Queue Implementations

Tiziana Ligorio

Hunter College of The City University of New York

Today's Plan

Announcements
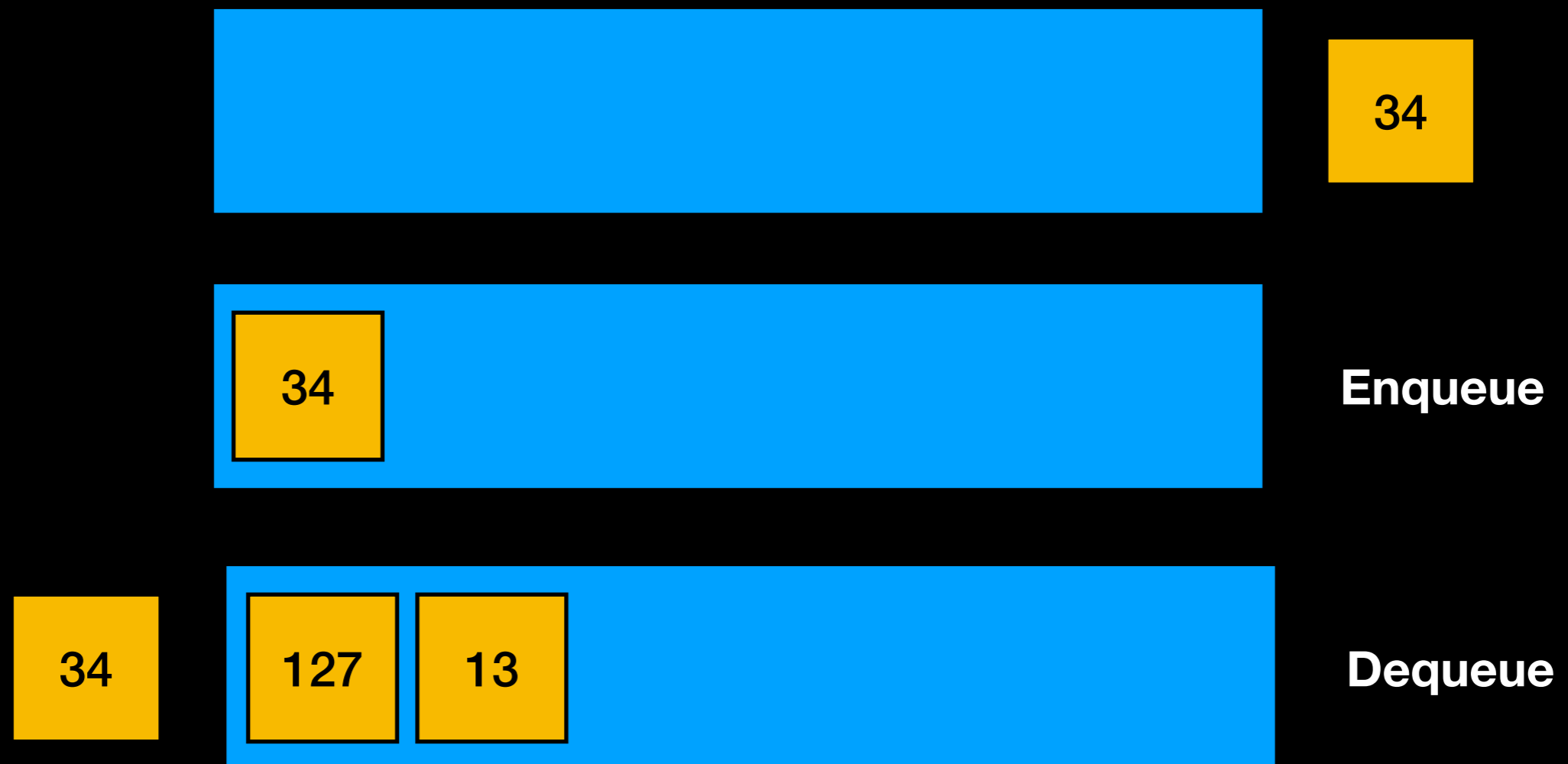
Recap

Queue Implementations

Announcements

Read Interlude 8 - STL

Recap

FIFO structure: First In First Out

	34

34		**Enqueue**

34	127	13		**Dequeue**

Queue ADT

```cpp
#ifndef QUEUE_H_
#define QUEUE_H_

template<class T>
class Queue
{

public:
    Queue();
    void enqueue(const T& new_entry); // adds an element to back queue
    void dequeue(); // removes element from front of queue
    T 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

	Big-O	Size unbounded
Array	O(1)	❌
Vector	O(1)+	✔
Linked Chain	O(1)	✔

Amortized Analysis

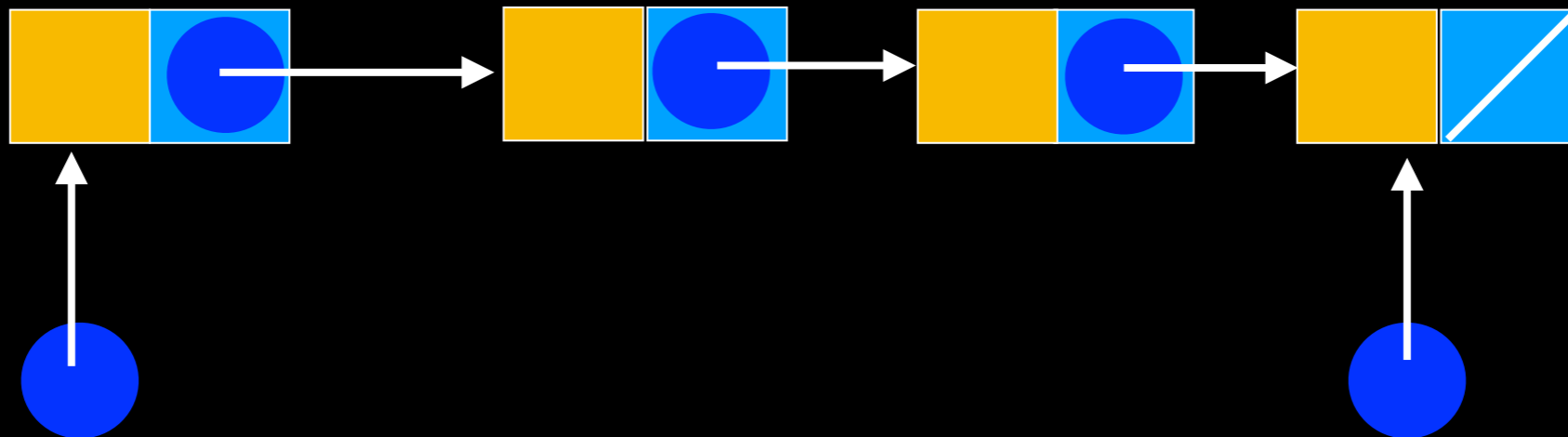# What is the main difference btw stack and queue?
