:

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





- Queue implemented as an array:

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

28	33						
0							
2							

- Queue implemented as an array:

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

28	33	19					
0							
2							

- Queue implemented as an array:

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

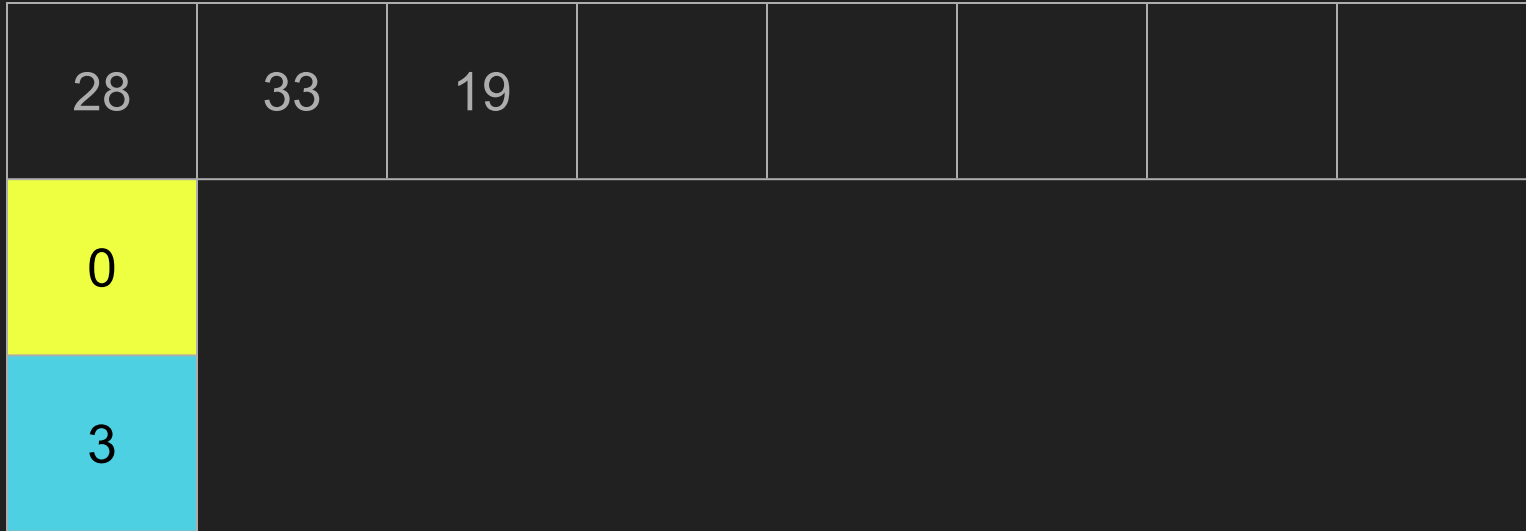
28	33	19					
0							
3							

- Dequeue:

- Accept a pointer to the queue (so that you can actually modify the contents notwithstanding dequeue being a separate function).
- Change the location of the front of the queue.
- Change the size of the queue.
- Return the value that was removed from the queue.

