YEAH 5: Patient Queue!

Avery Wang

YEAH Hours Agenda

- Pointers Crash Course
- Intro to Priority Queues
- Overview of the Assignment
- How to Get Started
- Tips for Parts I, II, and III
- Questions

Pointers Crash Course

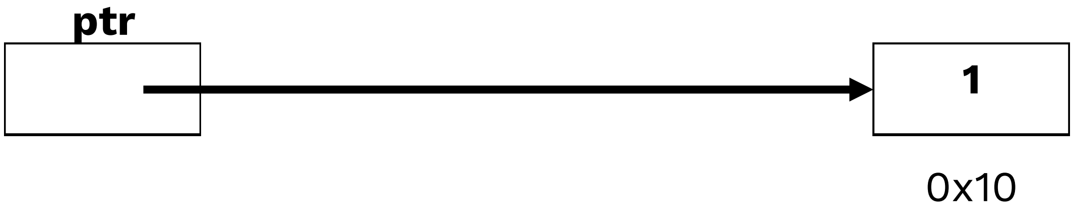
ptr

0x10

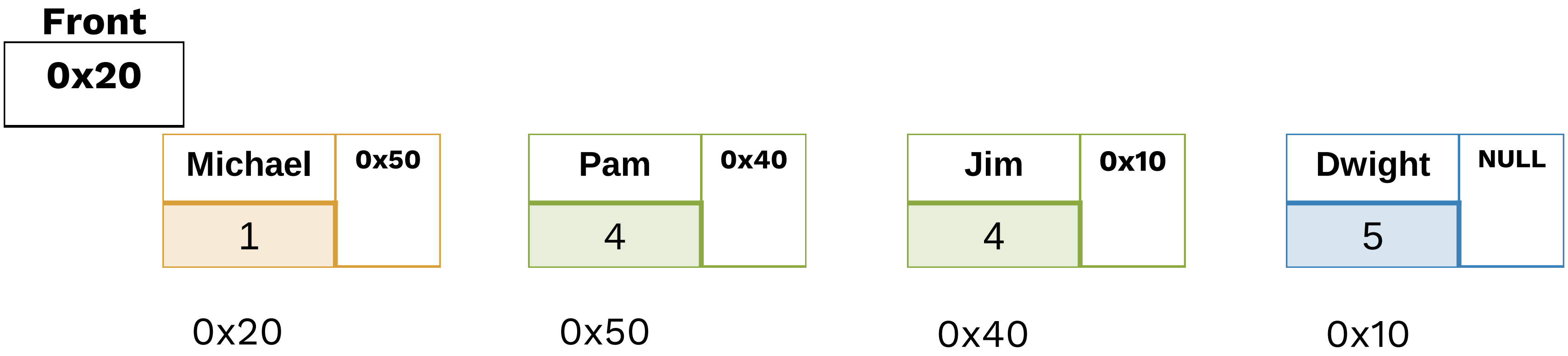
1

0x10

Pointers Crash Course



Linked List



Front

0x20

Michael	**0x50**
1	

0x20

Pam	**0x40**
4	

0x50

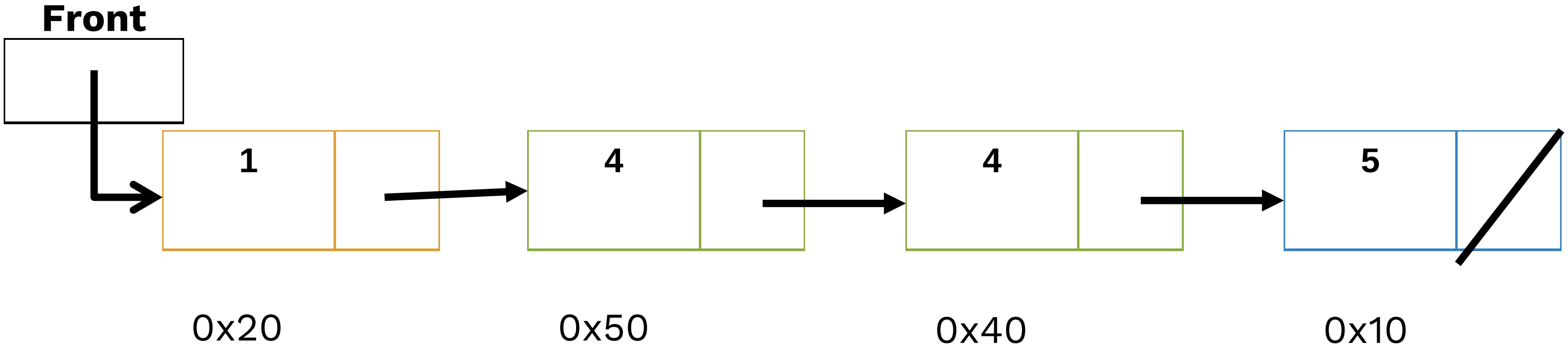
Jim	**0x10**
4	

0x40

Dwight	**NULL**
5	

0x10

Linked List



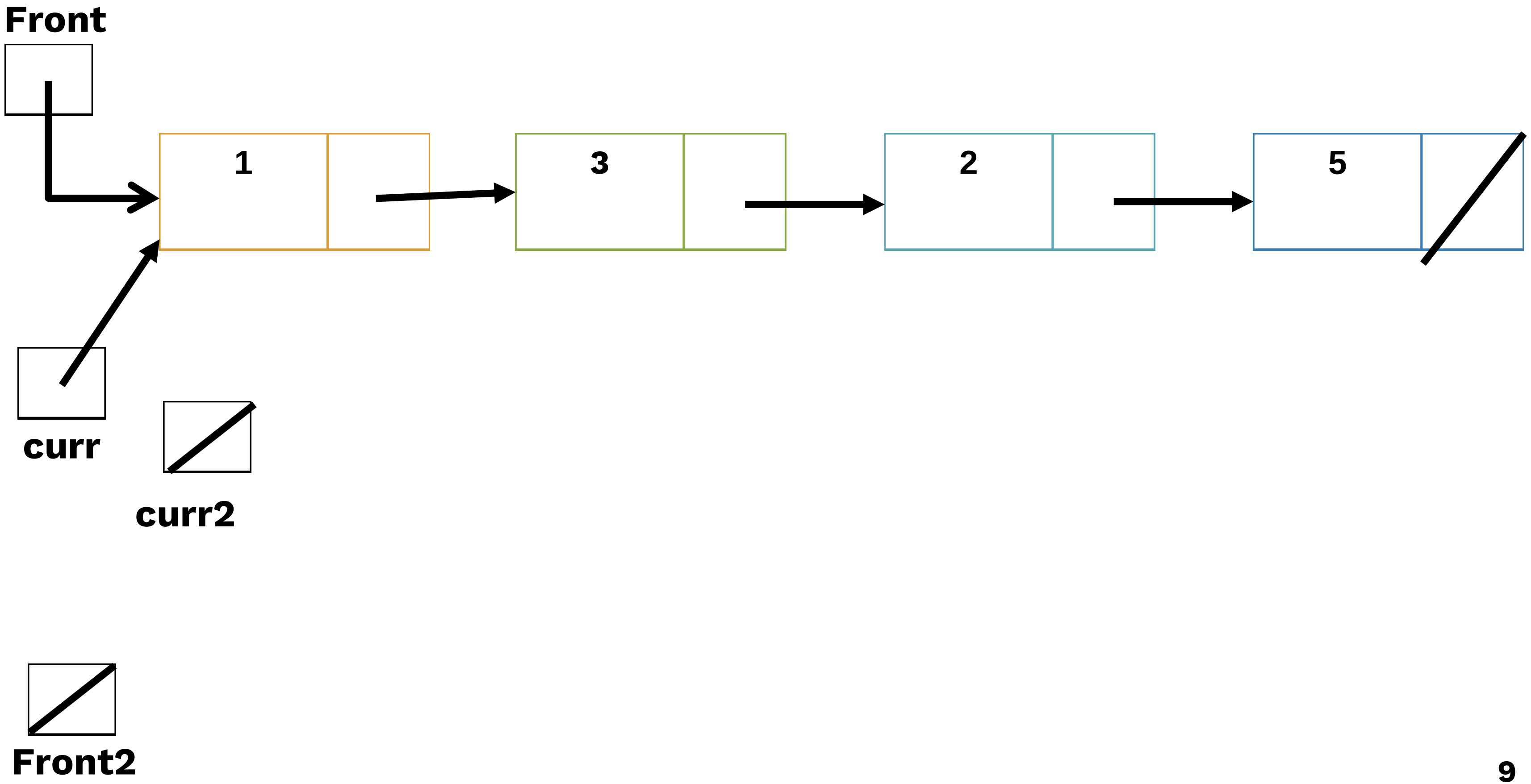
Manipulating Linked List

- Loop through list using a pointer variable (“curr”).
- Check when you’re at or close to the end of the list.
- Manipulate the list by changing “next” fields.

Double List (Section Problem)

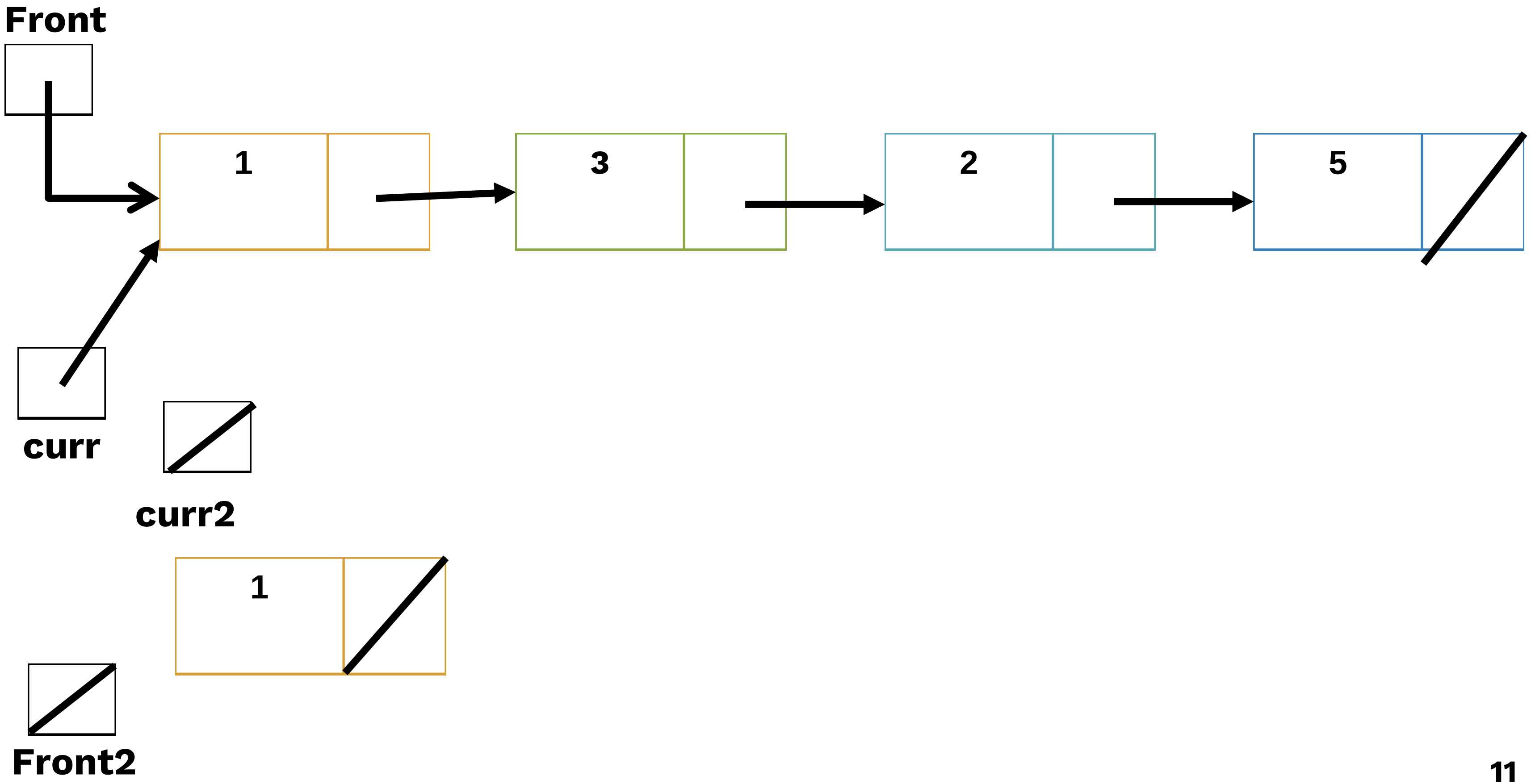
- Write a function that takes a pointer to the front of a linked list of integers and appends a copy of the original sequence to the end of the list.

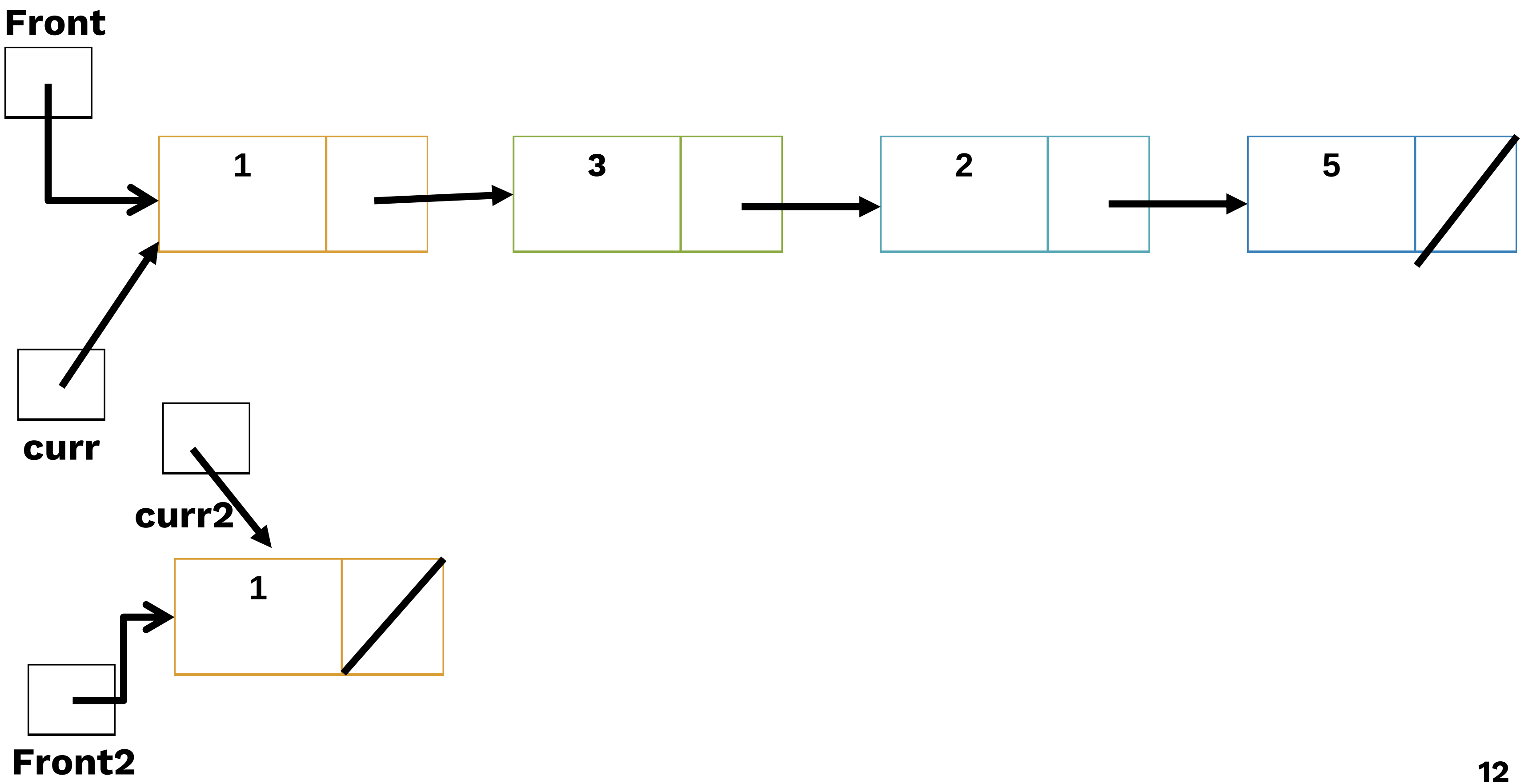
{1, 3, 2, 7} -> {1, 3, 2, 7, 1, 3, 2, 7}


Front
1
3
2
5
curr
curr2
Front2

```
void doubleList(ListNode *front){
        if (front == nullptr) return;
        ListNode *curr = front;








}
```

