CS50 Week 5

Connor Leggett
cjleggett@college.harvard.edu

Questions?

Logistics

- Pset due this Sunday
- No lecture Monday!
- No section Tuesday!
- Lecture Wednesday
- Supersection either Thursday or Friday

Grading Notes

Tips for Style and Design

- Never use "== true" or "== false"
- Don't overuse parentheses
- Don't return nothing at the end of a function
- Use i for loops unless it makes sense to do otherwise
- Use style50

Content Recap

Linked Lists

- Thus far, only arrays have provided us a means of representing a collection of *like* values.

- They are great for element lookup, but pretty terrible for inserting unless we happen to be tacking on to the end of an array.

- Arrays are also quite inflexible; what happens if we need a larger array than we thought? With clever use of structs and pointers - maybe there’s a fix!

- We refer to this combination of structs and pointers, when used to create a “chain” of nodes a **linked list**.

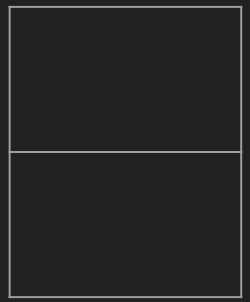
- A linked list **node** is a special type of struct with two fields:
  - Data of some type
  - A pointer to another linked list node.

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

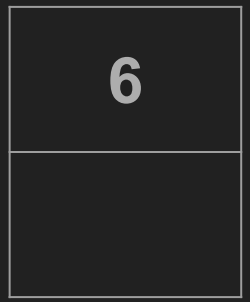
- In order to work with linked lists effectively, there are five key operations to understand (only the first four of which are really needed in this course):
  - Creating a linked list when it doesn’t exist.
  - Searching through a linked list to find an element.
  - Inserting a new node into a linked list.
  - Deleting an entire linked list.
  - Deleting a single element from a linked list.

- Create a linked list:
  - Dynamically allocate space for a new (your first!) node.
  - Check to make sure you didn’t run out of memory.
  - Initialize the value field.
  - Initialize the next field (specifically, to NULL).
  - Return a pointer to your newly created node.

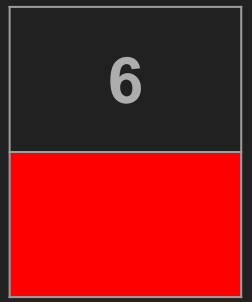
- Create a linked list:
  - Dynamically allocate space for a new (your first!) node.
  - Check to make sure you didn’t run out of memory.
  - Initialize the value field.
  - Initialize the next field (specifically, to NULL).
  - Return a pointer to your newly created node.



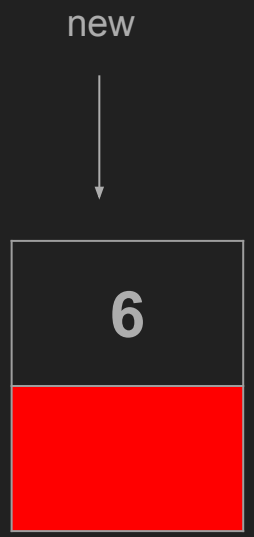
- Create a linked list:
  - Dynamically allocate space for a new (your first!) node.
  - Check to make sure you didn’t run out of memory.
  - Initialize the value field.
  - Initialize the next field (specifically, to NULL).
  - Return a pointer to your newly created node.



- Create a linked list:
  - Dynamically allocate space for a new (your first!) node.
  - Check to make sure you didn’t run out of memory.
  - Initialize the value field.
  - Initialize the next field (specifically, to NULL).
  - Return a pointer to your newly created node.



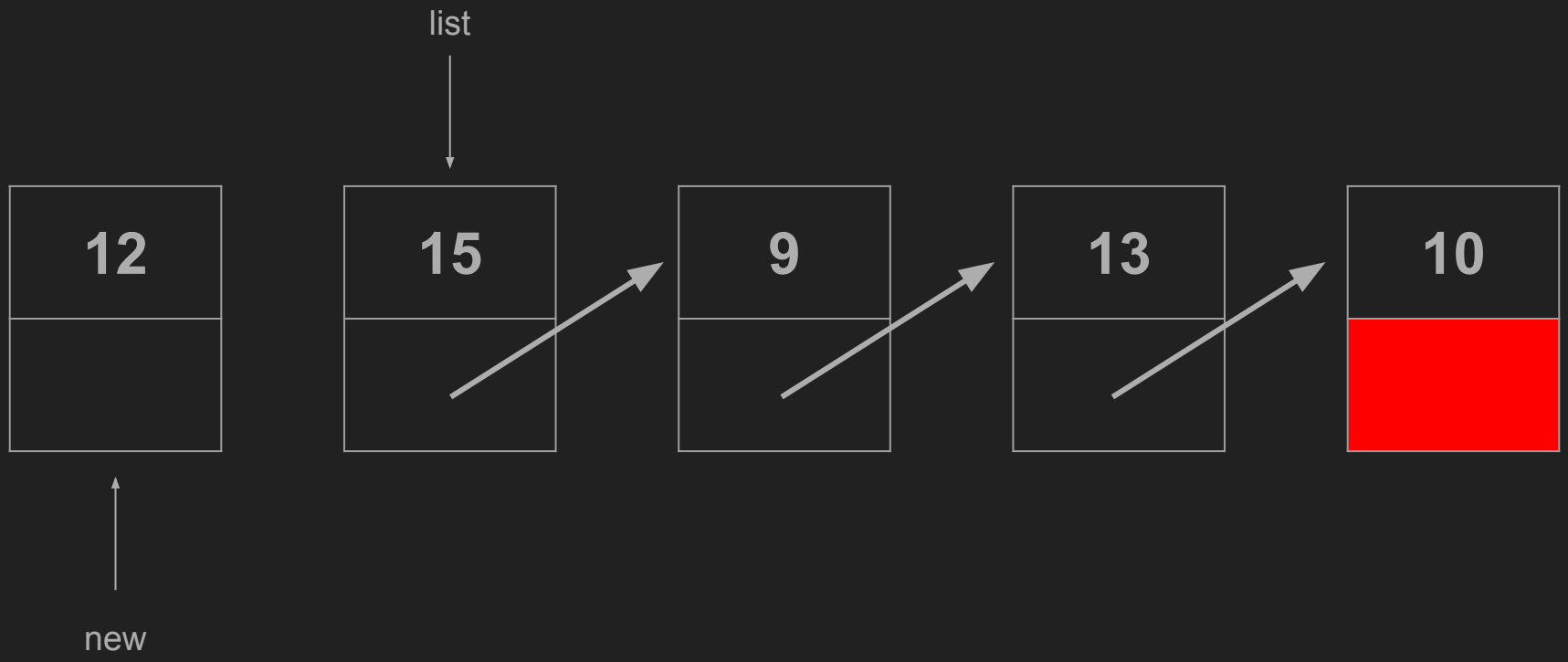
- Create a linked list:
  - Dynamically allocate space for a new (your first!) node.
  - Check to make sure you didn’t run out of memory.
  - Initialize the value field.
  - Initialize the next field (specifically, to NULL).
  - Return a pointer to your newly created node.

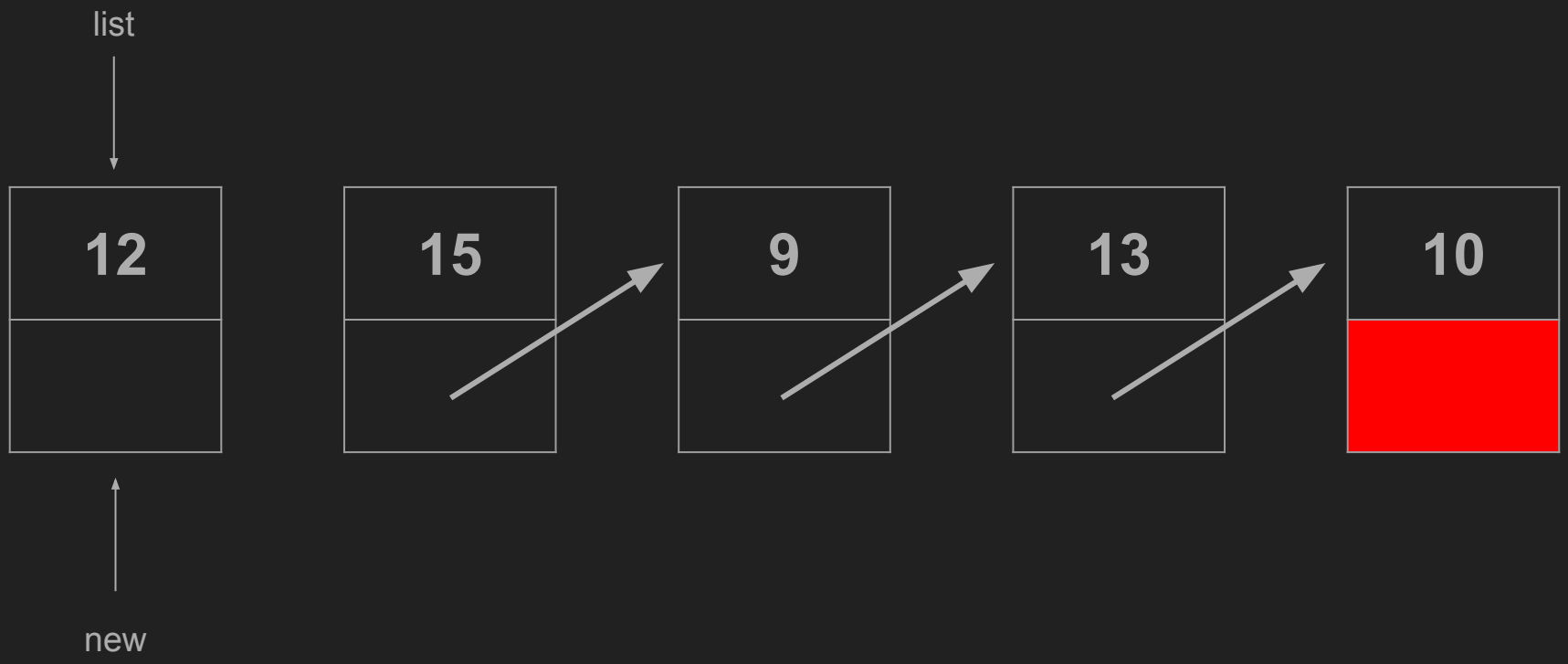


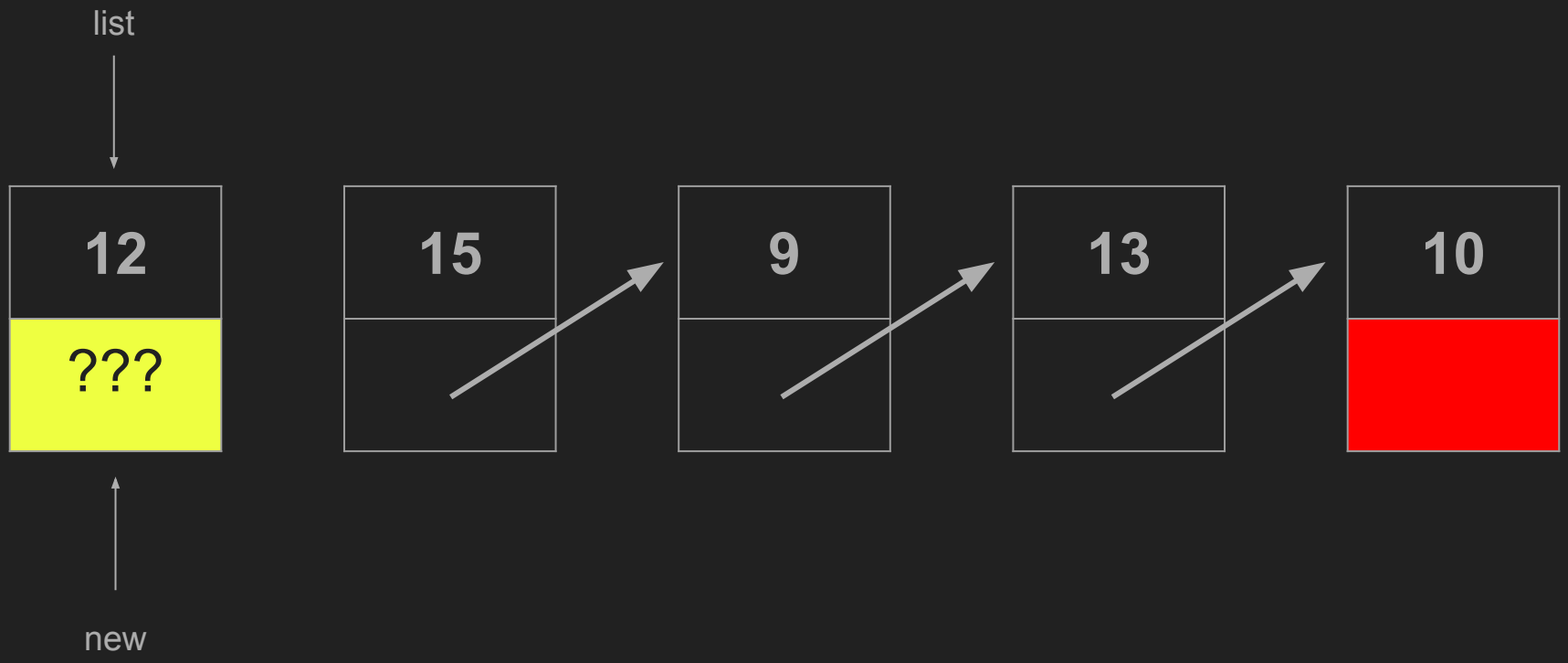
- Find an element:
  - Create a traversal pointer pointing to the list’s head (first element).
  - If the current node’s value field is what we’re looking for, return true.
  - If not, set the traversal pointer to the next pointer in the list and go back to the previous step.
  - If you’ve reached the element of the list, return false.

- Insert an element:
  - Dynamically allocate space for a new linked list node.
  - Check to make sure we didn’t run out of memory.
  - Populate and insert the node at the beginning of the linked list.
  - Return a pointer to the new head of the linked list.

