Chapter 18 - Stacks and Queues

Spring 2022

Objectives (1 of 2)

In this chapter, you will:

- Learn about stacks
- Examine various stack operations
- Learn how to implement a stack as an array
- Learn how to implement a stack as a linked list
- Learn about infix, prefix, and postfix expressions, and how to use a stack to evaluate postfix expressions

Objectives (2 of 2)

- Learn how to use a stack to remove recursion
- Learn about queues
- Examine various queue operations
- Learn how to implement a queue as an array
- Learn how to implement a queue as a linked list
- Discover how to use queues to solve simulation problems

Stacks (1 of 4)

- **Stack**: a data structure in which elements are added and removed from one end only
  - Addition/deletion occur only at the top of the stack
  - **Last in first out (LIFO)** data structure
- Operations:
  - **Push**: to add an element onto the stack
  - **Pop**: to remove an element from the stack

Stacks (2 of 4)

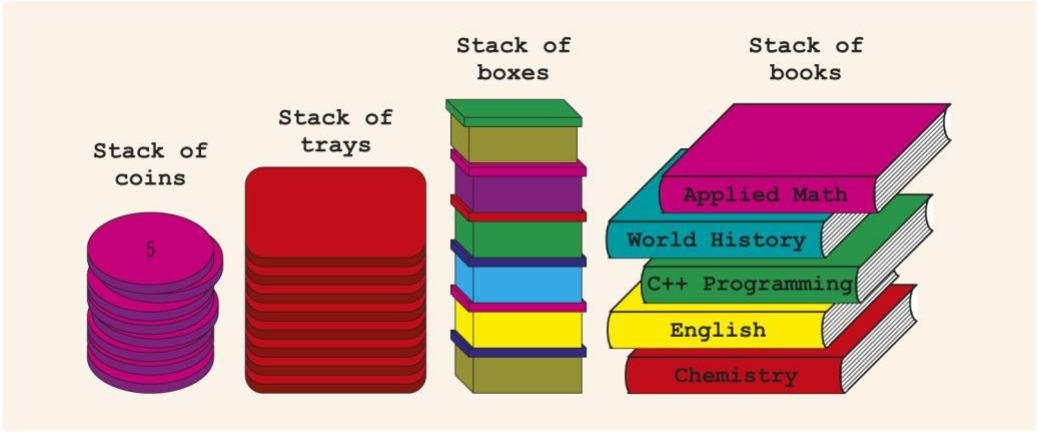

FIGURE 18-1 Various types of stacks

Stacks (3 of 4)

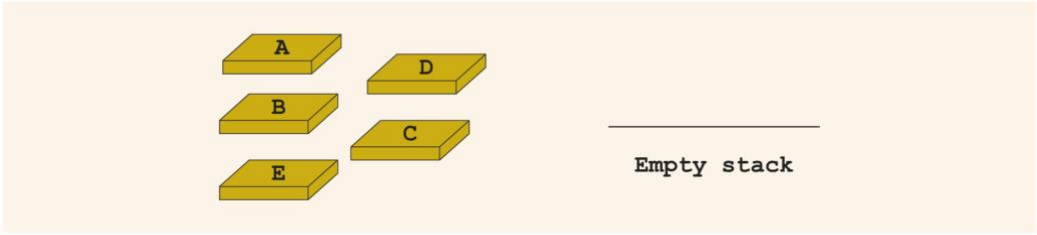

FIGURE 18-2 Empty stack

Stacks (4 of 4)