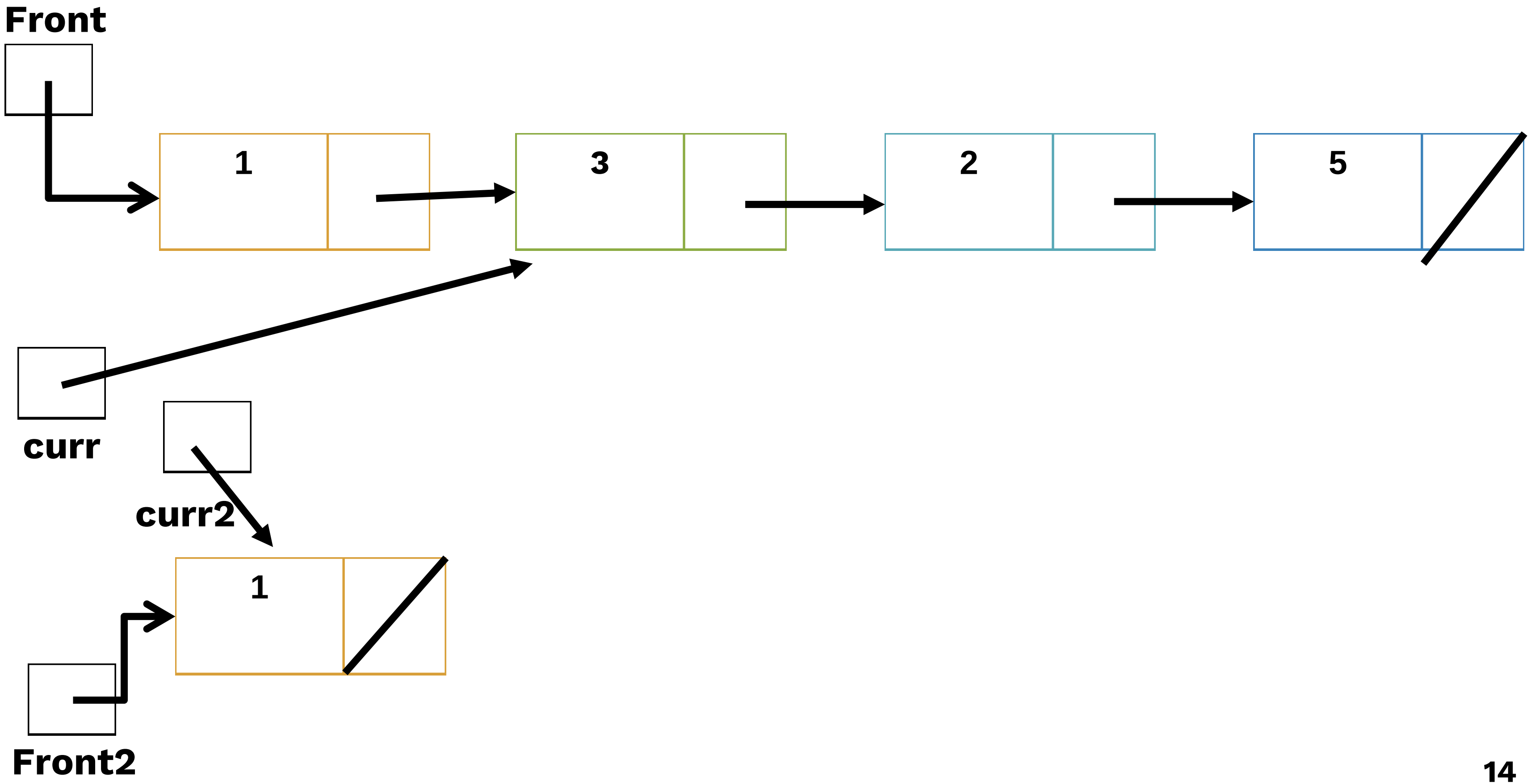
Check if list is empty!
Set up “curr1”

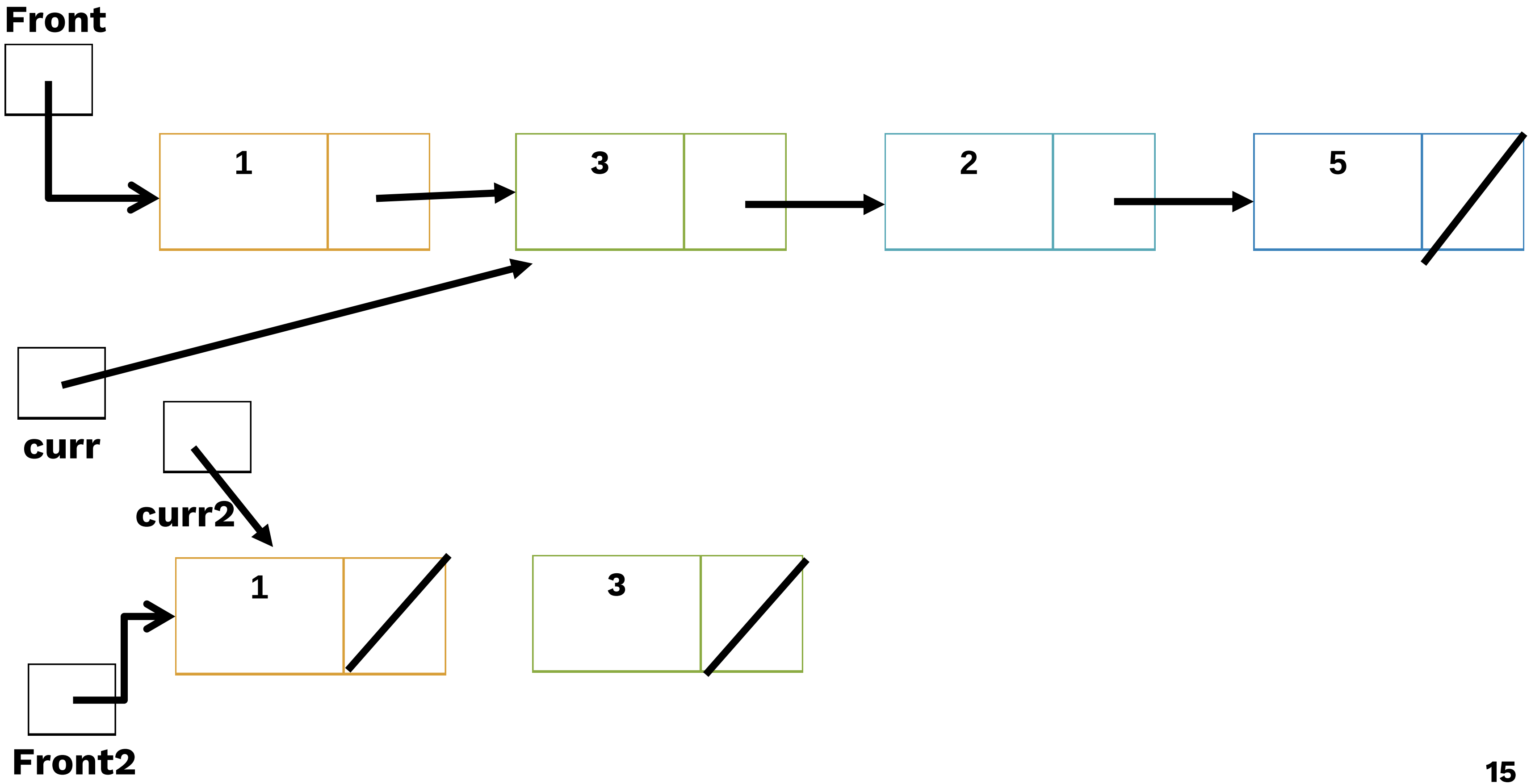

Front
1
3
2
5
curr
curr2
1
Front2

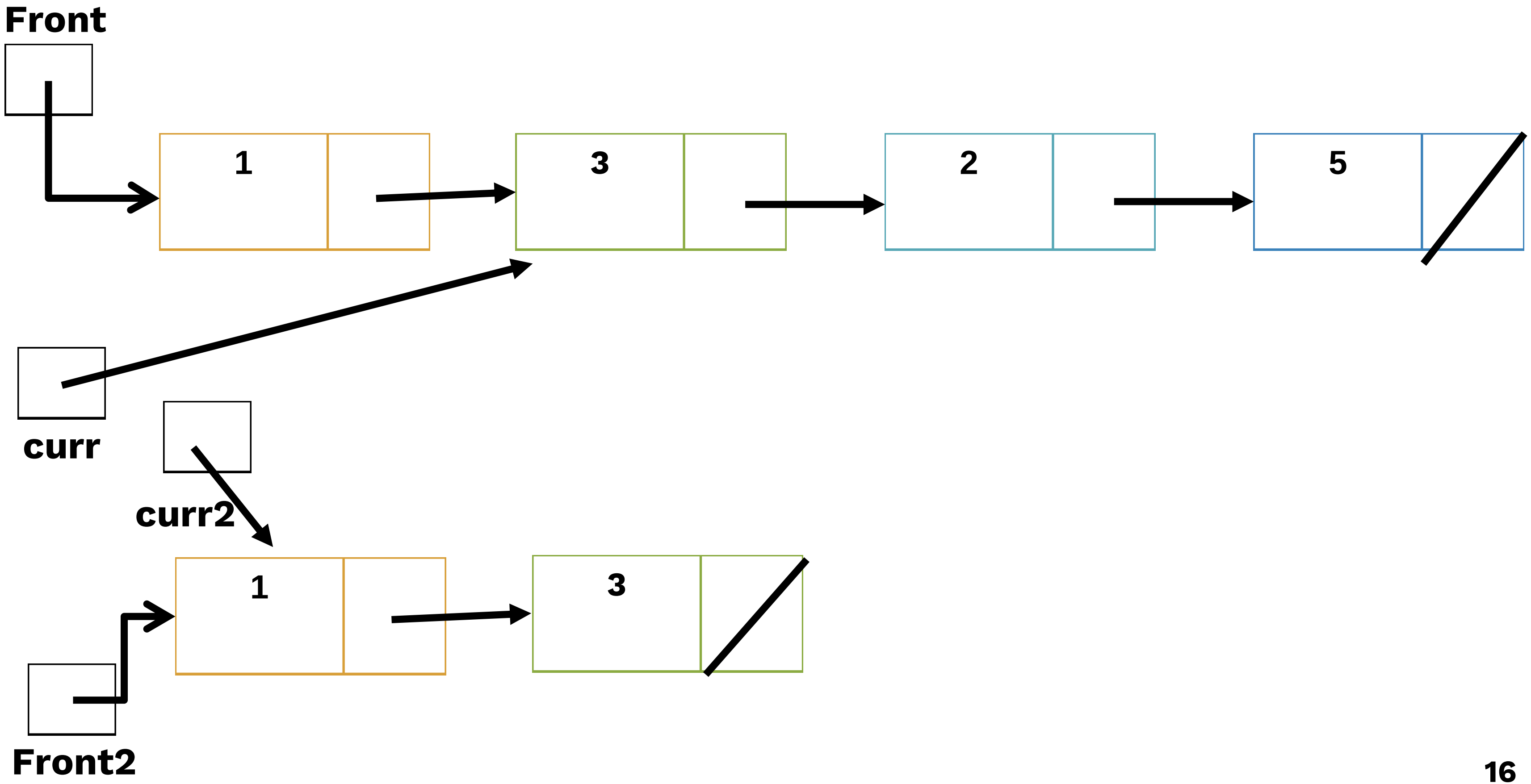

Front
1
3
2
5
curr
curr2
1
Front2

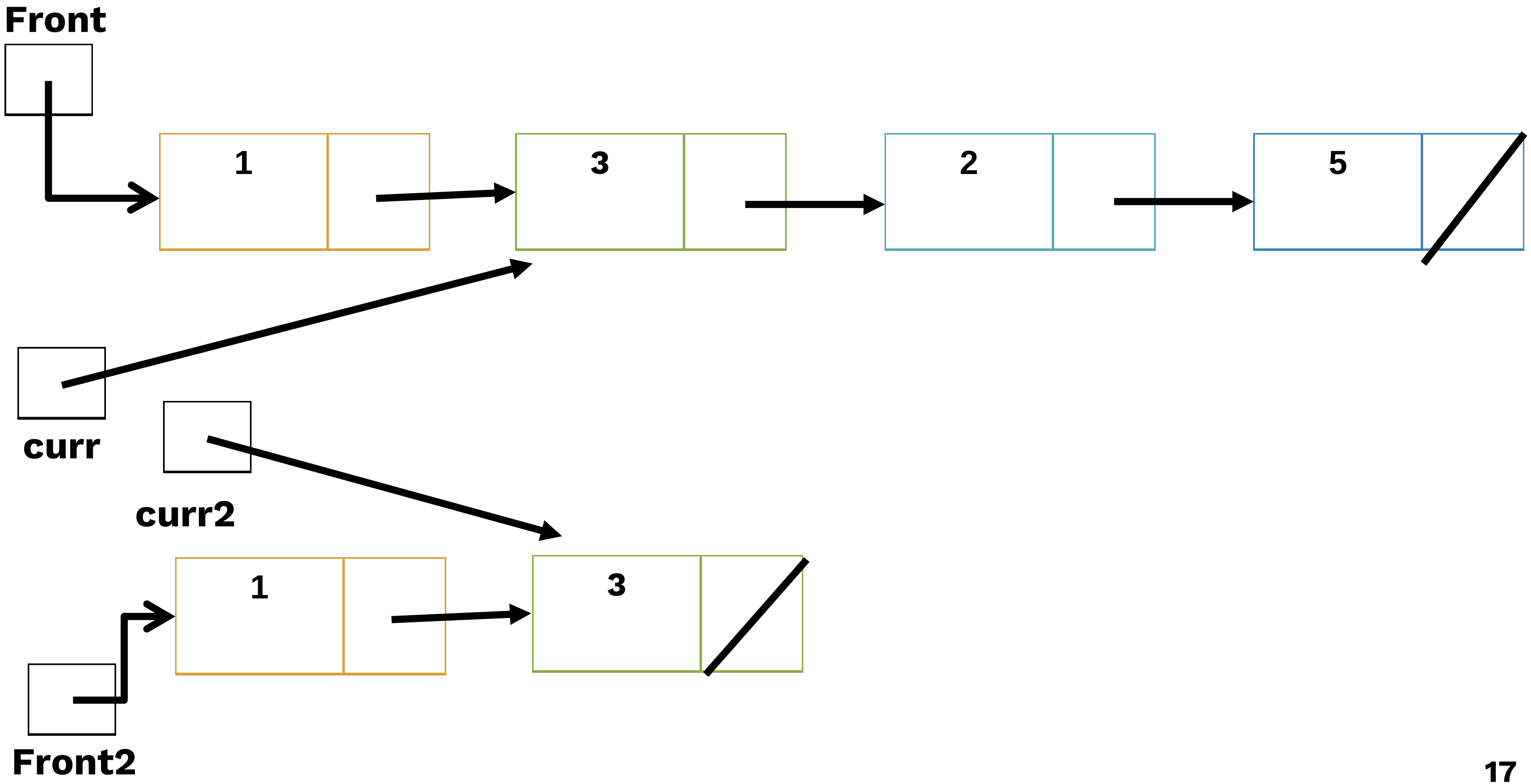
```
void doubleList(ListNode *front){
        if (front == nullptr) return;
        ListNode *curr = front;
        ListNode *curr2 = new ListNode(curr->data);
        ListNode *front2 = curr2;







}
```

