Queue Implementations

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today's Plan

Review

Queue Implementations

Where are we at?
(Learning goals Review/
Recap)

Announcements and Syllabus Check

Queue ADT

```
#ifndef QUEUE_H_
#define QUEUE_H_

template<class ItemType>
class Queue
{

public:
    Queue();
    void enqueue(const ItemType& newEntry); // adds an element to back queue
    void dequeue(); // removes element from front of queue
    ItemType front() const; // returns a copy of element at the front of queue
    int size() const; // returns the number of elements in the queue
    bool isEmpty() const; // returns true if no elements in queue, false otherwise

private:
        //implementation details here


};    //end Queue

#include "Queue.cpp"
#endif // QUEUE_H_ `
```

Choose a Data Structure

Array?

Vector?

Linked List?

We are looking to enqueue and dequeue in O(1) time

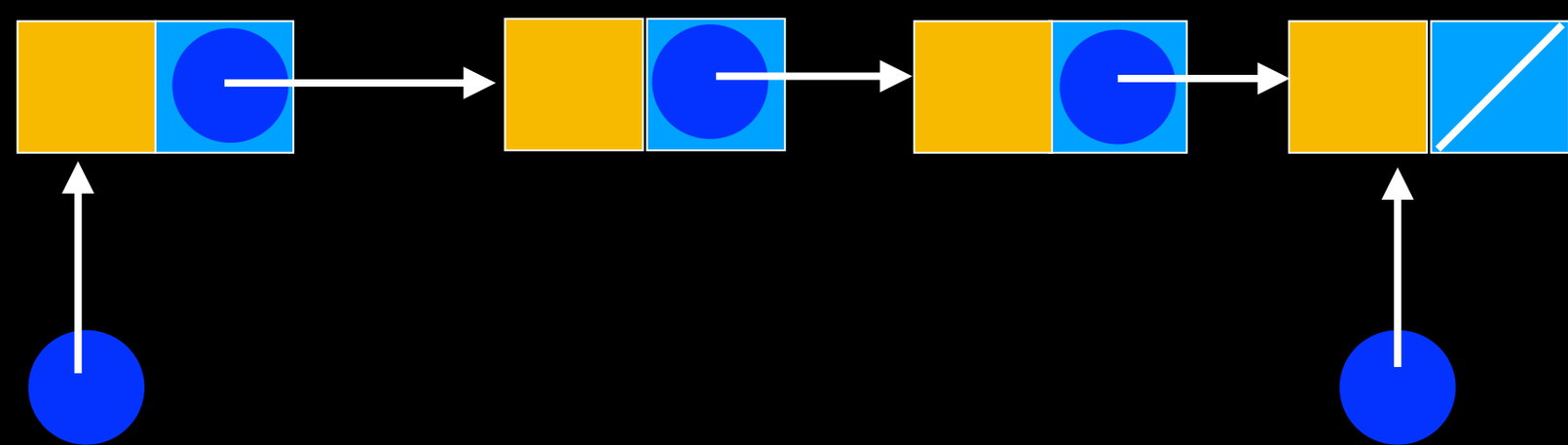
Recall Analysis for Stack

Amortized Analysis

	Big-O	Size unbounded
Array	O(1)	✗
Vector	O(1)+	✓
Linked Chain	O(1)	✓

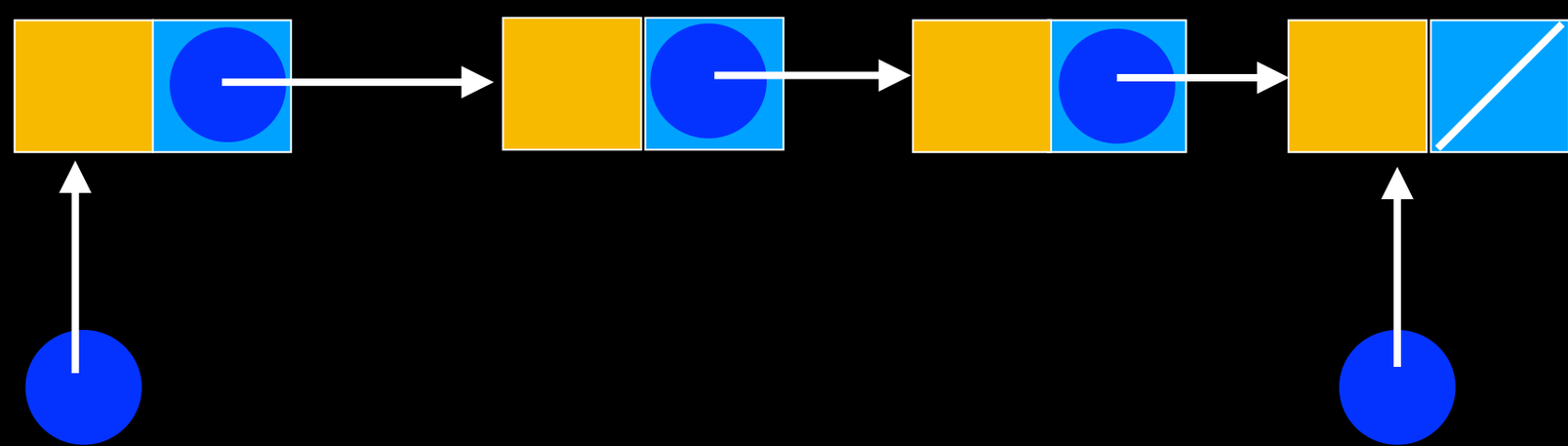
Singly Linked Chain

Where is front?
Where is back?