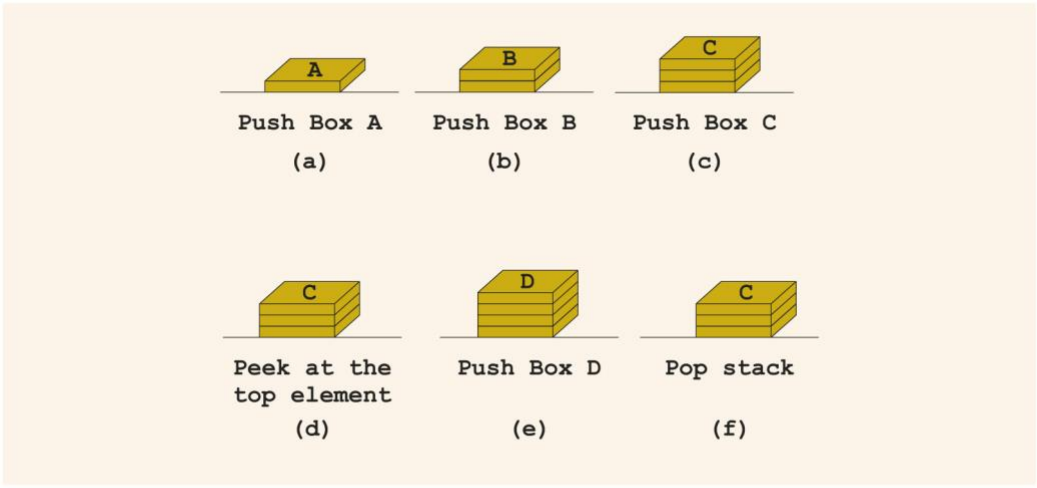

FIGURE 18-3 Stack operations

Stack Operations

- In the abstract class **stackADT**:
  - initializeStack

- isEmptyStack
- isFullStack
- push
- top
- pop

UML class diagram of the class stackADT

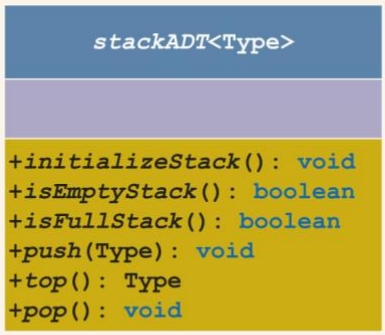

FIGURE 18-4 UML class diagram of the class stackADT

Implementation of Stacks as Arrays (1 of 5)

- First element goes in first array position, second in the second position, etc.
- Top of the stack is index of the last element added to the stack
- Stack elements are stored in an array, which is a random access data structure
  - Stack element is accessed only through top
- To track the top position, use a variable called **stackTop**

Implementation of Stacks as Arrays (2 of 5)

- Can dynamically allocate array
  - Enables user to specify size of the array
- class **stackType** implements the functions of the abstract class **stackADT**

Implementation of Stacks as Arrays (3 of 5)

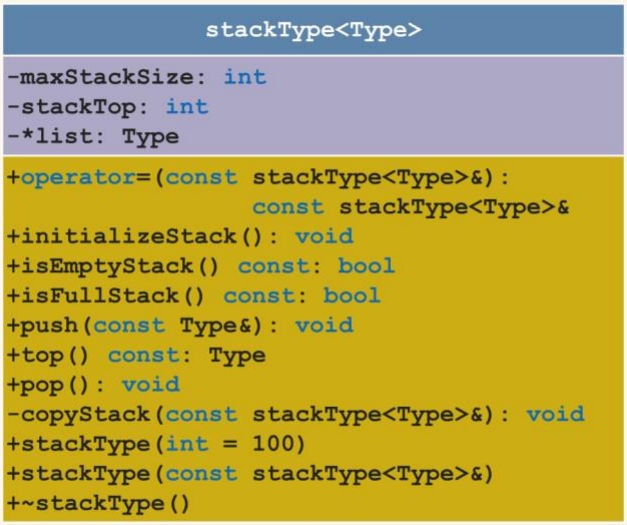

FIGURE 18-5 UML class diagram of the `class` **`stackType`**

Implementation of Stacks as Arrays (4 of 5)

- C++ arrays begin with the index 0
  - Must distinguish between:
    - Value of **stackTop**
    - Array position indicated by stackTop
- If **stackTop** is 0, stack is empty
- If **stackTop** is nonzero, stack is not empty
  - Top element is given by **stackTop - 1**

