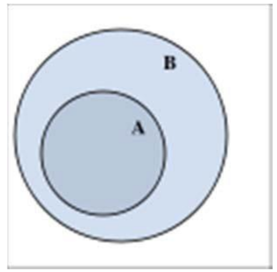
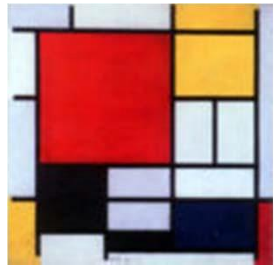




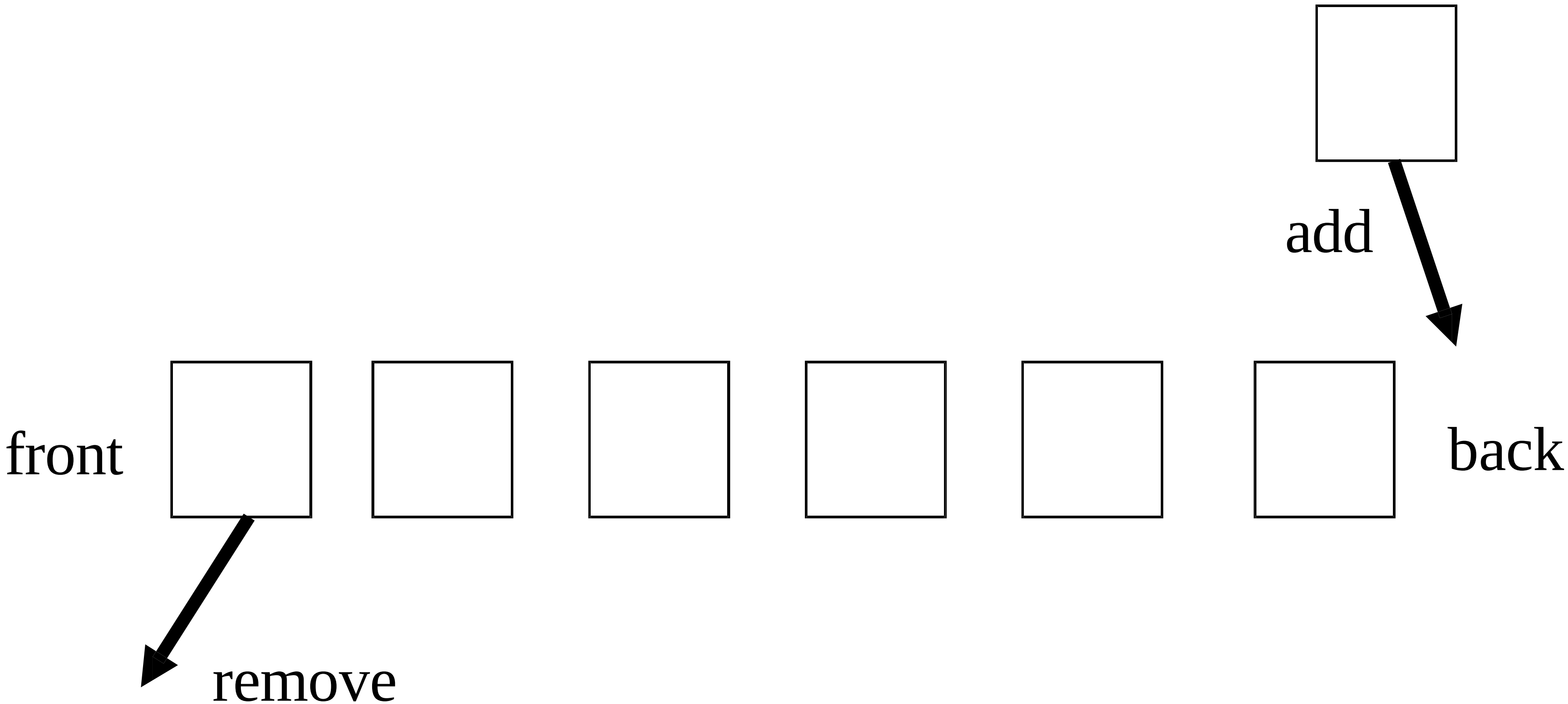
SCC120 Fundamentals of Computer Science
Unit 3: Queues



Jidong Yuan
yuanjd@bjtu.edu.cn



The Queue ADT



first in first out (FIFO)

Applications of the Queue ADT

- Jobs waiting to be executed by a computer operating system
  - at least in simple cases
- Simulations of real-world situations
  - for example, traffic approaching and crossing a road junction controlled by traffic lights

The Key Operations of a Queue ADT

- *add* to the back of the queue
- *remove* from the front of the queue
- *size*
- *isEmpty*

Implementing the Queue ADT

(1) We will first implement the queue with an **array**
- using a mechanism similar to the stack earlier

(2) Then a more efficient array implementation
- using a **circular buffer**

(3) Then an implementation using a **linked list**

(1) Implementing the Queue ADT with an Array

queue

0	1	2	3	4	5
22	44	22	33		

back

3

The add Method

```
if (back == limit - 1)
    PROBLEM - QUEUE FULL
else {
    back++ ;
    queue[back] = X ;
}
// like the stack push earlier
```

The remove Method

```
if (back == -1)
    PROBLEM - QUEUE EMPTY
else {
    Element X = queue[0] ;
    “SHUFFLE THE REST DOWN”
    back-- ;
    return X ;
}
```