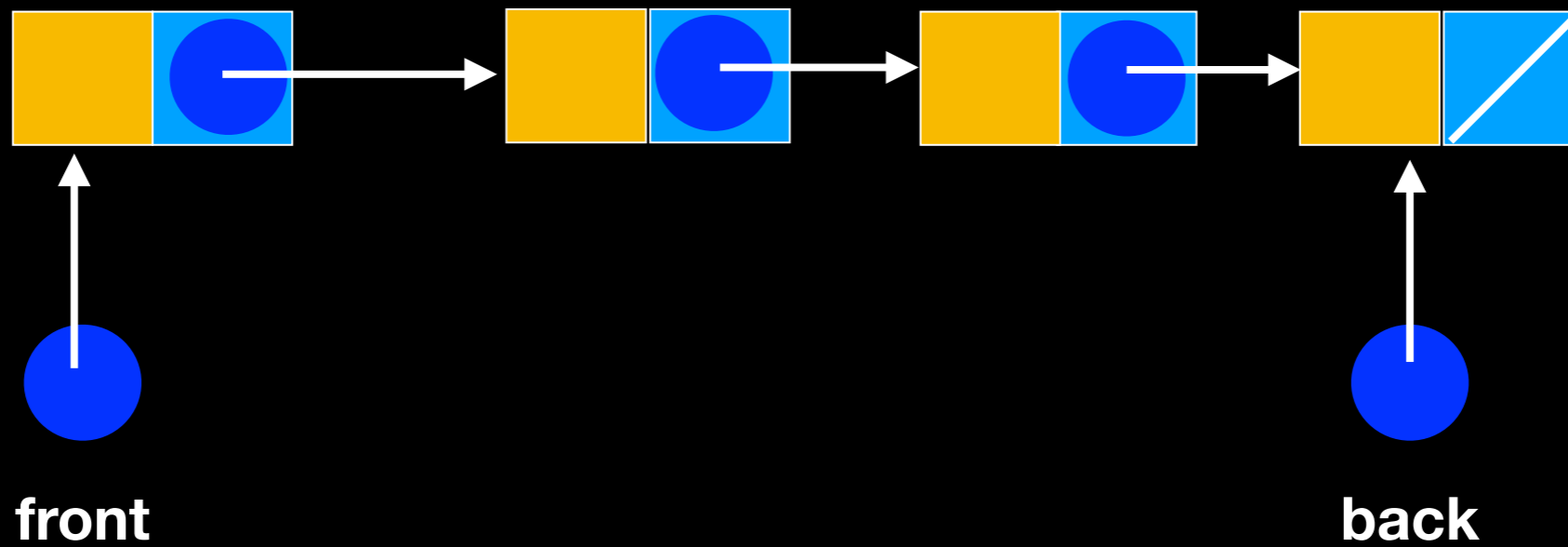
Singly Linked Chain

Where is front?
Where is back?
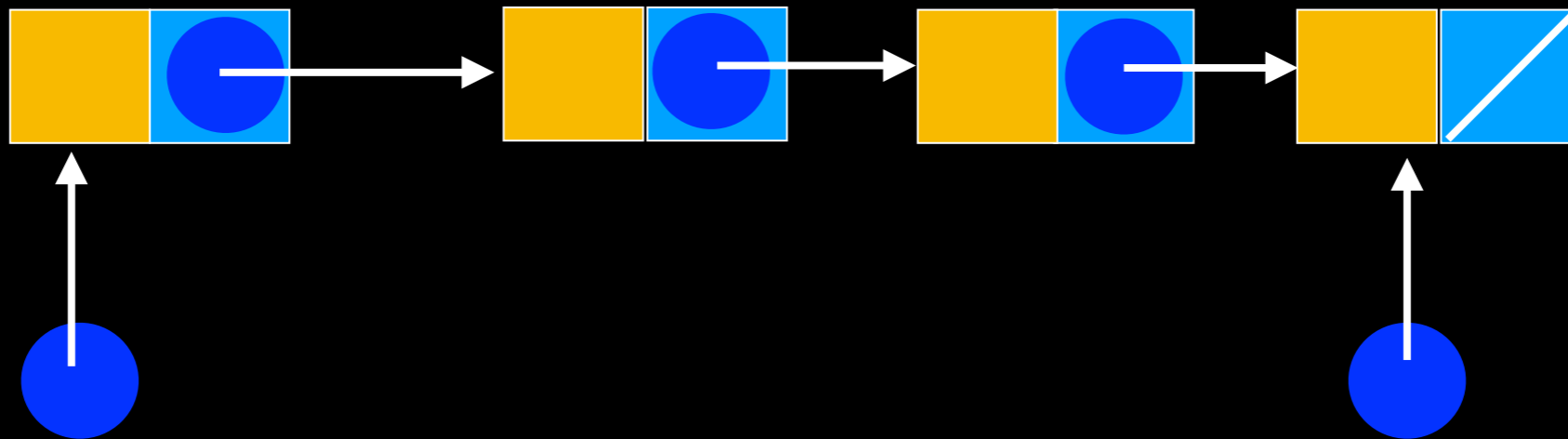
Singly Linked Chain

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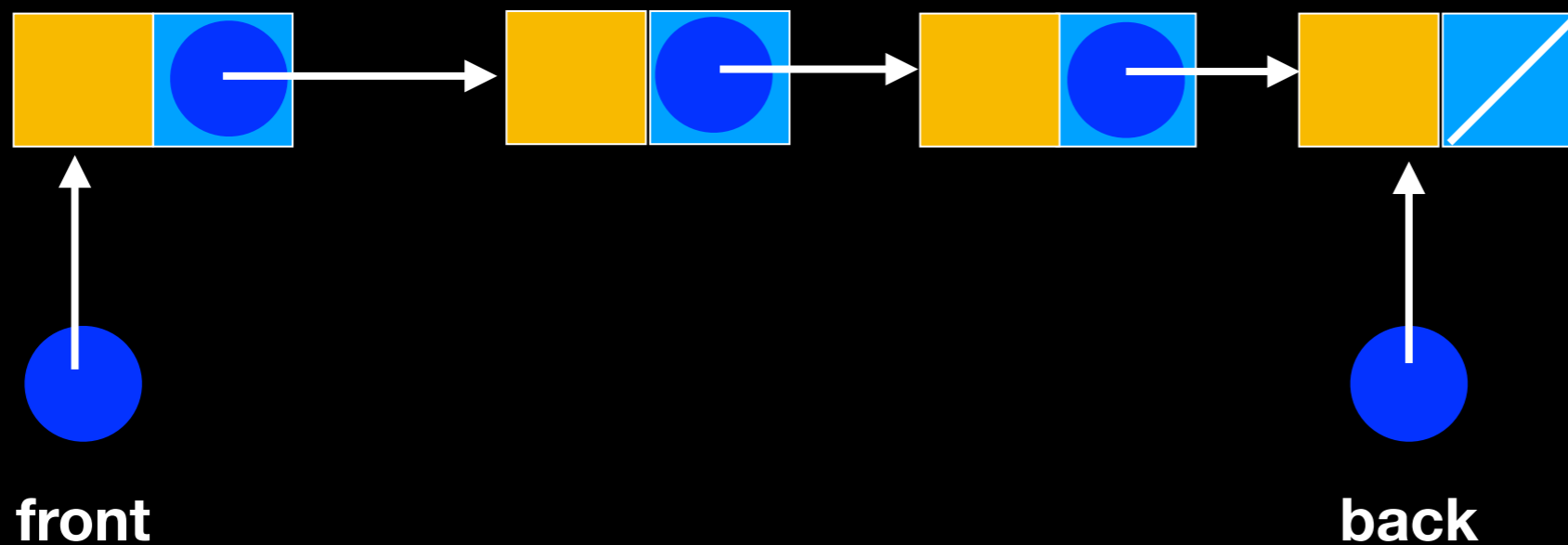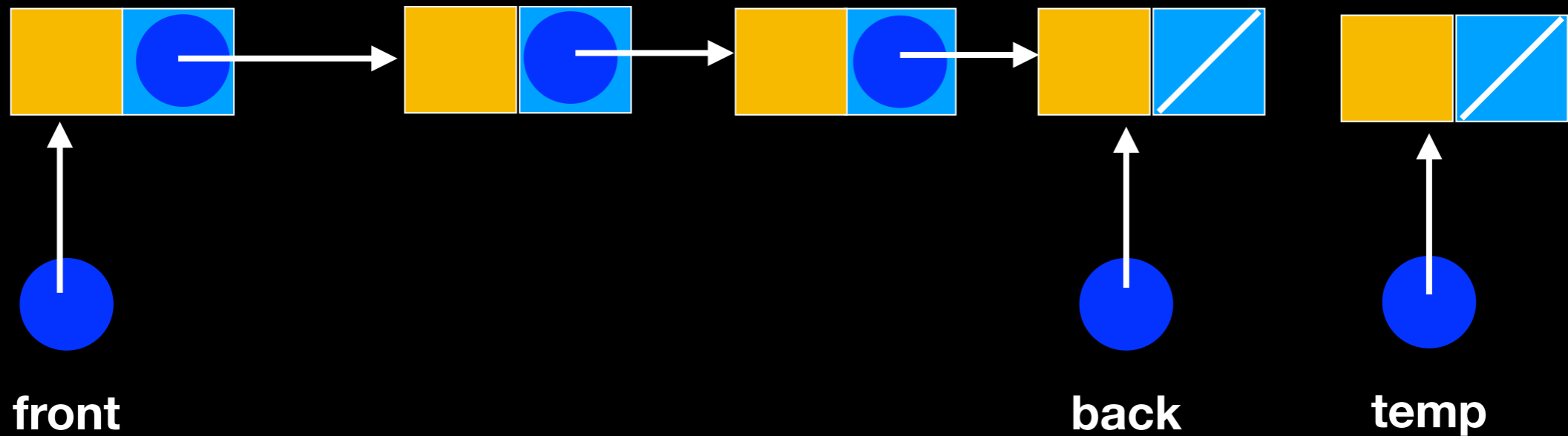Because we don't have
pointer to previous node

Singly Linked Chain

front   **back**
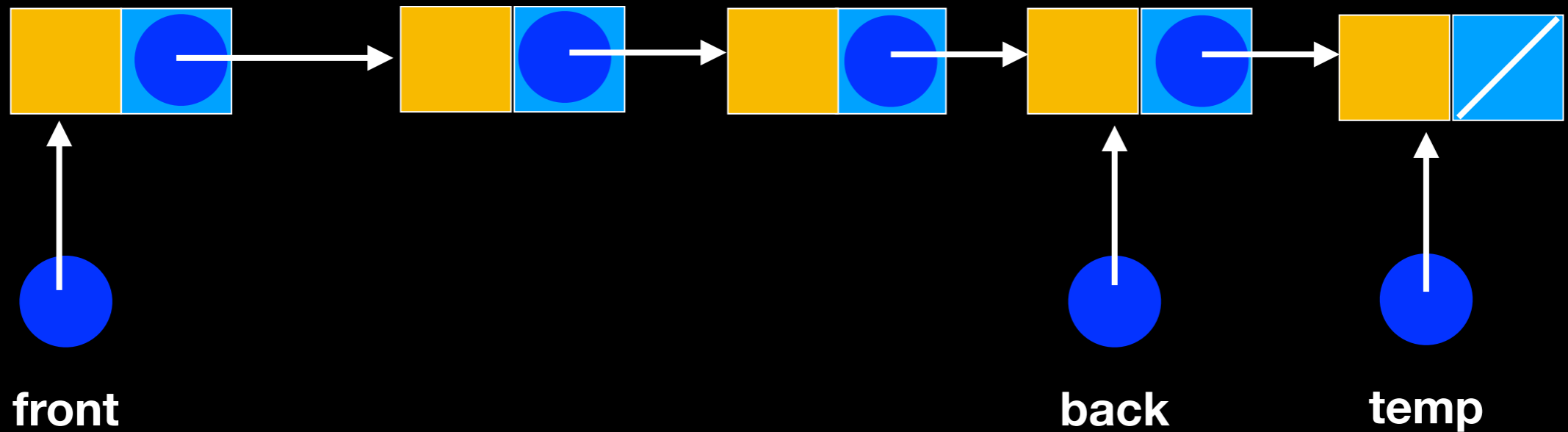
Singly Linked Chain

enqueue

front   back

Singly Linked Chain

enqueue

front   back   temp

Singly Linked Chain

enqueue

Singly Linked Chain

enqueue

front

back temp

Singly Linked Chain

enqueue

Singly Linked Chain

dequeue

front   back
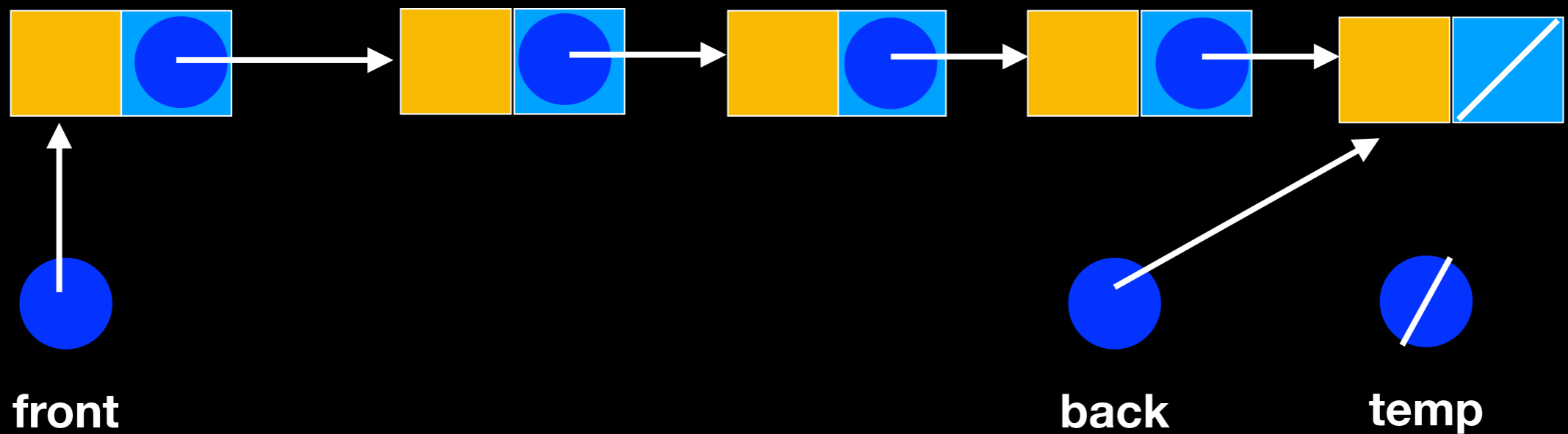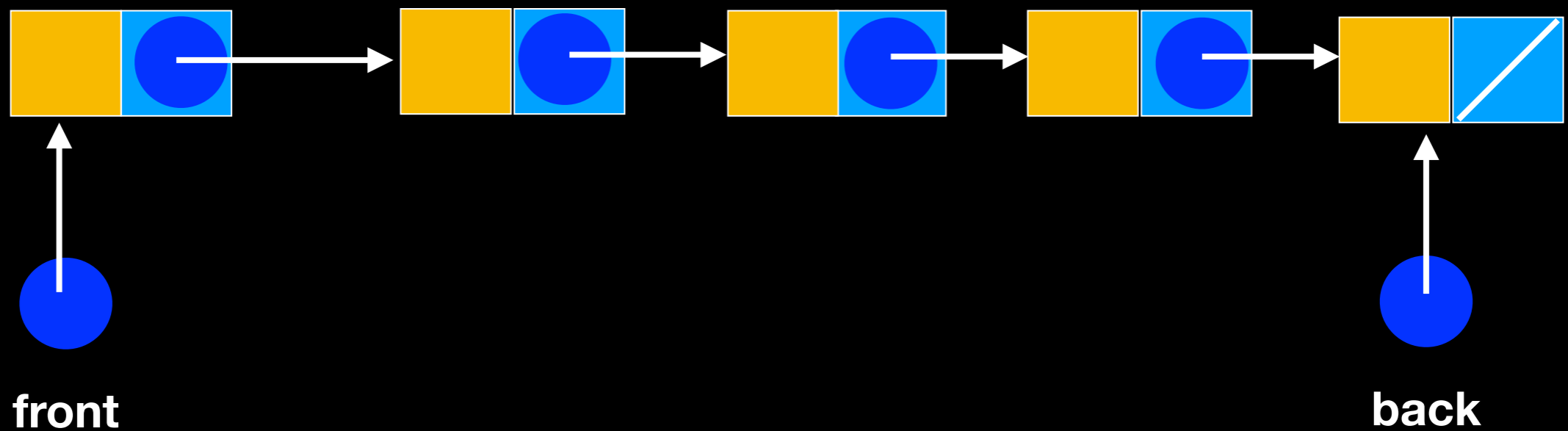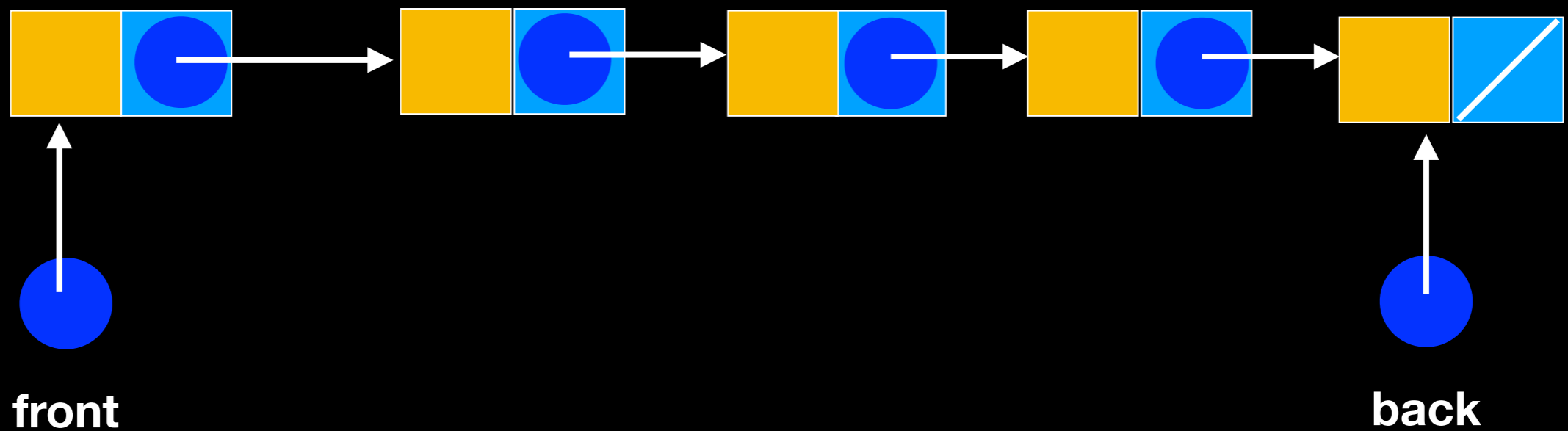
Singly Linked Chain