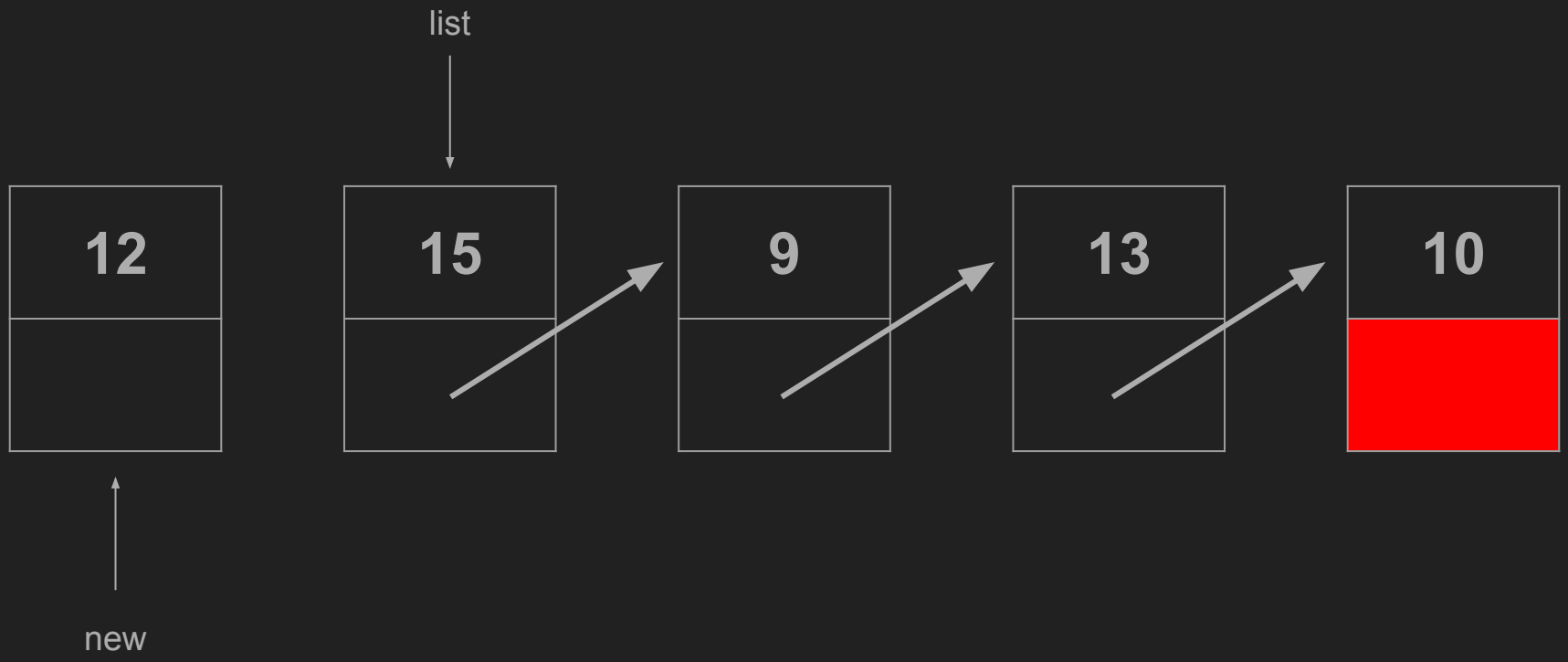
- Insert an element:
  - Dynamically allocate space for a new linked list node.
  - Check to make sure we didn’t run out of memory.
  - Populate and insert the node at the beginning of the linked list.
    - So which pointer do we move first? The pointer in the newly created node, or the pointer pointing to the *original* head of the linked list?
    - This choice matters!
  - Return a pointer to the new head of the linked list.

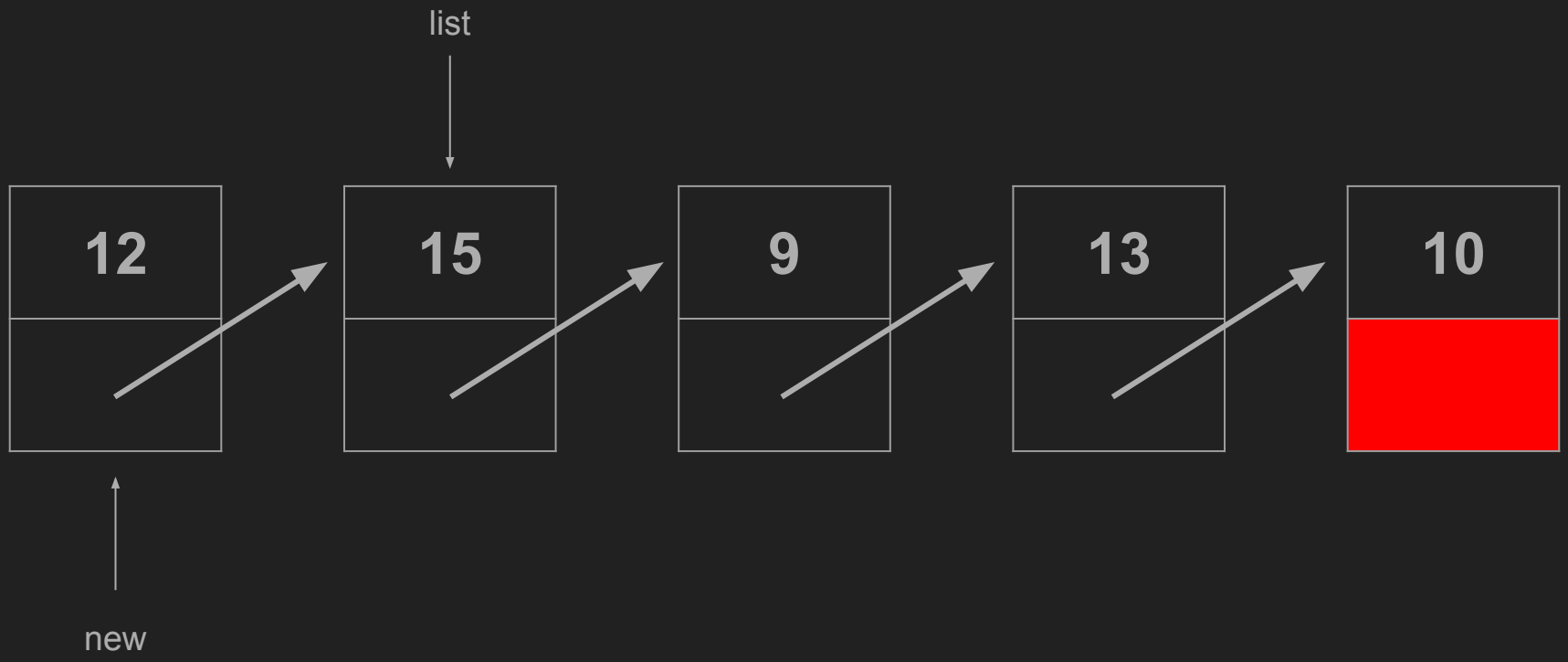

list
12
15
9
13
10
new

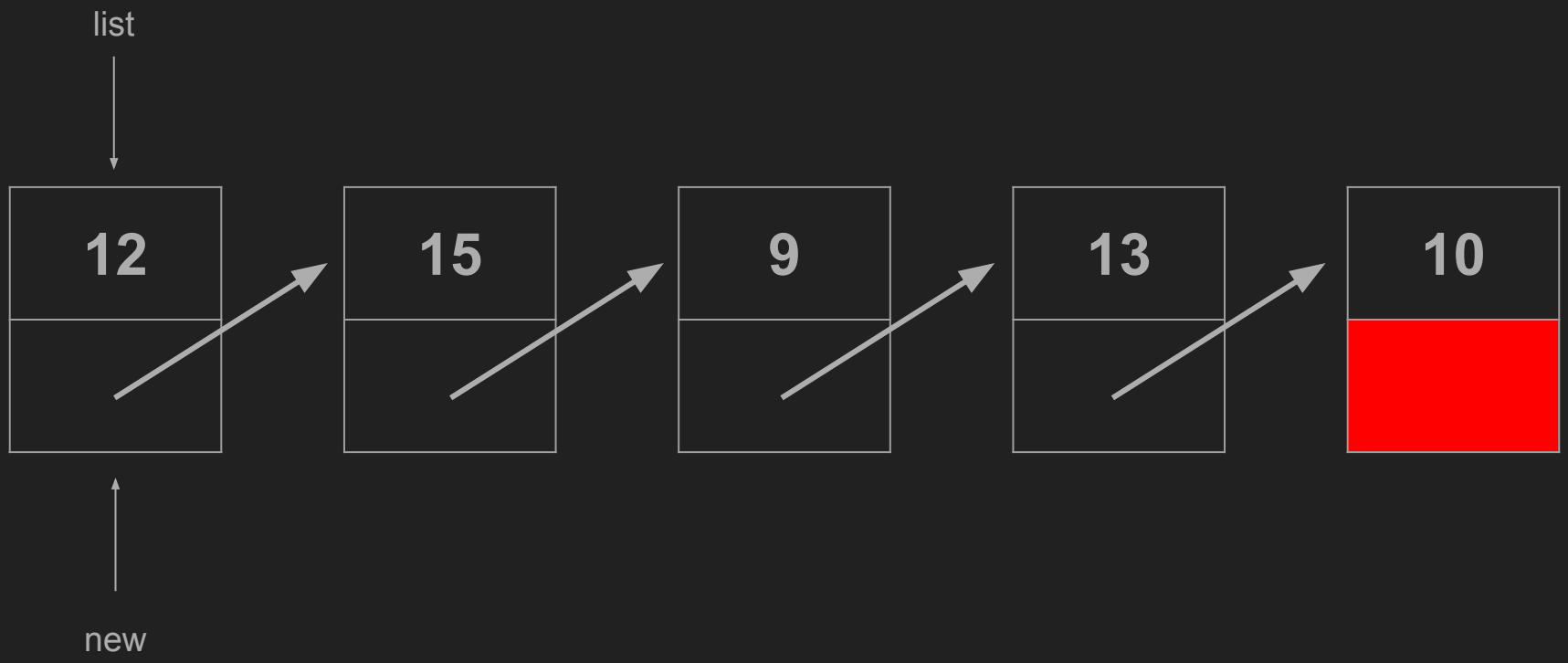

list
12
15
9
13
10
new


list
12
???
new
15
9
13
10

X


list
12
15
9
13
10
new


list
12
15
9
13
10
new


list
12
15
9
13
10
new

- Delete an entire linked list:
  - If you've reached a NULL pointer, stop.
  - Delete the rest of the list.
  - Free the current node.

- Sounds like recursion!

Hash Tables

- Hash tables combine the random access of an array with the dynamism of a linked list.

- This means insertion deletion, and lookup can all tend toward O(1)! We're gaining the advantages of both, and mitigating the disadvantages.

- To get this performance upgrade, the data structure will require that when we insert data into the structure, the data itself gives us a clue about where to find it.

- One tradeoff: Hash tables are bad for ordering and sorting data.

- Hash tables combine two things with which we're familiar:
  - First, a **hash function**, which typically returns a nonnegative integer value called a *hash code*.
  - Second, an **array** capable of storing data of the type we wish to place into the data structure.

- The basic idea is that we run our data through the hash function, which returns a number, and we store the data in the array at that index location.

- How to define a hash function? No limit to the possible ways.

- A good hash function should:
  - 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.

0	
1	
2	
3	
4	“John”
5	
6	“Paul”
7	
8	
9	

hash("John") returns 4

hash("Paul") returns 6

hash(“Ringo”) returns 6

- A **collision** occurs when two pieces of data, run through the hash function, yield the same hash code.

- Presumably we still want to store both pieces of data, so we shouldn’t simply clobber what was there first.

- We need to find a way to get both elements into the hash table, while trying to preserve quick insertion and lookup.

- 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.

- Subject to a problem called **clustering**. Once there’s a miss, two adjacent cells now contain data, and the likelihood of the cluster grows.

- Even worse, the amount of data we can store is now capped at the size of the array.

- 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.

- Insertion and deletion into a linked list is O(1).

- Search is still *technically* O(n), but here's a case where constants matter. If our array is of size *k*, it's really O(n/k), which is probably closer to O(1)!

Tries

Tries

- In a trie, our data structure is a roadmap that allows us to look up elements in O(1) time.
- Stored as an array of arrays of arrays of…
- Really weird to think about, so let's visualize!
  - There are only 2 letters: "x" and "y"
  - I can make book titles from these letters, but these titles can only be 1, 2, or 3 letters long.
  - I own the books "xy", "y", and "yy"

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).

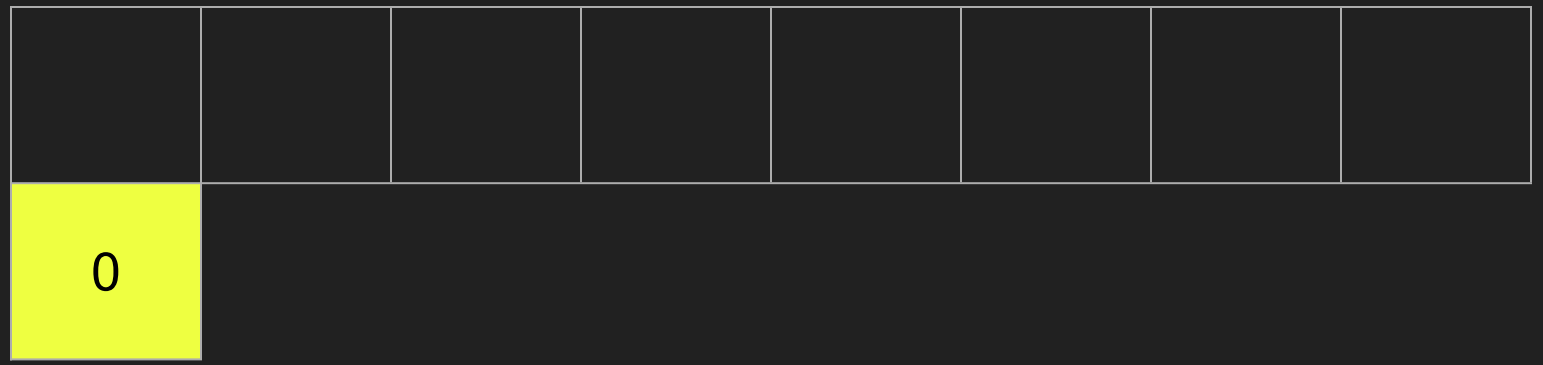
- Only two operations may be performed on stacks:
  - Push: Add a new element to the top of the stack.
  - Pop: Remove the most recently added element from the top of the stack.