The remove Method

```
if (back == -1)
    PROBLEM - QUEUE EMPTY
else {
    Element X = queue[0] ;
    for (int i = 0 ; i < back ; i++)
        queue[i] = queue[i + 1] ;
    back-- ;
    return X ;
}
```

Comments

- Needs to be initialised with *back* set to -1
- Array itself doesn't need to be initialised

Check “add” Works for an Empty Queue

- back = -1
- back != limit – 1; so “if” fails
- back = 0
- queue[0] = X

```
if (back == limit - 1)
    PROBLEM - QUEUE FULL
else {
    back++ ;
    queue[back] = X ;
}
```

Check “add” Works for a Partly-full Queue

- say, back = 3 and limit = 6
- back != limit – 1; so “if” fails
- back = 4
- queue[4] = X

```
if (back == limit - 1)
    PROBLEM - QUEUE FULL
else {
    back++ ;
    queue[back] = X ;
}
```

Check “add” Works for a Nearly Full Queue

- back = 4 and limit = 6
- back != limit – 1; so “if” fails
- back = 5
- queue[5] = X (the last place in the array)

```
if (back == limit - 1)
    PROBLEM - QUEUE FULL
else {
    back++ ;
    queue[back] = X ;
}
```

Check “add” Works for a Full Queue

- back = 5 and limit = 6
- back == limit – 1; so “if” succeeds
- indicate queue is full

```
if (back == limit - 1)
    PROBLEM - QUEUE FULL
else {
    back++ ;
    queue[back] = X ;
}
```

Check “remove” Works for an Empty Queue

- back = -1
- back == -1; so “if” succeeds
- indicate queue is empty

```
if (back == -1)
    PROBLEM - QUEUE EMPTY
else {
    Element X = queue[0] ;
    for (int i = 0 ; i < back ; i++)
        queue[i] = queue[i + 1] ;
    back-- ;
    return X ;
}
```

Check "remove" Works for a Nearly Empty Queue

- back = 0
- back != -1; so "if" fails
- X = queue[0]
- back = 0, i = 0; so no loop
- back = -1 (so queue is empty)

```
if (back == -1)
    PROBLEM - QUEUE EMPTY
else {
    Element X = queue[0] ;
    for (int i = 0 ; i < back ; i++)
        queue[i] = queue[i + 1] ;
    back-- ;
    return X ;
}
```

Check "remove" Works for a Partly-full Queue

- say, back = 3 and limit = 6
- back != -1; so "if" fails
- X = queue[0]
- back = 3, i = 0, 1, 2

  queue[0] = queue[1],
  queue[1] = queue[2],
  queue[2] = queue[3]
- back = 2

```
if (back == -1)
   PROBLEM - QUEUE EMPTY
else {
   Element X = queue[0] ;
   for (int i = 0 ; i < back ; i++)
      queue[i] = queue[i + 1] ;
   back-- ;
   return X ;
}
```

Check “remove” Works for a Full Queue

- back = 5 and limit = 6
- back != -1, so “if” fails
- X = queue[0]
- back = 5, i = 0, 1, 2, 3, 4

  queue[0] = queue[1],
  queue[1] = queue[2],
  queue[2] = queue[3],
  queue[3] = queue[4],
  queue[4] = queue[5]
- back = 4

```
if (back == -1)
    PROBLEM - QUEUE EMPTY
else {
    Element X = queue[0] ;
    for (int i = 0 ; i < back ; i++)
        queue[i] = queue[i + 1] ;
    back-- ;
    return X ;
}
```

Efficiency

- "add" is O(1)
- "remove" is O(N)
  - because of the shifting

```
if (back == limit - 1)
    PROBLEM - QUEUE FULL
else {
    back++ ;
    queue[back] = X ;
}
```

```
if (back == -1)
    PROBLEM - QUEUE EMPTY
else {
    Element X = queue[0] ;
    for (int i = 0 ; i < back ; i++)
        queue[i] = queue[i + 1] ;
    back-- ;
    return X ;
}
```

The size Method

- “size” method just returns “back + 1”
- so this operation is O(1) as well

Variation 1

queue

0	1	2	3	4	5
22	44	22	33		

front

3

Variation 1

- Store with back of queue at zero, front of queue is higher up the array (indicated by variable *front*)
- "remove" is now O(1)
- but "add" is now O(N) because of the shifting up to make room
- so it's no improvement on the original design

Variation 2

- What if we don't do the shifting?

Variation 2
after add(66) and remove()

queue

	44	22	33	66	
0	1	2	3	4	5

front	back
1	4

Variation 2
after add(44) and remove()

queue

		22	33	66	44
0	1	2	3	4	5

front	back
2	5

Variation 2

- We have to do the “shuffle down”
  - as soon as the queue reaches the top of the array
  - or when we want to add a new element and we’ve reached the top of the array
- If the array was nearly full, we would have to do the shuffle down every time we add an element
- Can we avoid the shuffling completely?