dequeue

Singly Linked Chain

dequeue

temp

front

back

Singly Linked Chain

dequeue

temp

front

back

Singly Linked Chain

dequeue

temp

front

back

Singly Linked Chain
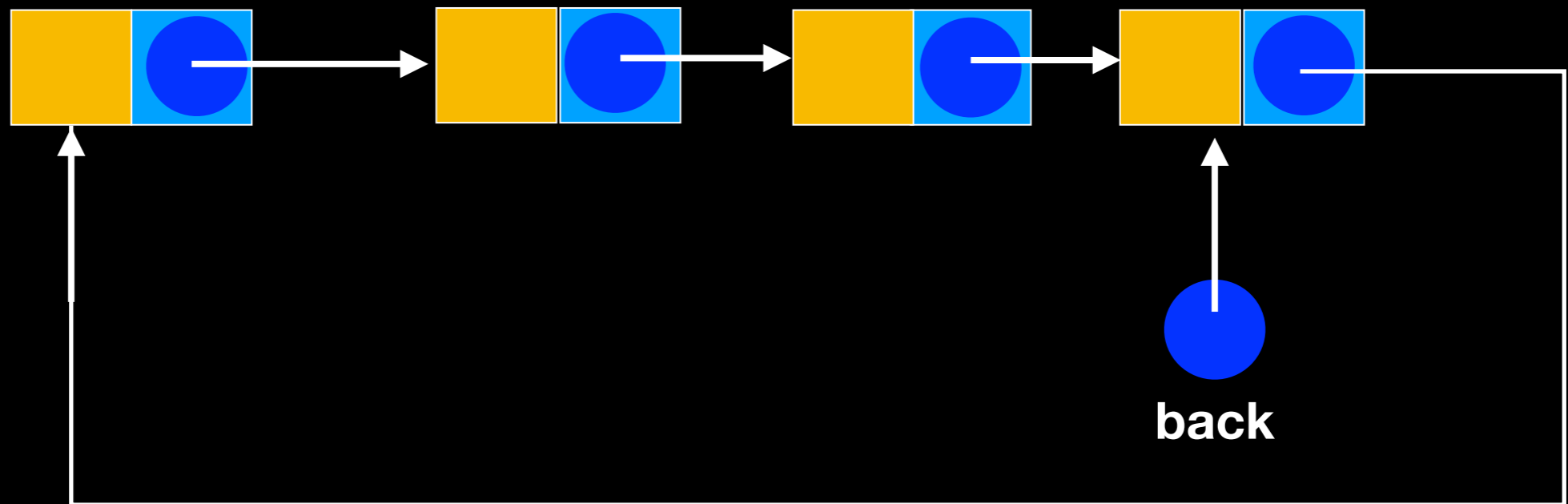

front **back**
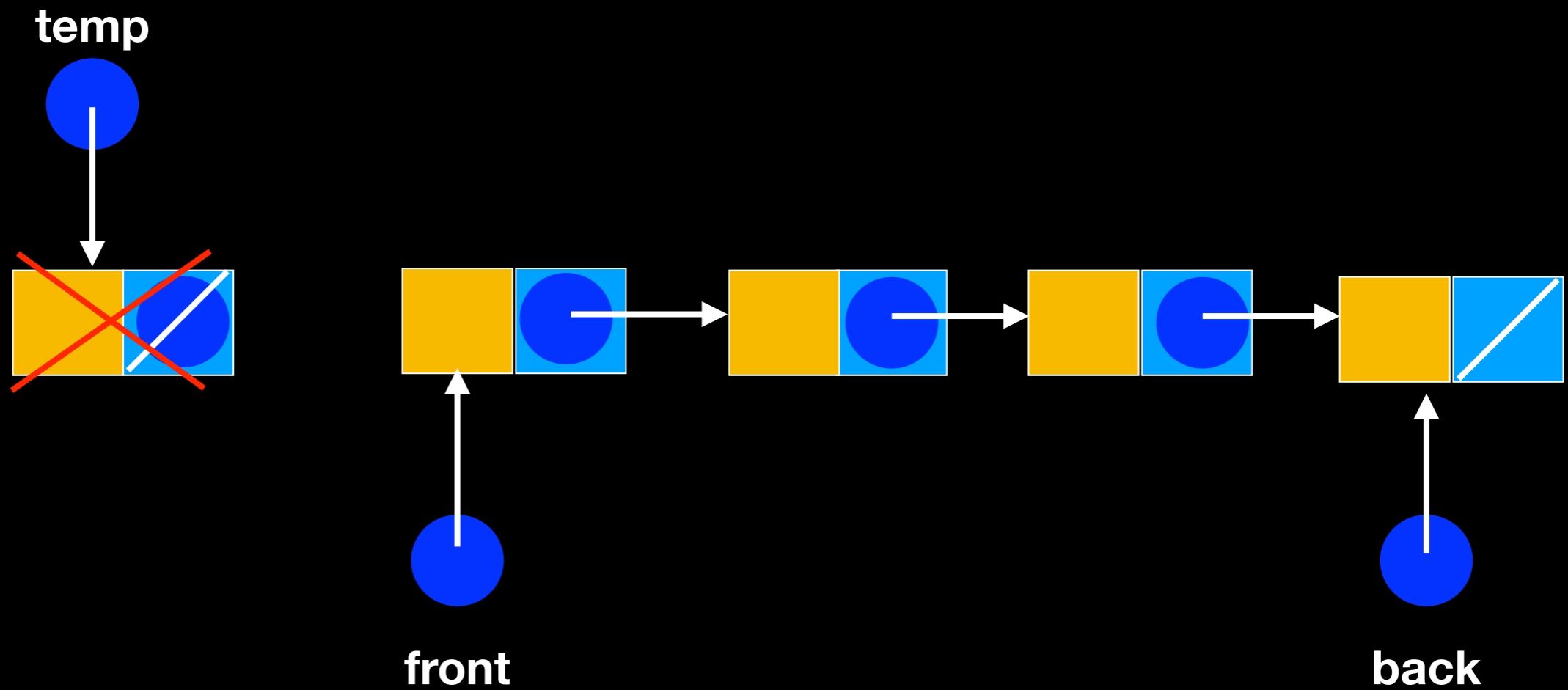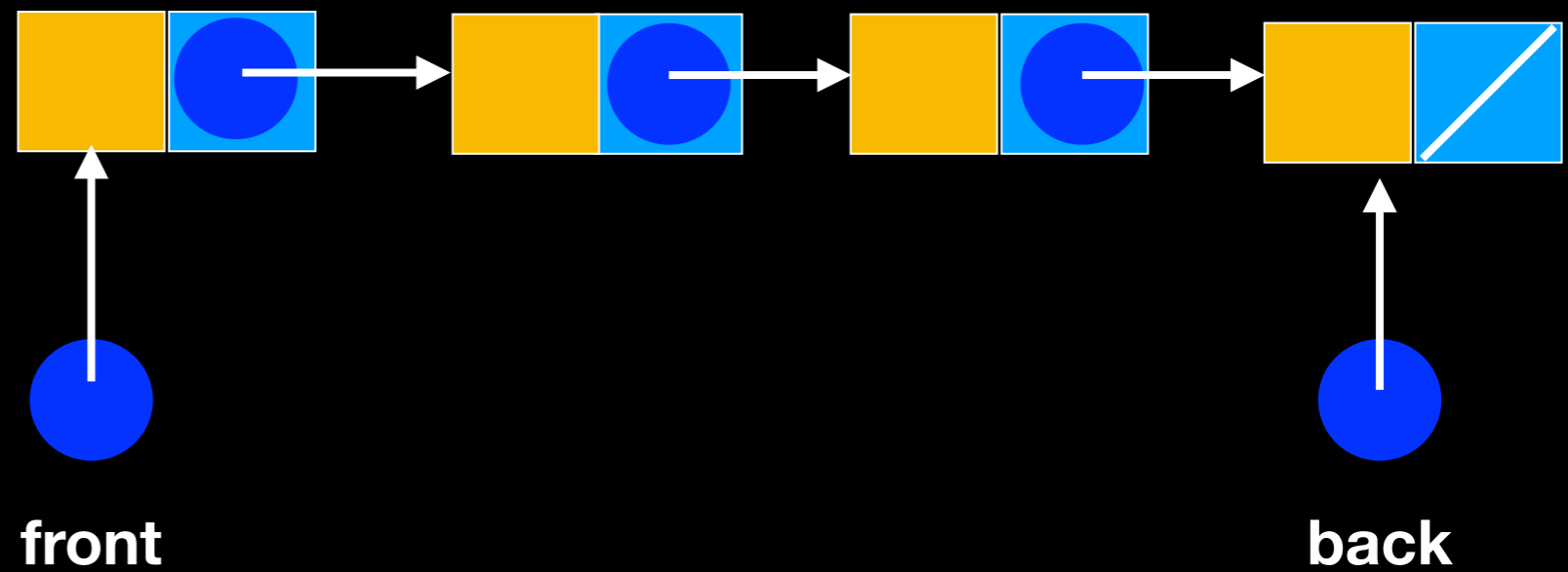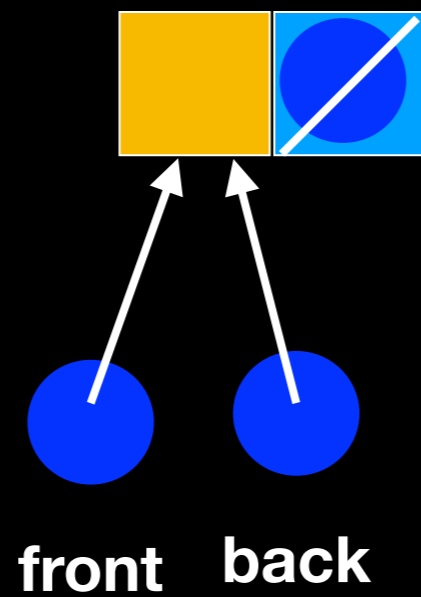
That's it!

Singly Linked Chain

An Alternative:
A Circular Linked Chain

back

Singly Linked Chain

enqueue

**An Alternative:
A Circular Linked Chain**

Instantiate new node

back

temp

Singly Linked Chain

enqueue

**An Alternative:
A Circular Linked Chain**

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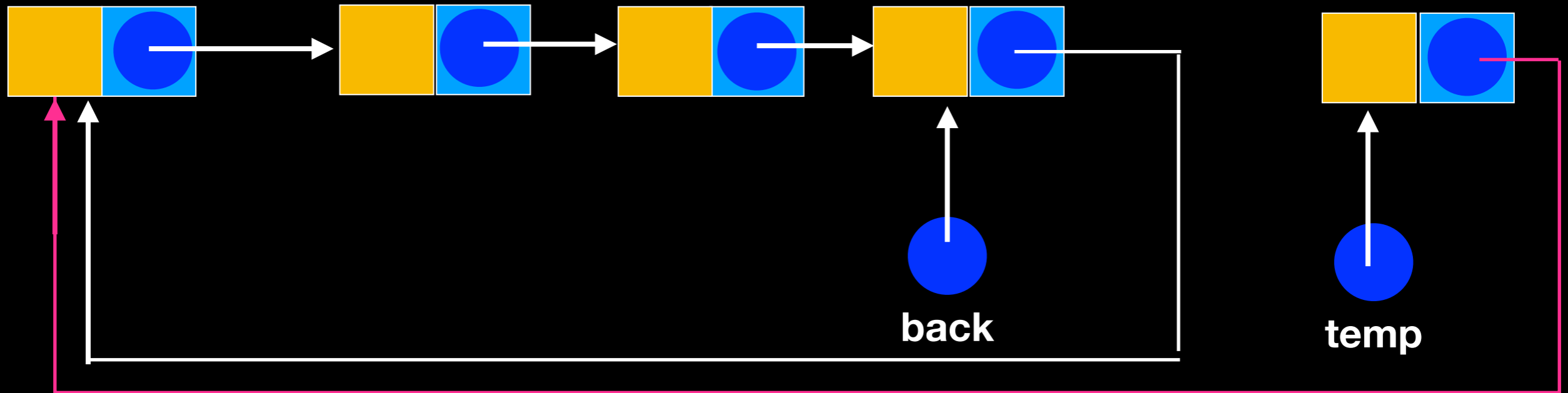

back

temp

Singly Linked Chain

enqueue

**An Alternative:
A Circular Linked Chain**

