

CS 106B

Lecture 18: Binary Heaps

Monday, July 31, 2017

Programming Abstractions
Summer 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

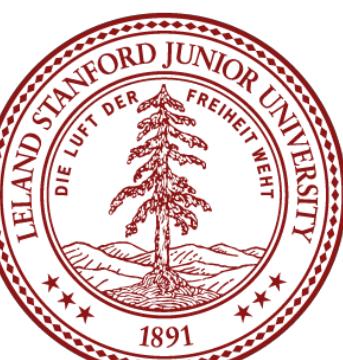
reading:

Programming Abstractions in C++, pp 721-722



Today's Topics

- Logistics
 - Okay Everybody! :D (re: Midquarter feedback)
 - HW 5: Today's lesson will cover the heap extension
 - Treat the code a bit like the `VectorInt` example we did in class
- Demo on why `delete` is important
- Binary Heaps
 - A "tree" structure
 - The Heap Property
 - Parents have higher priority than children



Why do we care about delete?

```
const int INIT_CAPACITY = 100000;

class Demo {
public:
    Demo(); // constructor
    string at(int i);
private:
    string *bigArray;
};

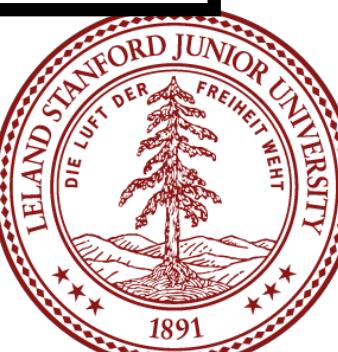
Demo::Demo()
{
    bigArray = new string[INIT_CAPACITY];
    for (int i=0;i<INIT_CAPACITY;i++) {
        bigArray[i] = "Lalalalalalalala!";
    }
}

string Demo::at(int i)
{
    return bigArray[i];
}
```

Let's see what happens!

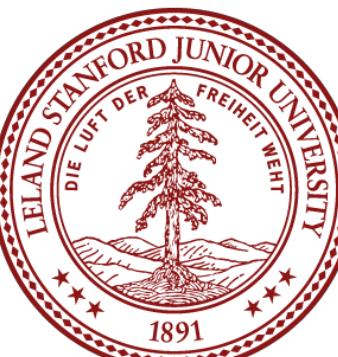


```
int main()
{
    for (int i=0;i<10000;i++){
        Demo demo;
        cout << i << ":" << demo.at(1234)
        << endl;
    }
    return 0;
}
```



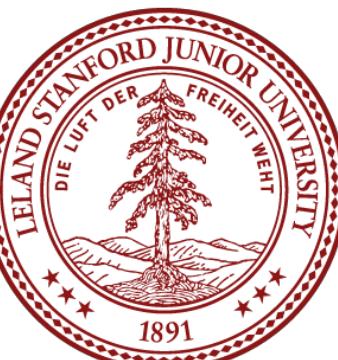
Priority Queues

- Sometimes, we want to store data in a “prioritized way.”
- Examples in real life:
 - Emergency Room waiting rooms
 - Professor Office Hours (what if a professor walks in? What about the department chair?)
 - Getting on an airplane (First Class and families, then frequent flyers, then by row, etc.)



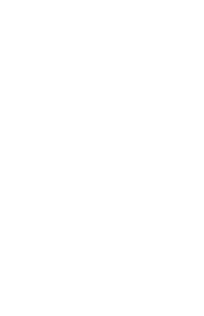
Priority Queues

- A “priority queue” stores elements according to their priority, and not in a particular order.
- This is fundamentally different from other position-based data structures we have discussed.
- There is no external notion of “position.”



Priority Queues

- A priority queue, P , has three fundamental operations:
- **enqueue (k, e)**: insert an element e with key k into P .
- **dequeue ()**: removes the element with the highest priority key from P .
- **peek ()**: return an element of P with the highest priority key (does not remove from queue).



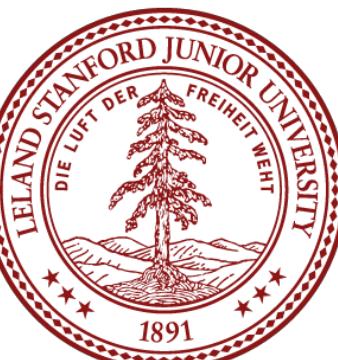
Priority Queues

- Priority queues also have less fundamental operations:
- **size ()**: returns the number of elements in P.
- **isEmpty ()**: Boolean test if P is empty.
- **clear ()**: empties the queue.
- **peekPriority ()**: Returns the priority of the highest priority element (why might we want this?)
- **changePriority (string value, int newPriority)**: Changes the priority of a value.



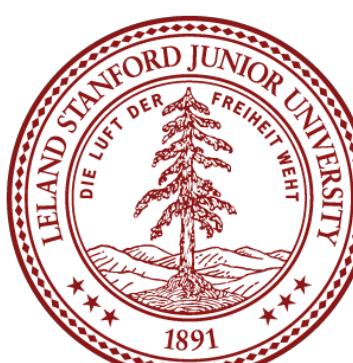
Priority Queues

- Priority queues are simpler than sequences: no need to worry about position (or **insert(index, value)**, **add(value)** to append, **get(index)**, etc.).
- We only need one **enqueue()** and **dequeue()** function



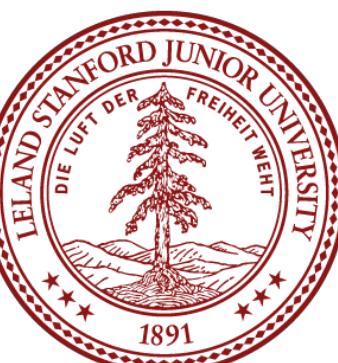
Priority Queues

<i>Operation</i>	<i>Output</i>	<i>Priority Queue</i>
enqueue(5,A)	-	$\{(5,A)\}$
enqueue(9,C)	-	$\{(5,A),(9,C)\}$
enqueue(3,B)	-	$\{(5,A),(9,C),(3,B)\}$
enqueue(7,D)	-	$\{(5,A),(9,C),(3,B),(7,D)\}$
peek()	B	$\{(5,A),(9,C),(3,B),(7,D)\}$
peekPriority()	3	$\{(5,A),(9,C),(3,B),(7,D)\}$
dequeue()	B	$\{(5,A),(9,C),(7,D)\}$
size()	3	$\{(5,A),(9,C),(7,D)\}$
peek()	A	$\{(5,A),(9,C),(7,D)\}$
dequeue()	A	$\{(9,C),(7,D)\}$
dequeue()	D	$\{(9,C)\}$
dequeue()	C	$\{\}$
dequeue()	error!	$\{\}$
isEmpty()	TRUE	$\{\}$



Binary Heaps

- For HW 5, you will build a priority queue using a Vector and a linked list, and as an extension, using a "binary heap"
- A heap is a *tree-based* structure that satisfies the heap property:
- Parents have a higher priority key than any of their children.

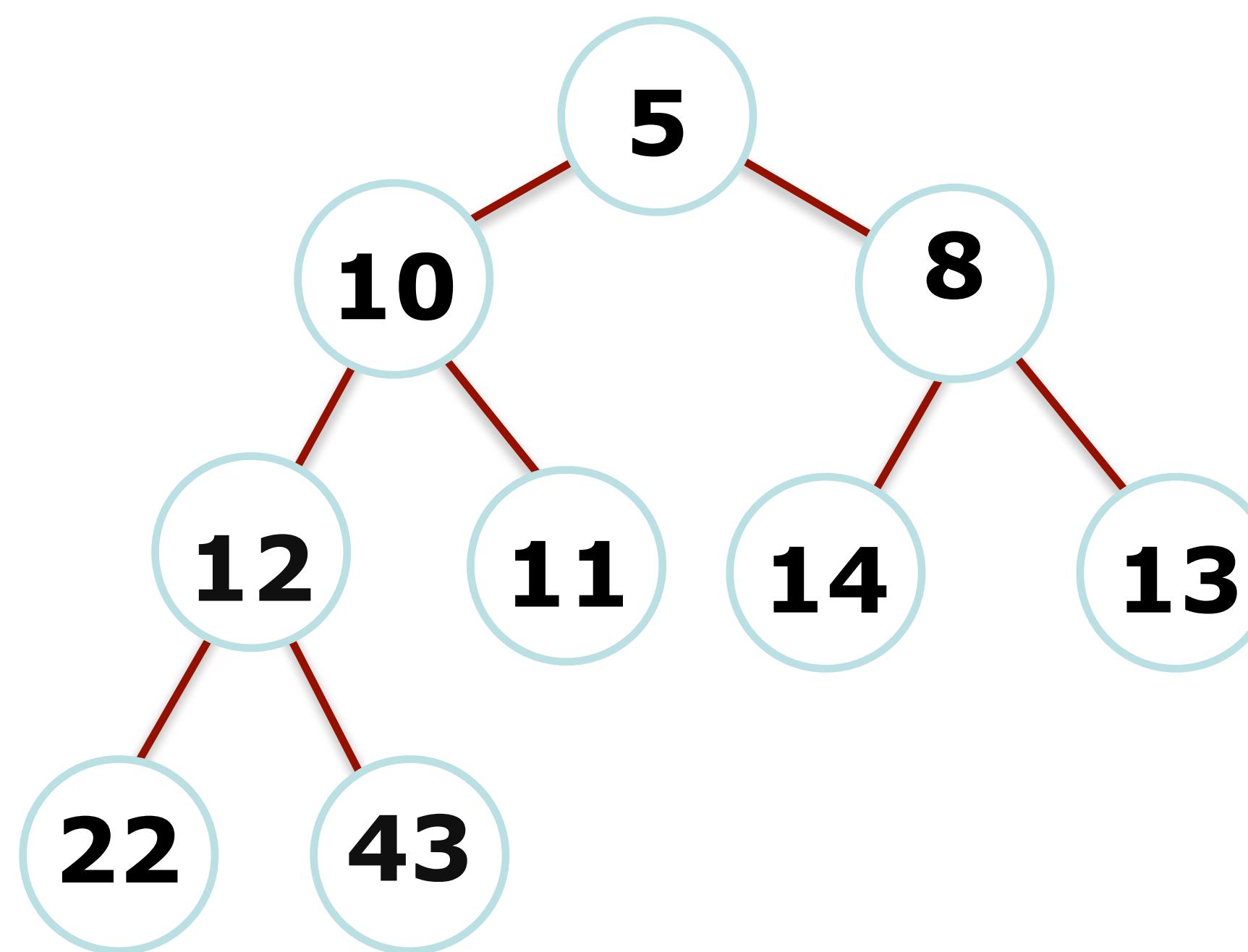


Binary Heaps

- There are two types of heaps:

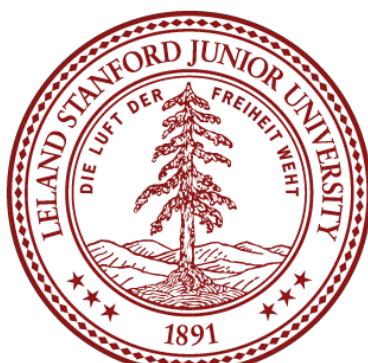
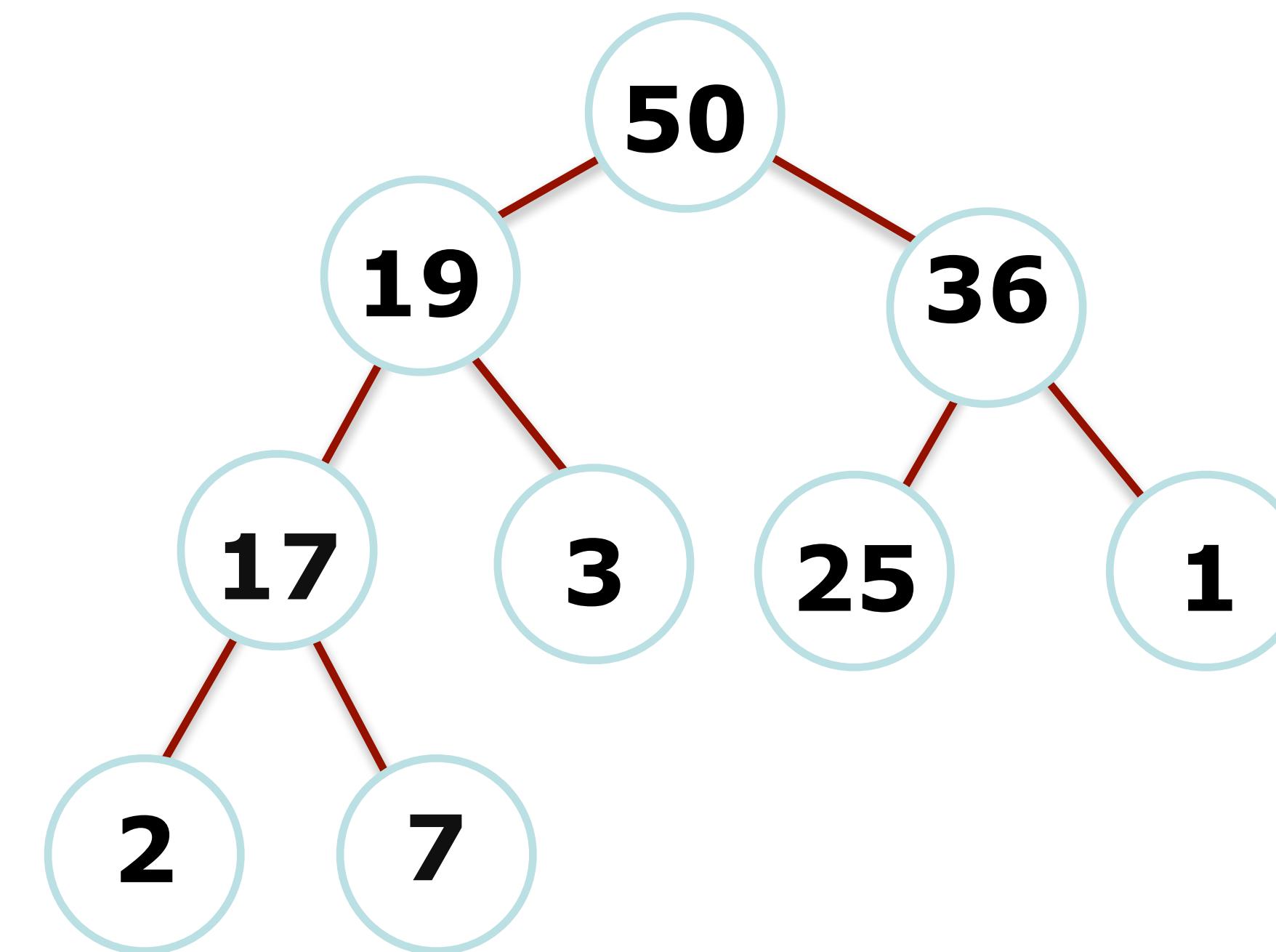
Min Heap

(root is the smallest element)



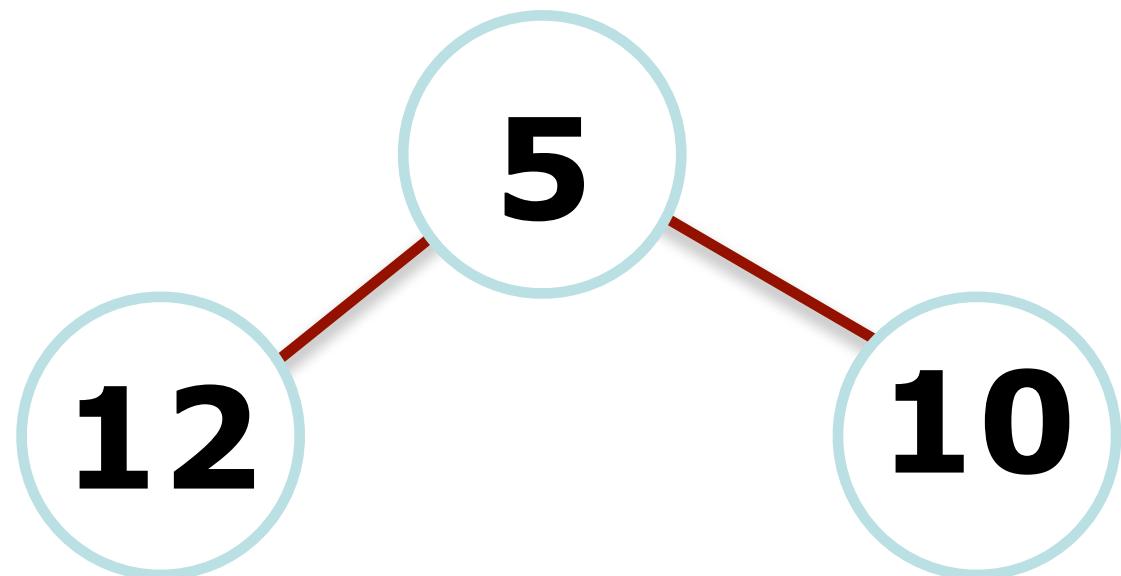
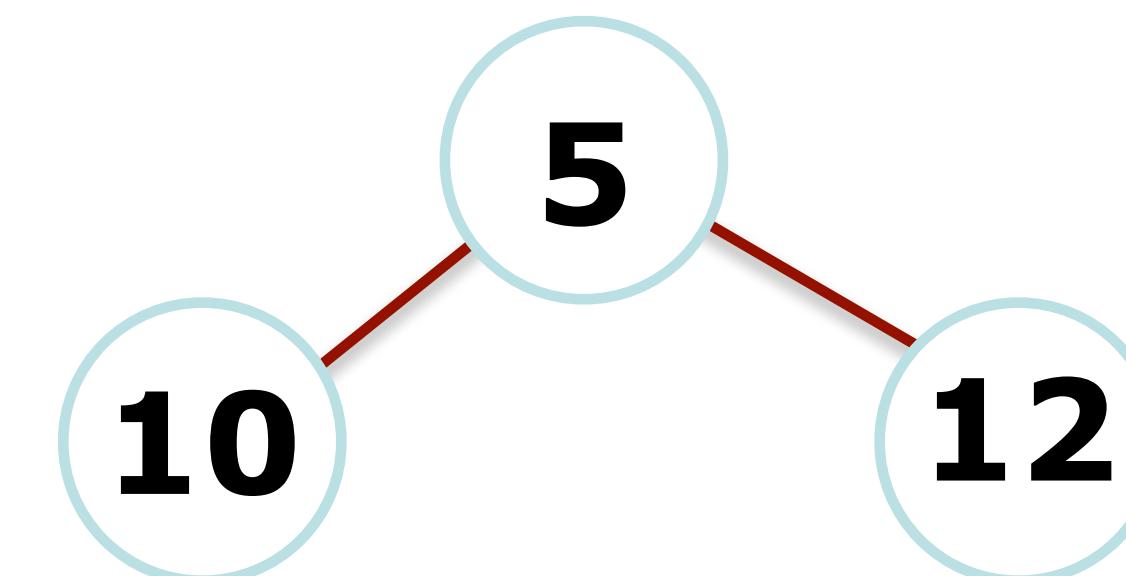
Max Heap

(root is the largest element)