Implementing the Queue ADT

(1) We will first implement the queue with an **array**

– using a mechanism similar to the stack earlier

(2) Then a more efficient array implementation

– using a **circular buffer**

(3) Then an implementation using a **linked list**

(2) A Circular Buffer

- Suppose we pretend that the end of the array is joined to the beginning
- We add new elements (at the back of the queue) at
  - queue[4], queue[5] and then queue[0], queue[1] … as long as there is room
- This is called a *circular buffer* or sometimes the *cyclic method*

A Circular Buffer

queue

33	55			66	44
0	1	2	3	4	5

front	back
4	1

The add Method (DRAFT)

```
if (queue full)
    PROBLEM - QUEUE FULL
else {
    back++ ;
    if (back == limit) back = 0 ;
    queue[back] = X ;
}
```

The remove Method (DRAFT)

```
if (queue empty)
    PROBLEM - QUEUE EMPTY
else {
    Element X = queue[front] ;
    front++ ;
    if (front == limit) front = 0 ;
    return X ;
}
```

Detecting Queue Full and Queue Empty

queue

0	1	2	3	4	5
33	55			66	44

front	back	length
4	1	4

The add Method

```
if (length == limit)
    PROBLEM - QUEUE FULL
else {
    back++ ;
    if (back == limit) back = 0 ;
    queue[back] = X ;
    length++ ;
}
```

After add(22)

queue

0	1	2	3	4	5
33	55	22		66	44

front	back	length
4	2	5

The remove Method

```
if (length == 0)
    PROBLEM - QUEUE EMPTY
else {
    Element X = queue[front] ;
    front++ ;
    if (front == limit) front = 0 ;
    length-- ;
    return X ;
}
```

After remove
[returns 66]

queue

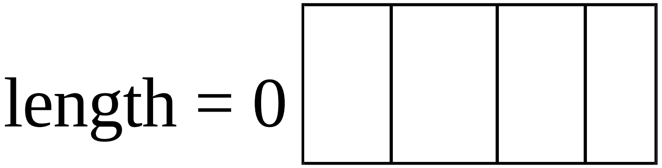
0	1	2	3	4	5
33	55	22			44

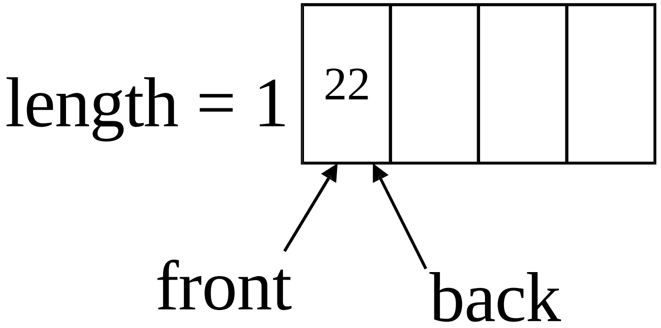
front	back	length
5	2	4

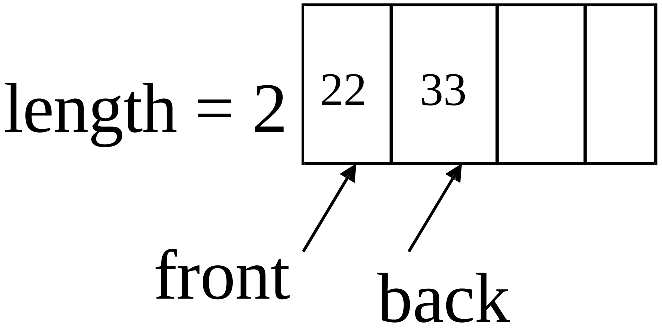
What about "add" into an Empty Queue?

- When we add an element into an empty queue, *front* and *back* both need to point to it

Examples of Short Queues







Check That It Works

- check “add” works for an empty queue
- check “add” works for a partly-full queue
- check “add” works for a nearly full queue
- check “add” works for a full queue

Check That It Works

- check “remove” works for an empty queue
- check “remove” works for a nearly empty queue
- check “remove” works for a partly-full queue
- check “remove” works for a full queue

Efficiency (for Circular Buffer version)

- “add” and “remove” both now O(1)
- because there is no shifting

The size Method

- “size” method just returns the value of *length*
- so this is also O(1)