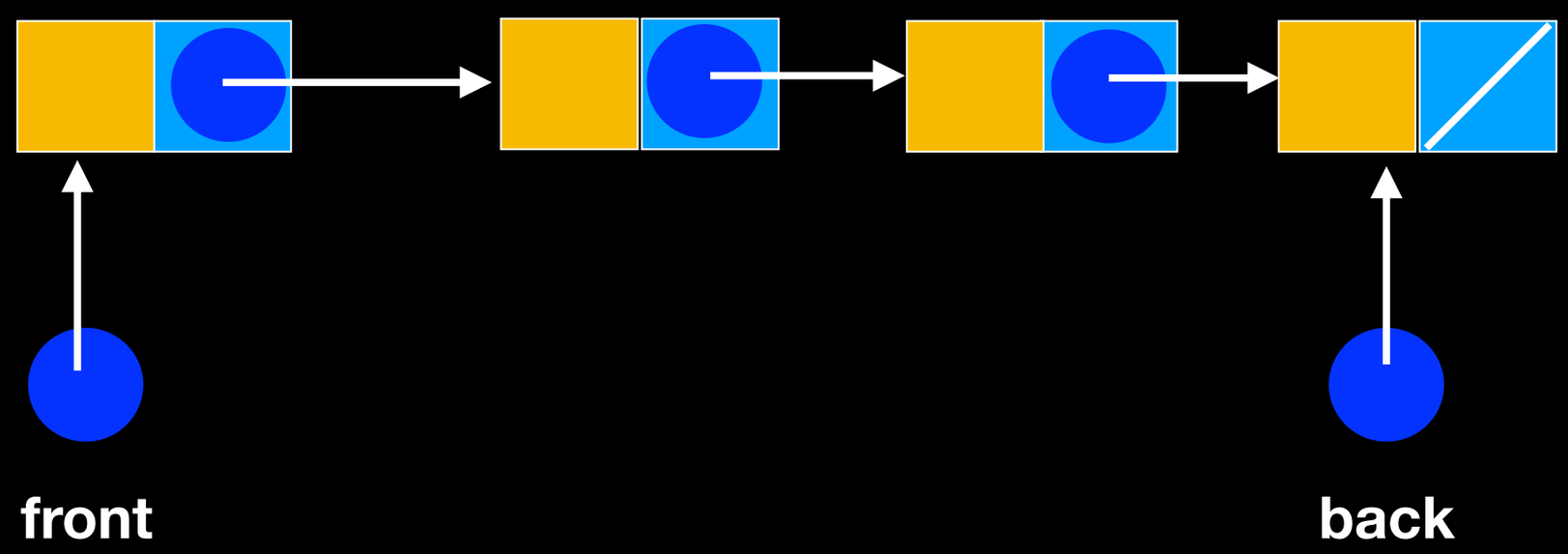


Singly Linked Chain

Deleting here is not O(1)
Because we don't have
pointer to previous node

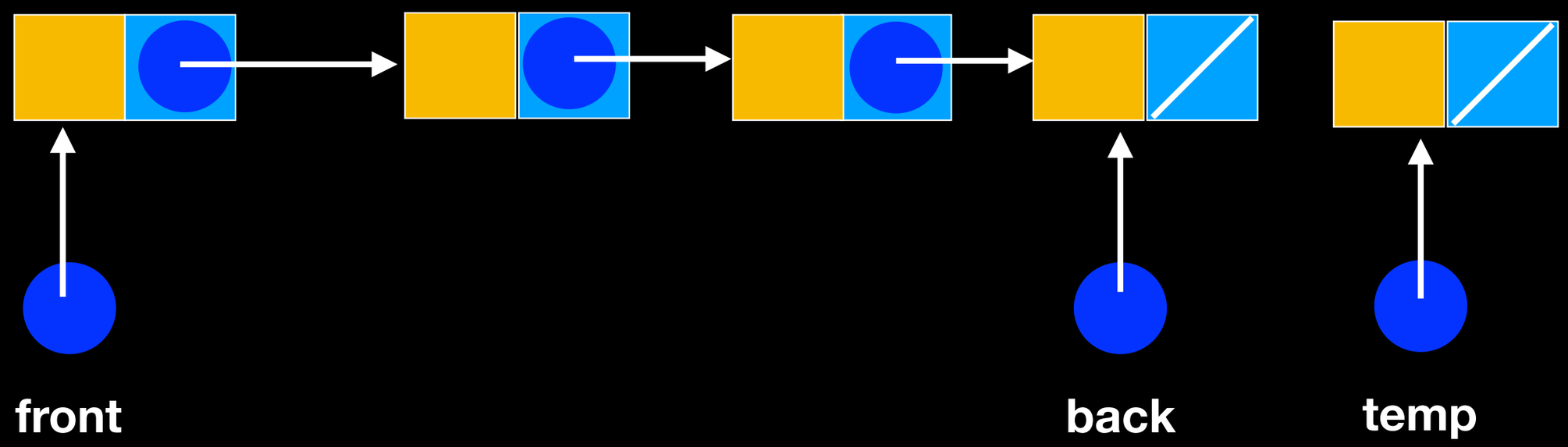


Singly Linked Chain

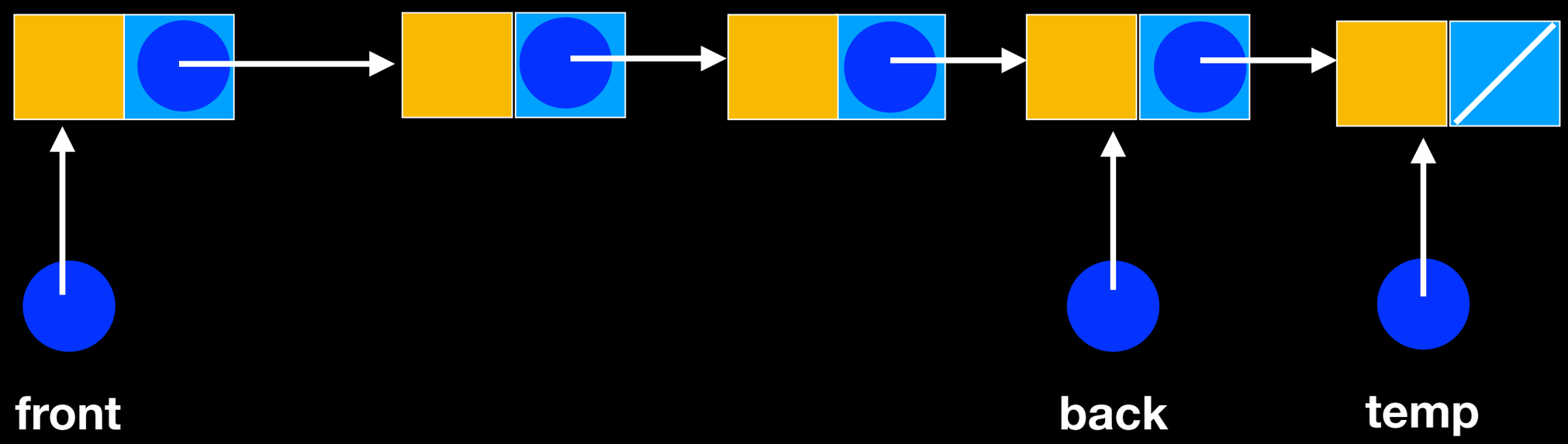


Singly Linked Chain

enqueue

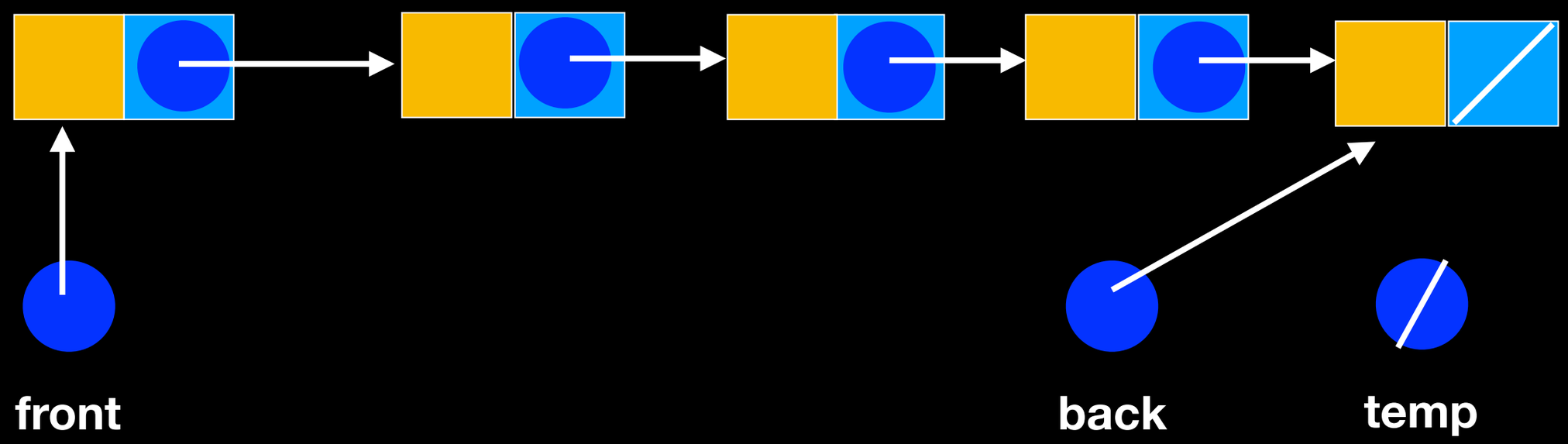


enqueue



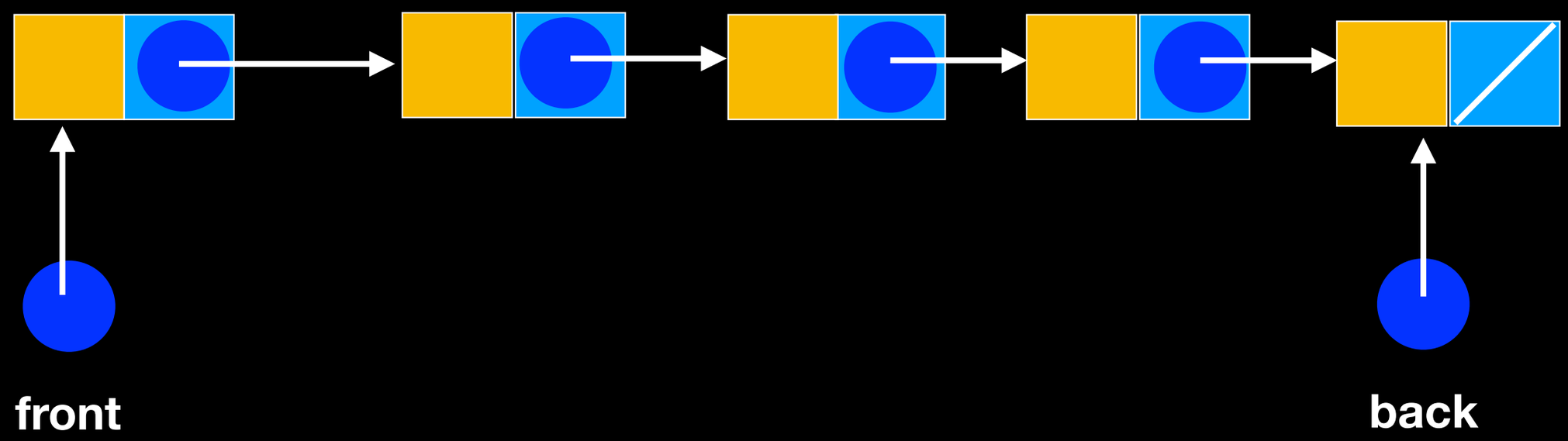
Singly Linked Chain

enqueue



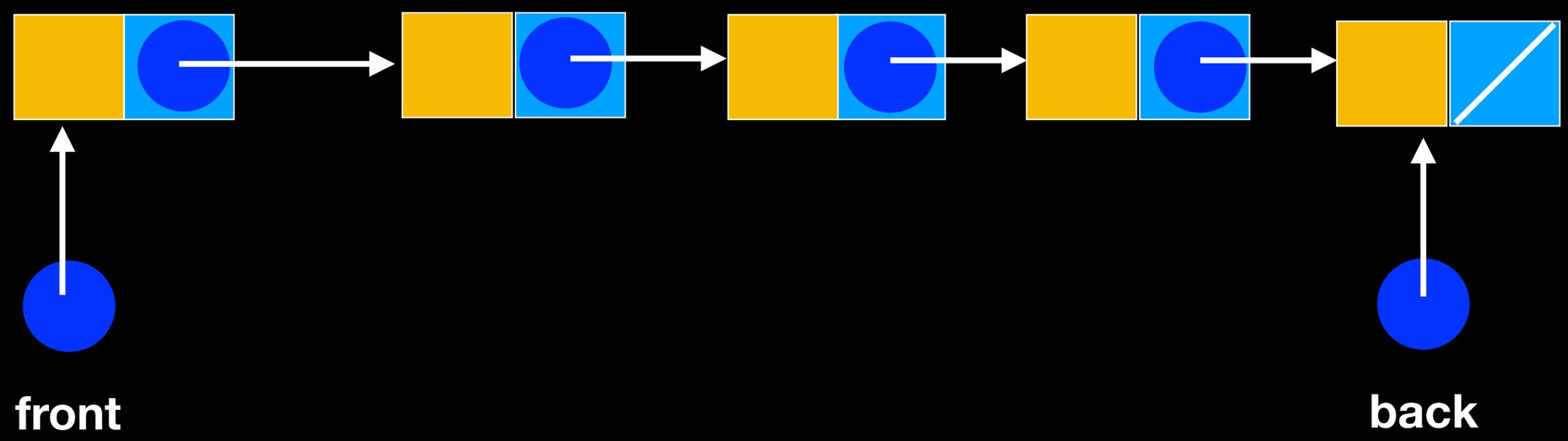
Singly Linked Chain

enqueue

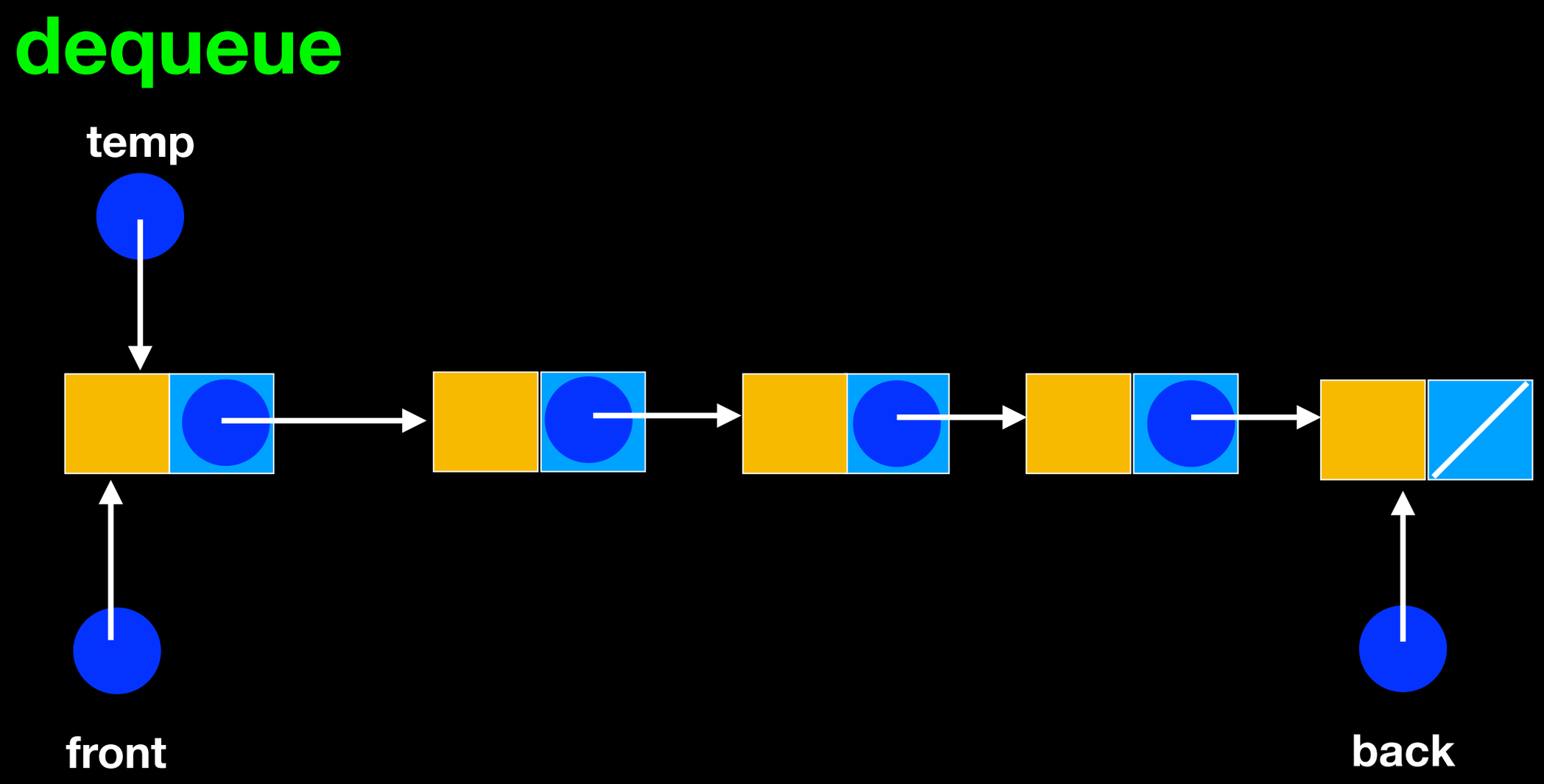


Singly Linked Chain