Implementation of Stacks as Arrays (5 of 5)

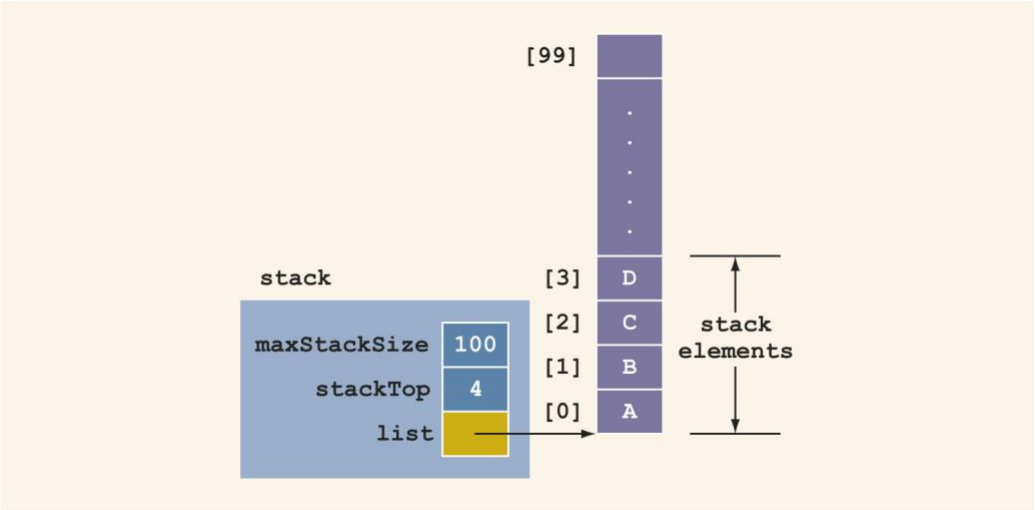

FIGURE 18-6 Example of a stack

Initialize Stack

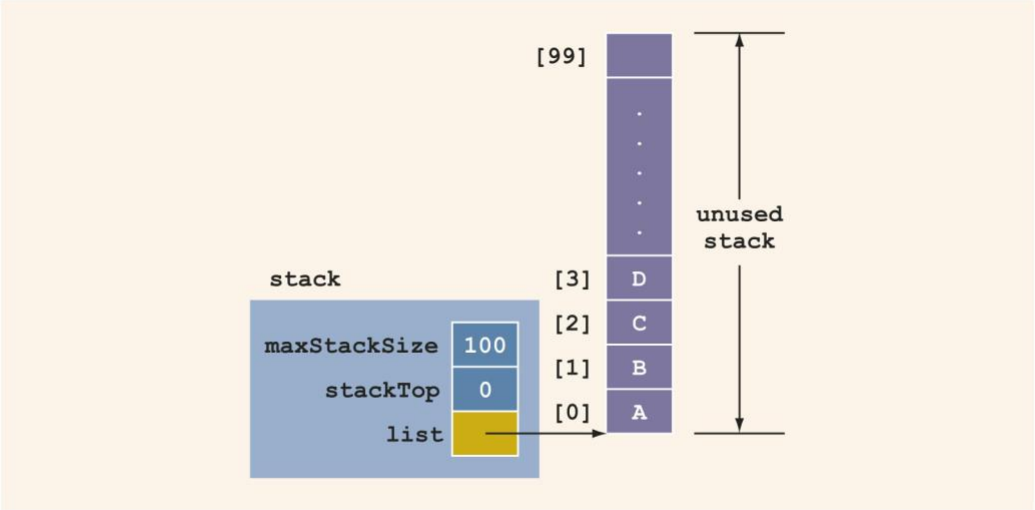

FIGURE 18-7 Empty stack

Empty Stack/Full Stack

- Stack is empty if `stackTop = 0`

```
template <class T>
bool stackType<T>::isEmptyStack() const {
    return stackTop == 0;
}
```

