Queue Implementations

Tiziana Ligorio

tligorio@hunter.cuny.edu

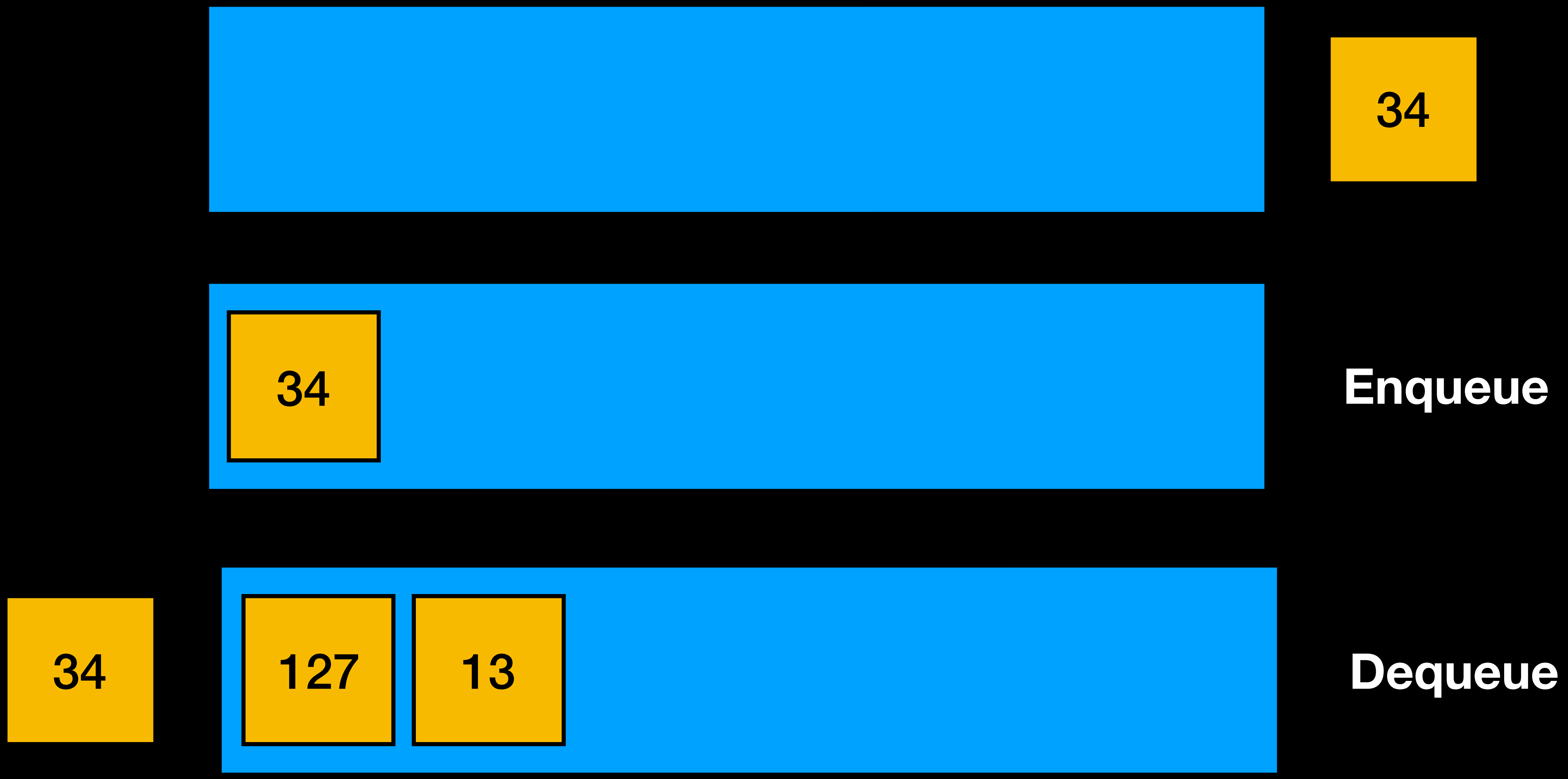
Today's Plan

Recap

Queue Implementations

Recap

FIFO structure: First In First Out



Enqueue

Dequeue

Queue ADT

```
#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

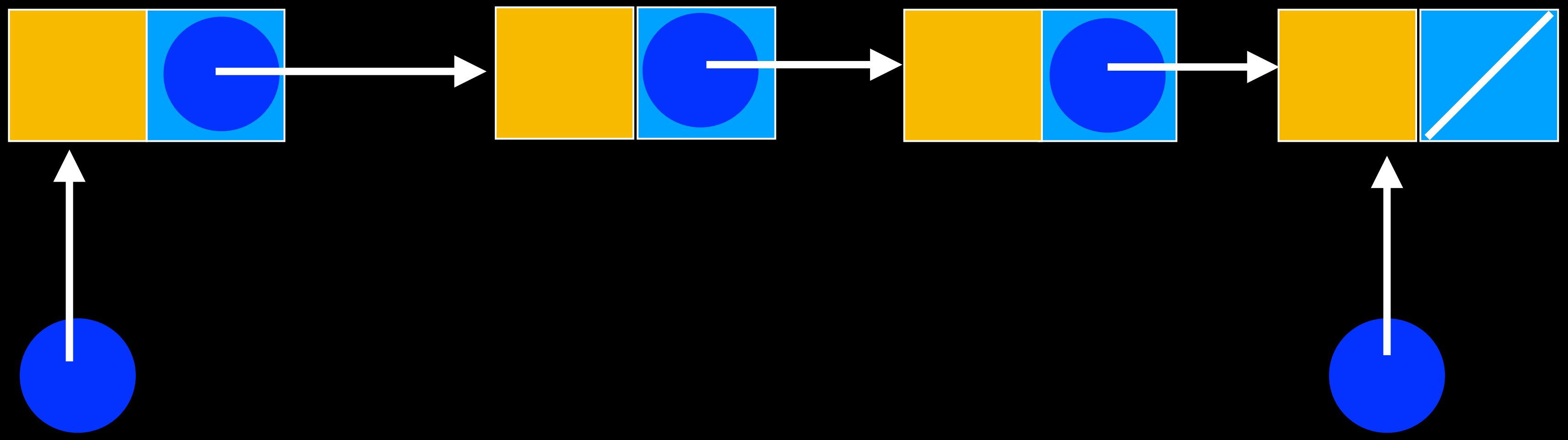
Amortized Analysis

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

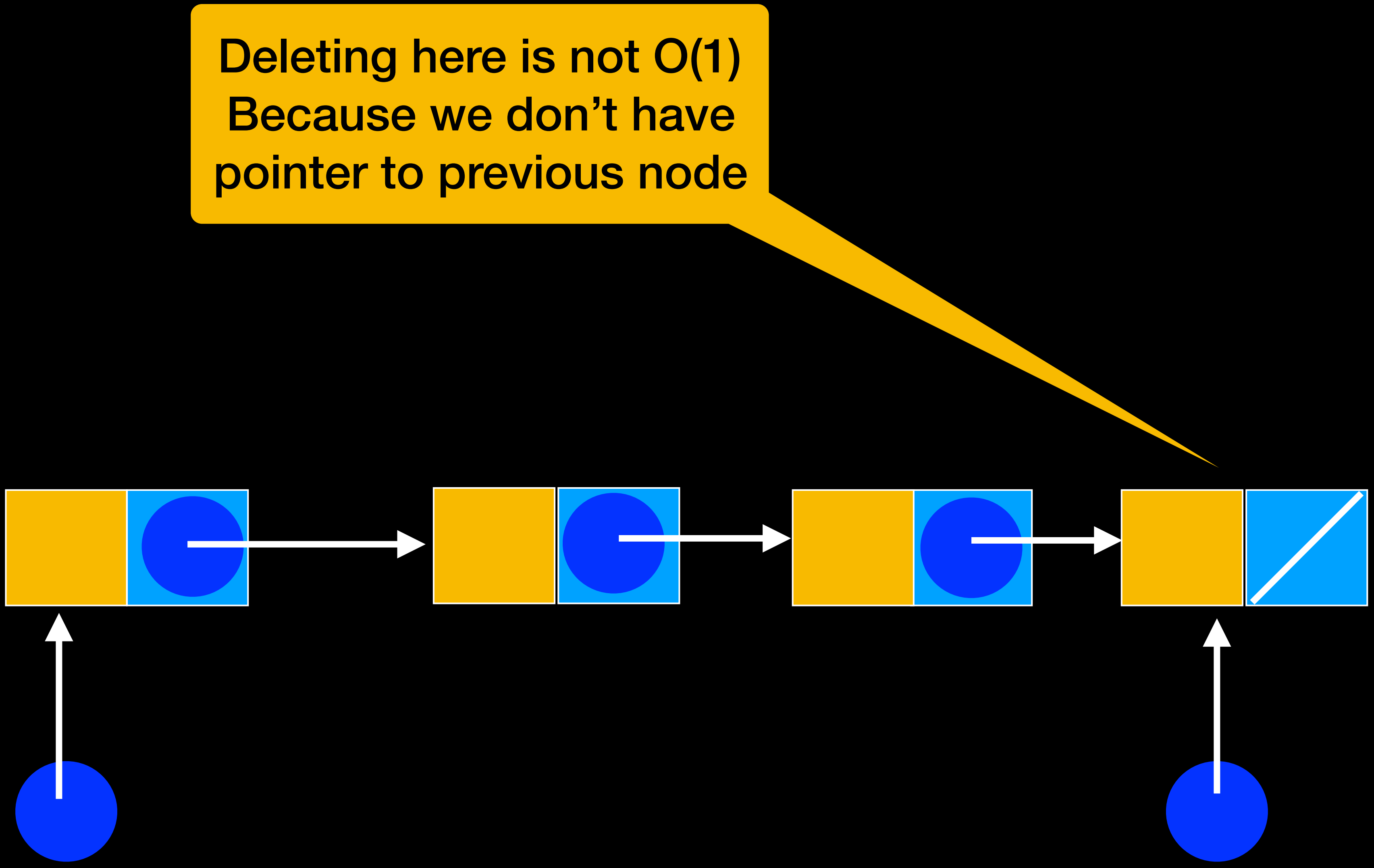
What is the main difference btw stack and queue?

Singly Linked Chain

Where is front?
Where is back?