Implementing the Queue ADT

(1) We will first implement the queue with an **array**

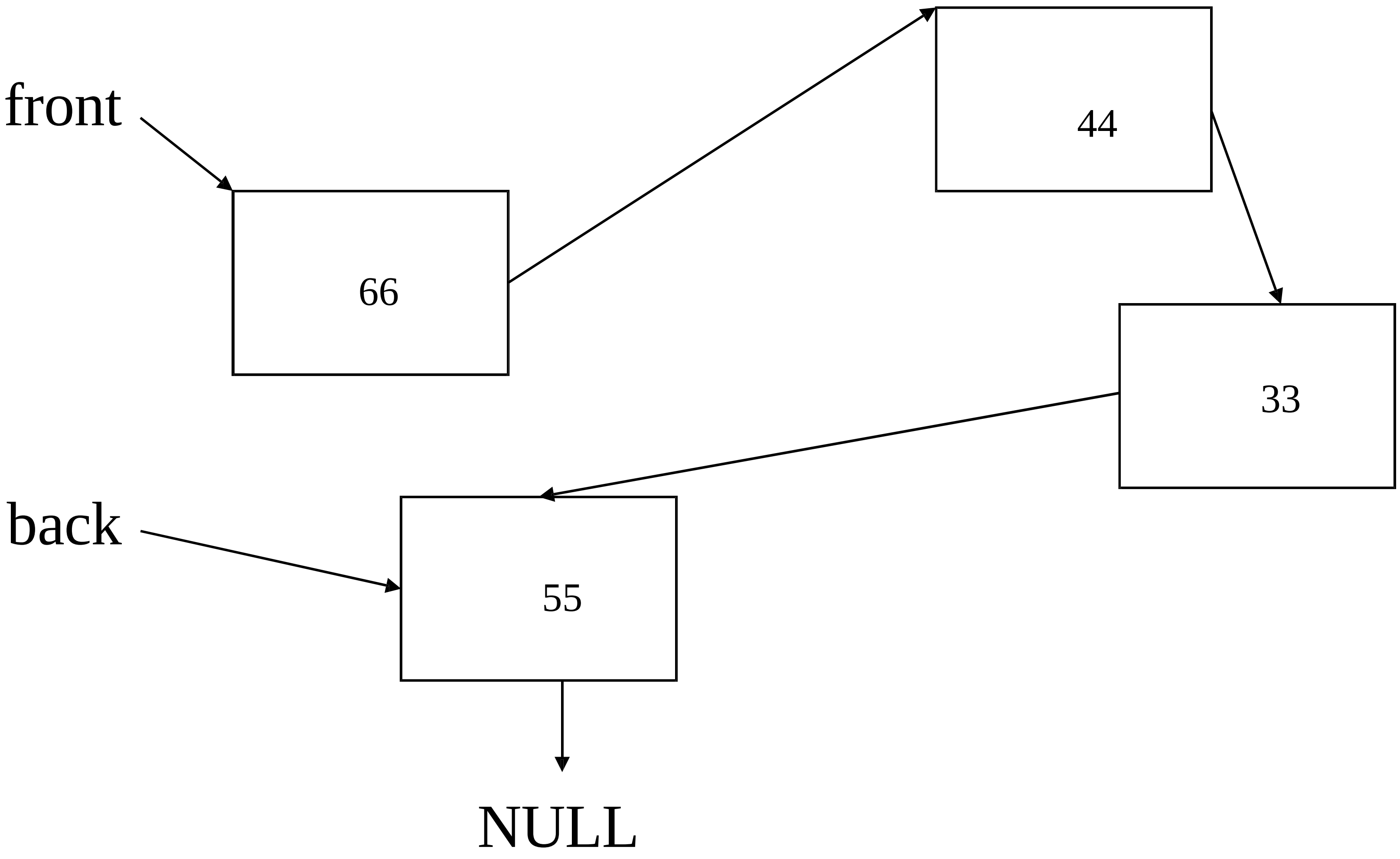
- using a mechanism similar to the stack earlier