- Stack is full if `stackTop = maxStackSize`

```
template <class T>
bool stackType<T>::isFullStack() const {
```

```
        return stackTop == masxStackSize;
    }
```

Push (1 of 3)

- Store the **newItem** in the array component indicated by **stackTop**
- Increment **stackTop**
- **Overflow** occurs if we try to add a new item to a full stack

Push (2 of 3)

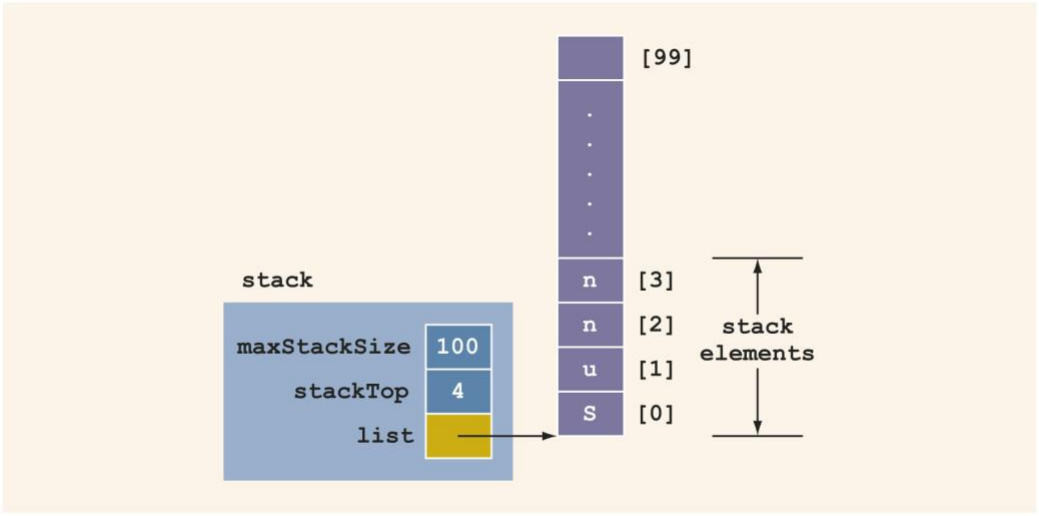

FIGURE 18-8 Stack before pushing `y`

Push (3 of 3)

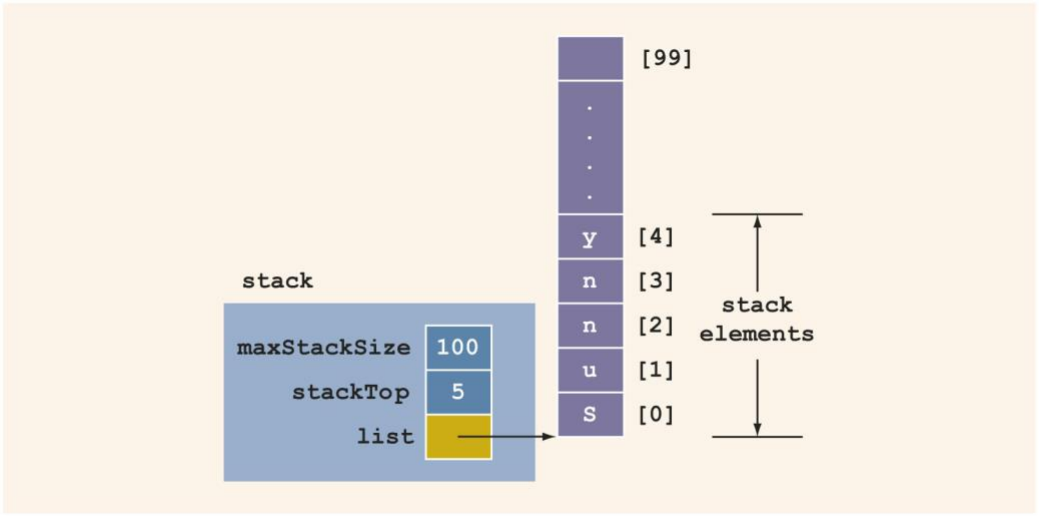

FIGURE 18-9 Stack after pushing `y`

Return the Top Element

- **top** operation:

- Returns the top element of the stack

```
template <class T>
T stackType<T>::top() const {
    assert(stackTop != 0);
    return list[stackTop - 1];
}
```

Pop (1 of 3)

- To remove an element from the stack, decrement **stackTop** by 1
- **Underflow** condition: trying to remove an item from an empty stack

```
template <class T>
void stackTypeT>::pop() {
    if (!isEmptyStack()) {
        --stackTop;
    }
}
```

Pop (2 of 3)

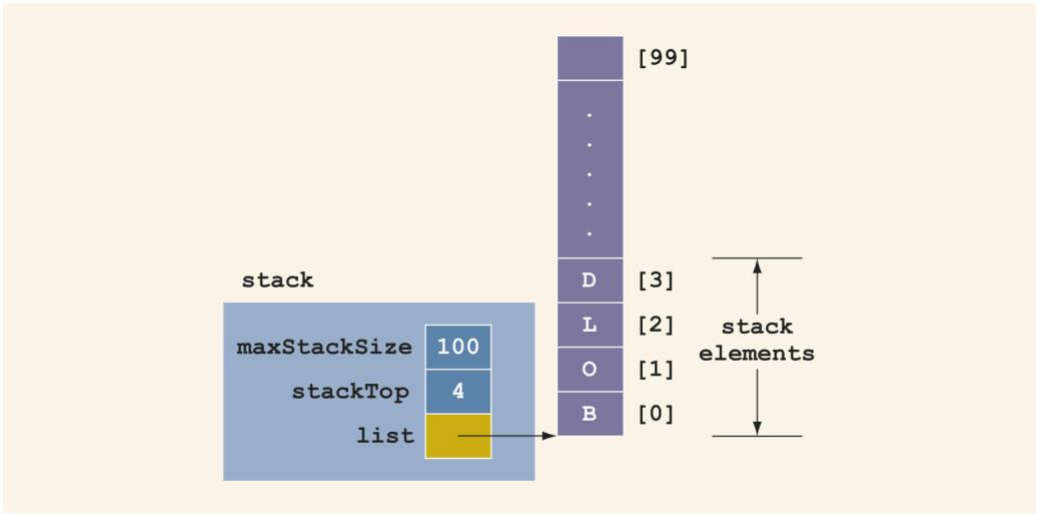

FIGURE 18-10 Stack before popping D

Pop (3 of 3)