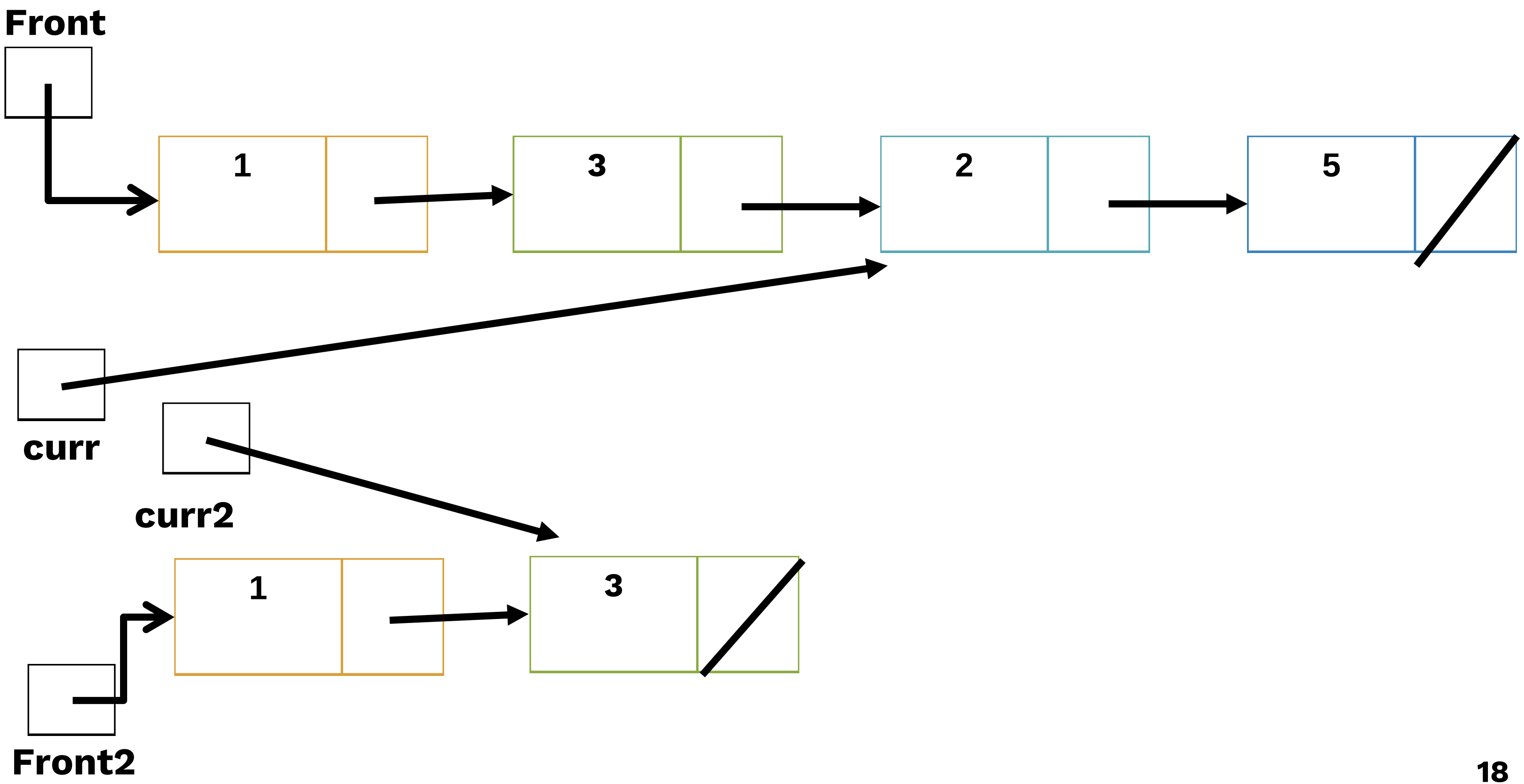
Deal with first node separately


Front
1
3
2
5
curr
curr2
1
Front2


Front
1
3
2
5
curr
curr2
1
3
Front2

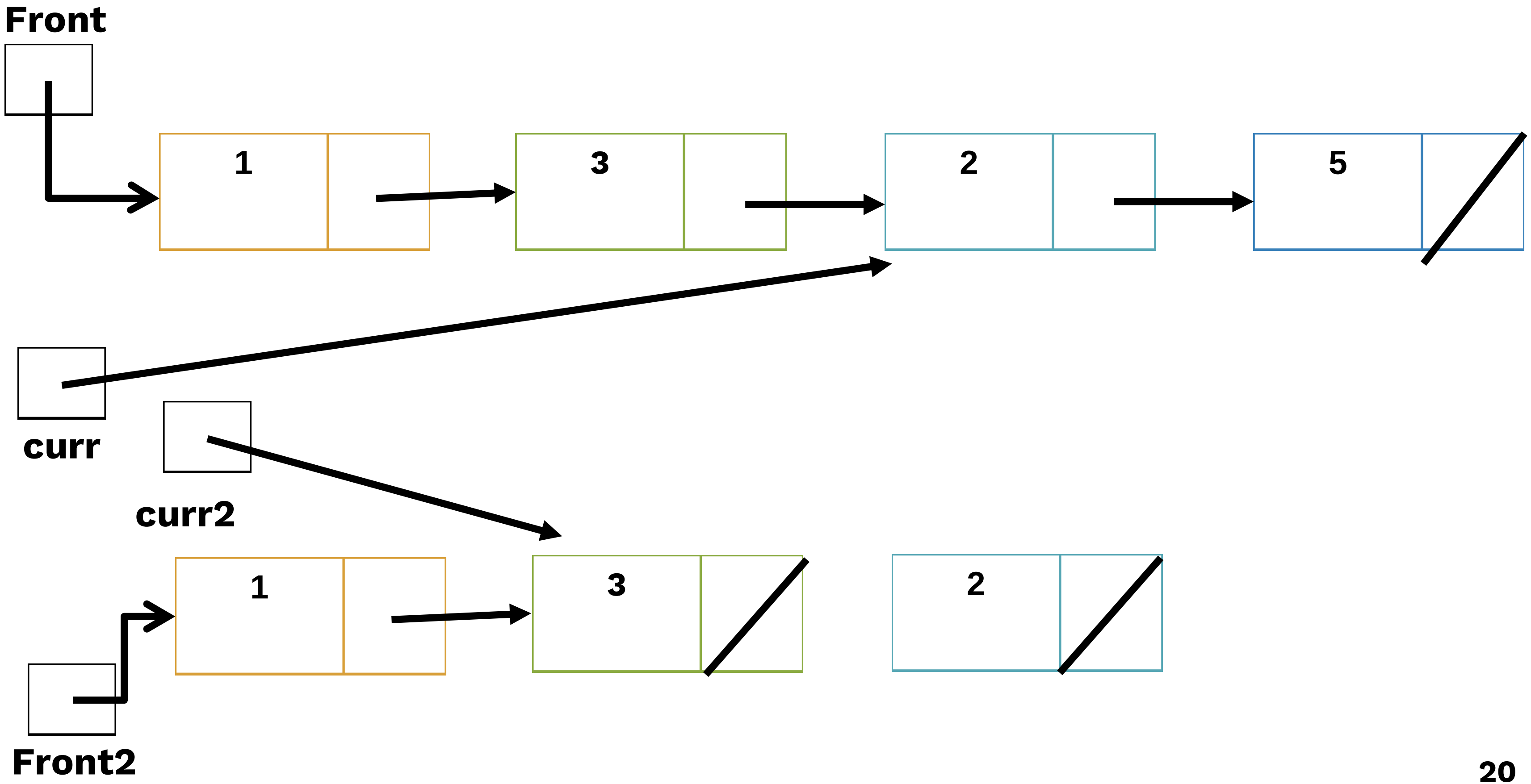

Front
1
3
2
5
curr
curr2
1
3
Front2

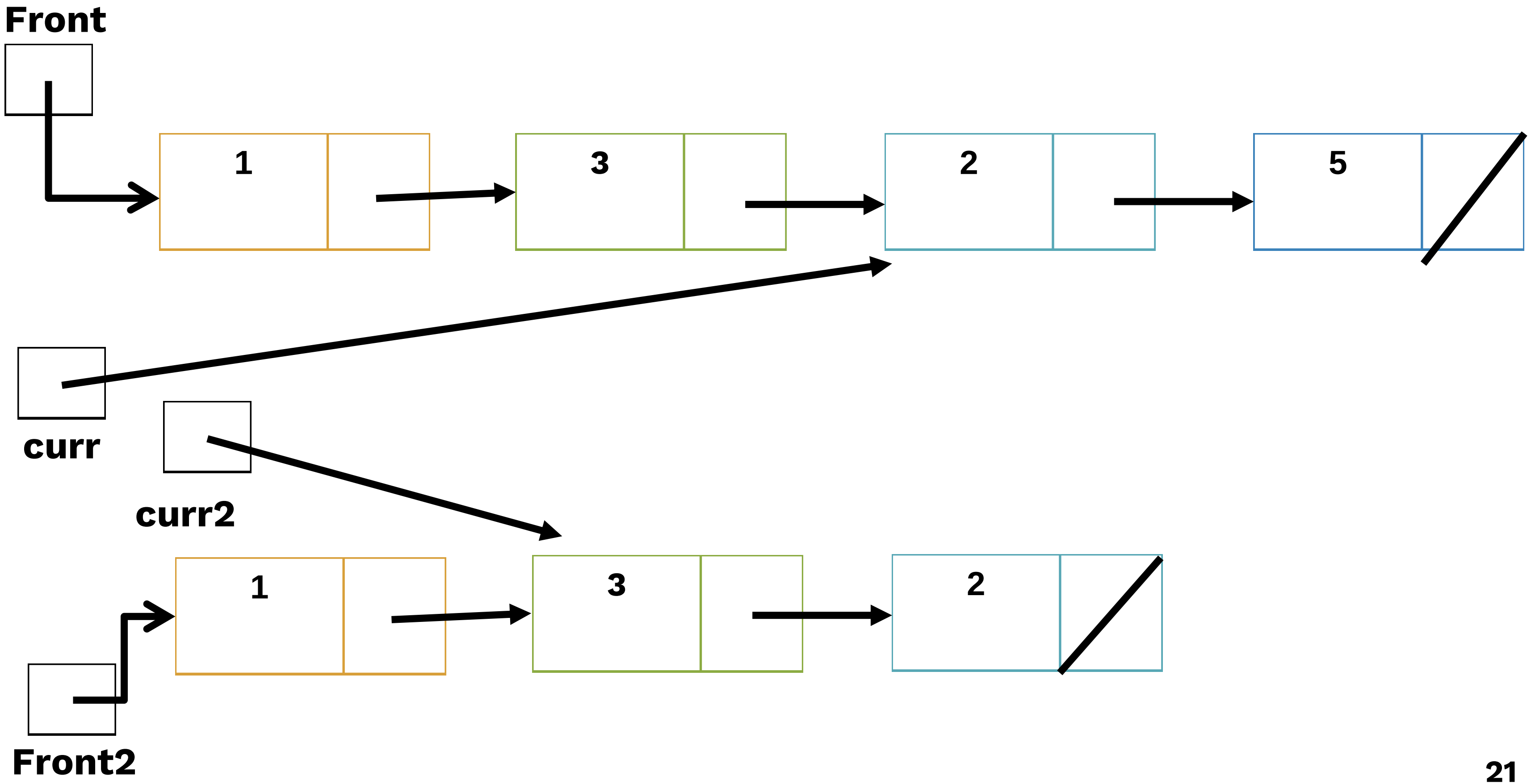

Front
1
3
2
5
curr
curr2
1
3
Front2

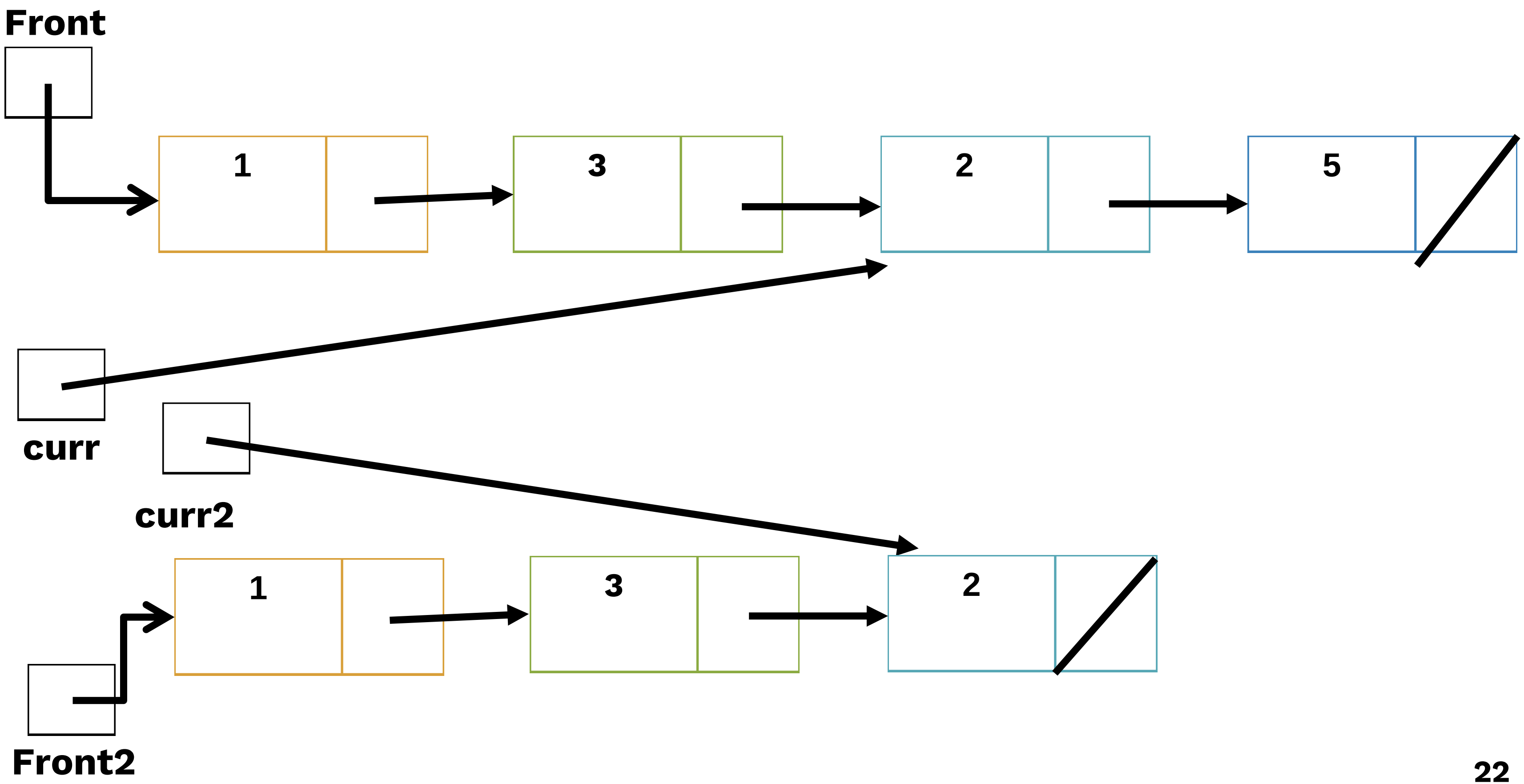

Front
1
3
2
5
curr
curr2
1
3
Front2

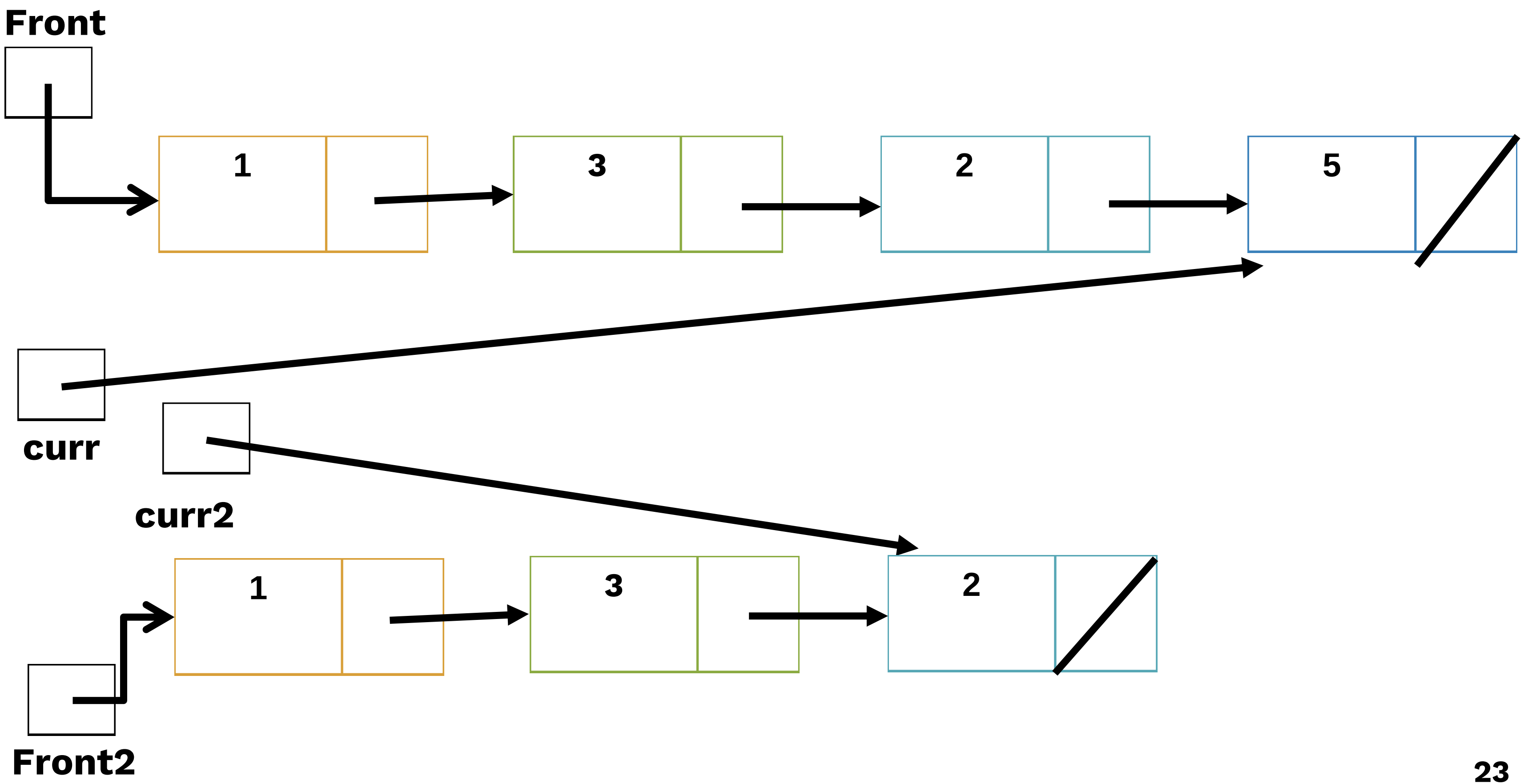
```
void doubleList(ListNode *front){
        if (front == nullptr) return;
        ListNode *curr = front;
        ListNode *curr2 = new ListNode(curr->data);
        ListNode *front2 = curr2;
        while (                     ){
                curr = curr->next;
                curr2->next = new ListNode(curr->data);
                curr2 = curr2->next;
        }
}
```