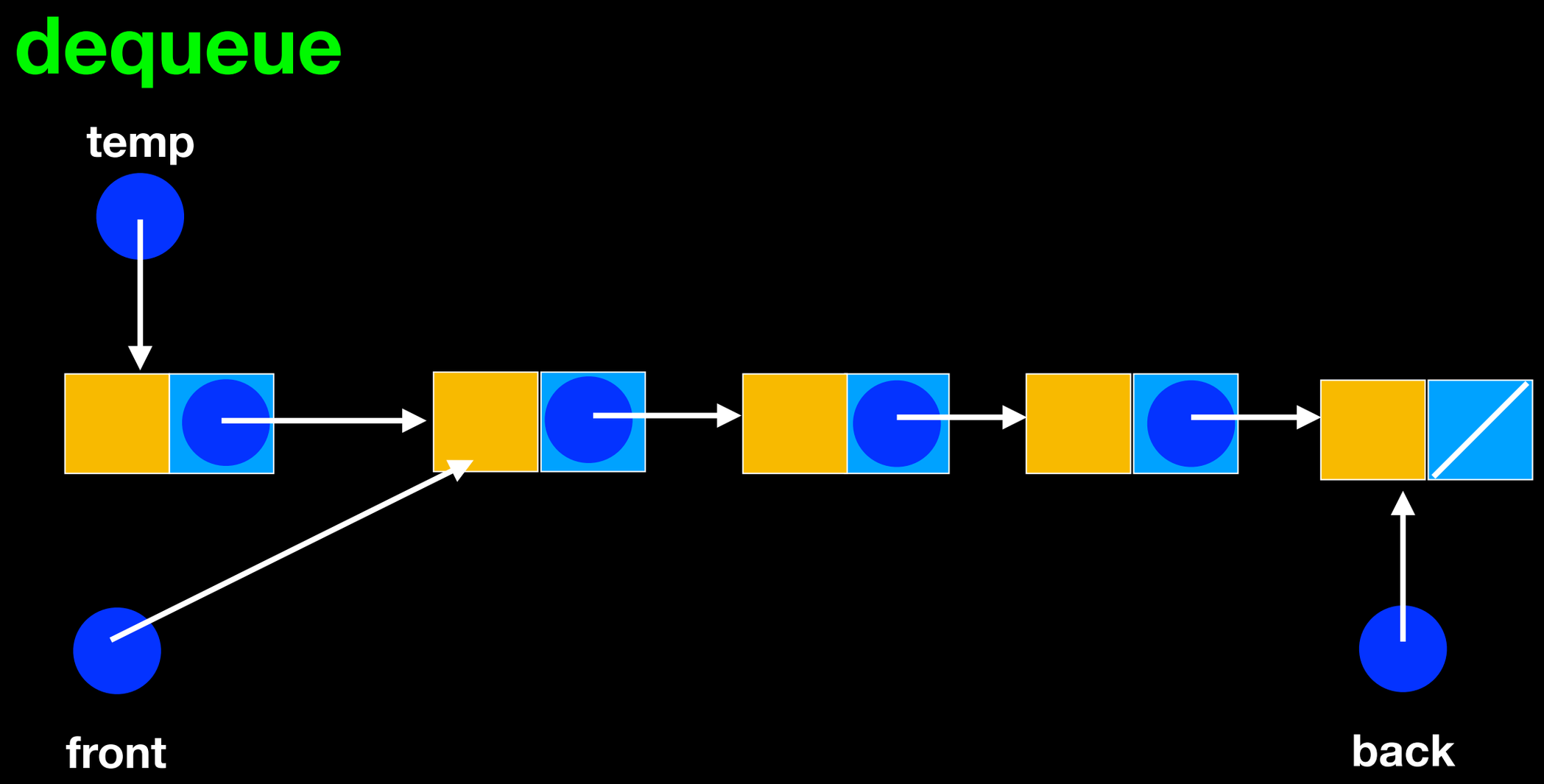
dequeue



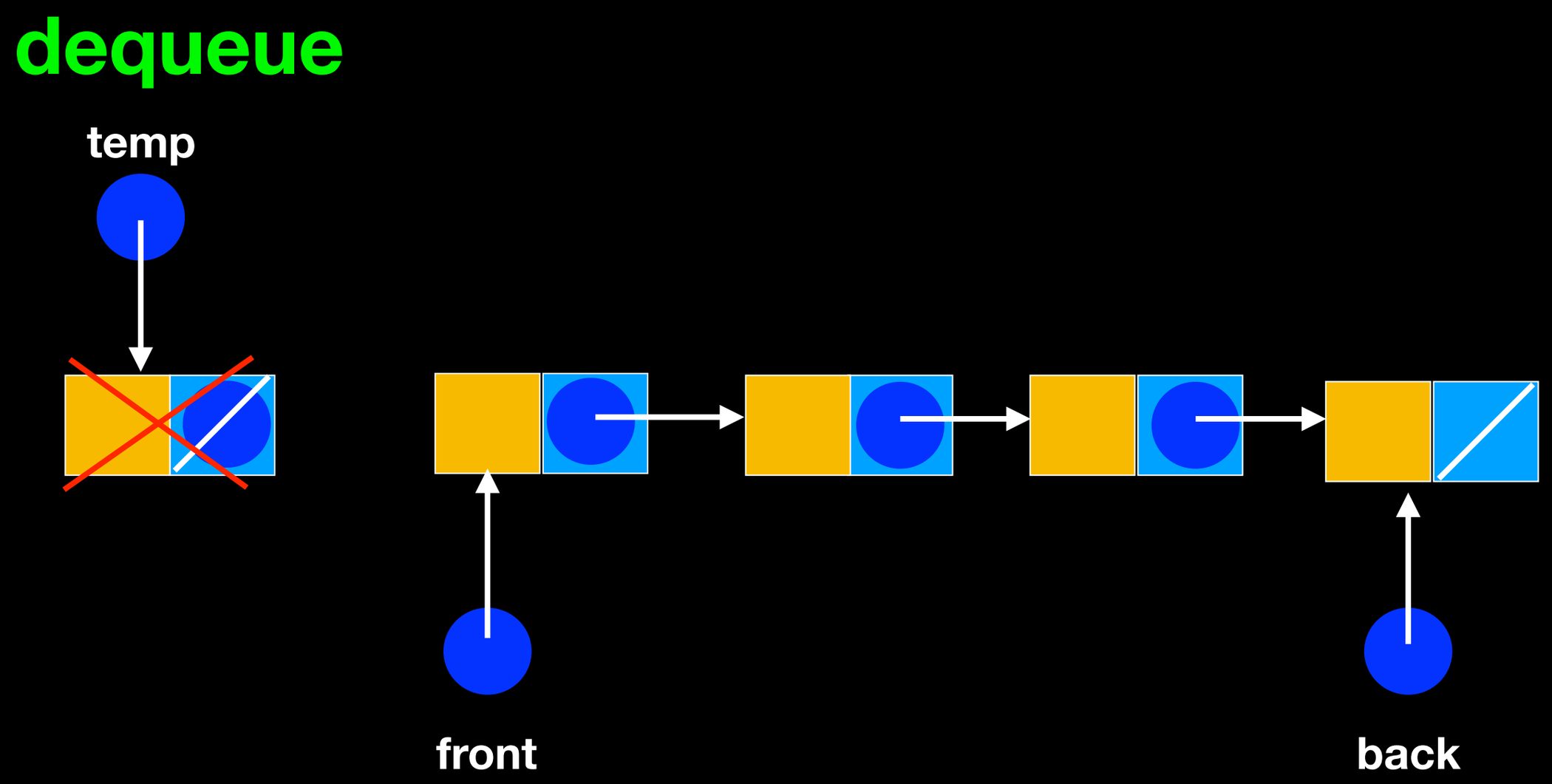
Singly Linked Chain



Singly Linked Chain

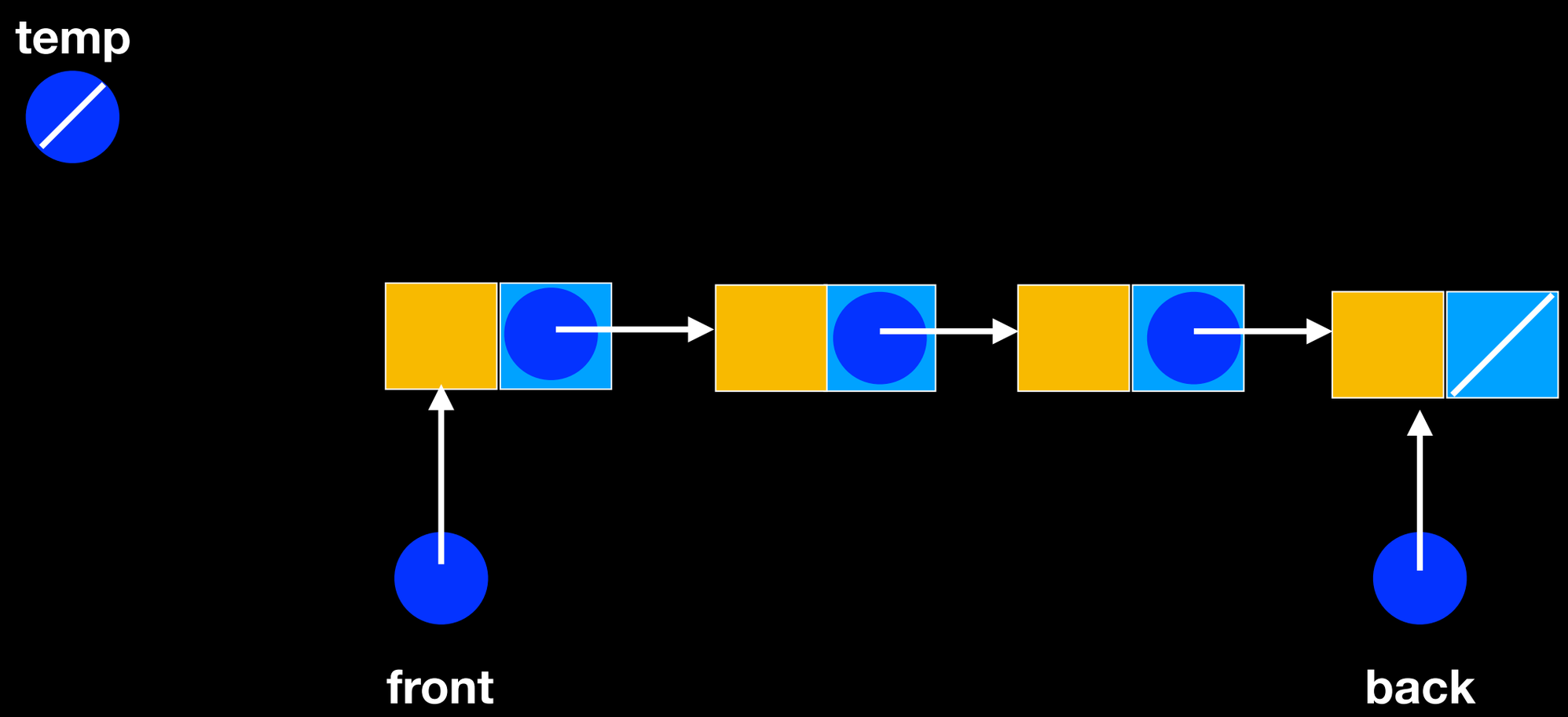


Singly Linked Chain

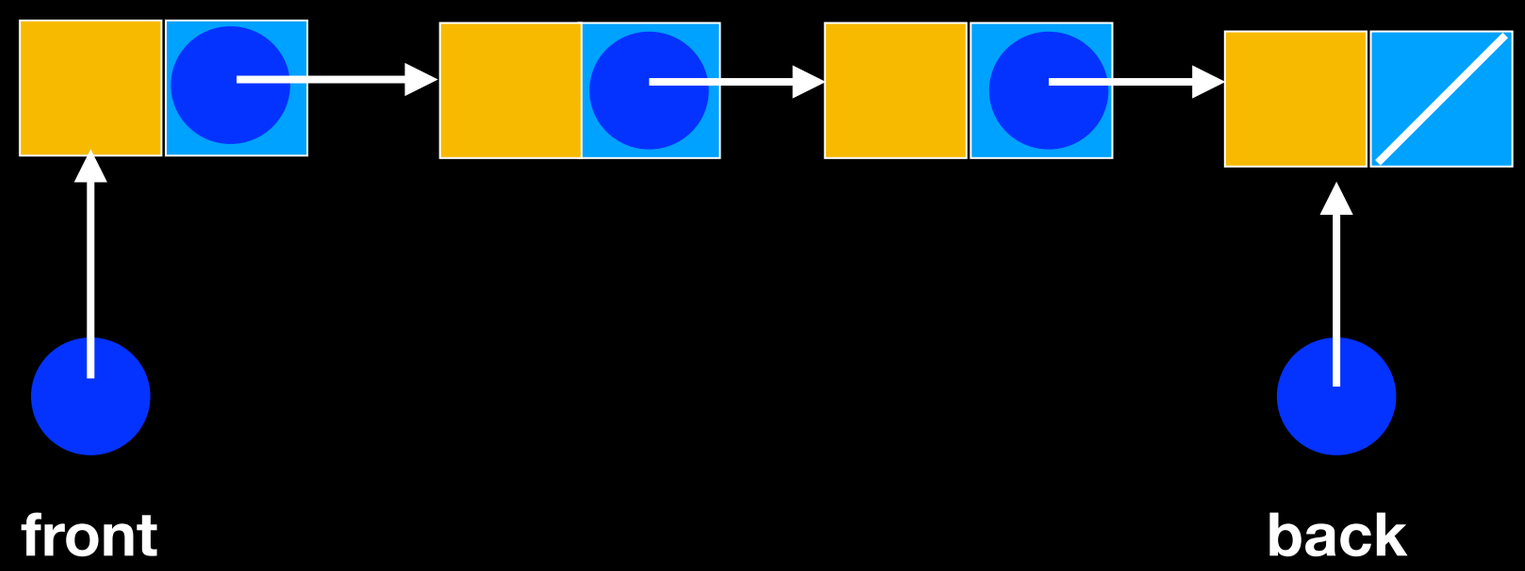


Singly Linked Chain

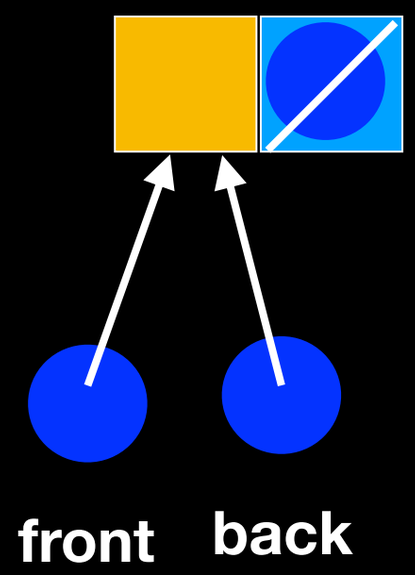
dequeue



Singly Linked Chain

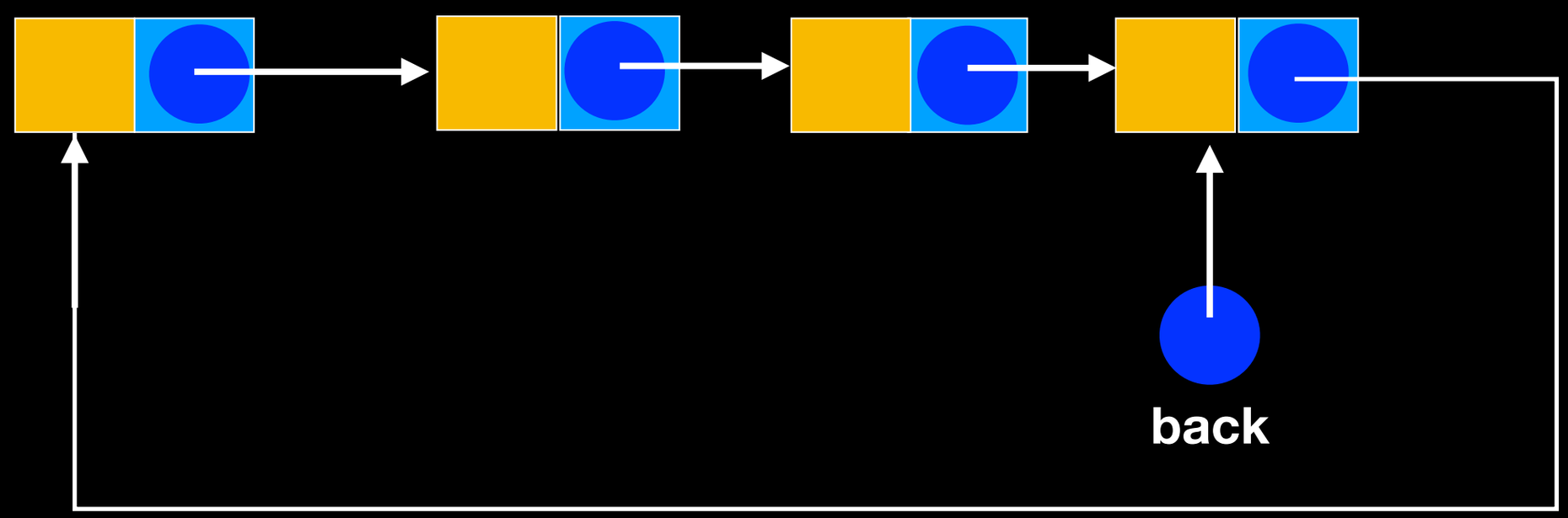


Singly Linked Chain



Singly Linked Chain

An Alternative:
A Circular Linked Chain



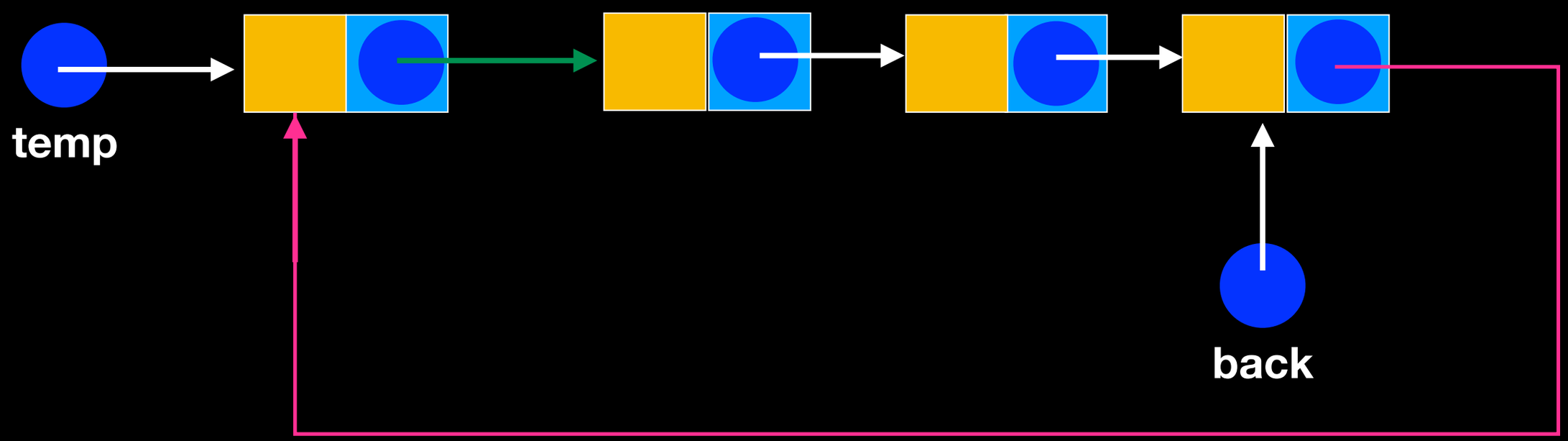
Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