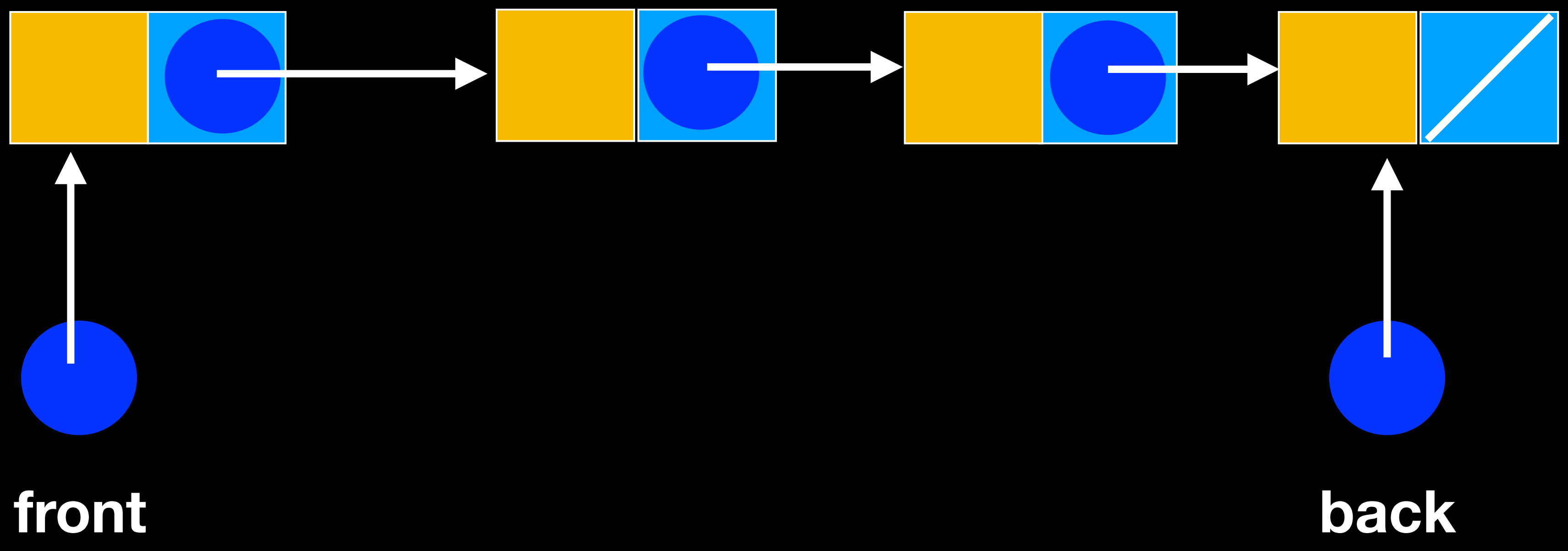


Singly Linked Chain



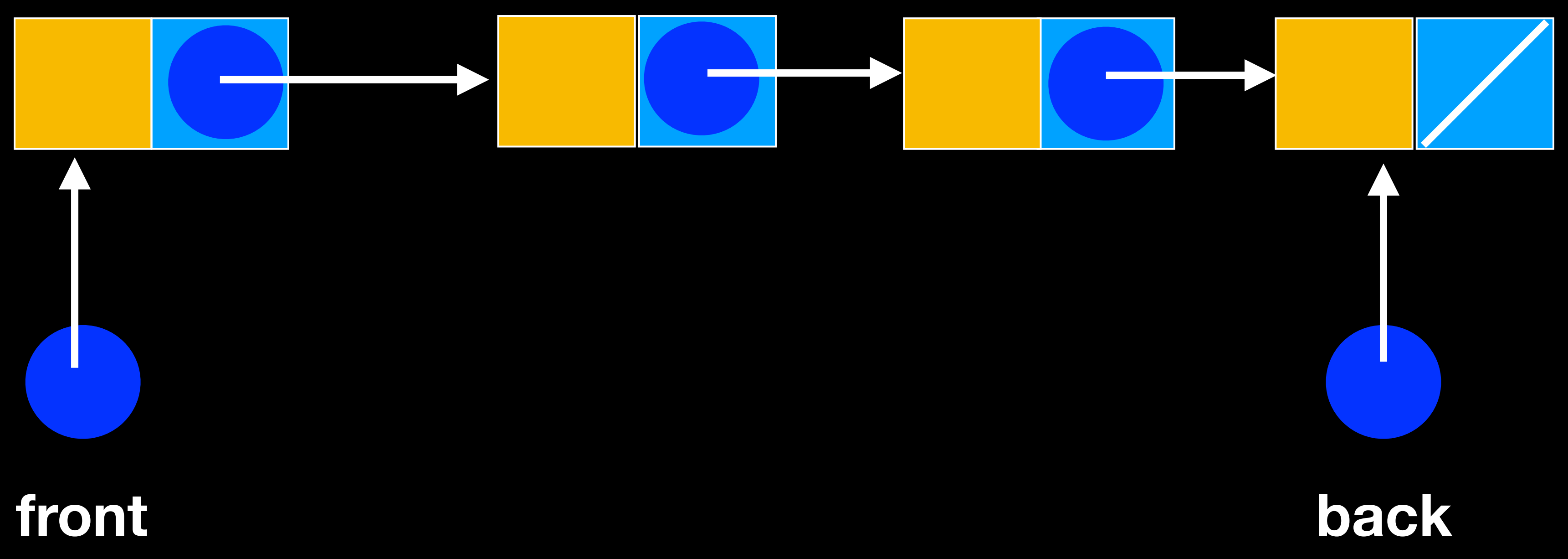
Deleting here is not O(1)
Because we don't have
pointer to previous node

Singly Linked Chain



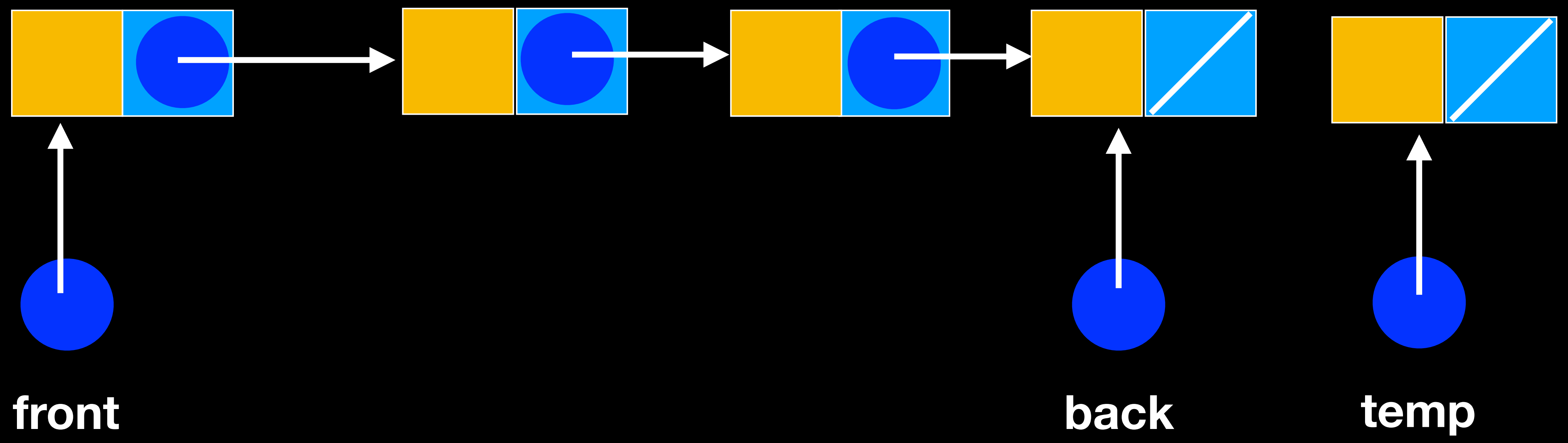
Singly Linked Chain

enqueue

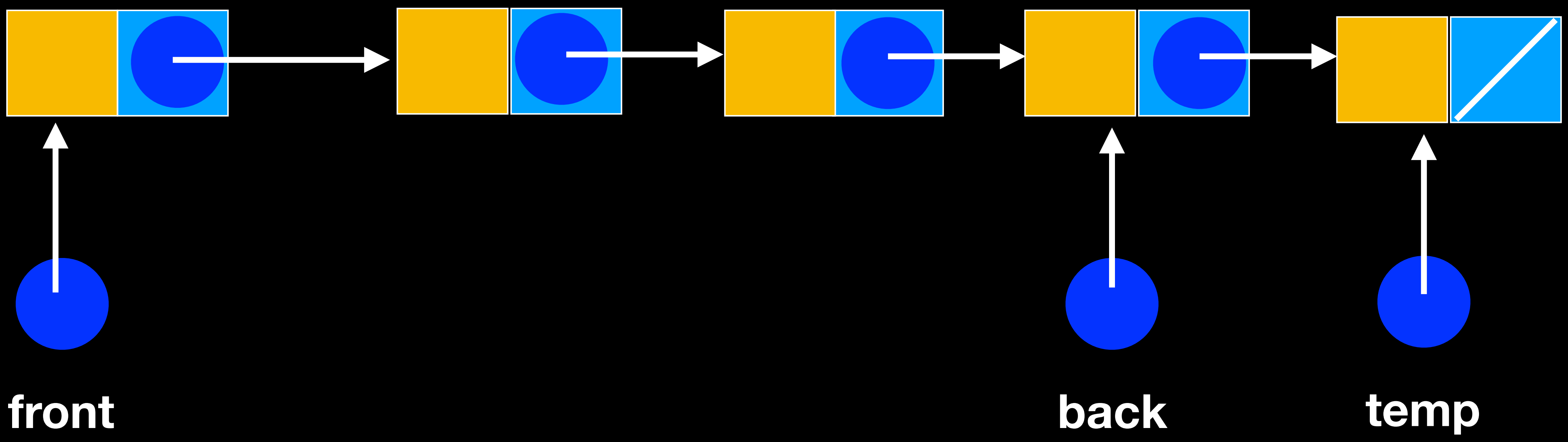


Singly Linked Chain

enqueue

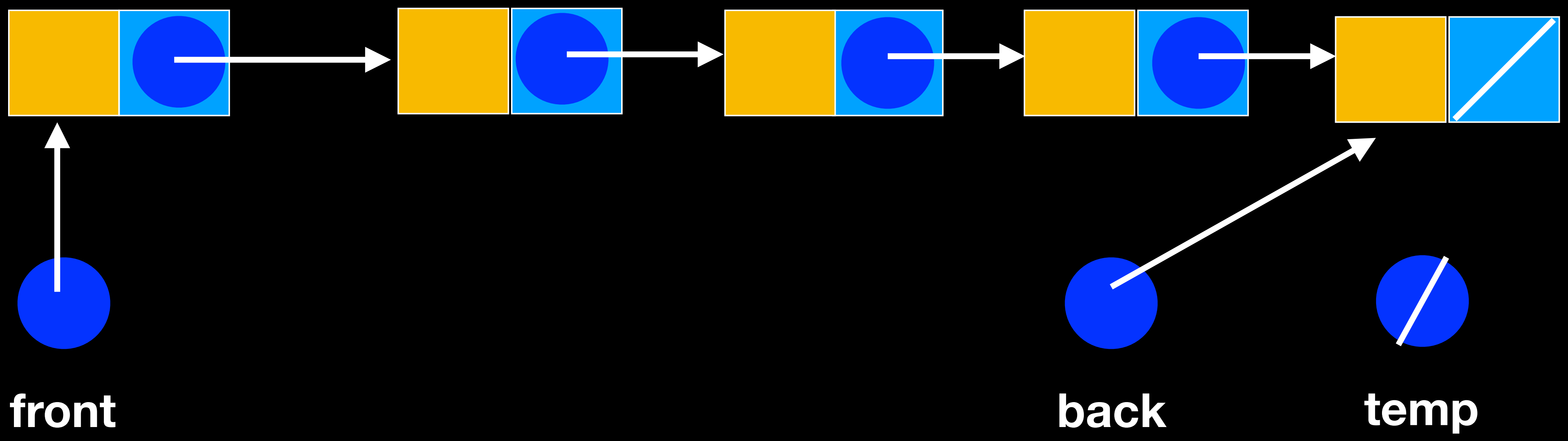


enqueue



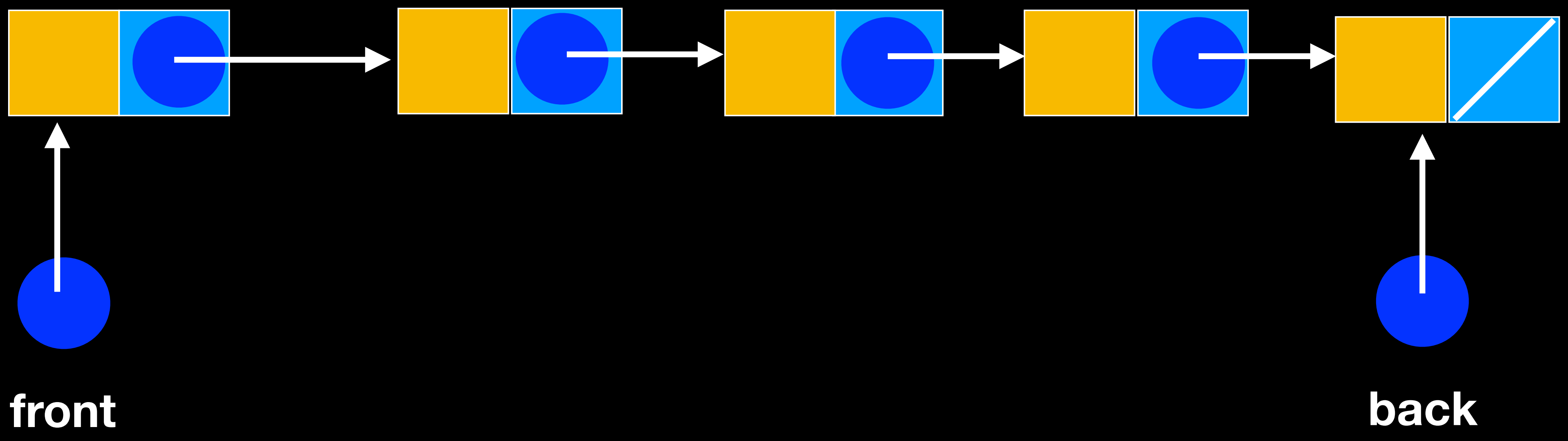
Singly Linked Chain

enqueue



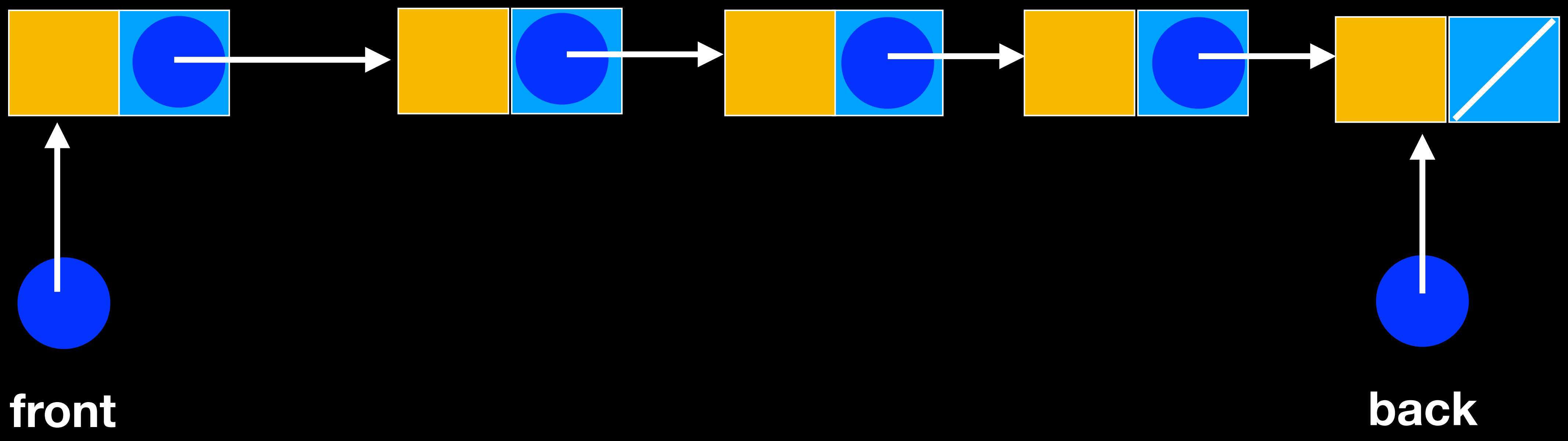
Singly Linked Chain

enqueue