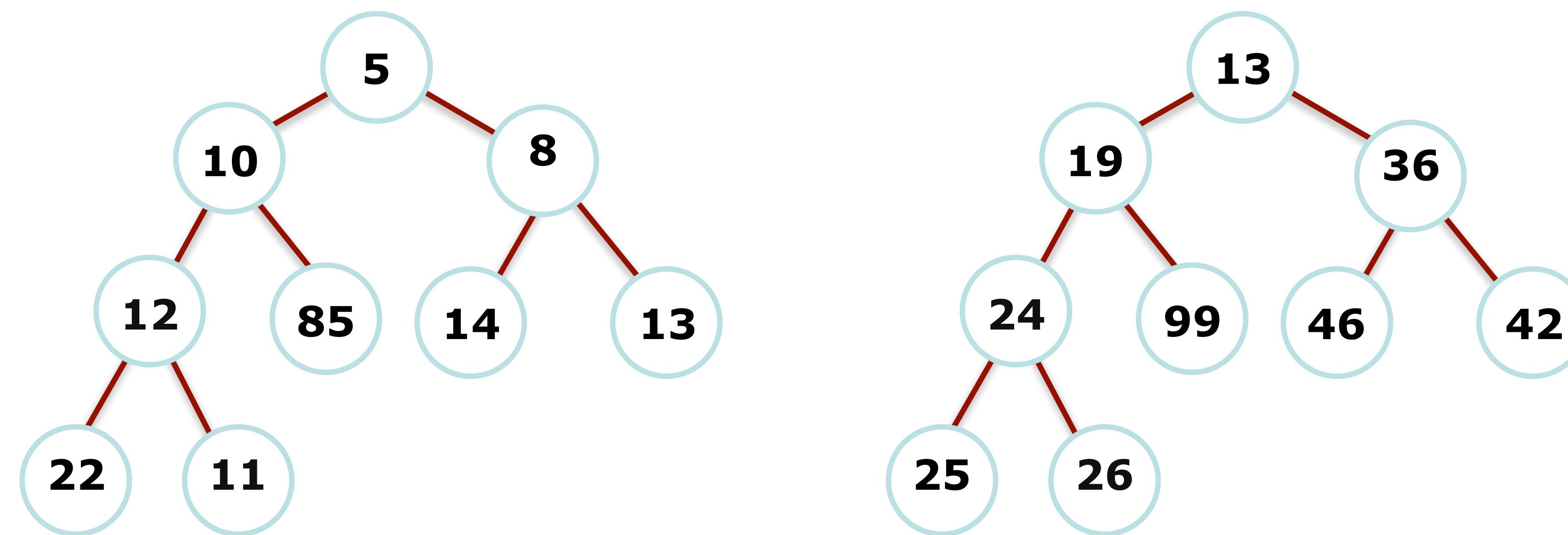
Binary Heaps

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



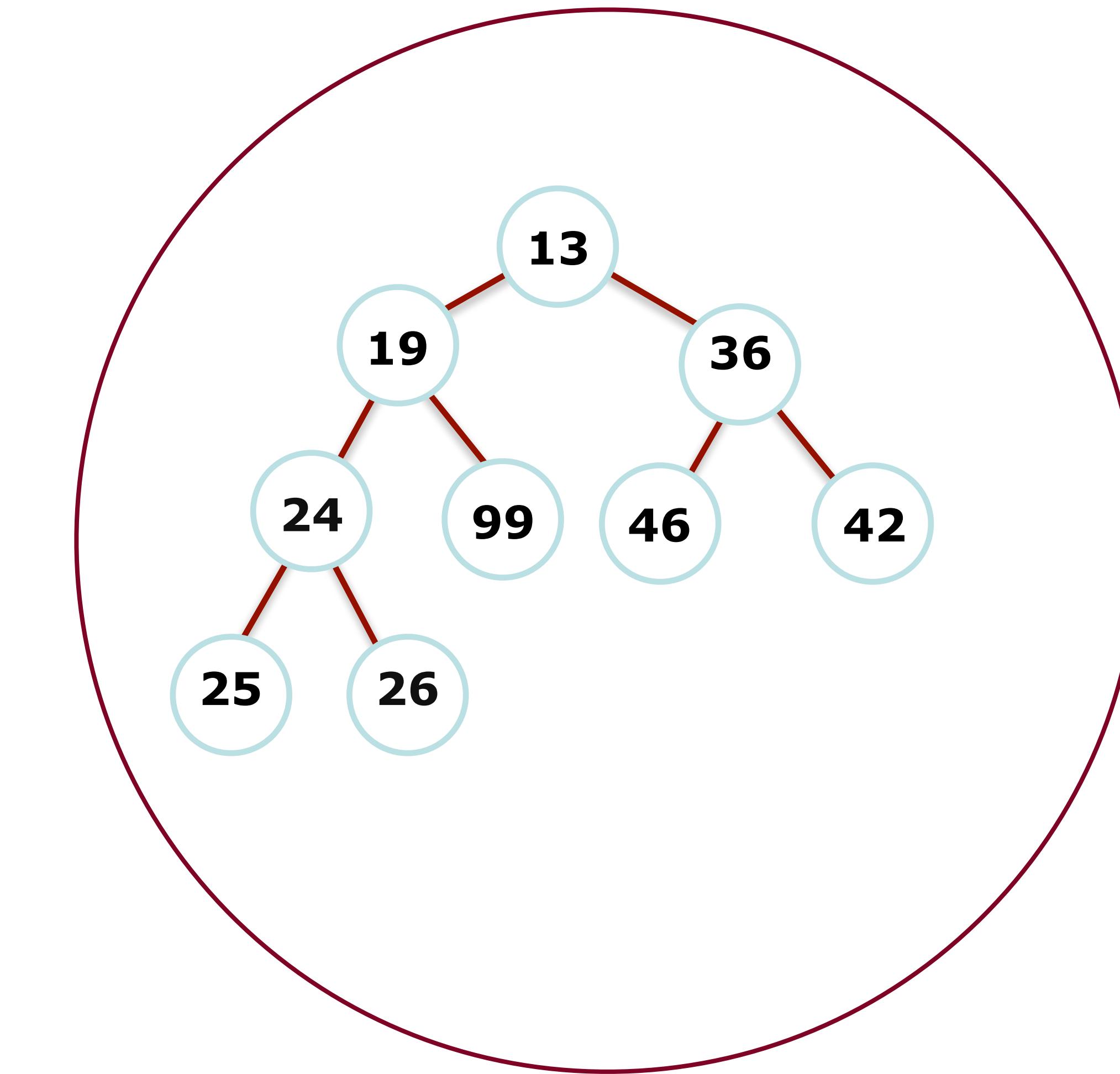
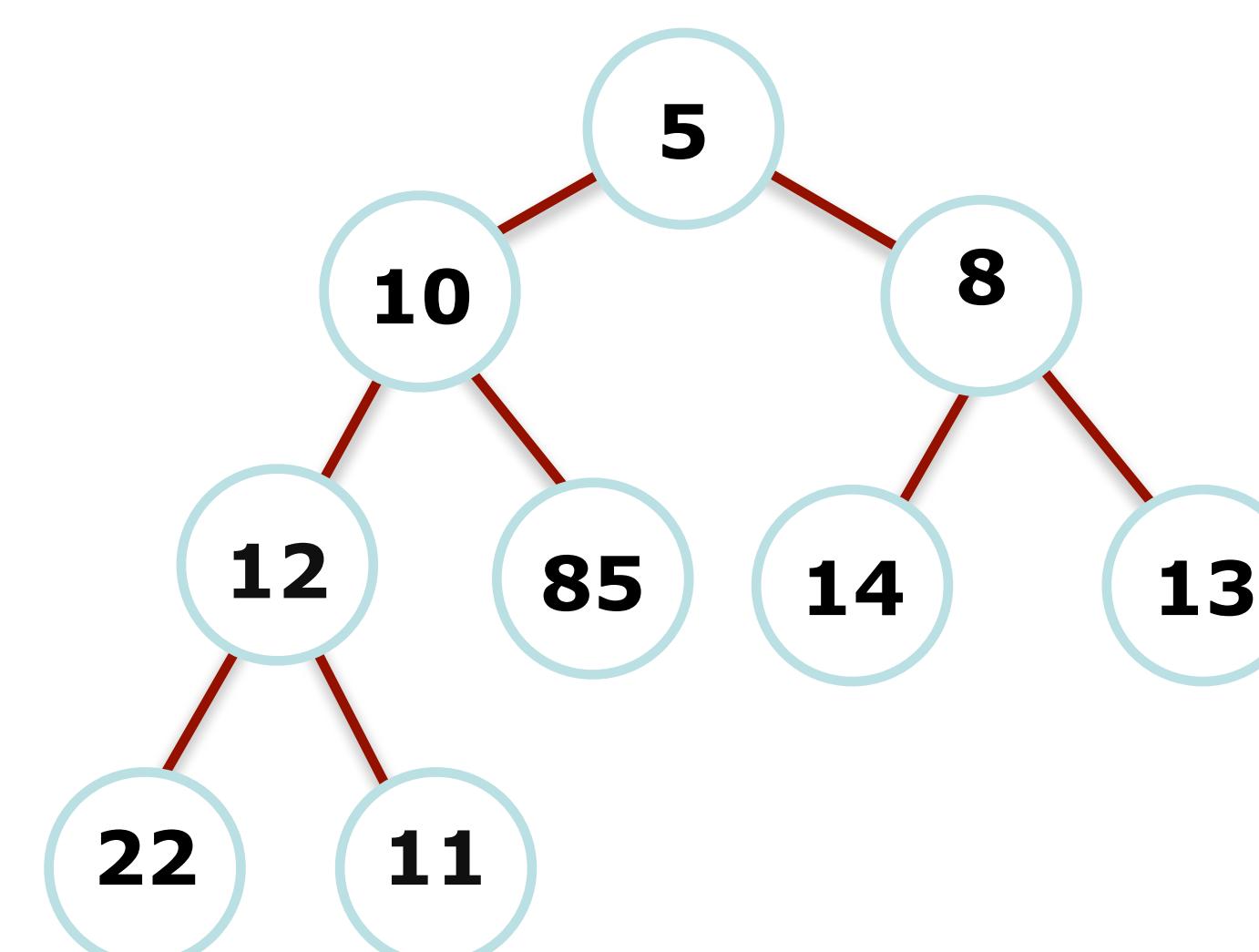
Binary Heaps

- Circle the min-heap(s):



Binary Heaps

- Circle the min-heap(s):

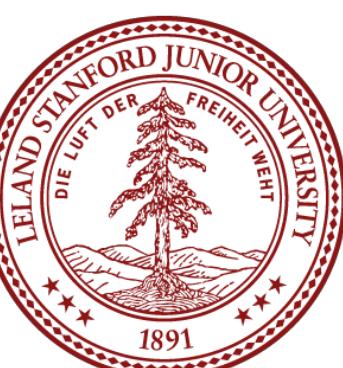
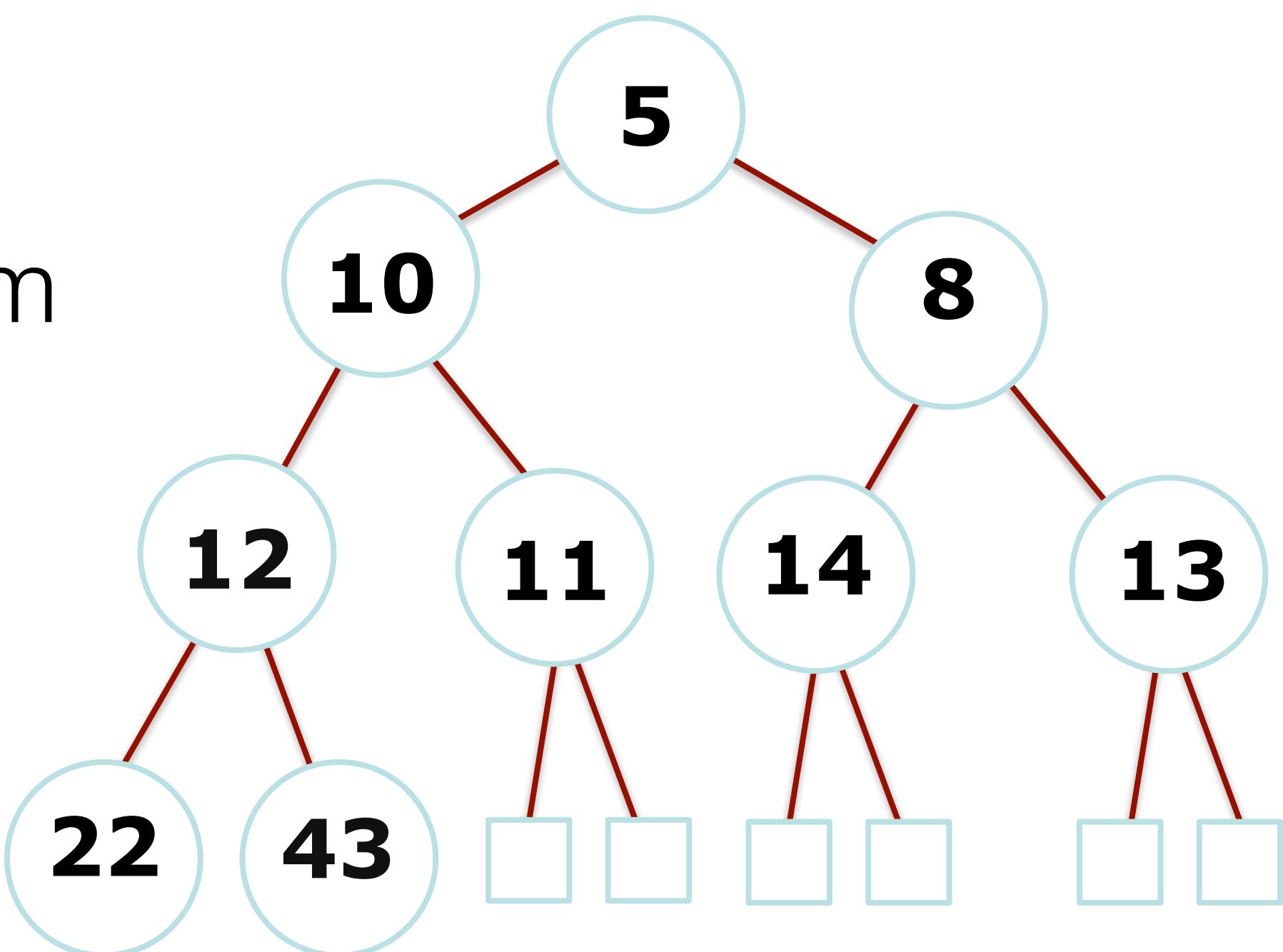


Binary Heaps

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

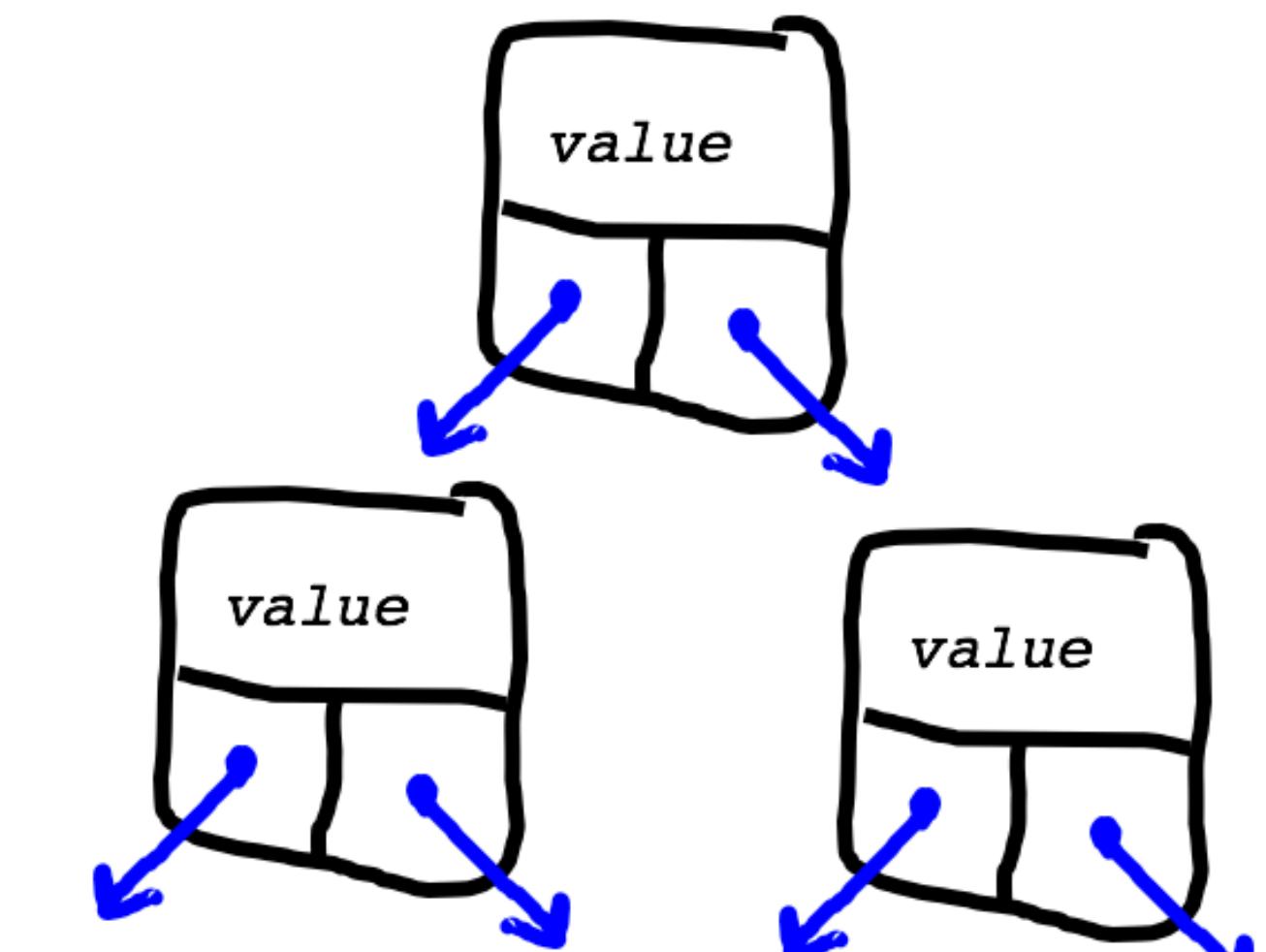
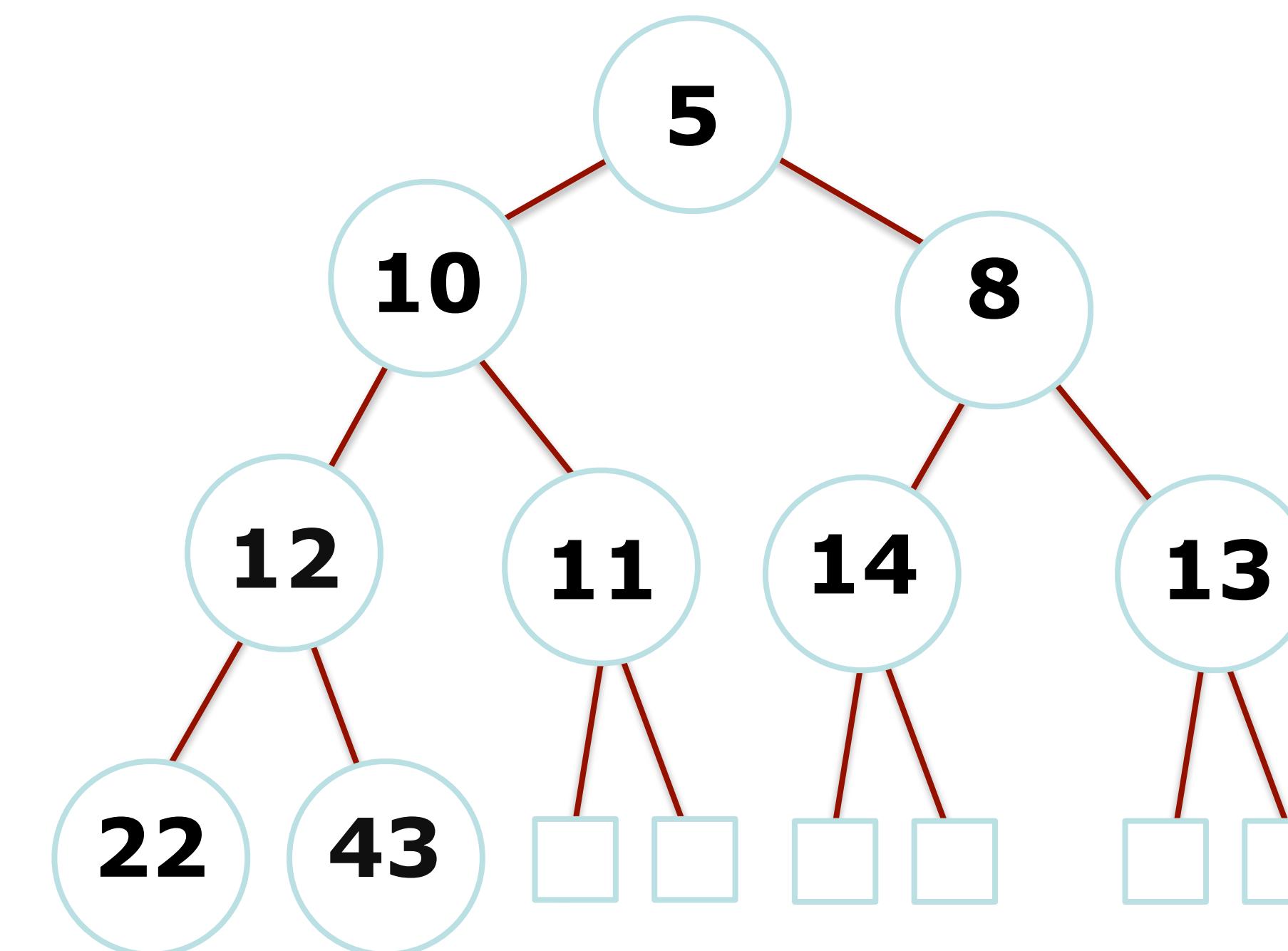
complete: all levels filled except the bottom
binary: two children per node (parent)