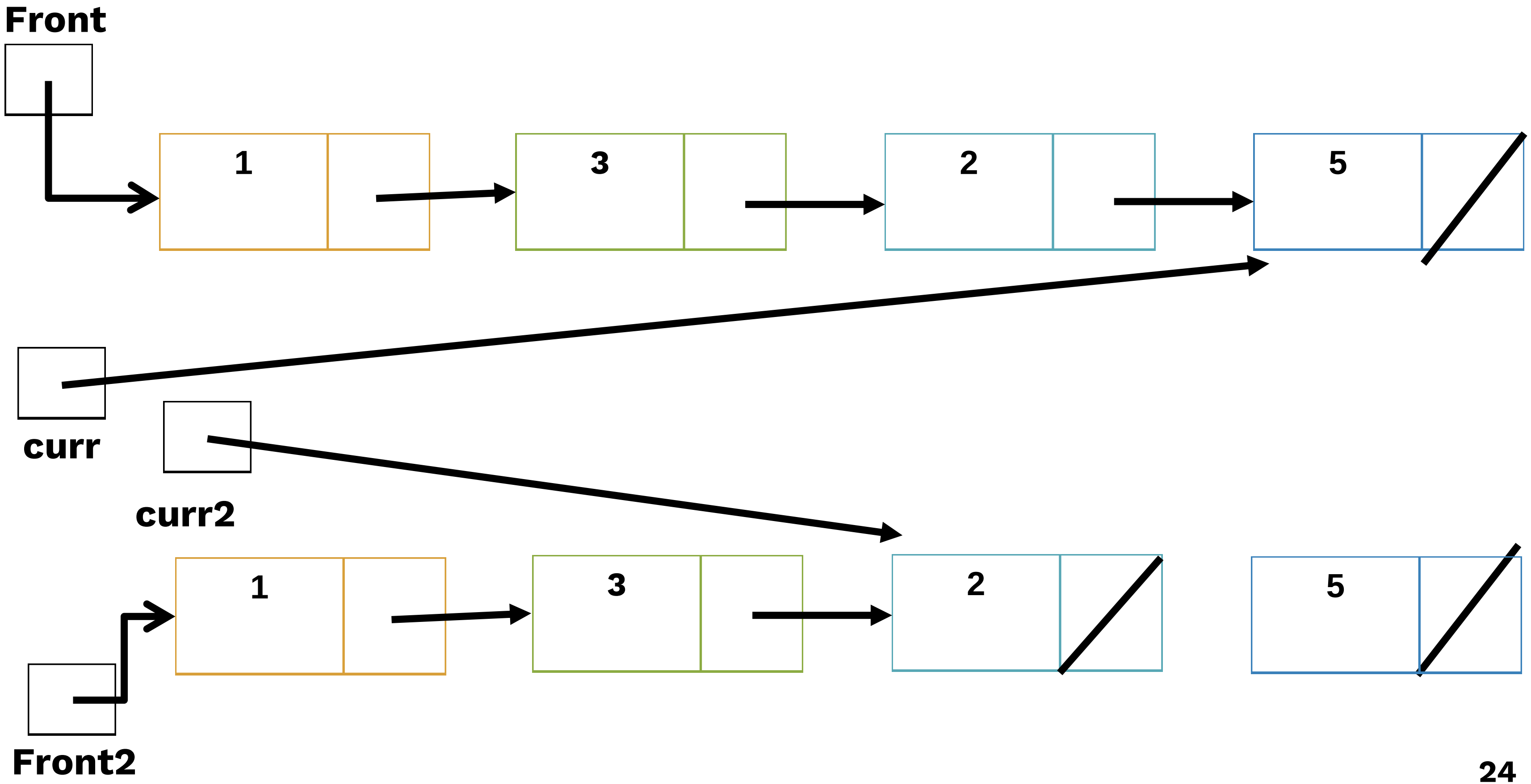
Create new node,
Adjust all pointers.

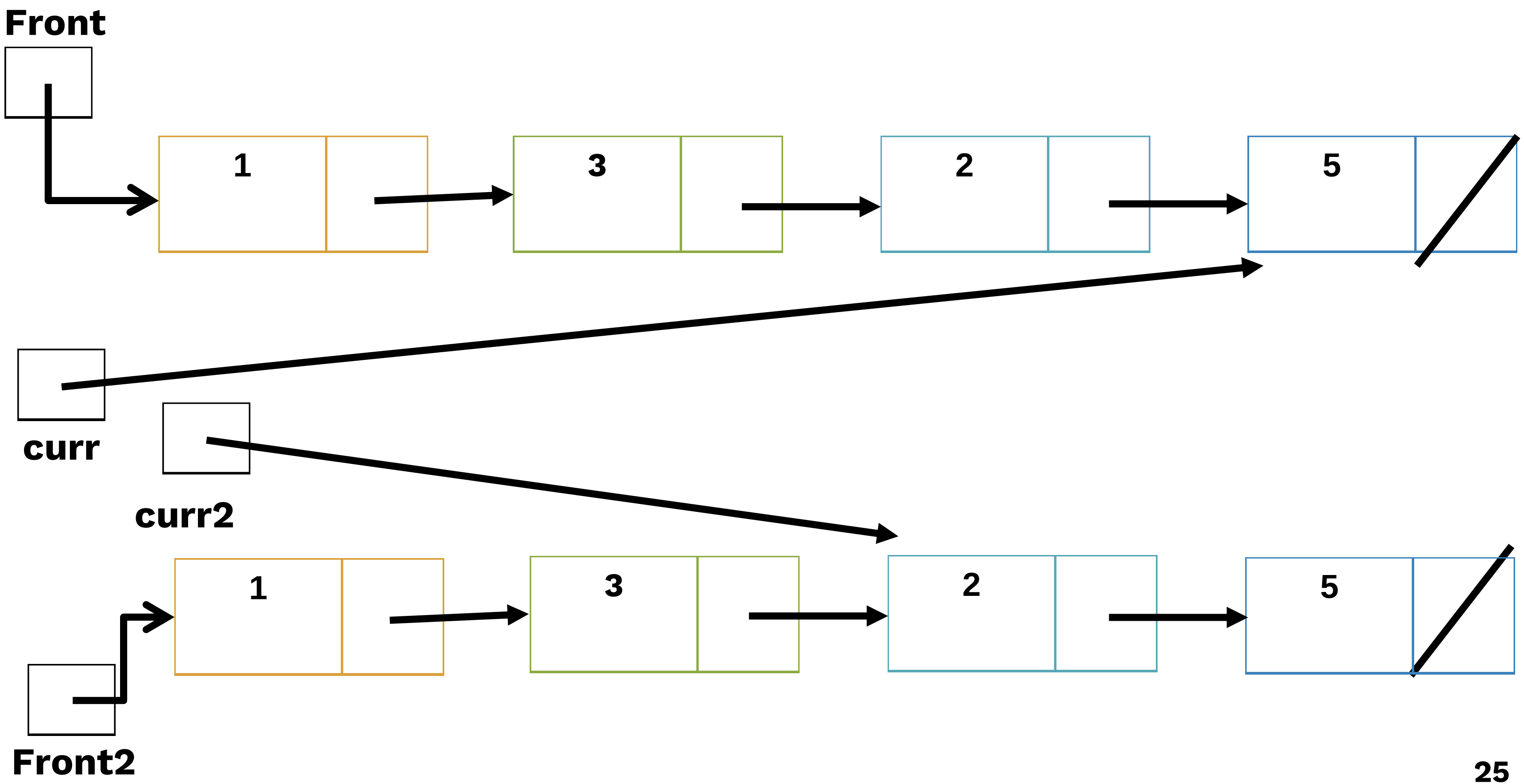

Front
1
3
2
5
curr
curr2
1
3
2
Front2


Front
1
3
2
5
curr
curr2
1
3
2
Front2


Front
1
3
2
5
curr
curr2
1
3
2
Front2


Front
1
3
2
5
curr
curr2
1
3
2
Front2


Front
1
3
2
5
curr
curr2
1
3
2
5
Front2

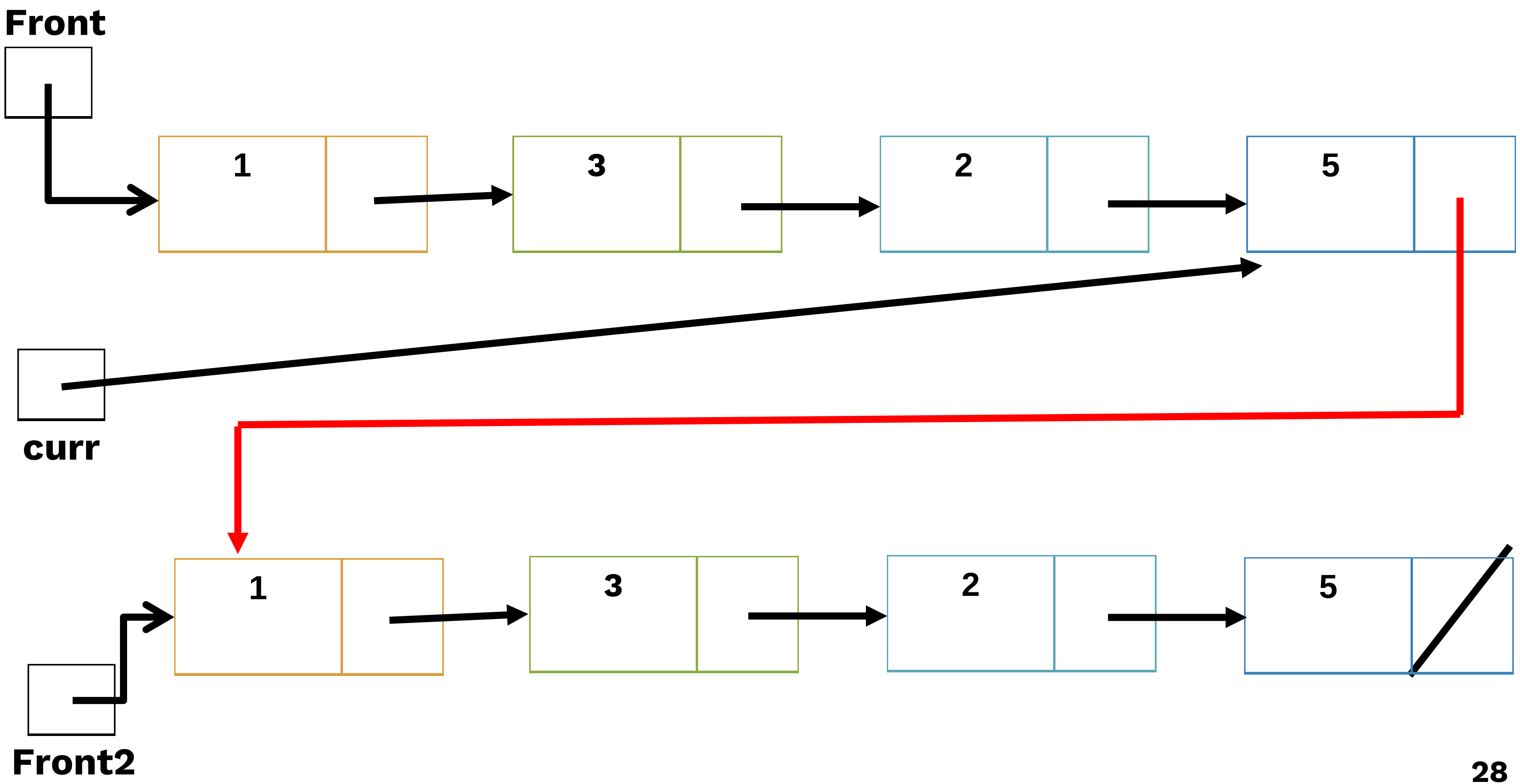

Front
1
3
2
5
curr
curr2
1
3
2
5
Front2

```
void doubleList(ListNode *front){
        if (front == nullptr) return;
        ListNode *curr = front;
        ListNode *curr2 = new ListNode(curr->data);
        ListNode *front2 = curr2;
        while (curr->next != nullptr){
                curr = curr->next;
                curr2->next = new ListNode(curr->data);
                curr2 = curr2->next;
        }

}
```

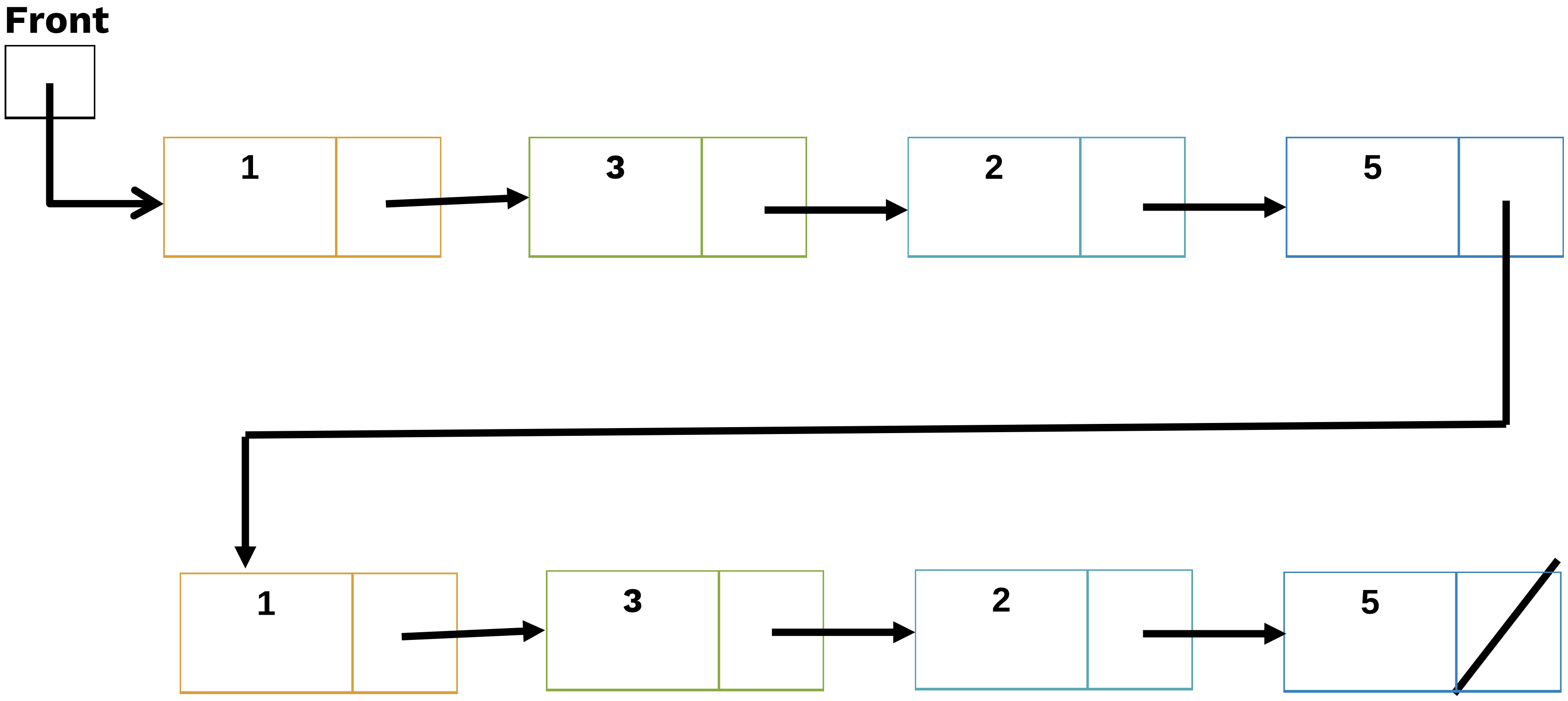
Stop when
we’re at end.

```
void doubleList(ListNode *front){
        if (front == nullptr) return;
        ListNode *curr = front;
        ListNode *curr2 = new ListNode(curr->data);
        ListNode *front2 = curr2;
        while (curr->next != nullptr){
                curr = curr->next;
                curr2->next = new ListNode(curr->data);
                curr2 = curr2->next;
        }
}
```

Why not
`curr1 != nullptr?`


Front
1
3
2
5
curr
1
3
2
5
Front2

```
void doubleList(ListNode *front){
        if (front == nullptr) return;
        ListNode *curr = front;
        ListNode *curr2 = new ListNode(curr->data);
        ListNode *front2 = curr2;
        while (curr->next != nullptr){
                curr = curr->next;
                curr2->next = new ListNode(curr->data);
                curr2 = curr2->next;
        }
        curr1->next = front2
}
```


Front
1
3
2
5
1
3
2
5

```
void doubleList(ListNode *front){
        if (front == nullptr) return;
        ListNode *curr = front;
        ListNode *curr2 = new ListNode(curr->data);
        ListNode *front2 = curr2;
        while (curr->next != nullptr){
                curr = curr->next;
                curr2->next = new ListNode(curr->data);
                curr2 = curr2->next;
        }
        curr1->next = front2
}
```

Lexicon

HashMap

Queue

Vector

Stack

Priority Queue

Map

Deque

DawgLexicon

Set

HashSet

Graph

Abstract Data Types (ADTs)

Focus on functions and behavior, not how they are implemented.

Implementing ADTs

- Wed: implementing Vector

- Stored data in an array.
- Managed dynamic memory.

- Many other ways to implement, as long as it behaves like a Vector.

Implementing ADTs

EXTERNALLY

- All three implementations have identical behavior.
- Exact same methods.

INTERNALLY

- Store data in completely different ways.
- Different Big-O runtimes (!)