To dequeue yo must repoint

```
back->setNext(back->getNext()->getNext())
```



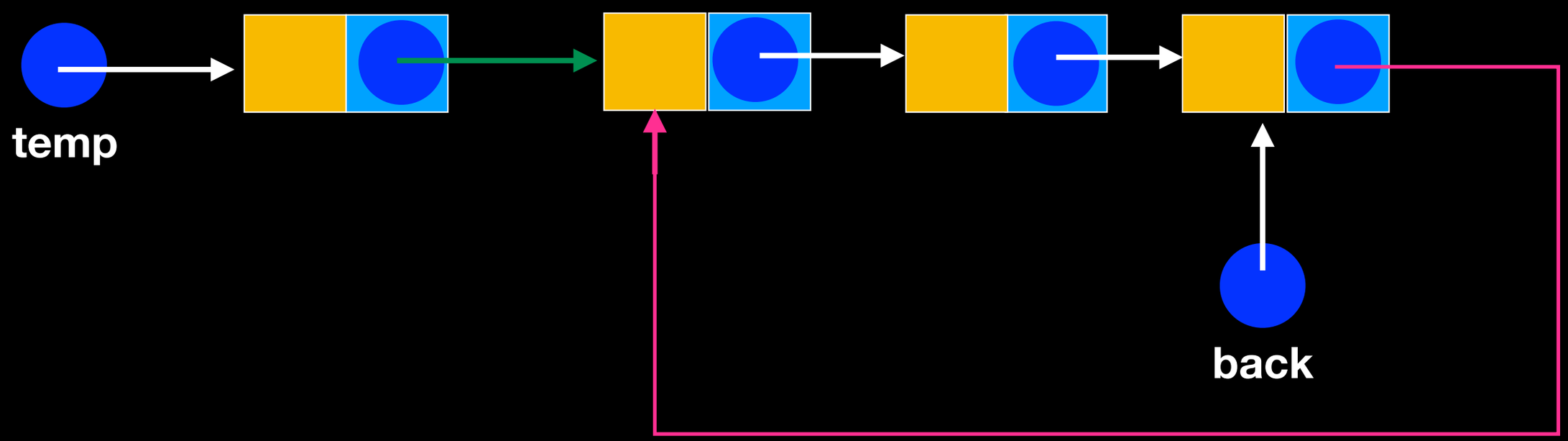
Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

To dequeue yo must repoint

```
back->setNext(back->getNext()->getNext())
```

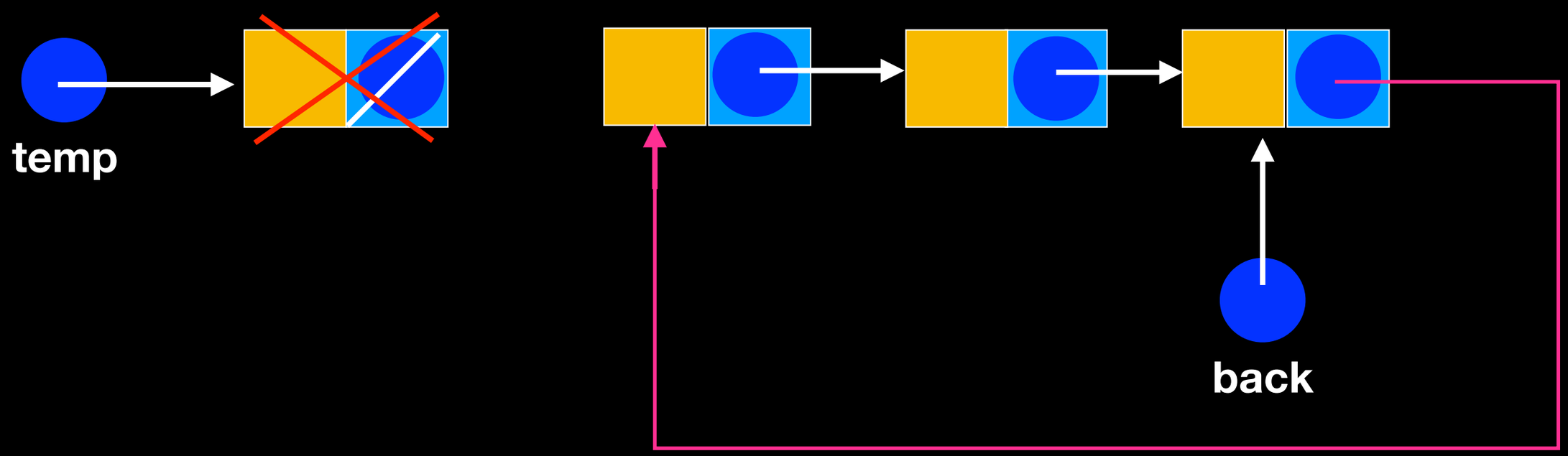


Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

`back->getNext()` **is the** `front` **pointer!**

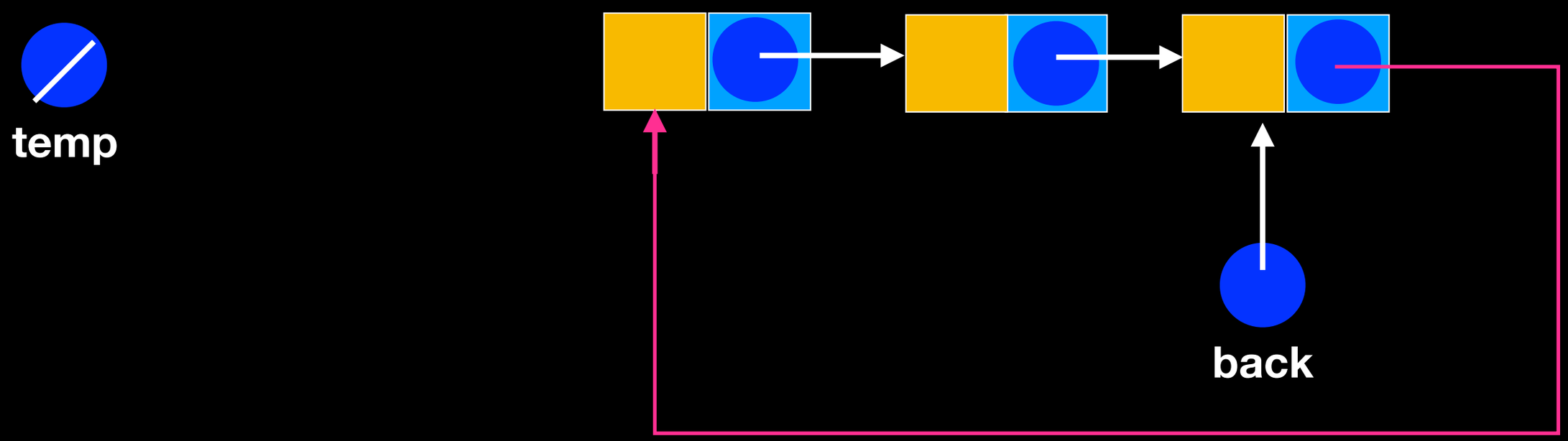


Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

`back->getNext()` **is the** `front` **pointer!**



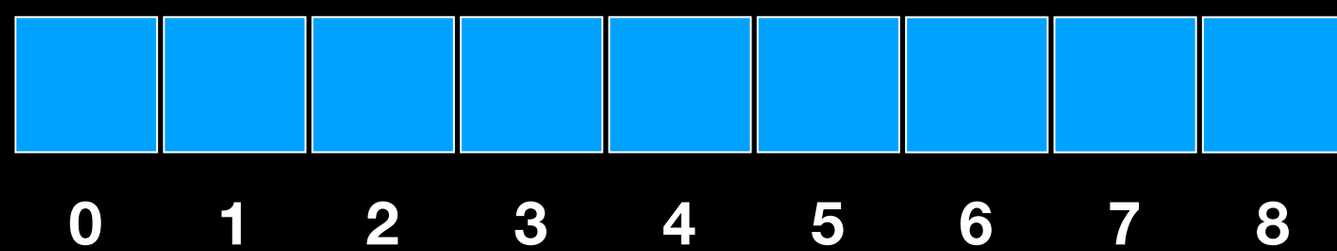
Queue ADT
(Circular Linked Chain)

```
#ifndef QUEUE_H_
#define QUEUE_H_

template<class ItemType>
class Queue
{

public:
    Queue();
    Queue(const Queue<ItemType>& aQueue); // Copy constructor
    ~Queue();
    void enqueue(const ItemType& newEntry); // adds an element to back queue
    void dequeue(); // removes element from front of queue
    ItemType front() const; // returns a copy of element at the front of queue
    int size() const; // returns the number of elements in the queue
    bool isEmpty() const; // returns true if no elements in queue,false otherwise

private:
   Node<ItemType>* back_; // Pointer to back of queue
    int itemCount;        // number of items currently on the stack

};    //end Queue

#include "Queue.cpp"
#endif // QUEUE_H_ `
```

How would you implement it using an array?
enqueue and dequeue in O(1)

Array Considerations

```
front = 0
back = -1
```

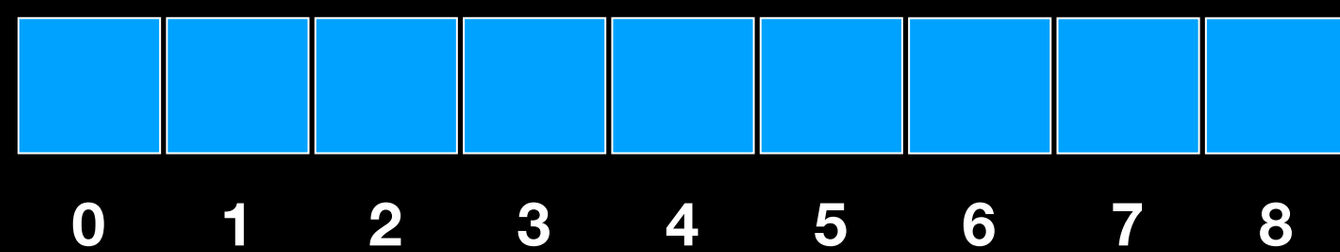


Array Considerations

enqueue

Increment `back` and add element to `items_[back]`

```
front = 0
back = -1
```



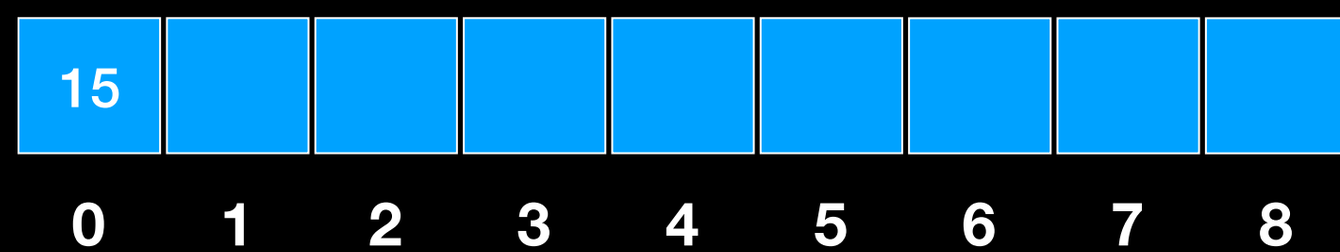
Array Considerations

enqueue

Increment `back` and add element to `items_[back]`

`front = 0`

`back = 0`



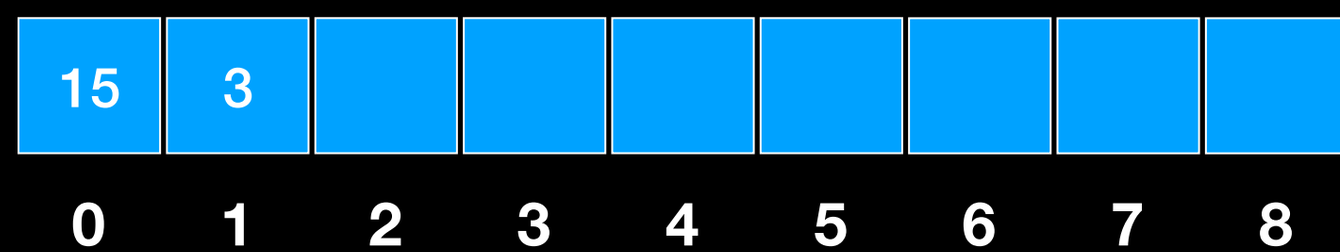
Array Considerations

enqueue

Increment `back` and add element to `items_[back]`

`front = 0`

`back = 1`



Array Considerations

enqueue

Increment `back` and add element to `items_[back]`

`front = 0`

`back = 2`

0	1	2	3	4	5	6	7	8
15	3	45						

Array Considerations

enqueue

Increment `back` and add element to `items_[back]`

`front = 0`

`back = 5`

15	3	45	13	75	84			
0	1	2	3	4	5	6	7	8

This seems to work, but what happens when we start dequeuing?

Array Considerations

dequeue

Increment `front`

`front = 1`

`back = 5`

15	3	45	13	75	84			
0	1	2	3	4	5	6	7	8

We want O(1) operations, so simply increment front!

Array Considerations

dequeue

Increment `front`

`front = 2`

`back = 5`

0	1	2	3	4	5	6	7	8
15	3	45	13	75	84			

Array Considerations

```
front = 3
back = 5
```

15	3	45	13	75	84	55	38	97
0	1	2	3	4	5	6	7	8

RIGHTWARD DRIFT!!!
At some point queue will be full even if it contains only a few elements

Array Considerations

```
front = 3
back = 5
```

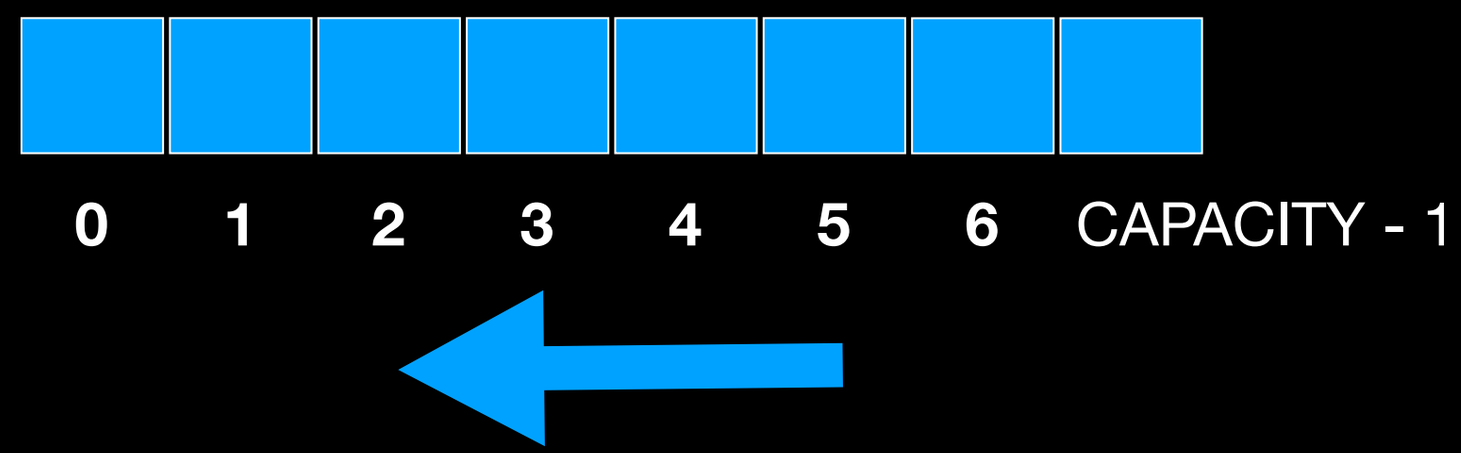
15	3	45	13	75	84	55	38	97
0	1	2	3	4	5	6	7	8