- Stack implemented as an array:

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

- Stack implemented as an array:

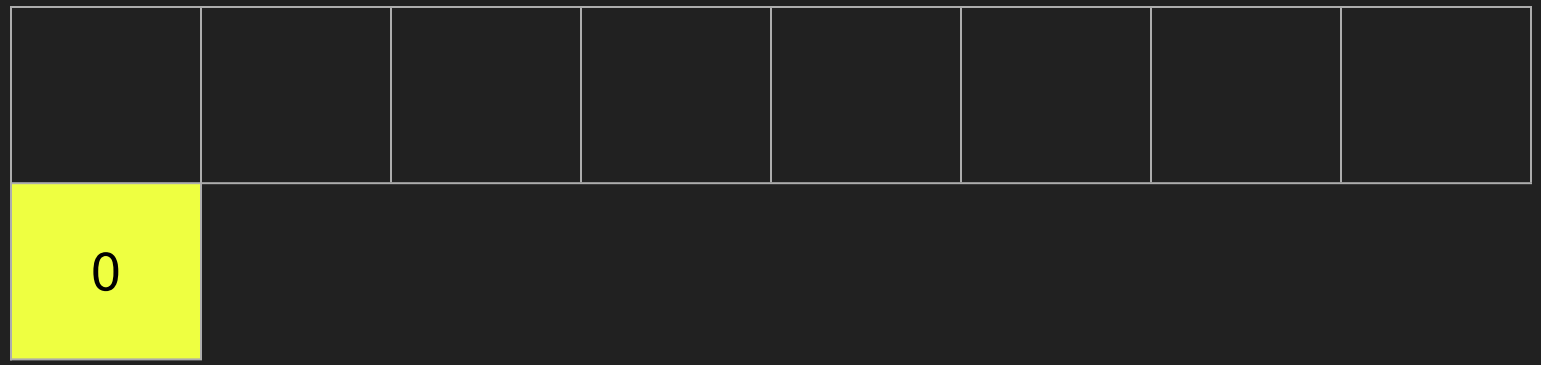
```
stack s;
s.top = 0;
```



- 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.)

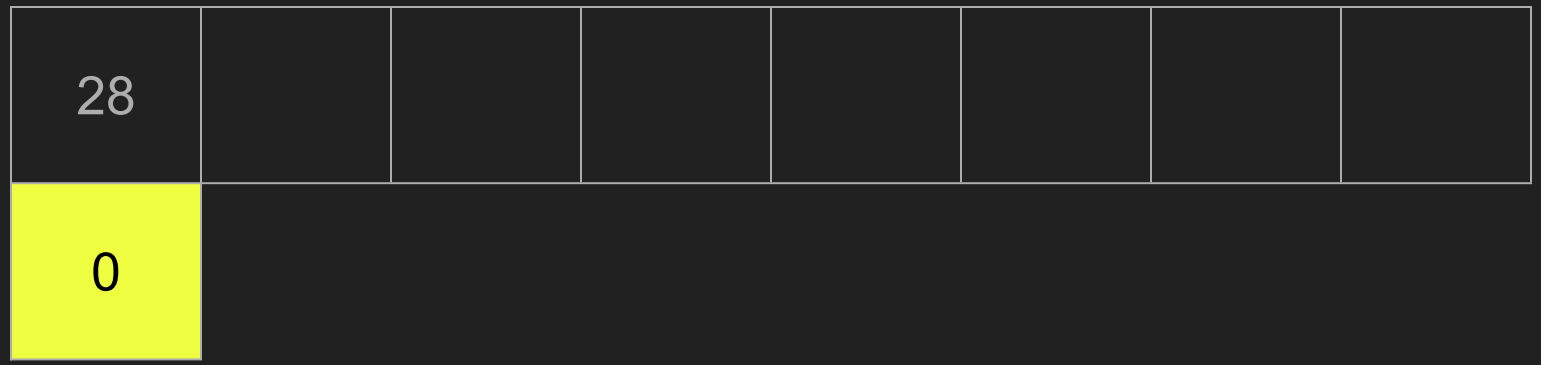
- Stack implemented as an array:

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



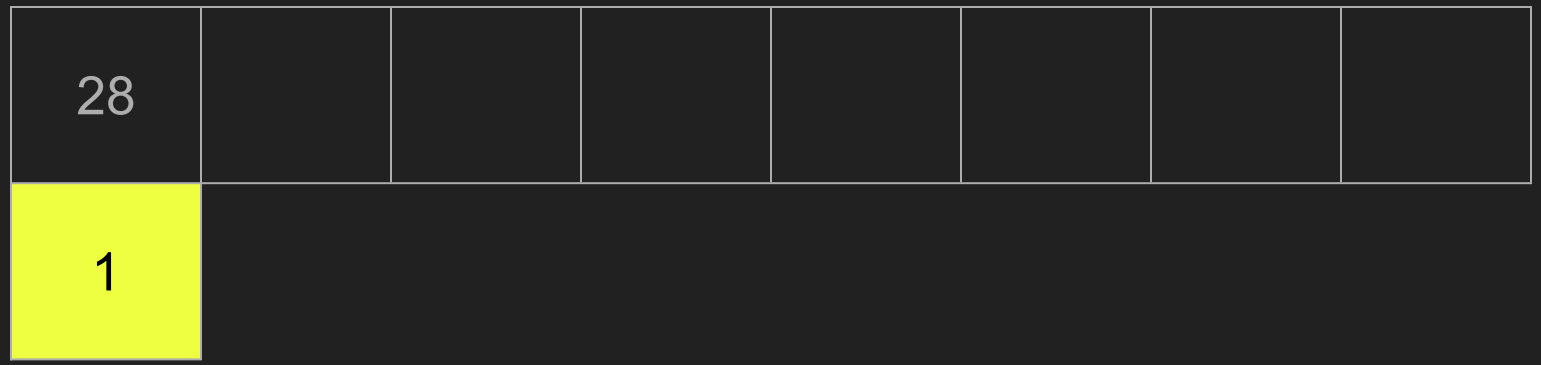
- Stack implemented as an array:

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



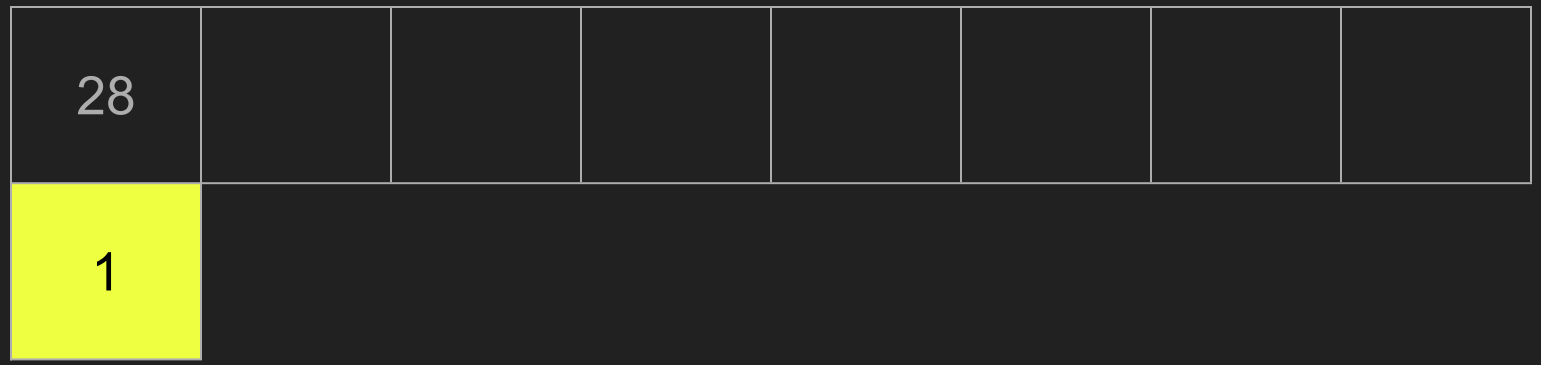
- Stack implemented as an array:

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



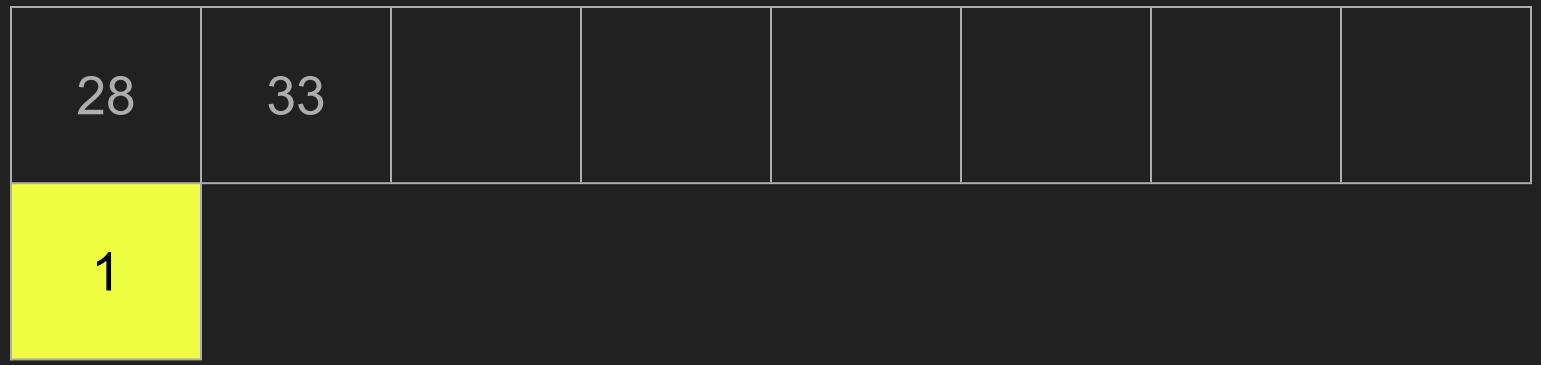
- Stack implemented as an array:

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



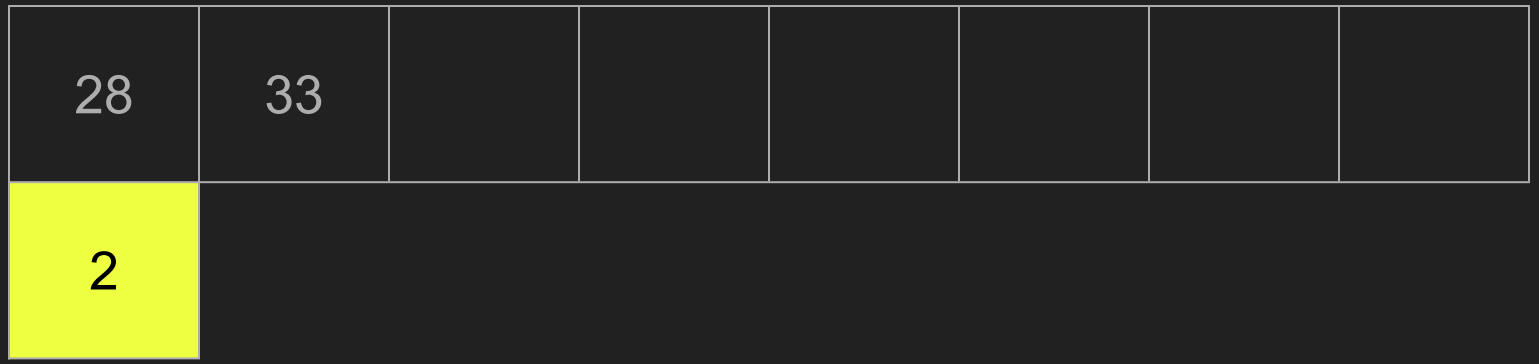
- Stack implemented as an array:

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



- Stack implemented as an array:

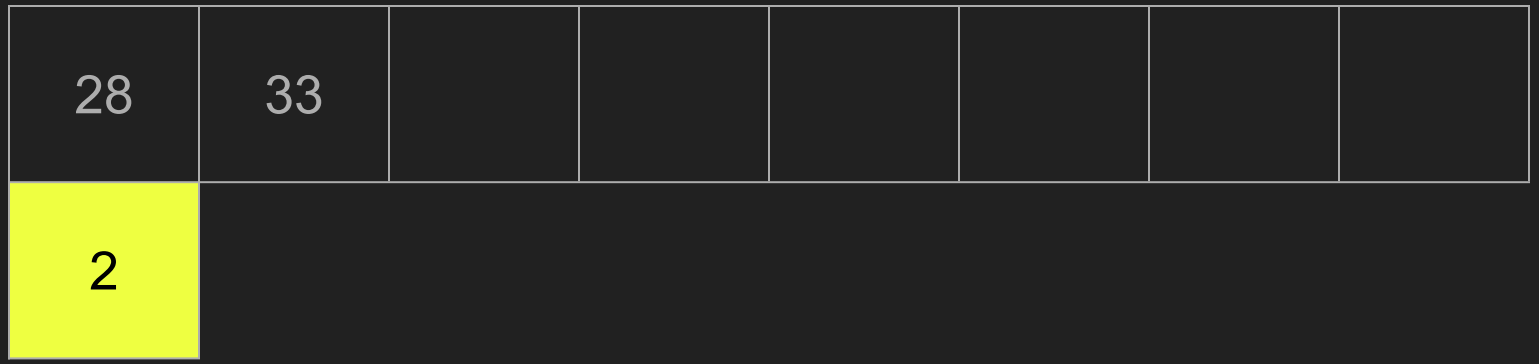
```
push(&s, 33);
```



- 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.