- Maximum number of nodes
- Filled from left to right



Binary Heaps

What is the best way to store a heap?



We could use a node-based solution, but...

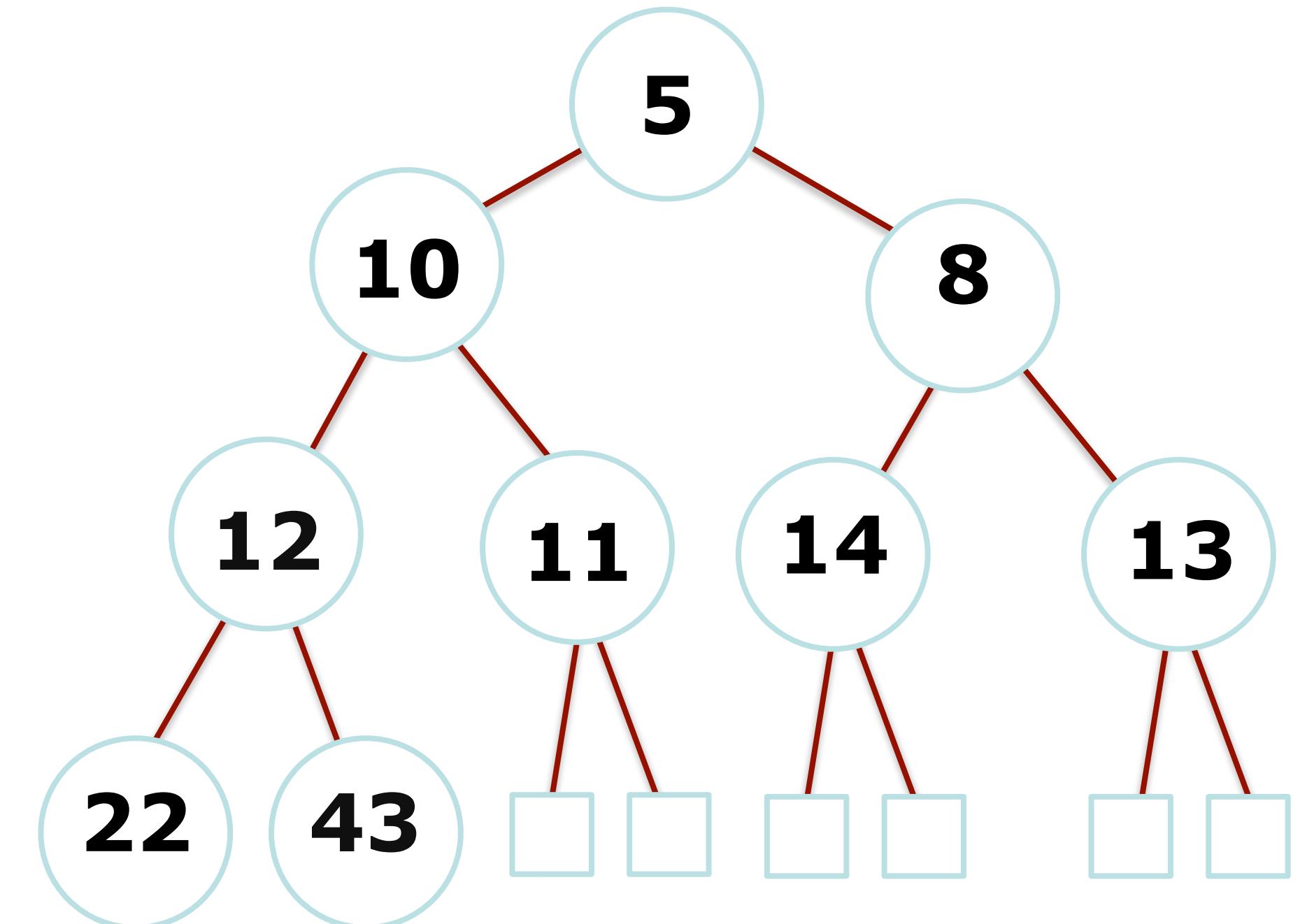


Binary Heaps

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]

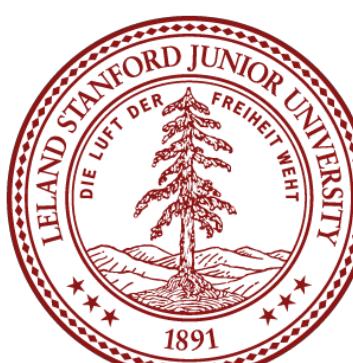
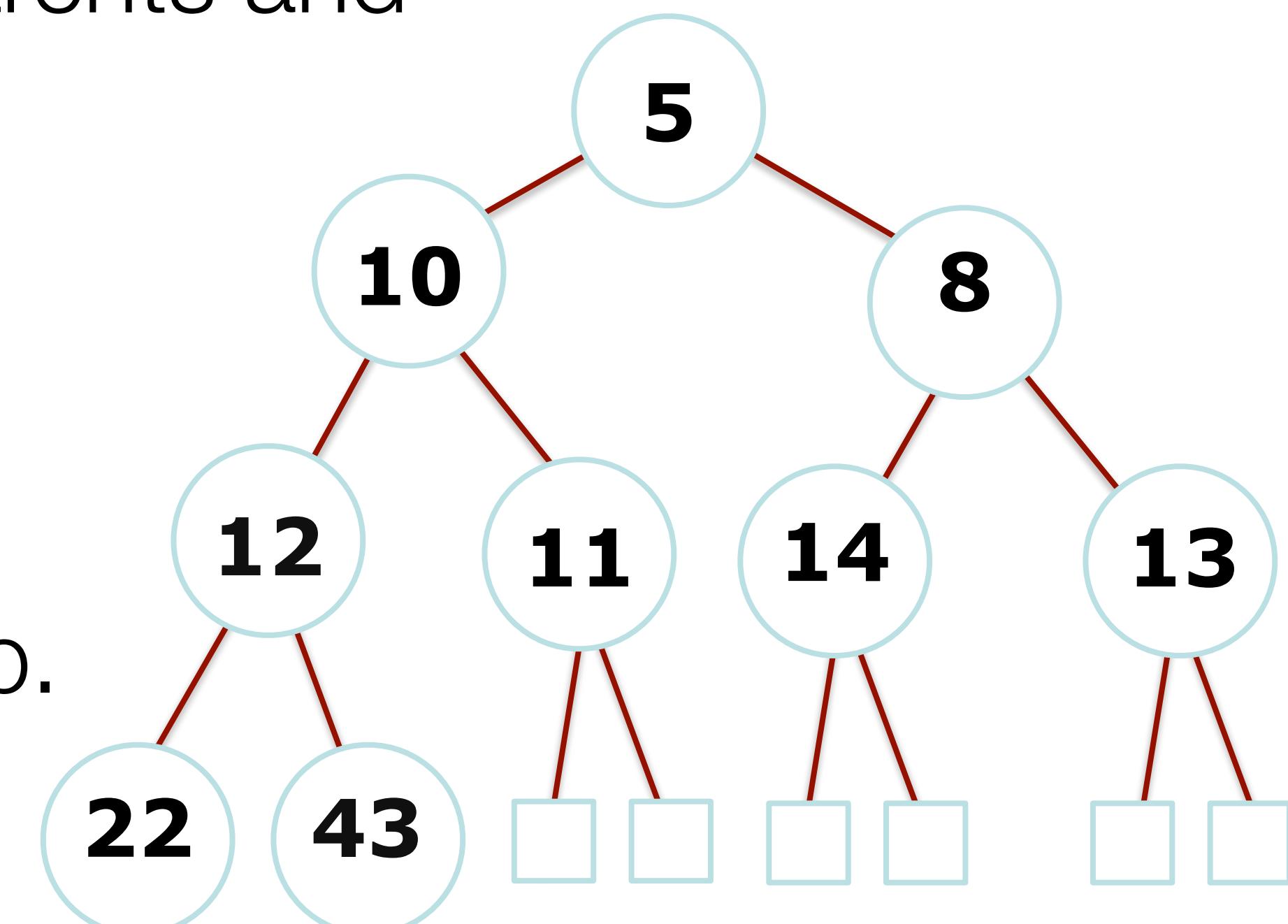


Binary Heaps

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

- For an element at position i :
 - left child is at $2i$
 - right child is at $2i+1$
 - parent is at $\lfloor i/2 \rfloor$
- *heapSize*: the number of elements in the heap.

