YEAH - Patient Queue

Jason Chen

Original slides by: Anton Apostolatos

prev->next = toDelete->next;
delete toDelete;
// if only forgetting were
// this easy for me.
SNIFF
assert "It's going to be okay.";

Source: XKCD

Queue: order items by when they were placed - first in, first out (*FIFO*)

Main Functions

```
void enqueue(string s)   // Inserts an element into the queue
string dequeue()         // Returns and removes the first element placed
```

PriorityQueue: order items by **priority**

Main Functions

```
void enqueue(string s, int priority)   // Puts element into priority queue
string dequeue()                 // Returns and removes the highest-priority item
```

PatientQueue: order **patients** by **priority**

Main Functions

```
void newPatient(string name, int priority) // Puts person into patient queue
string processPatient()      // Returns and removes the highest-priority person
```

Lower number = higher priority!

Note: time in queue is the tiebreaker

(“Merry” : 1) < (“Frodo” : 3) < (“Pippin” : 3) < (“Sam” : 6)

```
PatientQueue pq;

pq.newPatient(“Sam”, 6);
pq.newPatient(“Frodo”,   3);
pq.newPatient(“Pippin”,  3);
pq.newPatient(“Merry”, 1);

cout << pq.processPatient() << endl;
cout << pq.processPatient() << endl;
```

Console

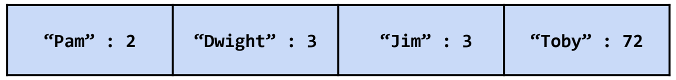
```
Merry
Frodo
```

Main Functions

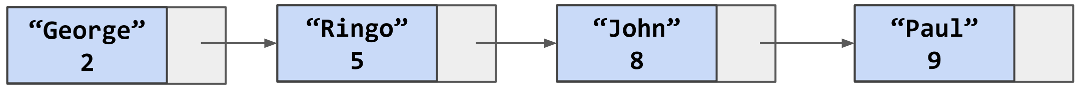
```
PatientQueue()                  // Constructor for PatientQueue

~PatientQueue()                 // Destructor for PatientQueue

void newPatient(string name, int priority) // Puts element into patient queue

string processPatient()         // Returns and removes the highest-priority item

string frontName()              // Name of highest-priority patient

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

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

void clear()                    // Removes all patients

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

A5: Patient Queue

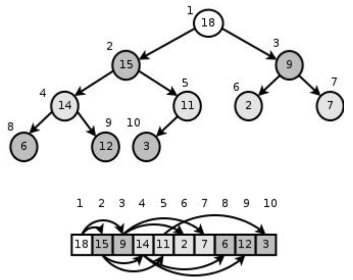
Unsorted Vector



Sorted Singly-Linked List



Extension: Binary Heap

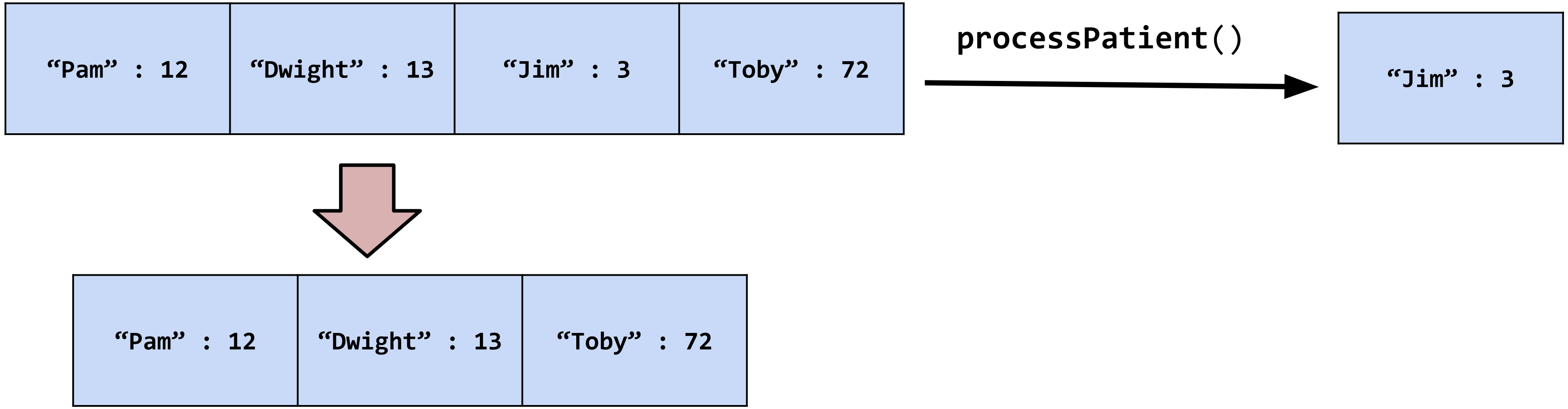


Unsorted Vector

“Pam” : 12	“Dwight” : 13	“Jim” : 3	“Toby” : 72

Unsorted and `Vector` wrapper - Simplest to implement and think about!

`newPatient(string name, int priority)`: append to a vector!
`processPatient()/front()`: scan vector and find smallest element



A vector of what?!

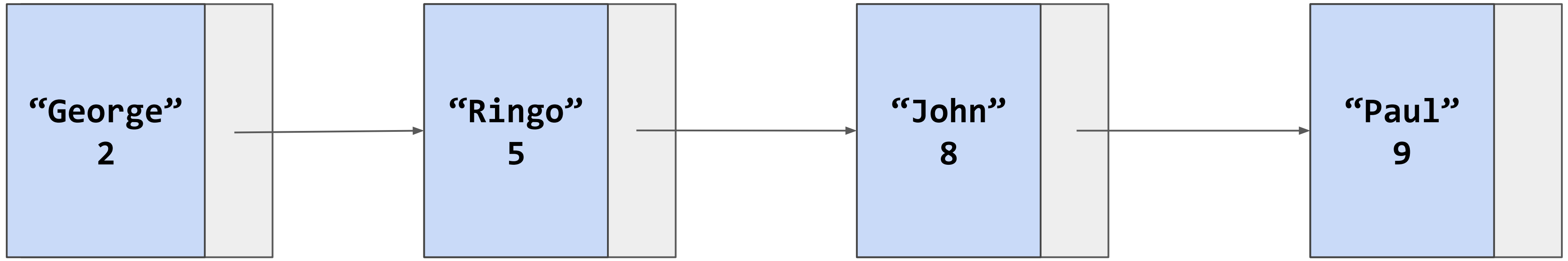
Struct!

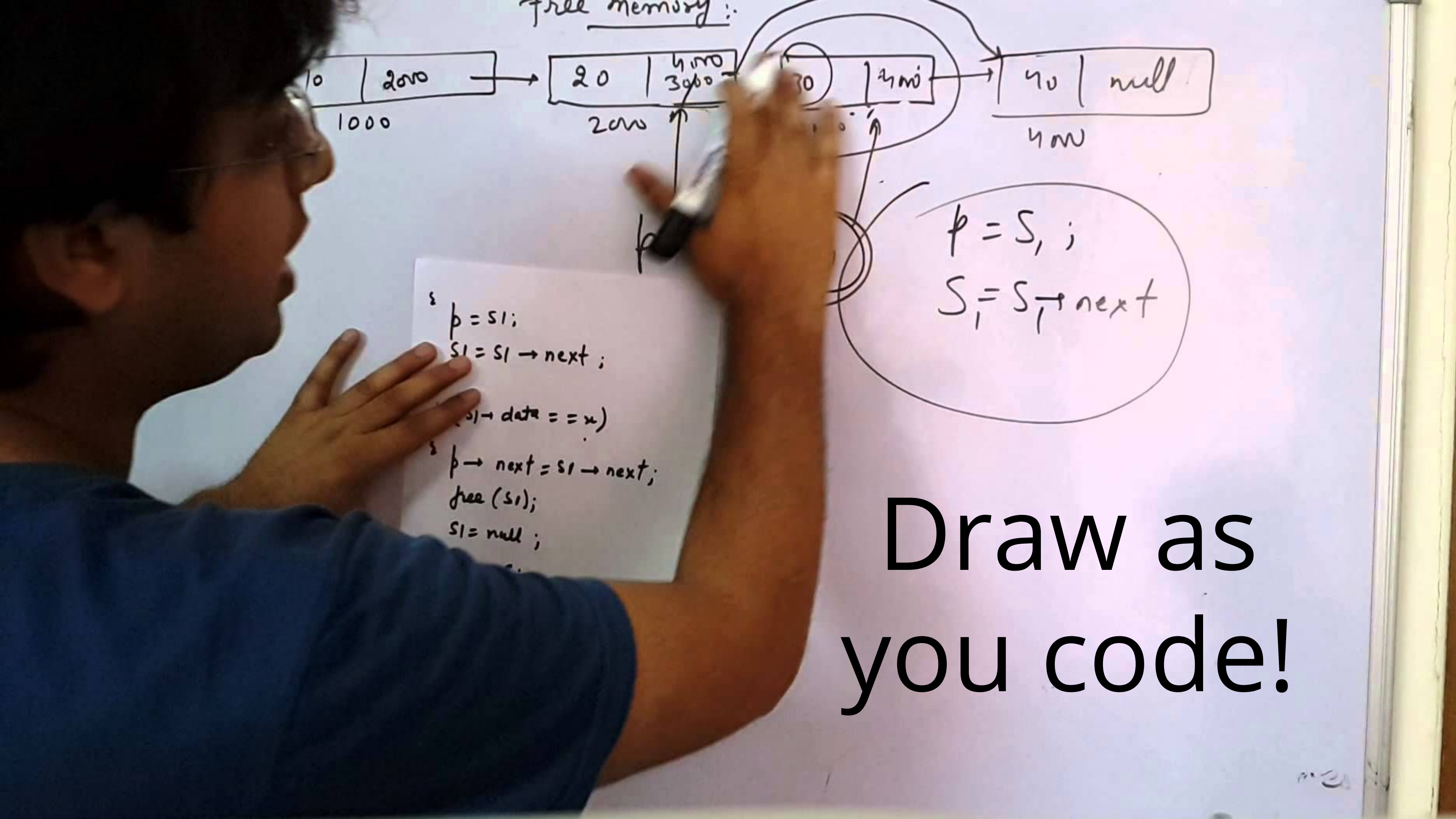
What does the struct store?

Up to you!

Questions?

Sorted Singly-Linked List