(2) Then a more efficient array implementation

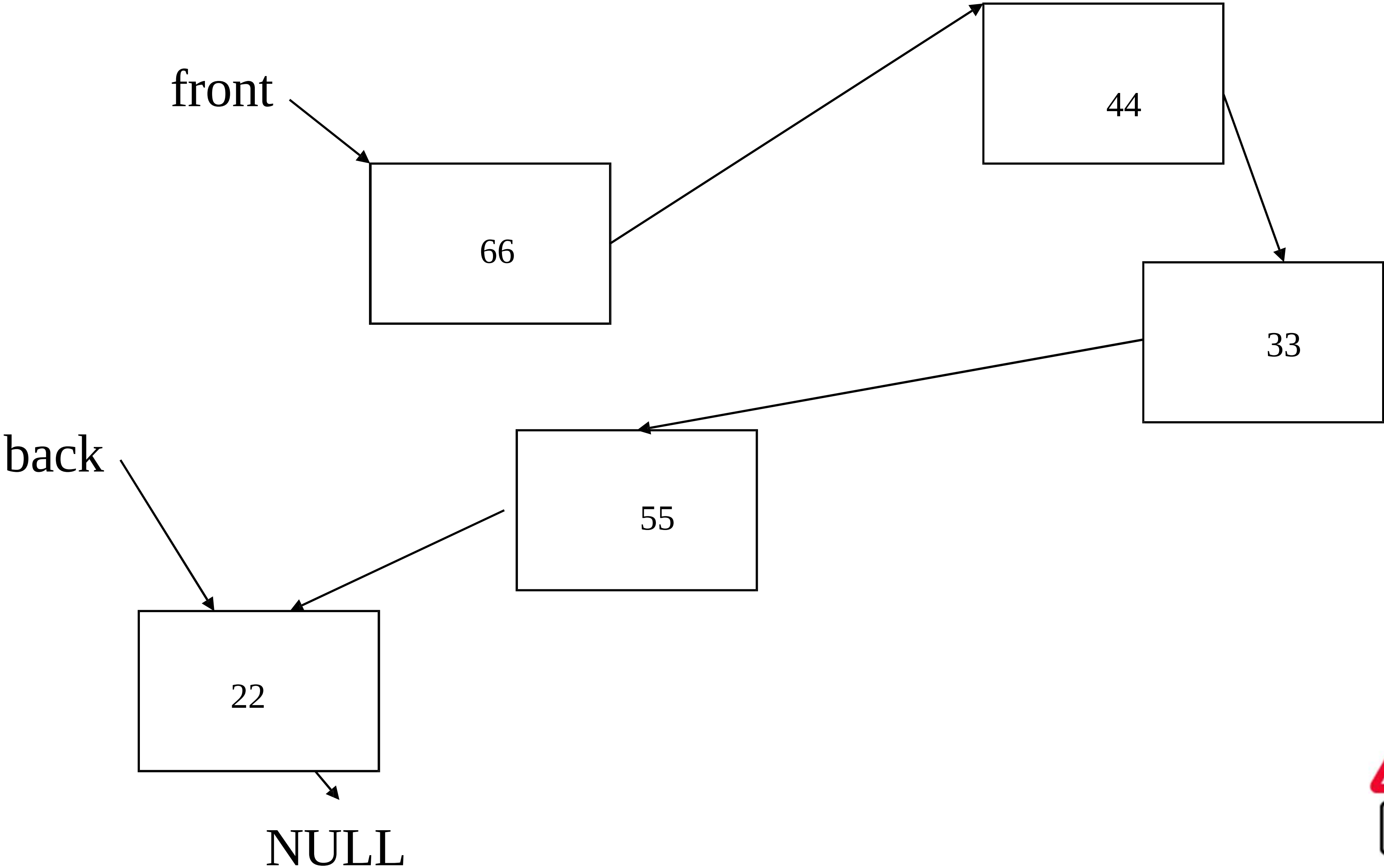
- using a **circular buffer**

(3) Then an implementation using a **linked list**

(3) Implementing the Queue ADT with a Linked List



After add(22)

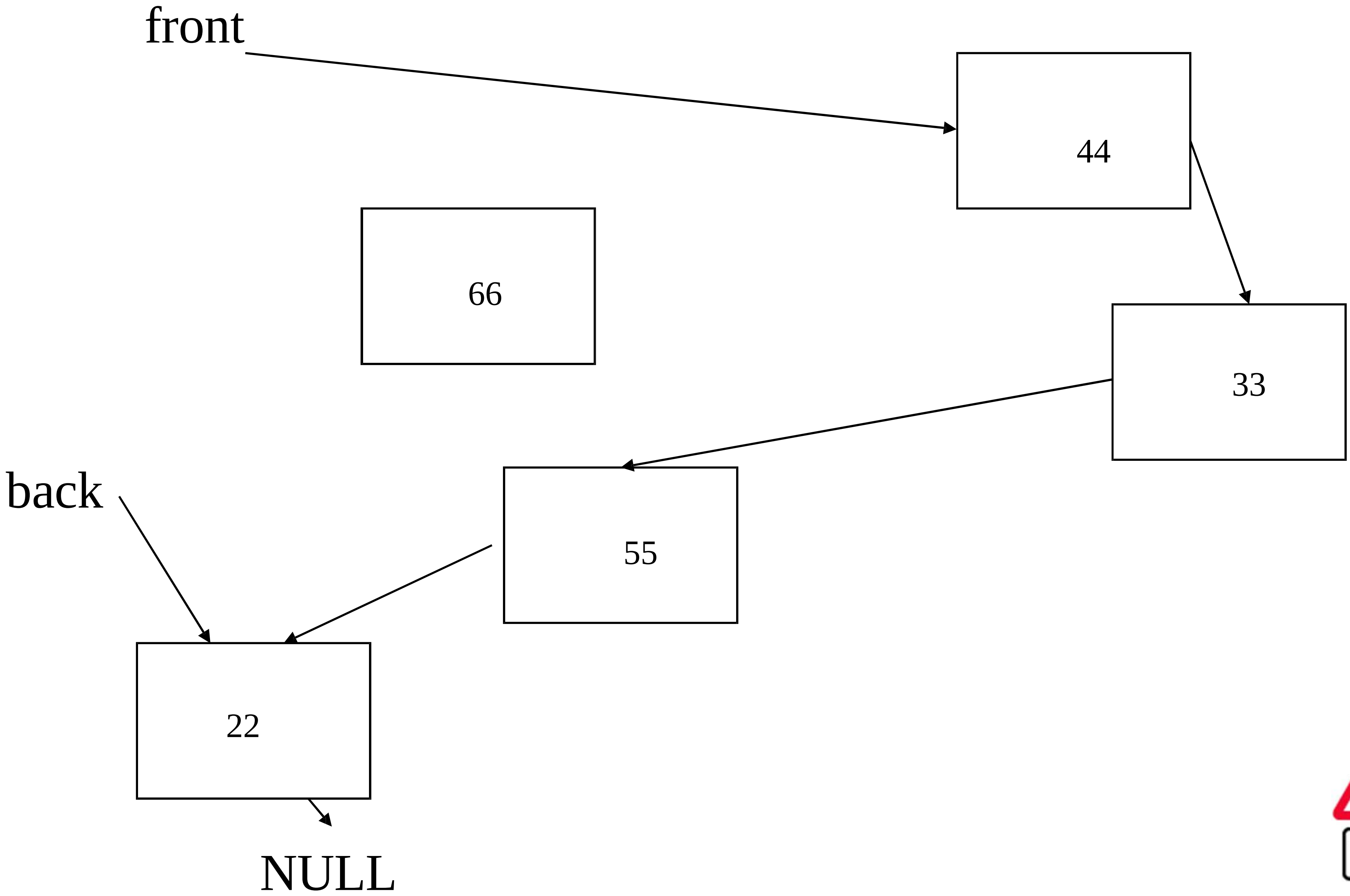


The add Method

```
QueueCell temp = new QueueCell(X, null) ;
back.next = temp ;
back = temp ;
```

- unlike the array implementation, there is no size restriction
- like the stack “push” method earlier