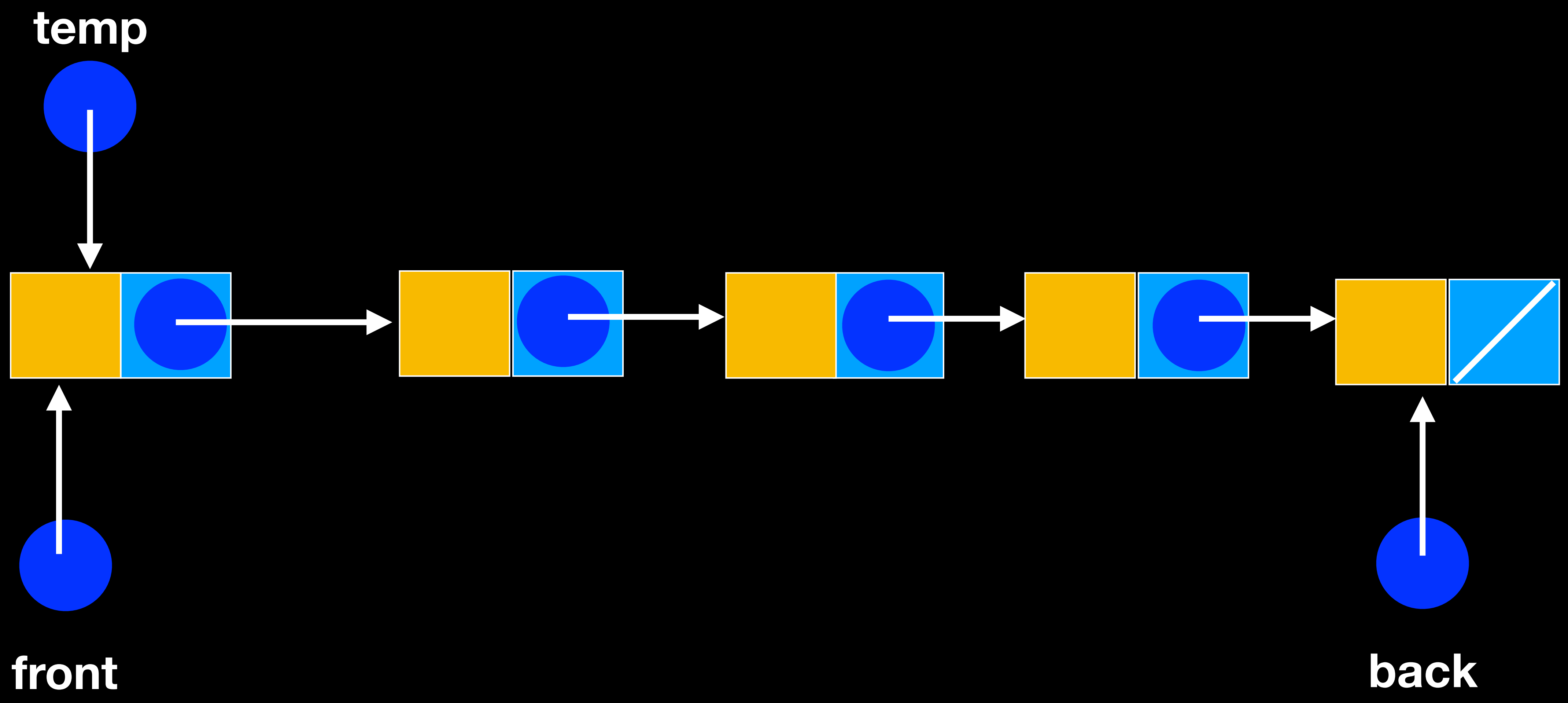
Singly Linked Chain

dequeue

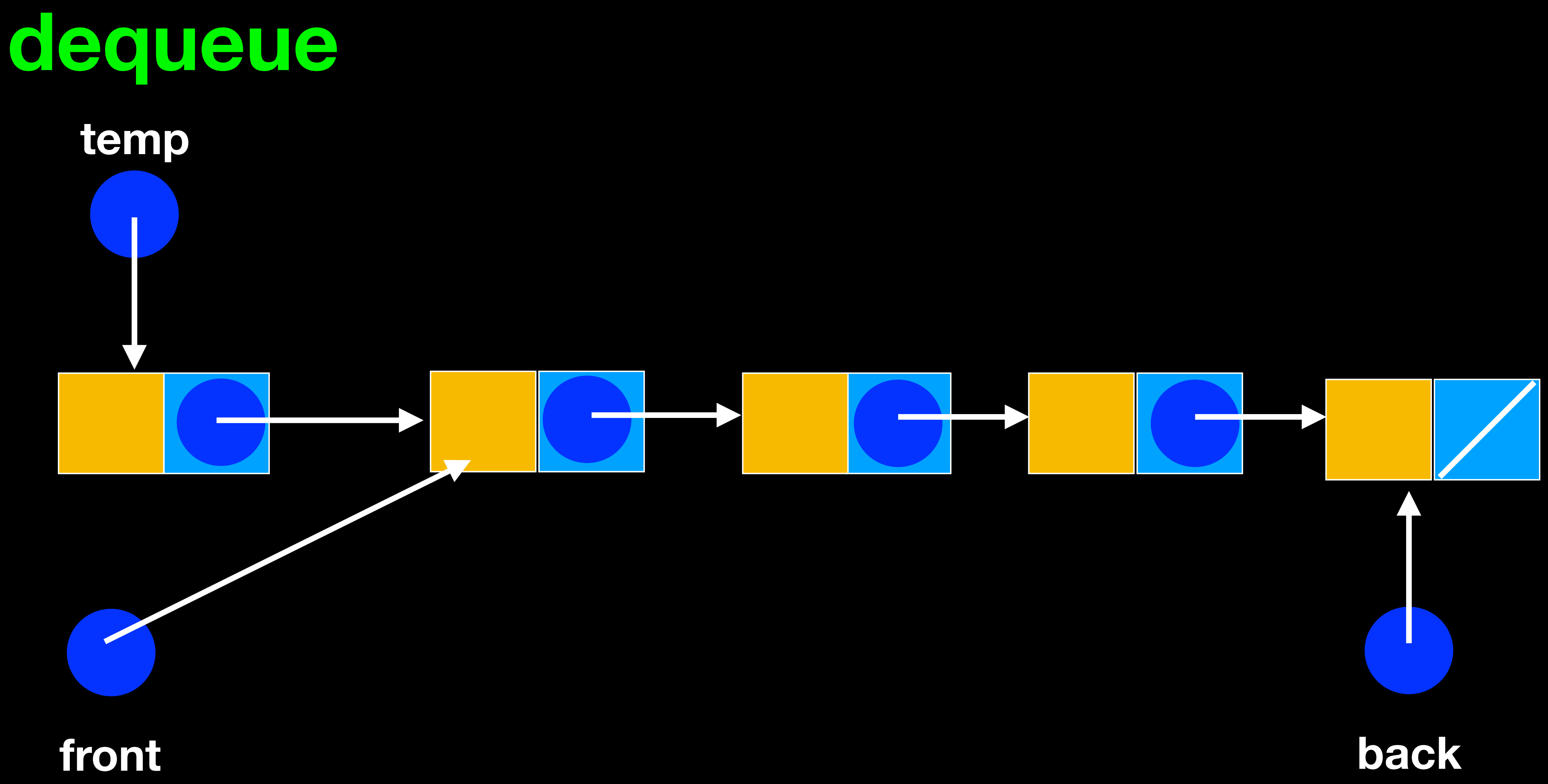


Singly Linked Chain

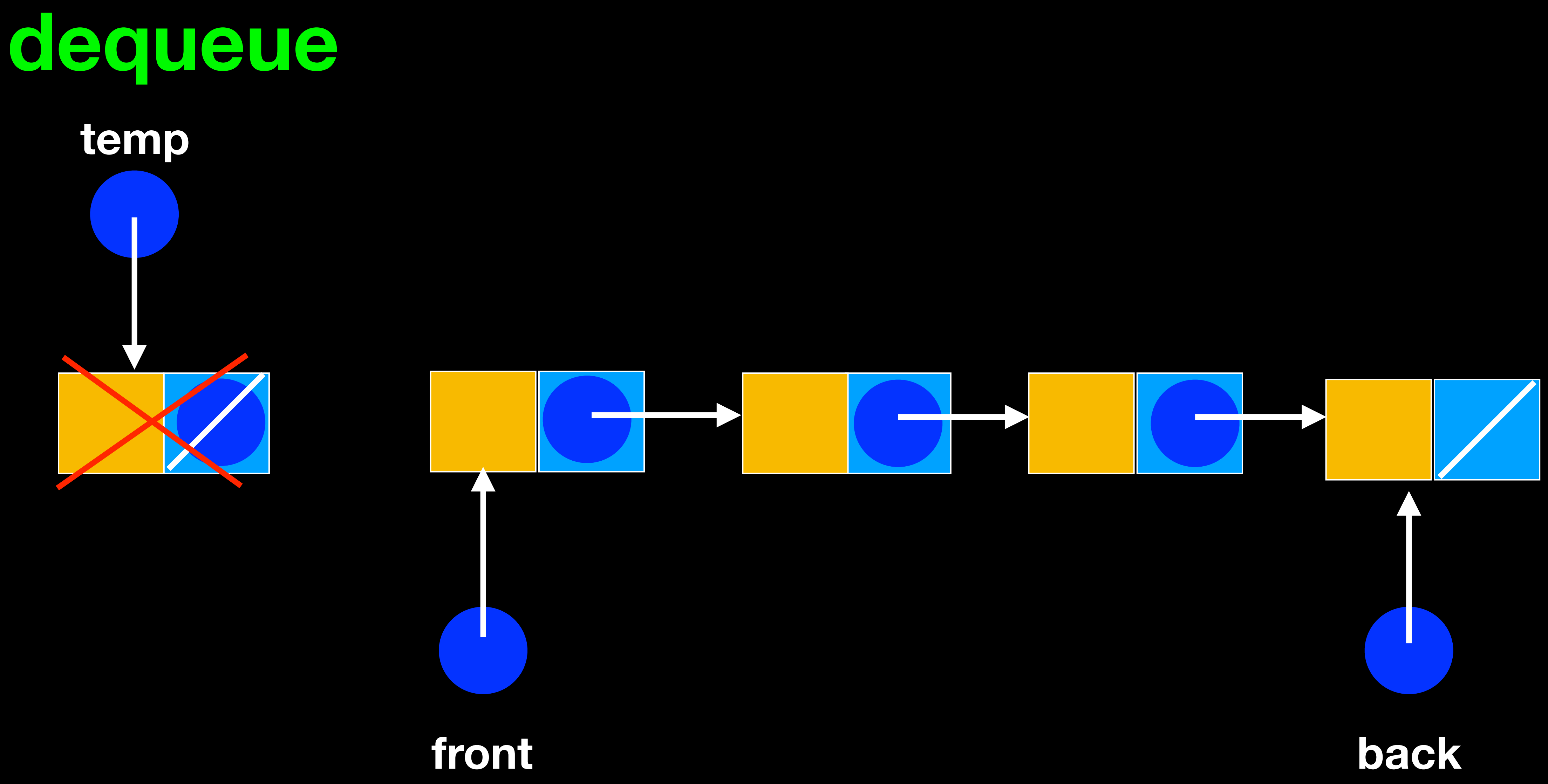
dequeue



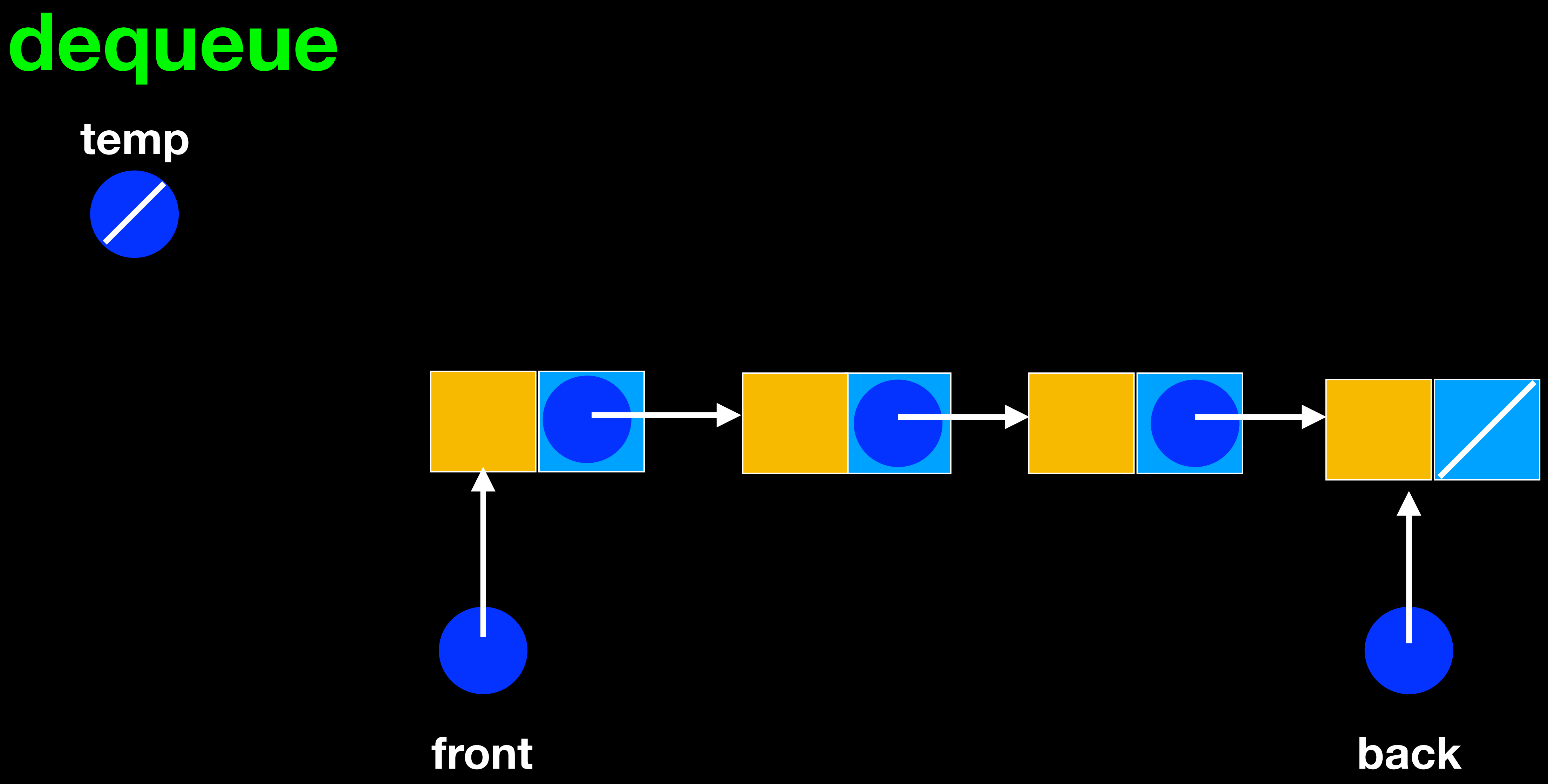
Singly Linked Chain



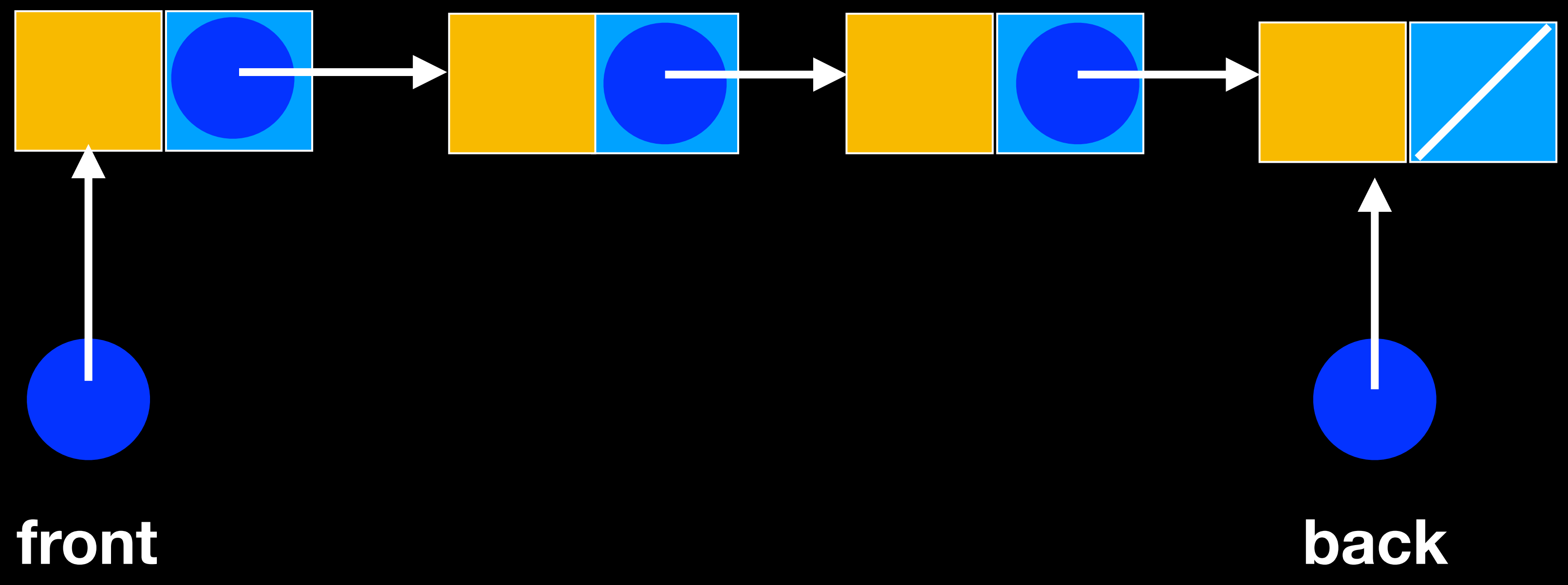
Singly Linked Chain



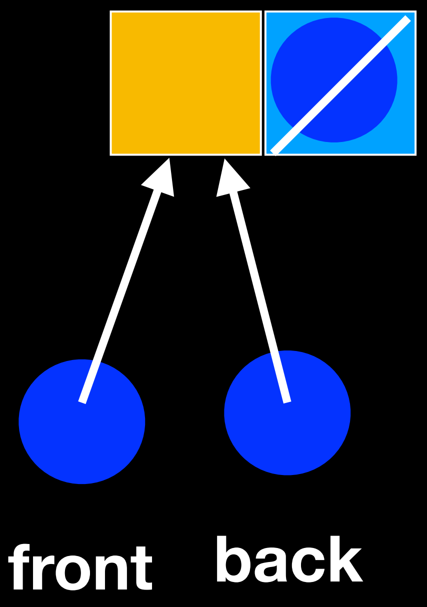
Singly Linked Chain



Singly Linked Chain



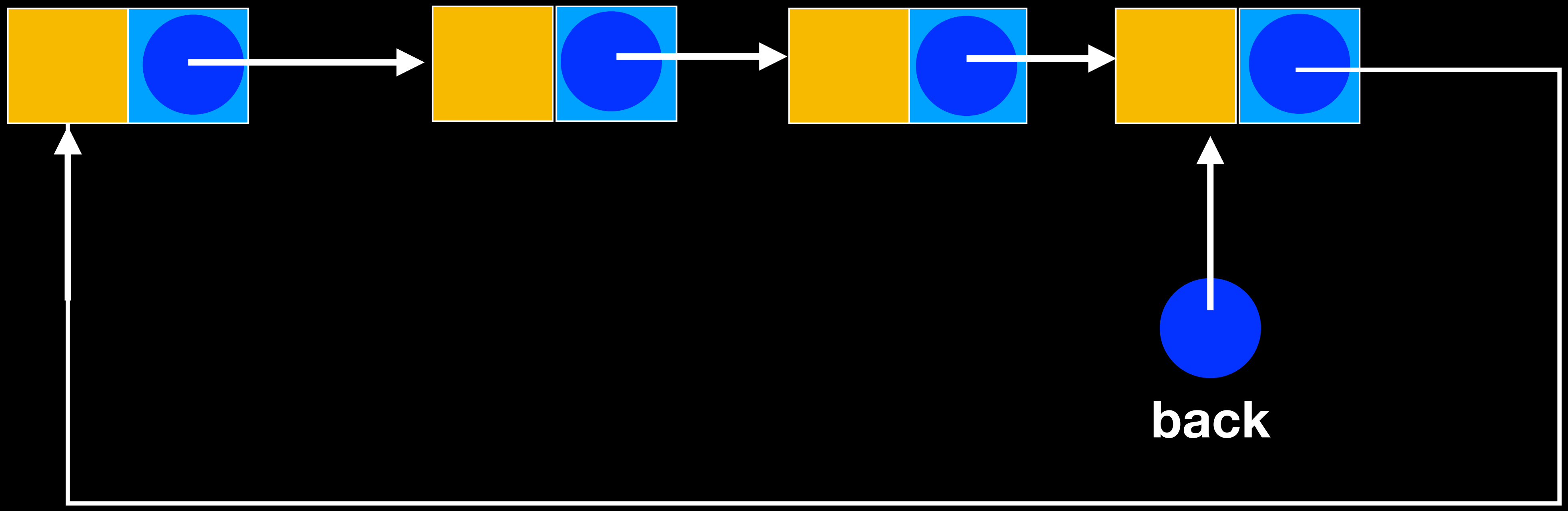
Singly Linked Chain



That’s it!

Singly Linked Chain

An Alternative:
A Circular Linked Chain

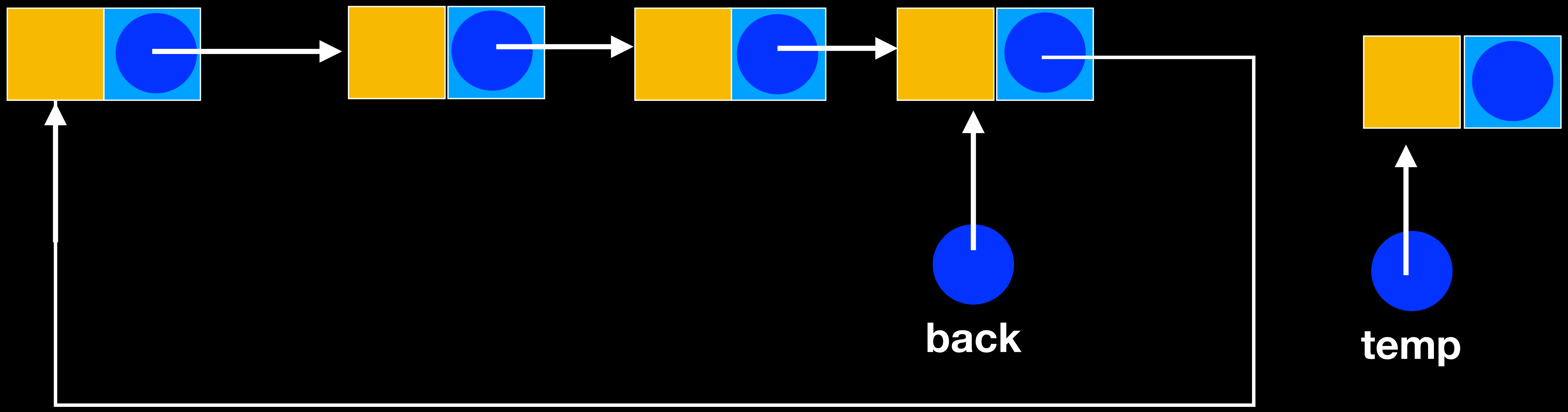