```
p = s1;
s1 = s1 → next ;

(s1 → data = = x)

p → next = s1 → next;
free (s1);
s1 = null ;
```

Draw as you code!

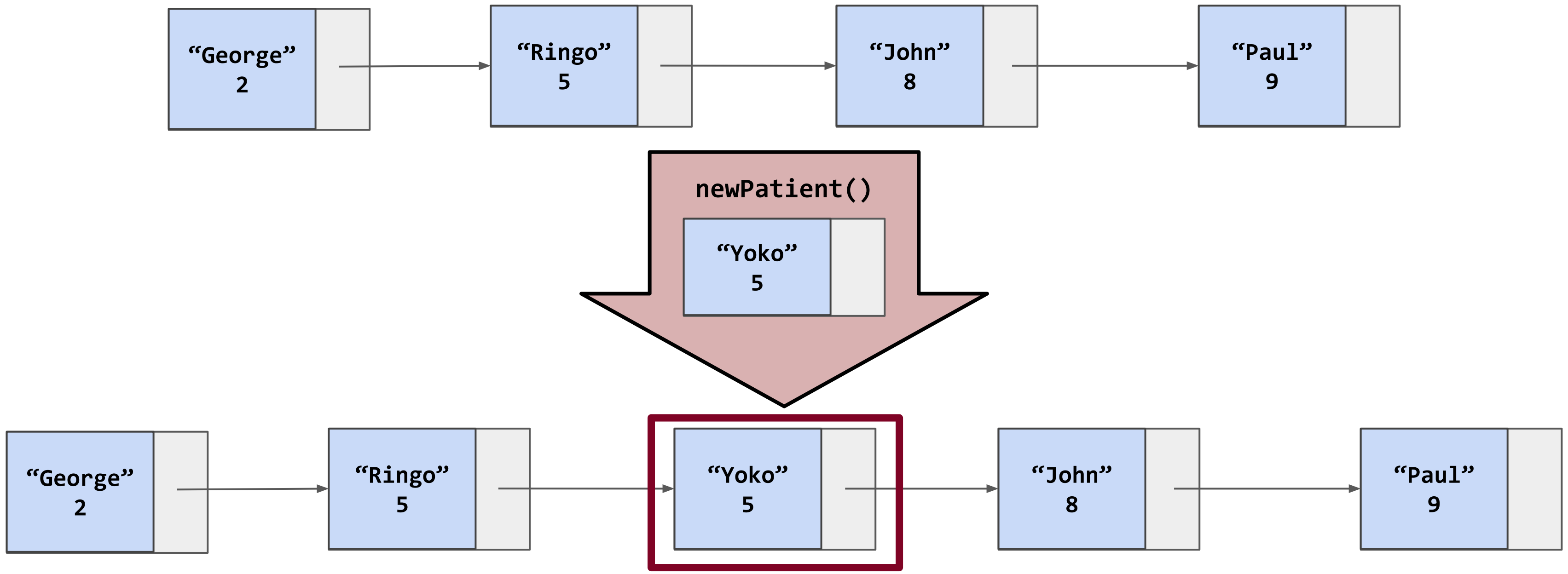
PatientNode Struct

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

     // Constructor - each parameter is optional
     PatientNode(string name, int priority, PatientNode* next);
}
```

You need to create a Linked List and enforce that all elements are stored in order of priority, with time in queue being tiebreaker

`newPatient()`: look for place in the linked list and place there

`processPatient()/front()`: first element!



Tips and Tricks

- Before writing any code, go through simple toy examples by hand to make sure your proposed solution’s logic is sound. Always think about edge cases like if the your patient queue was empty, had one element, or if you were inserting at the very front or end.

- Don’t forget the semicolon after a struct or class definition!

- Bad idea to declare multiple pointers on the same line:

```
Node * head, tail;
```

Tips and Tricks: Continued

- Nested structs are weird. This probably won’t come up, but if you create a cell inside of PQueue then a helper function that returns a PatientNode* would be *declared* as:

```
PatientNode* helperFunction(PatientNode* ptr);
```

- And would be *implemented* as

```
PQueue::PatientNode* PQueue::helperFunction(PatientNode* ptr);
```

- Do your best to make your size functions not O(n)! → how?