RIGHTWARD DRIFT!!!
At some point queue will be full even if it contains only a few elements

No Good

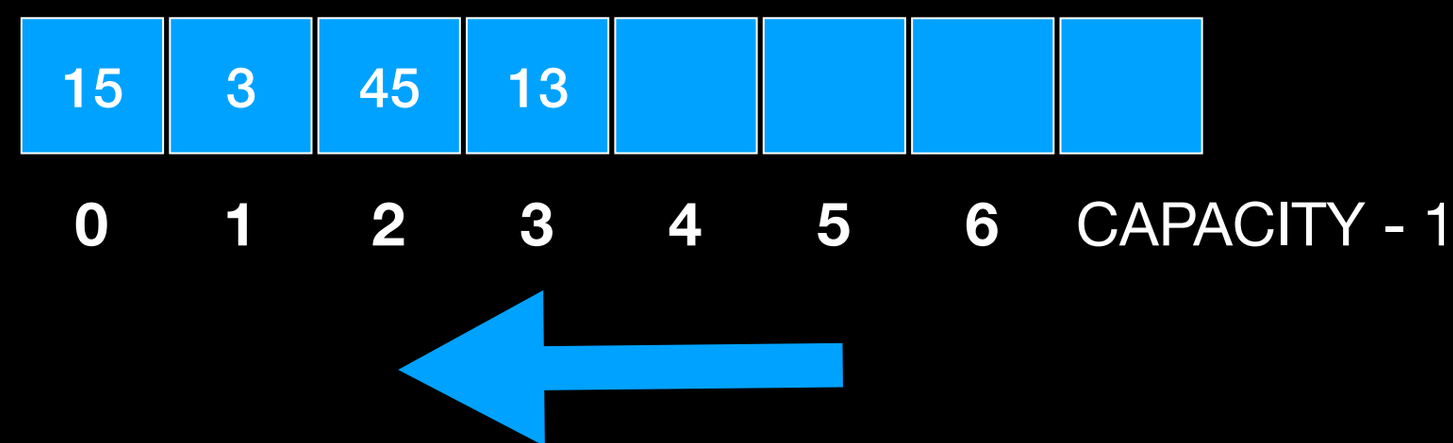
Circular Array Implementation

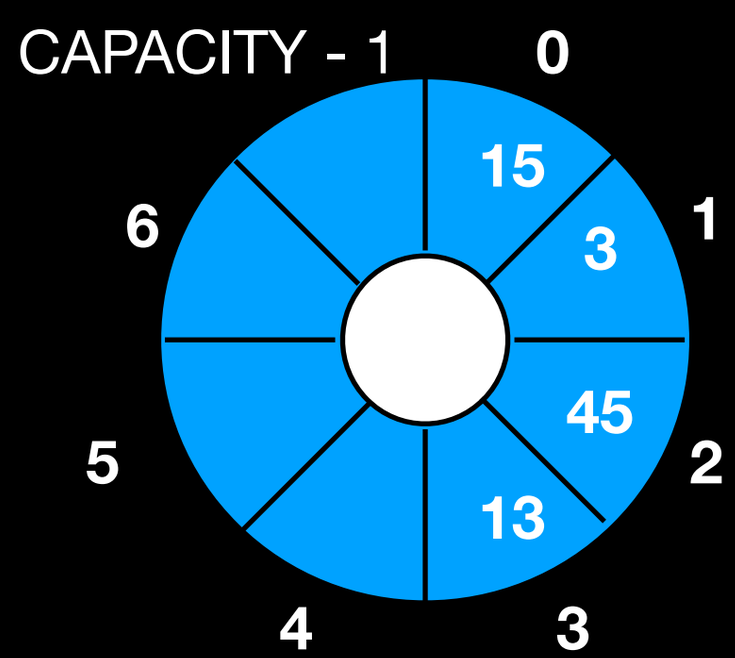
```
front = 0
back = -1
```



Circular Array Implementation

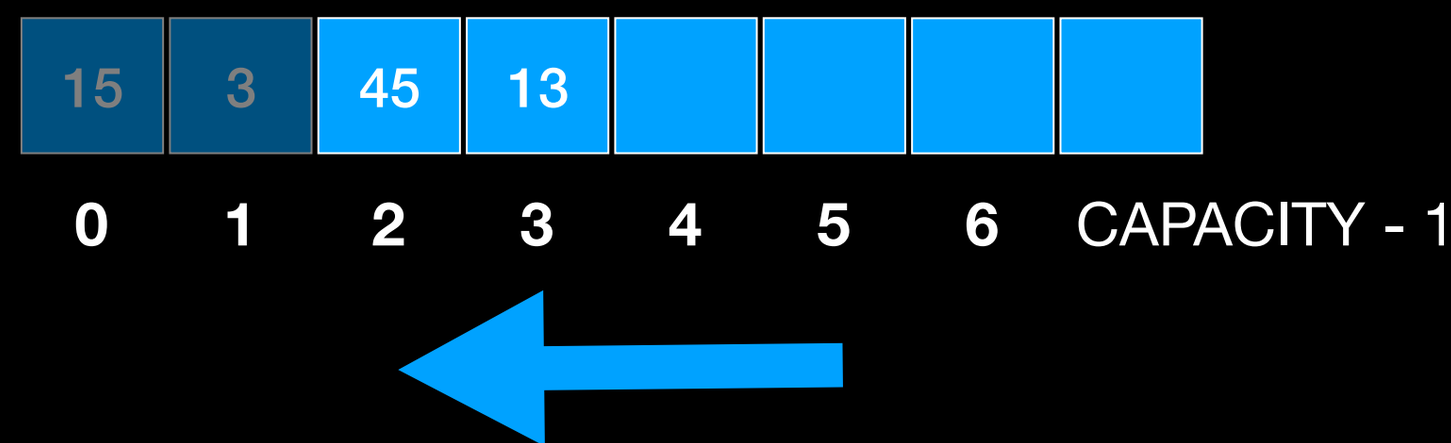
```
front = 0
back = 3
```

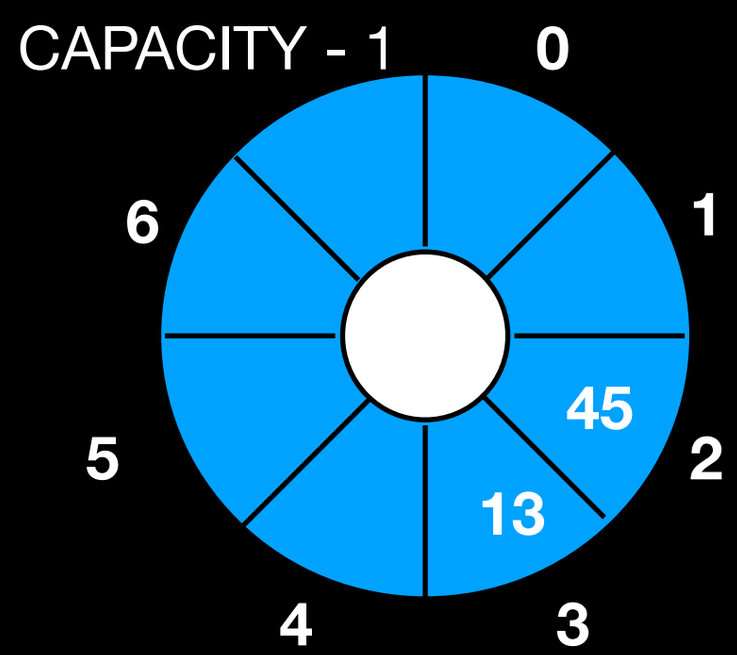




Circular Array Implementation

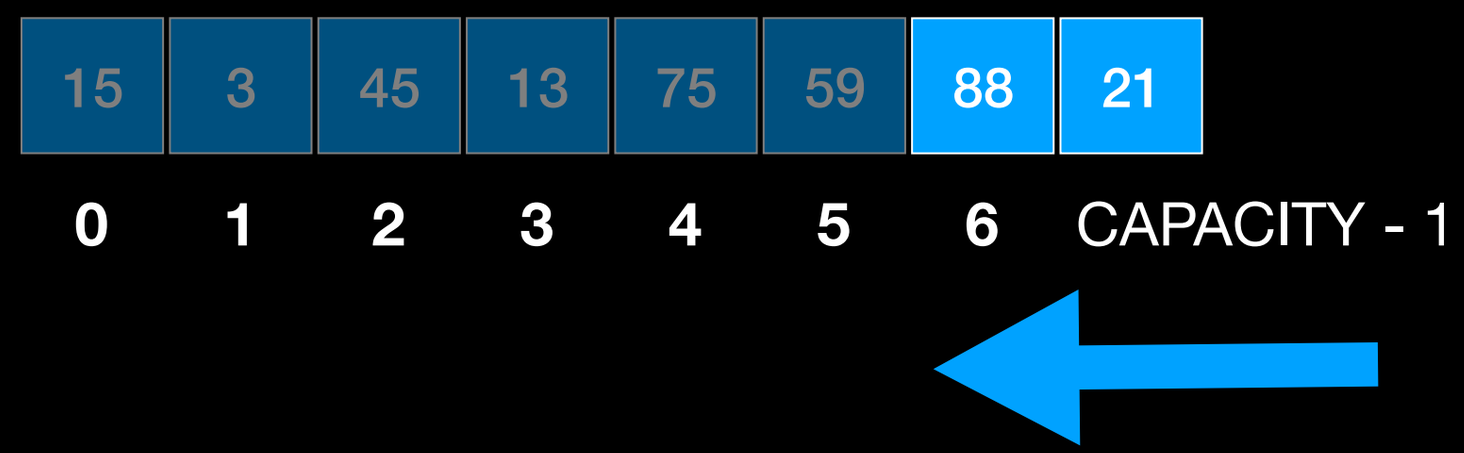
```
front = 2
back = 3
```

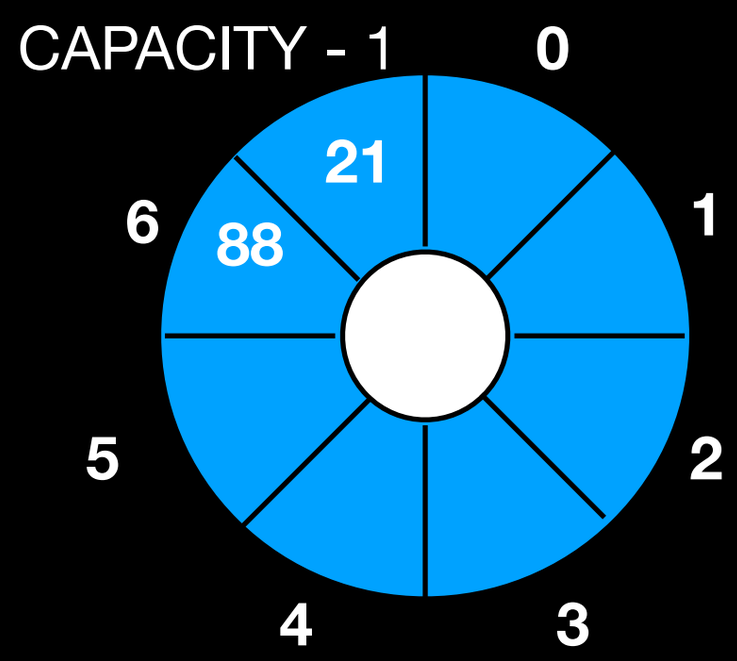




Circular Array Implementation

```
front = 6
back = CAPACITY - 1
```

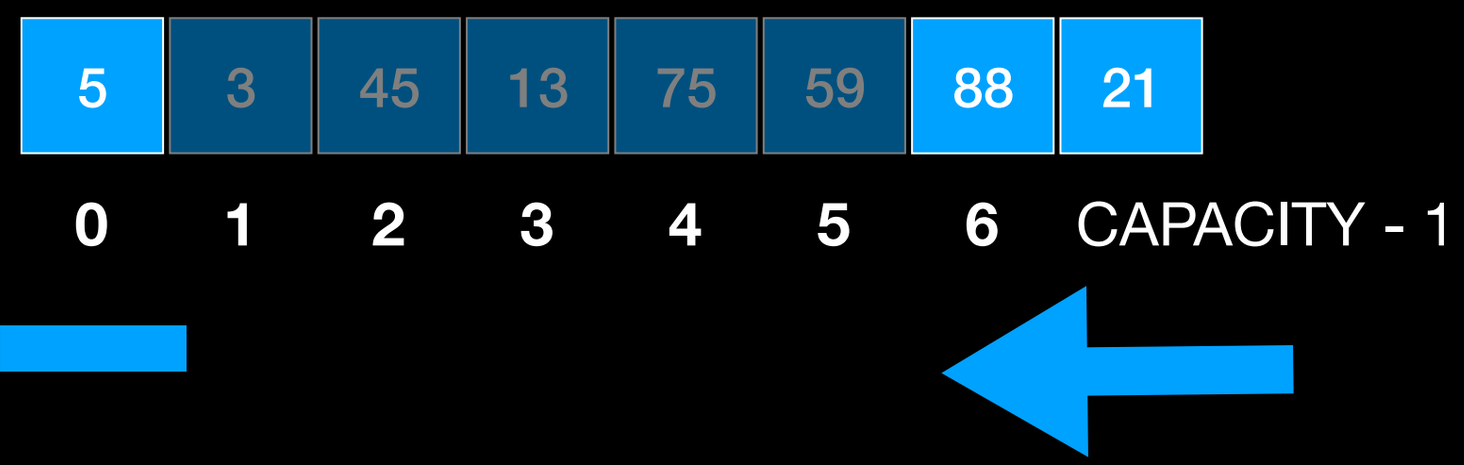