```
back->setNext(temp);
```

back

temp

Singly Linked Chain

enqueue

**An Alternative:
A Circular Linked Chain**

```
back = temp;
```

back

temp

Singly Linked Chain

enqueue

**An Alternative:
A Circular Linked Chain**
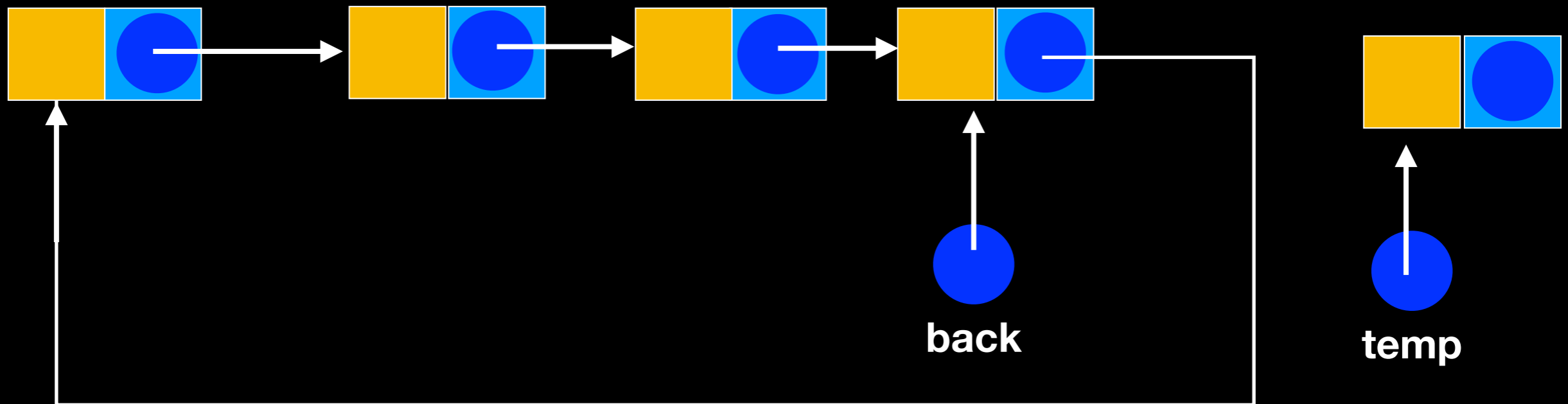
```
temp = nullptr;
```
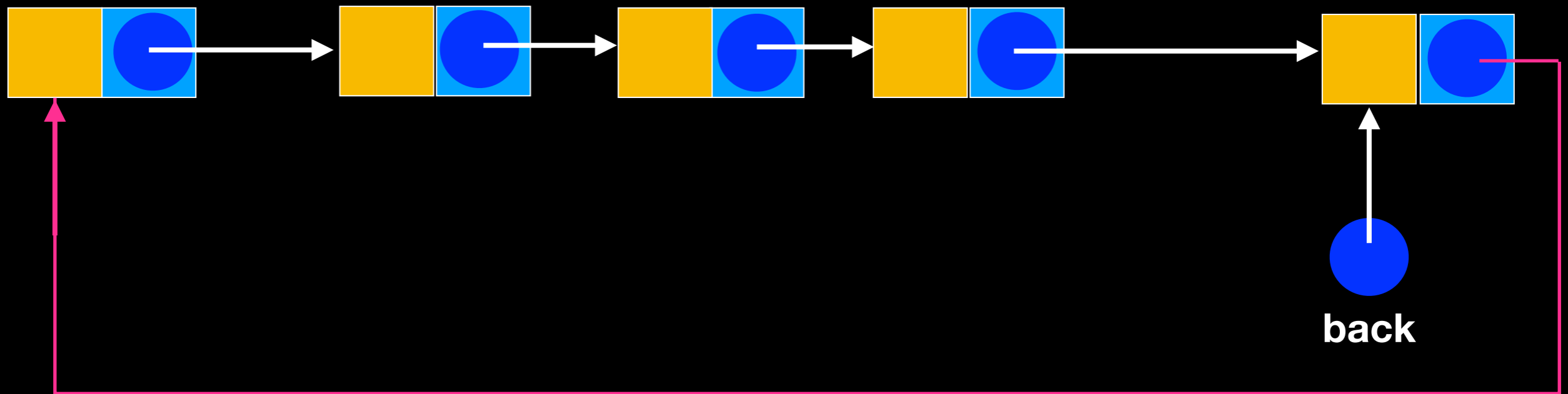

back

temp

Singly Linked Chain

enqueue
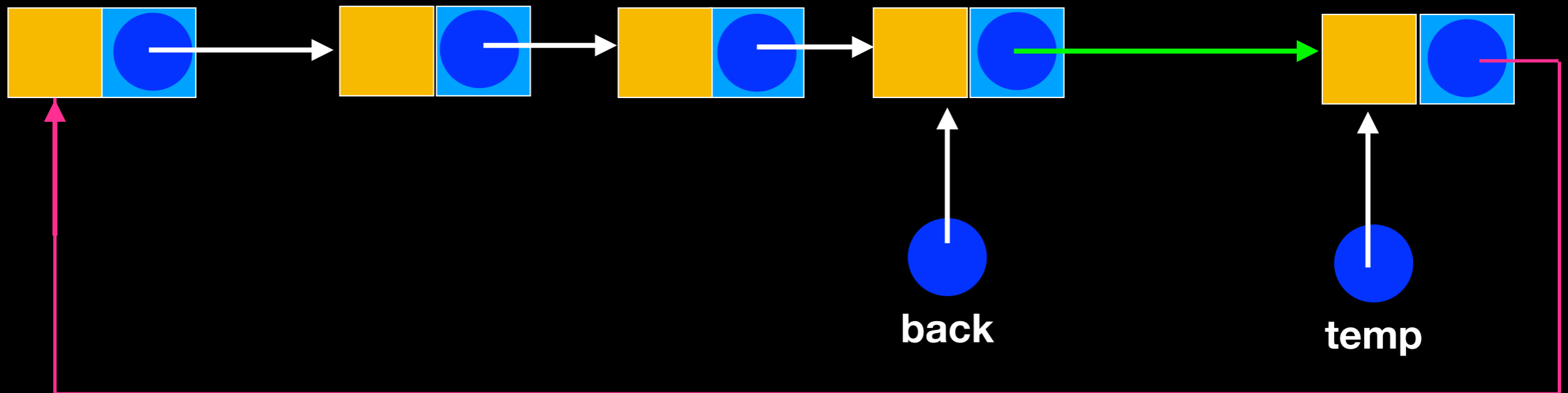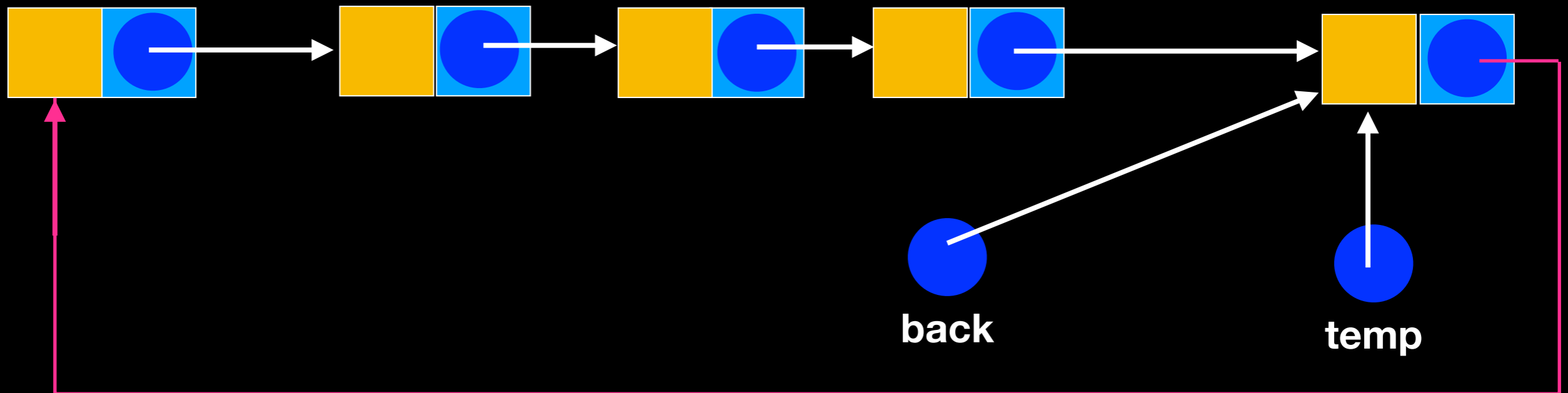
An Alternative:
A Circular Linked Chain

back

30

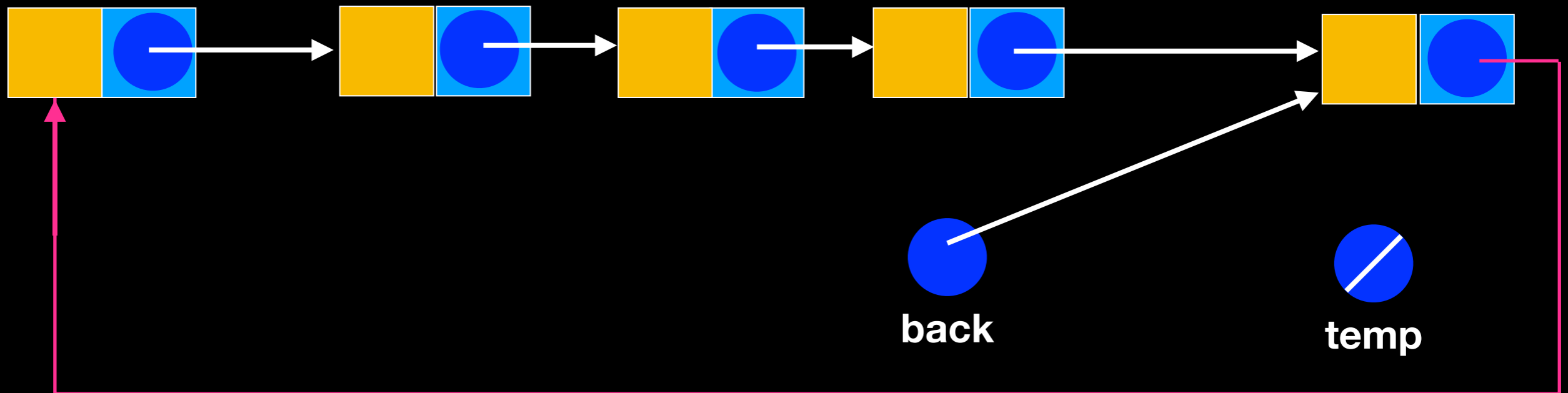# Singly Linked Chain

dequeue

**An Alternative:
A Circular Linked Chain**
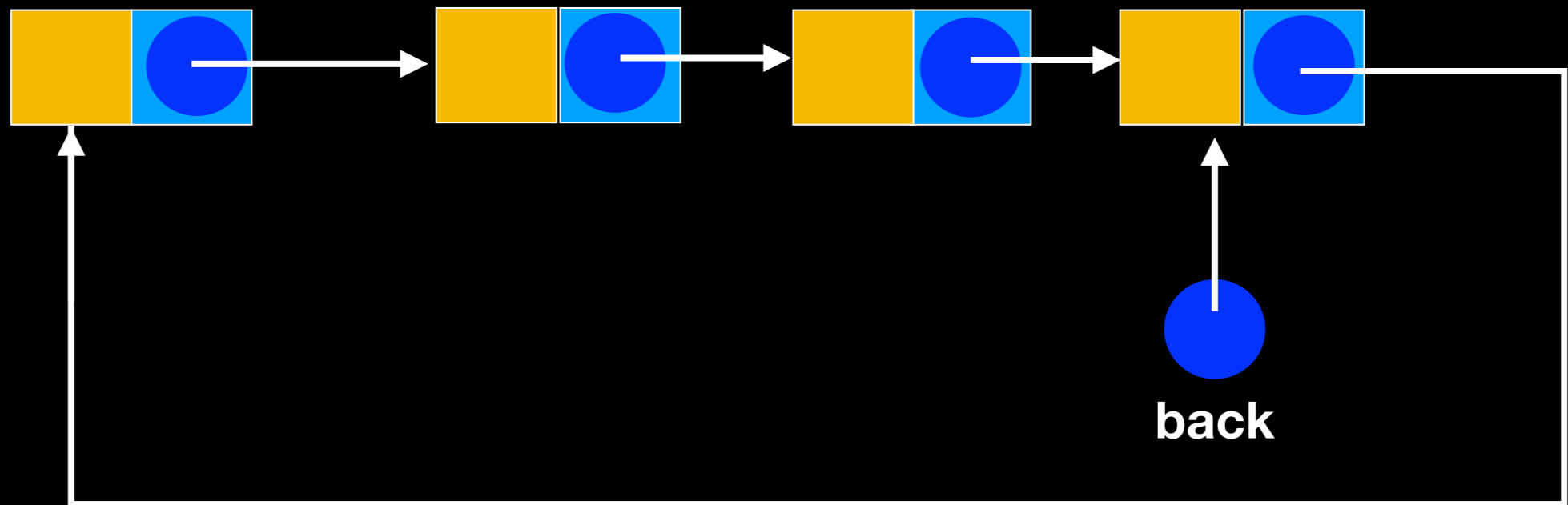
back

Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

```
temp = back->getNext()
```
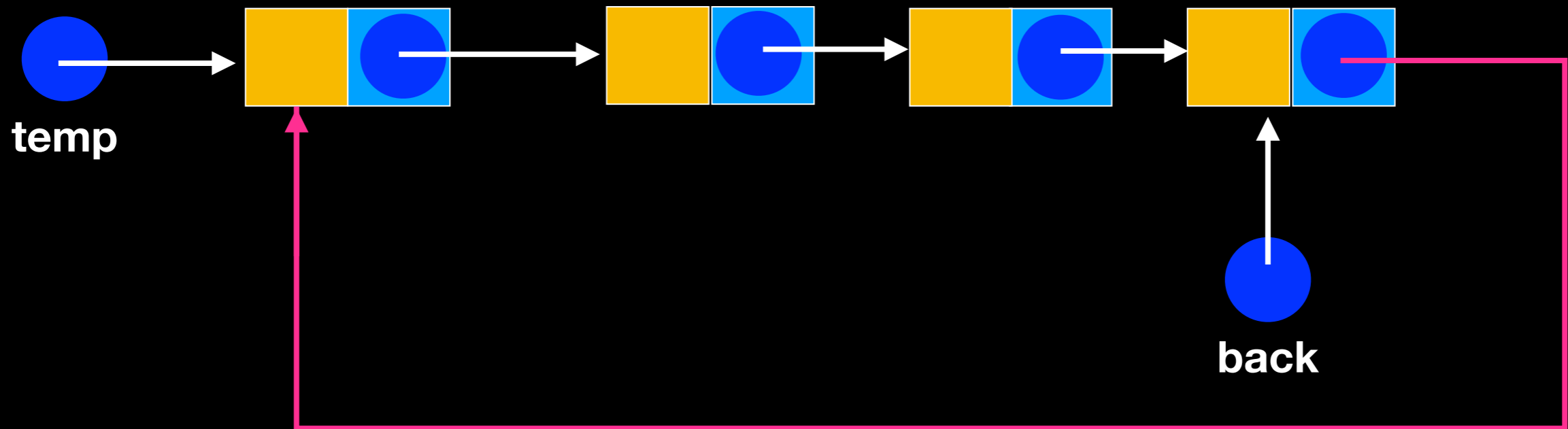
temp

back

Singly Linked Chain

dequeue

**An Alternative:
A Circular Linked Chain**

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

temp

back

Singly Linked Chain

dequeue

**An Alternative:
A Circular Linked Chain**

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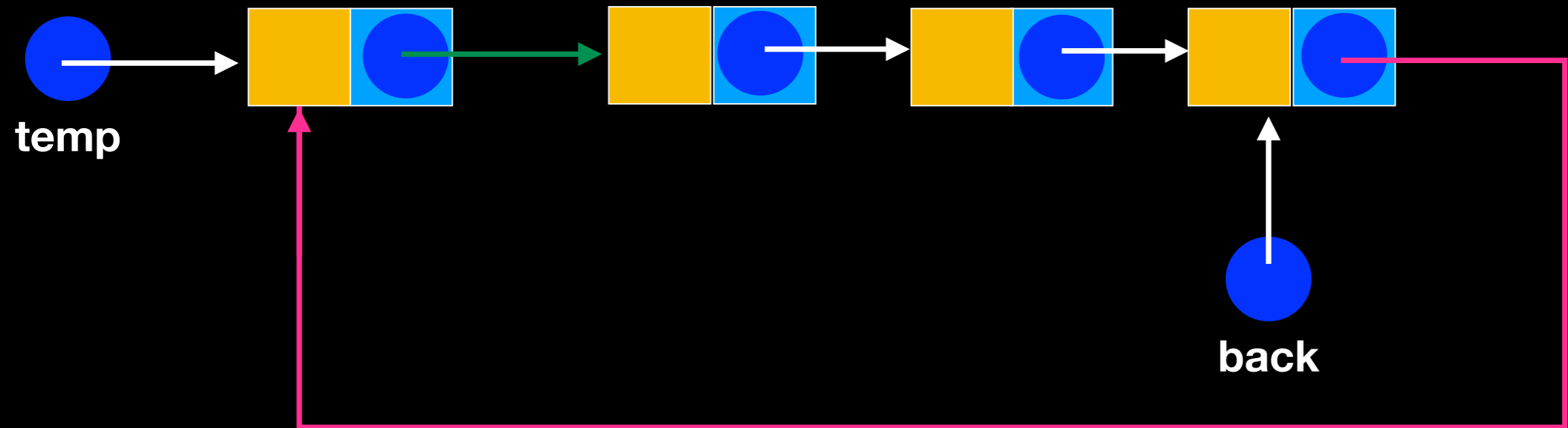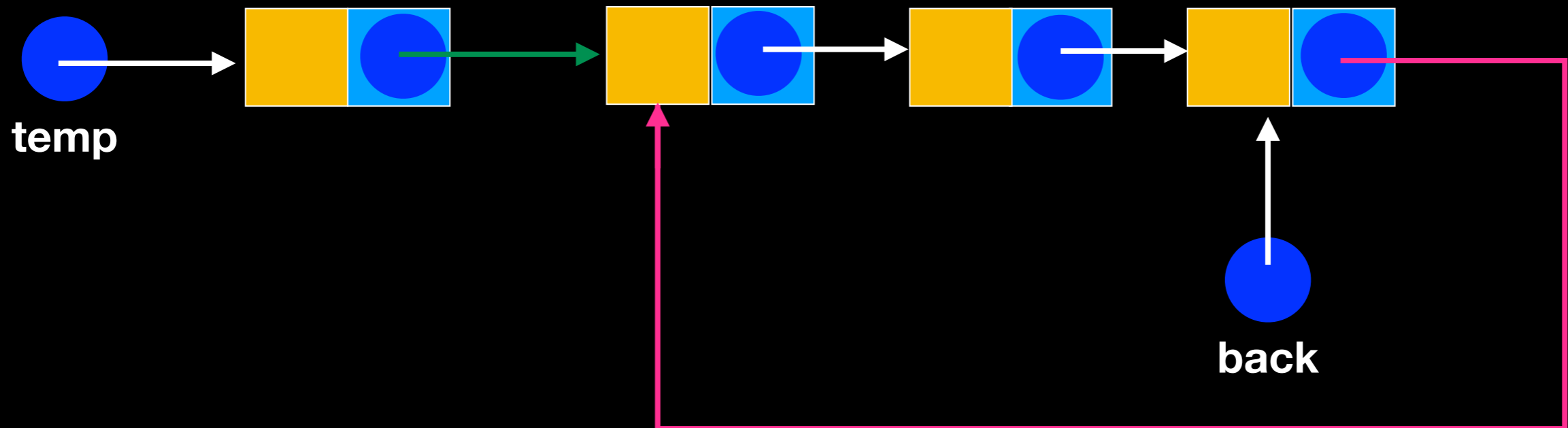
temp

back

Singly Linked Chain

dequeue

**An Alternative:
A Circular Linked Chain**