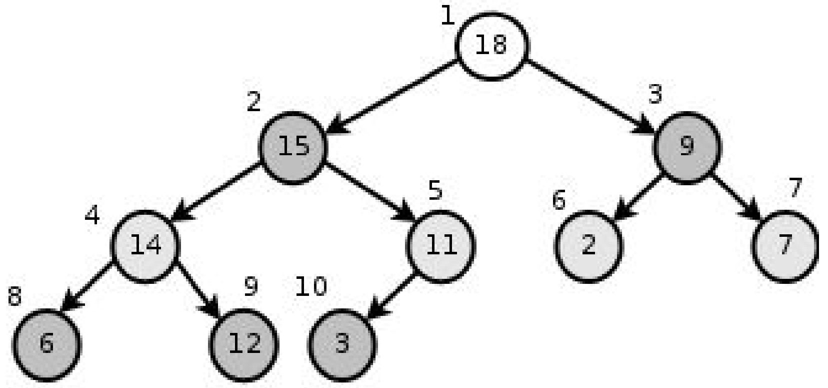
- **We’ll ask you for Big-O of every function you write!**

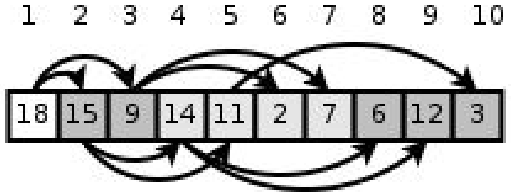
Main Functions

```
PatientQueue()                 // Constructor for PatientQueue

~PatientQueue()                // Destructor for PatientQueue

void newPatient(string name, int priority) // Puts element into patient queue

string processPatient()        // Returns and removes the highest-priority item

string frontName()             // Name of highest-priority patient

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

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

void clear()                   // Removes all patients

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

Questions?

Extension: Binary Heap





Binary Heaps

A heap is a *tree-based* structure that satisfies the heap property:

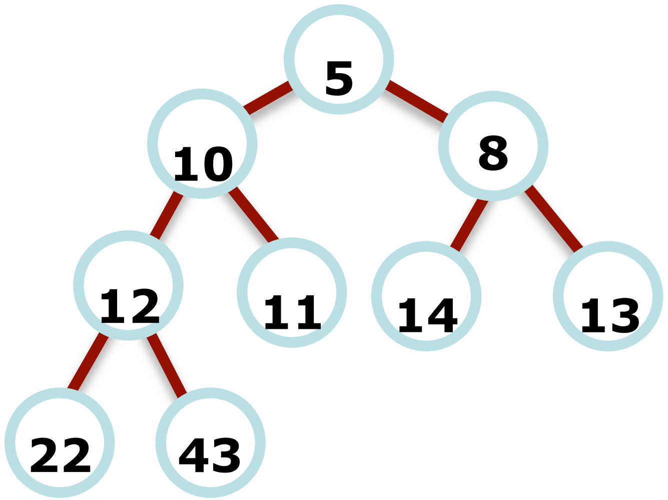
Parents have a higher priority than any of their children.

Binary Heaps

- There are two types of heaps:

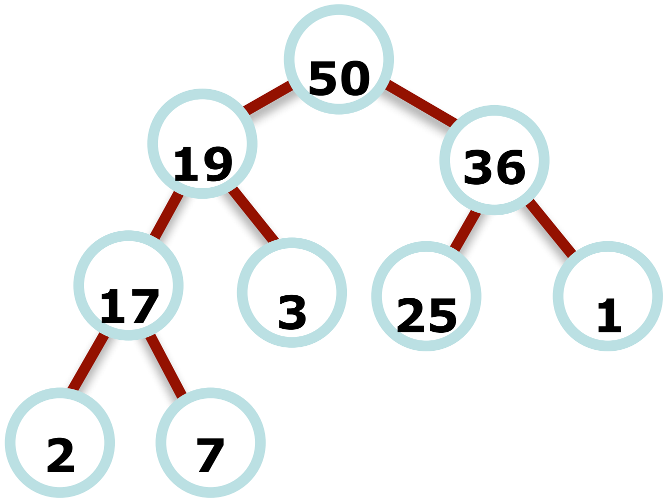
Min Heap

(root is the smallest element)



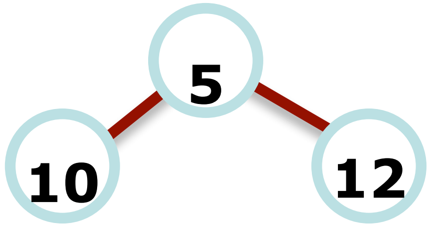
Max Heap

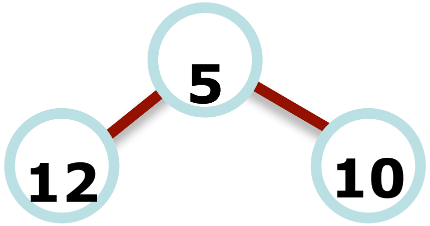
(root is the largest element)



Binary Heaps

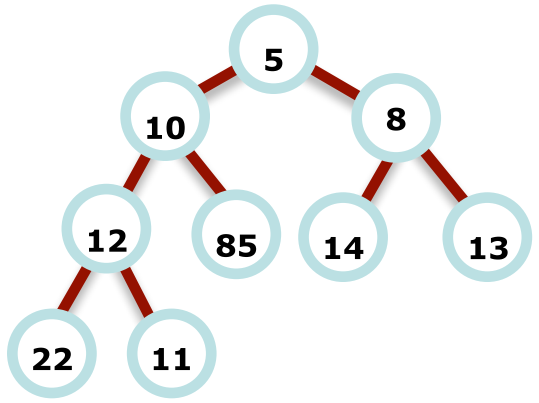
- There are no implied orderings between siblings, so both of the trees below are min-heaps:

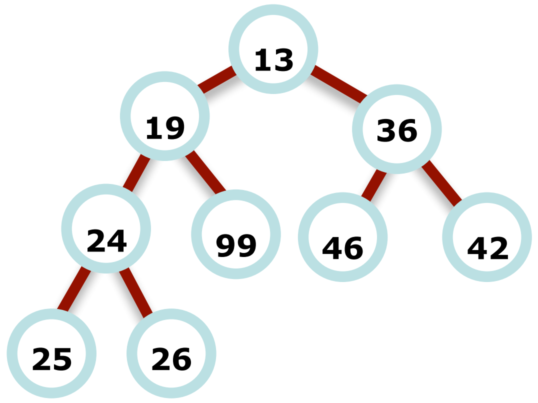




Binary Heaps

- Circle the min-heap(s):





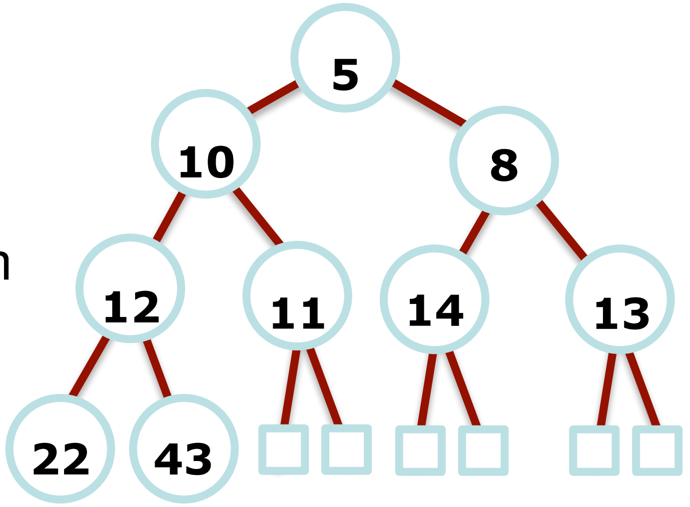
