Okay, human. Huh? Before you hit 'compile,' listen up.

You know when you're falling asleep, and you imagine yourself walking or something.

And suddenly you misstep, stumble, and jolt awake?

Yeah!

Well, that's what a segfault feels like. Double-check your damn pointers, okay?
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

```
ptr
```

```
1
0x10
```
Linked List

Front
0x20

Michael 0x50
1
0x20

Pam 0x40
4
0x50

Jim 0x10
4
0x40

Dwight NULL
5
0x10
Linked List

Front

1 -> 4 -> 4 -> 5

0x20 -> 0x50 -> 0x40 -> 0x10
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\} \rightarrow \{1, 3, 2, 7, 1, 3, 2, 7\}
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;
    }
    curr->next = front2;
}

Check if list is empty!
Set up “curr1”
void doubleList(ListNode *front){
    if (front == nullptr) return;
    ListNode *curr = front;
    ListNode *curr2 = new ListNode(curr->data);
    ListNode *front2 = curr2;
    while (curr1->next != nullptr){
        curr1 = curr1->next;
        curr2->next = new ListNode(curr1->data);
        curr2 = curr2->next;
    }
    curr1->next = front2;
}

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

Create new node,
Adjust all pointers.
```cpp
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;
}
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;
    }
    curr1->next = front2;
}

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
}

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
}
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”
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"
<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Colour</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Immediate resuscitation</td>
<td>Red</td>
<td>0 minutes</td>
</tr>
<tr>
<td>2</td>
<td>Very urgent</td>
<td>Orange</td>
<td>10 minutes</td>
</tr>
<tr>
<td>3</td>
<td>Urgent</td>
<td>Yellow</td>
<td>60 minutes</td>
</tr>
<tr>
<td>4</td>
<td>Standard</td>
<td>Green</td>
<td>120 minutes</td>
</tr>
<tr>
<td>5</td>
<td>Non-urgent</td>
<td>Blue</td>
<td>240 minutes</td>
</tr>
</tbody>
</table>
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"
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**
- VectorPatientQueue.h
- LinkedListPatientQueue.h
- HeapPatientQueue.h

**CPP Files**
- VectorPatientQueue.cpp
- LinkedListPatientQueue.cpp
- HeapPatientQueue.cpp

**Don’t Edit (unless extensions)**
- patientnode.h
- patientqueue.h
- hospital.cpp
- patientnode.cpp
CPP Files

- All three are nearly identical.
- Same public methods to implement.
- Do not change method headers!
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**
Create your own struct.
Store elements in *unsorted* order in a Vector of structs.

Maintain Vector.

**Sorted Linked List**
Use provided struct.
Store elements in sorted order using a linked list.

Maintain “front” pointer.

**Binary Heap**
Create your own struct.
Organized in a heap (stored as an array of structs).

Maintain array.
An unsorted vector:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
<td>Dwight</td>
<td>Jim</td>
<td>Michael</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Simple but slow implementation.
Vector Implementation

Empty Vector
Vector Implementation

<table>
<thead>
<tr>
<th>v[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Add patient Pam, priority 4
**Vector Implementation**

<table>
<thead>
<tr>
<th></th>
<th>v[0]</th>
<th>v[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
<td>4</td>
<td>Dwight</td>
</tr>
</tbody>
</table>

Add patient Dwight, priority 5
## Vector Implementation

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
<td>4</td>
<td>Dwight</td>
<td>5</td>
</tr>
<tr>
<td>Jim</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add patient Jim, priority 4
Vector Implementation

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dwight</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jim</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michael</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add patient Michael, priority 1
Vector Implementation

Upgrade Jim to priority 1
Vector Implementation

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
<td>Dwight</td>
<td>Jim</td>
<td>Michael</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dwight</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jim</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michael</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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!

<table>
<thead>
<tr>
<th>Name</th>
<th>Pam</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>4</td>
</tr>
<tr>
<td>Timestamp</td>
<td>1</td>
</tr>
</tbody>
</table>
Summary: Vector

<table>
<thead>
<tr>
<th>Function</th>
<th>Big-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(1)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(N)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(N)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(N)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(1)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

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

Don’t overthink it!
## Summary: Vector

<table>
<thead>
<tr>
<th>Function</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>processPatient()</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>frontName()</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>clear()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>toString()</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

**Vector is unsorted!**

Must loop over entire vector to find patient with minimum priority.
### Summary: Vector

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>processPatient()</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>frontName()</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>clear()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>toString()</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

**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 PatientNode {
    string name;
    int priority;
    PatientNode* next;
};
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.

Front

<table>
<thead>
<tr>
<th>Michael</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pam</td>
<td>4</td>
</tr>
<tr>
<td>Jim</td>
<td>4</td>
</tr>
<tr>
<td>Dwight</td>
<td>5</td>
</tr>
</tbody>
</table>
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!
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!
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Big-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(N)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(N)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(1)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(N)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Note: the Big-O are different. Use it to see if you are implementing it correctly!
Questions to Ask

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(N)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(N)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(1)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(N)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(N)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(N)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(1)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(N)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

What if this is the last patient?

What if there are no patients left?
Questions to Ask

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(N)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(N)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(1)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(N)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

- Does my code handle duplicates?
- Is my code breaking ties correctly?
- Do I make unnecessary passes (loops)?
Questions to Ask

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(N)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(N)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(1)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(N)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Am I freeing memory correctly?
Questions?
Binary Heap

Fun with arrays and heaps!
What is a Heap?

- Tree-based structure
- Parents have higher priority than any of their children
- No implied ordering with siblings
**Summary: Binary Heap**

<table>
<thead>
<tr>
<th>Method</th>
<th>Big-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>processPatient()</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>frontName()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>clear()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>toString()</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

Note: the Big-O are different.

Can you figure it out?
Summary: Binary Heap

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(log N)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(log N)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(1)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(1)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

- Only instance variable is size, capacity, and a pointer to an internal array of elements.
- Do not use a Vector!
### Summary: Binary Heap

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(log N)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(log N)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(1)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(1)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

- When array is full, resize to larger array.
- See Wed lecture.
Summary: Binary Heap

<table>
<thead>
<tr>
<th>Function</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(log N)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(log N)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(1)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(1)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>~PatientQueue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>newPatient(name, priority)</td>
<td>O(log N)</td>
</tr>
<tr>
<td>processPatient()</td>
<td>O(log N)</td>
</tr>
<tr>
<td>frontName()</td>
<td>O(1)</td>
</tr>
<tr>
<td>frontPriority()</td>
<td>O(1)</td>
</tr>
<tr>
<td>upgradePatient(name, newP)</td>
<td>O(N)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>clear()</td>
<td>O(1)</td>
</tr>
<tr>
<td>toString()</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

- Only time when you need to loop through the entire heap.
Questions?
Thanks!

Any questions?