After remove
[returns 66]



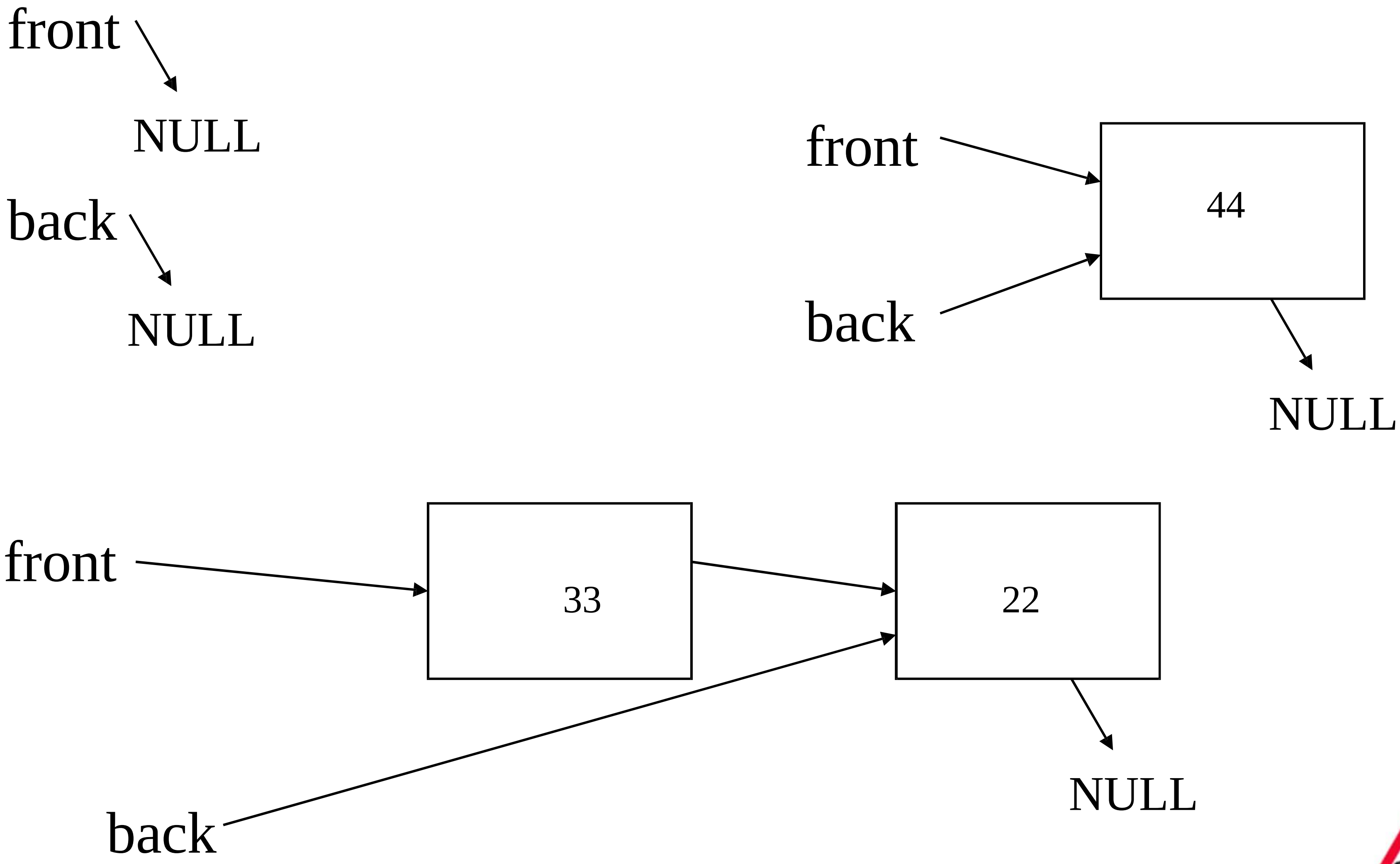
The remove Method

```
if (front == null)
    PROBLEM - QUEUE EMPTY
else {
    Element X = front.data ;
    front = front.next ;
    return X ;
}
```

What about “add” into an Empty Queue?

- When we insert an element into an empty queue, *back* will point to it
- and in this case we need to set *front* pointing to it as well

Examples of Short Queues



Check That It Works

- check “add” works for an empty queue
- check “add” works for a queue with one element
- check “add” works for a queue with some elements

Check That It Works

- check “remove” works for an empty queue
- check “remove” works for a queue with one element
- check “remove” works for a queue with some elements

Efficiency (for Linked List version)

- “add” and “remove” both O(1)

The size Method

- We would have to scan the linked list and count the elements, which would be O(N)
- Instead we could have another variable *length*
  - which is initialised to zero
  - incremented by “add”, decremented by “remove”
  - and then “size” method is also O(1)

Implementing the Queue ADT

(1) We will first implement the queue with an **array**

– using a mechanism similar to the stack earlier

(2) Then a more efficient array implementation

– using a **circular buffer**

(3) Then an implementation using a **linked list**

Next:

A Queue Class

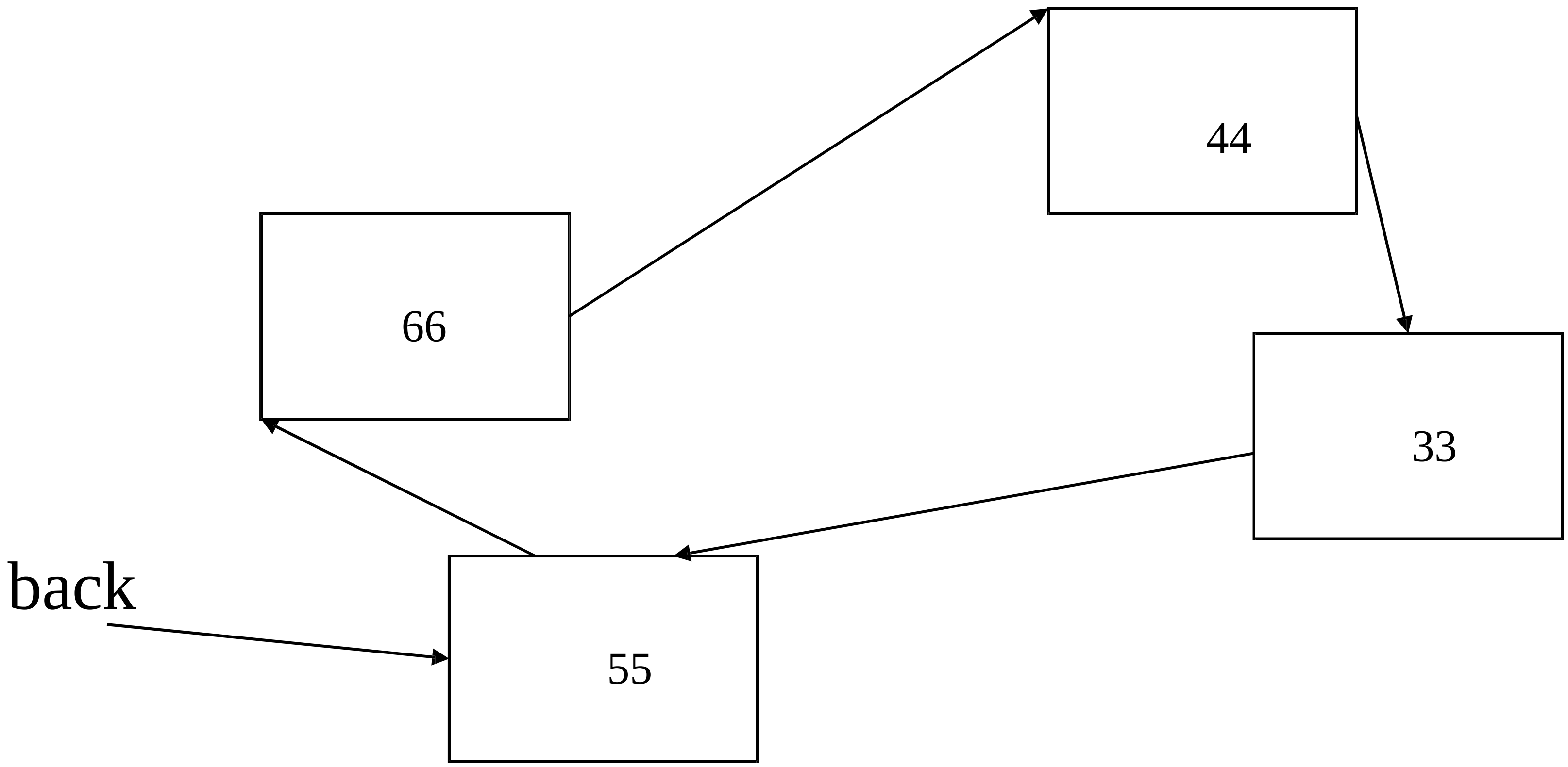
Additional Types of Queues

- A Circular Linked List
- A Double-Ended Queue (or a Two-Way Linked List)
- Priority Queues

A Queue Class

```
public class Queue
{
    public Queue() ;
    public void add(Element X) ;
    public Element remove() ;
    public boolean isEmpty() ;
    public int size() ;
    public Element peek() ;
}
```