```
temp->setNext(nullptr);
        delete temp;
```
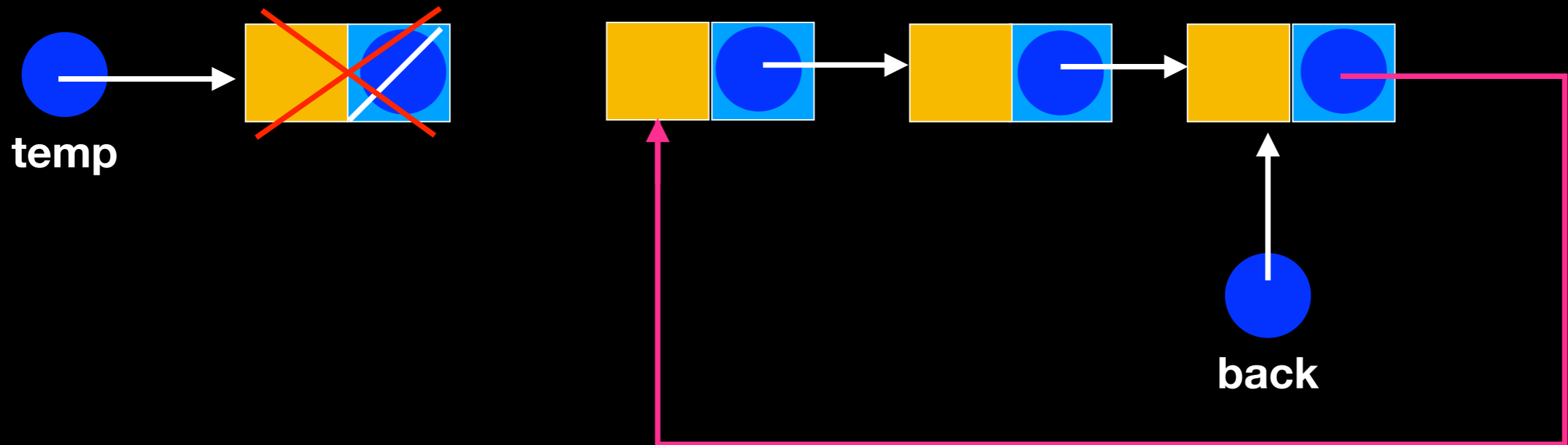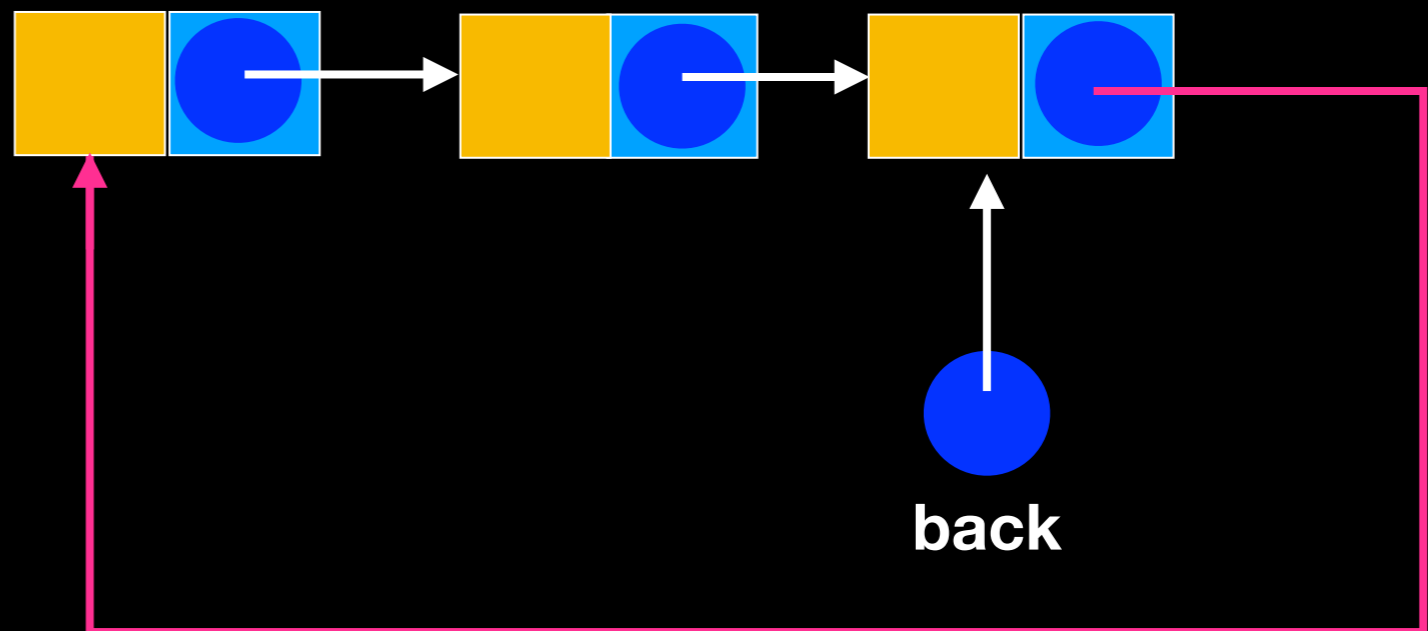
temp

back

Singly Linked Chain

dequeue

**An Alternative:
A Circular Linked Chain**

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

temp

back

Queue ADT
(Circular Linked Chain)

```cpp
#ifndef QUEUE_H_
#define QUEUE_H_

template<class T>
class Queue
{

public:
    Queue();
    Queue(const Queue<T>& a_queue); // Copy constructor
    ~Queue();
    void enqueue(const T& new_entry); // adds an element to back queue
    void dequeue(); // removes element from front of queue
    T 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<T>* back_; // Pointer to back of queue
    int item_count;          // number of items currently on the stack


};    //end Queue

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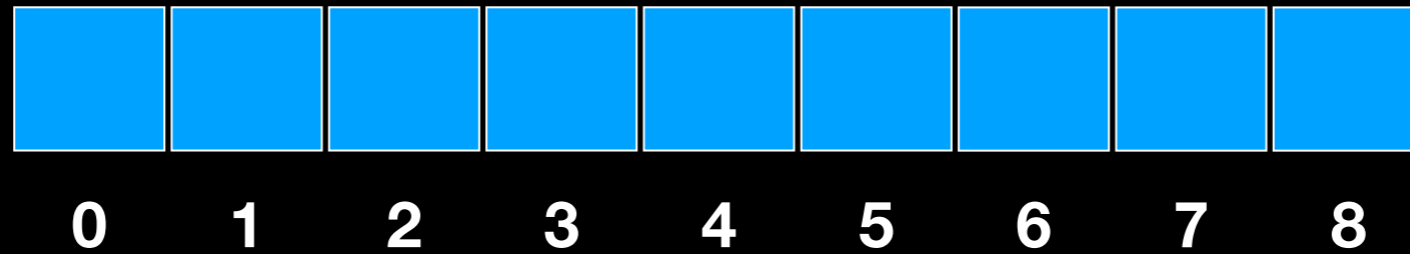
Lecture Activity

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

Array Considerations

front = ?
back = ?

0   1   2   3   4   5   6   7   8

Array Considerations

front = 0

back = -1
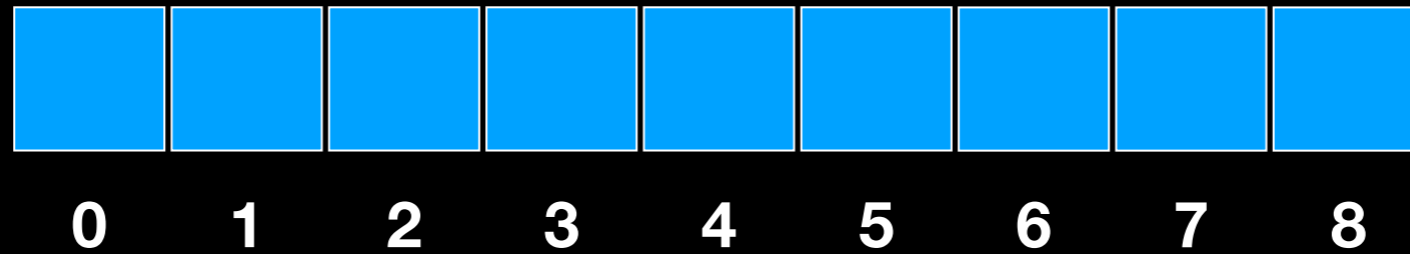
0   1   2   3   4   5   6   7   8

Array Considerations

enqueue

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

front = 0
back = -1

0	1	2	3	4	5	6	7	8

Array Considerations

enqueue

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

front = 0

back = 0
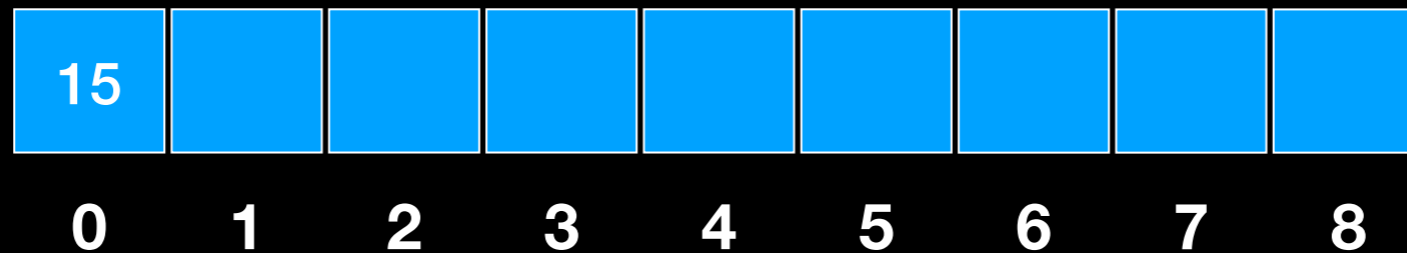
15								
0	1	2	3	4	5	6	7	8

Array Considerations

enqueue

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

front = 0
back = 1

15	3							
0	1	2	3	4	5	6	7	8

Array Considerations

enqueue

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

front = 0
back = 2

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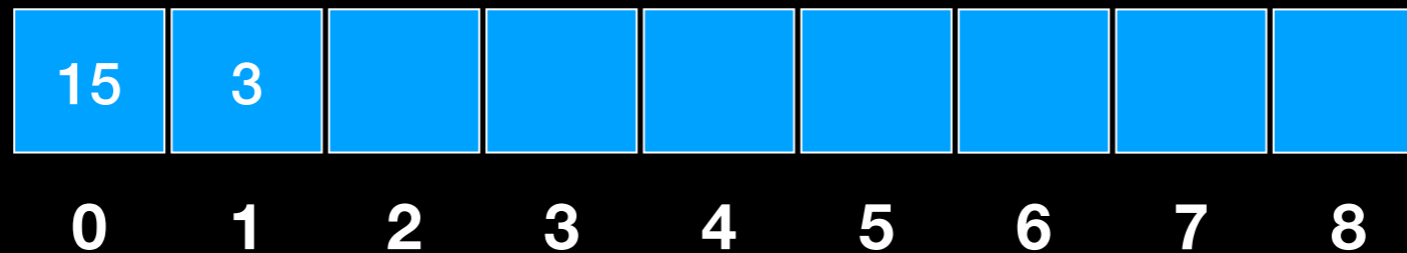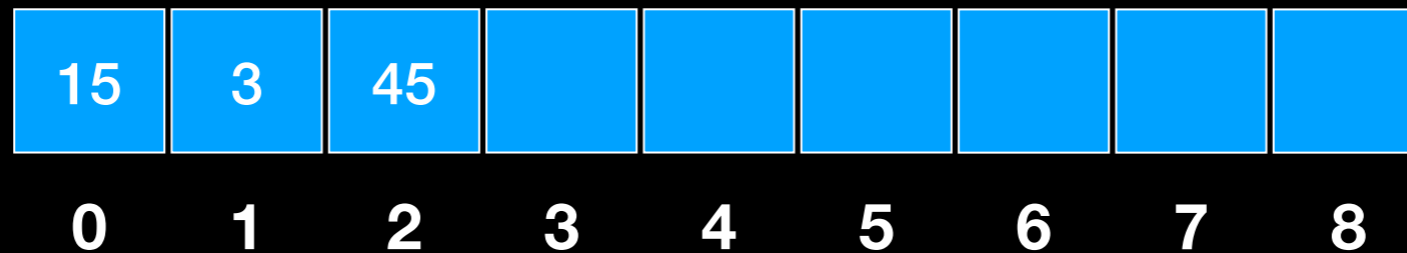
Array Considerations

enqueue

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

front = 0

back = 5

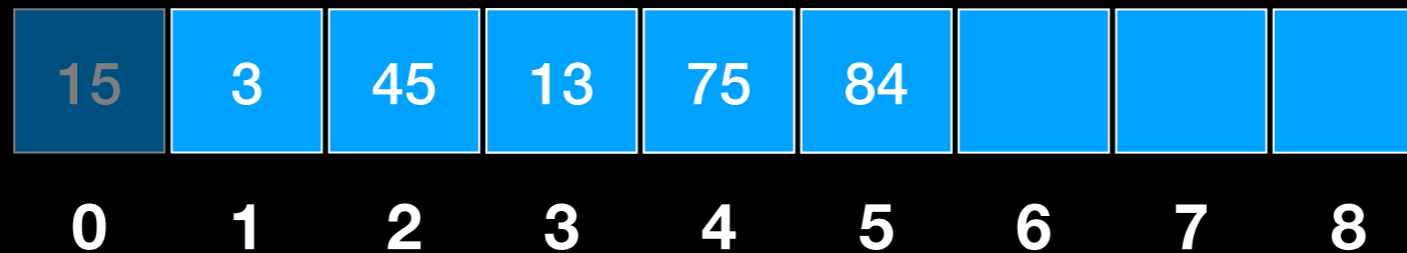| 15	3	45	13	75	84			
0	1	2	3	4	5	6	7	8
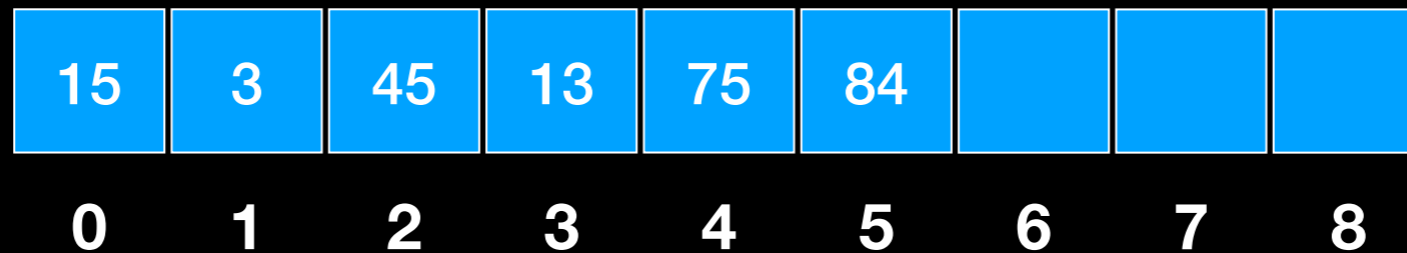
This seems to work, but what happens when we start dequeuing?