Binary Heaps

Heaps are completely filled, with the exception of the bottom level. They are, therefore, "**complete binary trees**":

complete: all levels filled except the bottom

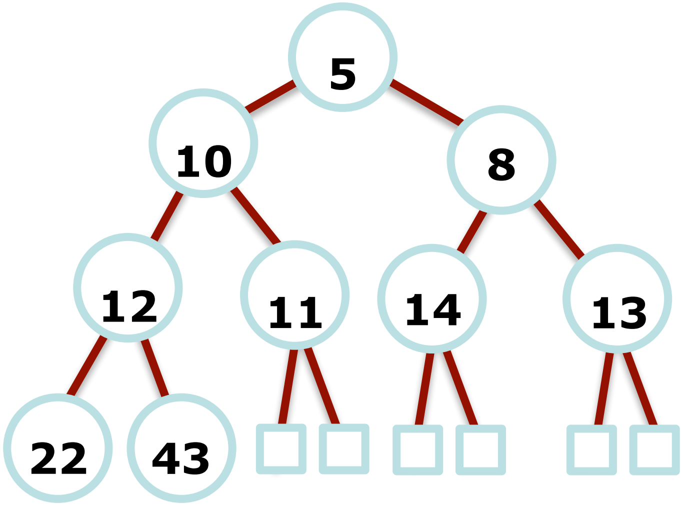
binary: two children per node (parent)

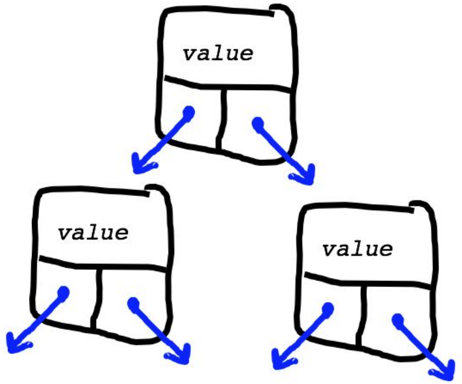
height? log(n)



Binary Heaps

What is the best way to store a heap?



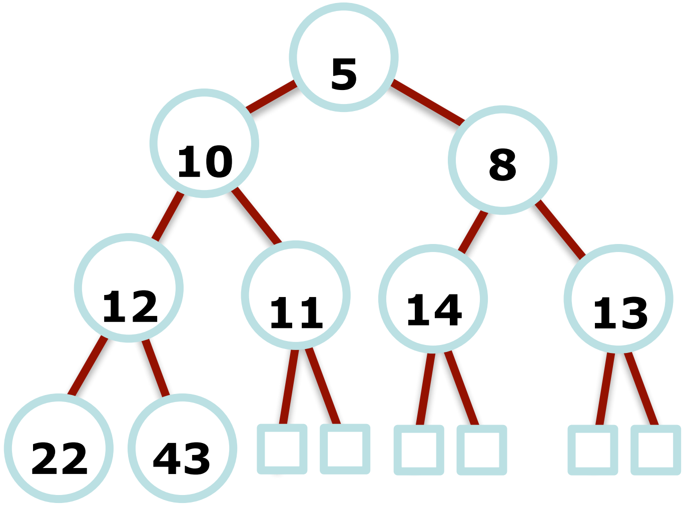


We could use a node-based solution, but…

Binary Heaps

It turns out that an array works **great** for storing a binary heap!

We will put the root at index 1 instead of index 0 (this makes the math work out just a bit nicer).



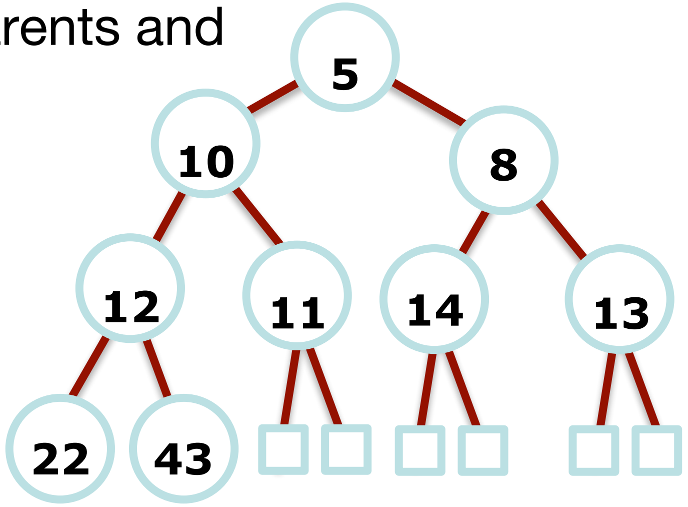
	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Binary Heaps

The array representation makes determining parents and children a matter of simple arithmetic:

For an element at position ***i***:

- left child is at **2*i***
- right child is at **2*i*+1**
- parent is at ***⌊i/2⌋***



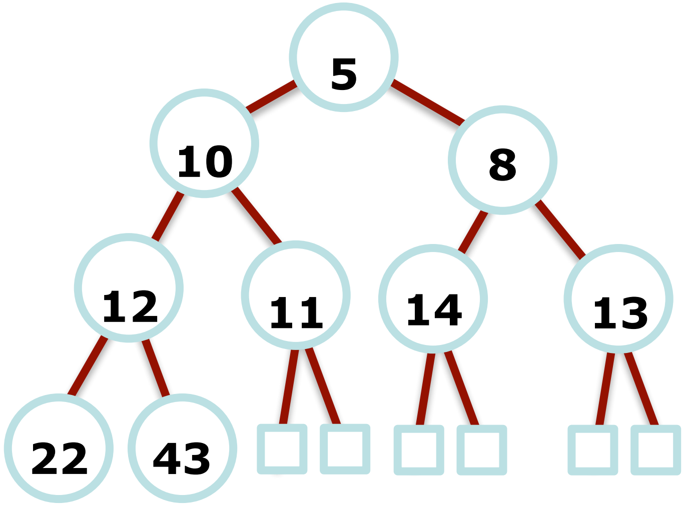
	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operations

Remember that there are three important priority queue operations:

- **`peek()`**: return an element of h with the smallest key.
- **`enqueue(e)`**: insert element e into the heap.
- **`dequeueMin()`**: removes the smallest element from h.

We can accomplish this with a heap!

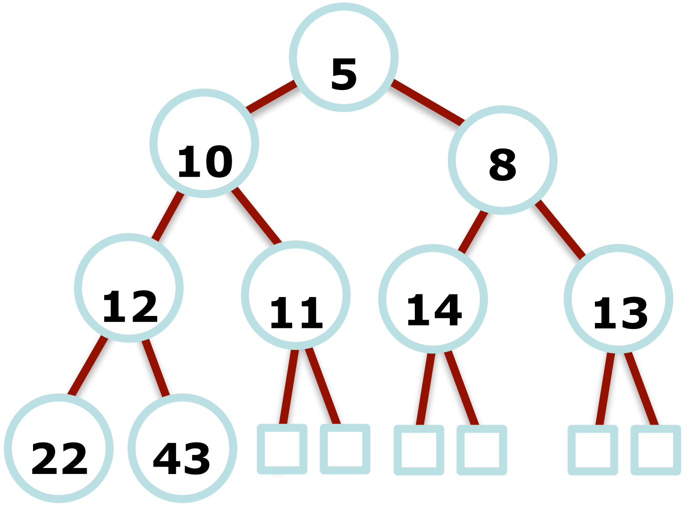


Heap Operations: peek()

```
peek()
```

Just return the root!