Singly Linked Chain

enqueue

An Alternative:
A Circular Linked Chain

Instantiate new node

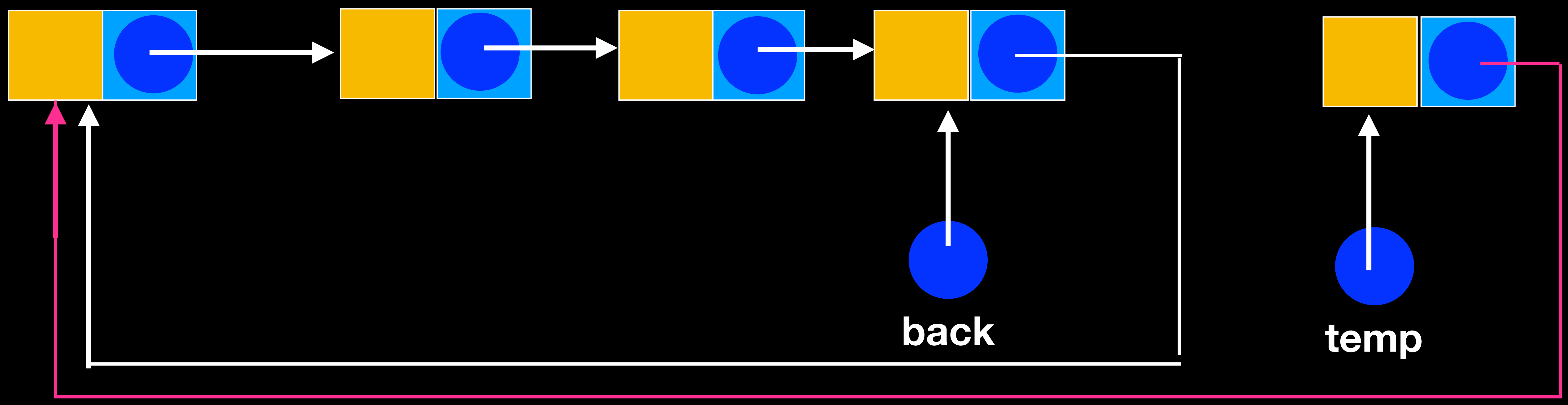


Singly Linked Chain

enqueue

An Alternative:
A Circular Linked Chain

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

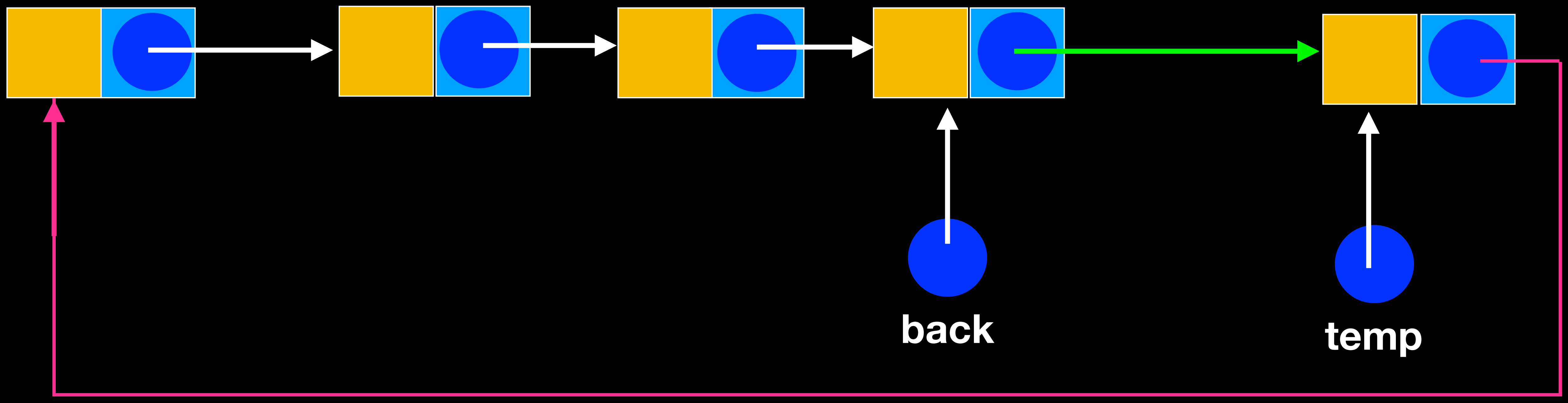


Singly Linked Chain

enqueue

An Alternative:
A Circular Linked Chain

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

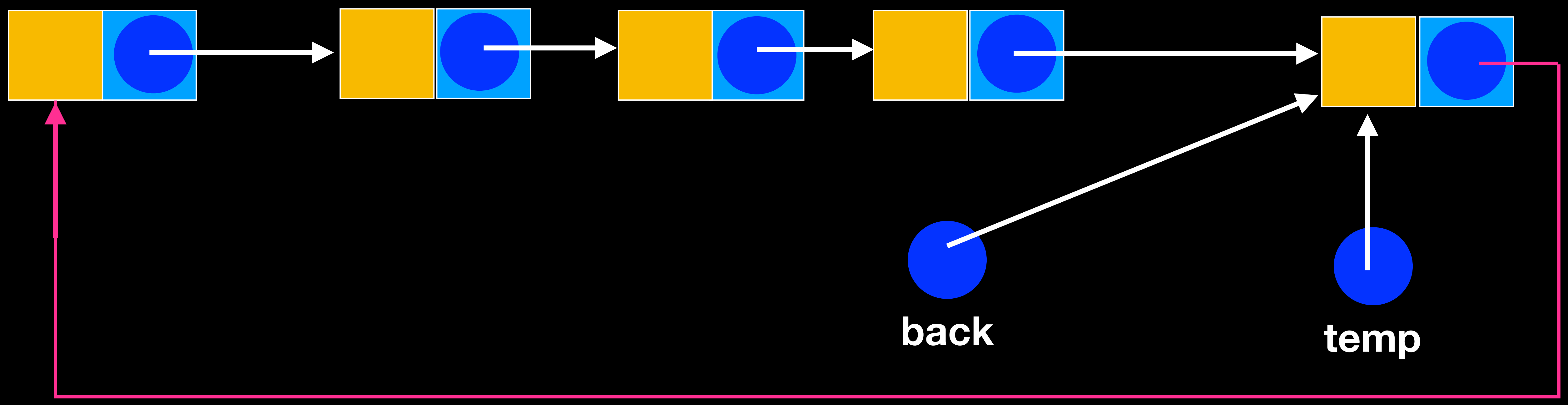


Singly Linked Chain

enqueue

An Alternative:
A Circular Linked Chain