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

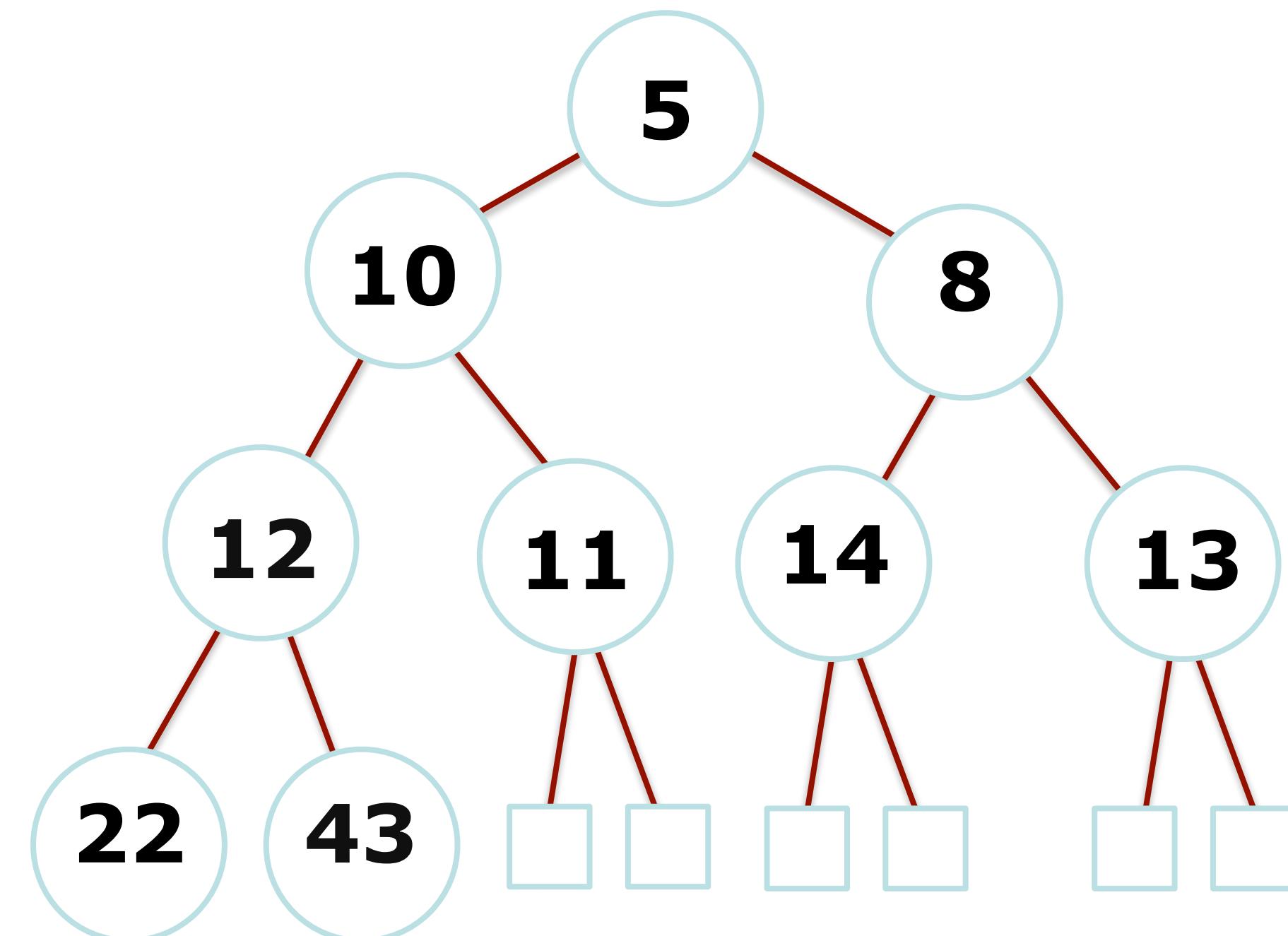


Heap Operations

Remember that there are three important priority queue operations:

1. **peek ()** : return an element of h with the smallest key.
2. **enqueue (k, e)** : insert an element e with key k into the heap.
3. **dequeue ()** : removes the smallest element from h .

We can accomplish this with a heap!
We will just look at keys for now -- just know that we will also store a value with the key.



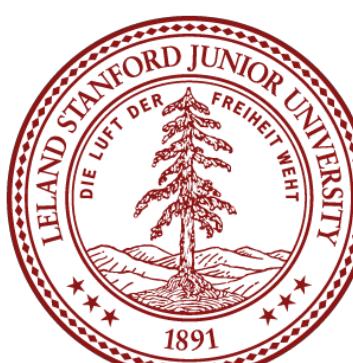
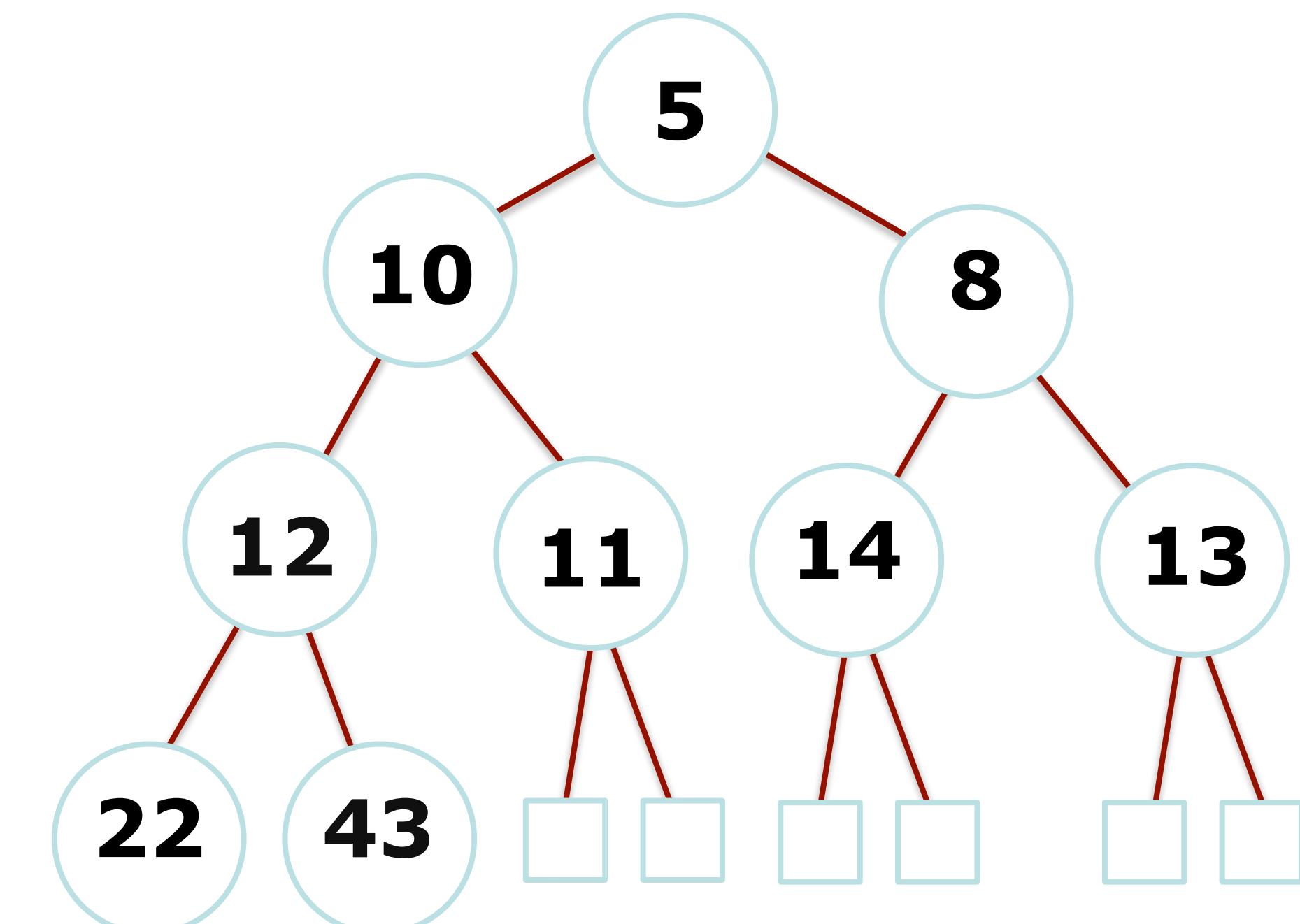
Heap Operations: peek()

peek () :

Just return the root!
return heap[1]

O(1) yay!

	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)

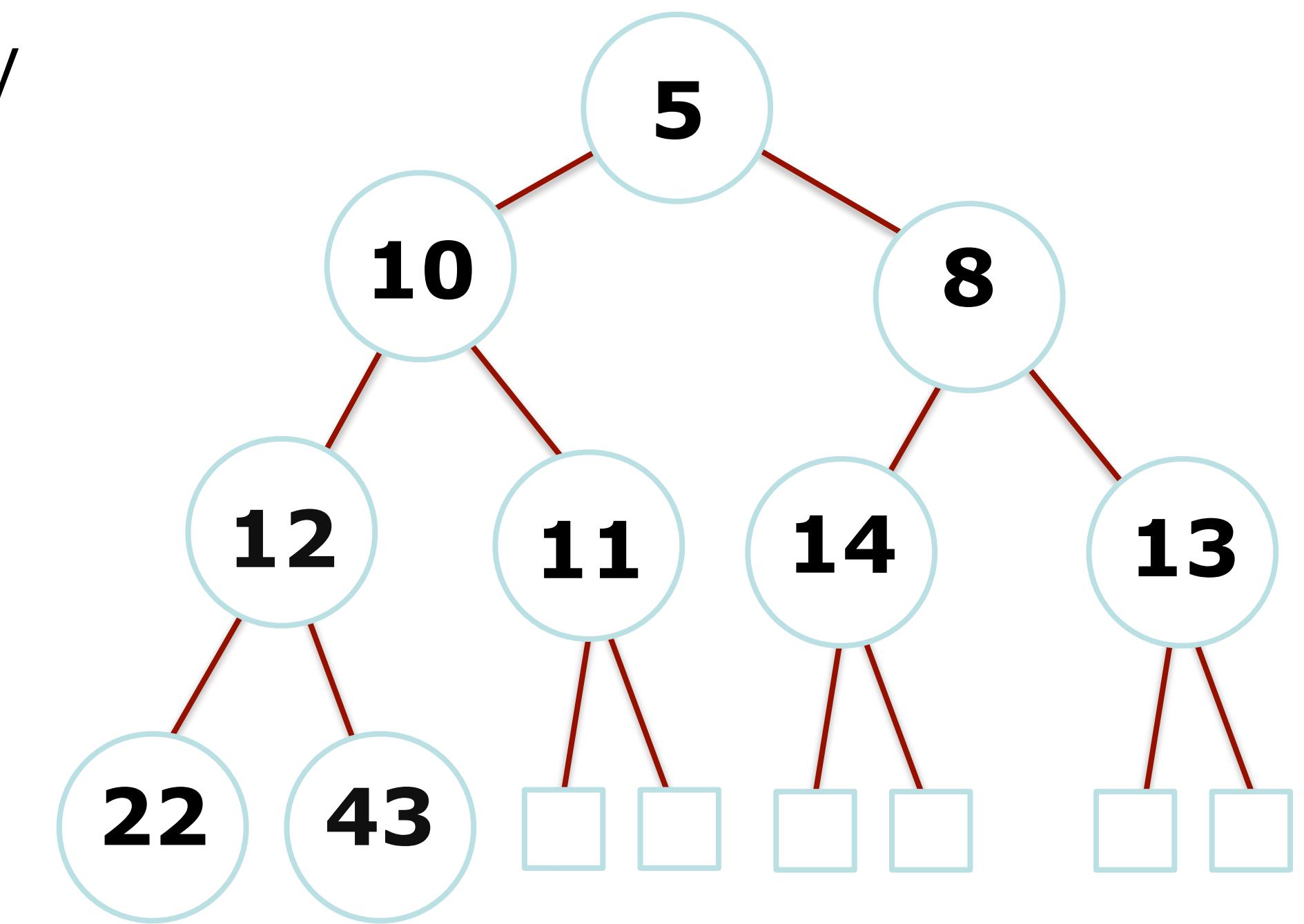
enqueue (k)

- How might we go about inserting into a binary heap?

enqueue (9)