Queue

- First In, First Out (FIFO)

`Queue<Stack<string> > wordLadders;`

Key Methods

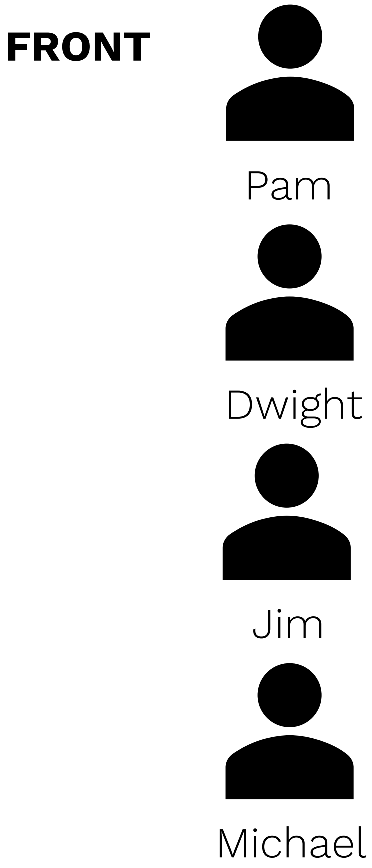
`enqueue`
`dequeue`

`front`

`isEmpty`
`clear`
`toString`

Queue

```
q.enqueue(“Pam”)
q.enqueue(“Dwight”)
q.enqueue(“Jim”)
q.enqueue(“Michael”)

q.dequeue()  // returns “Pam”
q.dequeue()  // returns “Dwight”
q.dequeue()  // returns “Jim”
q.dequeue()  // returns “Michael”
```



FRONT

Pam

Dwight

Jim

Michael

Priority Queue

- **Most urgent** priority item is dequeued.

Key Methods

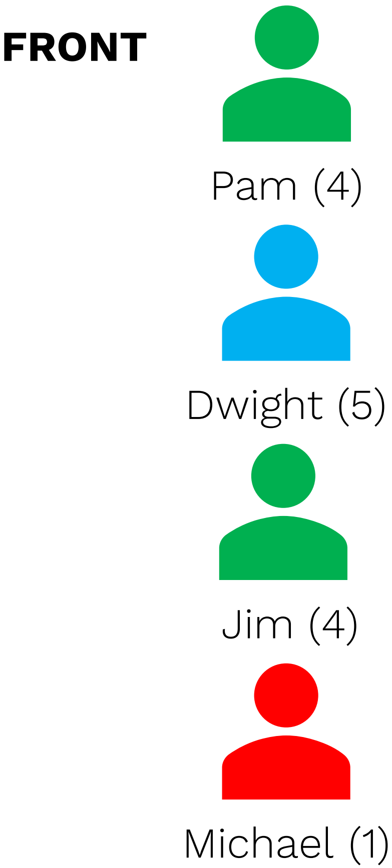
`enqueue`
`dequeue`

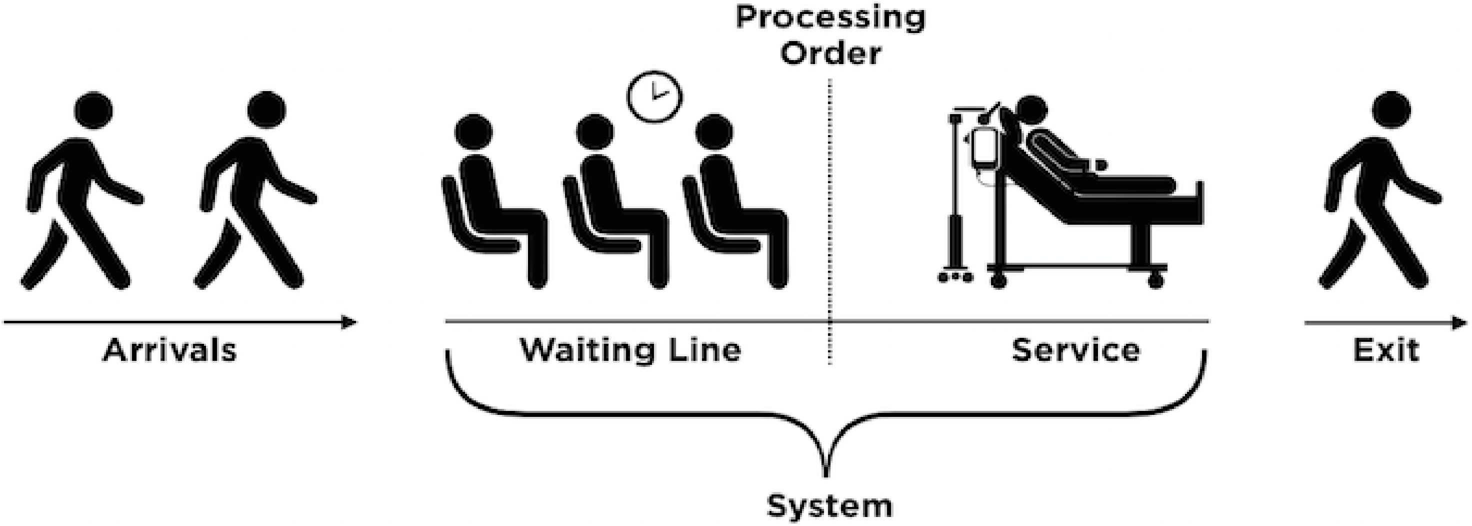
`front`

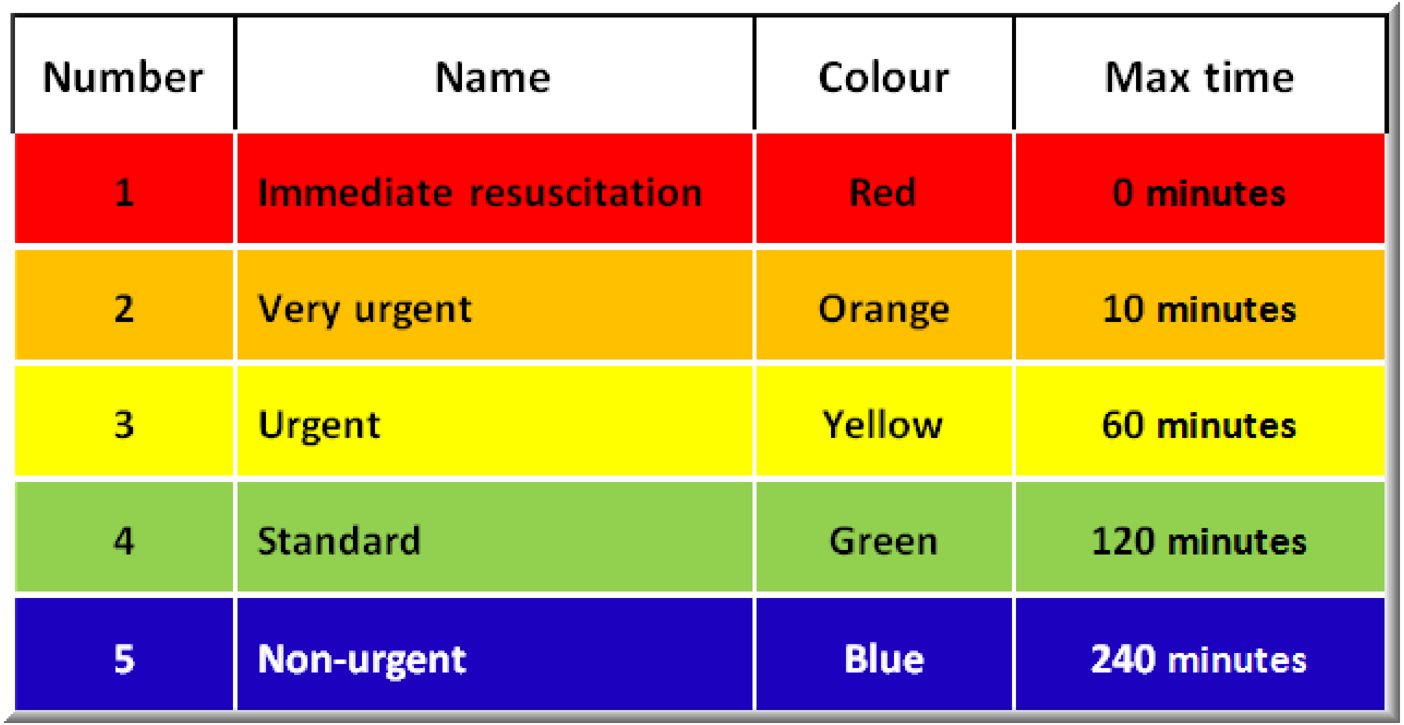
`isEmpty`
`clear`
`toString`

Priority Queue

```
pq.enqueue(“Pam”, 4)
pq.enqueue(“Dwight”, 5)
pq.enqueue(“Jim”, 4)
pq.enqueue(“Michael”, 1)

pq.dequeue() // returns “Michael”
pq.dequeue() // returns “Pam”
pq.dequeue() // returns “Jim”
pq.dequeue() // returns “Dwight”
```




Processing
Order
Arrivals
Waiting Line
Service
Exit
System



Number	Name	Colour	Max time
1	Immediate resuscitation	Red	0 minutes
2	Very urgent	Orange	10 minutes
3	Urgent	Yellow	60 minutes
4	Standard	Green	120 minutes
5	Non-urgent	Blue	240 minutes

Patient Queue

- **Most urgent** priority patient is dequeued.

Key Methods

`addPatient`
`processPatient`
`upgradePatient`

`frontPatient`
`frontPriority`

`isEmpty`
`clear`
`toString`

Demo!

Priority Queue

```
pq.addPatient(“Pam”, 4)
pq.addPatient(“Dwight”, 5)
pq.addPatient(“Jim”, 4)
pq.addPatient(“Michael”, 1)
pq.processPatient() // returns “Michael”
pq.processPatient() // returns “Pam”
pq.upgradePatient(“Dwight”, 3)
pq.dequeue() // returns “Dwight”
pq.dequeue() // returns “Jim”
```

FRONT

Pam (4)

Dwight (5)

Jim (4)

Michael (1)

Priorities

- Most urgent = lowest priority number

MOST URGENT

Michael (1)

Pam (4)

Dwight (5)

LEAST URGENT

Tiebreaker and Duplicates

`upgradePatient`

- Vector: find patient, most urgent priority, break ties by earlier timestamp.
- Linked List: find patient, most urgent priority, break ties by order of linked list.
- Heap: find patient, most urgent priority, break ties by lexicographical order (use string comparison).

PatientQueue Constructor

```
// Constructor
PatientQueue()


// Destructor
~PatientQueue()
```

PatientQueue Member Methods

```
// adds new patient to queue
void newPatient(string name, int priority)


// returns and removes highest priority patient
string processPatient()


// updates patient to higher priority
void upgradePatient(string name, int newPriority)
```

PatientQueue Member Methods

```
// returns name of highest-priority patient
string frontName()

// returns priority of highest-priority patient
int frontPriority()

// removes all patients
void clear()

// returns the PatientQueue as a string
string toString()
```

```
PatientQueue()

~PatientQueue()

void newPatient(string name, int priority)

string processPatient()

void upgradePatient(string name, int newPriority)

string frontName()

int frontPriority()

void clear()

string toString()
```

Don’t Change the Header or Add Public Methods!

The Assignment

Implement a Priority Queue in three **different** ways.

Unsorted Vector

Sorted Linked List

Binary Min-Heap

Getting Started

Tip: complete Vector implementation by tonight!

Files

Header Files	**CPP Files**	**Don’t Edit** (unless extensions)
VectorPatientQueue.h	VectorPatientQueue.cpp	patientnode.h
LinkedListPatientQueue.h	LinkedListPatientQueue.cpp	patientqueue.h
HeapPatientQueue.h	HeapPatientQueue.cpp	hospital.cpp
		patientnode.cpp

CPP Files

- All three are nearly identical.
- Same public methods to implement.
- **Do not change method headers!**

```cpp
VectorPatientQueue::VectorPatientQueue() {
    // TODO: write this constructor
}

VectorPatientQueue::~VectorPatientQueue() {
    // TODO: write this destructor
}

void VectorPatientQueue::clear() {
    // TODO: write this function
}

string VectorPatientQueue::frontName() {
    // TODO: write this function
    return "";   // this is only here so it will compile
}

int VectorPatientQueue::frontPriority() {
    // TODO: write this function
    return 0;   // this is only here so it will compile
}

bool VectorPatientQueue::isEmpty() {
    // TODO: write this function
    return false;   // this is only here so it will compile
}

void VectorPatientQueue::newPatient(string name, int priority) {
    // TODO: write this function
}

string VectorPatientQueue::processPatient() {
    // TODO: write this function
    return "";   // this is only here so it will compile
}

void VectorPatientQueue::upgradePatient(string name, int newPriority) {
    // TODO: write this function
}

string VectorPatientQueue::toString() {
    // TODO: write this function
    return ""; // this is only here so it will compile
}
```

Header Files

- Add your instance variables.
- Add your private member methods.
- Add your structs (if necessary).
- **Don’t change public methods!**

```cpp
#pragma once

#include <iostream>
#include <string>
#include "patientqueue.h"
using namespace std;

class VectorPatientQueue : public PatientQueue {
public:
    VectorPatientQueue();
    ~VectorPatientQueue();
    string frontName();
    void clear();
    int frontPriority();
    bool isEmpty();
    void newPatient(string name, int priority);
    string processPatient();
    void upgradePatient(string name, int newPriority);
    string toString();

private:
    // TODO: add specified member variable(s)
    // TODO: add any member functions necessary

};
```

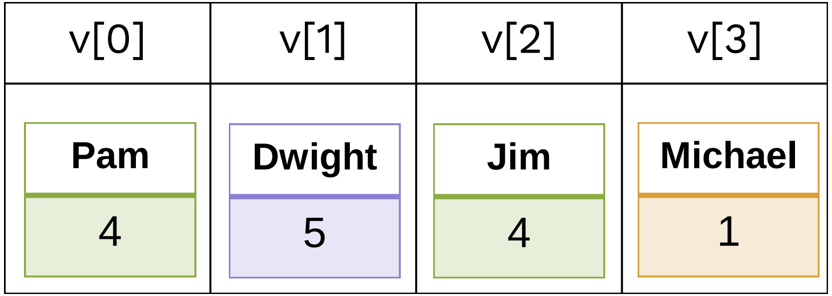
Summary of Assignment

For Vector, Linked List, and Heap:

- Add instance variables.
- Implement constructor and destructor.
- Implement all 7 member methods.
- Test, test, test!

Summary of Assignment

Unsorted Vector	**Sorted Linked List**	**Binary Heap**
Create your own struct.	Use provided struct.	Create your own struct.
Store elements in **unsorted** order in a Vector of structs.	Store elements in sorted order using a linked list.	Organized in a **heap** (stored as an array of structs).
Maintain Vector.	Maintain “front” pointer.	Maintain array.



v[0]	v[1]	v[2]	v[3]
Pam 4	**Dwight** 5	**Jim** 4	**Michael** 1

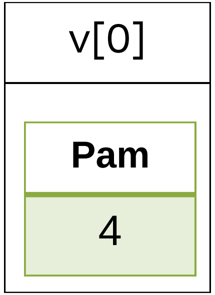
Unsorted Vector

Simple but slow implementation.

Vector Implementation

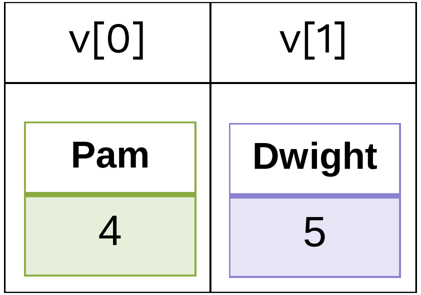
Empty Vector

Vector Implementation



Add patient Pam, priority 4

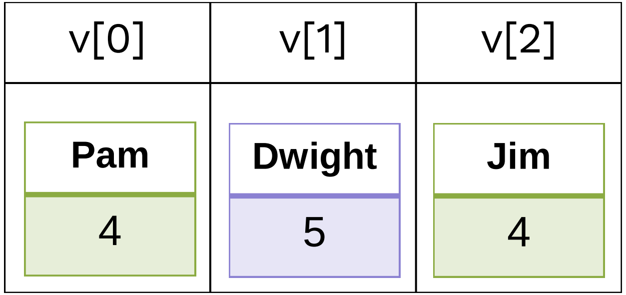
Vector Implementation



v[0]	v[1]
Pam 4	**Dwight** 5

Add patient Dwight, priority 5

Vector Implementation



v[0]	v[1]	v[2]
Pam 4	**Dwight** 5	**Jim** 4

Add patient Jim, priority 4

Vector Implementation

v[0]	v[1]	v[2]	v[3]
Pam 4	**Dwight** 5	**Jim** 4	**Michael** 1

Add patient Michael, priority 1

Vector Implementation

v[0]	v[1]	v[2]	v[3]
Pam 4	**Dwight** 5	**Jim** 1	**Michael** 1

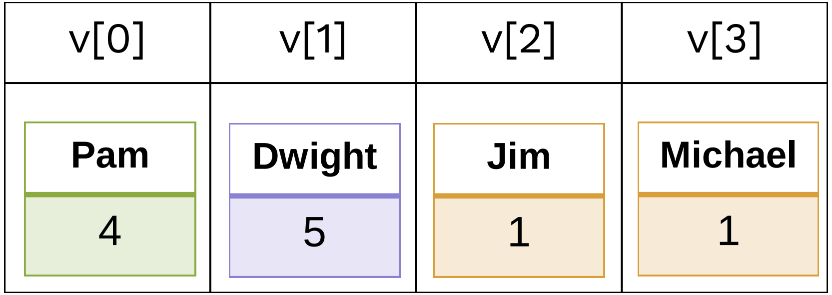
Upgrade Jim to priority 1

Vector Implementation

v[0]	v[1]	v[2]	v[3]
Pam	**Dwight**	**Jim**	**Michael**
4	5	1	1

Now we process a patient.
Do we process Jim or Michael?