```
back = temp;
```

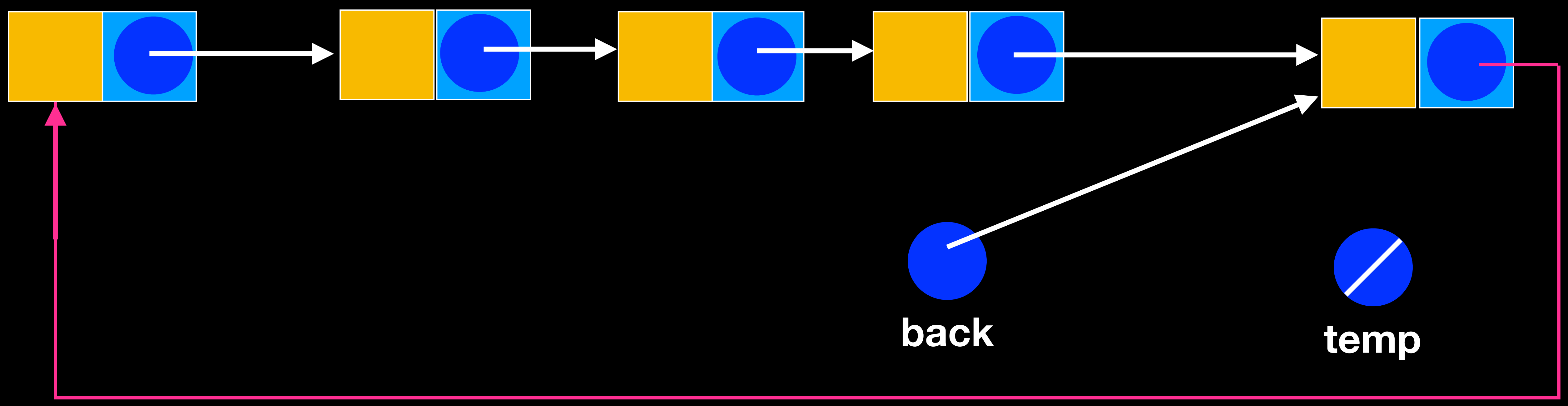


Singly Linked Chain

enqueue

**An Alternative:
A Circular Linked Chain**

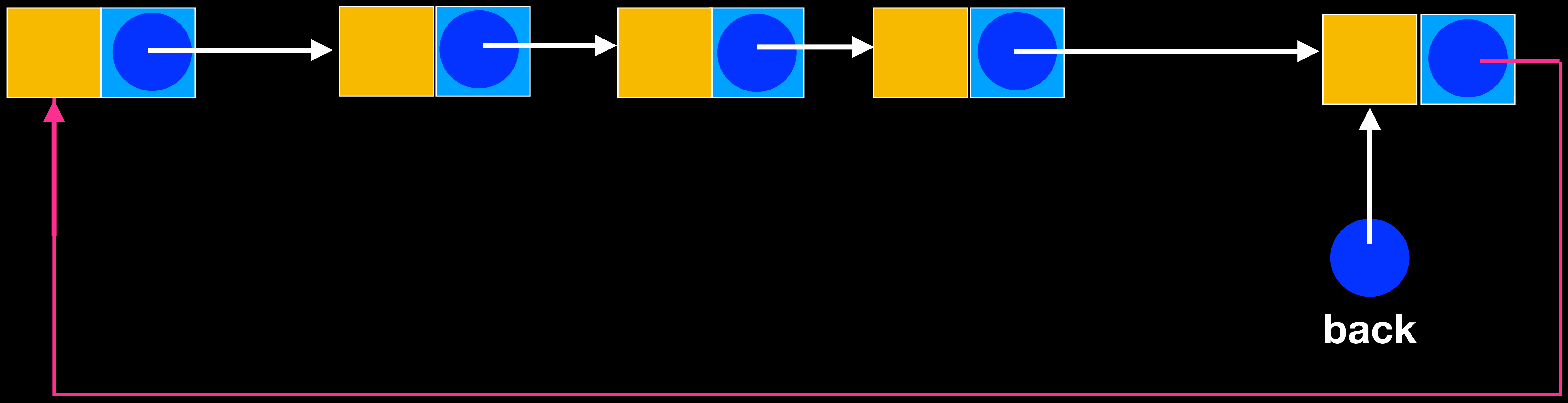
```
temp = nullptr;
```



Singly Linked Chain

enqueue

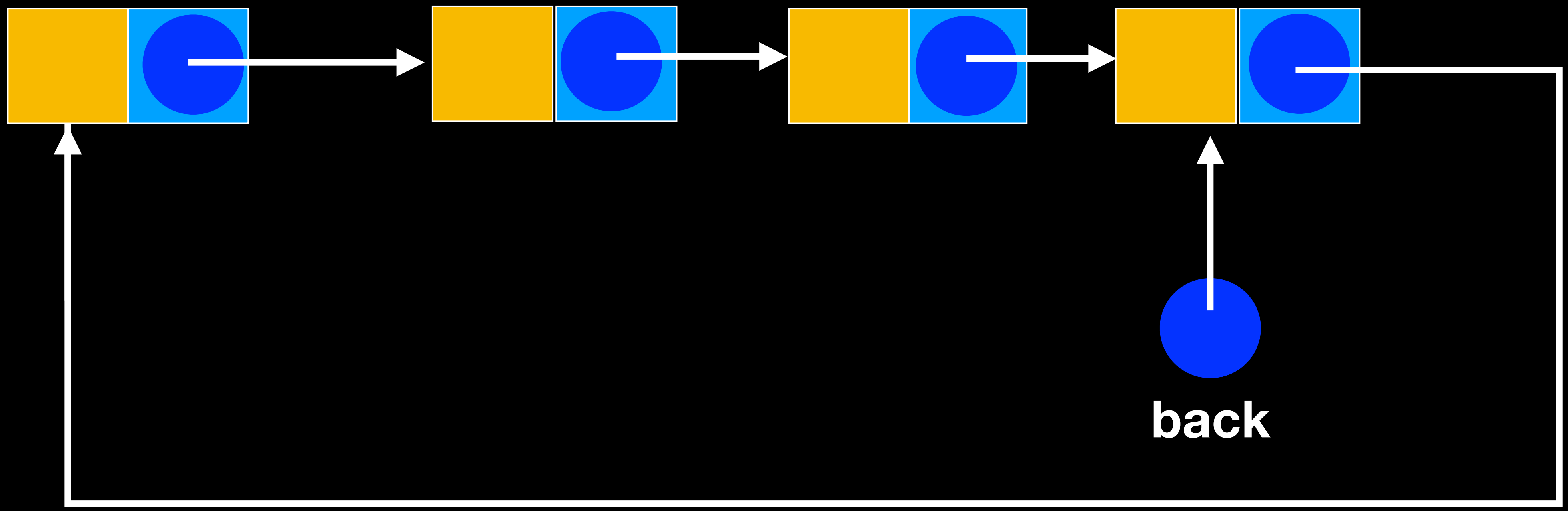
An Alternative:
A Circular Linked Chain



Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

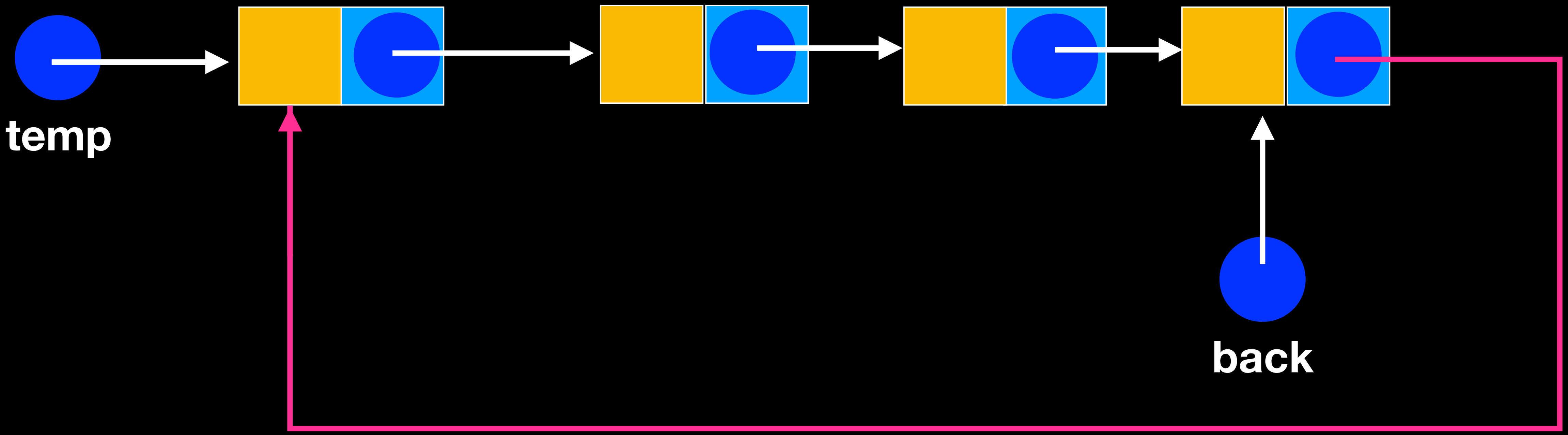


Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

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

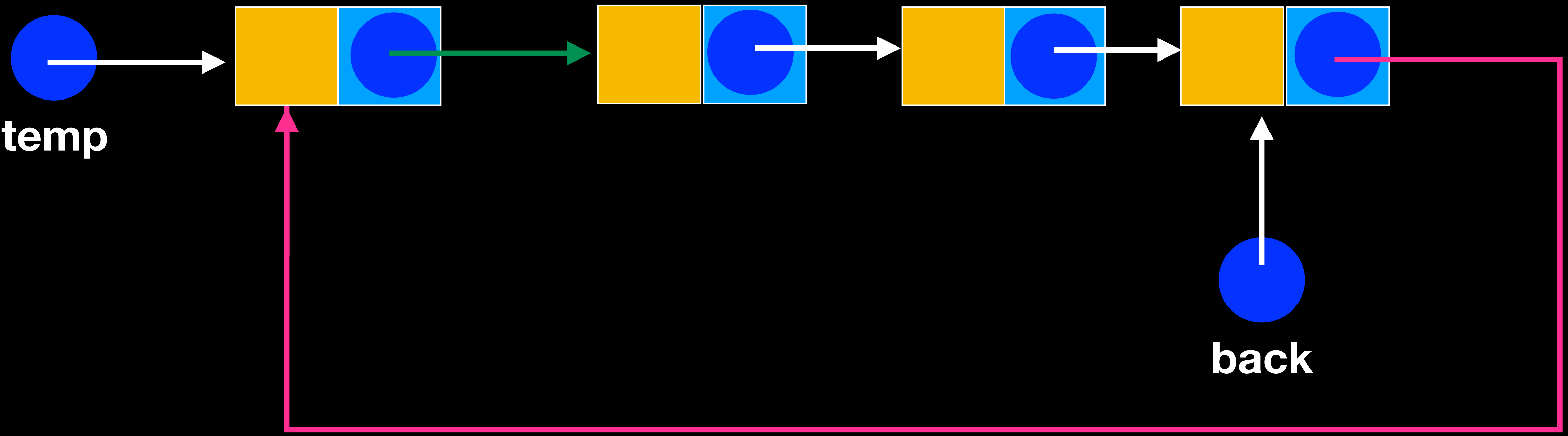


Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

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

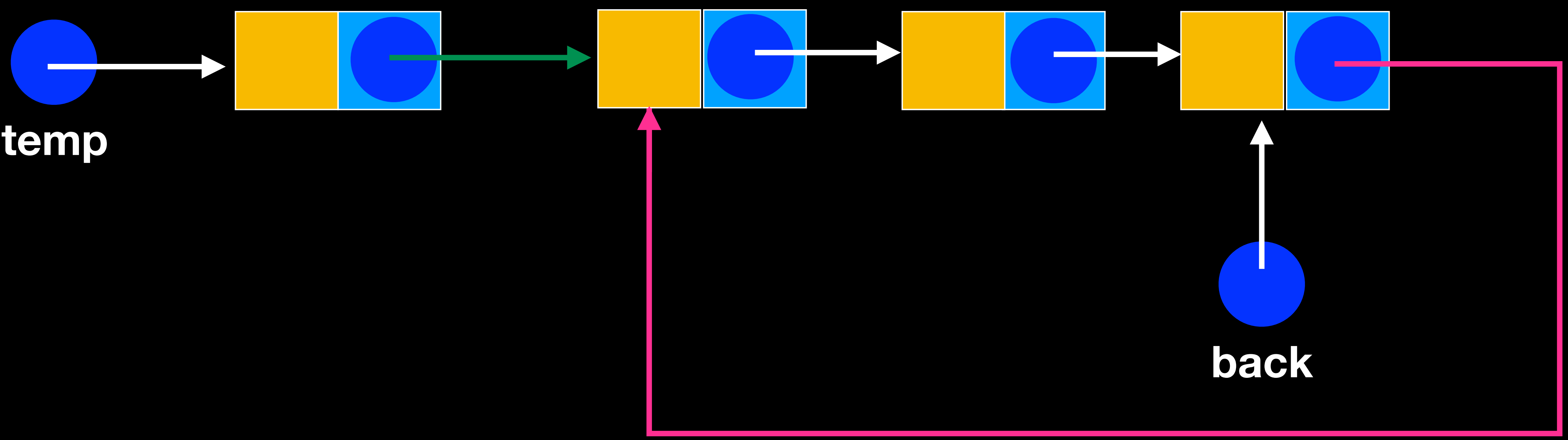


Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

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

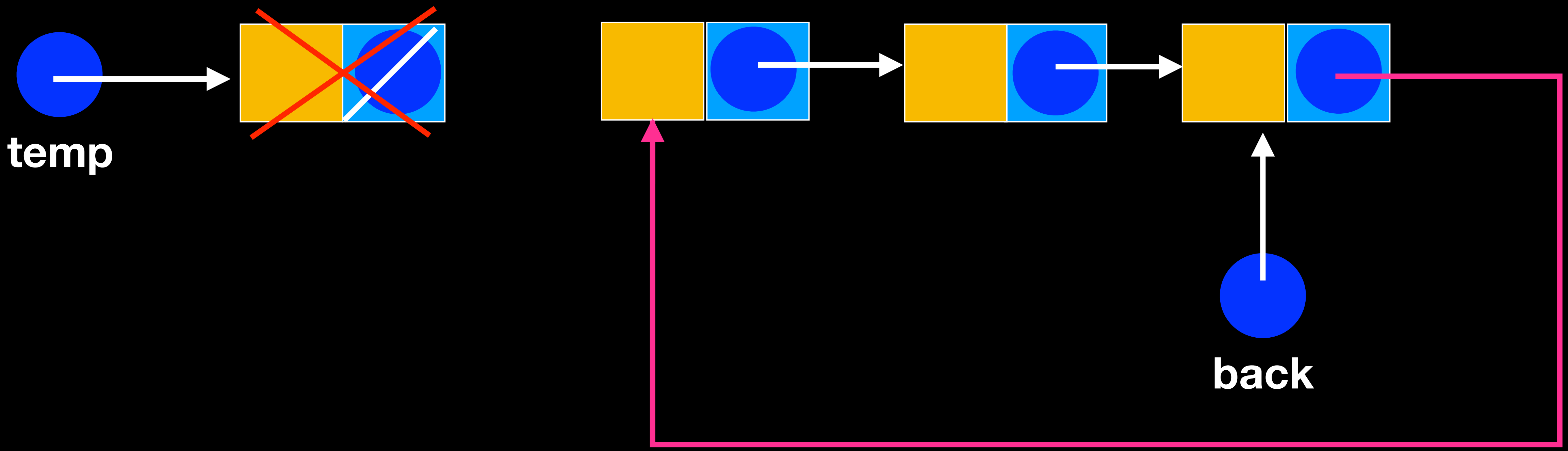


Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

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

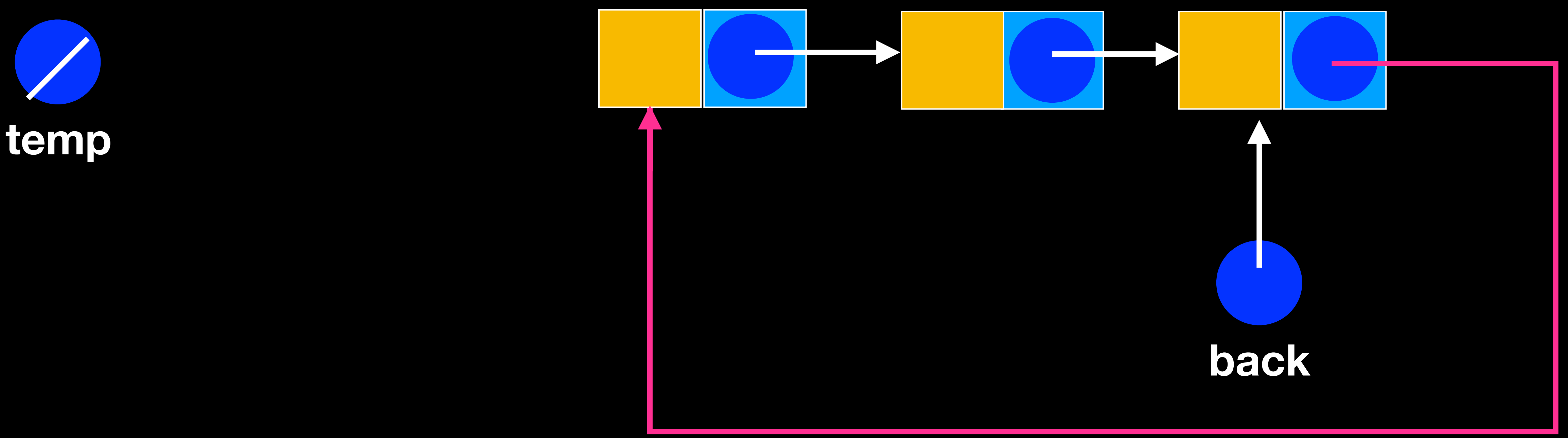


Singly Linked Chain

dequeue

An Alternative:
A Circular Linked Chain

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



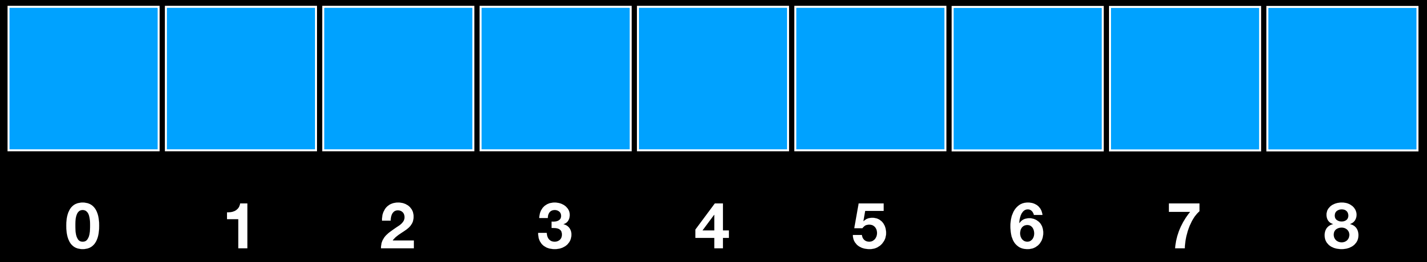
Queue ADT
(Circular Linked Chain)