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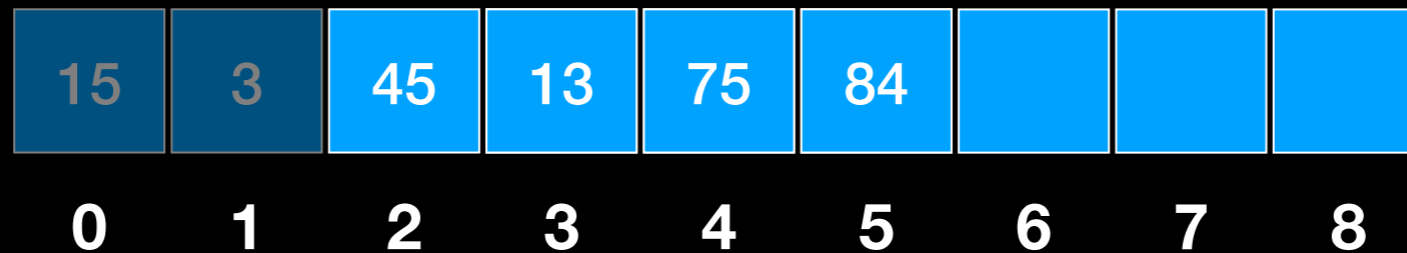
15	3	45	13	75	84			
0	1	2	3	4	5	6	7	8

Array Considerations

front = 6

back = 8

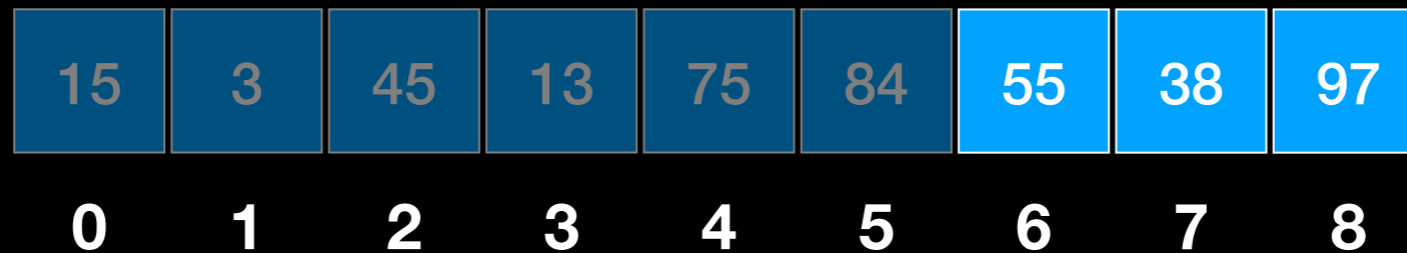| 15	3	45	13	75	84	55	38	97
0	1	2	3	4	5	6	7	8
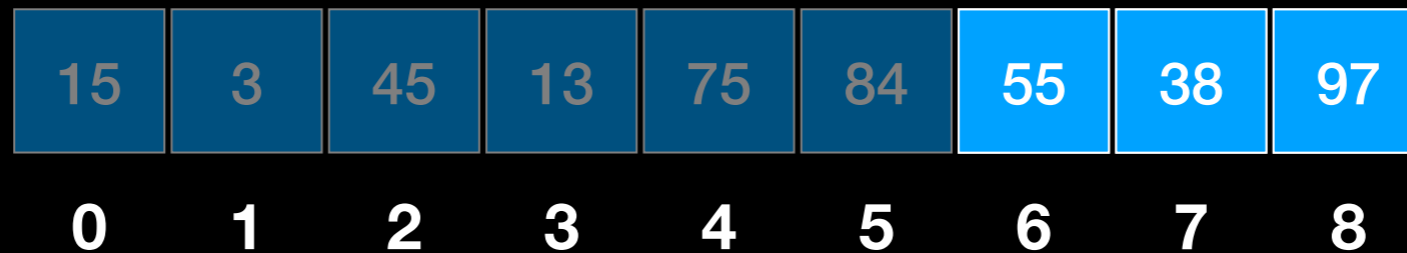
RIGHTWARD DRIFT!!!
At some point queue will be full even if it contains only a few elements

Array Considerations

front = 6

back = 8

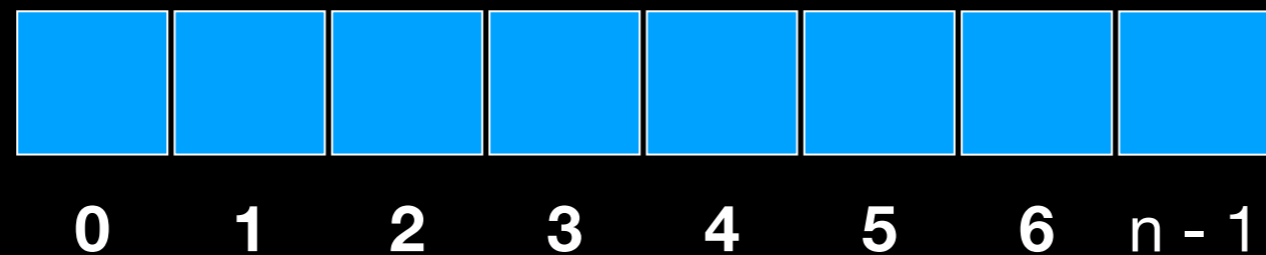| 15	3	45	13	75	84	55	38	97
0	1	2	3	4	5	6	7	8

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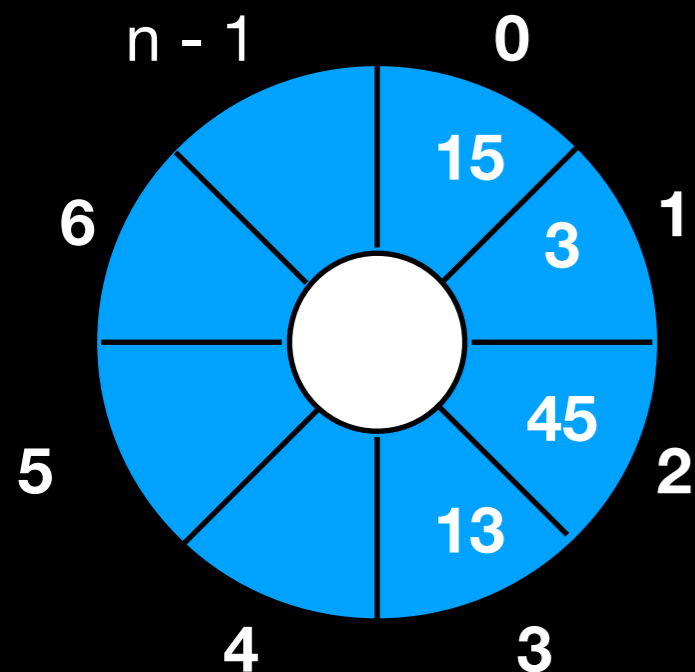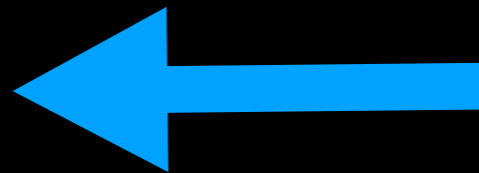At some point queue will be full even if it contains only a few elements

No Good

49

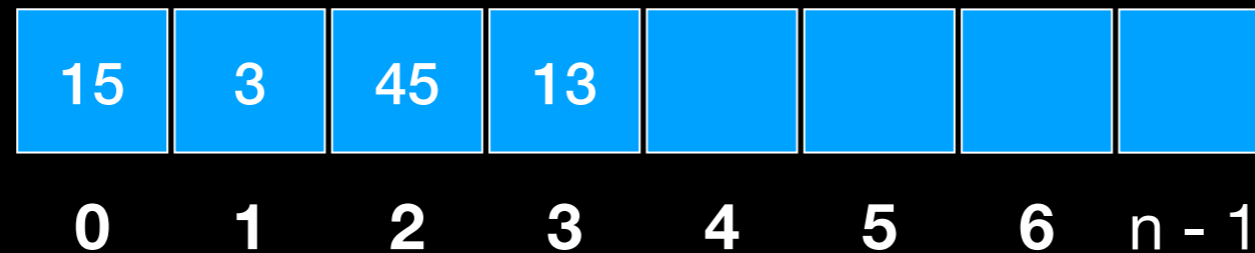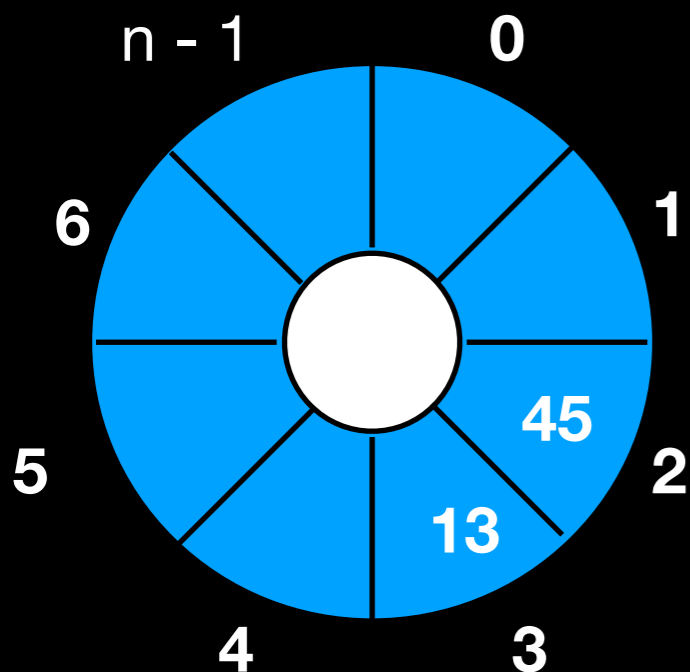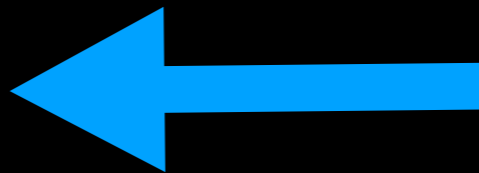# Circular Array Implementation

`front = 0`

`back = -1`

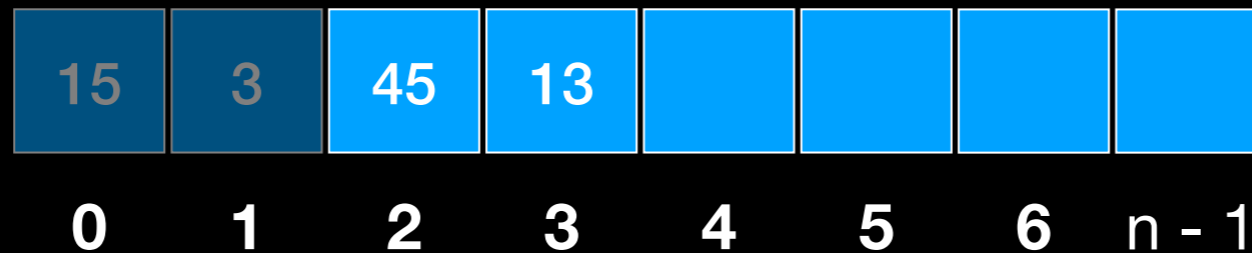0　1　2　3　4　5　6　n - 1

Circular Array Implementation

front = 0

back = 3

15	3	45	13				
0	1	2	3	4	5	6	n - 1

Circular Array Implementation

front = 2

back = 3

15	3	45	13				
0	1	2	3	4	5	6	n - 1

Circular Array Implementation

front = 6

back = n - 1

15	3	45	13	75	59	88	21
0	1	2	3	4	5	6	n - 1

Circular Array Implementation

front = 6

back = 0

5	3	45	13	75	59	88	21
0	**1**	**2**	**3**	**4**	**5**	**6**	n - 1
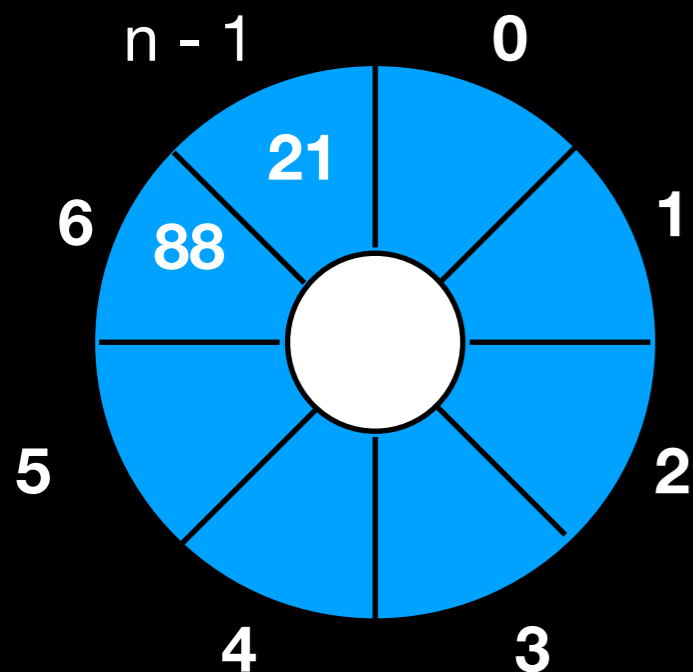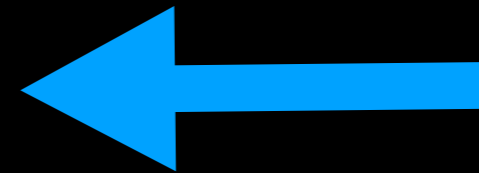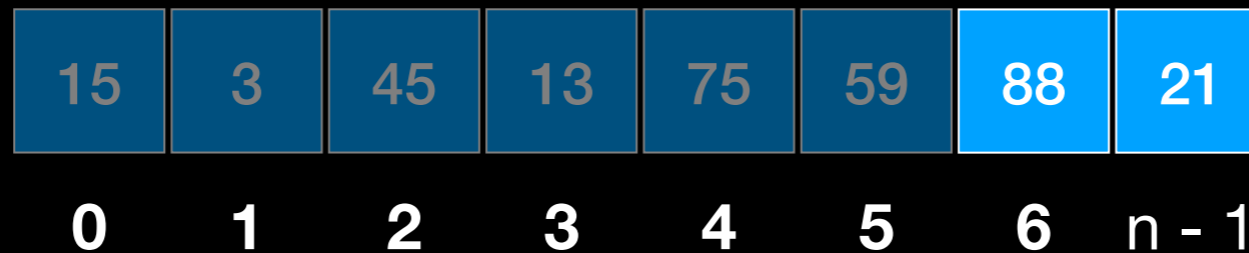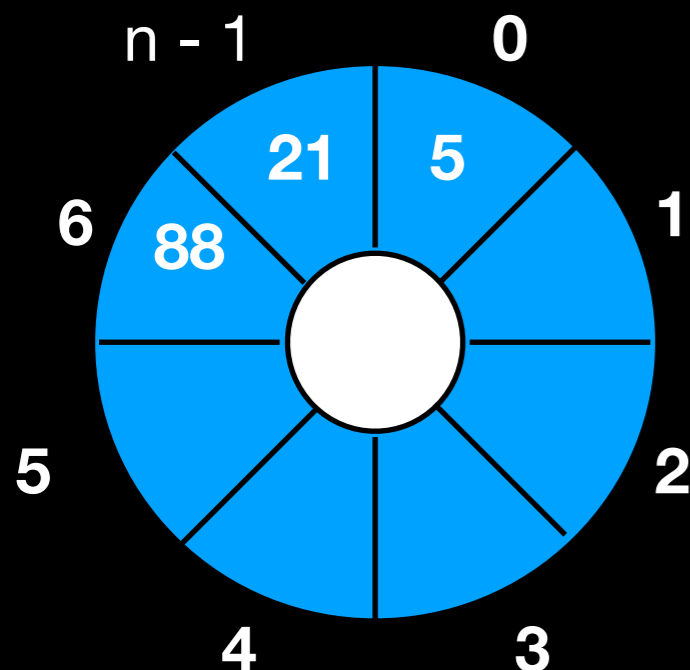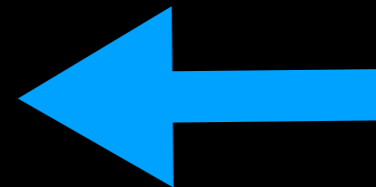
WRAP AROUND USING MODULO ARITHMETIC