```
return heap[1]
```

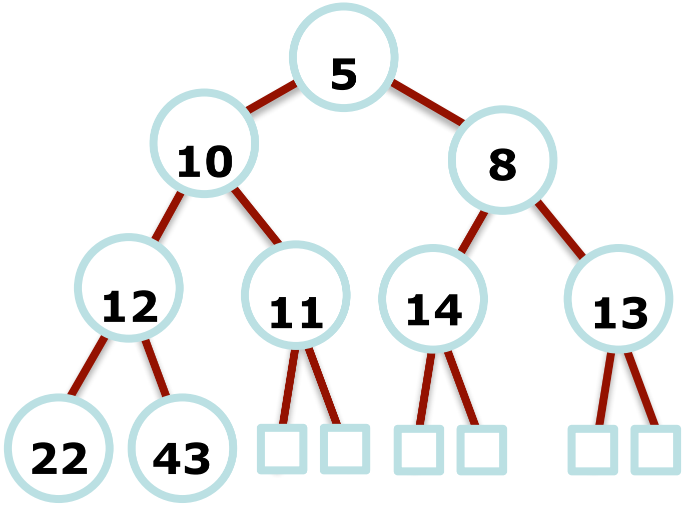


	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operations: enqueue(k)

`enqueue(k)`

How might we go about inserting into a binary heap?



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

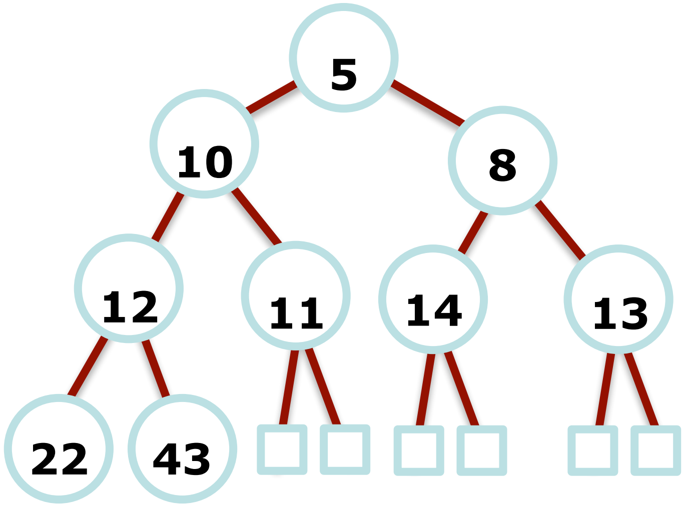
Heap Operations: enqueue(k)

Heap Operations: **`enqueue(k)`**

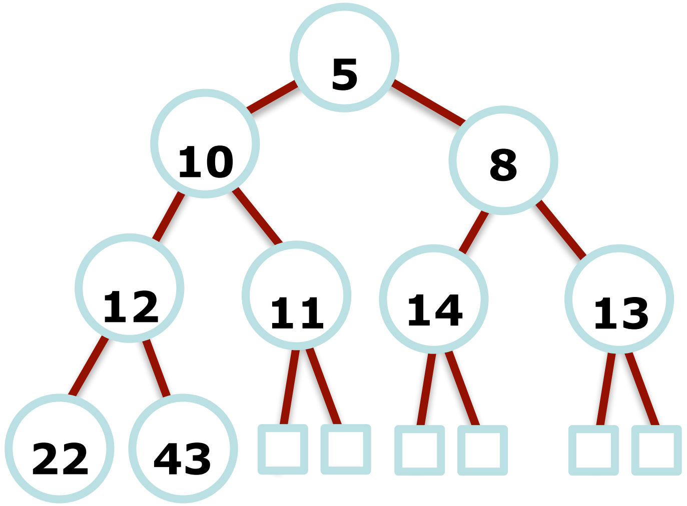
Insert item at element **`array[heap.size()+1]`**
(this probably destroys the heap property)

Perform a “**bubble up**” operation:

- Compare the added element with its parent
    - if in correct order, stop
    - If not, swap and repeat



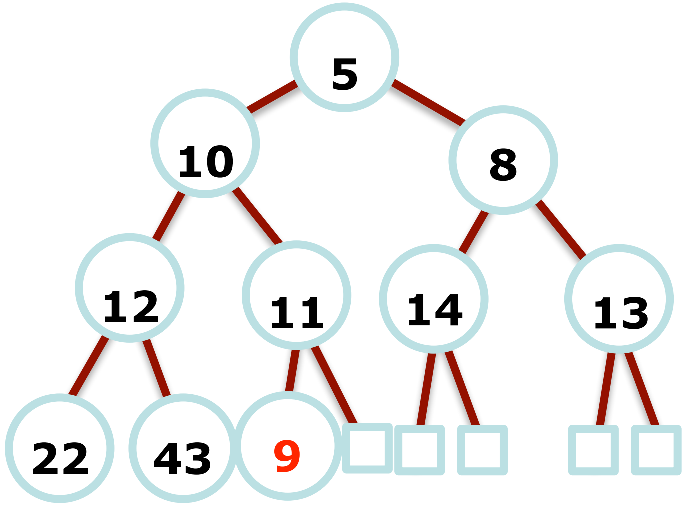
Heap Operations: enqueue(9)



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

Start by inserting the key at the first empty position. This is always at index **`heap.size()+1`**.

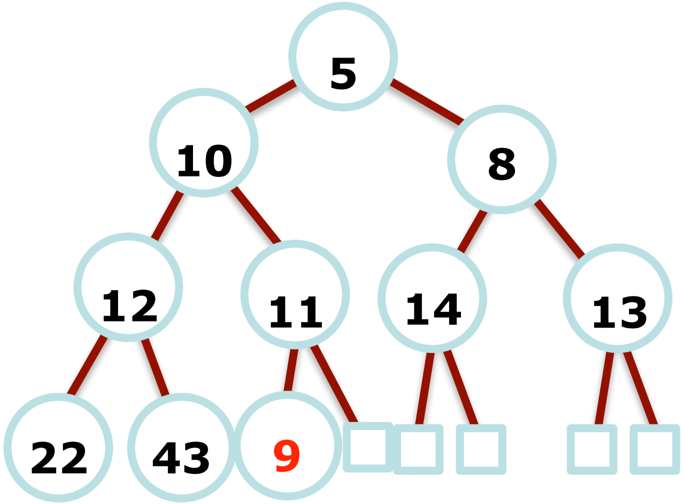
Heap Operations: enqueue(9)



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

Start by inserting the key at the first empty position. This is always at index `heap.size()+1`.

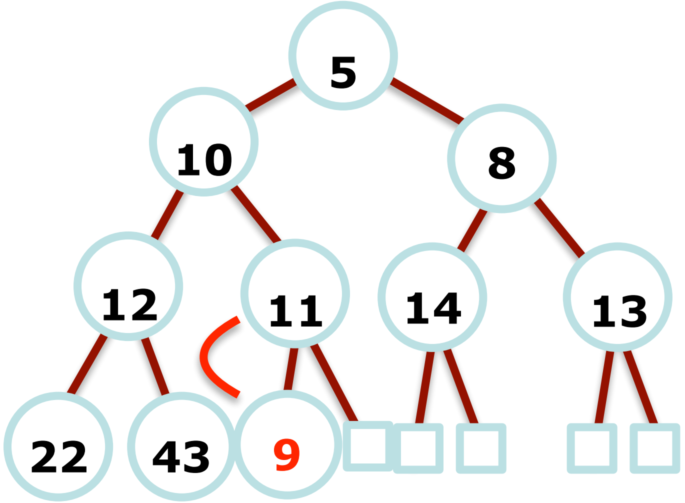
Heap Operations: enqueue(9)

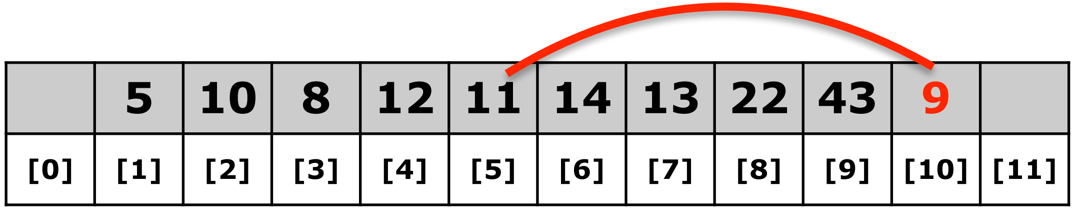


	5	10	8	12	11	14	13	22	43	9	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Look at parent of index 10, and compare: do we meet the heap property requirement?