Vector Implementation



v[0]	v[1]	v[2]	v[3]
Pam	**Dwight**	**Jim**	**Michael**
4	5	1	1

Michael - he had priority 1 first

Vector Implementation

- You may use an **int** for a **timestamp** in your struct.
- You have to determine how to track that!

Name	Pam
Priority	4
Timestamp	1

Summary: Vector

`PatientQueue()`	O(1)
`~PatientQueue()`	O(1)
`newPatient(name, priority)`	O(1)
`processPatient()`	O(N)
`frontName()`	O(N)
`frontPriority()`	O(N)
`upgradePatient(name, newP)`	O(N)
`isEmpty()`	O(1)
`clear()`	O(1)
`toString()`	O(N)

Use the Big-O as a hint as to how to implement.

Don’t overthink it!

Summary: Vector

`PatientQueue()`	`O(1)`
`~PatientQueue()`	`O(1)`
`newPatient(name, priority)`	`O(1)`
`processPatient()`	**`O(N)`**
`frontName()`	**`O(N)`**
`frontPriority()`	**`O(N)`**
`upgradePatient(name, newP)`	**`O(N)`**
`isEmpty()`	`O(1)`
`clear()`	`O(1)`
`toString()`	`O(N)`

Vector is unsorted!

Must loop over entire vector to find patient with minimum priority.

Summary: Vector

`PatientQueue()`	O(1)
`~PatientQueue()`	O(1)
`newPatient(name, priority)`	O(1)
`processPatient()`	O(N)
`frontName()`	O(N)
`frontPriority()`	O(N)
`upgradePatient(name, newP)`	O(N)
`isEmpty()`	O(1)
`clear()`	O(1)
`toString()`	**O(N)**

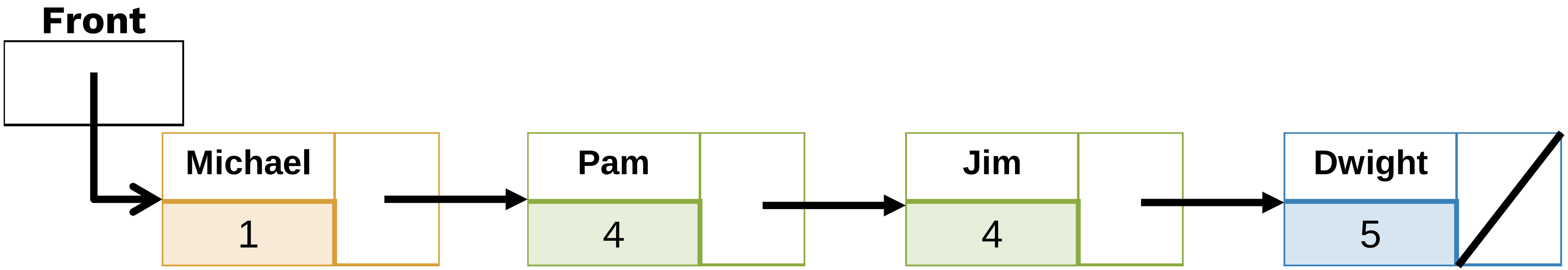
CONSOLE

```
"{4:Pam, 5:Dwight,
1:Jim, 1:Michael}"
```

For Vector, order of printing is not important

Questions?

I’m being a little vague so you have some design choices as well!

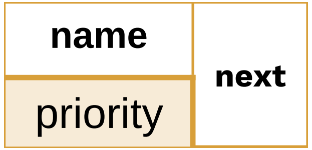


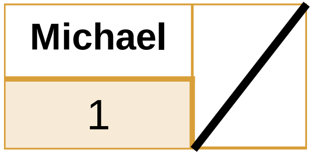
Linked List

Show off your new shiny pointer skills!

Struct Given to You

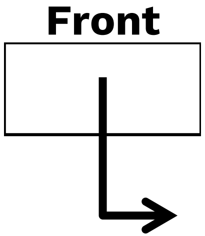
```
struct PatientNode {
        string name;
        int priority;
        PatientNode* next;
};
```





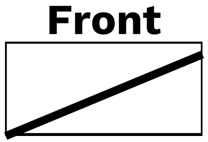
Instance Variables

```
class VectorPatientQueue : public PatientQueue {
      public:
            …
      private:
            PatientNode* front;
            // nothing else is allowed!!!
};
```



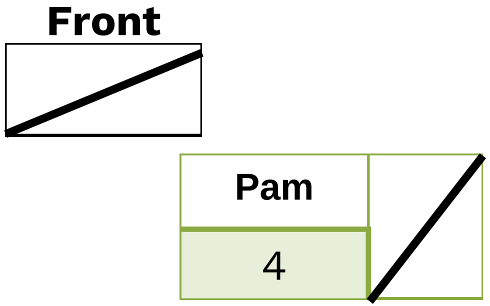
Linked List

- Maintain a **front** pointer to a linked list.
- Initially **`nullptr`**.



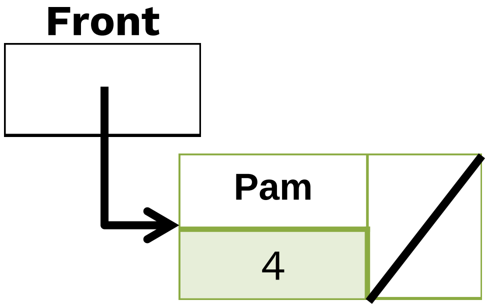
Linked List

- As patients added, keep them **sorted** in priority.



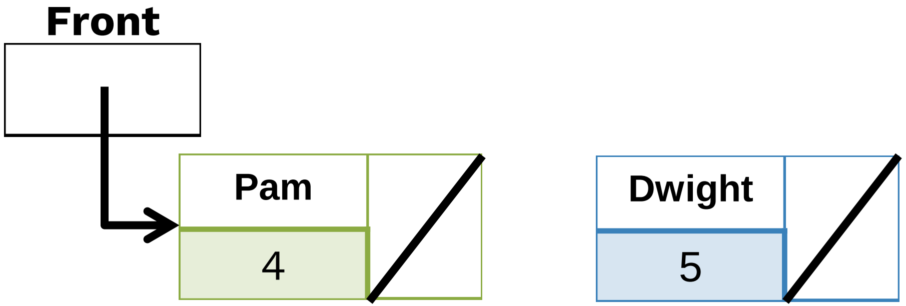
Linked List

- As patients added, keep them **sorted** in priority.
- Last patient has next pointer of **`nullptr`**.



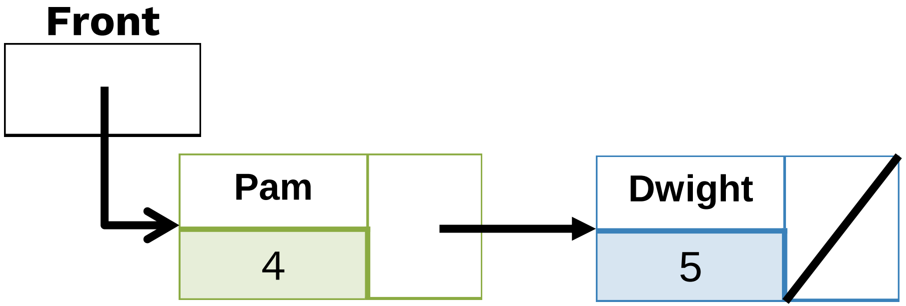
Linked List

- As patients added, keep them **sorted** in priority.



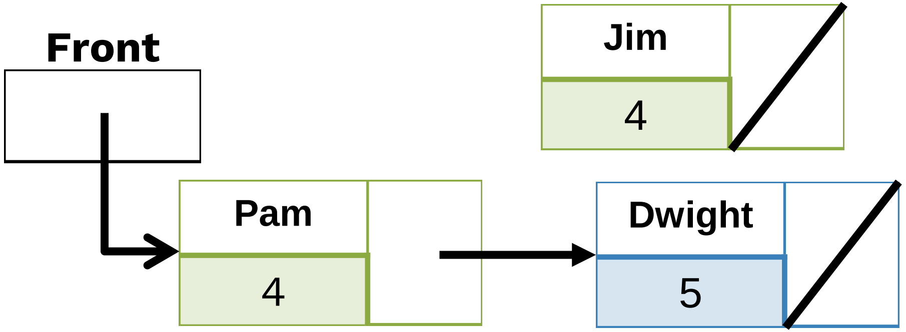
Linked List

- Keep how adding to different parts of the list require different pointer gymnastics.



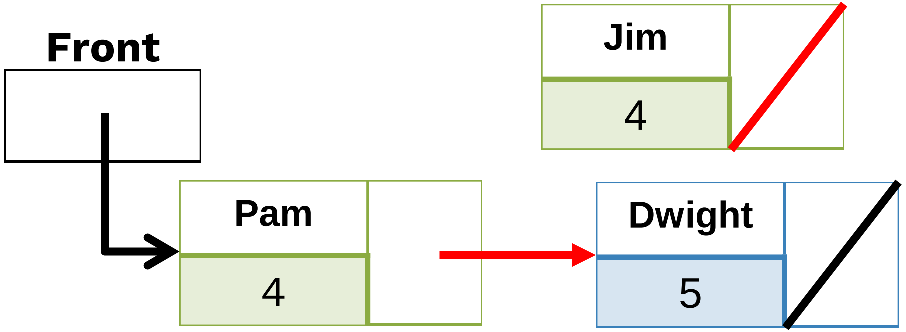
Linked List

- What happens if we try to insert between two existing patients?



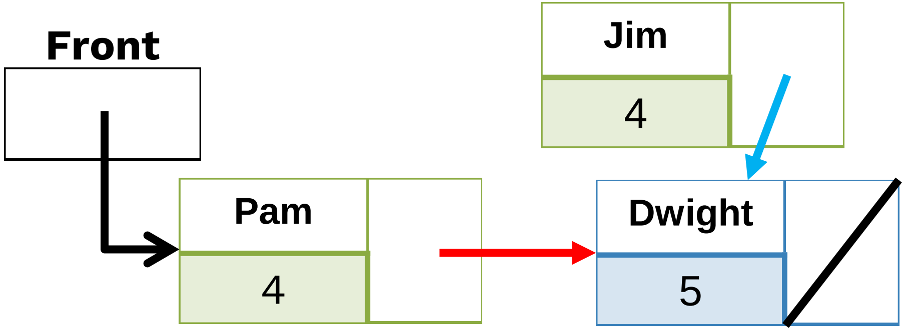
Linked List

- Which pointers need to be modified?



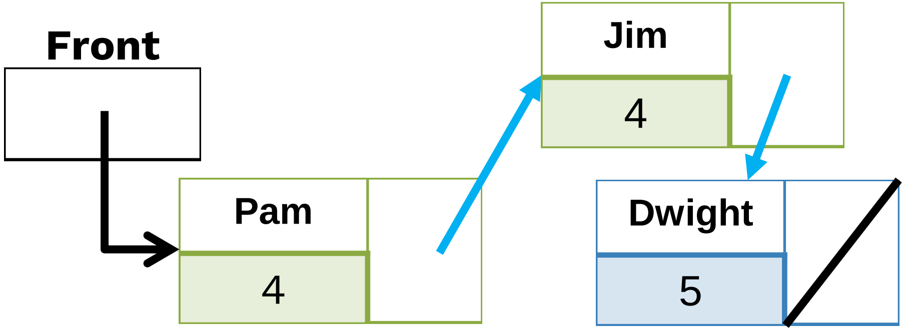
Linked List

- We deal with this pointer first. Why?



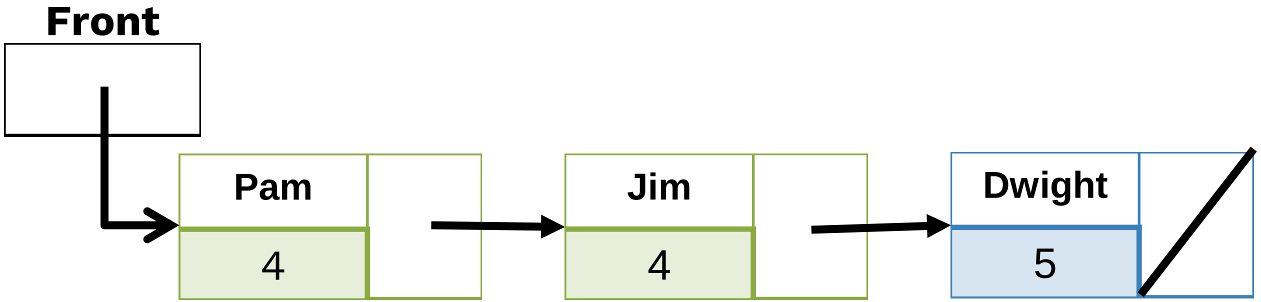
Linked List

- Order matters! Don’t lose the rest of your list!



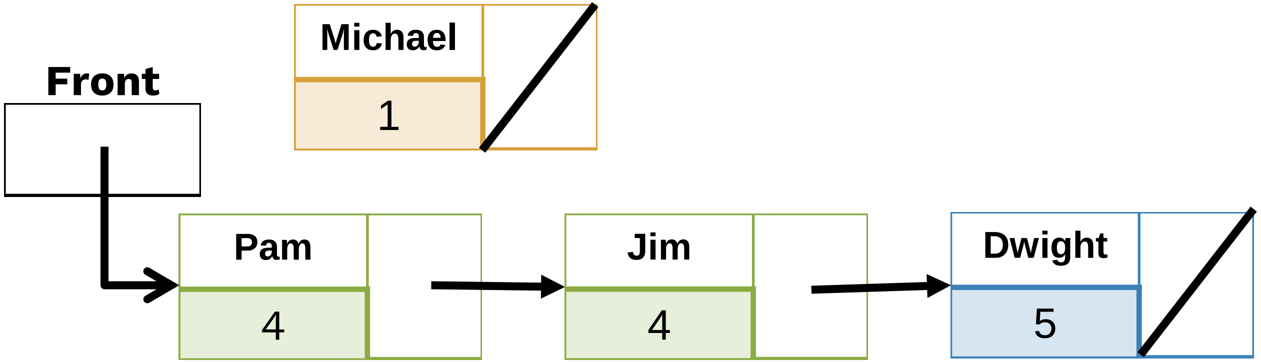
Linked List

- And here’s our new list.



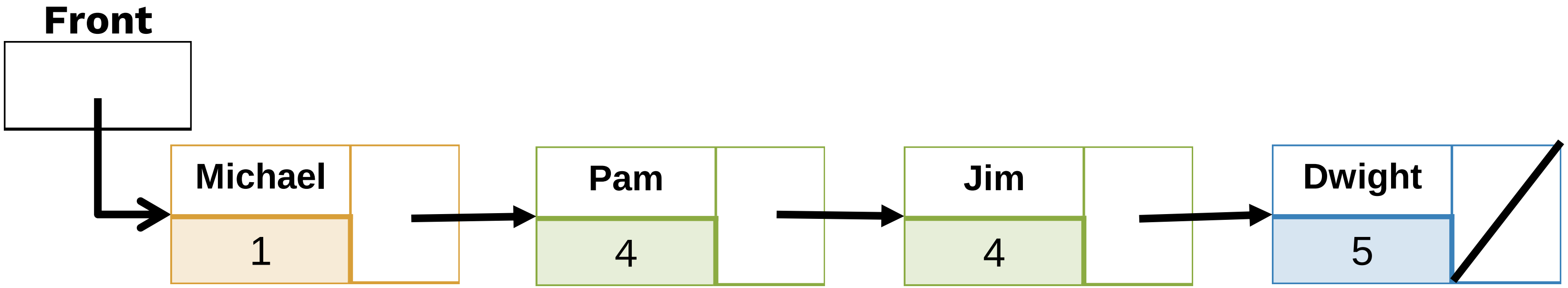
Linked List

- Let’s add one more.



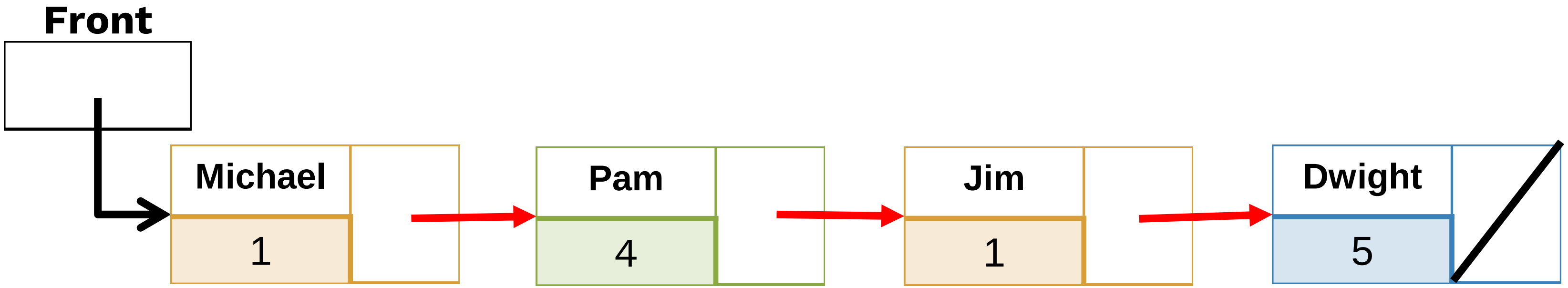
Linked List

- Notice that different pointers were being moved depending on where the patient is added.