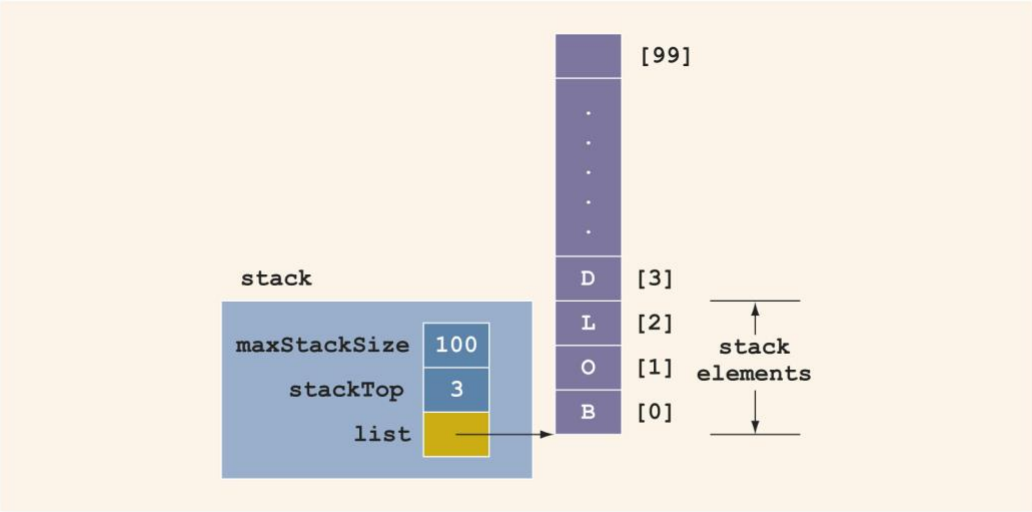

FIGURE 18-11 Stack after popping D

Queues

- **Queue**: set of elements of the same type
- Elements are:
  - Added at one end (the **back** or **rear**)
  - Deleted from the other end (the **front**)
- **First In First Out (FIFO)** data structure
  - Middle elements are inaccessible
- Example:
  - Waiting line in a bank

Queue Operations

- Queue operations include: – **initializeQueue** – **isEmptyQueue** – **isFullQueue** – **front** – **back** – **addQueue** – **deleteQueue**
- Abstract class **queueADT** defines these operations

Implementation of Queues as Arrays (1 of 15)

- Need at least four (member) variables:
  - Array to store queue elements
  - **queueFront** and **queueRear**
    - To track first and last elements
  - **maxQueueSize**
    - To specify maximum size of the queue

Implementation of Queues as Arrays (2 of 15)

- To add an element to the queue:
  - Advance queueRear to next array position

- Add element to position pointed by queueRear

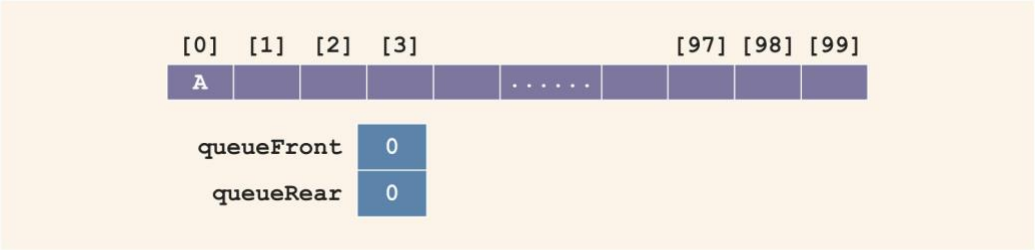

FIGURE 18-26 Queue after the first `addQueue` operation

Implementation of Queues as Arrays (3 of 15)

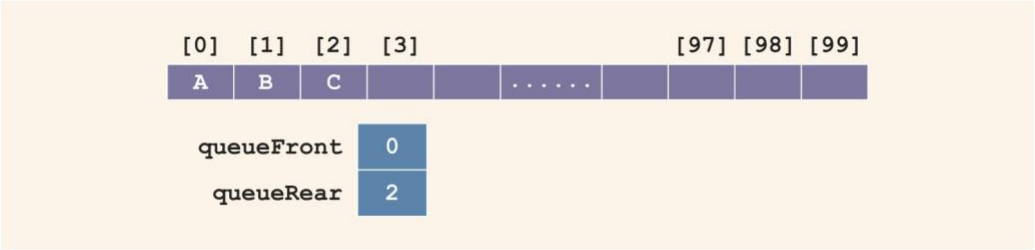

FIGURE 18-27 Queue after two more `addQueue` operations

Implementation of Queues as Arrays (4 of 15)

- To delete an element from the queue:
  - Retrieve element pointed to by **queueFront**
  - Advance **queueFront** to next queue element

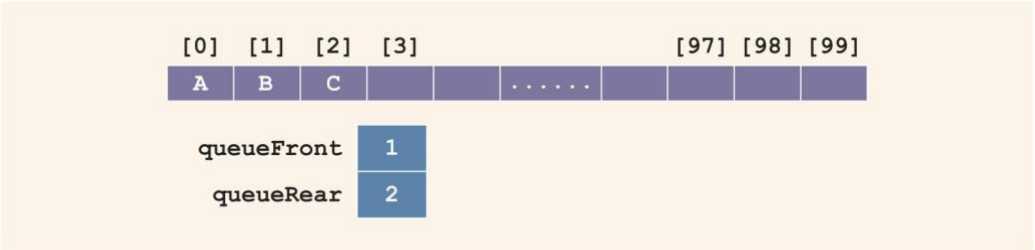

FIGURE 18-28 Queue after the `deleteQueue` operation

Implementation of Queues as Arrays (5 of 15)