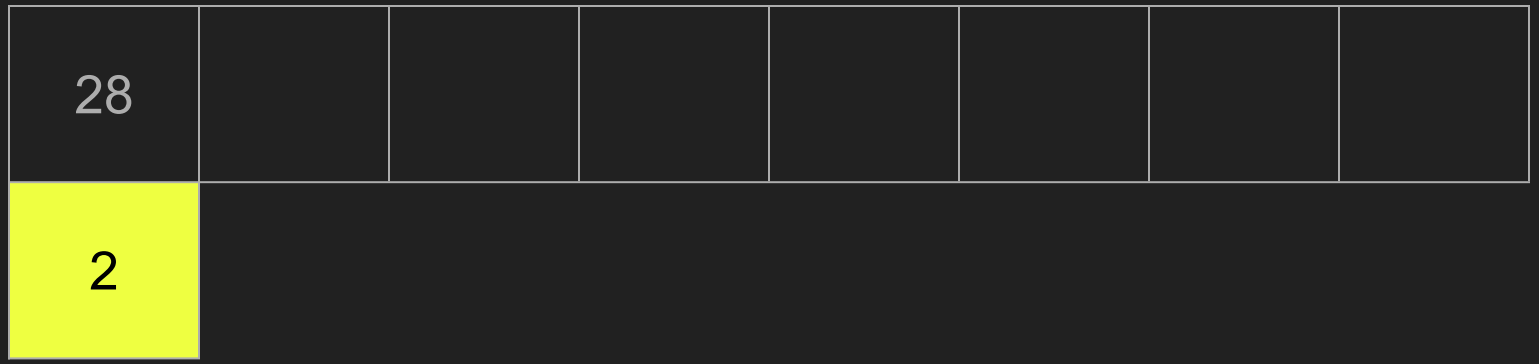
- Stack implemented as an array:

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



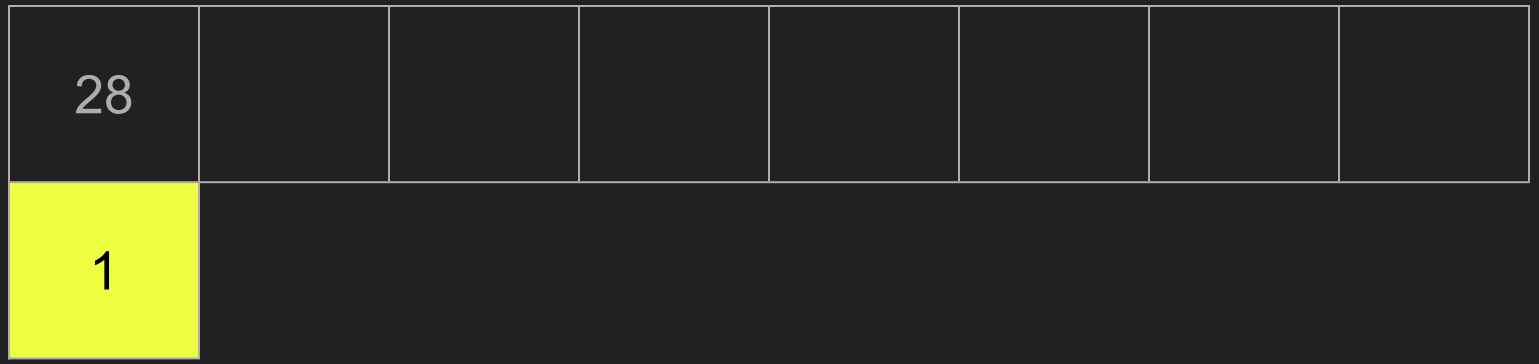
- Stack implemented as an array:

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



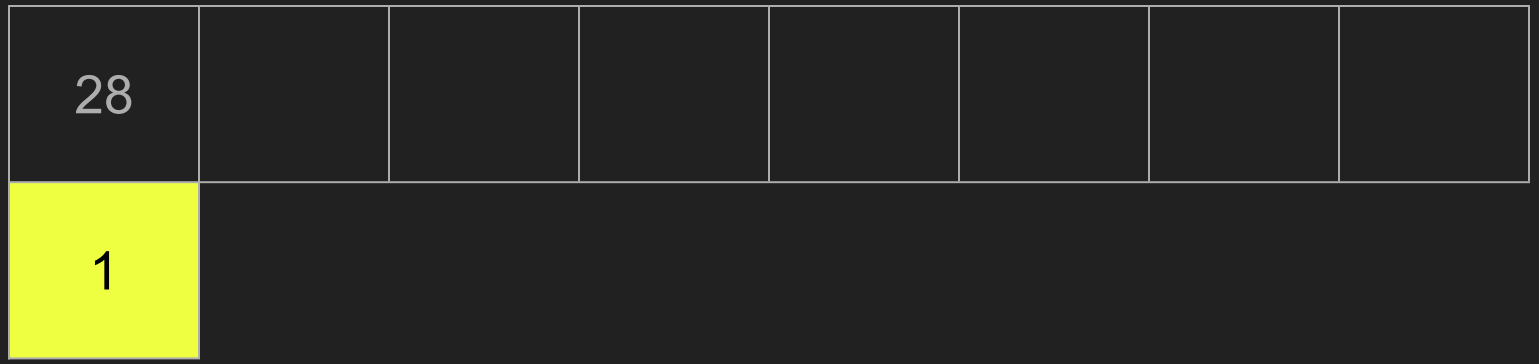
- Stack implemented as an array:

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



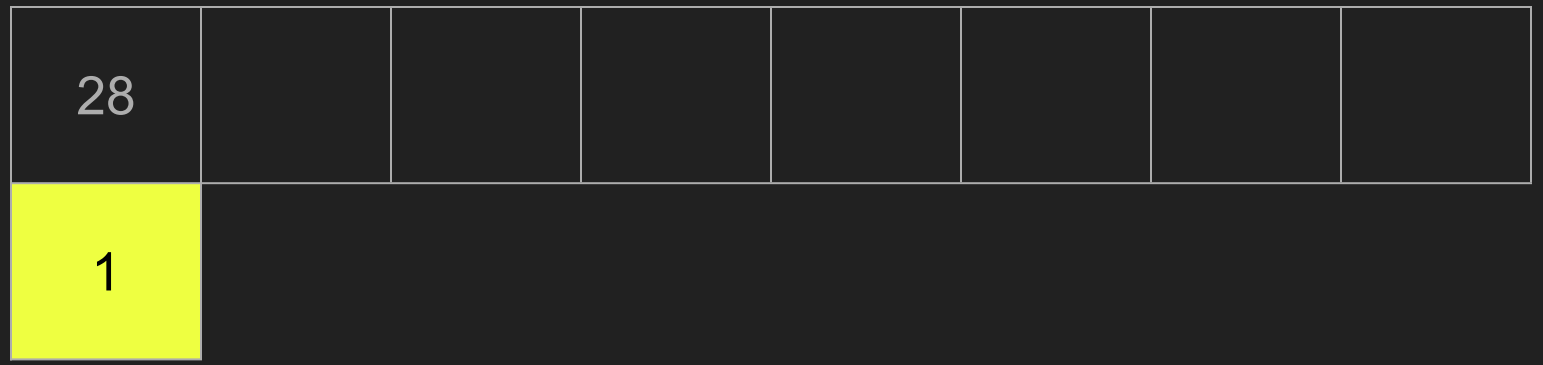
- Stack implemented as an array:

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



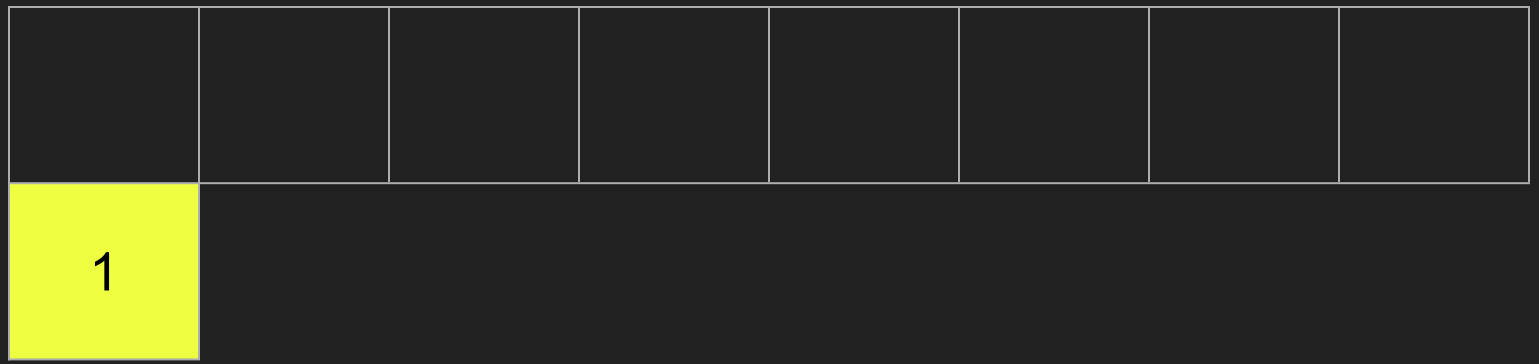
- Stack implemented as an array:

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



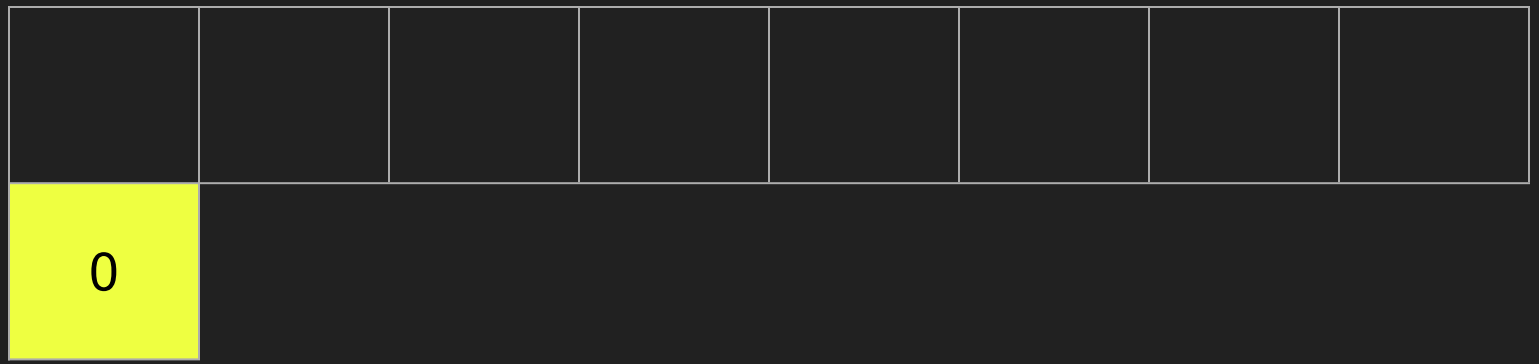
- Stack implemented as an array:

```
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).

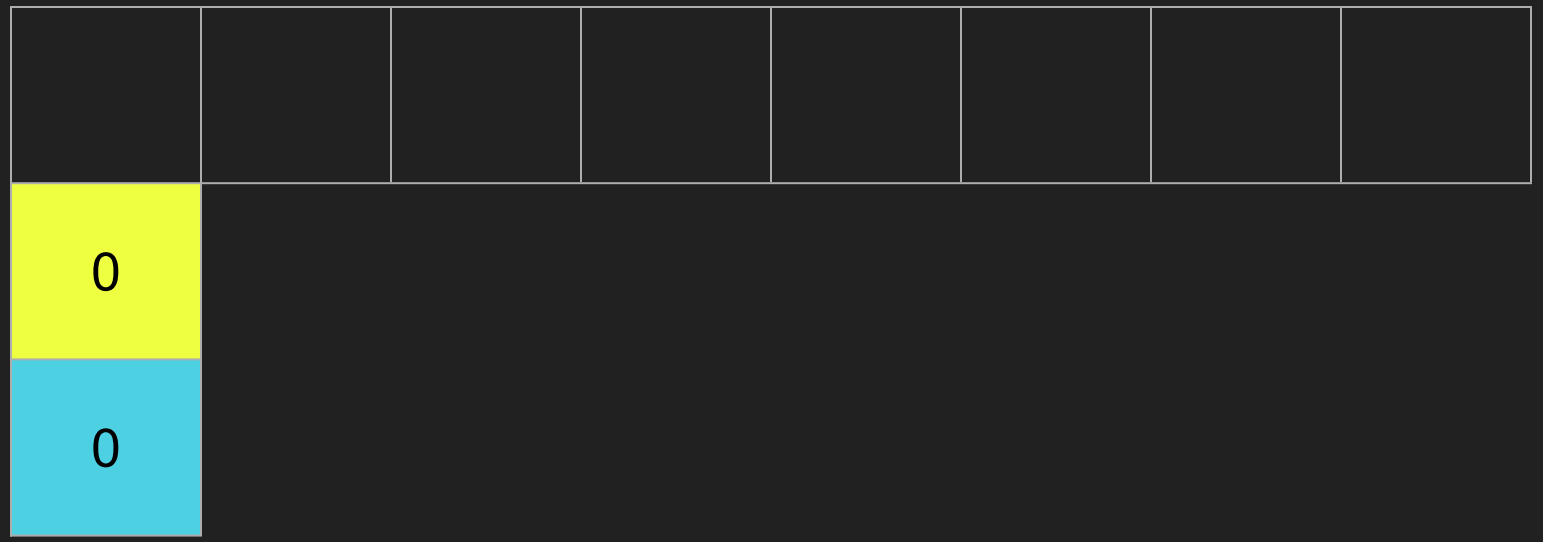
- Only two operations may be performed on queues:
  - Enqueue: Add a new element to the end of the queue.
  - Dequeue: Remove the oldest element from the front of the queue.

- Queue implemented as an array:

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

- Queue implemented as an array:

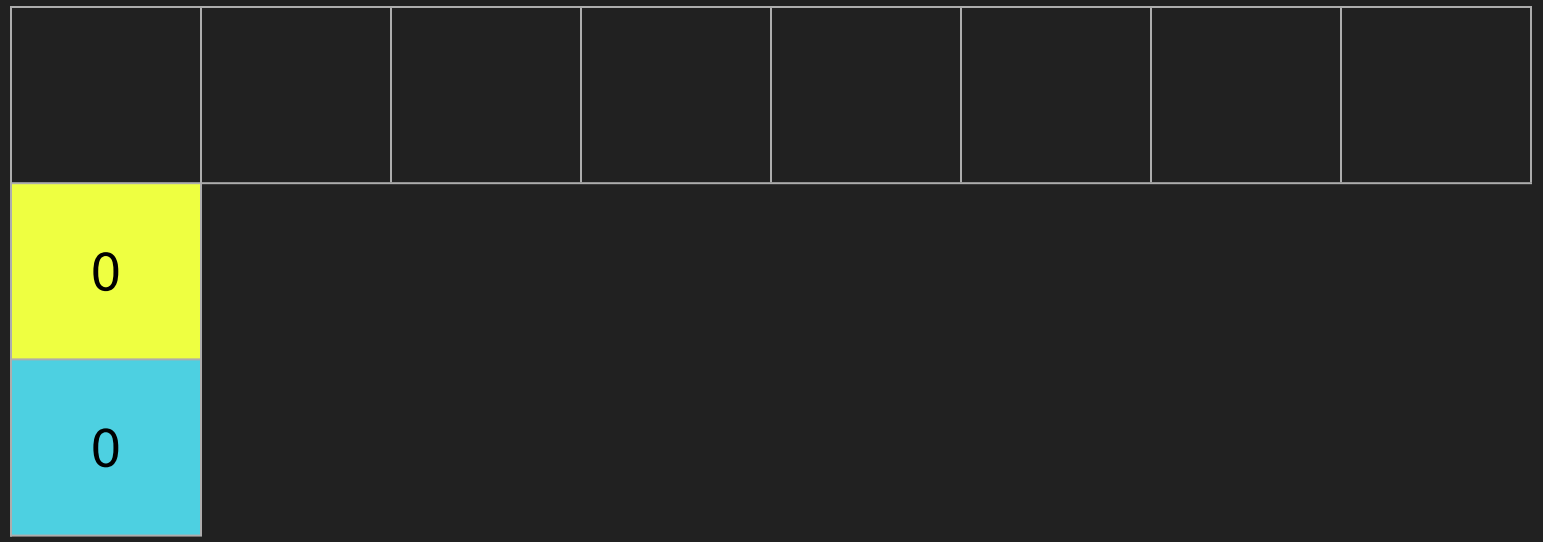
```
queue q;
q.front = 0;
q.size = 0;
```



- 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.)

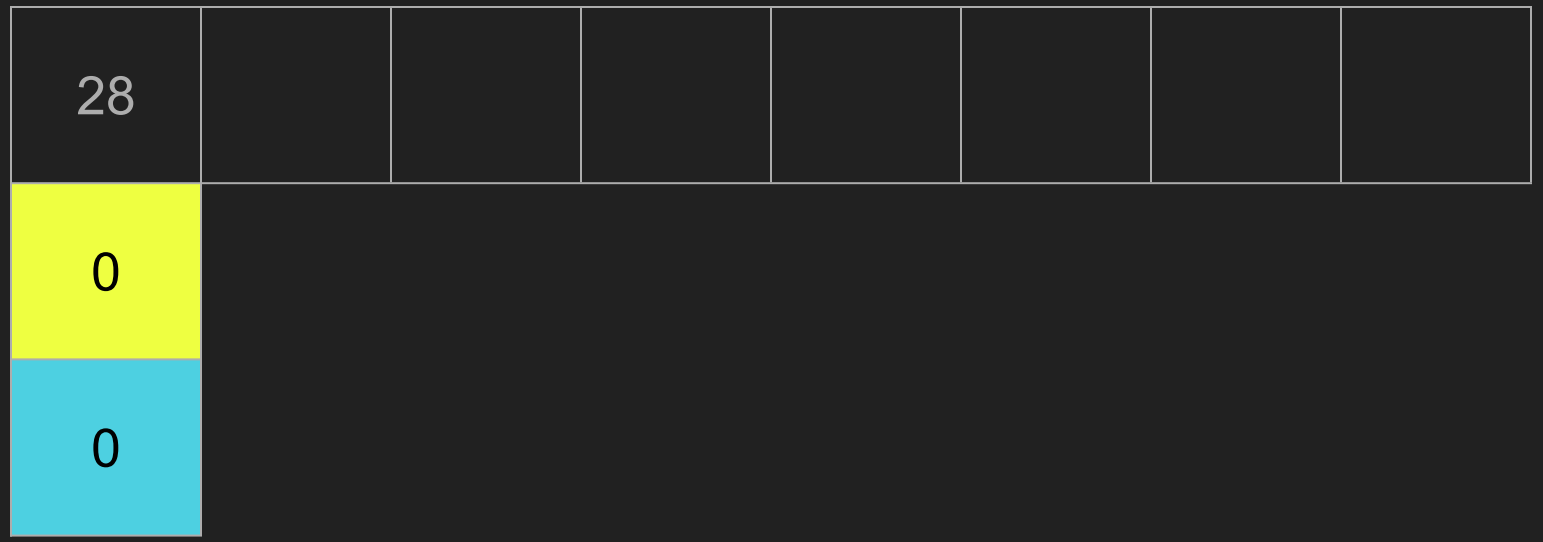
- Queue implemented as an array:

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



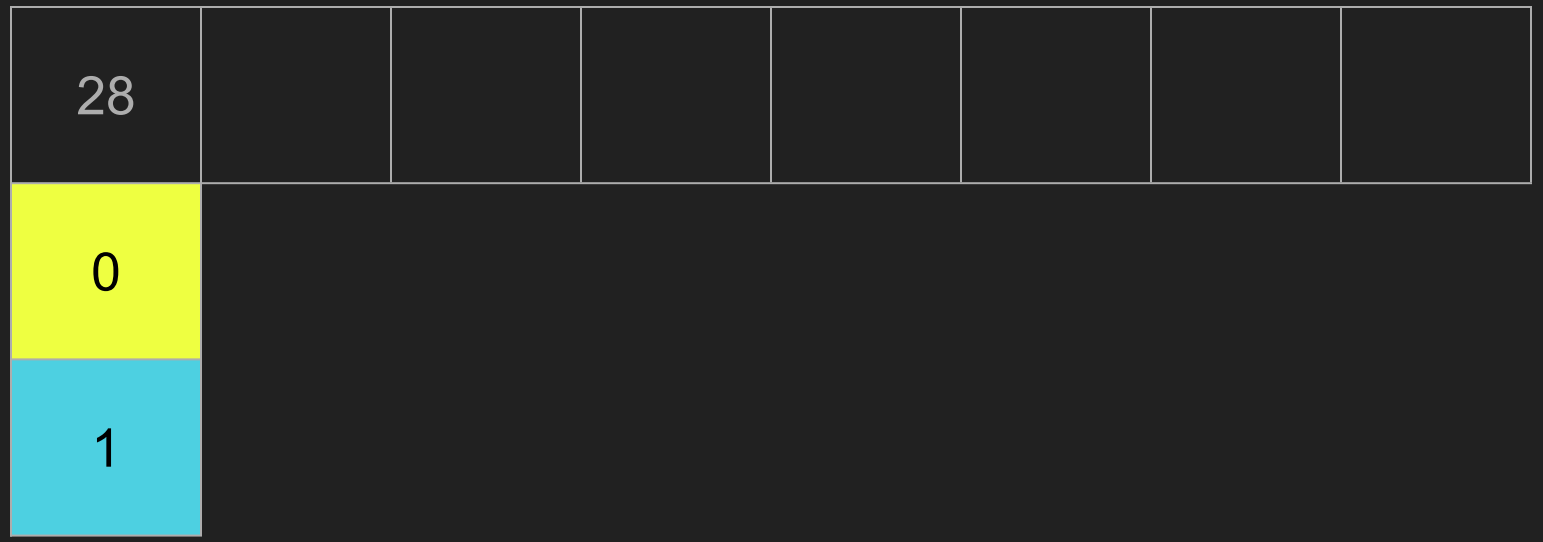
- Queue implemented as an array:

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



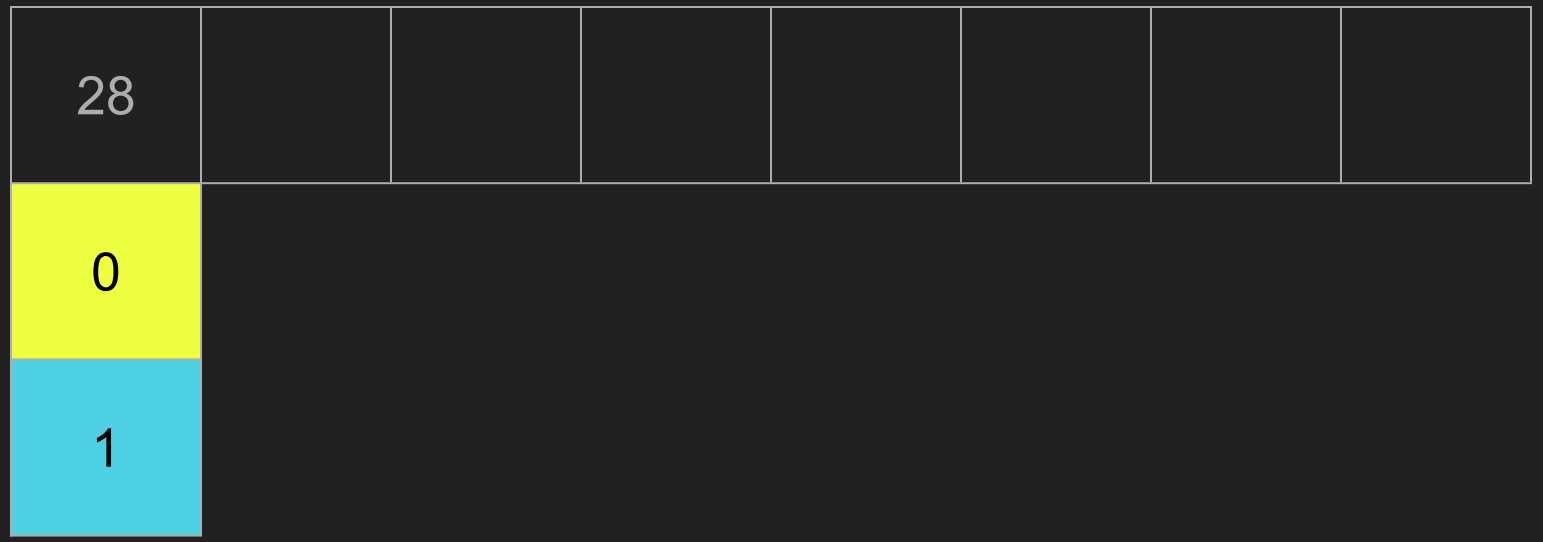
- Queue implemented as an array:

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