Heap Operations: enqueue(9)



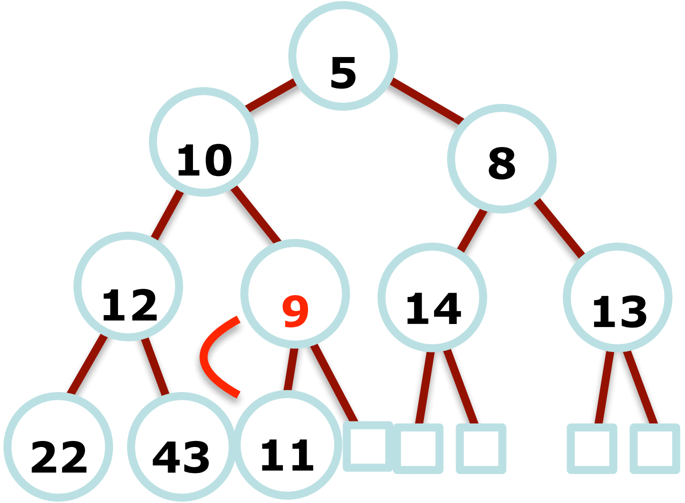


	5	10	8	12	11	14	13	22	43	9	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Look at parent of index 10, and compare: do we meet the heap property requirement?

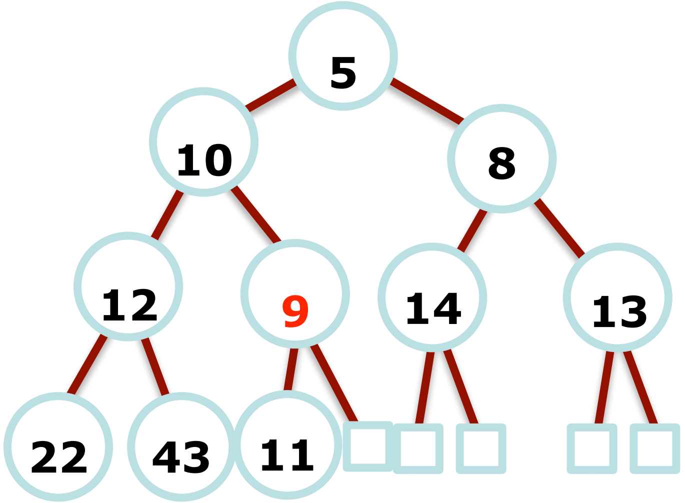
No -- we must swap.

Heap Operations: enqueue(9)



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

Heap Operations: enqueue(9)

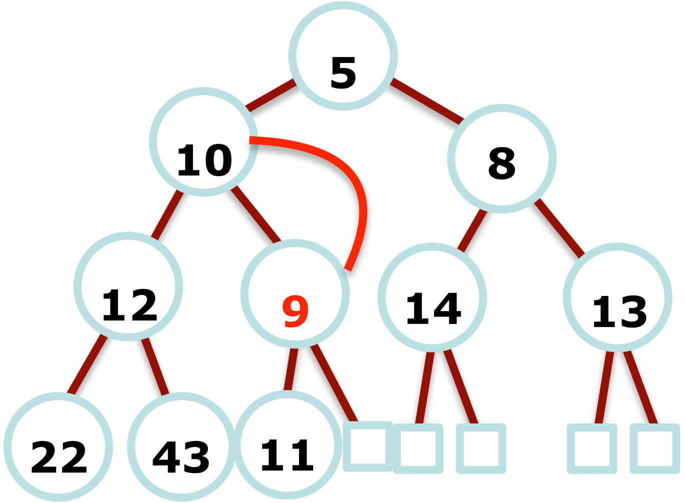


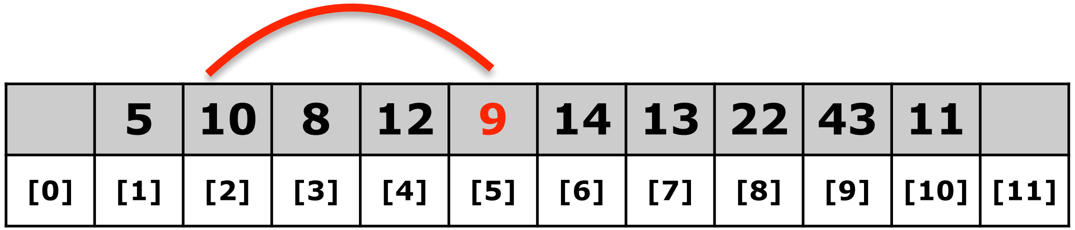
	5	10	8	12	9	14	13	22	43	11	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Look at parent of index 5, and compare: do we meet the heap property requirement?

No -- we must swap.

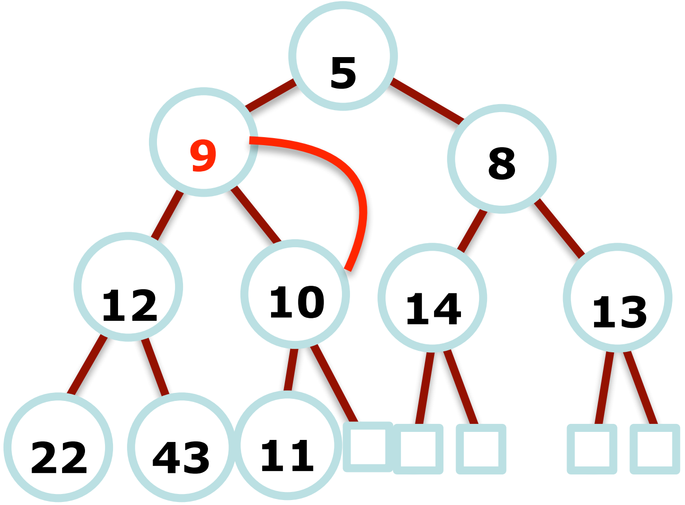
Heap Operations: enqueue(9)

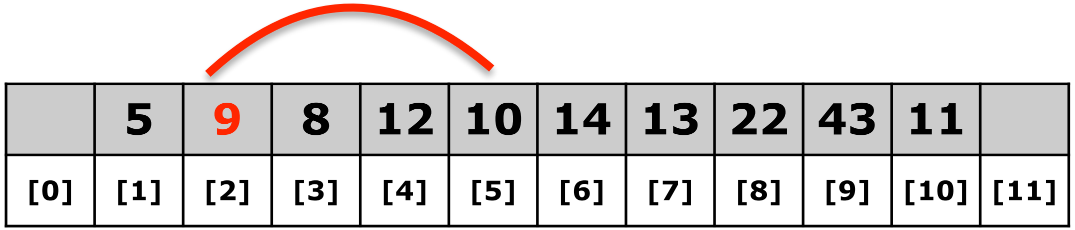




	5	10	8	12	9	14	13	22	43	11	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operations: enqueue(9)

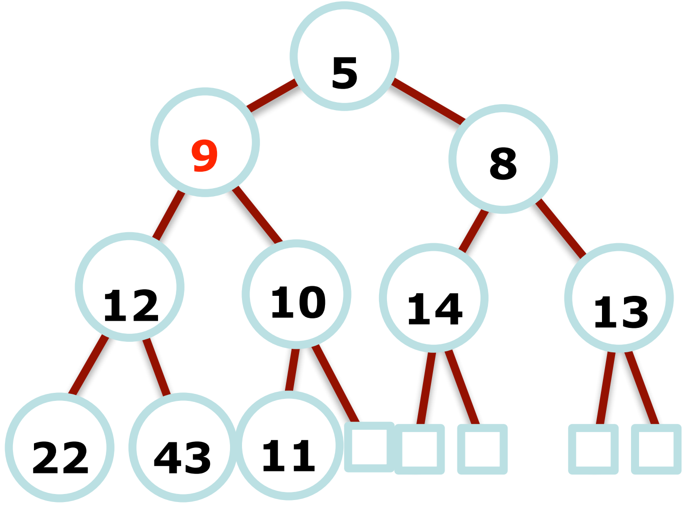




	5	9	8	12	10	14	13	22	43	11	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operations: enqueue(9)

No swap necessary between index 2 and its parent.
We're done bubbling up!



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

Demo!

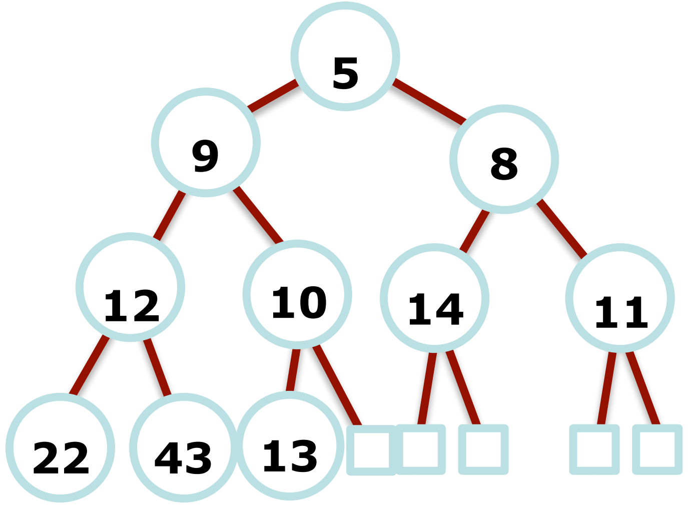
http://www.cs.usfca.edu/~galles/visualization/Heap.html

Heap Operations: dequeue()

- How might we go about removing the minimum?