A Circular Linked List



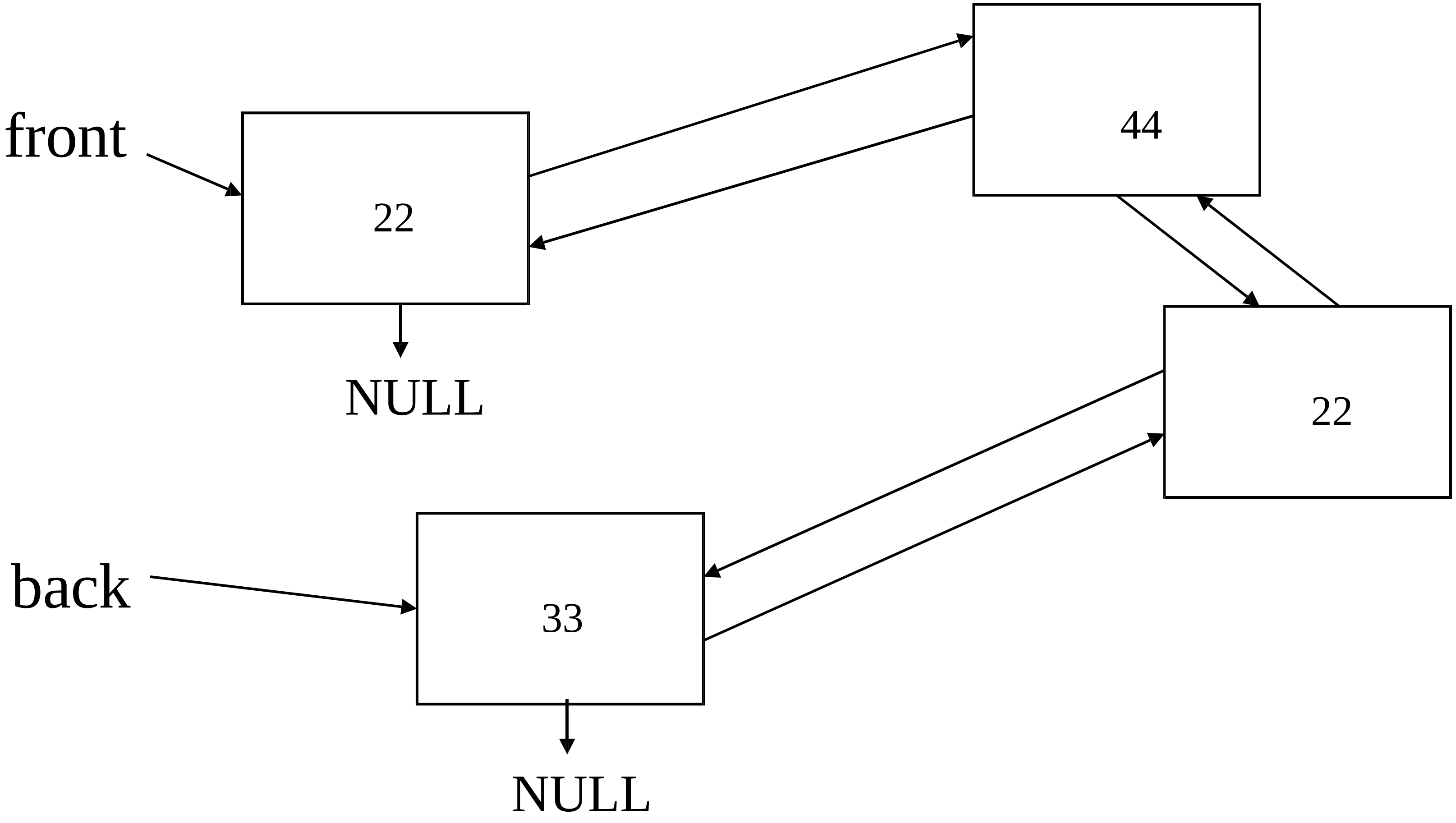
A Double-Ended Queue

- Sometimes called a “deque” (pronounced DQ)
- You can add and remove at each end (but not in the middle)
- Can be implemented with a linked list

A Double-Ended Queue

- An alternative implementation is a *two-way* linked list
- Each element has pointers to the next and previous elements

A Two-Way Linked List



Priority Queues

- A priority queue is a dynamic ADT in which every item added has an associated priority value
- When a remove is done, the item taken is always that with the highest priority
- If two or more items have the same highest priority, they should normally be removed in the normal queue order (that is, first in first out)

Applications of Priority Queues

- Jobs waiting to be executed by a computer operating system
  - in more complicated cases
- Simulations of real-world situations
  - for example an A & E department, where the nurse assigns a priority to each patient
- A way of sorting a set of objects
  - adding a set of items to a priority queue, and then removing them, sorts the elements into priority order

The Key Operations of a Priority Queue

- *add* and *remove* are the key operations
- *member*, *size* and *isEmpty* are also often useful
- another important operation is *promote*

The Promote Operation

- If the queue is “busy”, low priority items will rarely get processed
  - they can remain stuck in the queue for a long time
- To avoid this, items can be promoted after a period of time in the queue
  - by increasing their priority levels
  - for example, this operation can be carried out on low-priority items at regular intervals

Priority Queues: “add” and “remove”

- We do the work when we *add* the element
  - scan to the correct position, and insert it there
- Alternatively (but less common) we could *add* the element at the end, and then scan to find the highest priority element when we want to *remove* it
  - Here we would do the work in the “remove” method

The “add” Method

- Scan to find the correct position
  - from the beginning of the queue
  - past all elements with higher or the same priority
  - then insert it
- The scan means the operation is O(N); the array version requires shuffling up the elements with lower priority than the one being inserted

The “remove” Method

- Just remove the first item (no scanning required)
- But in the array implementation, we need to shuffle down the remaining elements in the queue
- So it’s O(1) for the linked list implementation, and O(N) for the array implementation

“Promotion” Method

- One option would be to recompute all the priorities
  - increasing each priority by an amount proportional to the time the item has been waiting
  - so we need to timestamp the elements when we add them to the queue
- Then we reorder the queue using the new priority values
- So we require a linear scan to recompute the priorities, then a sort (which is generally worse than linear)

“Promotion” Method: via “Deletion”

- Another way of handling promotion is to pick out just one element to promote (perhaps the oldest or one element near the back):
  - Delete it from the queue (possibly from somewhere in the middle)
  - Recompute the priority
  - Add it back into the queue

SCC120 ADT (weeks 9-13)

• Week 7	Abstractions; Set
	Stack
• Week 8	Queues (add and remove operations, various types of implementations)
• Week 9	
• Week 10	
• Week 11	
• Week 12	