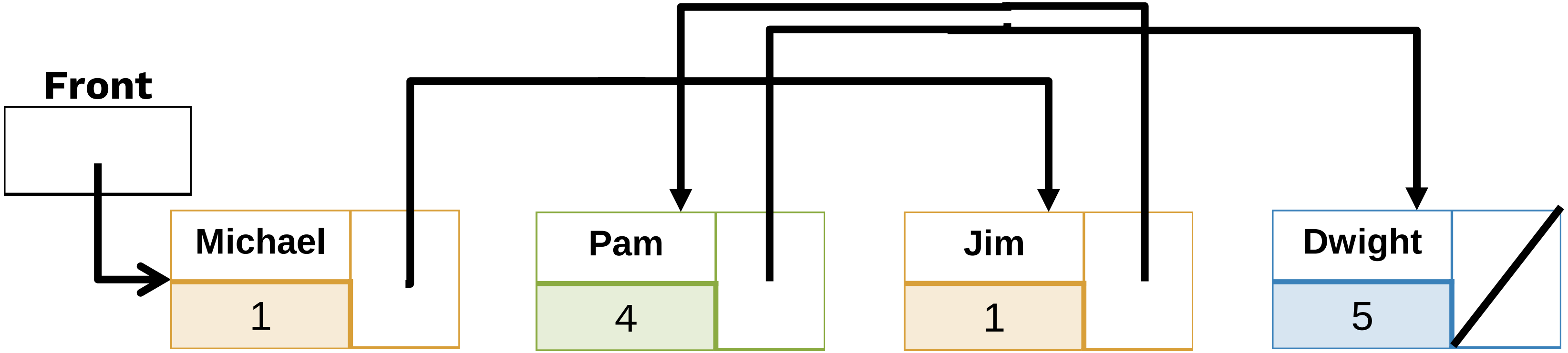
Linked List

- Same deal with upgrading Jim to priority 1.



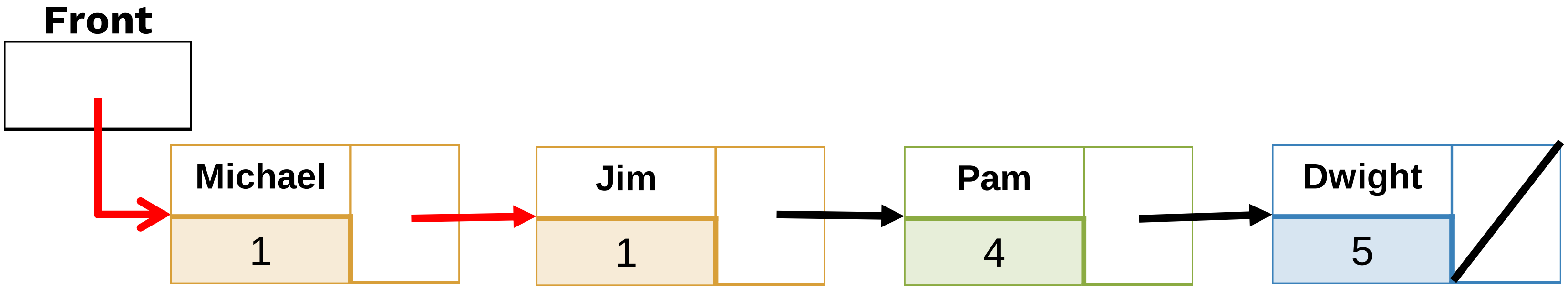
Linked List

- Same deal with upgrade and removing.



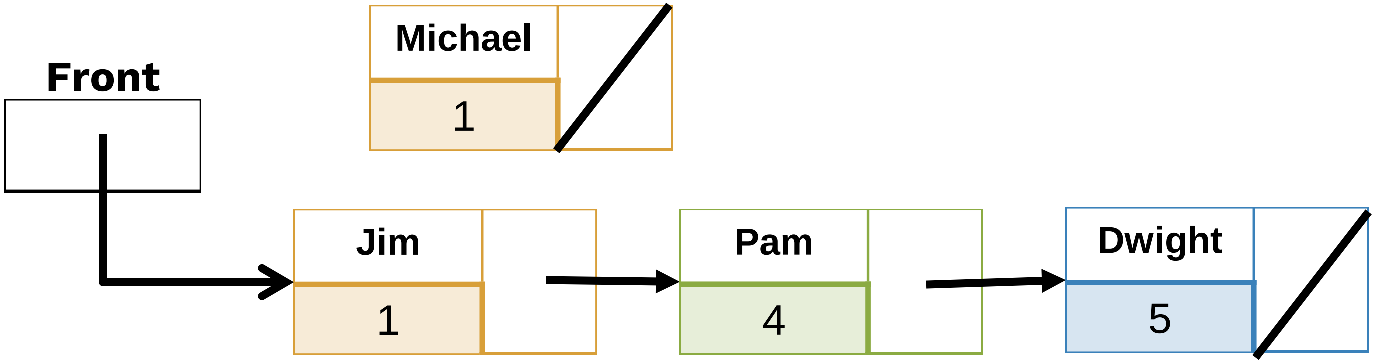
Linked List

- And processing patient?



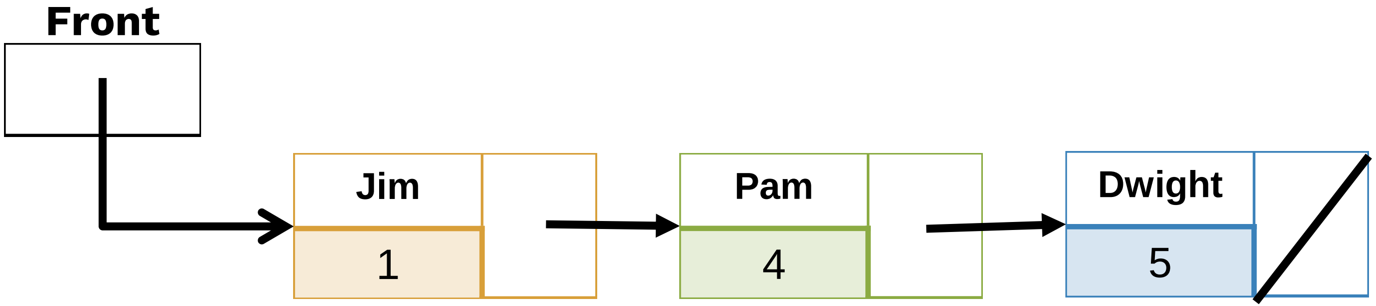
Linked List

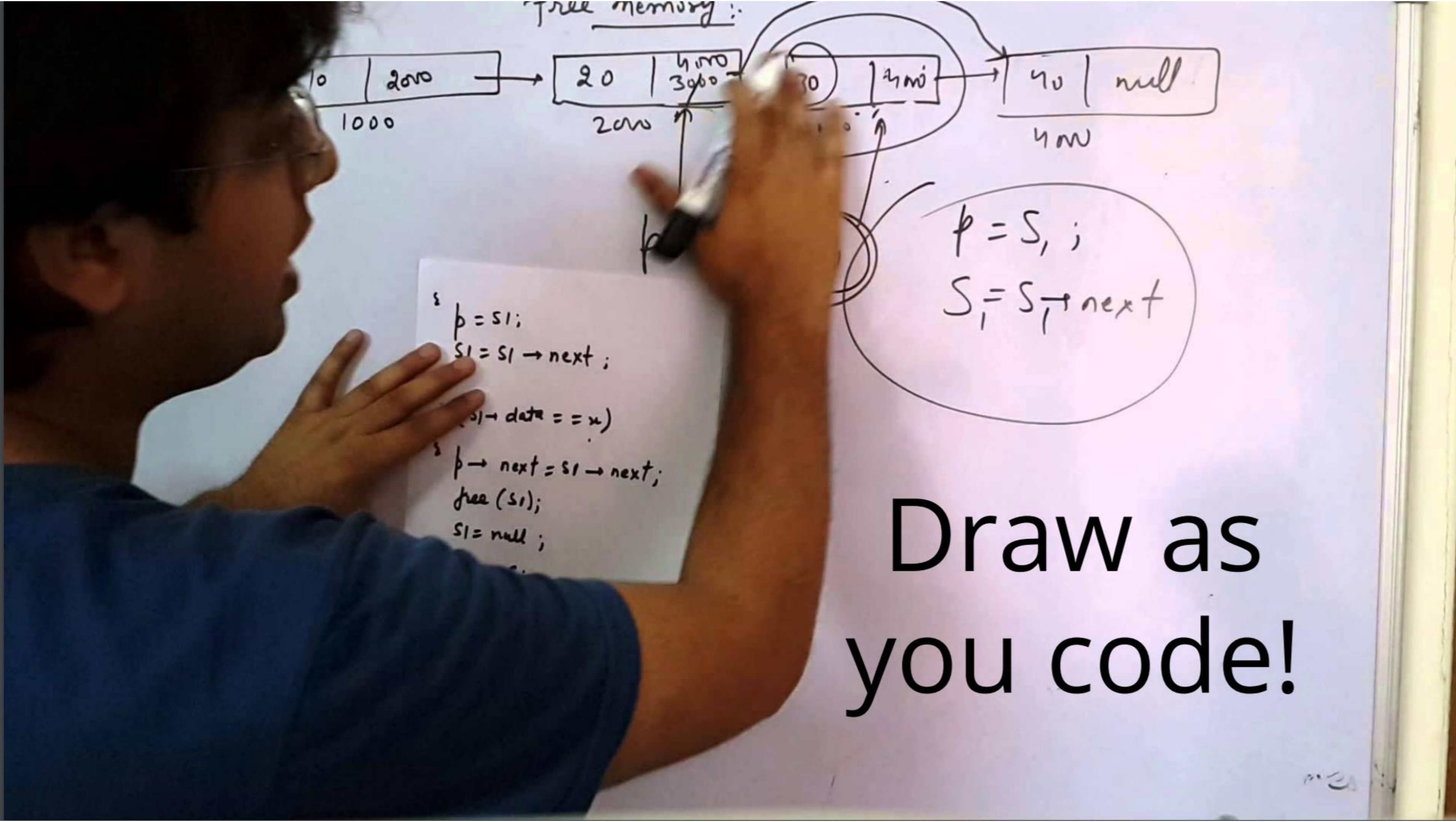
- What happens to Michael?



Linked List

- Michael gets deleted. **Don’t forget to free memory!**



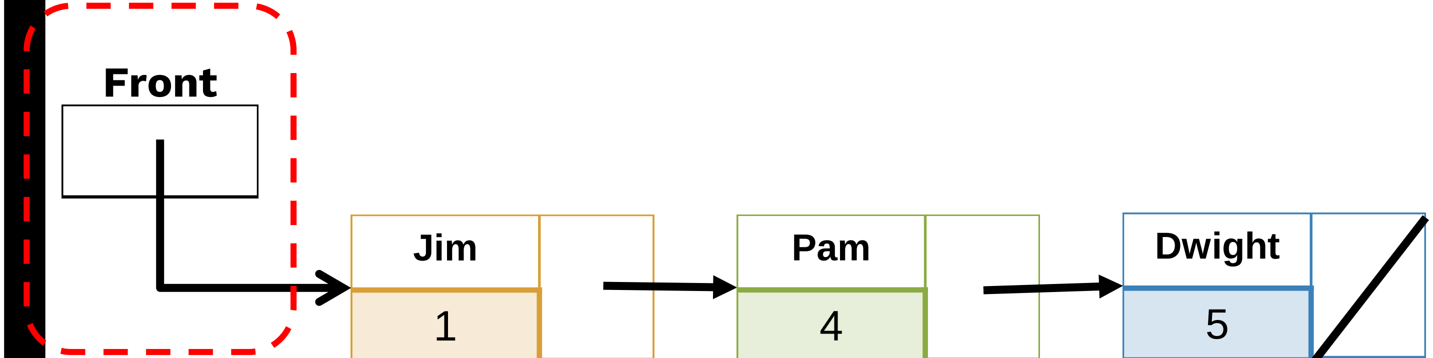


```
p = s1;
s1 = s1 → next ;

s1→ data = = x)
{
p → next = s1 → next;
free (s1);
s1 = null ;
```

Draw as you code!

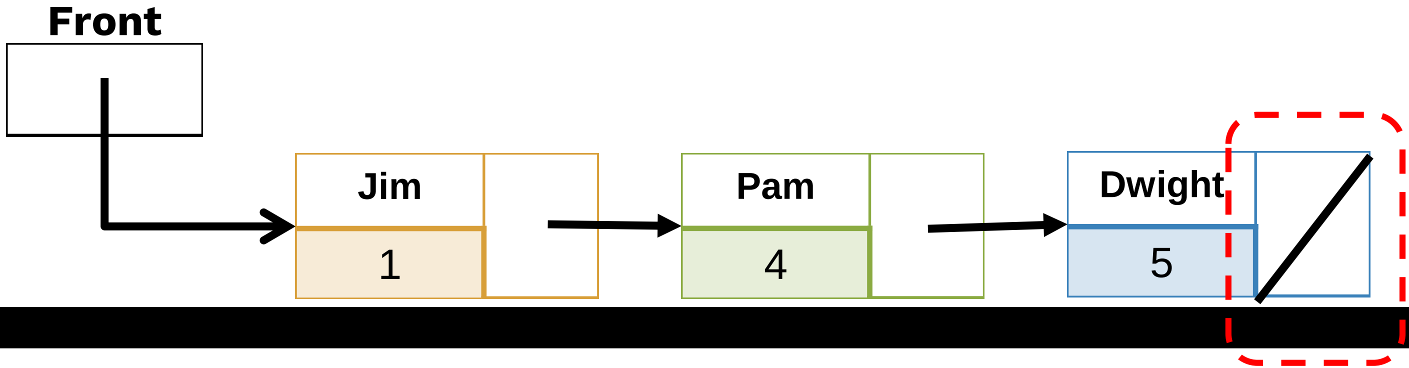
Reminders

- The class should **only** maintain your **front** pointer.



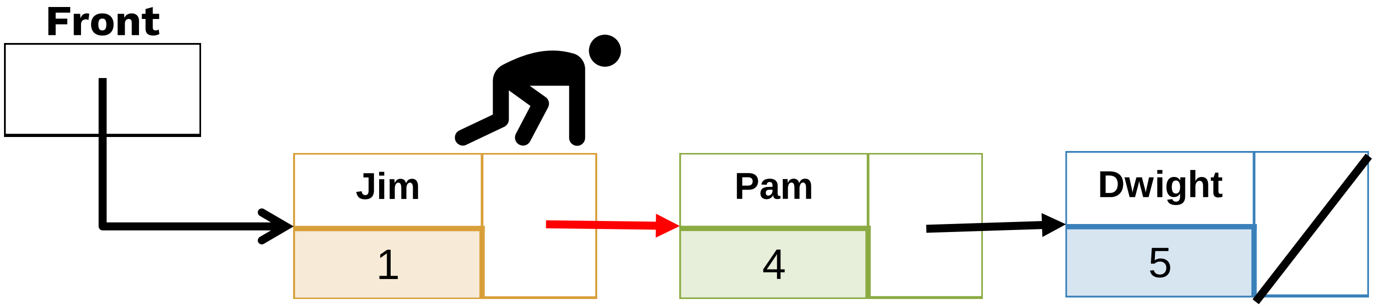
Reminders

- Last node should always be a **nullptr**!



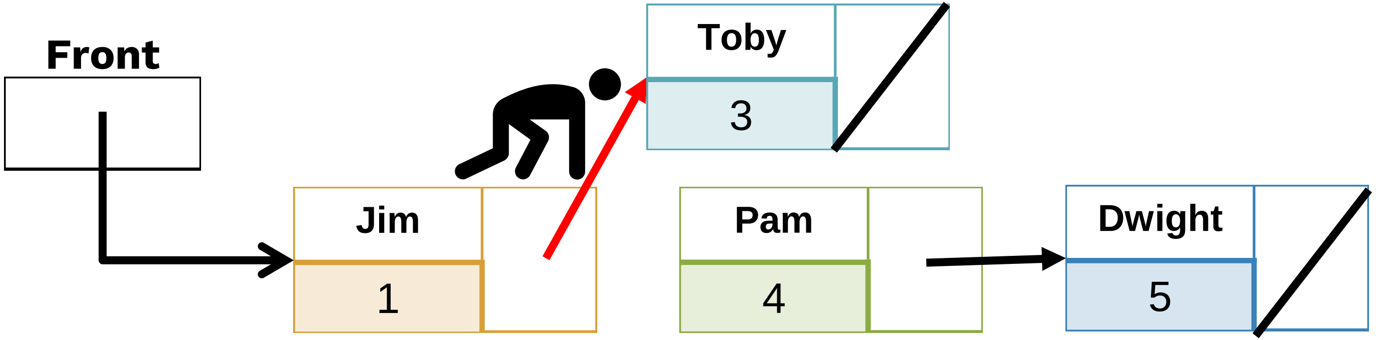
Reminders

- When adding or removing nodes, you should be working from the previous node.