Circular Array Implementation

enqueue

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

front = 6

back = 0

5	3	45	13	75	59	88	21
0	1	2	3	4	5	6	n - 1

Circular Array Implementation

front = n-1

back = 2
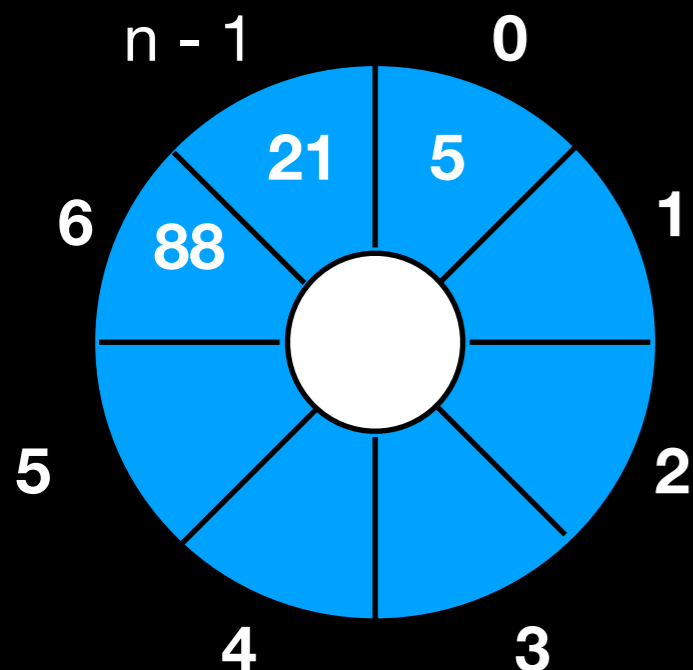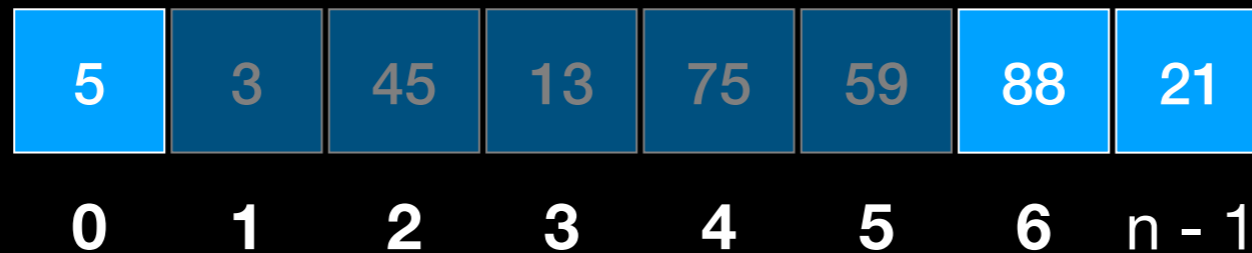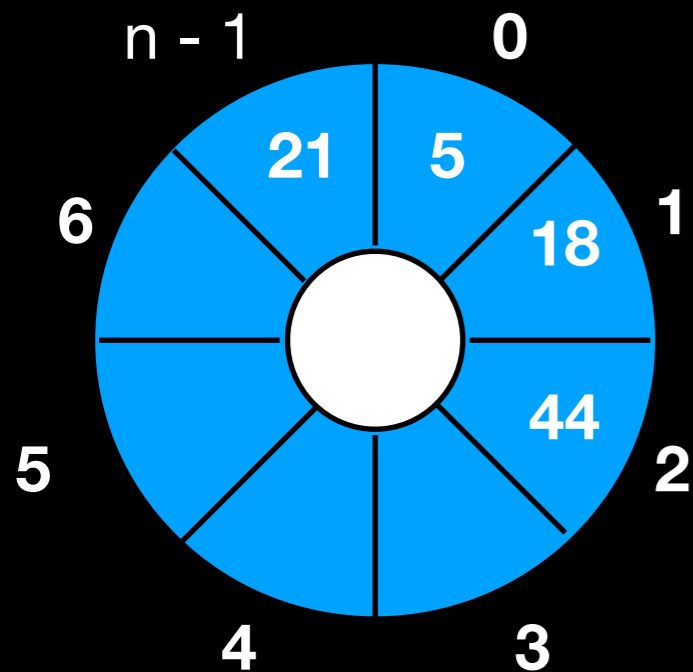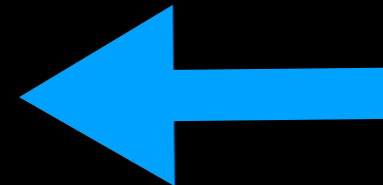
5	18	44	13	75	59	88	21
0	1	2	3	4	5	6	n - 1

n - 1   0

21   5

6   1

18

5   2

44

4   3
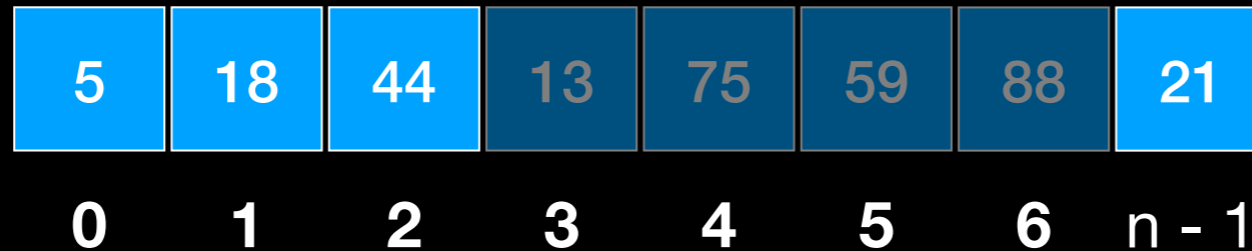
Circular Array Implementation

dequeue   `front = (front + 1) % n`

front = n-1

back = 2

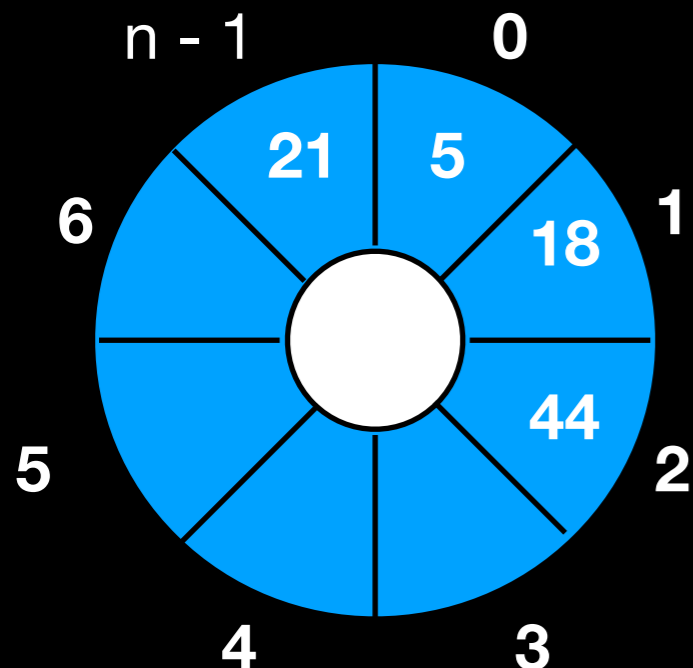| 5	18	44	13	75	59	88	21
0	1	2	3	4	5	6	n - 1

n - 1   0

21   5

6   18   1

44

5   2

4   3
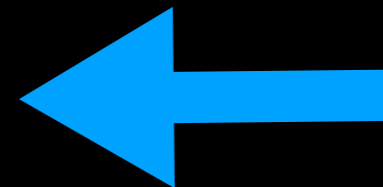
Circular Array Implementation

dequeue

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

front = 0

back = 2
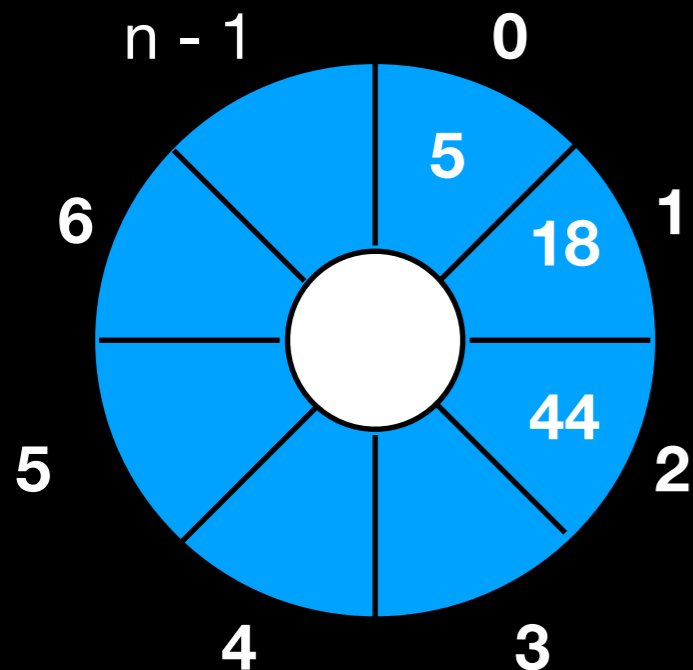
5	18	44	13	75	59	88	21
0	1	2	3	4	5	6	n - 1

Circular Array Implementation

front = 2

back = 2

5	18	44	13	75	59	88	21

0 **1** **2** **3** **4** **5** **6** n - 1

n - 1   **0**
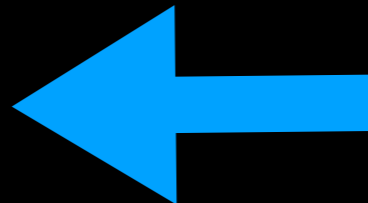
6   **1**

5   **2**

4   **3**

44   ← **front**
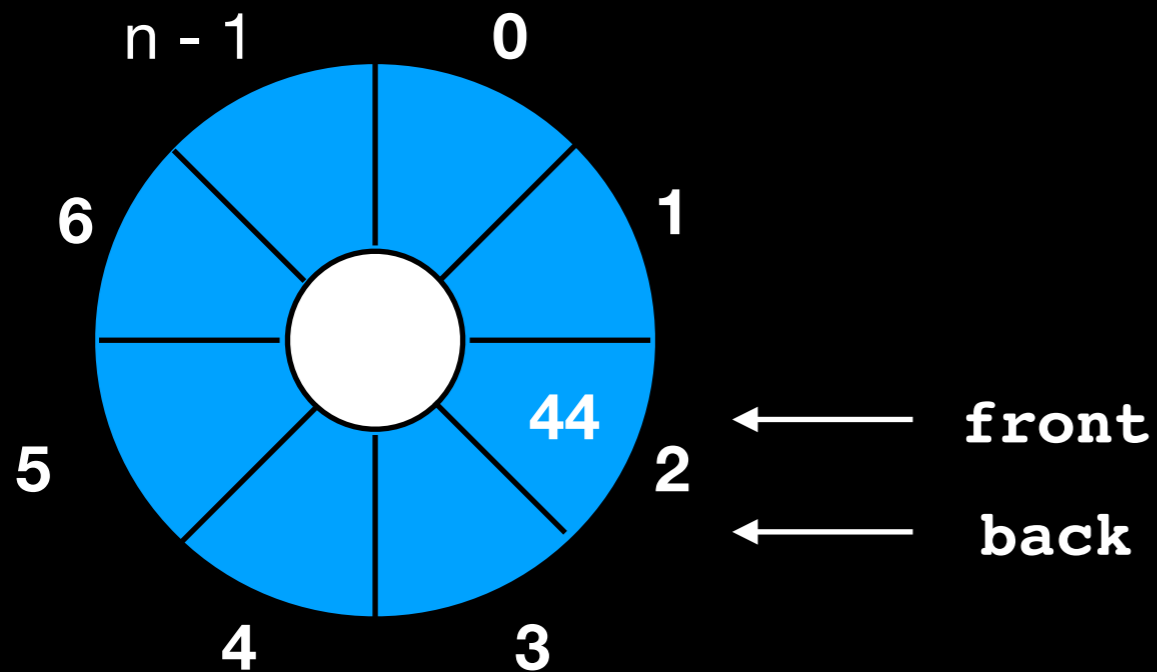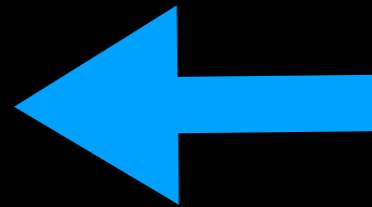
← **back**

