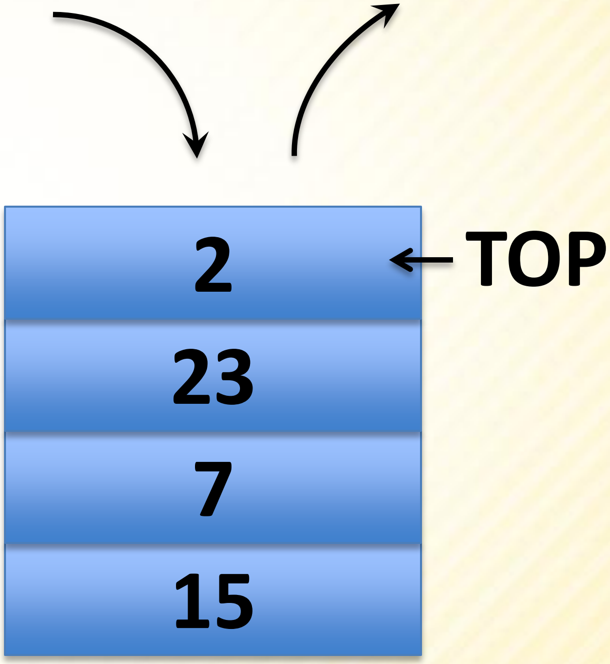
STACK & QUEUES

COP 3502

Stacks

- A stack is a data structure that stores information arranged like a stack.
  - We have seen stacks before when we used a stack to trace through recursive programs.
- The essential idea is that the last item placed into the stack is the first item removed from the stack
  - Or LIFO(Last In, First Out) for short.