```
#ifndef QUEUE_H_
#define QUEUE_H_

template<class 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_ `
```

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

Array Considerations

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

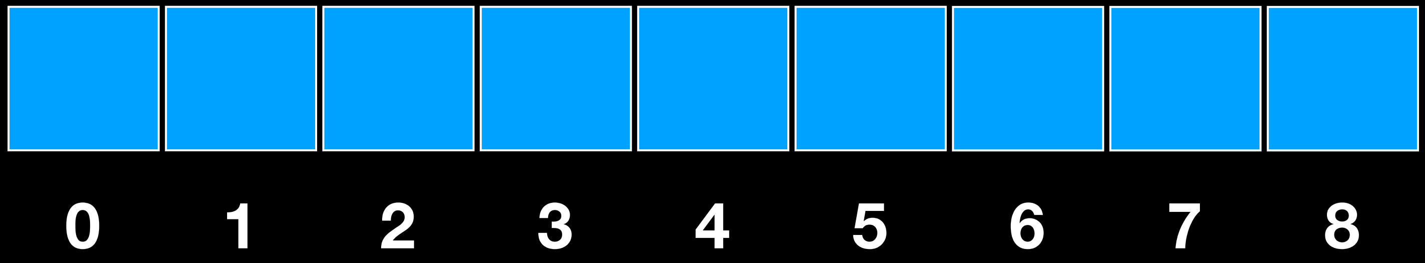


Array Considerations

enqueue

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

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

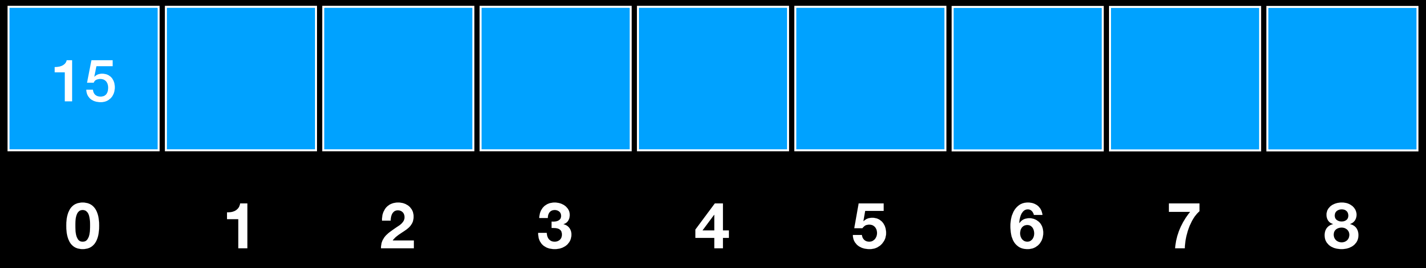


Array Considerations

enqueue

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

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



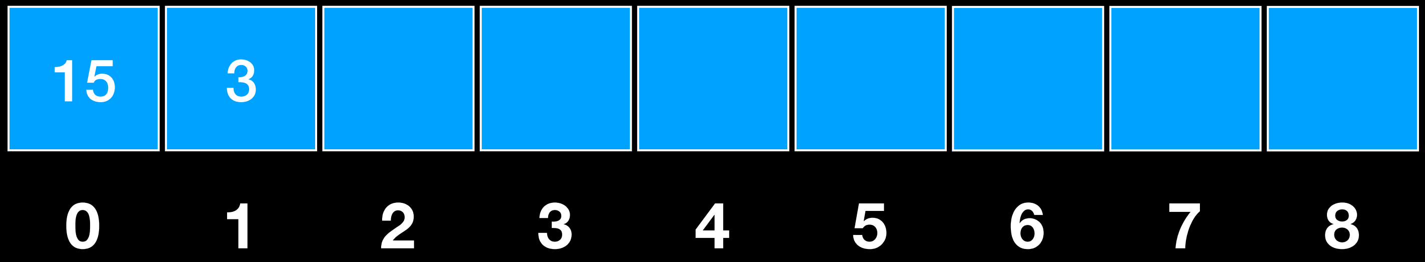
15								
0	1	2	3	4	5	6	7	8

Array Considerations

enqueue

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

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



0	1	2	3	4	5	6	7	8
15	3							

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

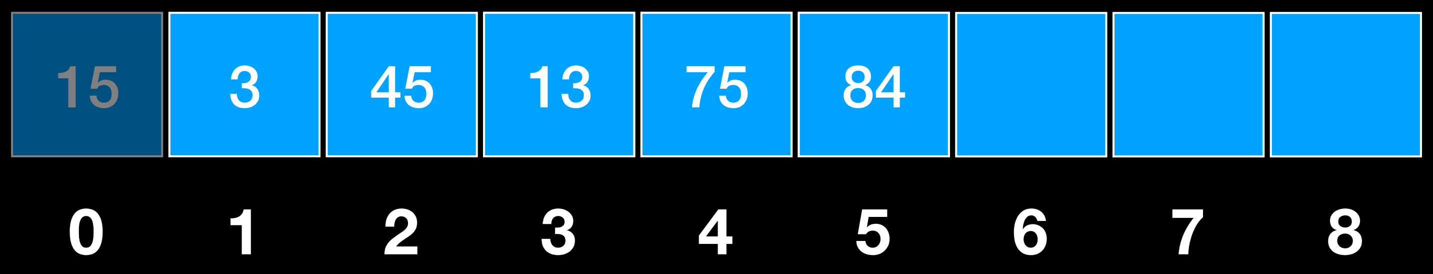
This seems to work, but what happens when we start dequeuing?

Array Considerations

dequeue

Increment `front`

```
front = 1
back = 5
```



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

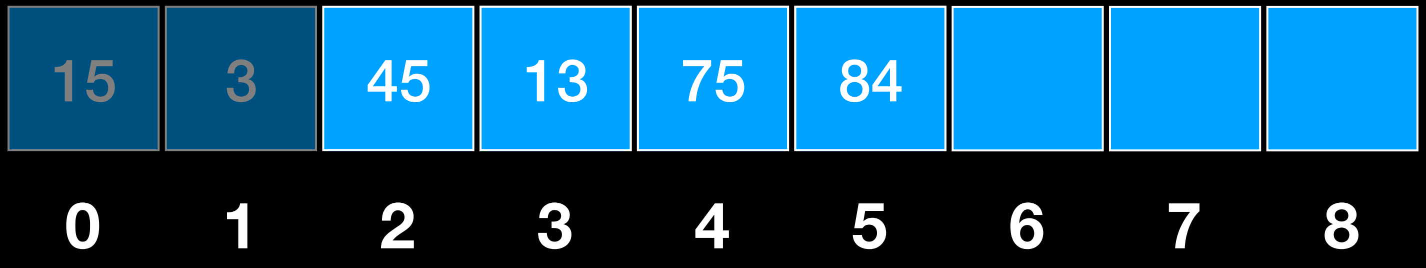
We want O(1) operations, so simply increment front!

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

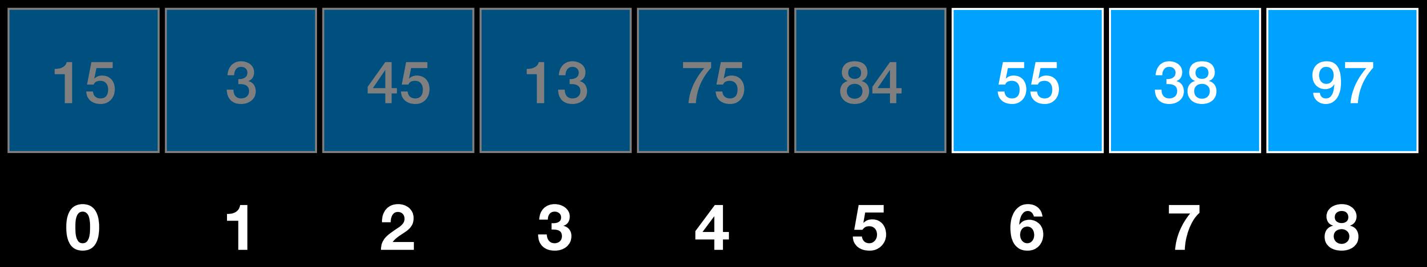
```
front = 3
 back = 5
```

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
```



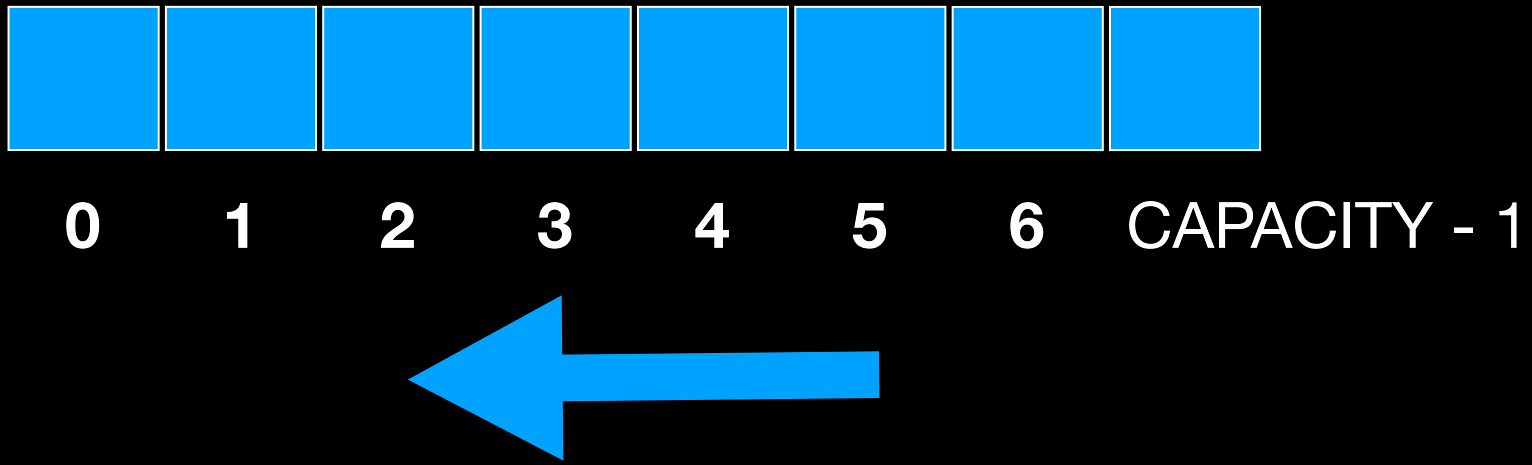
0	1	2	3	4	5	6	7	8
15	3	45	13	75	84	55	38	97

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
```

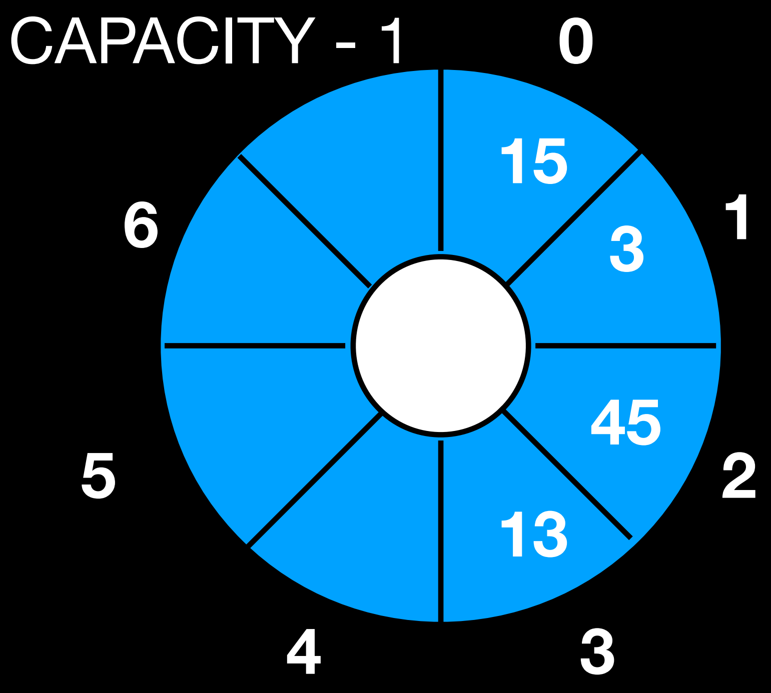


0 1 2 3 4 5 6 CAPACITY - 1

Circular Array Implementation

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

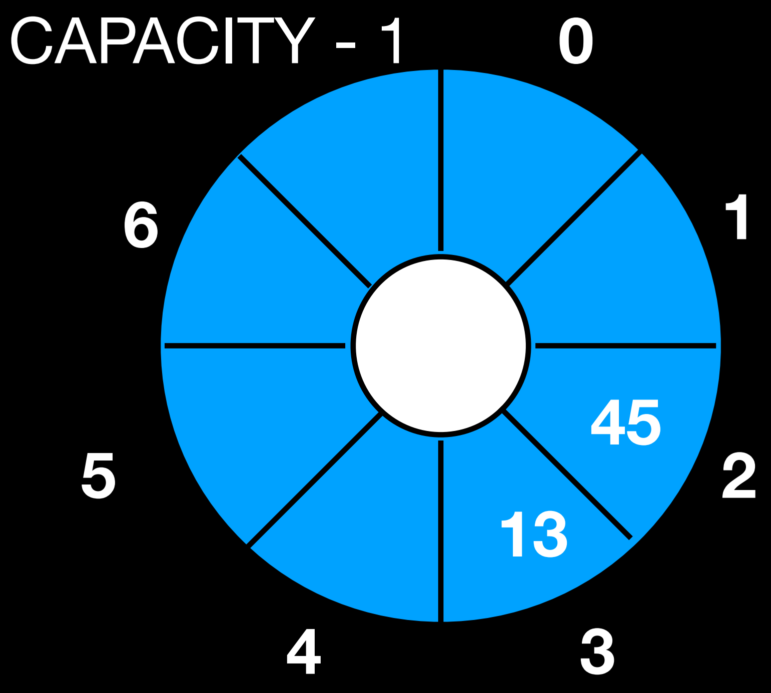
15	3	45	13				
0	1	2	3	4	5	6	CAPACITY - 1



Circular Array Implementation

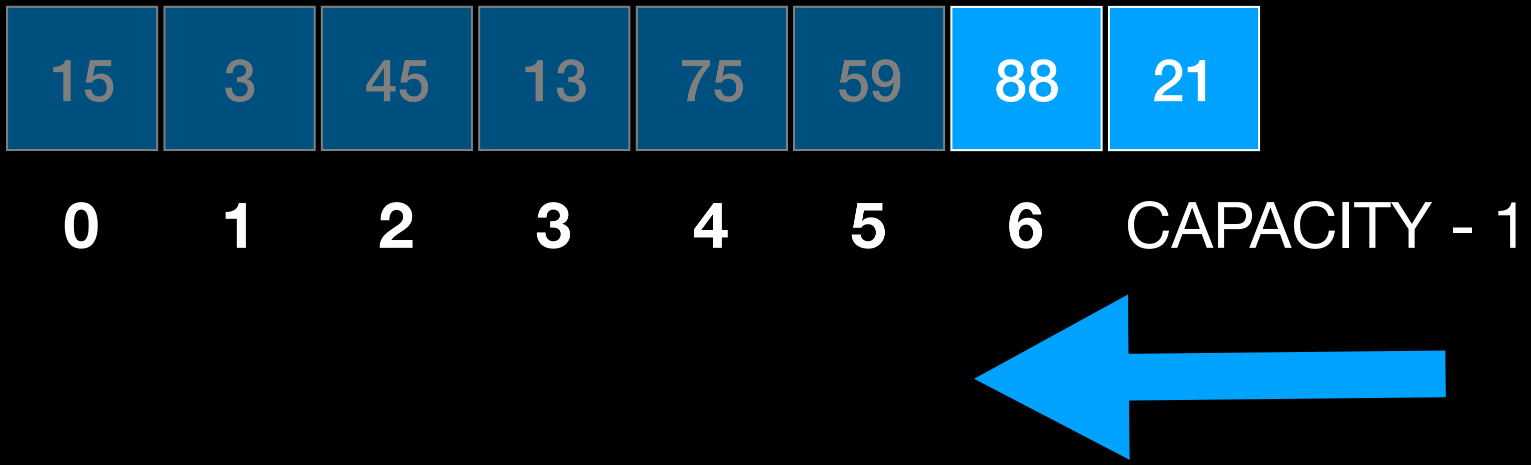
```
front = 2
back = 3
```

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

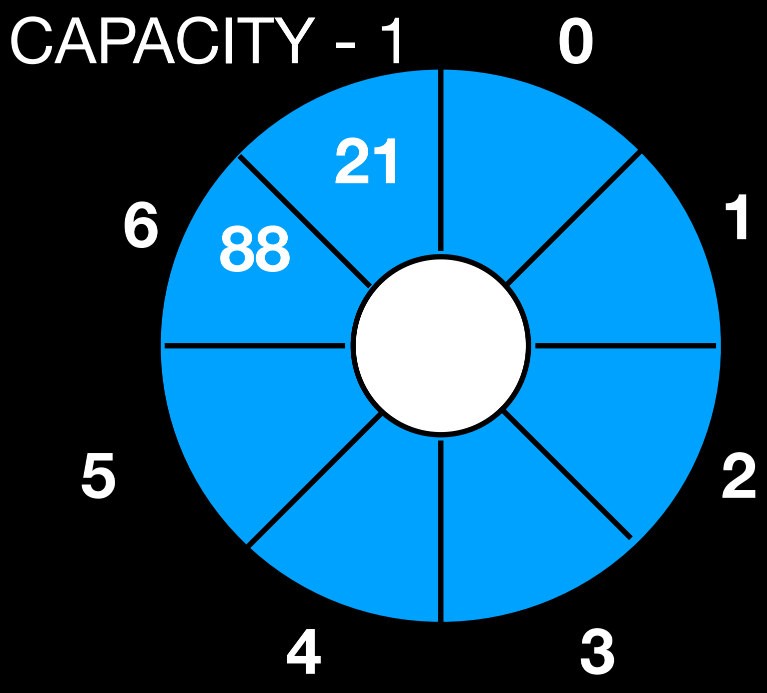


Circular Array Implementation

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



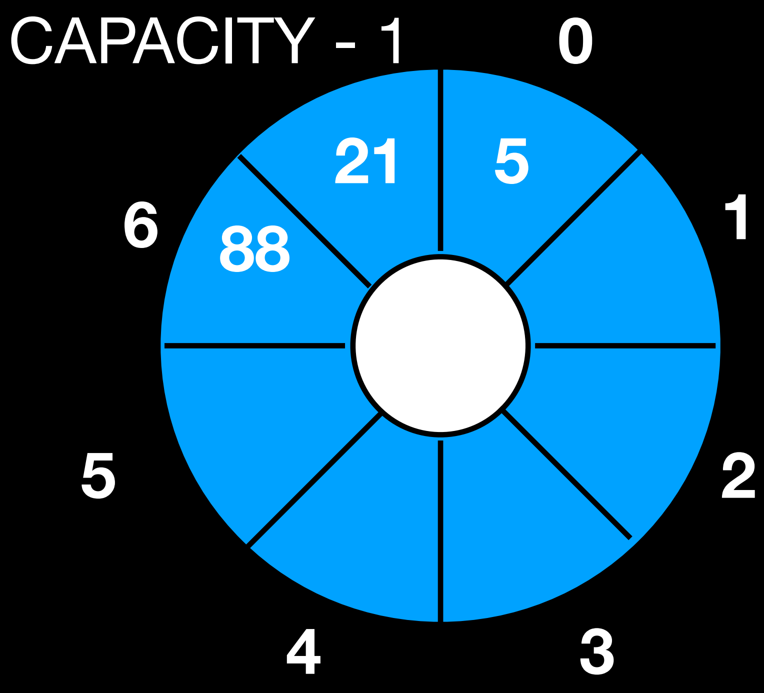
15	3	45	13	75	59	88	21
0	1	2	3	4	5	6	CAPACITY - 1



Circular Array Implementation