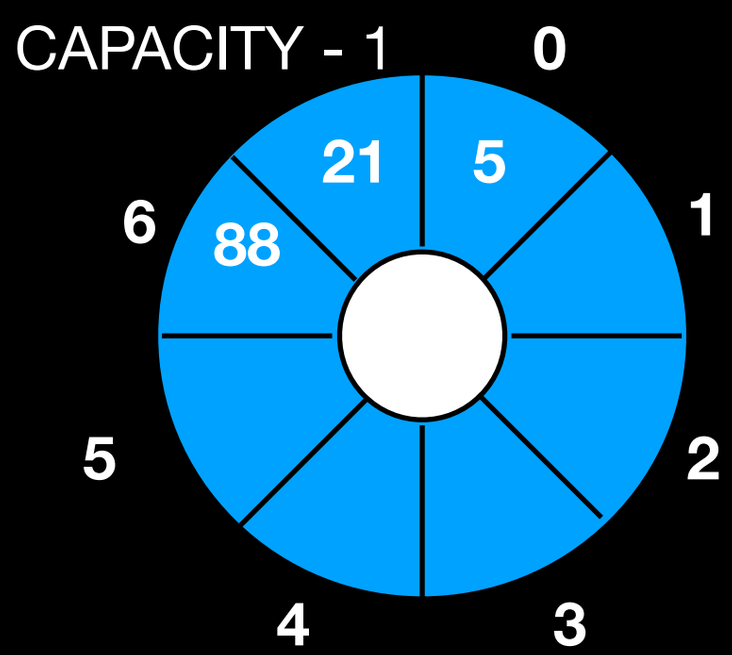


Circular Array Implementation

front = 6

back = 0





WRAP AROUND USING MODULO ARITHMETIC

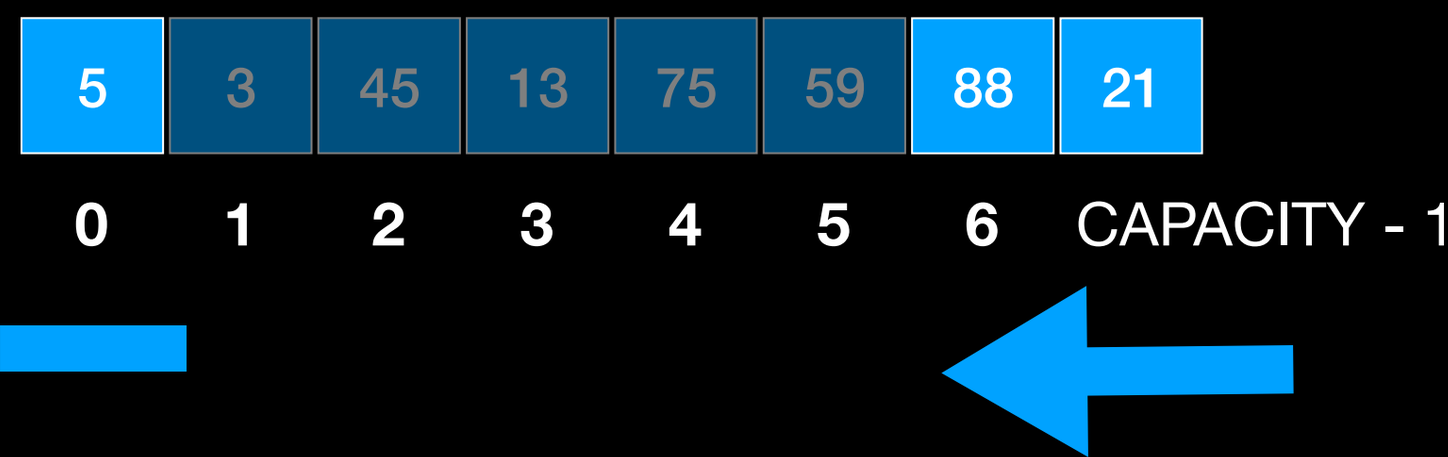
Circular Array Implementation

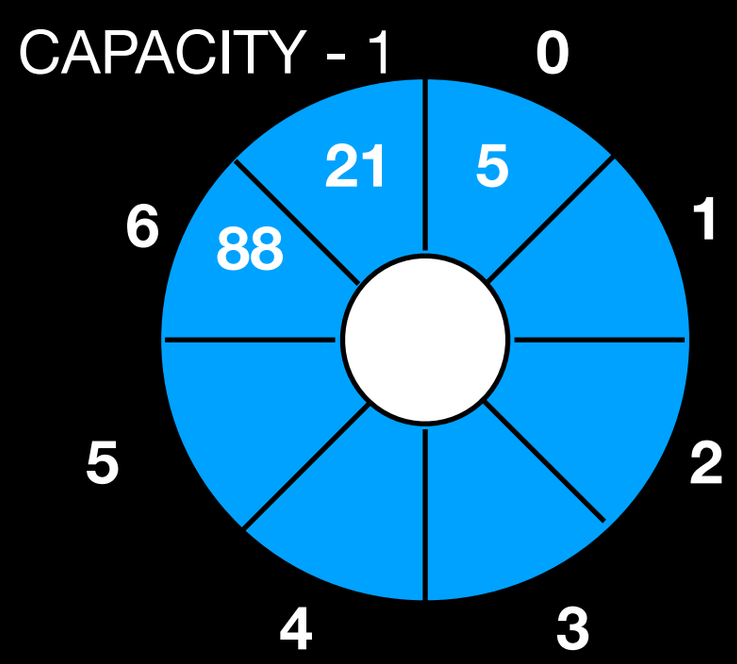
`front = 6`

`back = 0`

enqueue

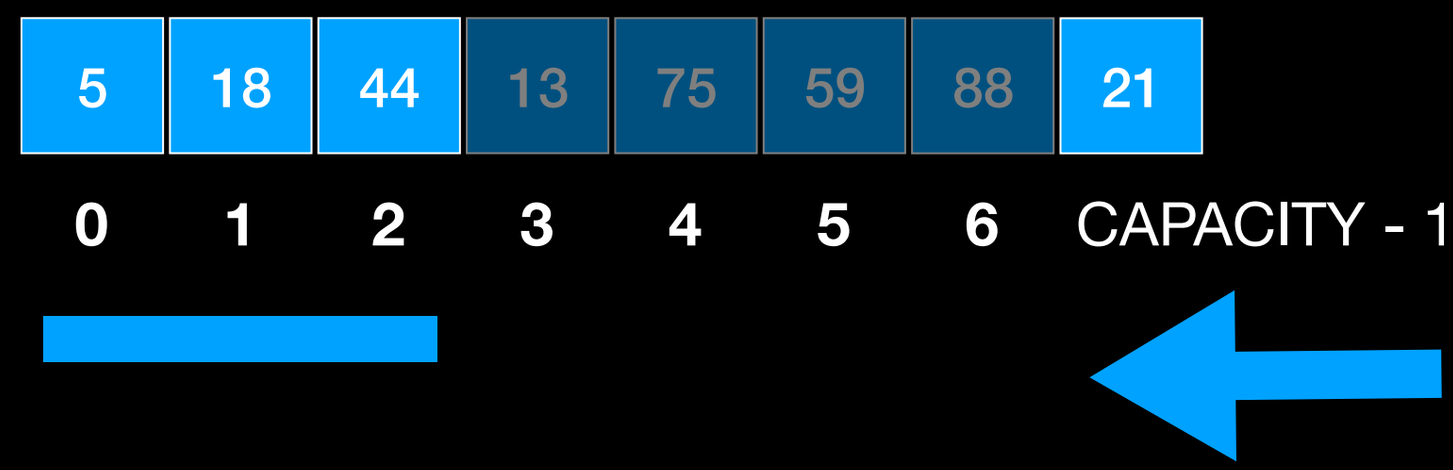
`back = (back + 1) % CAPACITY`
add element to `items_[back]`

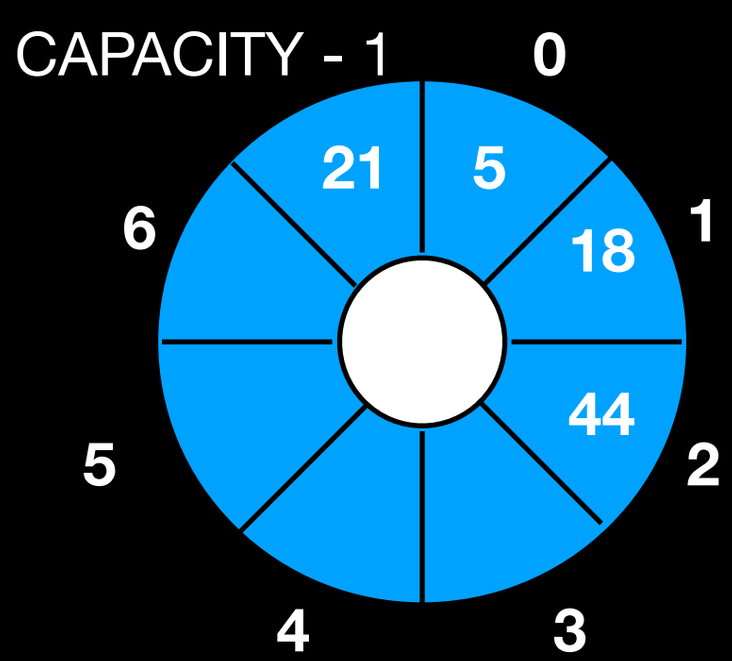




Circular Array Implementation

```
front = CAPACITY - 1
back = 2
```



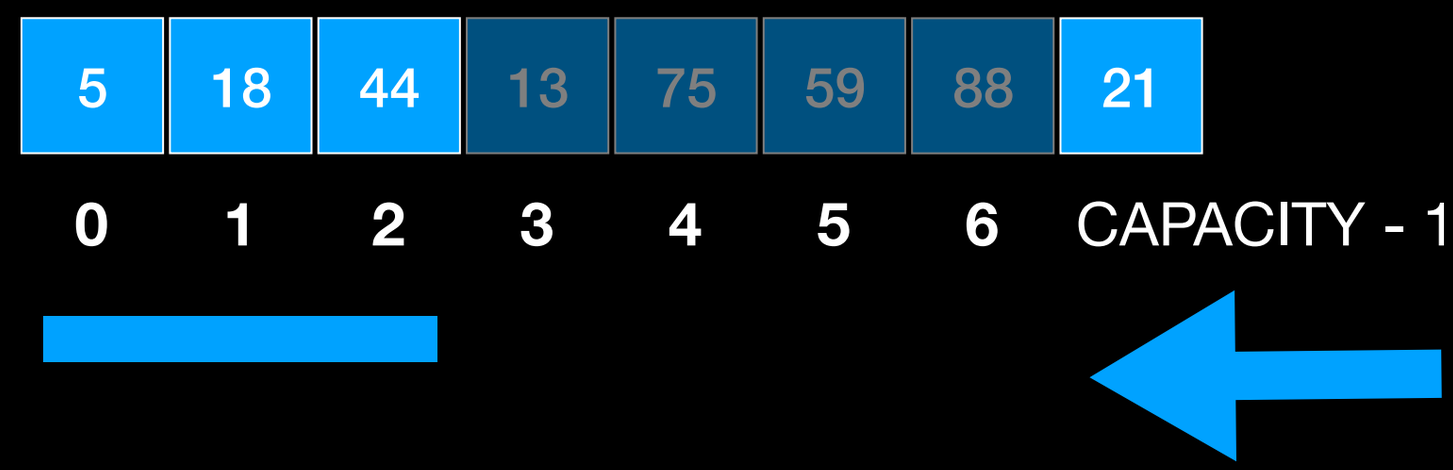


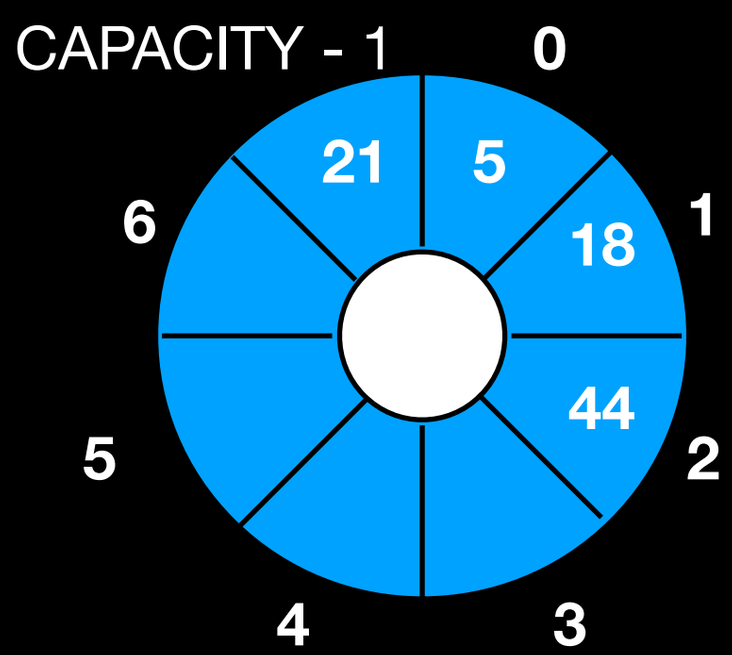
Circular Array Implementation

```
front = CAPACITY - 1
back = 2
```