- Will this queue design work?
  - Let **A** represent adding an element to the queue
  - Let **D** represent deleting an element from the queue
  - Consider the following sequence of operations:
    - **AAADADADADADADADA**…

Implementation of Queues as Arrays (6 of 15)

- This would eventually set **queueRear** to point to the last array position
  - Giving the impression that the queue is full

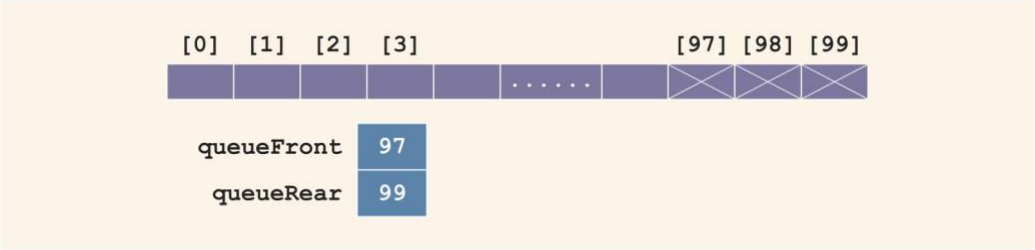

FIGURE 18-29 Queue after the sequence of operations AAADADADADADA . . .

Implementation of Queues as Arrays (7 of 15)

- Solution 1: When queue overflows at rear (**queueRear** points to the last array position):
  - Check value of **queueFront**
  - If **queueFront** indicates there is room at front of array, slide all queue elements toward the first array position
  - Problem: too slow for large queues
- Solution 2: Assume that the array is circular

Implementation of Queues as Arrays 8 of 15)

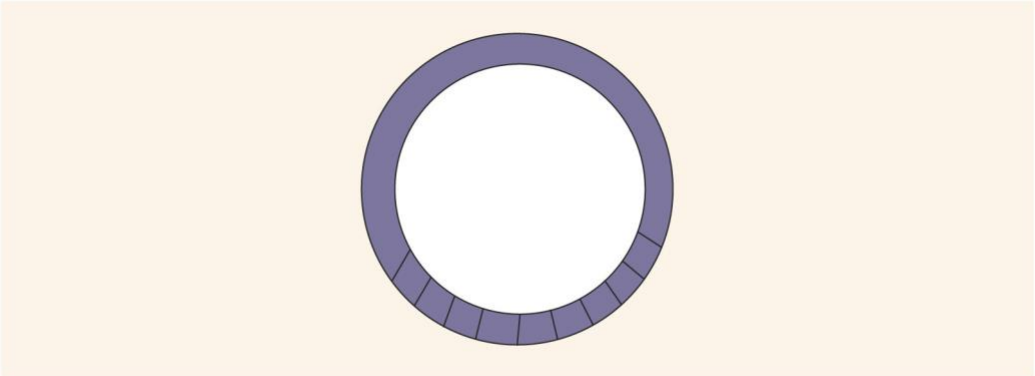

FIGURE 18-30 Circular queue

Summary (1 of 2)

- **Stack**: items are added/deleted from one end
  - Last In First Out (LIFO) data structure
  - Operations: push, pop, initialize, destroy, check for empty/full stack
  - Can be implemented as array
  - Middle elements should not be accessed directly

Summary (2 of 2)

- **Queue**: items are added at one end and removed from the other end
  - First In First Out (FIFO) data structure
  - Operations: add, remove, initialize, destroy, check if queue is empty/full

- Can be implemented as array

- Middle elements should not be accessed directly
- Is a restricted version of array and linked list

Questions?