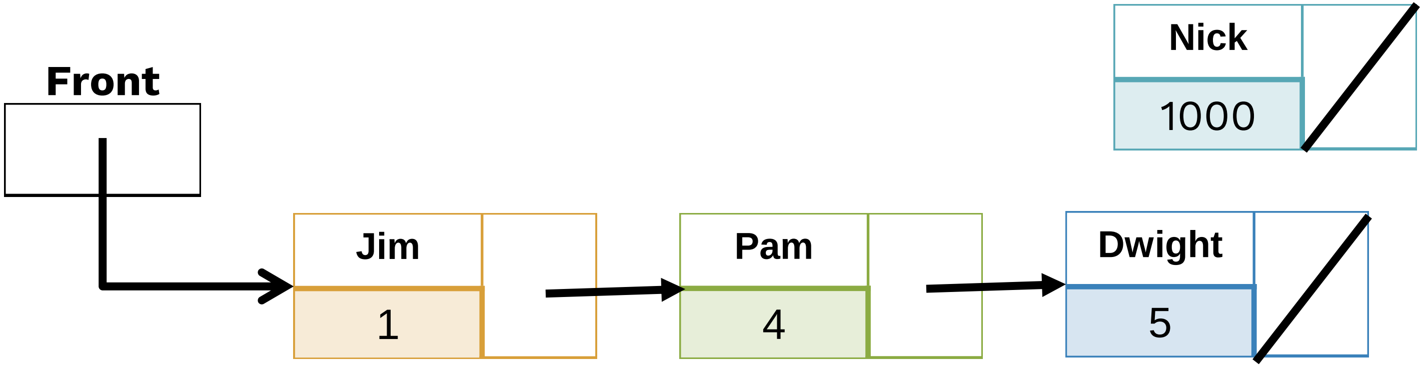
Reminders

- When adding or removing nodes, you should be working from the previous node.



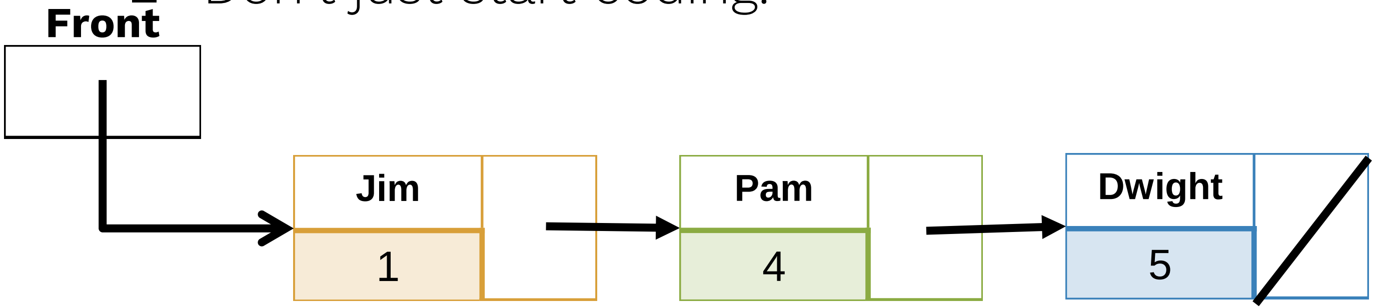
Reminders

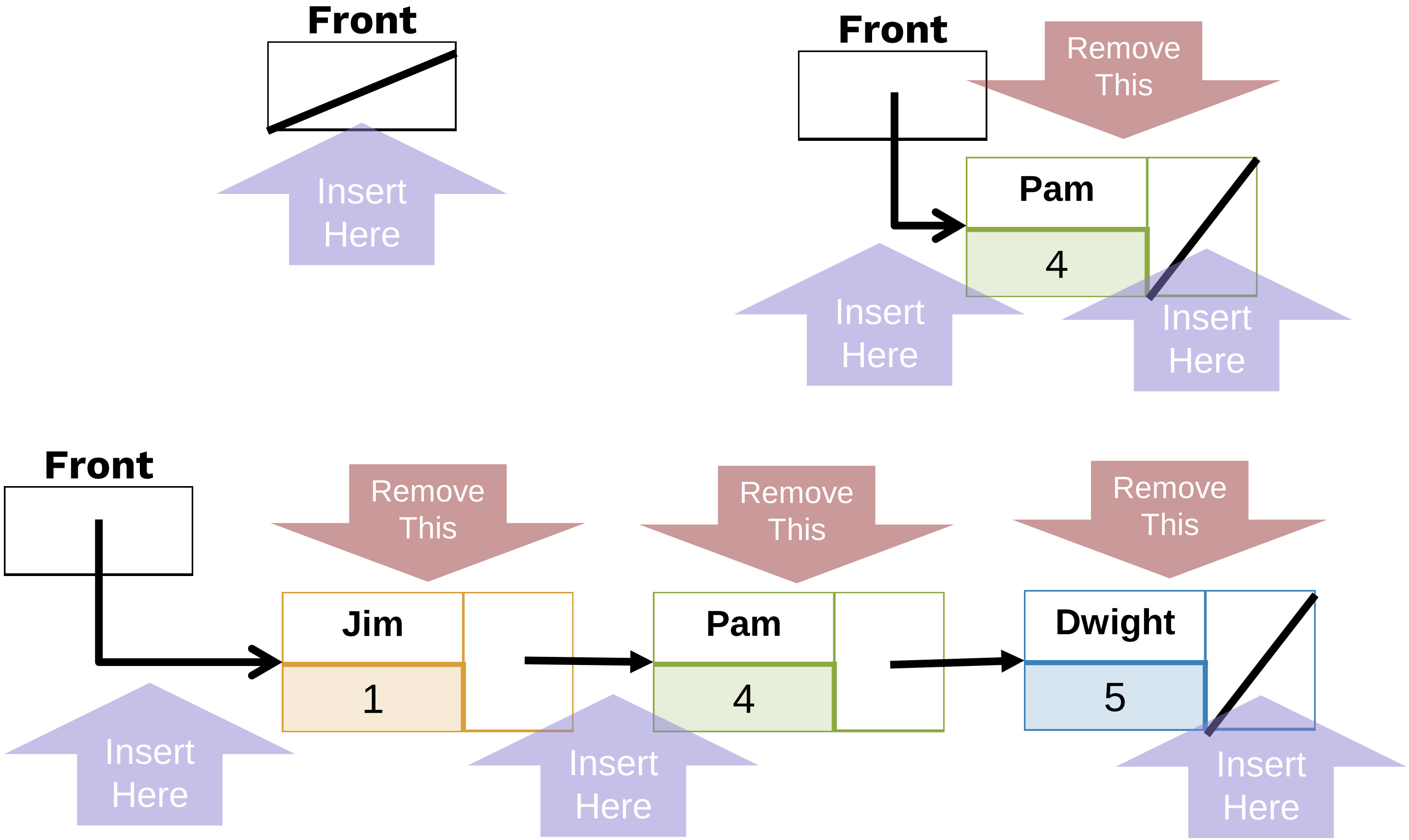
- Don’t create extra (or dummy) nodes.



More Tips

- After you’ve come up with your logic, draw baby examples (like this one) to see if it works.
- Don’t just start coding!





Front

Insert Here

Front

Remove This

Pam

4

Insert Here

Insert Here

Front

Remove This

Remove This

Remove This

Jim

1

Pam

4

Dwight

5

Insert Here

Insert Here

Insert Here

Questions

- Why don’t we need a timestamp?
- Is enqueuing or dequeuing faster?
- We don’t know the size. How do we know we’re at the end of a list?


KEEP
CALM
THEN
...
SEGFAULT

How to deal with seg faults?

- Did you do necessary checks `if (ptr == nullptr)`?
- Is a pointer still pointing to deleted garbage?
- Draw pictures! Stray arrows will speak for themselves.
- Come to LaIR, and we'll struggle together ☺

Questions to Ask

`PatientQueue()`	O(1)
`~PatientQueue()`	**O(N)**
`newPatient(name, priority)`	**O(N)**
`processPatient()`	O(1)
`frontName()`	O(1)
`frontPriority()`	O(1)
`upgradePatient(name, newP)`	**O(N)**
`isEmpty()`	O(1)
`clear()`	**O(N)**
`toString()`	**O(N)**

Note: the Big-O are different.

Use it to see if you are implementing it correctly!

Questions to Ask

`PatientQueue()`	`O(1)`
`~PatientQueue()`	`O(N)`
`newPatient(name, priority)`	**`O(N)`**
`processPatient()`	`O(1)`
`frontName()`	`O(1)`
`frontPriority()`	`O(1)`
`upgradePatient(name, newP)`	`O(N)`
`isEmpty()`	`O(1)`
`clear()`	`O(N)`
`toString()`	`O(N)`

Does my code take care of all cases (front, middle, back)?

What if this is the first patient?

Does my code take care of duplicates? Ties?

Questions to Ask

PatientQueue()	O(1)
~PatientQueue()	O(N)
newPatient(name, priority)	O(N)
processPatient()	**O(1)**
frontName()	O(1)
frontPriority()	O(1)
upgradePatient(name, newP)	O(N)
isEmpty()	O(1)
clear()	O(N)
toString()	O(N)

What if this is the last patient?

What if there are no patients left?

Questions to Ask

`PatientQueue()`	O(1)
`~PatientQueue()`	O(N)
`newPatient(name, priority)`	O(N)
`processPatient()`	O(1)
`frontName()`	O(1)
`frontPriority()`	O(1)
`upgradePatient(name, newP)`	**O(N)**
`isEmpty()`	O(1)
`clear()`	O(N)
`toString()`	O(N)

Does my code handle duplicates?

Is my code breaking ties correctly?

Do I make unnecessary passes (loops)?

Questions to Ask

`PatientQueue()`	O(1)
`~PatientQueue()`	O(N)
`newPatient(name, priority)`	O(N)
`processPatient()`	**O(1)**
`frontName()`	O(1)
`frontPriority()`	O(1)
`upgradePatient(name, newP)`	O(N)
`isEmpty()`	O(1)
`clear()`	**O(N)**
`toString()`	O(N)

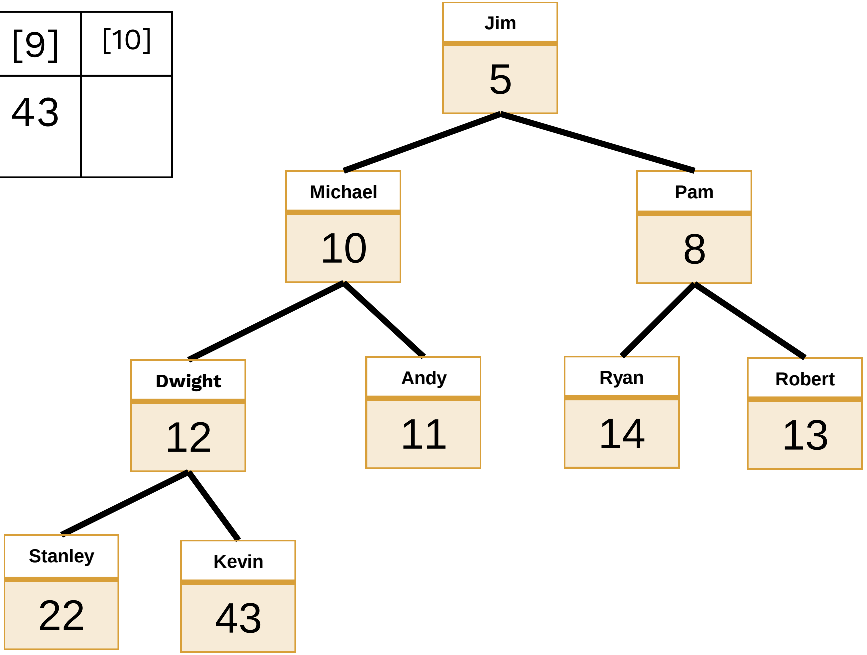
Am I freeing memory correctly?

Questions?

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	5	10	8	12	11	14	13	22	43	

Binary Heap

Fun with arrays and heaps!



Jim
5

Michael
10

Pam
8

Dwight
12

Andy
11

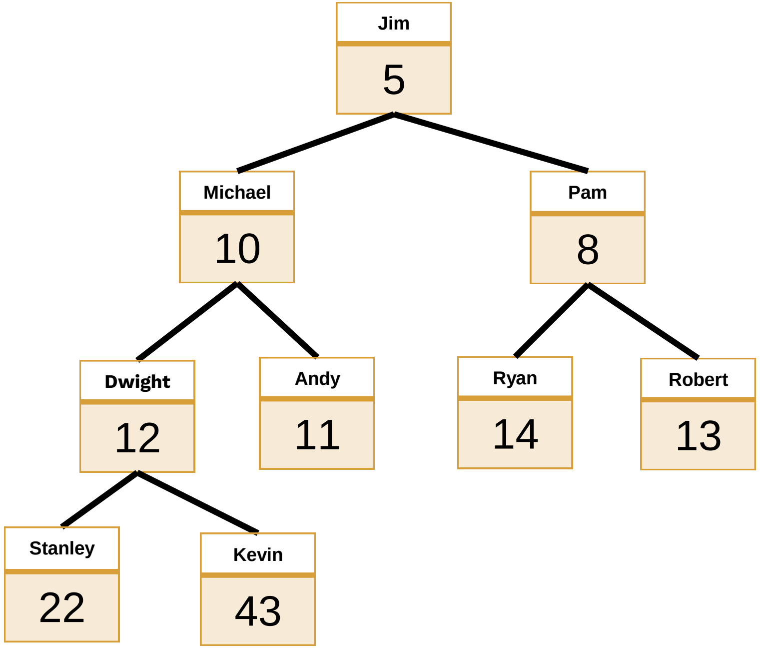
Ryan
14

Robert
13

Stanley
22

Kevin
43

What is a Heap?



- Tree-based structure
- Parents have higher priority than any of their children
- No implied ordering with sibilings

Summary: Binary Heap

`PatientQueue()`	O(1)
`~PatientQueue()`	O(1)
`newPatient(name, priority)`	O(log N)
`processPatient()`	O(log N)
`frontName()`	O(1)
`frontPriority()`	O(1)
`upgradePatient(name, newP)`	O(N)
`isEmpty()`	O(1)
`clear()`	O(1)
`toString()`	O(N)

Note: the Big-O are different.

Can you figure it out?

Summary: Binary Heap

`PatientQueue()`	**`O(1)`**
`~PatientQueue()`	`O(1)`
`newPatient(name, priority)`	`O(log N)`
`processPatient()`	`O(log N)`
`frontName()`	`O(1)`
`frontPriority()`	`O(1)`
`upgradePatient(name, newP)`	`O(N)`
`isEmpty()`	`O(1)`
`clear()`	`O(1)`
`toString()`	`O(N)`

- Only instance variable is size, capacity, and a **pointer to an internal array of elements**.
- Do not use a Vector!

Summary: Binary Heap

`PatientQueue()`	O(1)
`~PatientQueue()`	O(1)
`newPatient(name, priority)`	**O(log N)**
`processPatient()`	O(log N)
`frontName()`	O(1)
`frontPriority()`	O(1)
`upgradePatient(name, newP)`	O(N)
`isEmpty()`	O(1)
`clear()`	O(1)
`toString()`	O(N)

- When array is full, resize to larger array.
- See Wed lecture.

Summary: Binary Heap

`PatientQueue()`	`O(1)`
`~PatientQueue()`	`O(1)`
`newPatient(name, priority)`	**`O(log N)`**
`processPatient()`	**`O(log N)`**
`frontName()`	`O(1)`
`frontPriority()`	`O(1)`
`upgradePatient(name, newP)`	`O(N)`
`isEmpty()`	`O(1)`
`clear()`	`O(1)`
`toString()`	`O(N)`

- Are you bubbling up or down correctly?
- The log N runtime is very important!

Summary: Binary Heap

`PatientQueue()`	`O(1)`
`~PatientQueue()`	`O(1)`
`newPatient(name, priority)`	**`O(log N)`**
`processPatient()`	**`O(log N)`**
`frontName()`	`O(1)`
`frontPriority()`	`O(1)`
`upgradePatient(name, newP)`	**`O(N)`**
`isEmpty()`	`O(1)`
`clear()`	`O(1)`
`toString()`	`O(N)`

- When ties occur, use comparative operations (<, >, ==, !=).
- Only applies for the Heap!

Summary: Binary Heap

`PatientQueue()`	`O(1)`
`~PatientQueue()`	`O(1)`
`newPatient(name, priority)`	`O(log N)`
`processPatient()`	`O(log N)`
`frontName()`	`O(1)`
`frontPriority()`	`O(1)`
`upgradePatient(name, newP)`	**`O(N)`**
`isEmpty()`	`O(1)`
`clear()`	`O(1)`
`toString()`	**`O(N)`**

- Only time when you need to loop through the entire heap.

Questions?

Thanks!

Any questions?