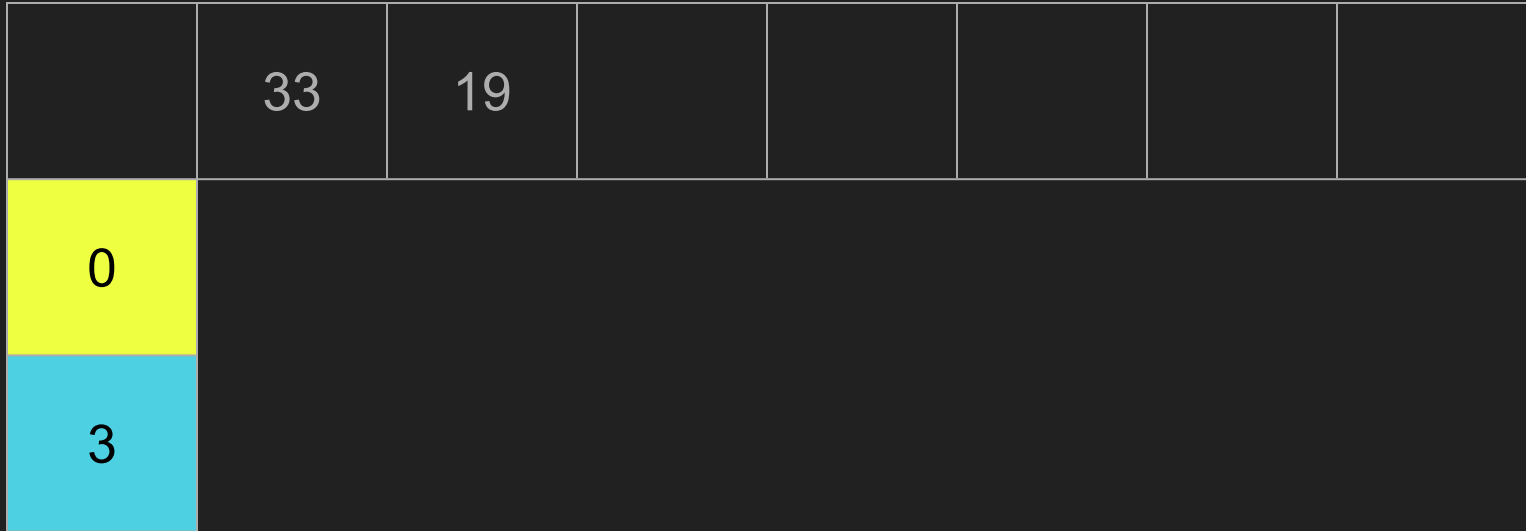
- Queue implemented as an array:

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



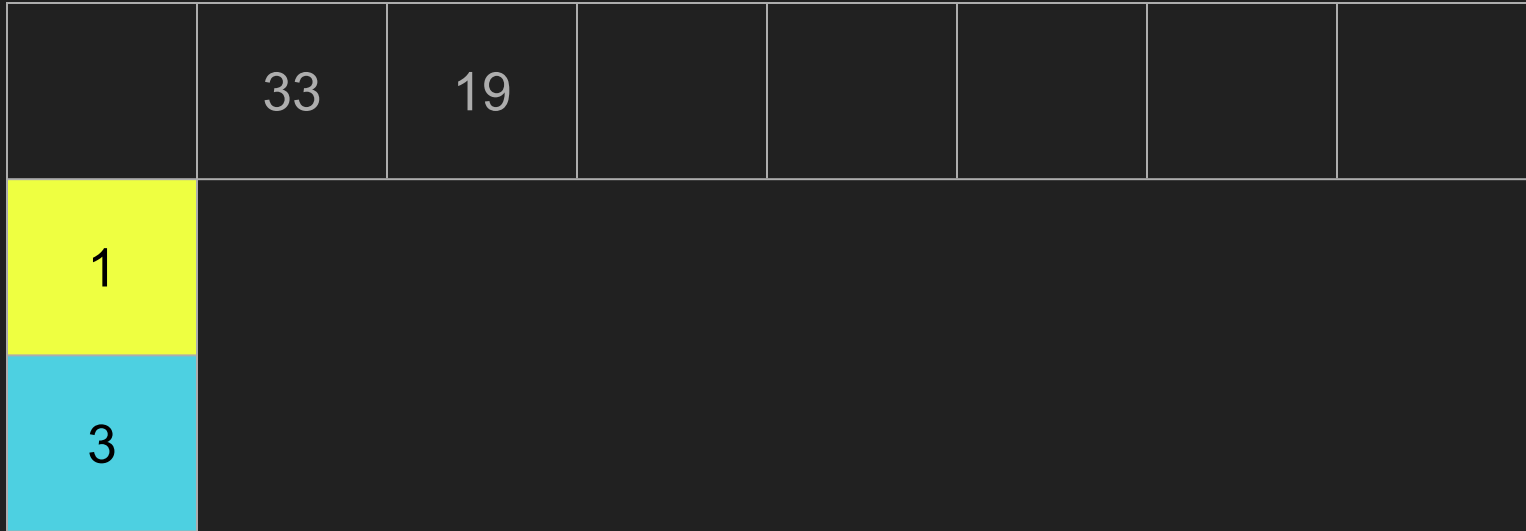
- Queue implemented as an array:

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



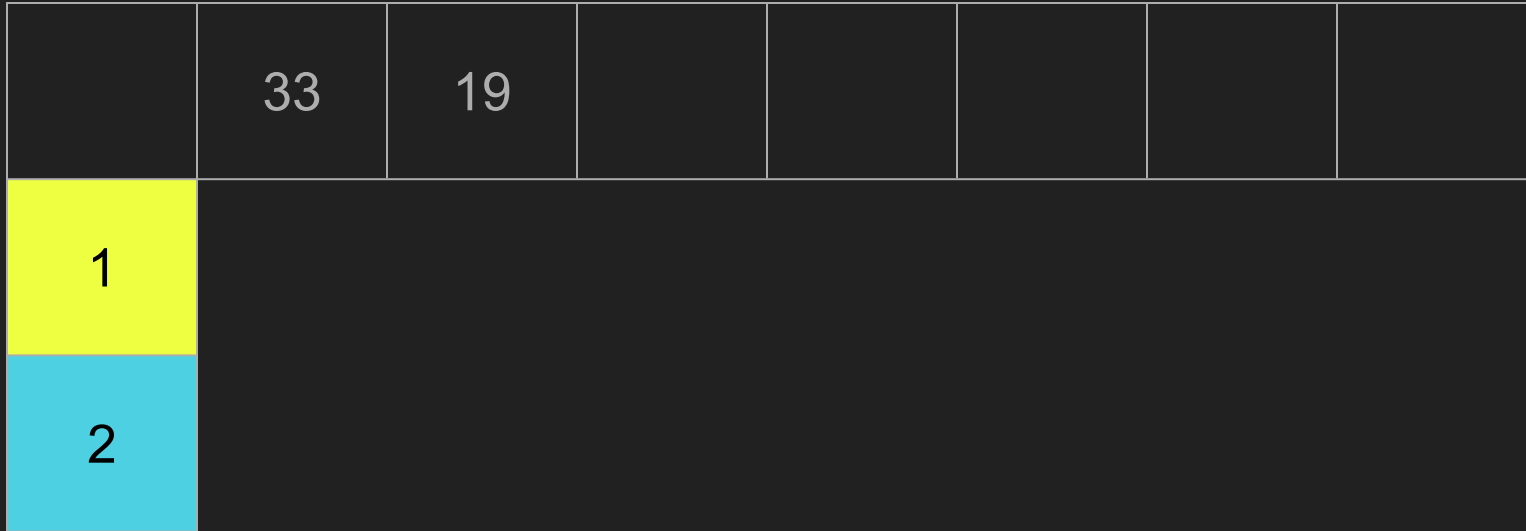
- Queue implemented as an array:

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



- Queue implemented as an array:

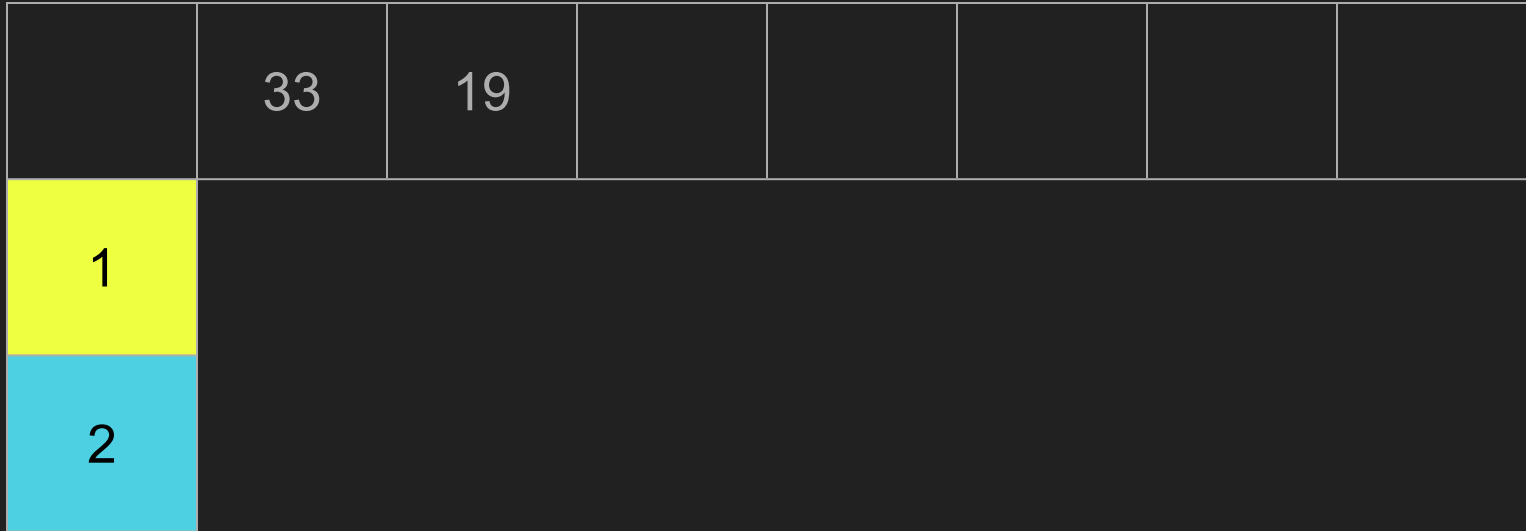
```
int x = dequeue(&q); // x gets 28
```





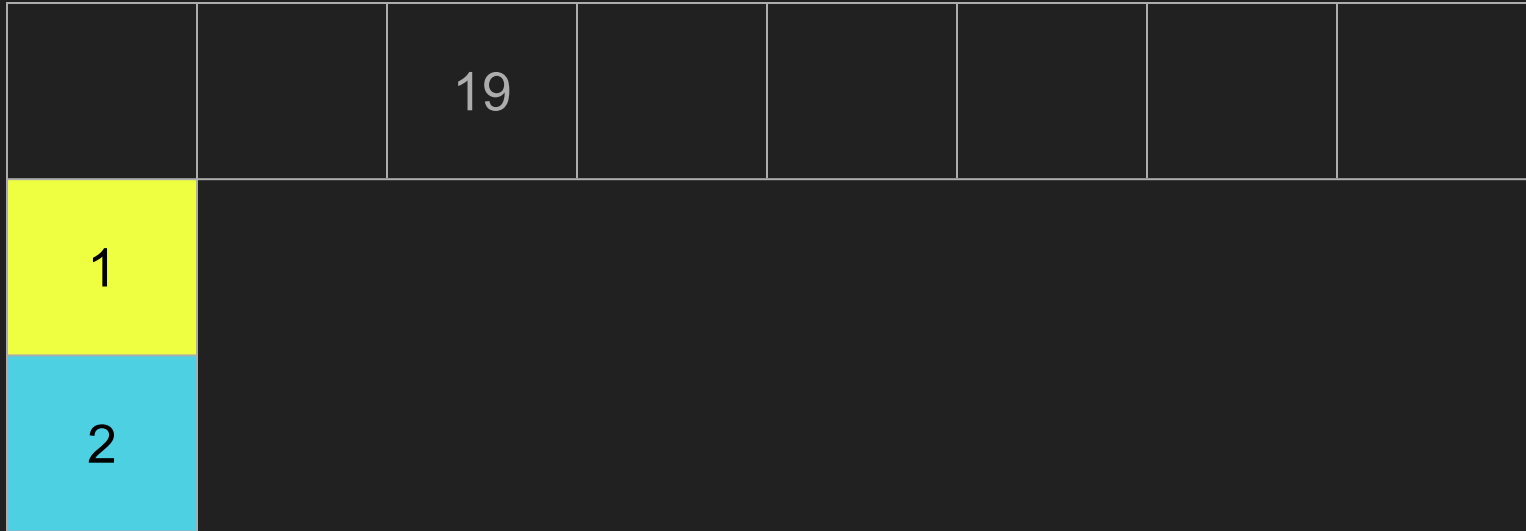
- Queue implemented as an array:

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



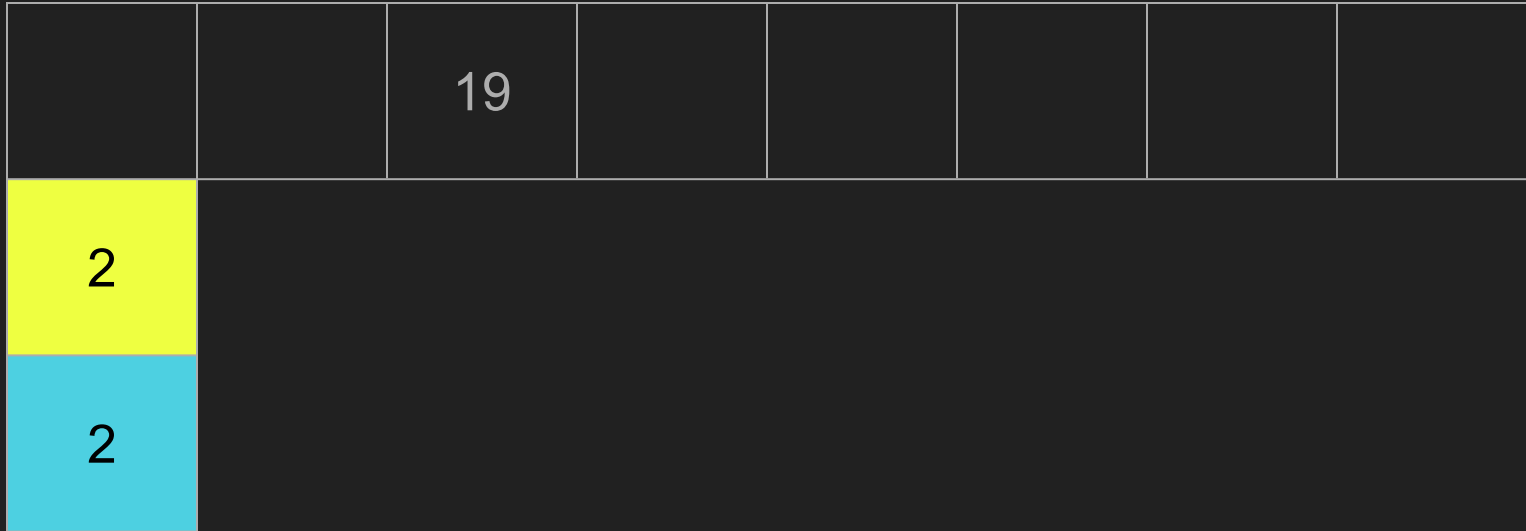
- Queue implemented as an array:

```
int x = dequeue(&q); // x gets 33
```