```
front = 6
back = 0
```

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



WRAP AROUND USING MODULO ARITHMETIC

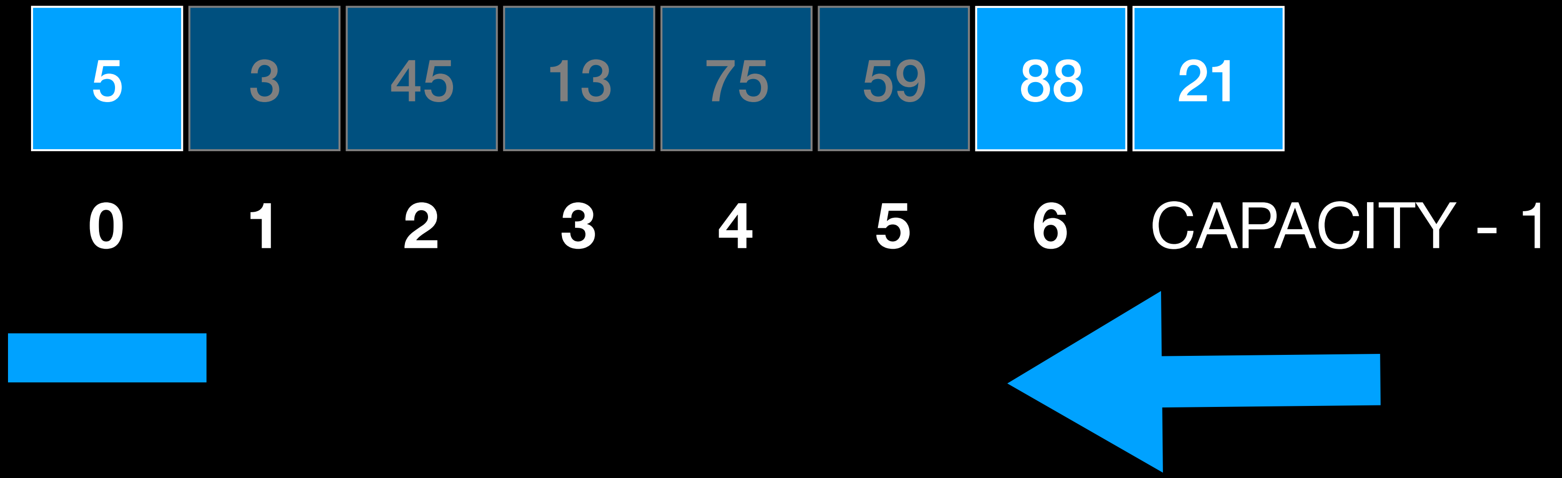
Circular Array Implementation

front = 6

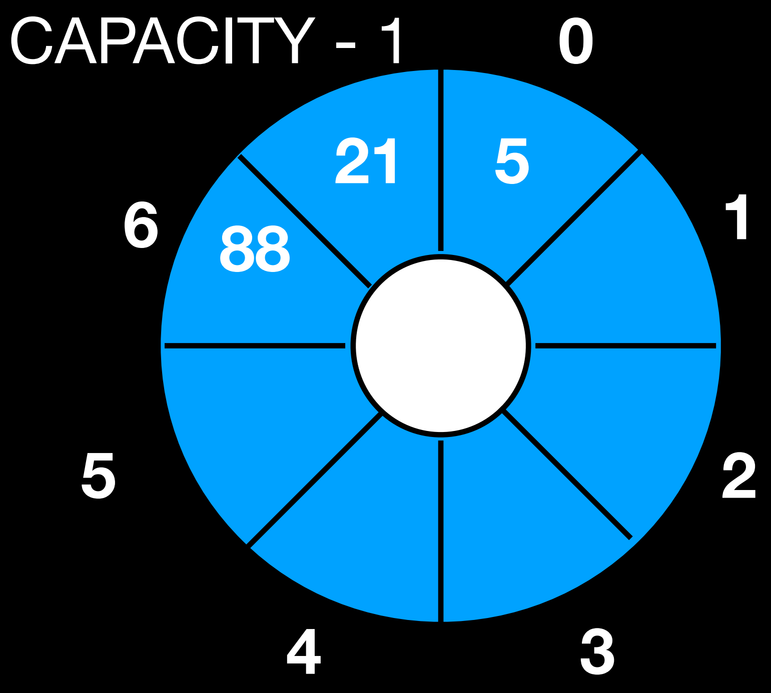
back = 0

enqueue

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

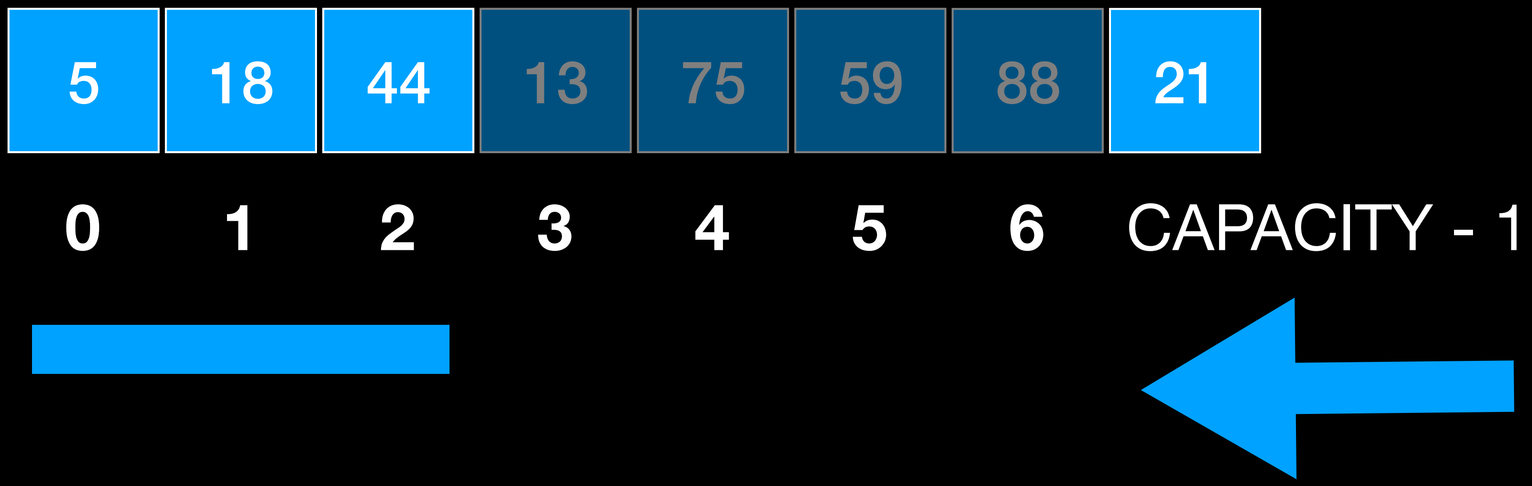


5	3	45	13	75	59	88	21
0	1	2	3	4	5	6	CAPACITY - 1

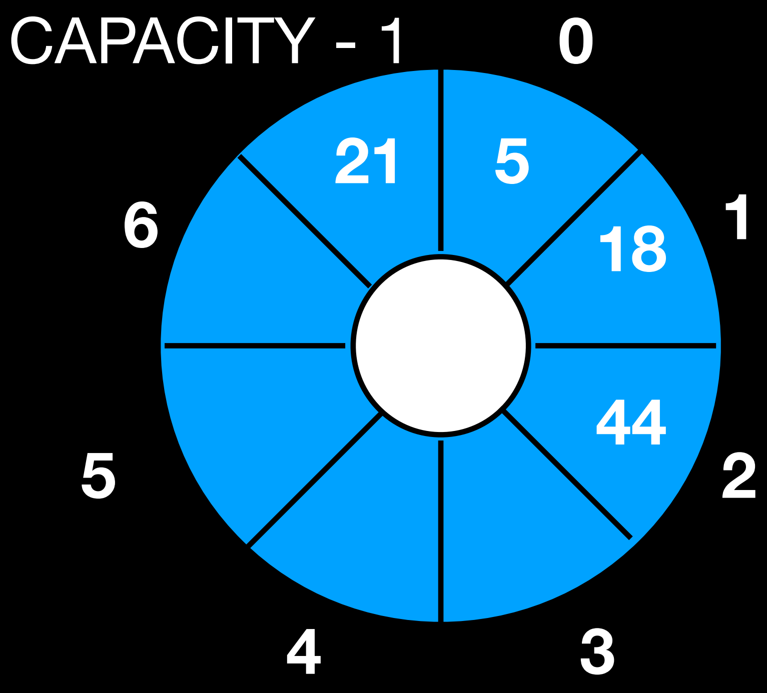


Circular Array Implementation

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



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



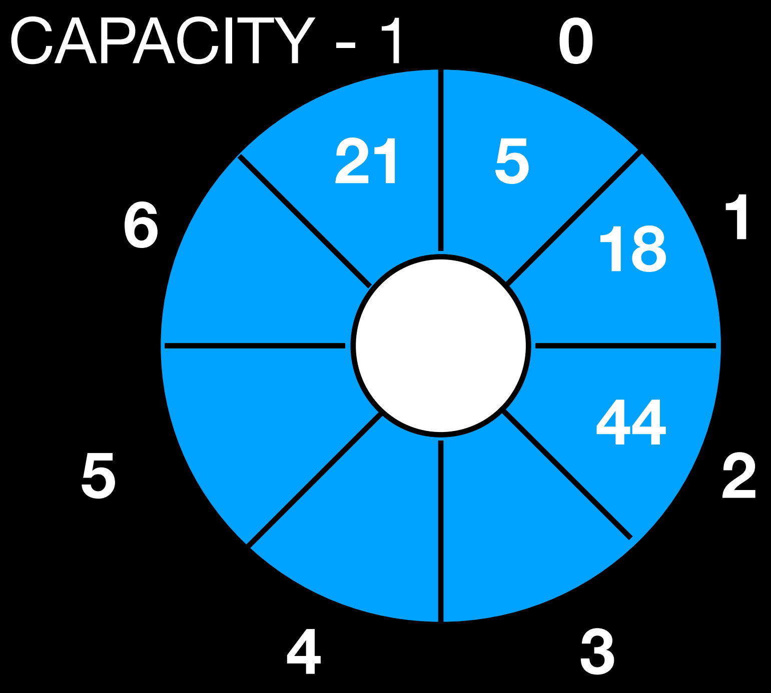
Circular Array Implementation

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

dequeue

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

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



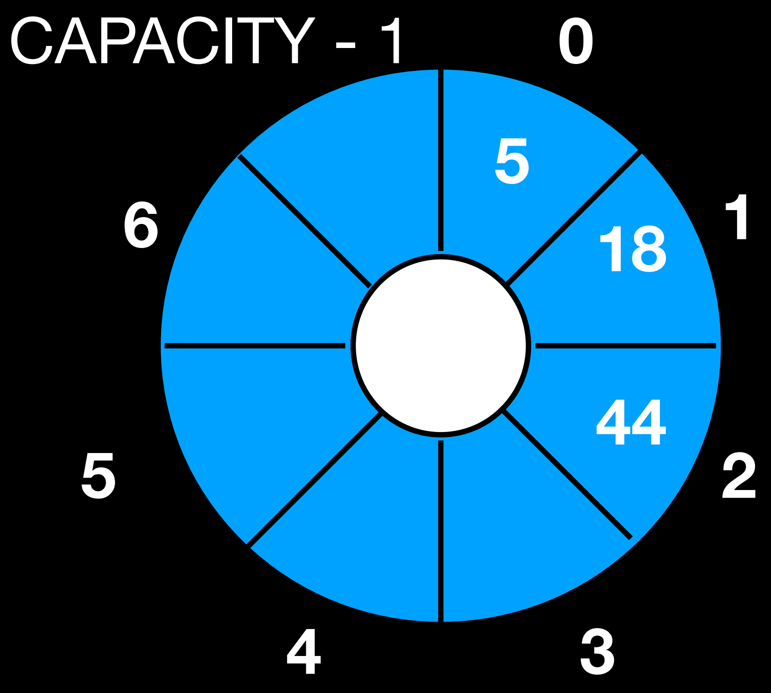
Circular Array Implementation