```
        dequeue()
```

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

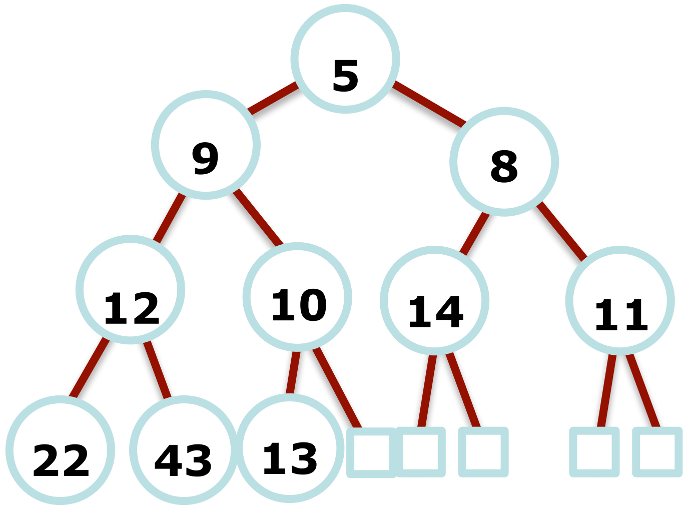


Heap Operations: dequeue()

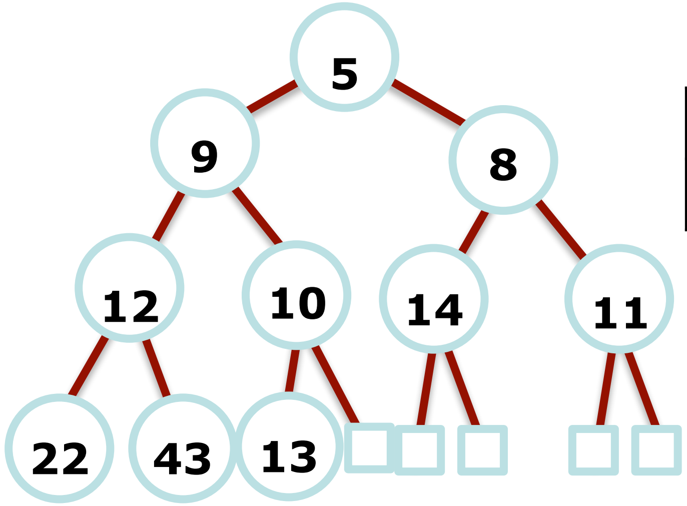
We are removing the root, and we need to retain a complete tree: replace root with last element.

"**bubble-down**" or "down-heap" the new root:

- Compare the root with its children:
    - if in correct order, stop.
    - if not, swap with smallest child, and repeat



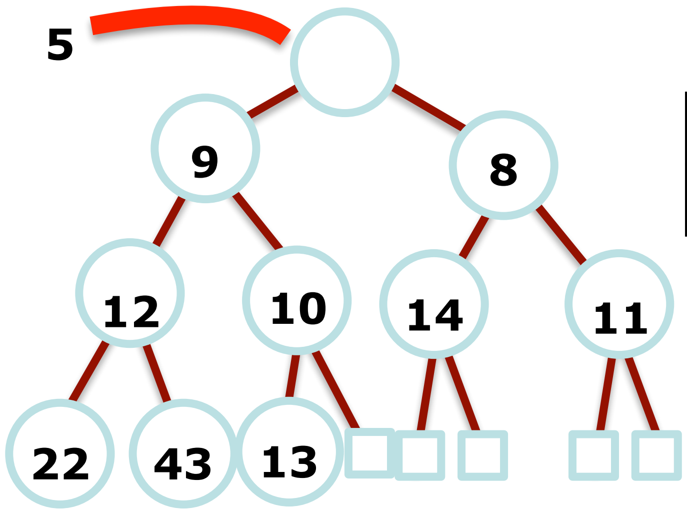
Heap Operations: dequeue()



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

Heap Operations: dequeue()

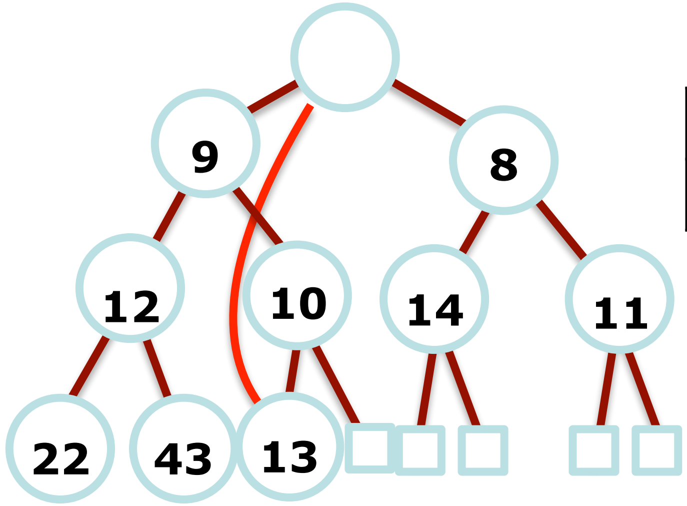
Remove root (will return at the end)

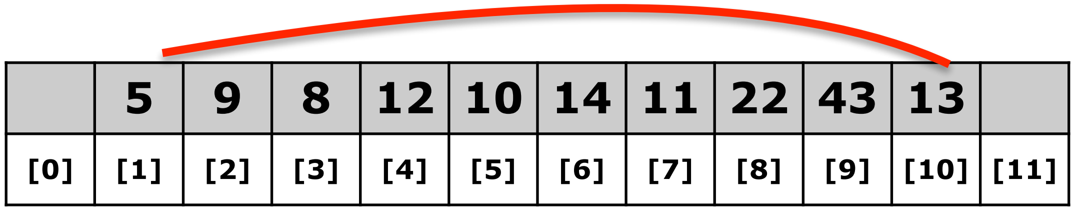


	5	9	8	12	10	14	11	22	43	13	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operations: dequeue()