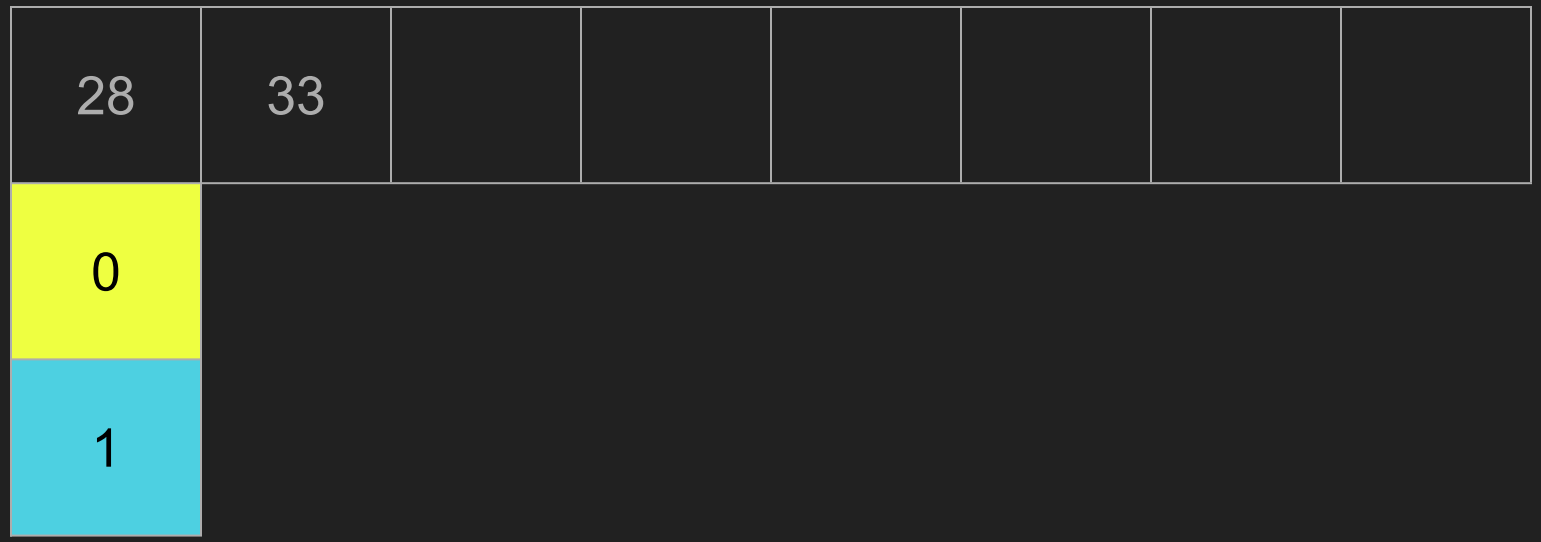
- Queue implemented as an array:

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



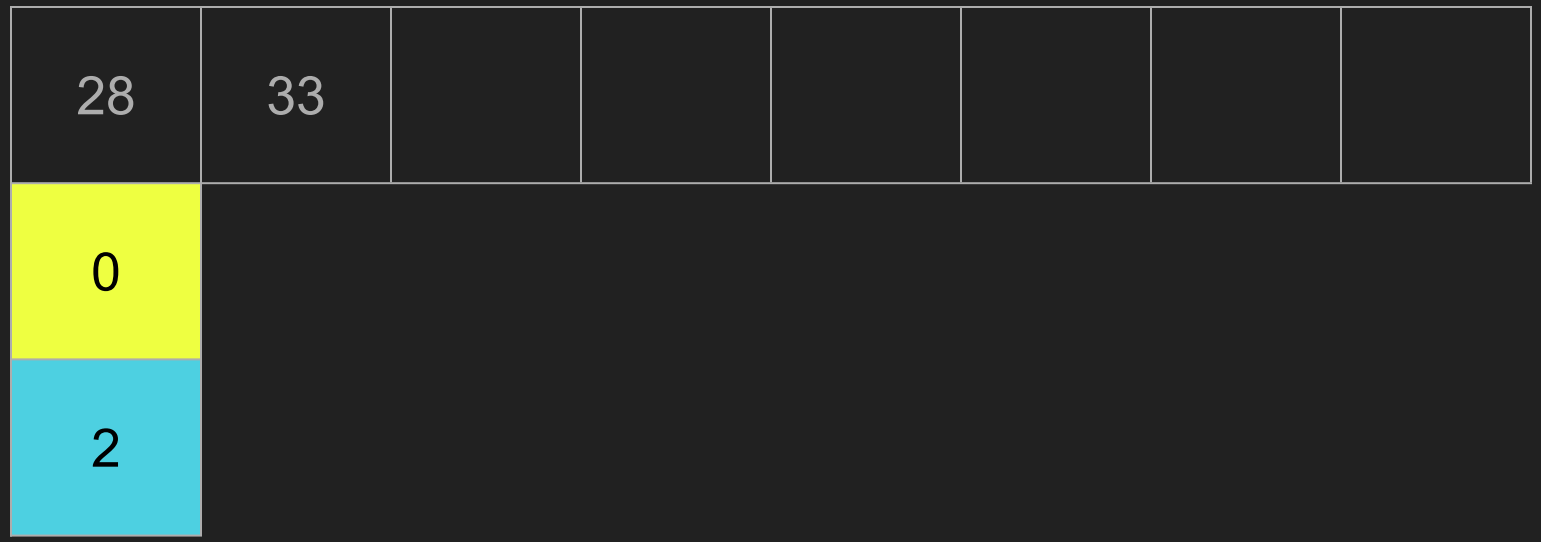
- Queue implemented as an array:

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



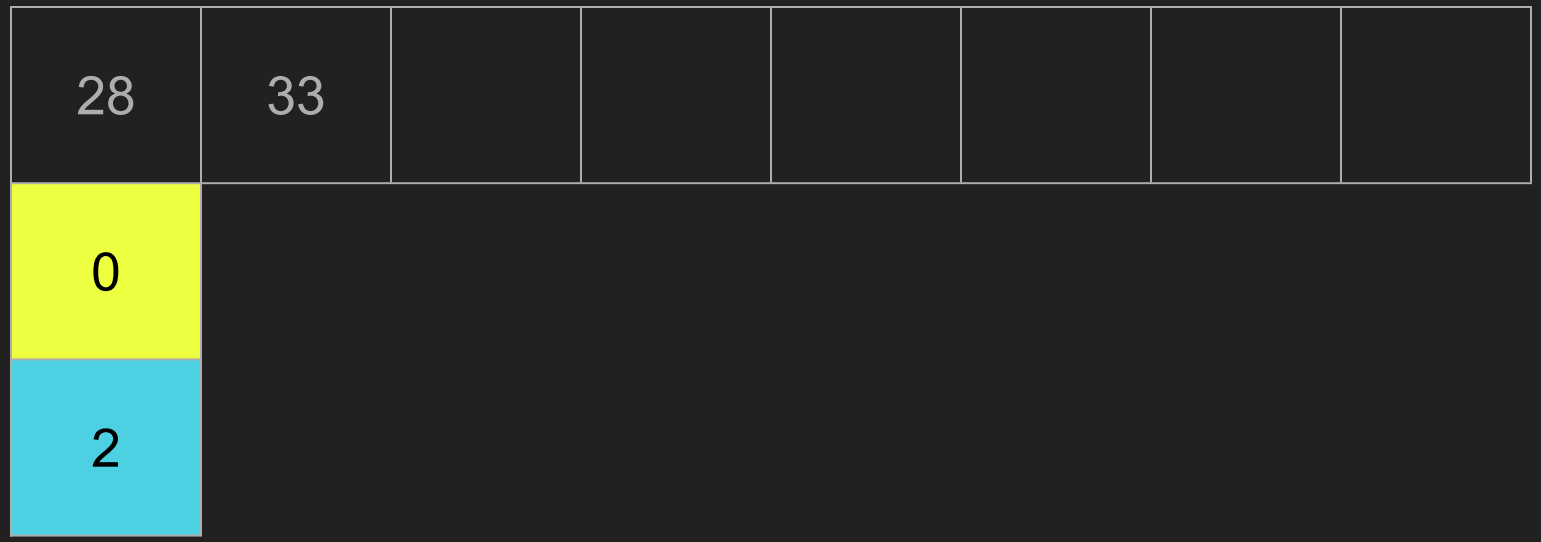
- Queue implemented as an array:

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



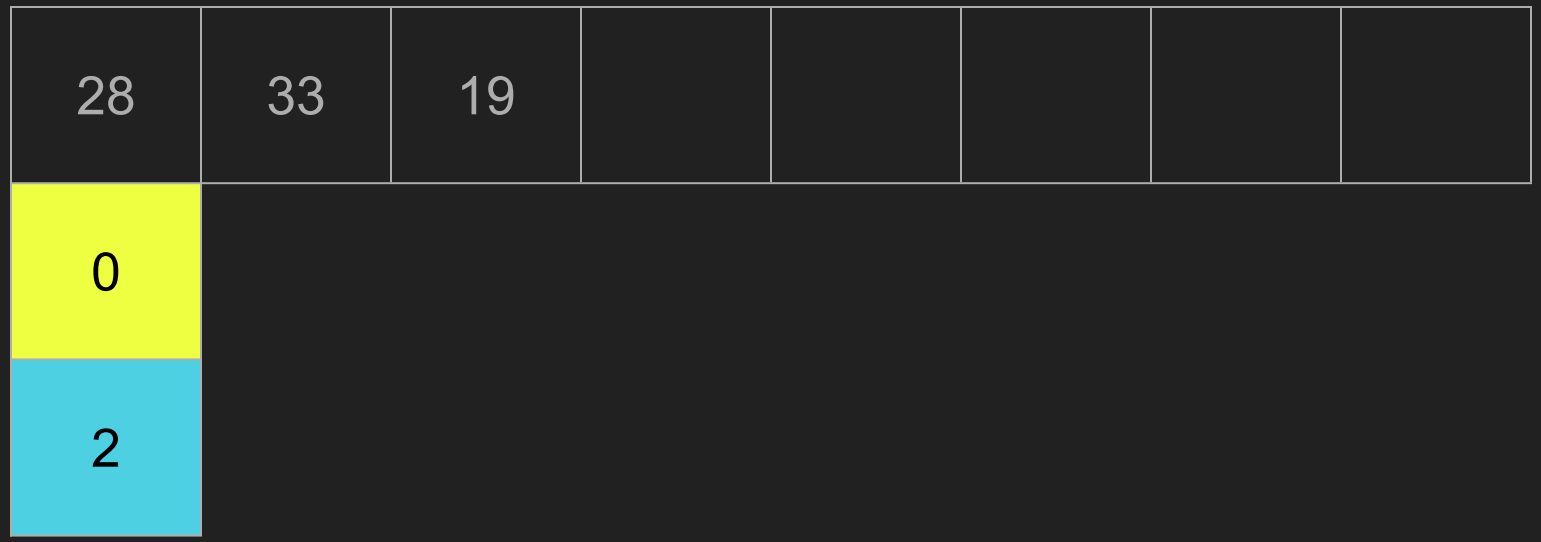
- Queue implemented as an array:

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



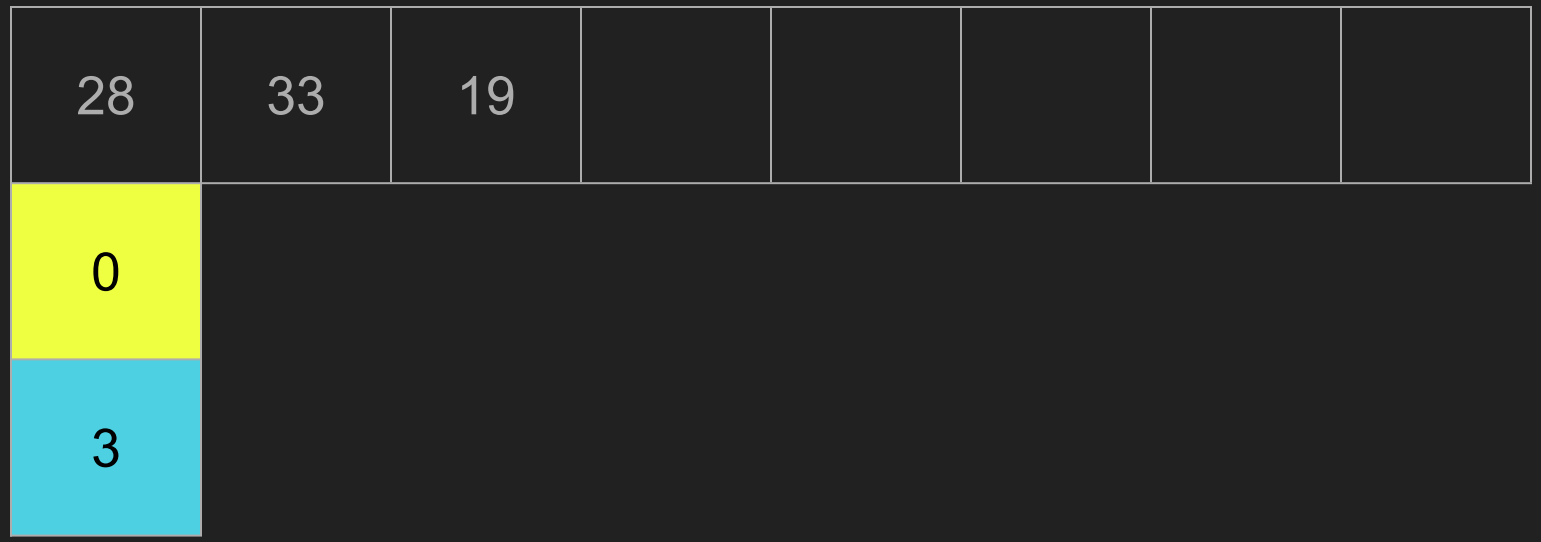
- Queue implemented as an array:

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



- Queue implemented as an array:

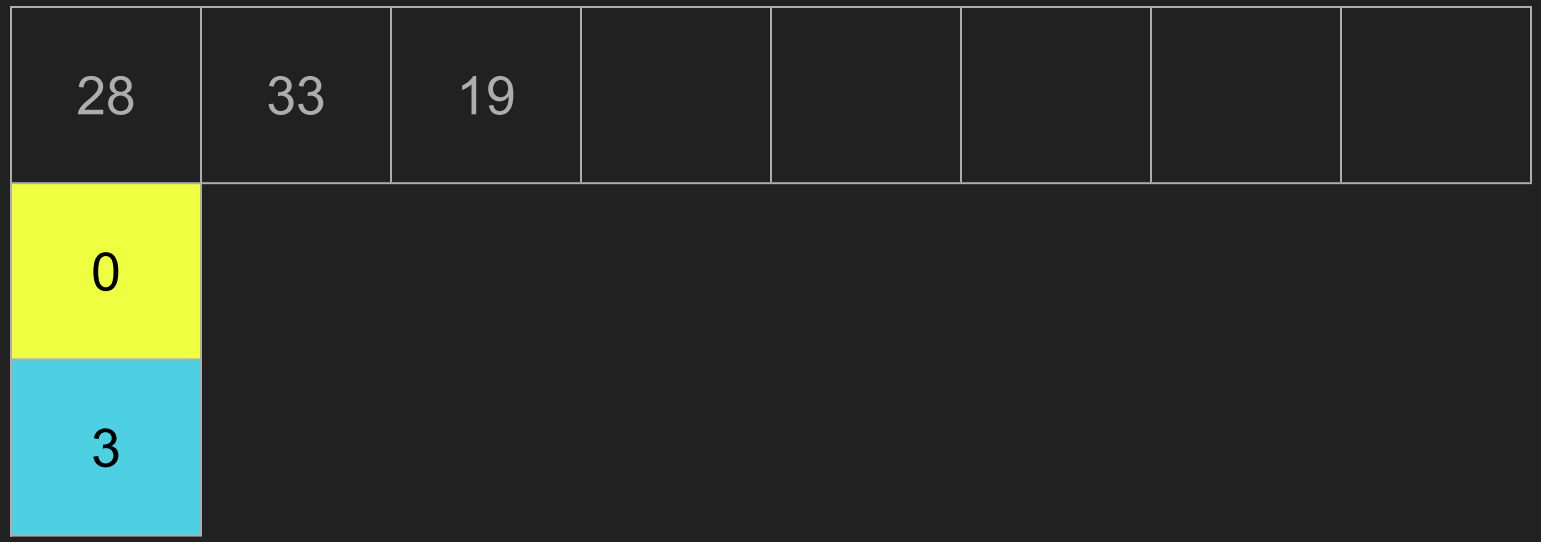
```
enqueue(&q, 19);
```



- 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.

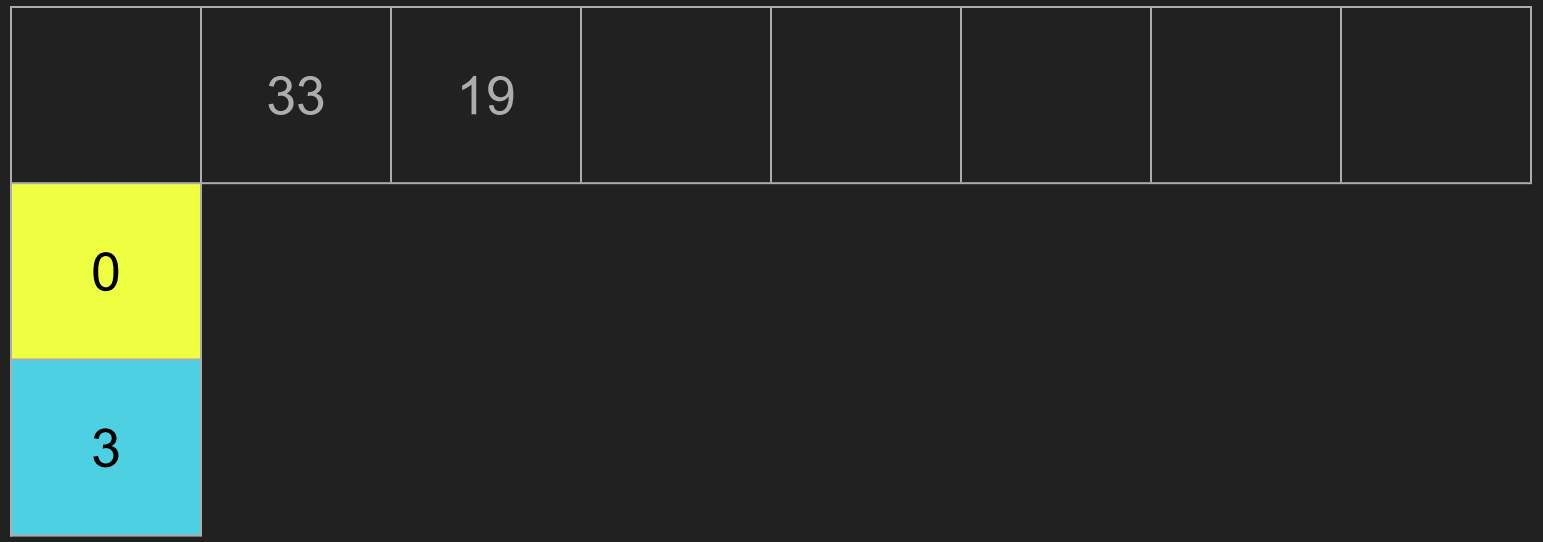
- Queue implemented as an array:

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



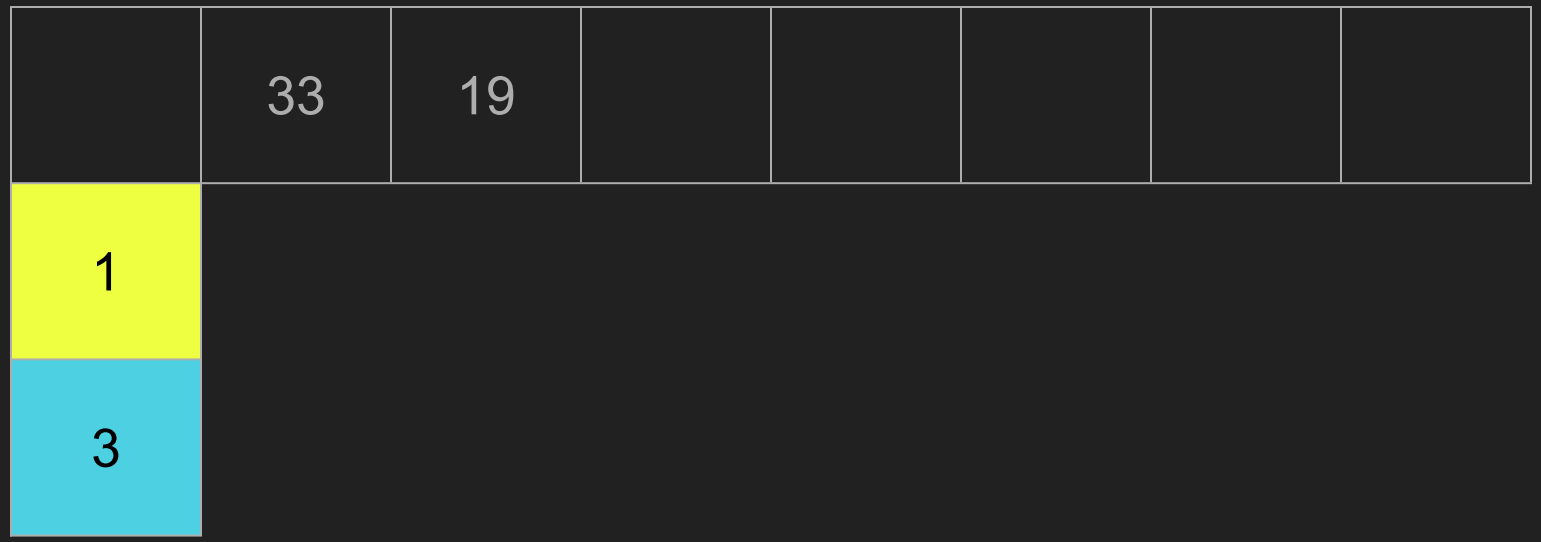
- Queue implemented as an array:

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



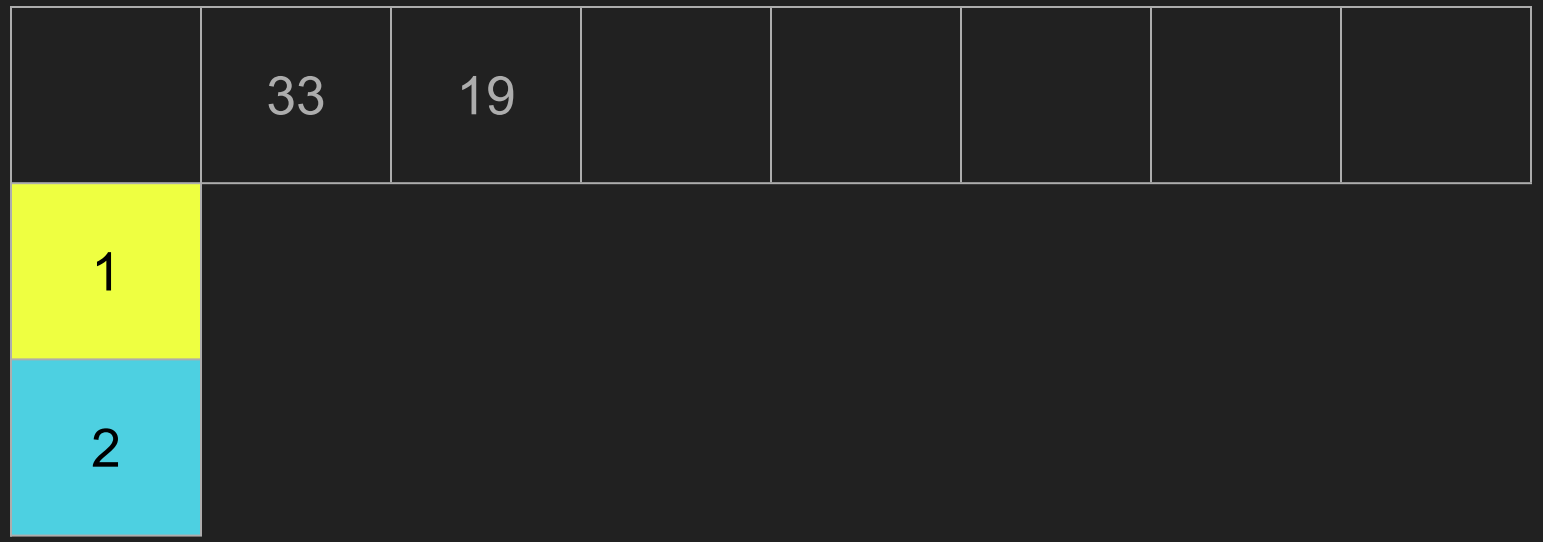
- Queue implemented as an array:

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



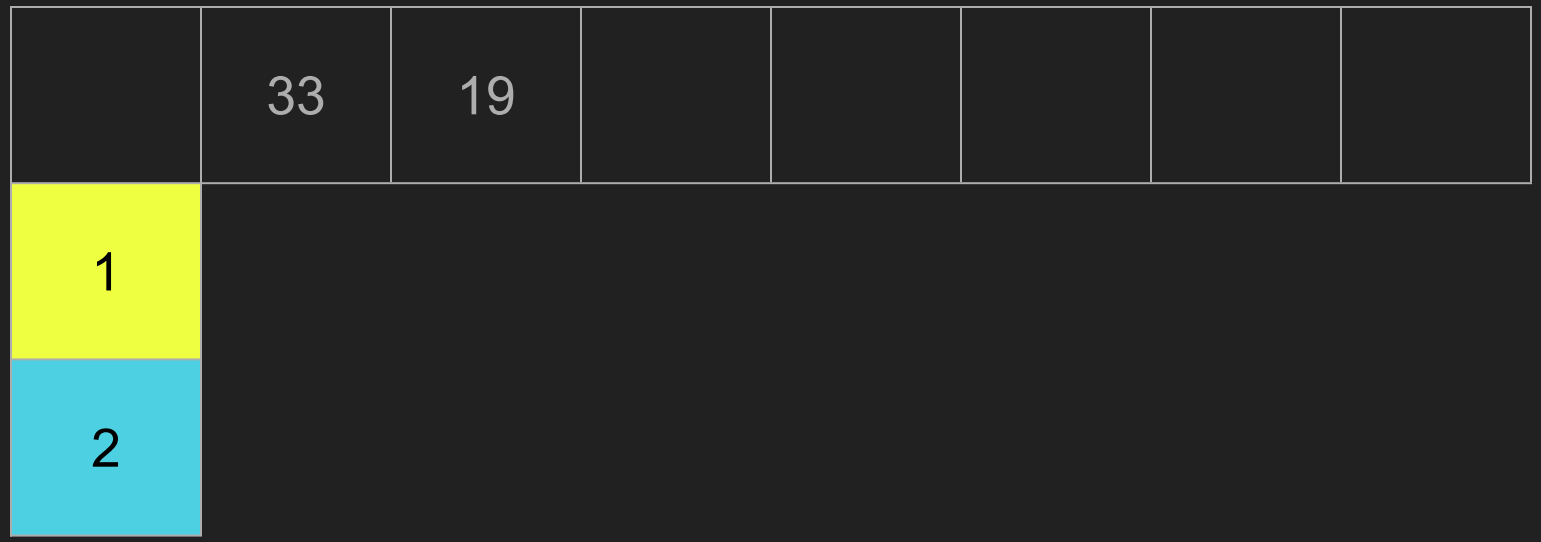
- Queue implemented as an array:

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



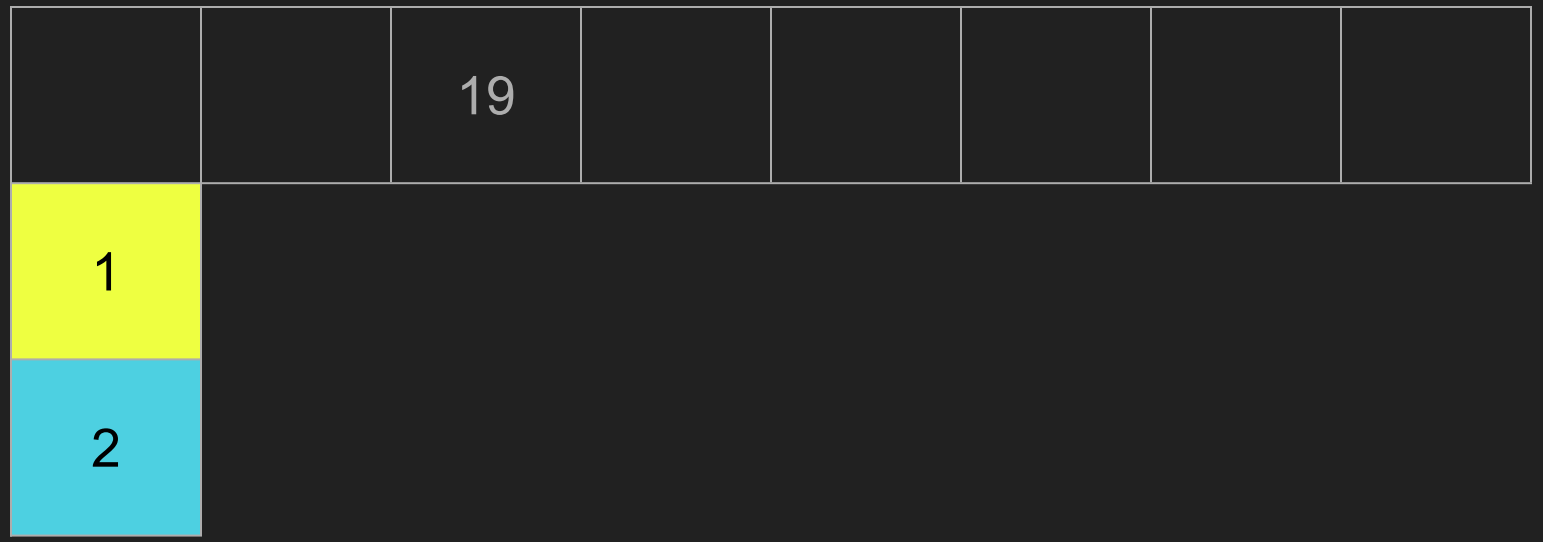
- Queue implemented as an array:

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



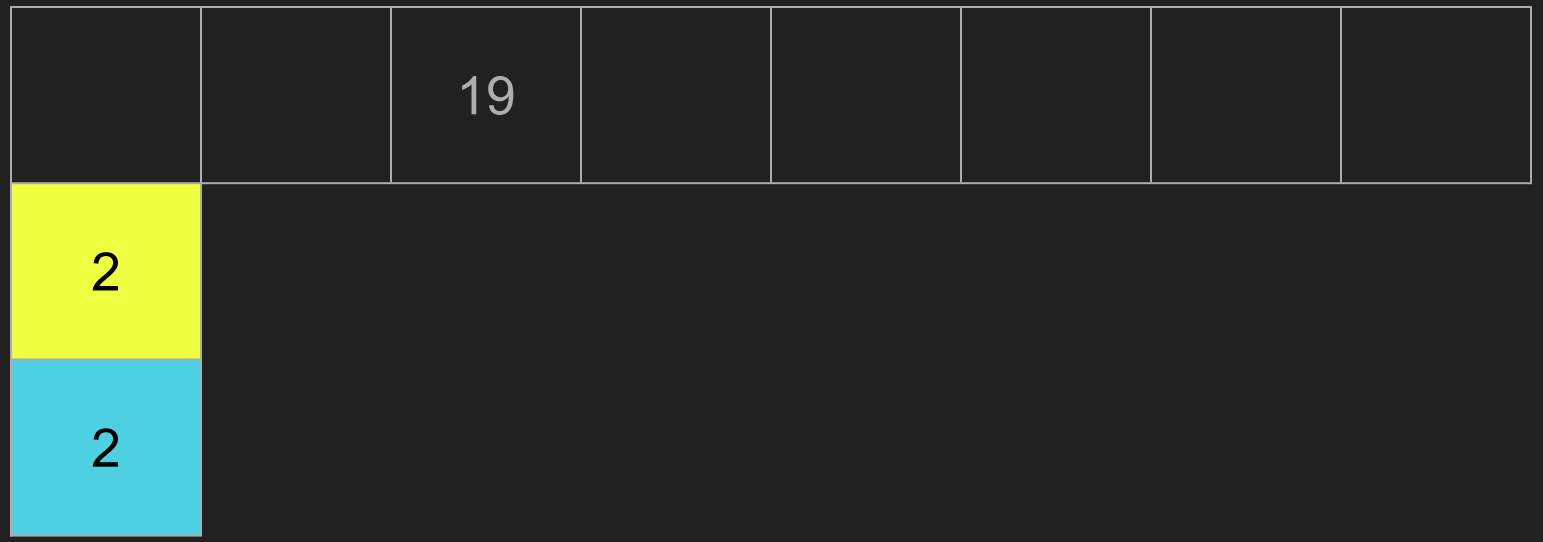
- Queue implemented as an array:

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



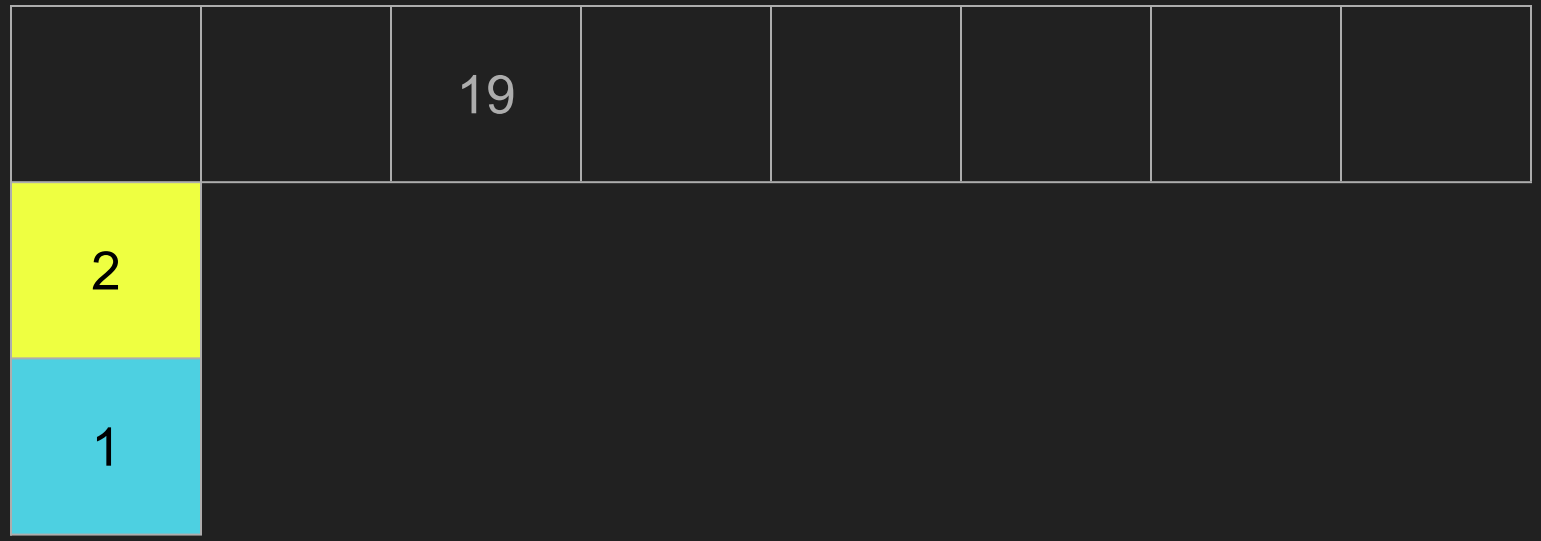
- Queue implemented as an array:

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



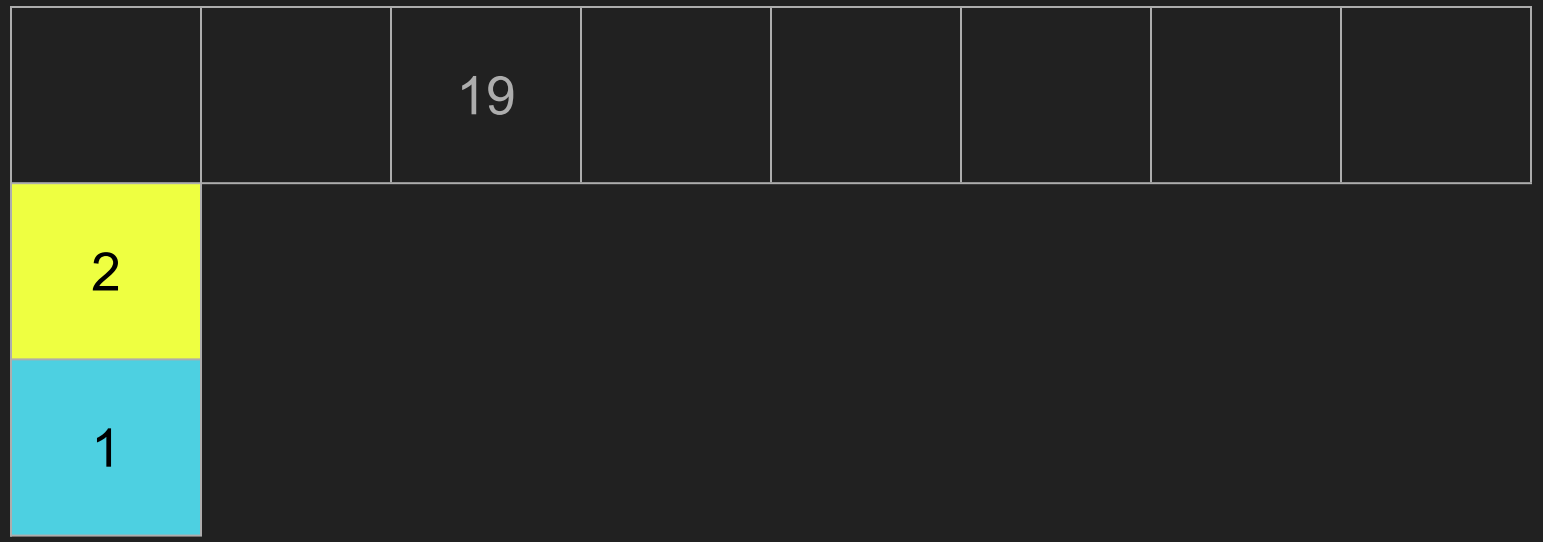
- Queue implemented as an array:

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



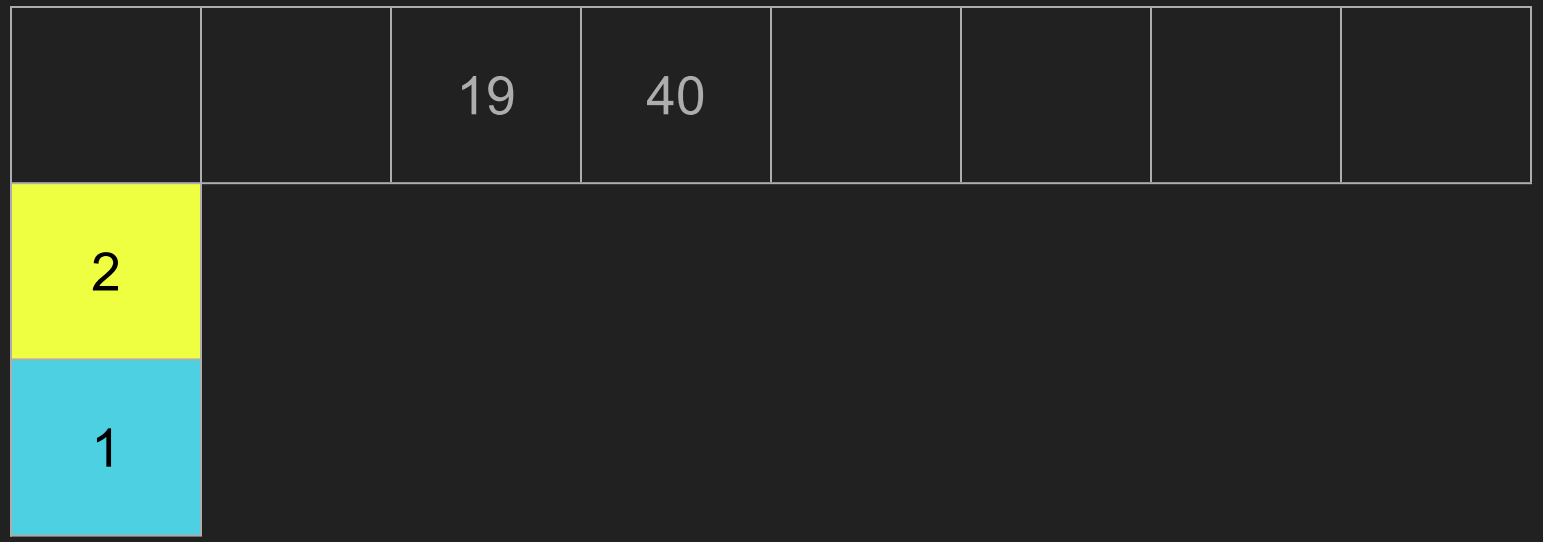
- Queue implemented as an array:

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



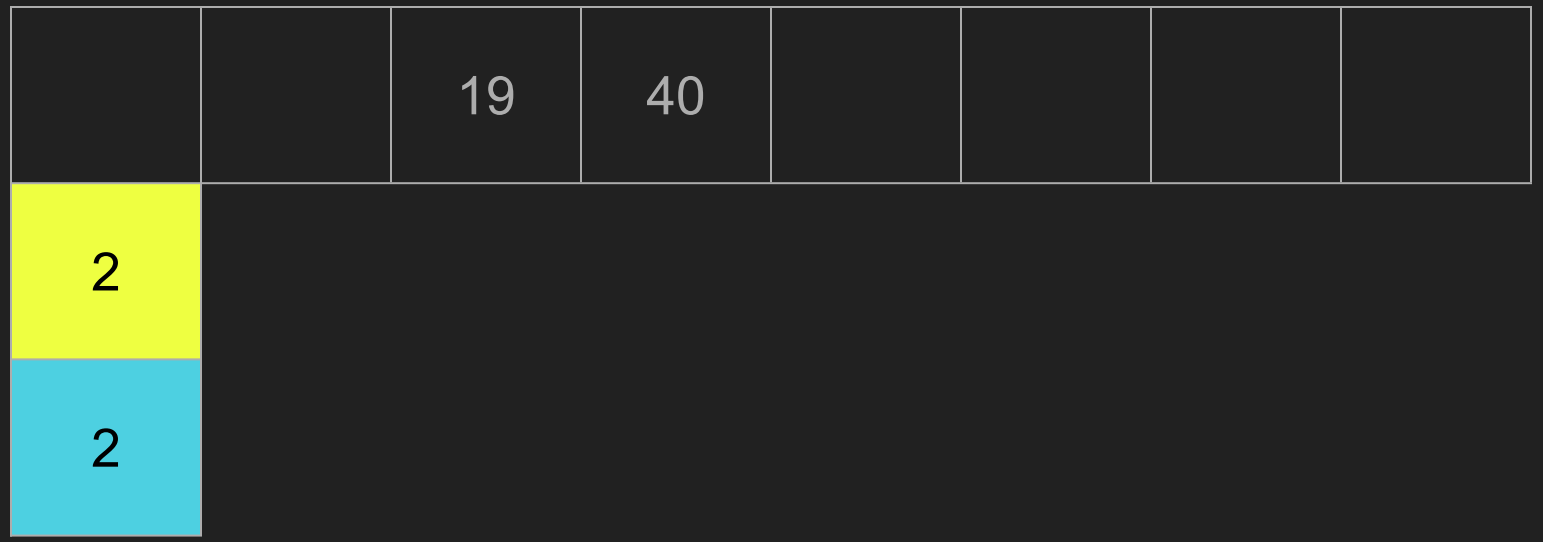
- Queue implemented as an array:

```
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.

New Syntax

- We’re used to using ‘dot notation’ with structs, like this: candidates[0].votes
- When we’re using pointers to structs though, we use **->** instead, like this: node->next or node->number

Practice Problems

github.com/cjleggett/week5section

linked_lists_1.c

- Task: write a program that allows a user to input numbers until they type CTRL-D (this is standard, so don’t worry about implementing this), add these numbers to the head of a linked list, print them out, then free them.
- Expected behavior:
  ./linked_lists_1.c
  num: 4
  num: 5
  num: 3
  num: CTRL-D
  3
  5
  4

Steps:

1. Allocate Memory for new node of linked list
2. Add new node to head of current linked list
3. Print each element of linked list
4. Free memory of linked list

- Steps 1 and 2 or steps 3 and 4 can be combined if you want
- There are several ways to do this, but I would think about either using loops, or writing helper functions that are recursive.

Lab!

hash_table_1.c

- Task: write a program that allows a user to input numbers. add these numbers to a hash table. Then allow the user to search for numbers in the hash table.
- Expected behavior:
  ./linked_lists_1.c
  demonstration….

linked_lists_2.c

- Task: write a program that allows a user to input numbers until they type CTRL-D (this is standard, so don’t worry about implementing this), add these numbers to the TAIL of a linked list, print them out, then free them.
- Expected behavior:
  ./linked_lists_1.c
  num: 4
  num: 5
  num: 3
  num: CTRL-D
  4
  5
  3

linked_lists_3.c

- Task: Same as before, but add these numbers IN ASCENDING ORDER to a linked list, print them out, then free them.
- Expected behavior:
  ./linked_lists_1.c
  num: 4
  num: 5
  num: 3
  num: CTRL-D
  3
  4
  5

If there’s time:

- Try implementing a stack using linked lists by creating your own struct, and then writing add() and pop() functions

Problem Set Preview: Speller

- You'll be implementing a dictionary using a data structure we've learned about (probably a hash table, maybe a trie)
- Start with an easy hash function, then move on to more exciting ones if you have time.