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

21

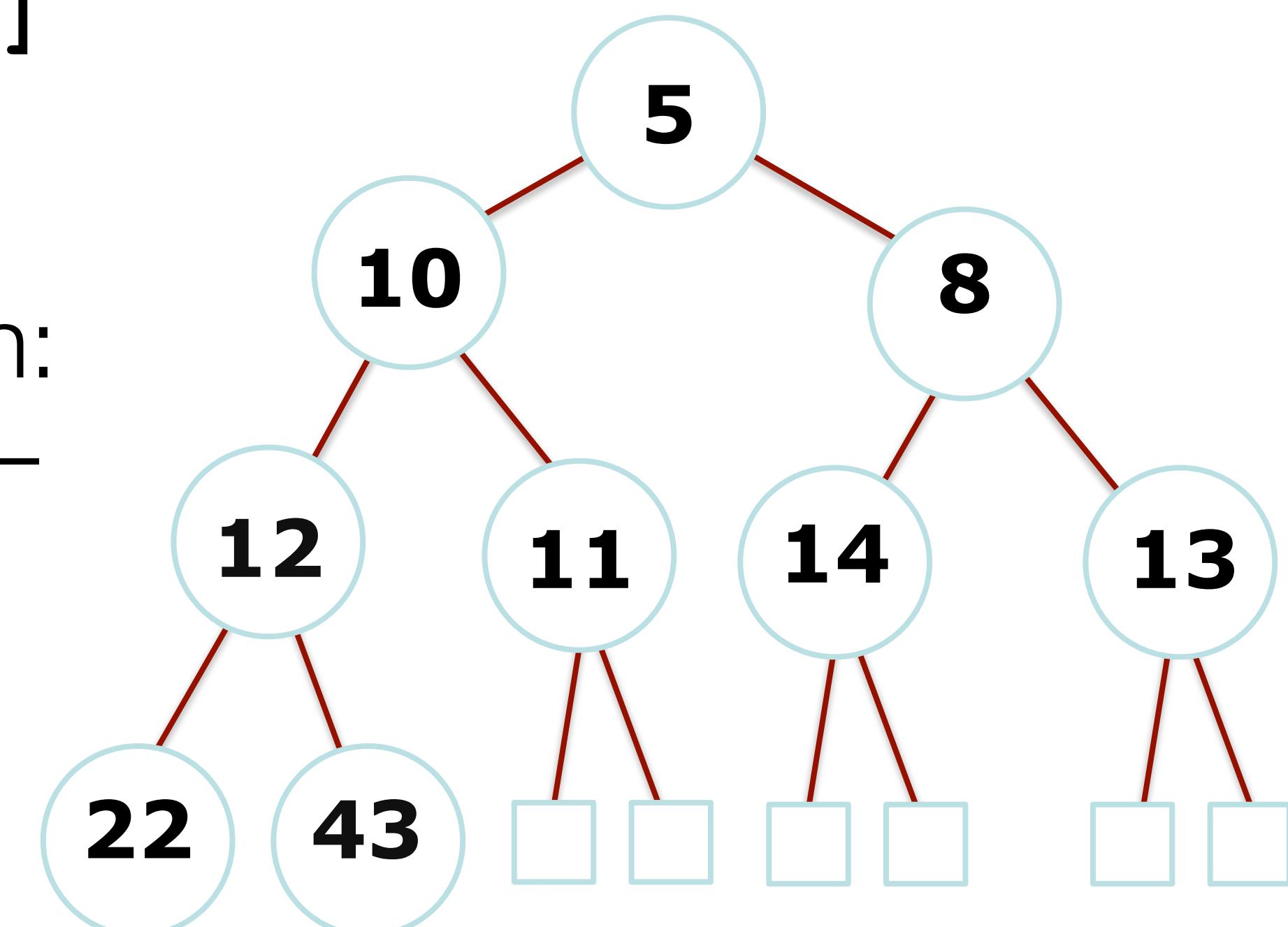


Heap Operations: enqueue(k)

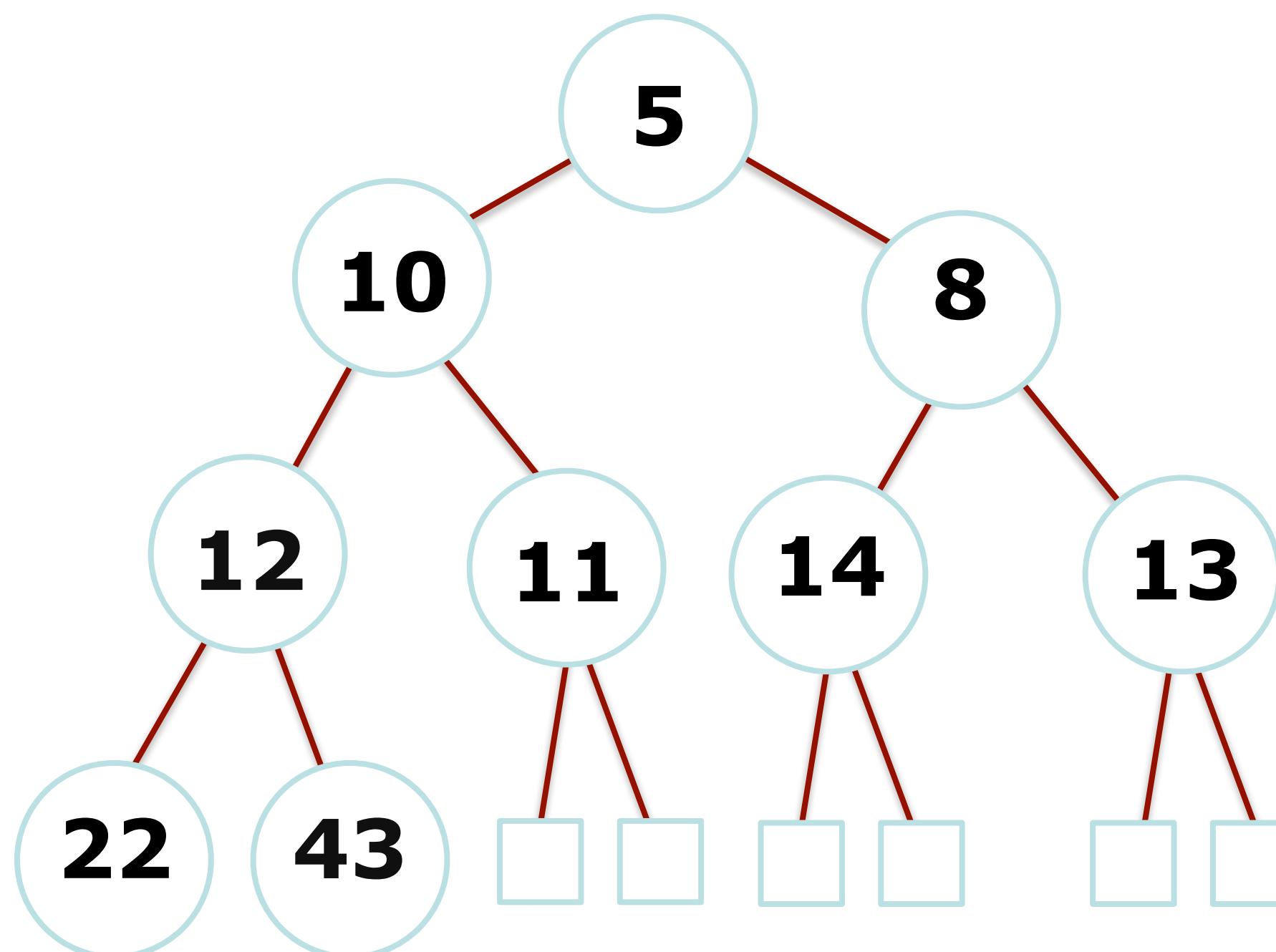
Heap Operations: **enqueue (k)**

1. Insert item at element **array [heap.size ()+1]**
(this probably destroys the heap property)
2. Perform a “bubble up,” or “up-heap” operation:
 - a. Compare the added element with its parent — if in correct order, stop
 - b. If not, swap and repeat step 2.

See animation at: <http://www.cs.usfca.edu/~galles/visualization/Heap.html>

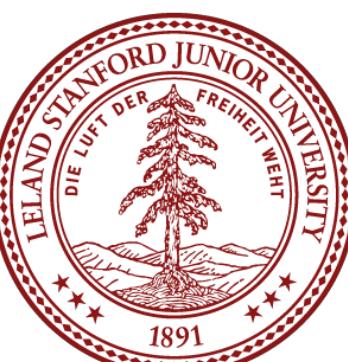


Heap Operations: enqueue(9)

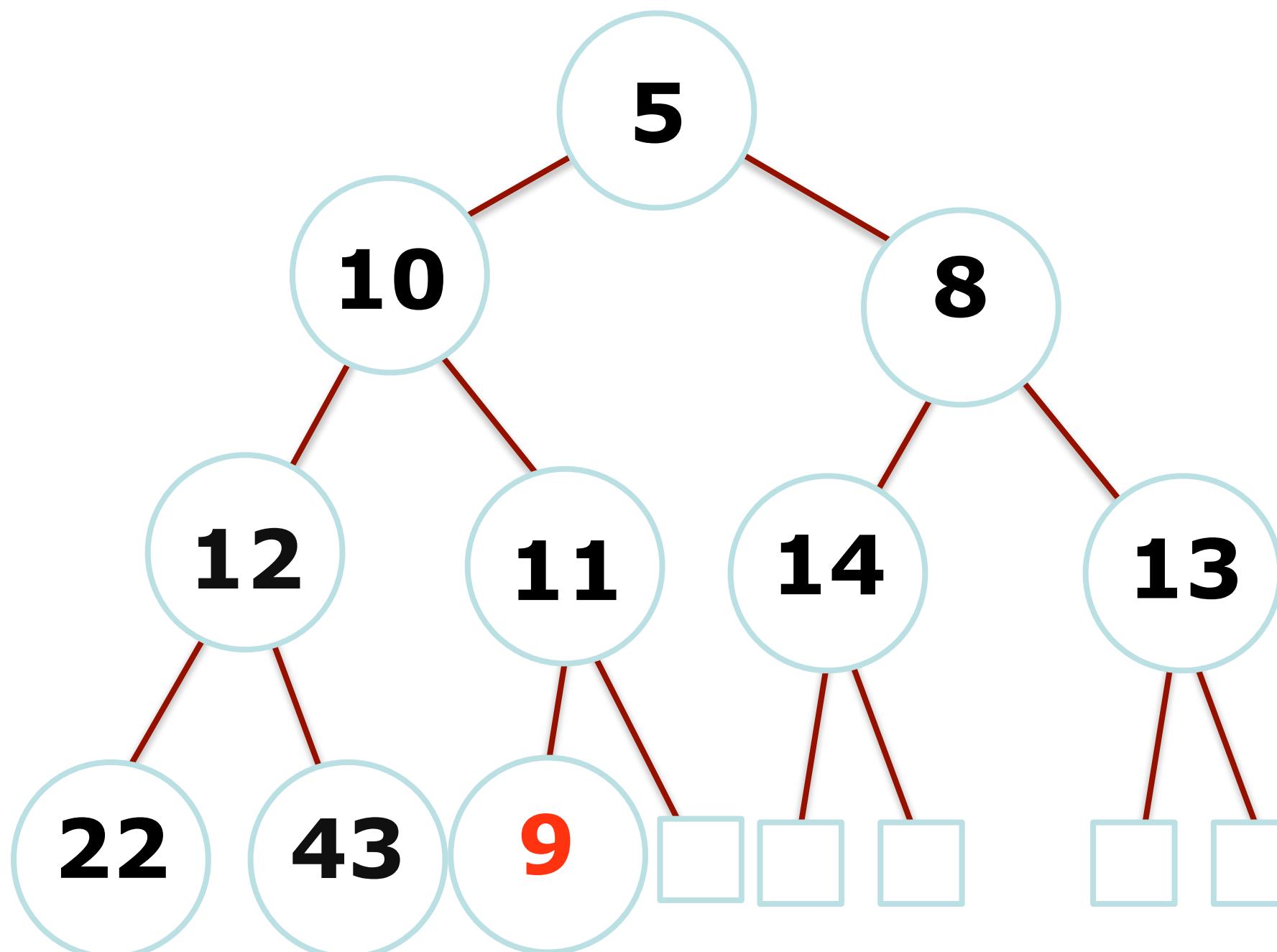


	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

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



Heap Operations: enqueue(9)