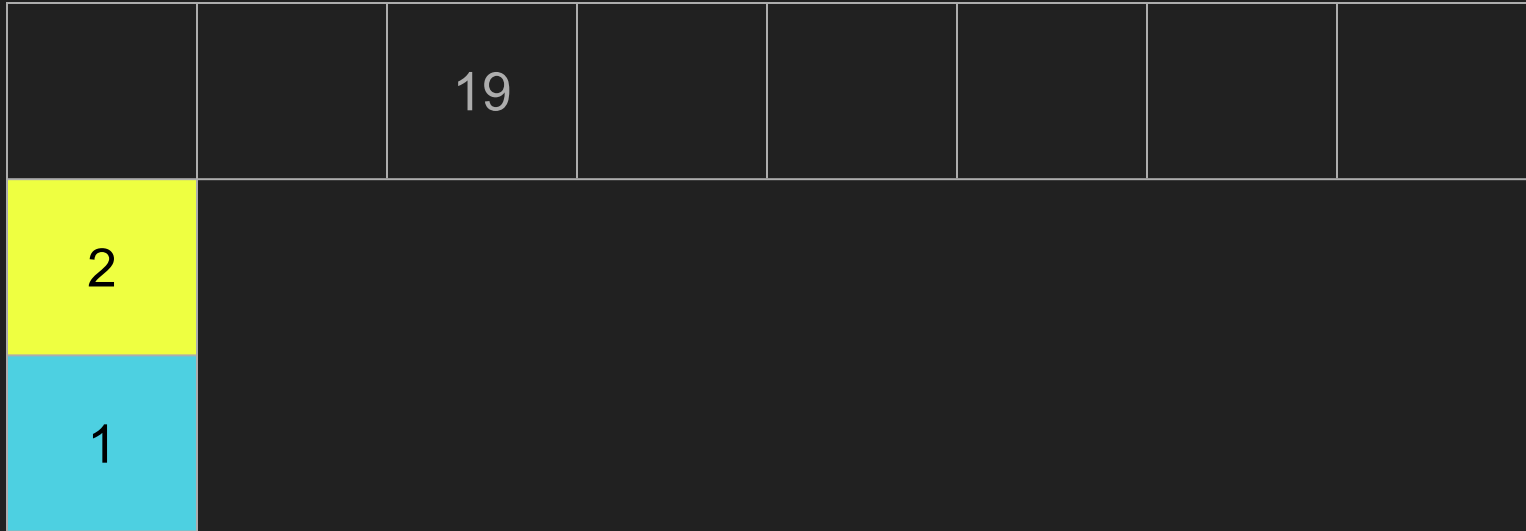
- Queue implemented as an array:

```
int x = dequeue(&q); // x gets 33
```



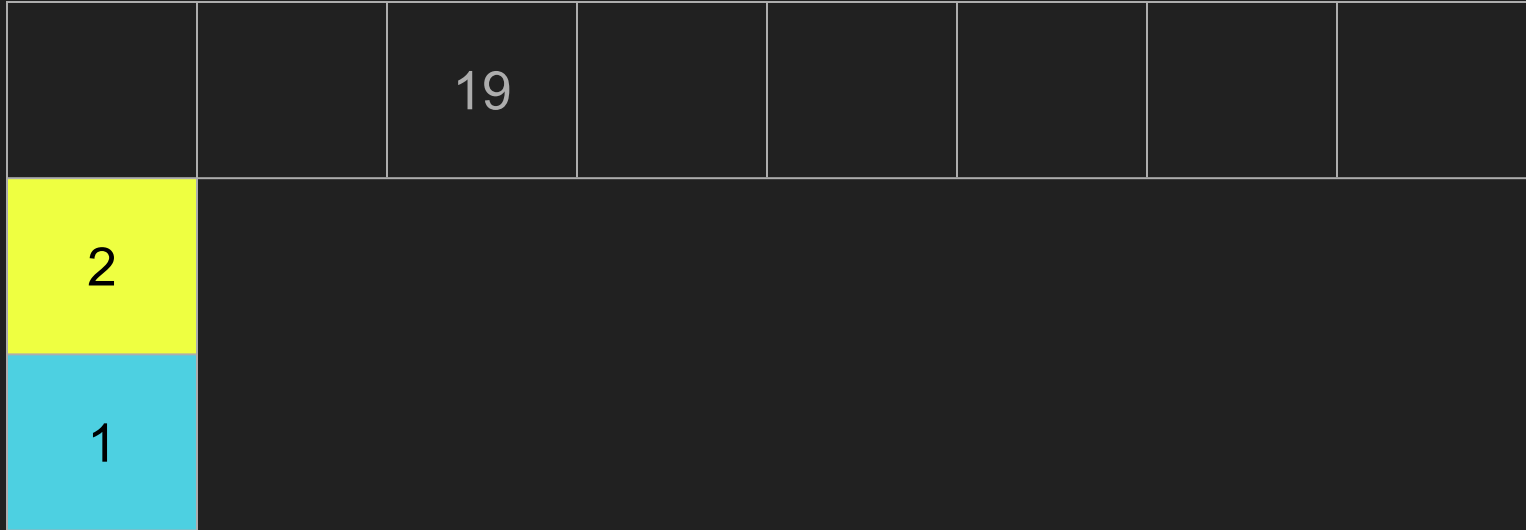
- Queue implemented as an array:

```
int x = dequeue(&q); // x gets 33
```