Move last element (at `heap[heap.size()]`) to the root (this may be unintuitive!) to begin bubble-down

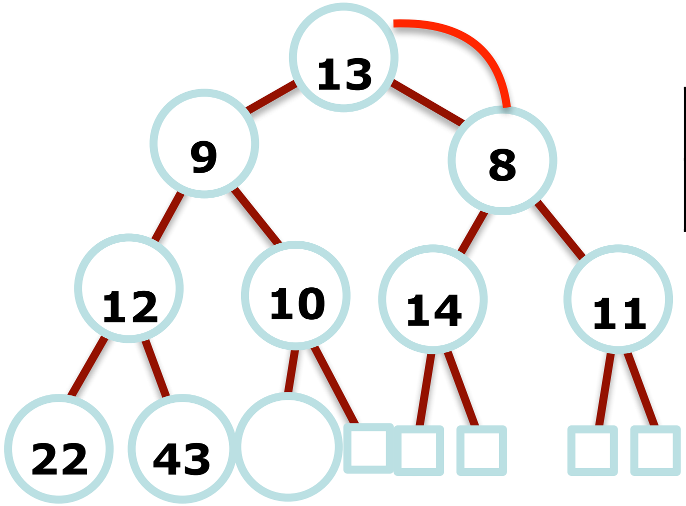




	5	9	8	12	10	14	11	22	43	13	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operations: dequeue()

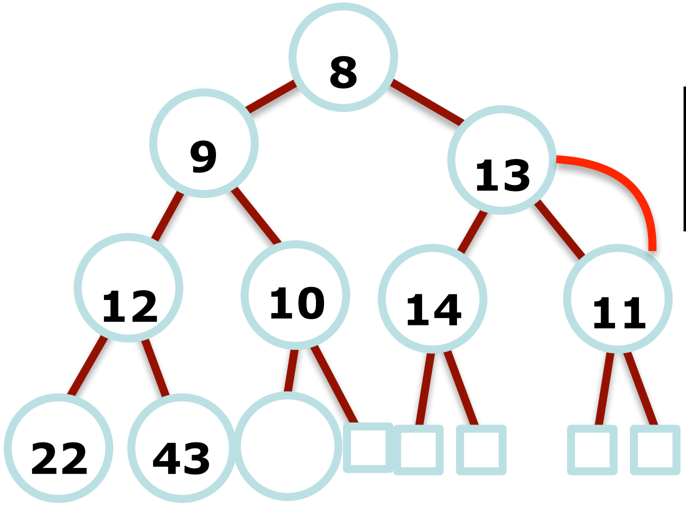
Compare children of root with root: swap root with the smaller one (why?)

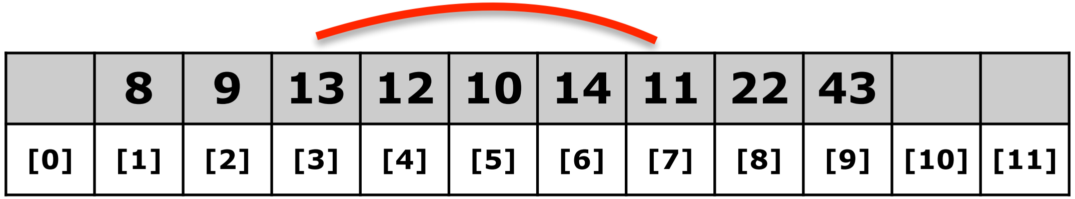


	13	9	8	12	10	14	11	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operations: dequeue()

Keep swapping new element if necessary. In this case: compare 13 to 11 and 14, and swap with smallest (11).

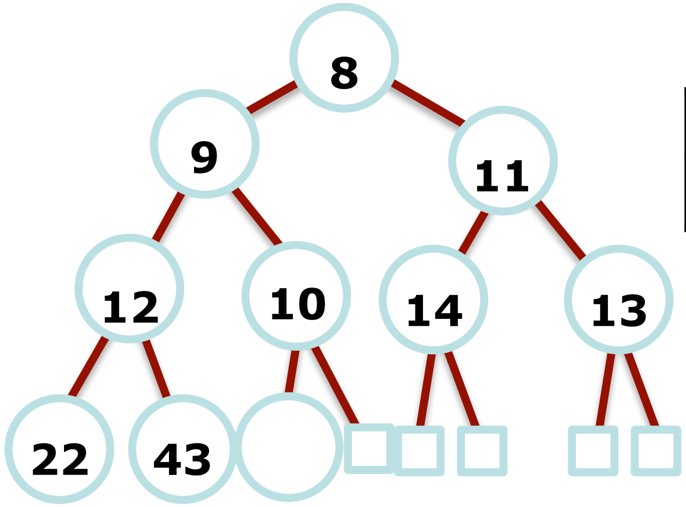




	8	9	13	12	10	14	11	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operations: dequeue()

13 has now bubbled down until it has no more children, so we are done!



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

Questions?