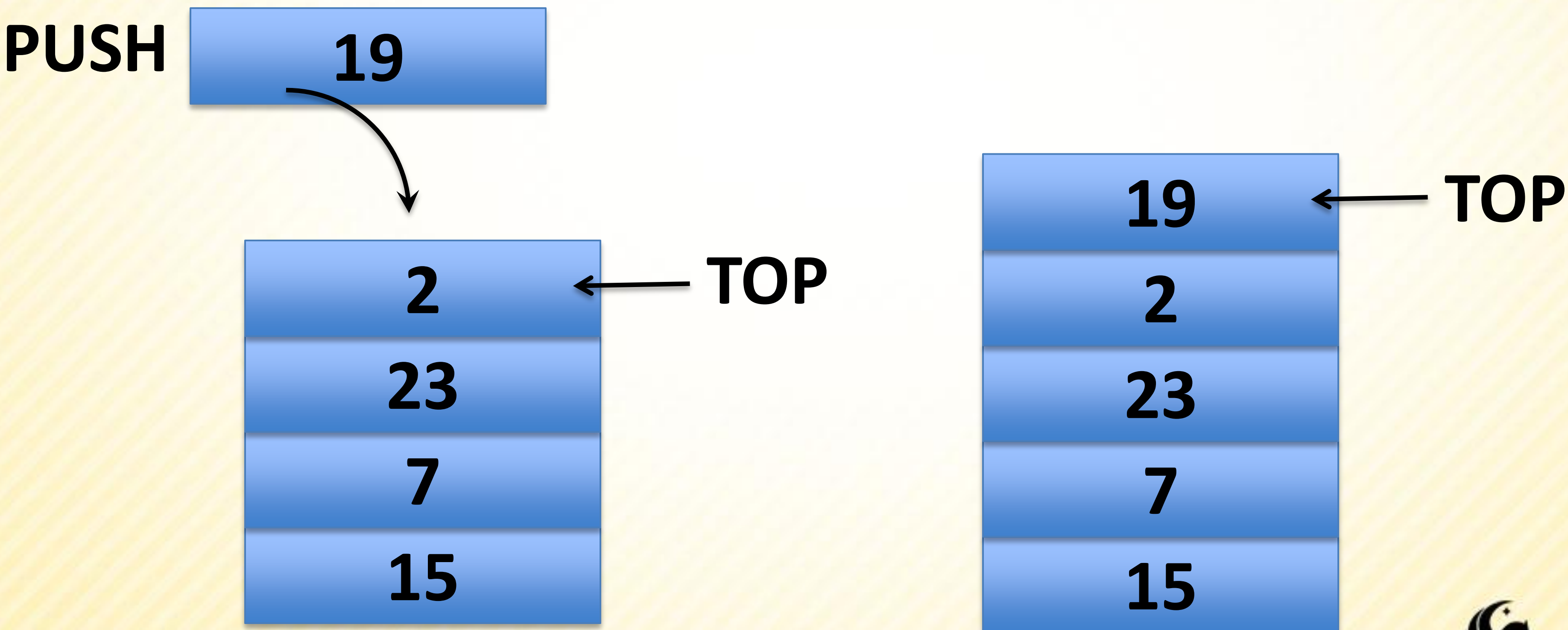


Stacks

- Stacks:
  - Stacks are an Abstract Data Type
    - ➢ They are NOT built into C
  - So we must define them and their behaviors
- Stack behavior:
  - A data structure that stores information in the form of a stack.
    - ➢ Contains any number of elements of the same type.
  - Access policy:
    - ➢ The last item placed on the stack is the first item removed from the stack.

Stacks

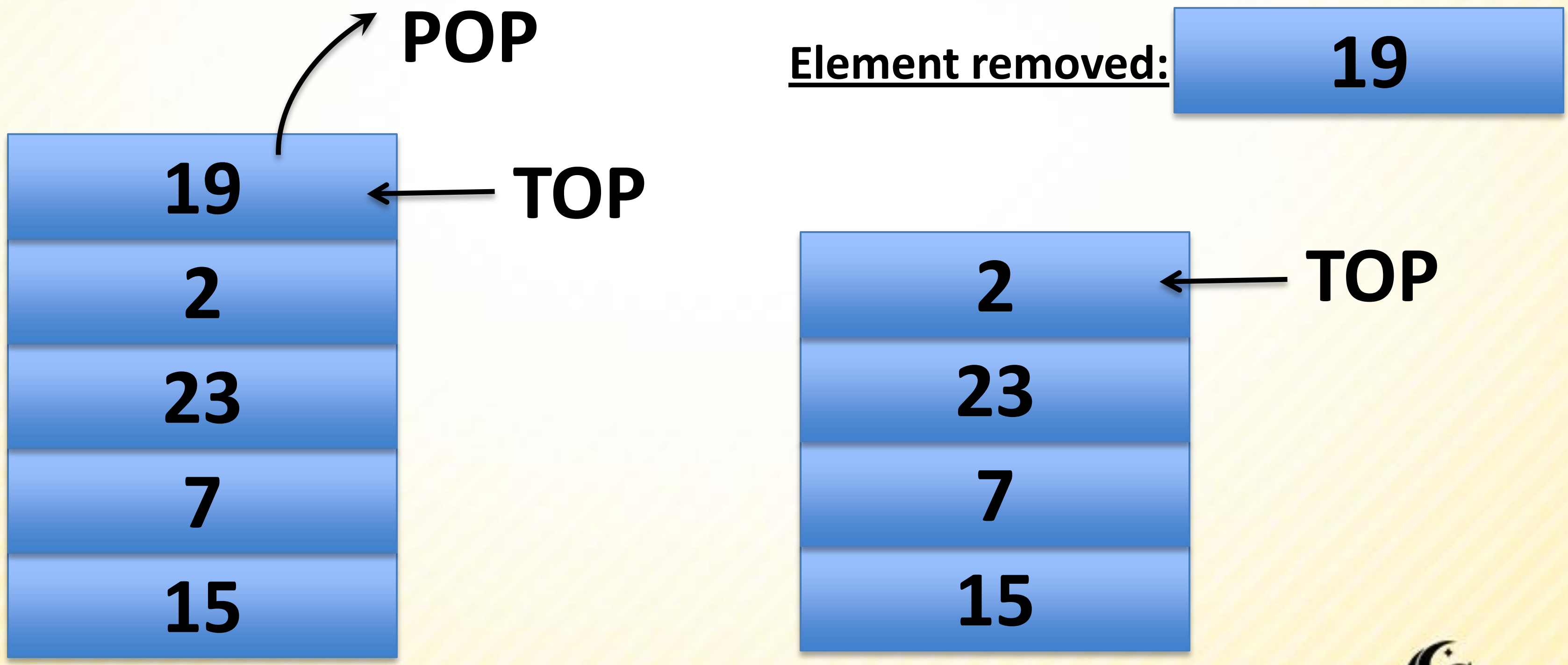
- Stacks:
  - Basic Operations: PUSH and POP
    - ➢PUSH – PUSH an item on the top of the stack.