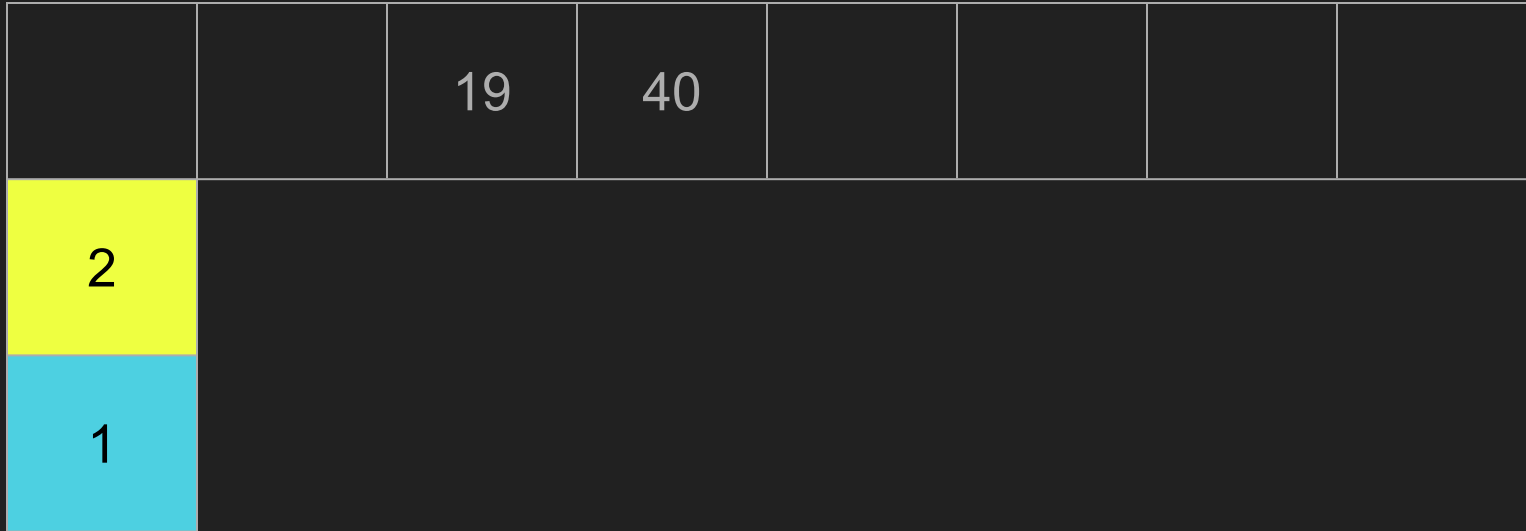
- Queue implemented as an array:

```
enqueue(&q, 40);
```