Circular Array Implementation

dequeue   `front = (front + 1) % n`

front = 3

back = 2

5	18	44	13	75	59	88	21
0	1	2	3	4	5	6	n - 1
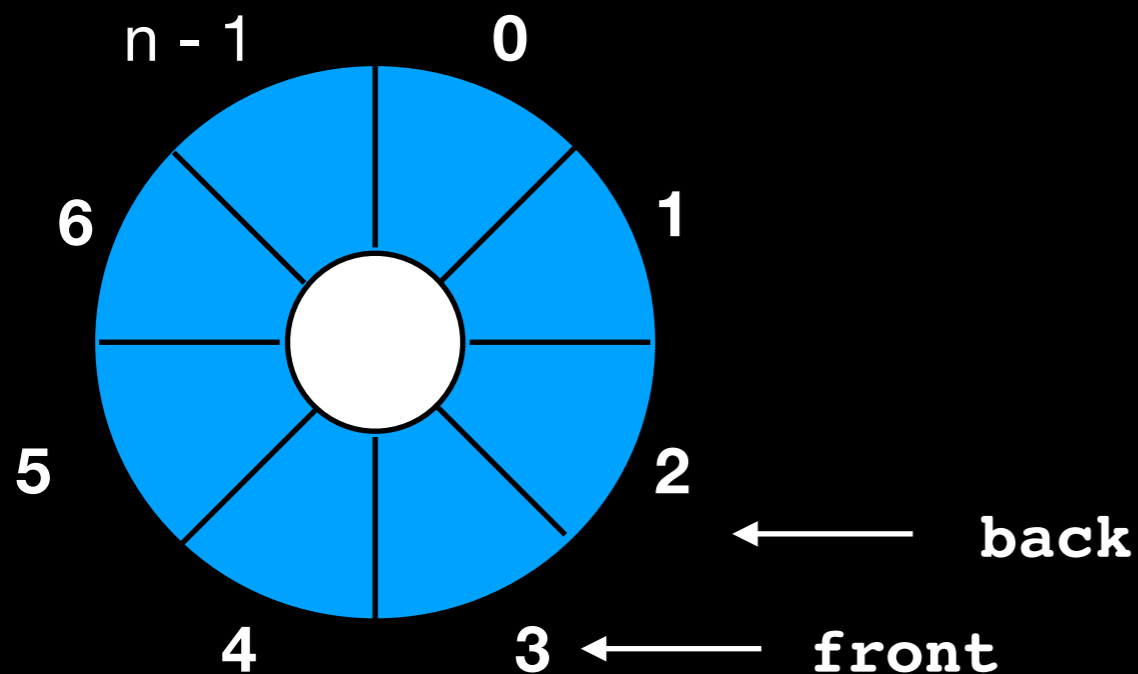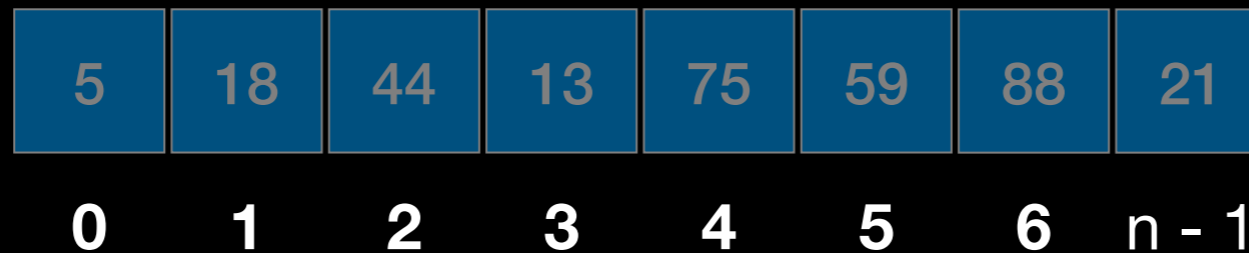
front **passes** back **when queue is EMPTY**

Circular Array Implementation

enqueue

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

front = 3

back = 1

5	18	44	13	75	59	88	21
0	1	2	3	4	5	6	n - 1

n - 1   0

21   5

6

88   18   1   ← **back**
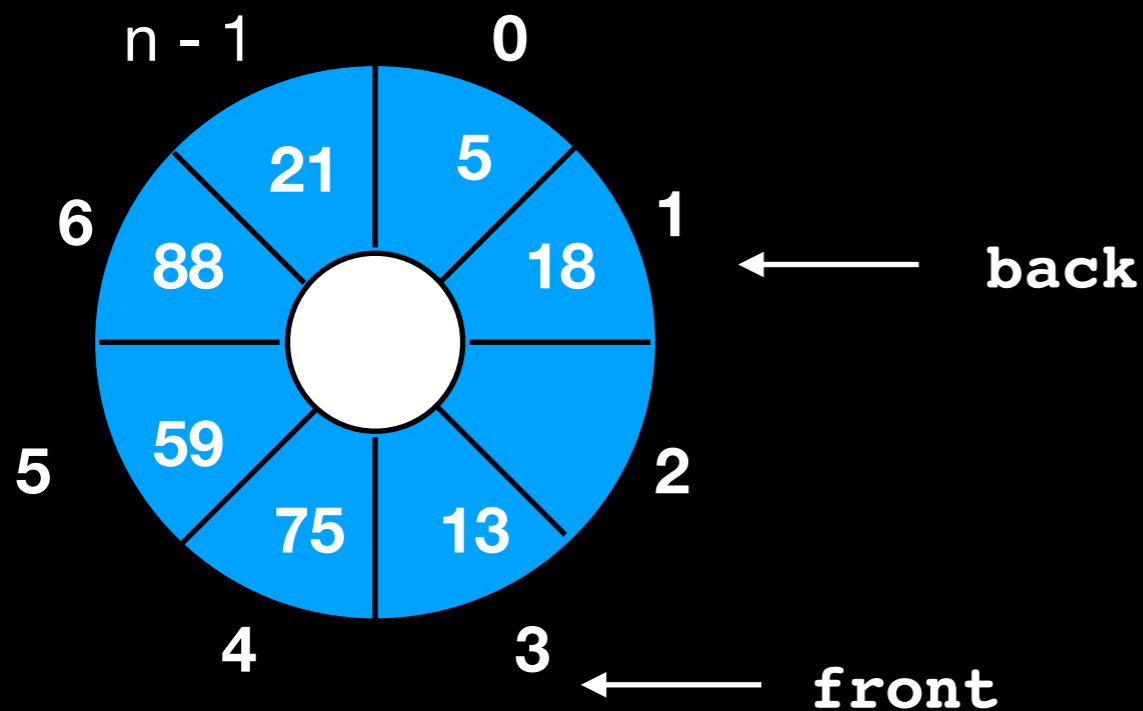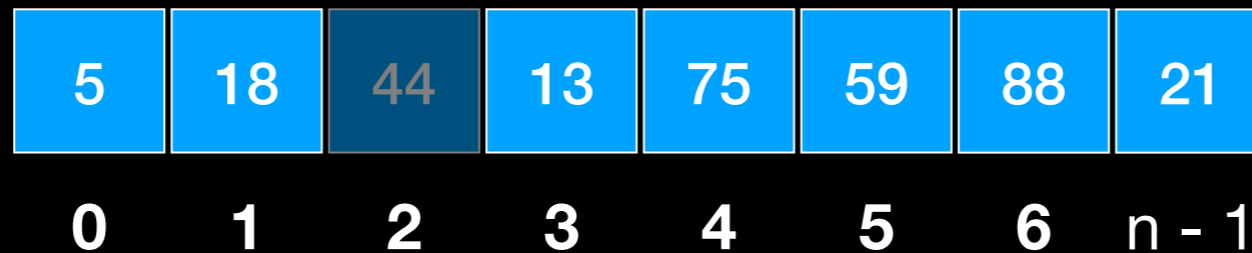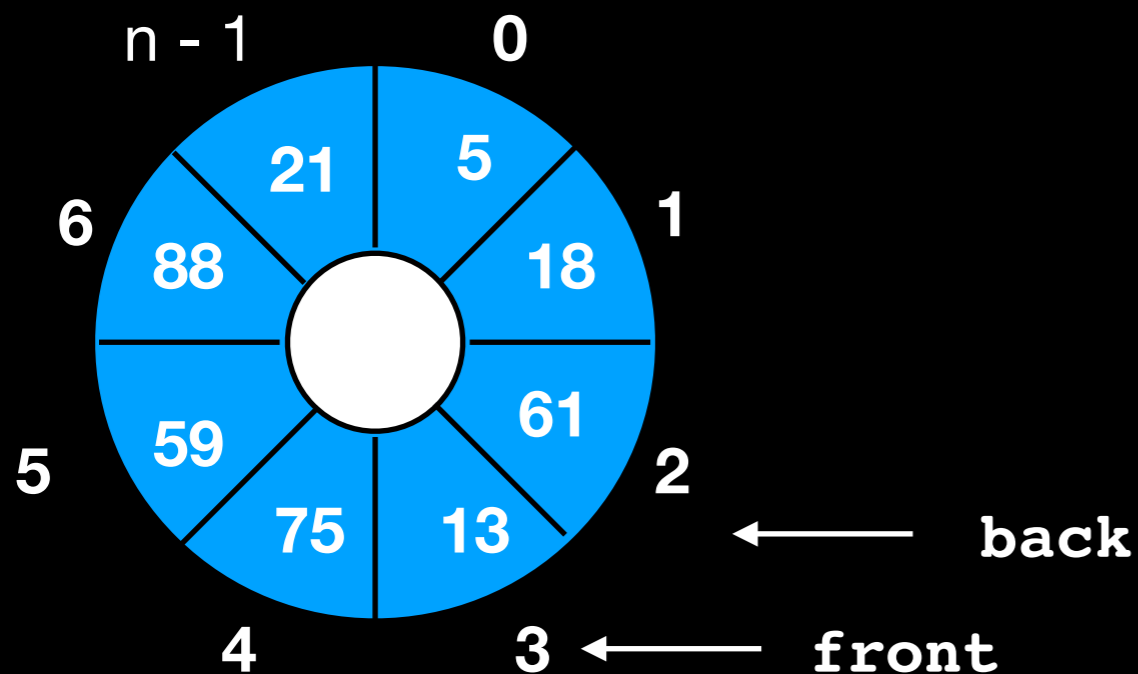
59   2

5

75   13

4   3   ← **front**

Circular Array Implementation

enqueue

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

front = 3

back = 2

5	18	61	13	75	59	88	21
0	1	2	3	4	5	6	n - 1
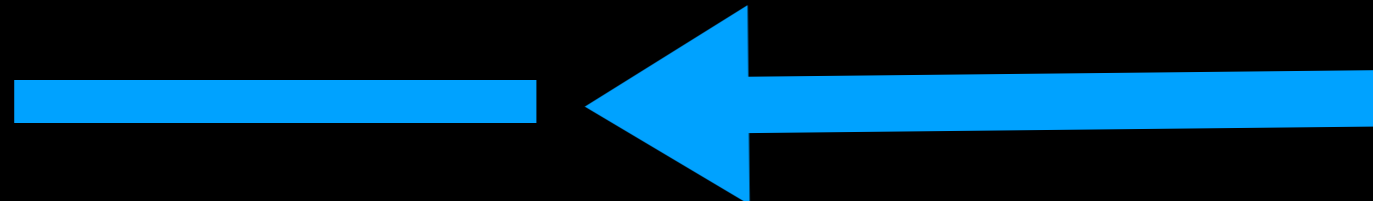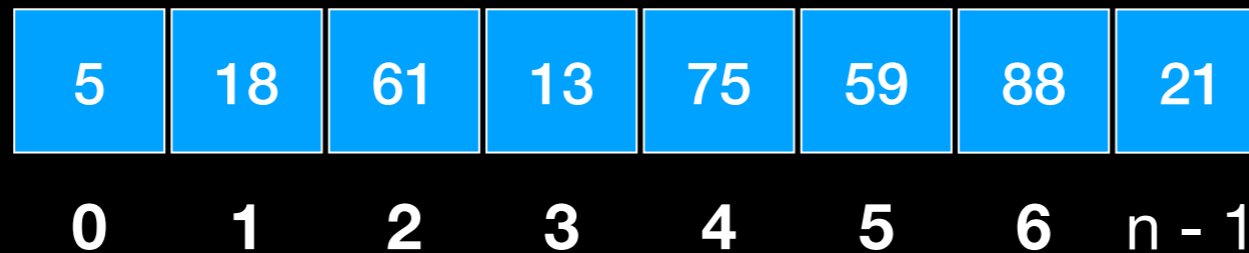
front passes back ALSO
when queue is FULL

n - 1   0

6   21   5   1

88   18

59   61

5   75   13   2

4   3

← back

← front

Circular Array Implementation

enqueue

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

front = 3

back = 2

5	18	61	13	75	59	88	21
0	1	2	3	4	5	6	n - 1
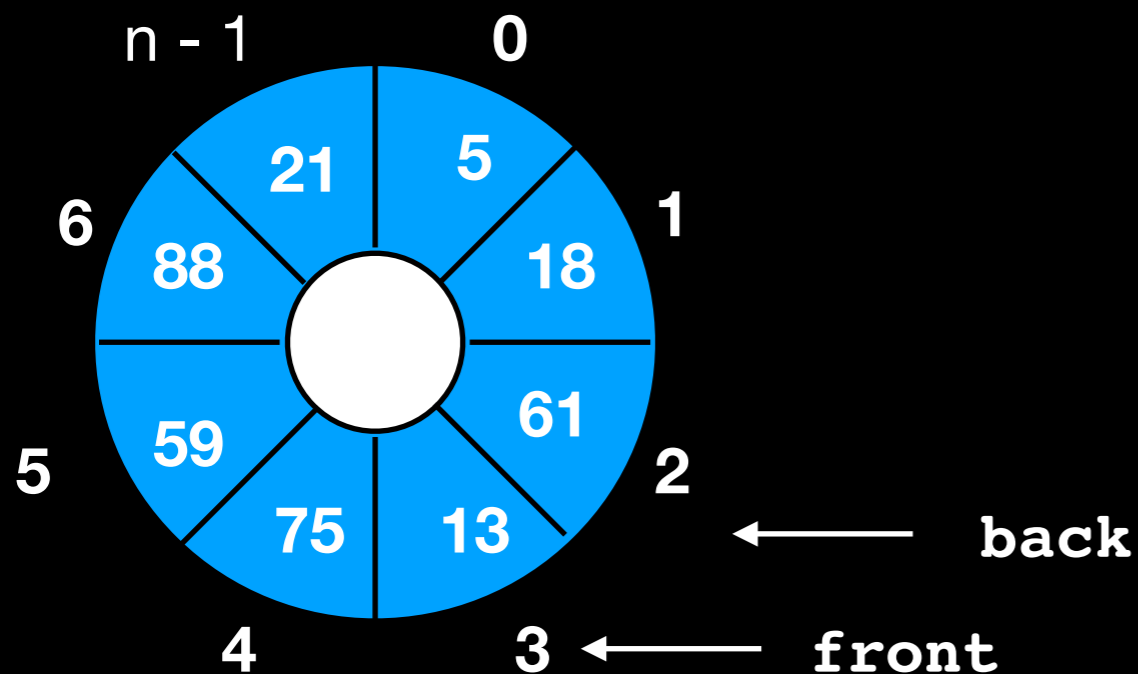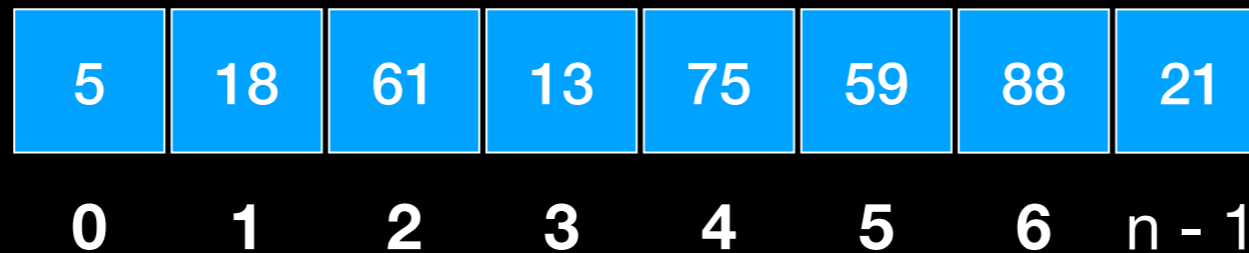
To distinguish between **empty** and **full** queue must keep a **COUNTER** for number of items

back

front