dequeue

```
front = (front + 1) % CAPACITY
```





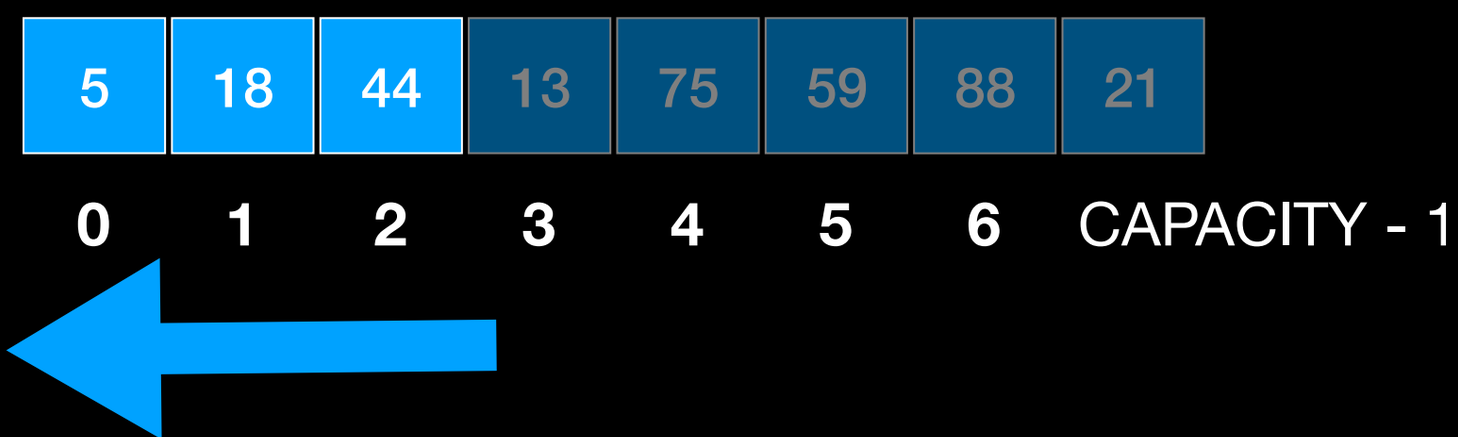
Circular Array Implementation

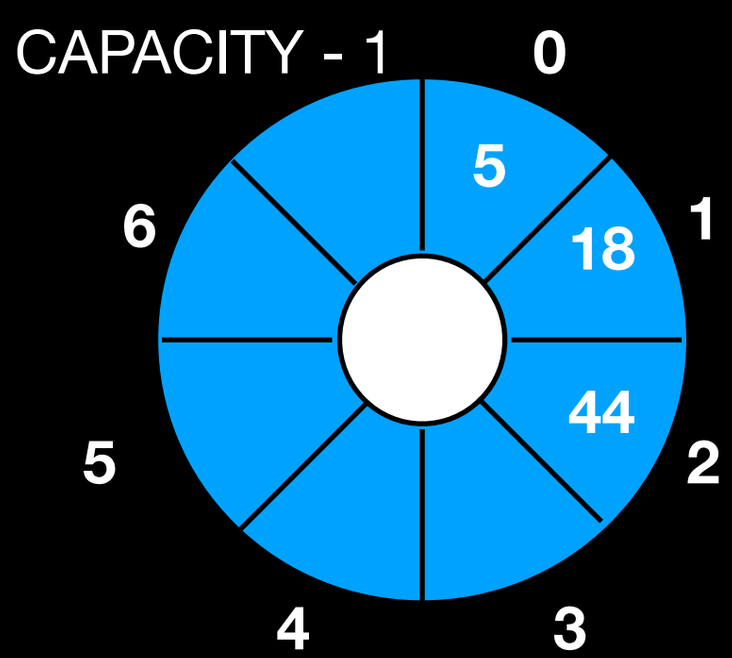
front = 0

back = 2

dequeue

```
front = (front + 1) % CAPACITY
```

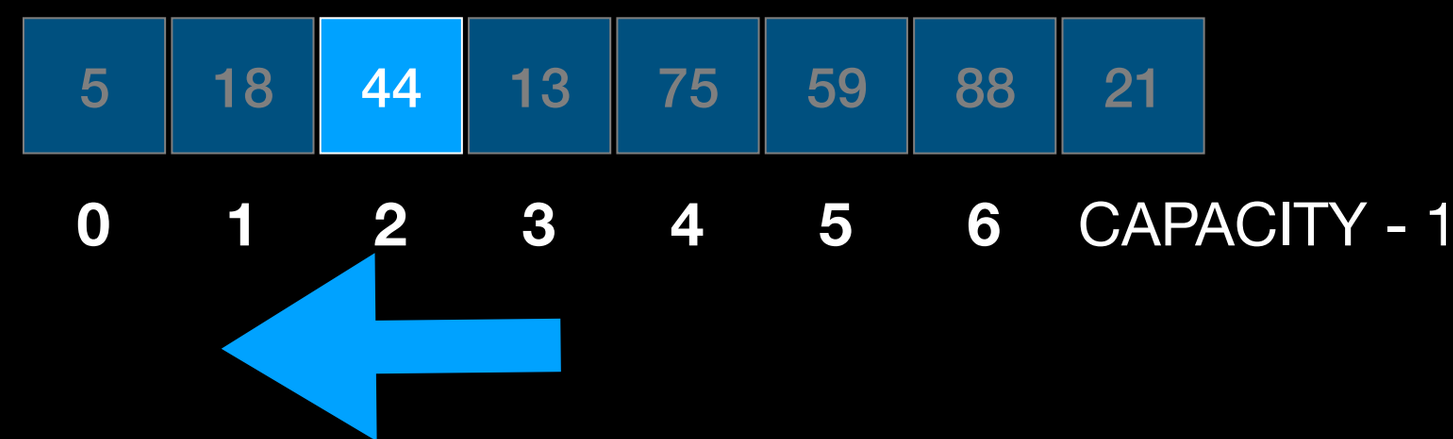


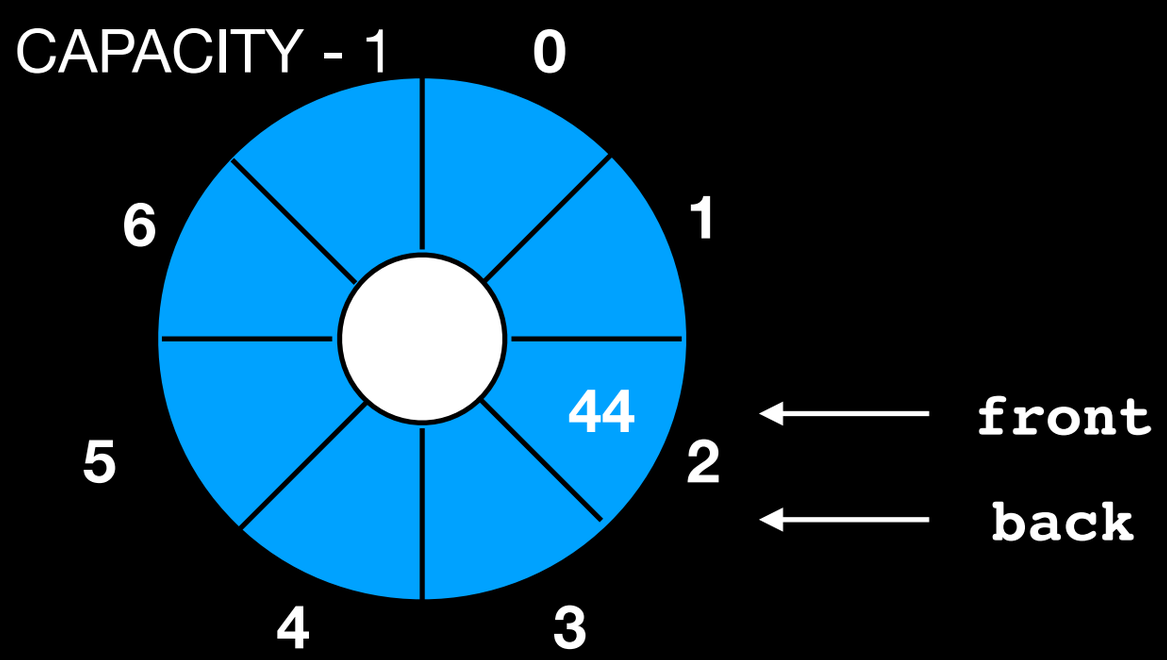


Circular Array Implementation

front = 2

back = 2





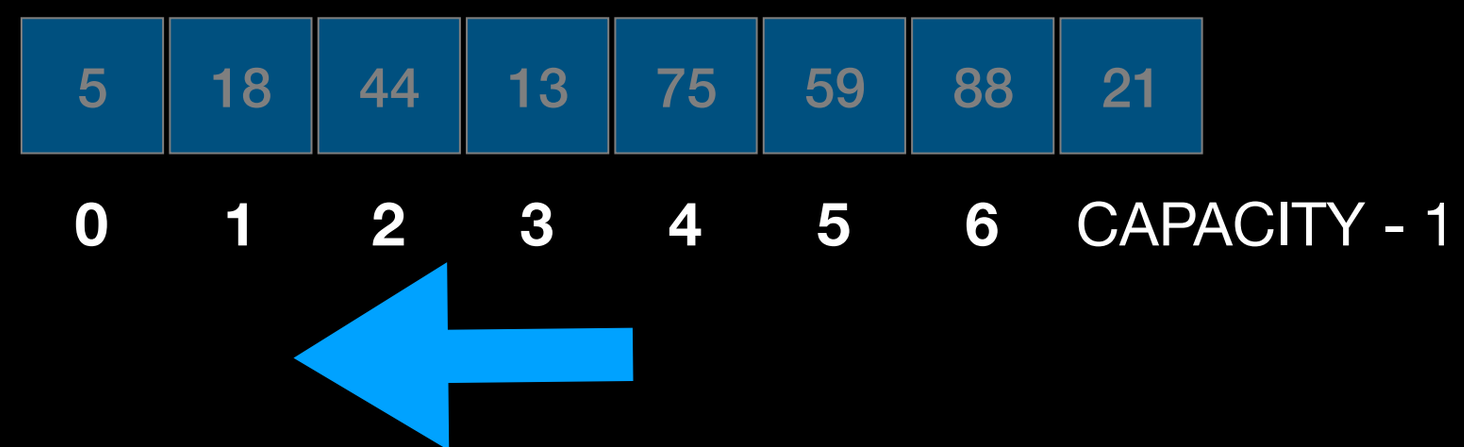
Circular Array Implementation

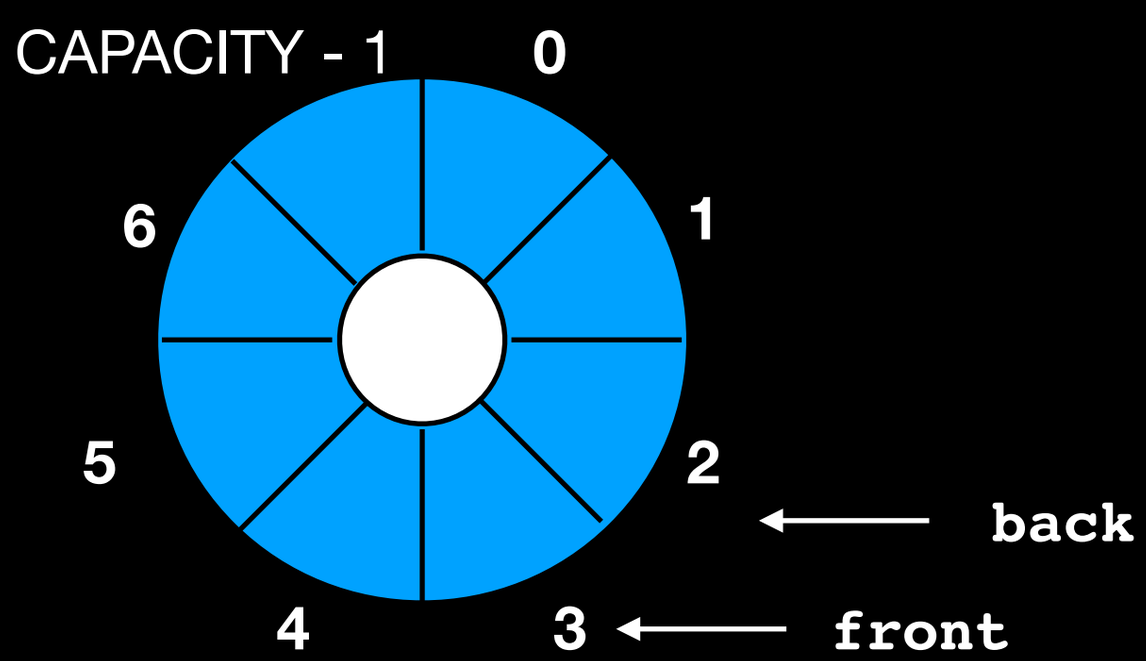
front = 3

back = 2

dequeue

```
front = (front + 1) % CAPACITY
```





`front` passes `back` when queue is EMPTY

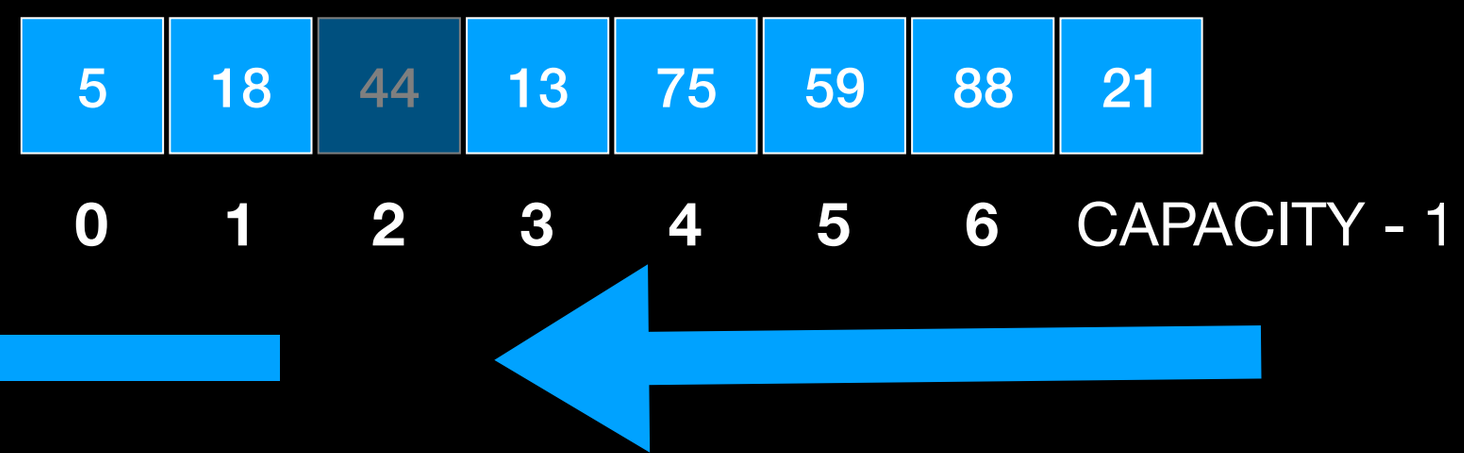
Circular Array Implementation

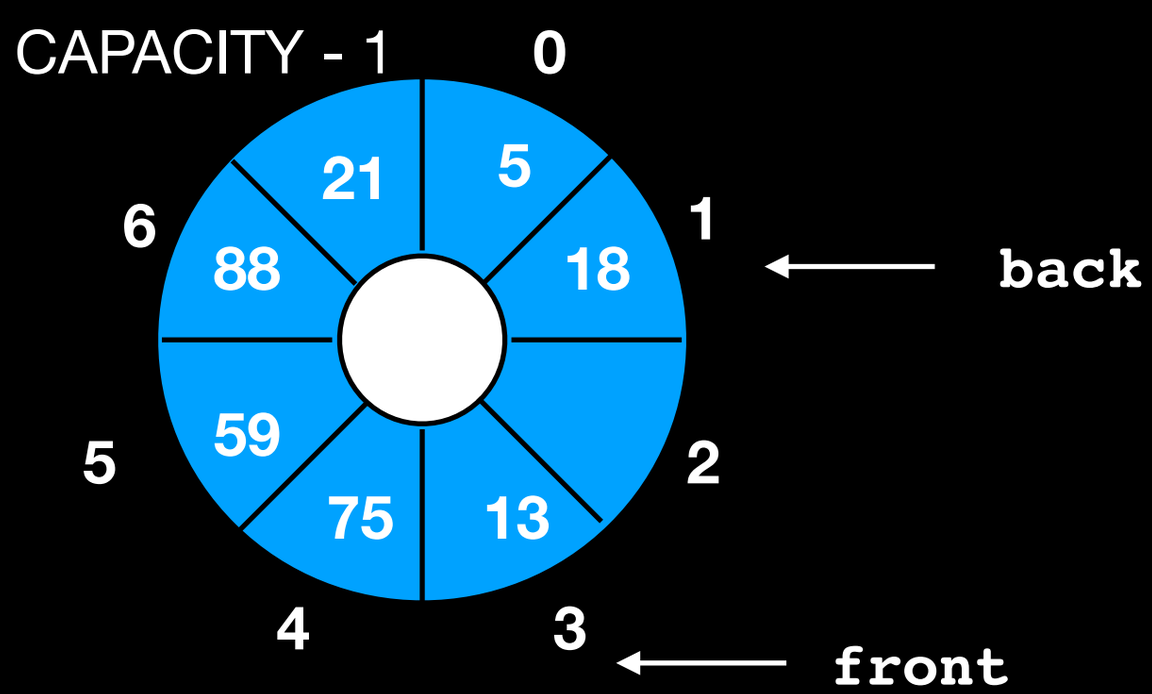
`front = 3`

`back = 1`

enqueue

`back = (back + 1) % CAPACITY`
add element to `items_[back]`





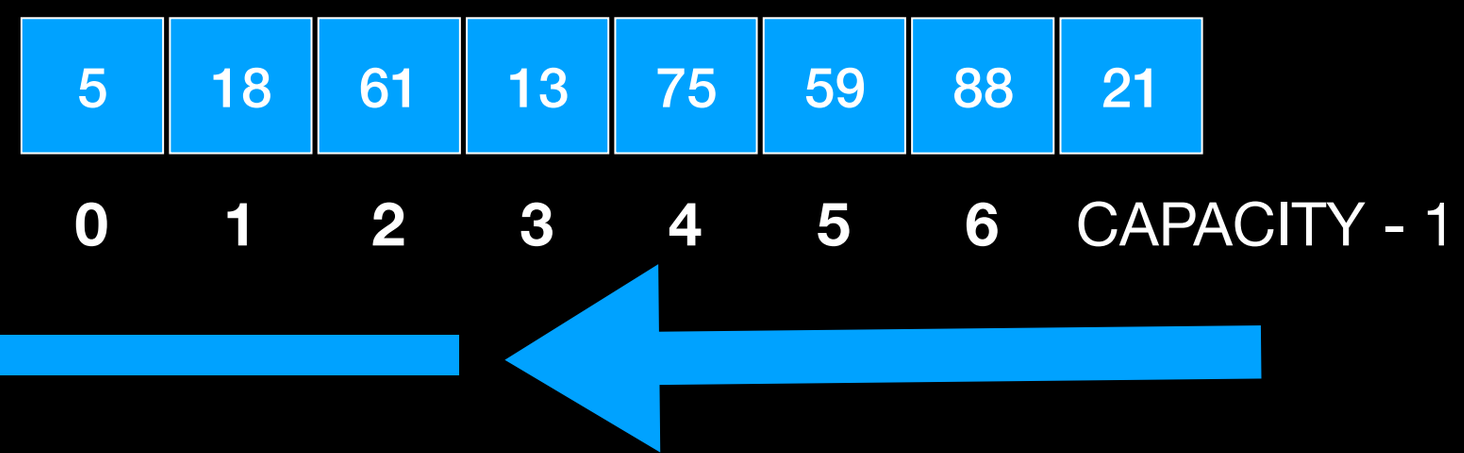
Circular Array Implementation

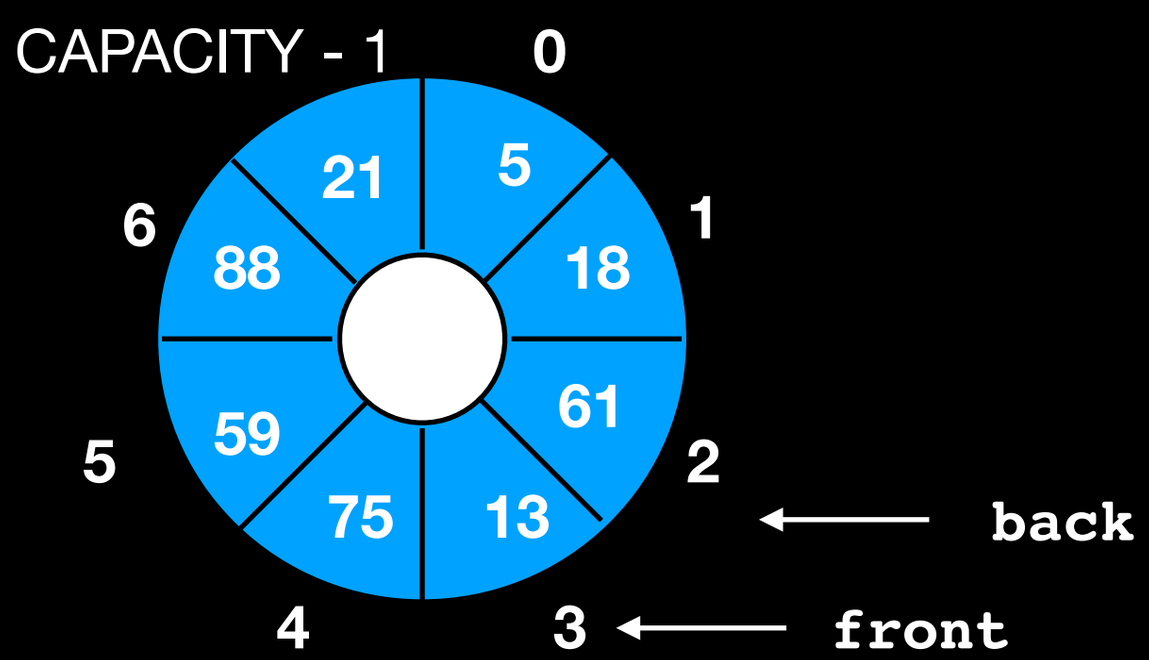
`front = 3`

`back = 2`

enqueue

`back = (back + 1) % CAPACITY`
add element to `items_[back]`





`front` passes `back` ALSO when queue is FULL

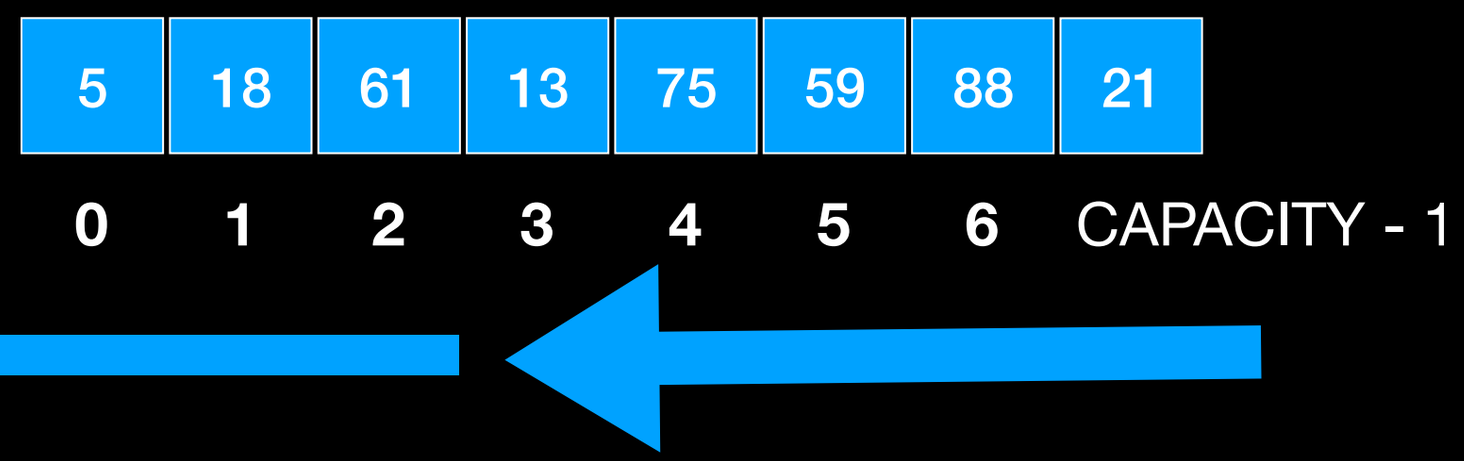
Circular Array Implementation

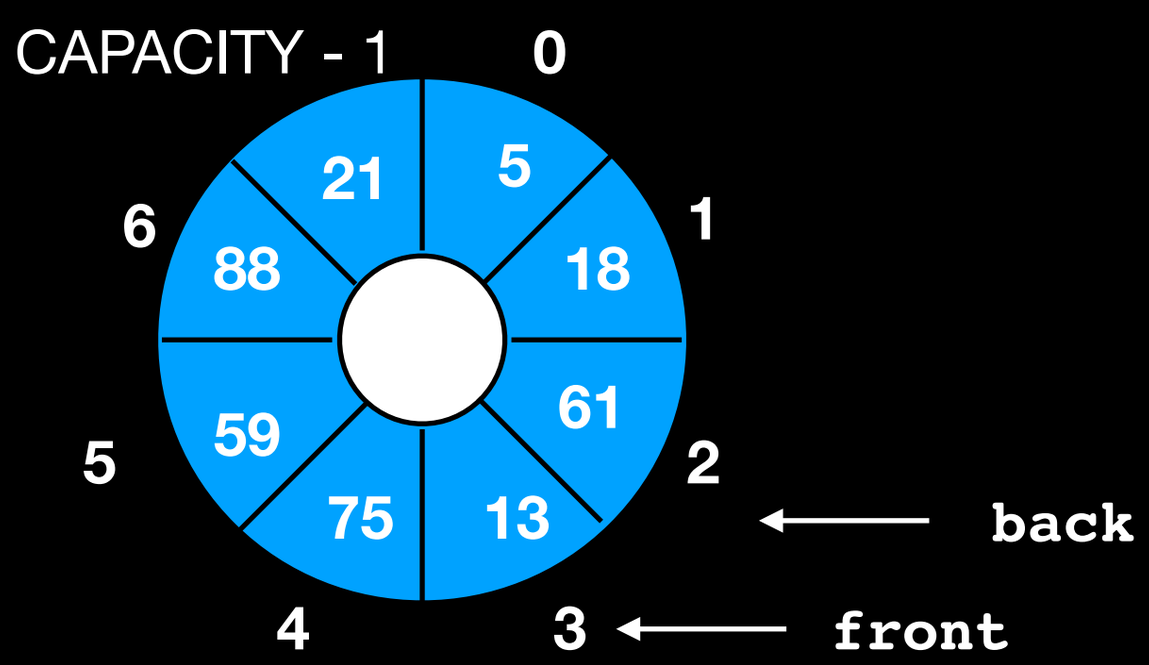
`front = 3`

`back = 2`

enqueue

`back = (back + 1) % CAPACITY`
add element to `items_[back]`





To distinguish between **empty** and **full** queue must keep a **COUNTER** for number of items

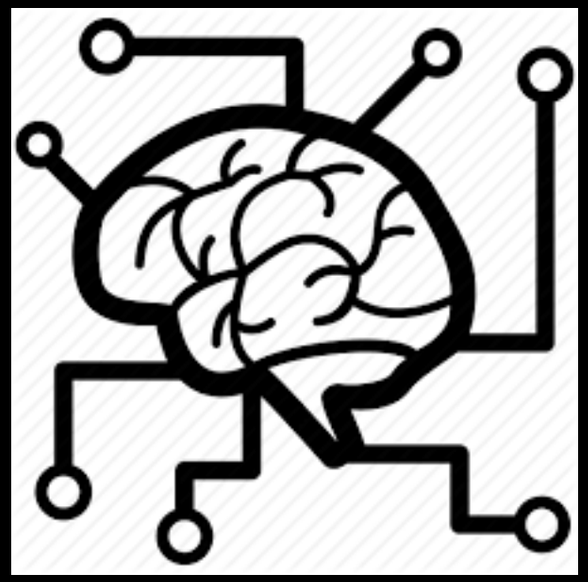
Queue ADT (Circular Array)

```
#ifndef QUEUE_H_
#define QUEUE_H_

template<class ItemType>
class Queue
{

public:
    Queue();
    void enqueue(const ItemType& newEntry); // adds an element to back queue
    void dequeue(); // removes element from front of queue
    ItemType front() const; // returns a copy of element at the front of queue
    int size() const; // returns the number of elements in the queue
    bool isEmpty() const; // returns true if no elements in queue, false otherwise

private:
    static const int DEFAULT_CAPACITY = 100  // Max queue size
    ItemType items_[DEFAULT_CAPACITY];      // the queue
    int front_;              // index of front of queue
    int back_;               // index of back of queue