(Stack before Push Operation) **(Stack after Push Operation)**

Stacks

- Stacks:
  - Basic Operations: PUSH and POP
    - POP – POP off the item in the stack and return it.



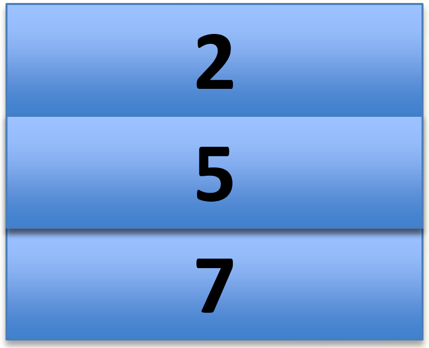
(Stack before Pop Operation) **(Stack after Pop Operation)**

Stacks

- Stacks:
  - Other useful operations:
    - **empty** – typically implemented as a boolean function that returns true if no items are in the stack.
    - **full** – returns true if no more items can be added to the stack.
      - In theory a stack should never be full, but any actual implementation has a limit on the # of elements it can store.
    - **top** – simply returns the value stored at the top of the stack without popping it off the stack.

Stacks

- Simple Example:
  - PUSH(7)
  - PUSH(3)
  - PUSH(2)
  - POP
  - POP
  - PUSH(5)
  - POP



Stacks

- On Monday we used stacks to:
  1) Convert infix expressions to postfix expressions
  2) Evaluate postfix expressions

  - Infix: 2 * 3 + 5 + 3 * 4
  - **Postfix: 2 3 * 5 + 3 4 * +**
    - The operator follows all of its operands
    - Reduces computer memory access and utilizes the stack to evaluate expressions.
    - No parentheses are necessary
    - Still used by some calculators.

Stack Implementation

```
struct stack {
    int items[SIZE];
    int top;
};
```

- Last class we talked about 2 typ implementations – Array and L
- **<u>Array Implementation:</u>**
  - When we initialize the array,
  - There are no items in the stack, so what is top set to in the initialize function?
    - -1
  - What if we push(10), then what is top?
    - 0
  - What if we pop(), then what is top?
    - -1
  - For all of these operations we either access top + 1, or top
    - so all operations are O(1) (we don't have to traverse the array.)

Stack Implementation

```
struct stack {
    int data;
    struct stack *next;
};
```

- Linked List Implementation
  - Notice that we don't have a 'top'
  - Why?
    - The top will ALWAYS be the first node
    - And we don't need to worry about the size since it's a linked list that can expand while there's heap memory available.
    - So we only need to either add an element to the front or take an element off of the front
      - The runtime for either operation is O(1)

Stack – Linked List Implementation

- Why do we use double pointers?
  - If we want to be able to return a 0 or a 1 if a Push is successful,
    - We can't return the address of the new front of the list (which is what we would usually do with a linked list)
  - So we'll pass a pointer to the front of the list (stack **front)
    - and then if we modify (*front) in the Push function we are changing not the local value of the pointer,
    - but we're changing the contents of the pointer in the same memory address that was passed from main so those changes will be reflected in main.

Stack Application

- We did 2 examples last time:
  1) Reading in a list of numbers from a user and printing it in backwards order.
     - We also talked about reading in each character of a string and printing it out in backwards order.
  2) Checking if we have matching parentheses

Queues

- If we wanted to simulate customers waiting in a line to be served,
  - We wouldn't use a stack…
    - LIFO is only going to make the person that got in line first mad.