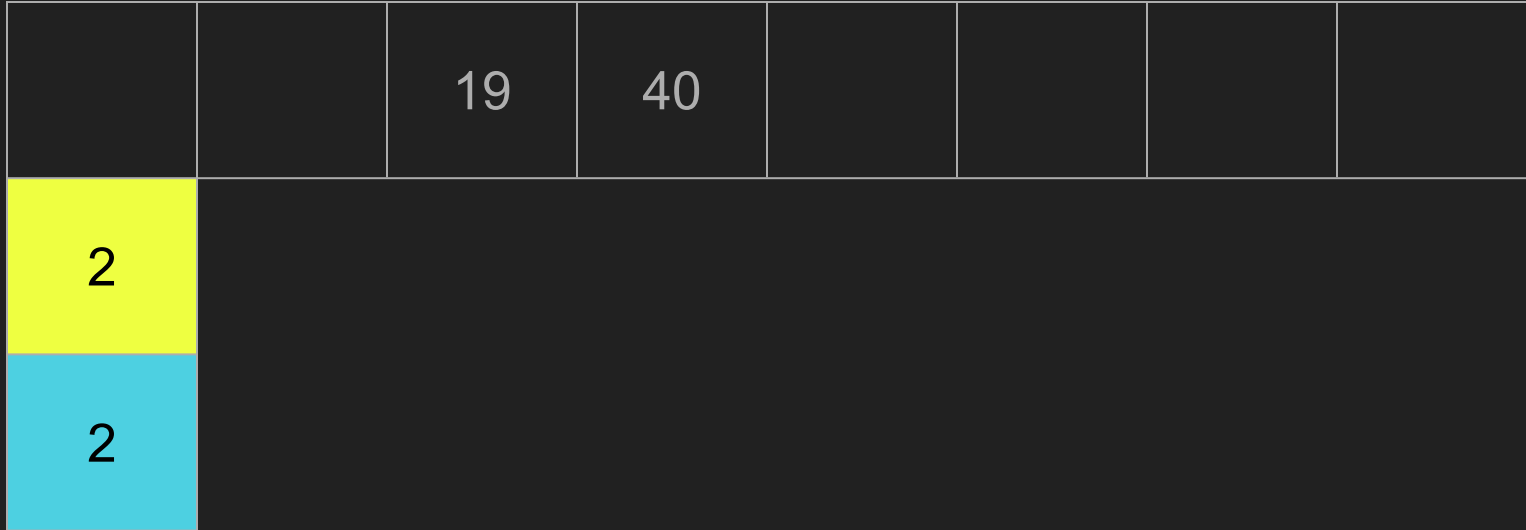
- Queue implemented as an array:

```
enqueue(&q, 40);
```



- Queue implemented as an array:

```
enqueue(&q, 40);
```



- Queue implemented as a **doubly** linked list:

```
typedef struct queue
{
    int value;
    struct queue *prev;
    struct queue *next;
}
queue;
```



- Enqueue:

- Now maintain pointers to both the head (front) and tail (back) of the linked list.
- Just like maintaining any other linked list, except now more pointers to rearrange!
- Make space for a new list node, populate it.
- Chain that node into the front (or end) of the linked list.
- Make the new head (or tail) of the linked list the node you just added.

- Dequeue:

- Extract the data from the node at the tail (or head) of the linked list.
- Adjust the tail (or head) to be one node prior (or forward) in the linked list.
- Free the node you just extracted data from.

# PROBLEM SET 5 PREVIEW

Due Sun 10/6 @ 11:59pm

You will need to implement:

- `Speller`



**NO SECTION NEXT WEEK: OCTOBER RECESS**

**Section Feedback:** <https://tinyurl.com/cs50rwfeedback>