```
front = 0
back = 2
```

dequeue

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

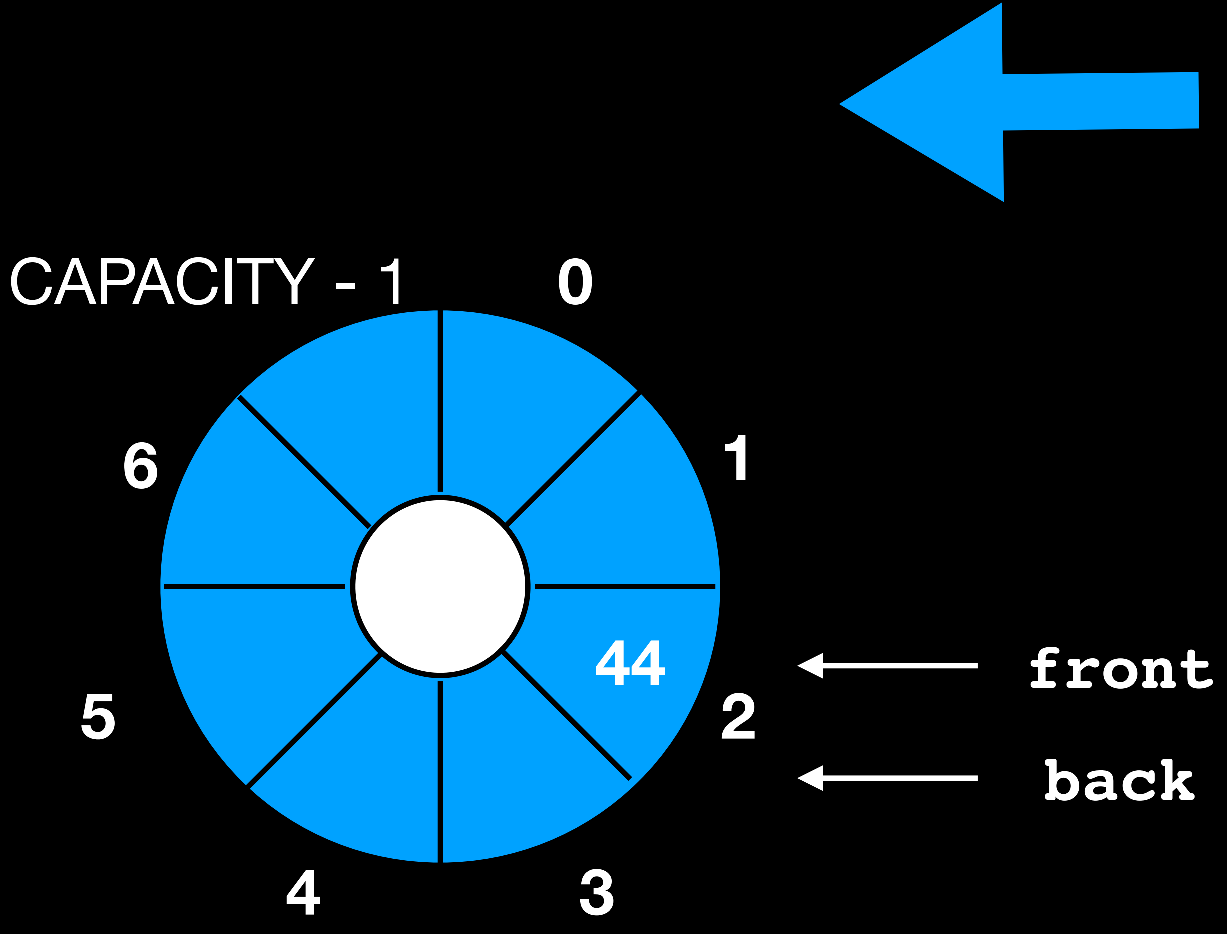
5	18	44	13	75	59	88	21
0	1	2	3	4	5	6	CAPACITY - 1



Circular Array Implementation

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

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



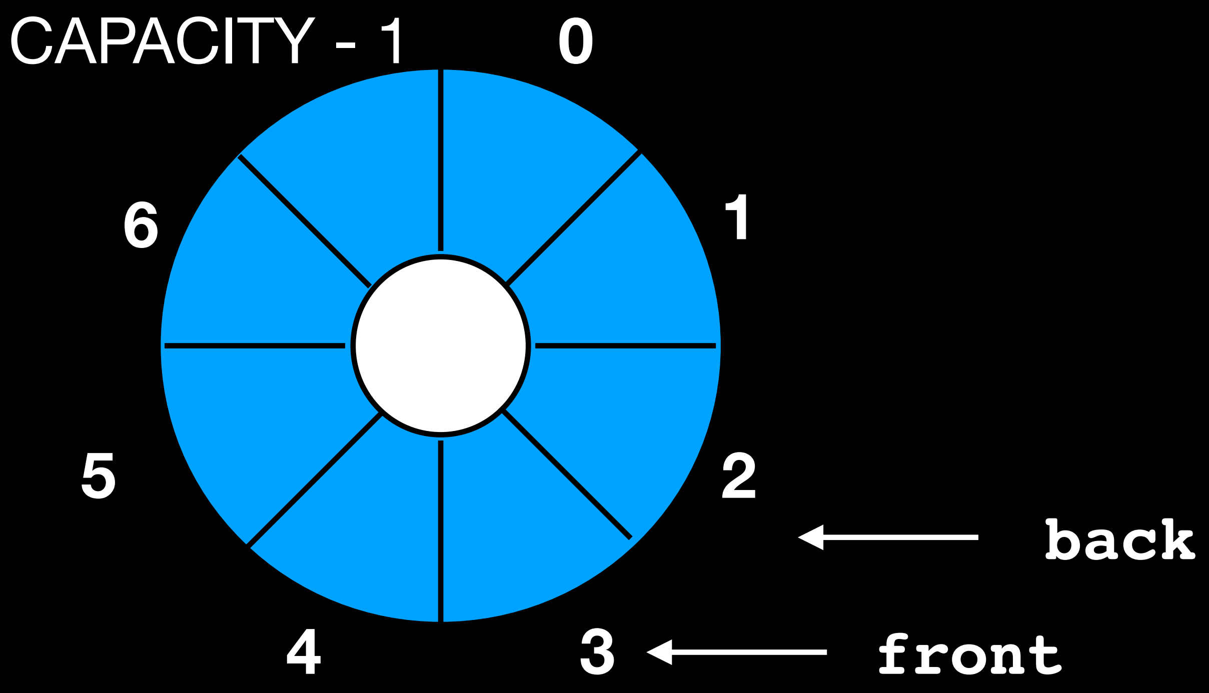
Circular Array Implementation

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

dequeue

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

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



`front` passes `back` when queue is EMPTY

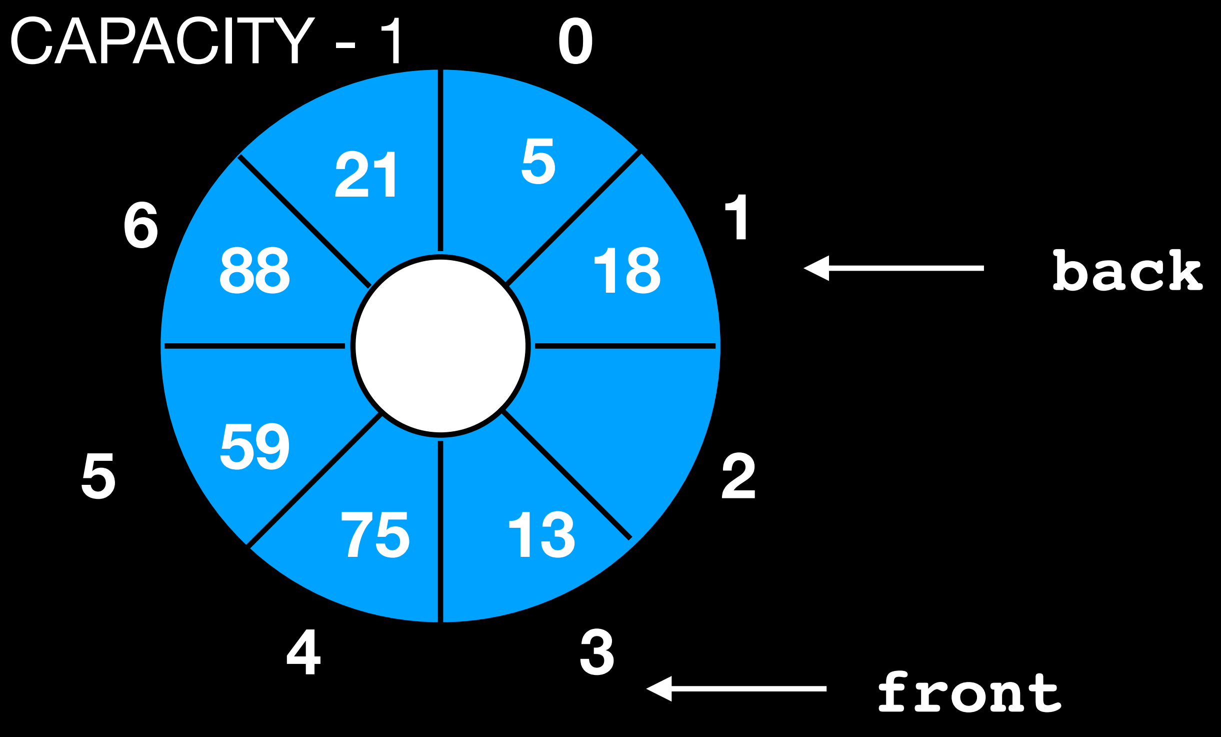
Circular Array Implementation

```
front = 3
back = 1
```

enqueue

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

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



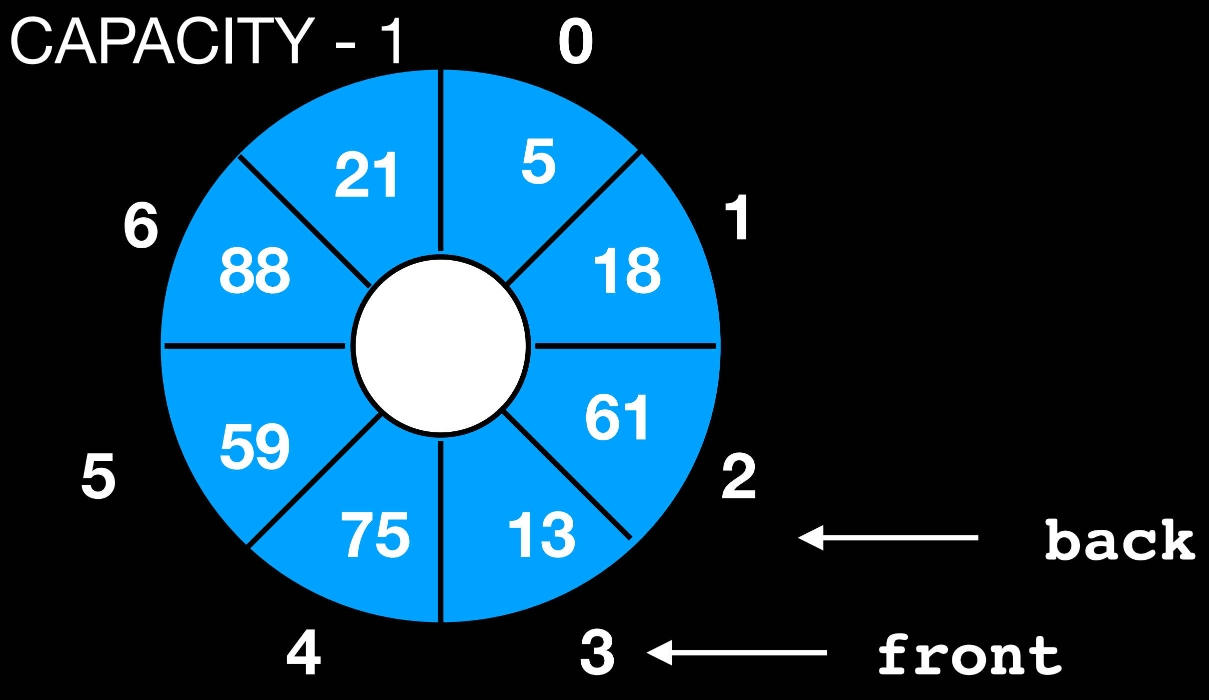
Circular Array Implementation

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

enqueue

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

5	18	61	13	75	59	88	21
0	1	2	3	4	5	6	CAPACITY - 1



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

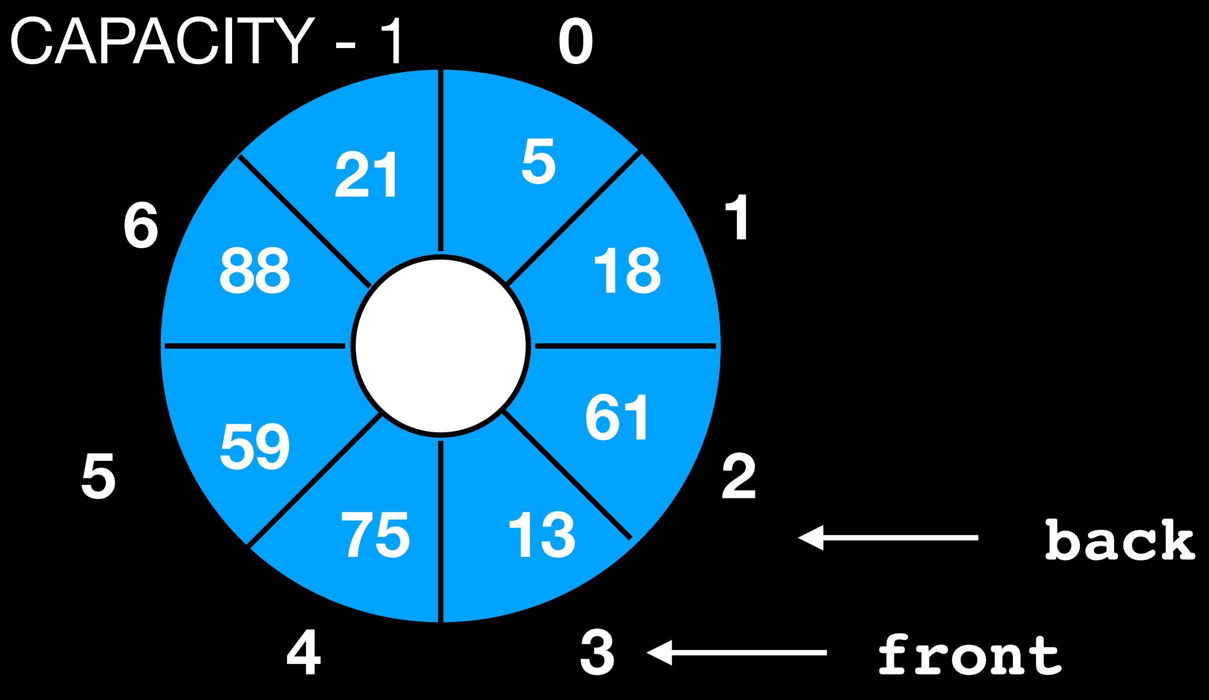
Circular Array Implementation

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

enqueue

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

5	18	61	13	75	59	88	21
0	1	2	3	4	5	6	CAPACITY - 1



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

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