    int itemCount;          // number of items currently on the stack
};    //end Queue

#include "Queue.cpp"
#endif // QUEUE_H_ `
```

Questions?

Where are we at?

Let’s stay focused on our learning goals!!!



What is CSCI 235?

Programming => Software Analysis and Design

Think like a Computer Scientist:

Design and maintain complex programs

- Software Engineering, Abstraction, OOP

Design and represent data and its management

- Abstract Data Types

Implement data representation and operations

- Data Structures
- Algorithms

Understand Algorithm Complexity

Design and Maintain Complex Programs

OOP: Work with multiple classes

(some you write, others you read/understand/use)

- accessor / mutator / helper methods (public vs private)
- constructors / copy constructors / destructors / operator overloading

Write templated classes

Inheritance/polymorphism => understand the difference between the two and when appropriate to use them

Memory management (and how it relates to destructors / copy constructors / overloaded= operator)

Exception Handling

Design and Represent Data and its Management

Abstract Data Types

- Bag
- Set
- List
- Stack
- Queue
- Trees

Represent and Implement Data

Data Structures

- Array / Circular Array
- Vector
- Linked Chain / Circular Chain
- Doubly-Linked Chain

Represent and Implement Operations

Much more in CSCI 335

Algorithms

- **Recursive Algorithms:**
  reverse, factorial, backtracking (n-queens problem / mazeSolver), recursive decision tree (permutations, combinations)
- **Searching algorithms:**
  Linear Search, Binary Search
- **Sorting Algorithms**
  Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, Quick Sort
- **Stack Algorithms**
  Balancing, Arithmetic Expression calculation and conversion, Reversing, Backtracking, Depth-First Traversal
- **Queue Algorithms**
  Preserving Order (job queue), Breadth-First Traversal
- **Tree Traversals**

Understand Algorithm Complexity

Much more in CSCI 335

What is Algorithm Complexity

Analysis of algorithms (how to)

Big-O: worst-case analysis

T(n) is O(f(n))

Omega: best-case analysis

Mentioned

T(n) is Ω(f(n))

Theta: average-case analysis

Mentioned

T(n) is Θ(f(n))