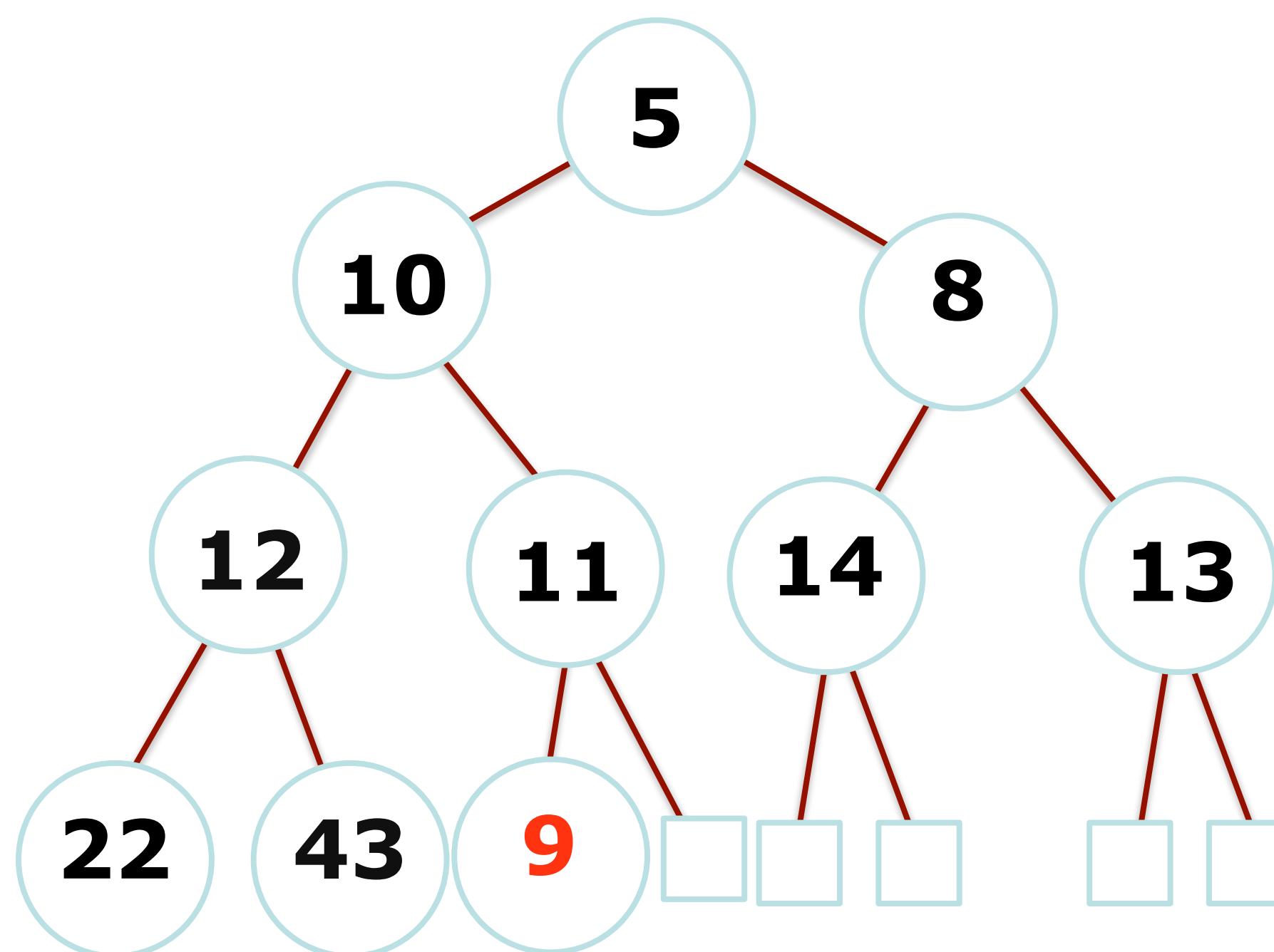


	5	10	8	12	11	14	13	22	43	9	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

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



Heap Operations: enqueue(9)



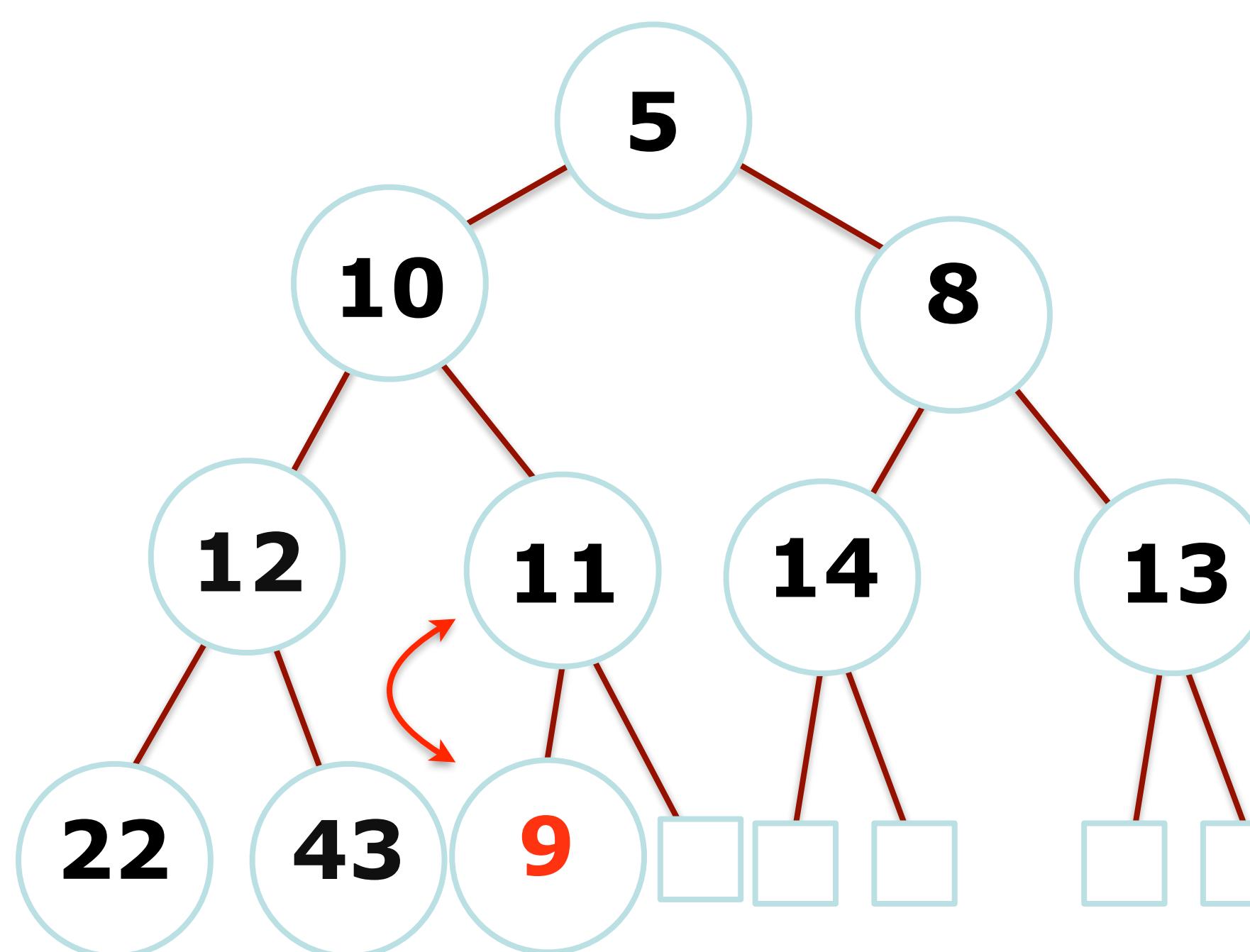
	5	10	8	12	11	14	13	22	43	9	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

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

No -- we must swap.



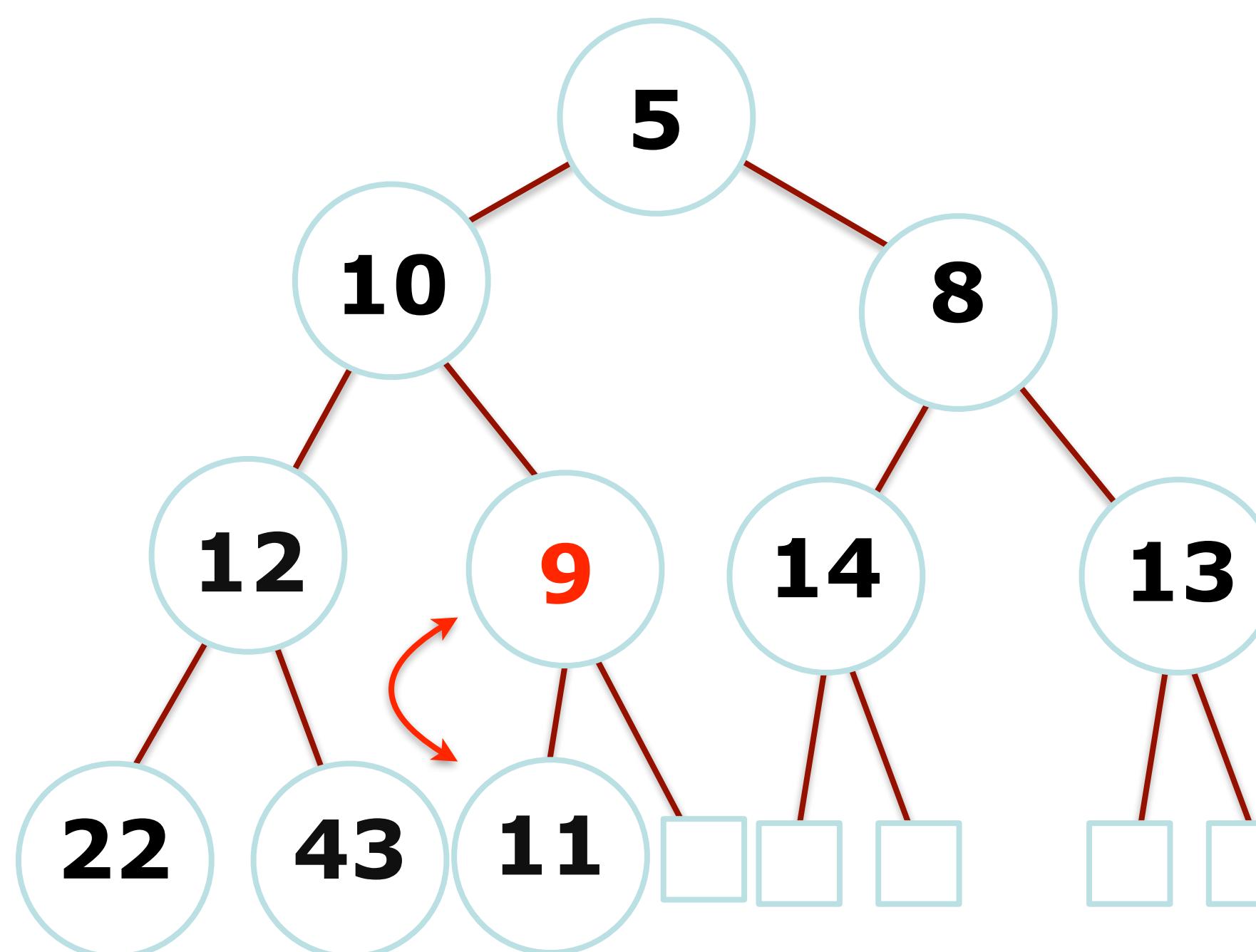
Heap Operations: enqueue(9)



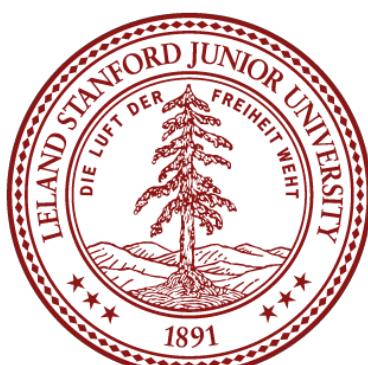
	5	10	8	12	11	14	13	22	43	9	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



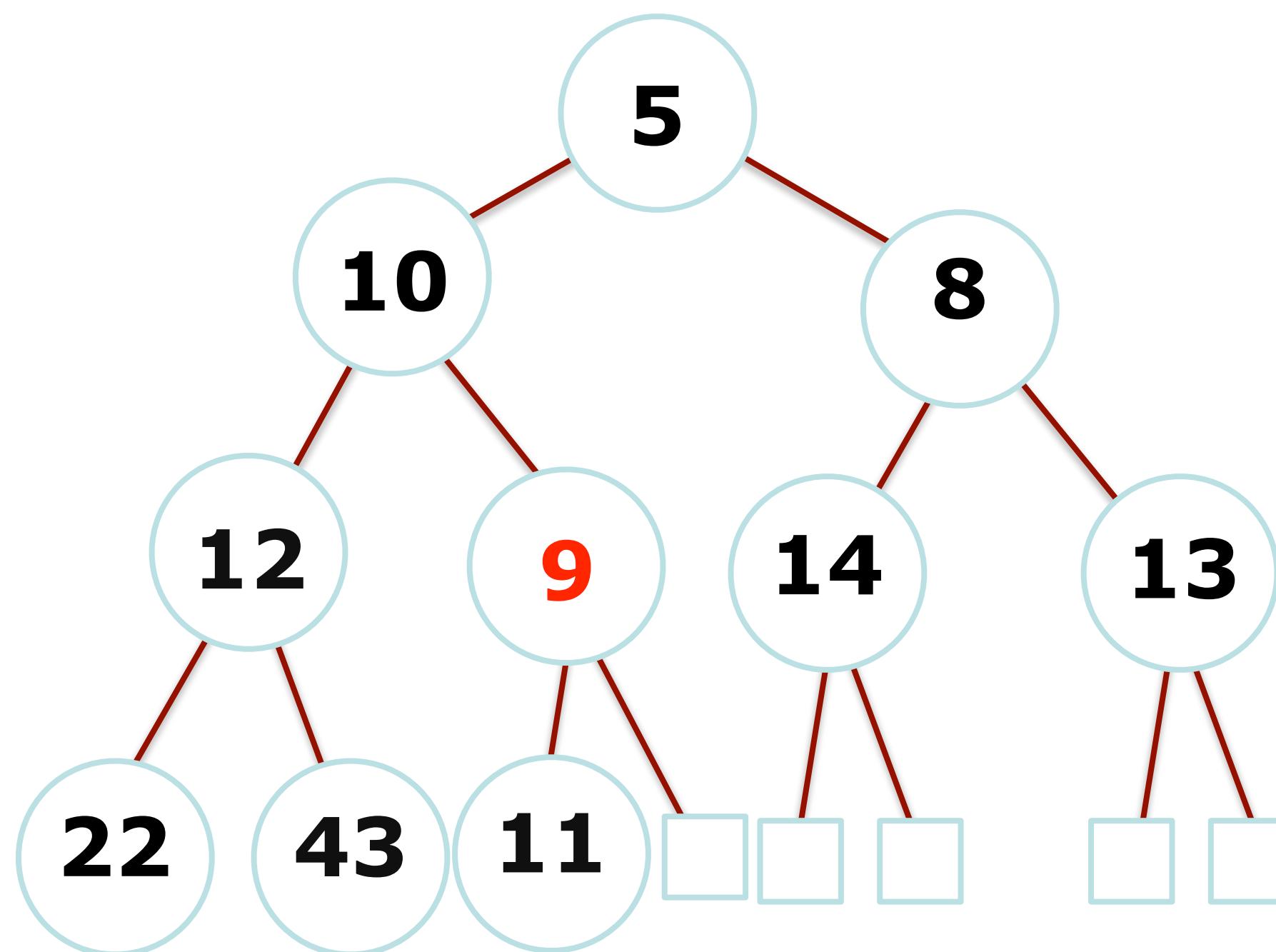
Heap Operations: enqueue(9)