Queues

- We would want to use FIFO
  - First In First Out, or 1st in line 1st one to get served.
- Instead of push and pop, we have the operations
  - Enqueue and Dequeue that add/remove elements from the list.

Queue Basic Operations

- **Enqueue:**
  - Inserts an element at the back of the queue
  - Returns 1 if successful, 0 otherwise.
- **Dequeue:**
  - Removes the element at the front of the queue.
  - Returns the removed element.
- **Peek**
  - Looks at the element at the front of the queue without removing it.
  - Returns the front element.
- **isEmpty**
  - Checks to see if the queue is empty.
  - Returns true or false.
- **isFull**
  - Checks to see if the queue is full.
  - Returns true or false.

Queue Example

Starting Queue:

3	5	7	9	11

front … back

Time 1:

3	5	7	9	11	13

Time 2:

5	7	9	11	13

Time 3:

5	7	9	11	13	15

Time 4:

7	9	11	13	15

Time 5:

9	11	13	15

TIME	OPERATION
1	Enqueue(13)
2	Dequeue()
3	Enqueue(15)
4	Dequeue()
5	Dequeue()

Queues - Array Implementation

- What would we need for an array implementation?
  - We need an array obviously
  - And we need to keep track of the front and the back.

BAD Queue Implementation Example

Starting Queue:

3	5	7	9	11	
front				back	

Time 1:

3	5	7	9	11	13
front					back

Time 2:

5	7	9	11	13	
front				back	

Time 3:

5	7	9	11	13	15
front					back

Time 4:

7	9	11	13	15	
front				back	

Time 5:

9	11	13	15		
front			back		

TIME	OPERATION
1	Enqueue(13)
2	Dequeue()
3	Enqueue(15)
4	Dequeue()
5	Dequeue()

Notice that you have to Shift the contents of the Array over each time front changes

Queues: Array Implementation

```
struct queue {
    int *elements;
    int front;
    int numElements;
};
```

- We will use the following revamped idea to store our queue structure:
  - Keep track of the array, the front, and the current number of elements.

Queues: Array Implementation

```
struct queue {
     int *elements;
     int front;
     int numElements;
};
```

- Enqueue:
  - We'll simply add the given element to the index "back" in the array.
  - BUT we're not storing "back"!!!!!
  - What must we do instead?
    - Add it to the index: front + numElements
    - But what if this goes outside the bounds of our array?

numElements = 4

9	11	13	15		

front

Queues: Array Implementation

```
struct queue {
    int *elements;
    int front;
    int numElements;
    int queueSize;
};
```

- Enqueue(17):
  - Add it to the index: front + numElements
    - ➢ But what if this goes outside the bounds of our array?
    - ➢ Front = 2, plus numElements = 4, gives us 6
    - ➢ We can mod by the queueSize
    - ➢ (front + numElements) % queueSize = 0

numElements = 5

17		**9**	**11**	**13**	**15**
		front			

Queues: Array Implementation

```
struct queue {
    int *elements;
    int front;
    int numElements;
    int queueSize;
};
```

- So we're allowing our array to essentially wrap around.
  - This way we don't have to copy the contents of our array over if front or back moves

numElements = 5

17		9	11	13	15
		front			

Queues: Array Implementation

```
struct queue {
    int *elements;
    int front;
    int numElements;
    int queueSize;
};
```

- Dequeue
  - If the numElements > 0
    - ➢numElements--;
    - ➢front = (front + 1) % queueSize

numElements = 4

17			11	13	15

front front

Queues - Dynamically Allocated Array Imple

```
struct queue {
      int *elements;
      int front;
      int numElements;
      int queueSize;
};
```

- What if our numElements == queueSize?
  - We can realloc more memory for our array and update queueSize!
  - But we also need to make sure we copy over the wraparound values correctly.

Queues - Linked List Implementation

- We are going to need a linked list
  - So we'll use the same node implementation as before.
- But we'll need to keep track of the front and the back.
  - Otherwise either enqueue or dequeue would require an O(n) traversal each time.
- So we'll keep a front and back pointer inside of a structure called queue.

```
struct node {
     int data;
     struct node *next;
};
struct queue {
     struct node *front;
     struct node *back;
};
```