	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)



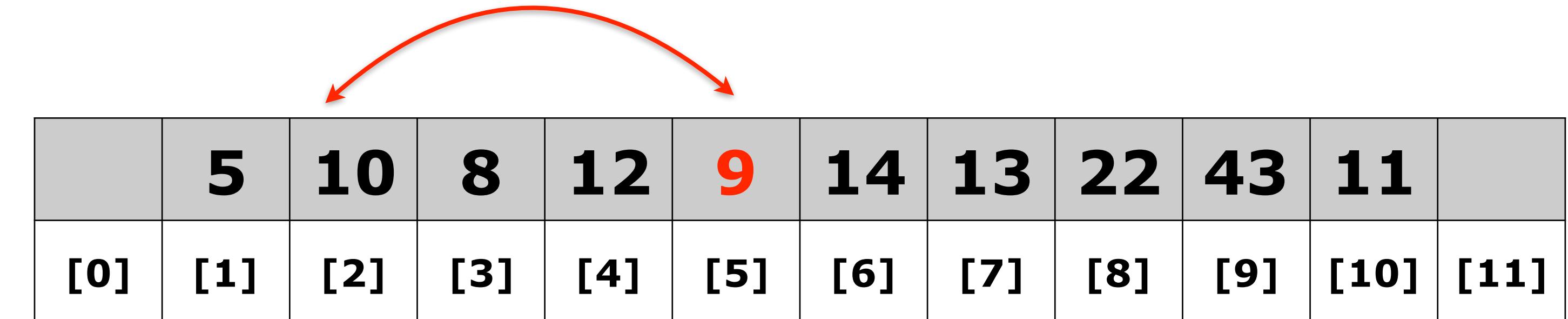
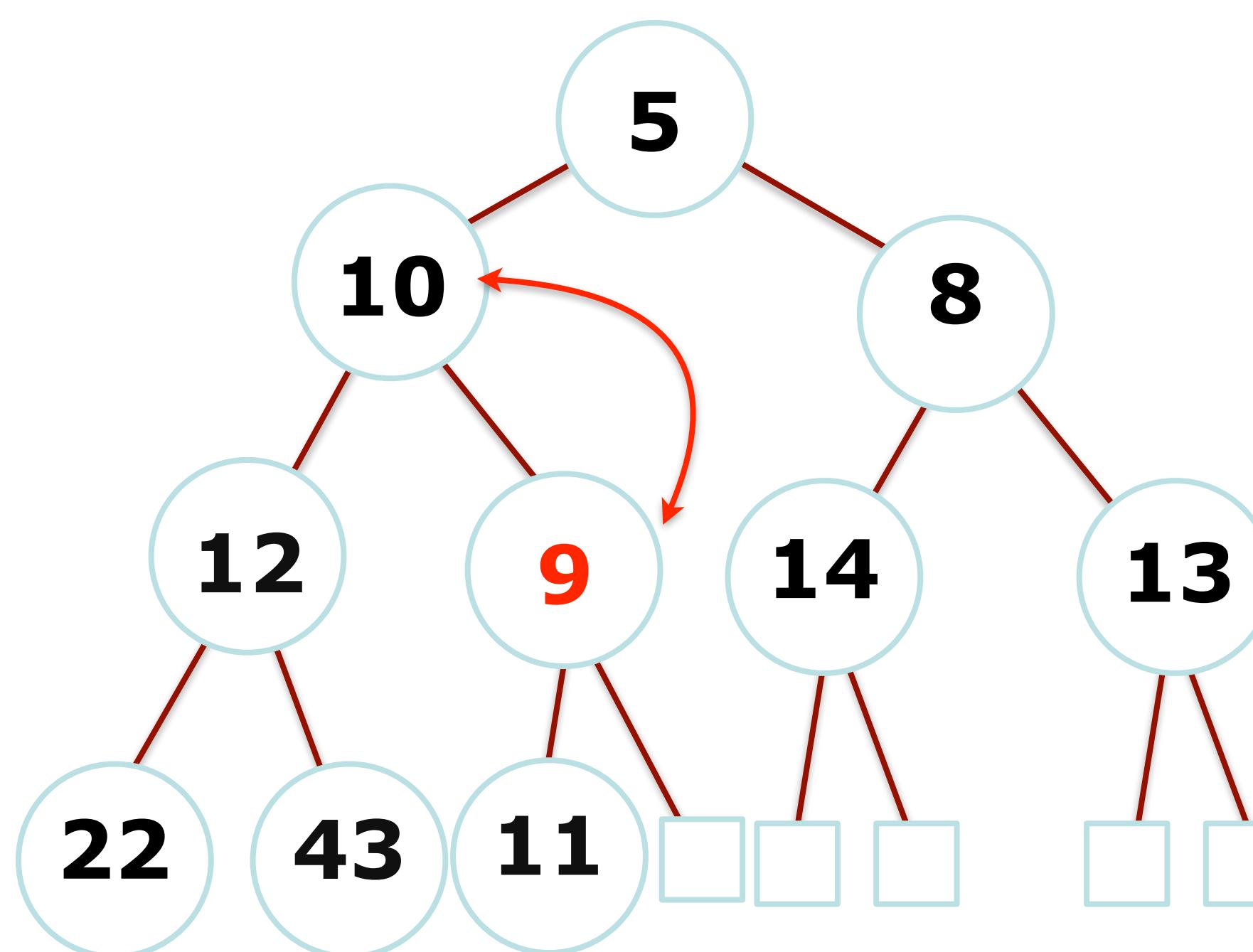
	5	10	8	12	9	14	13	22	43	11	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

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

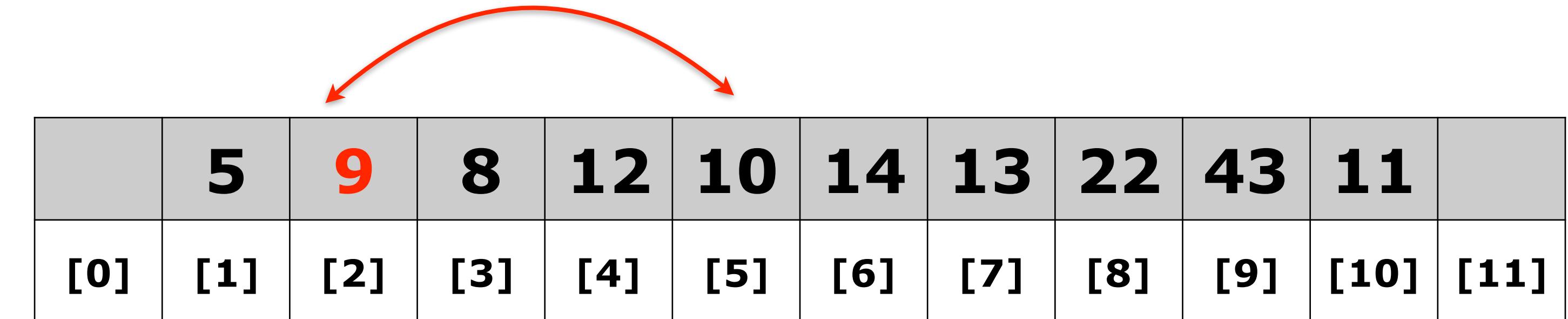
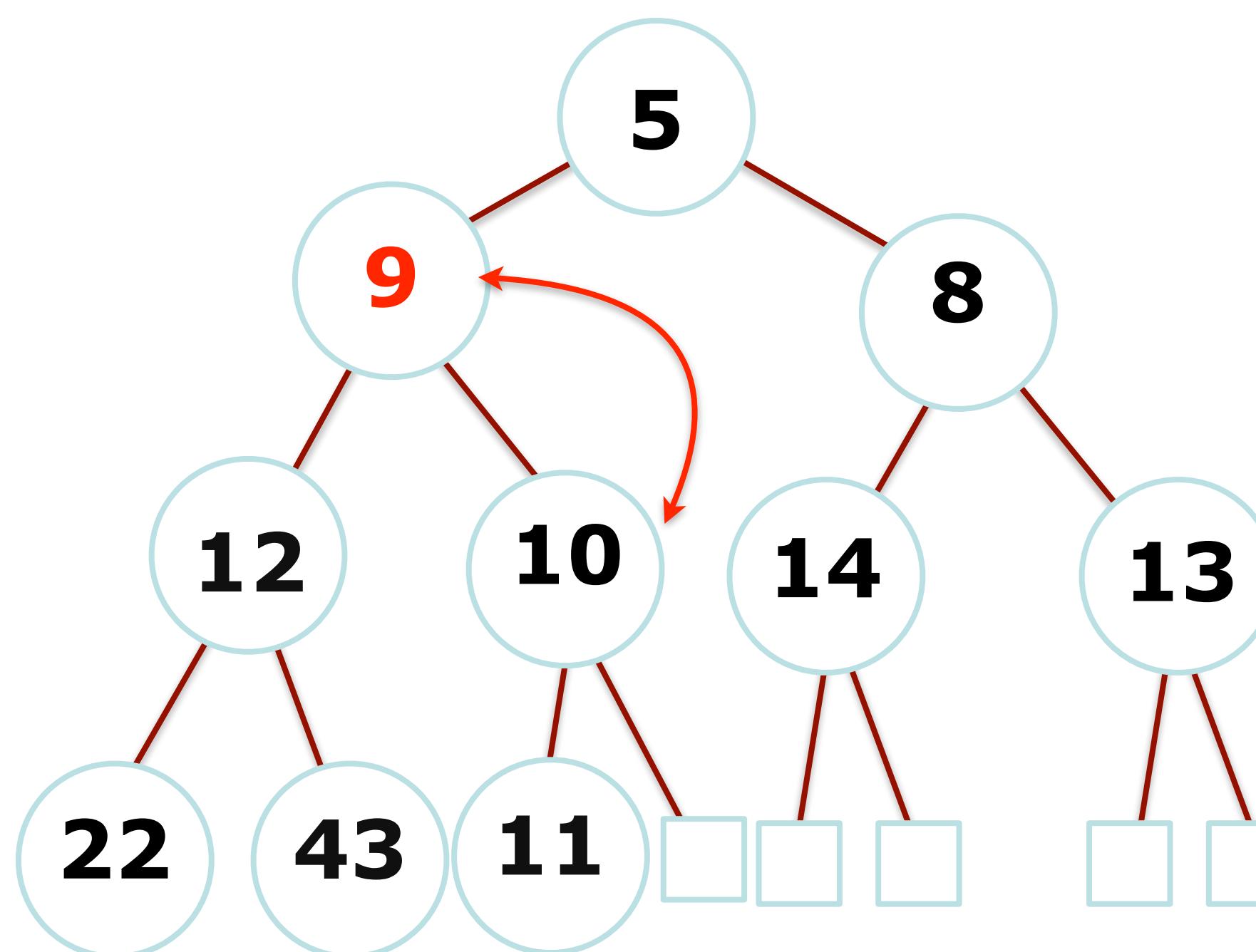
No -- we must swap. This "bubbling up" won't ever be a problem if the heap is "already a heap" (i.e., already meets heap property for all nodes)



Heap Operations: enqueue(9)

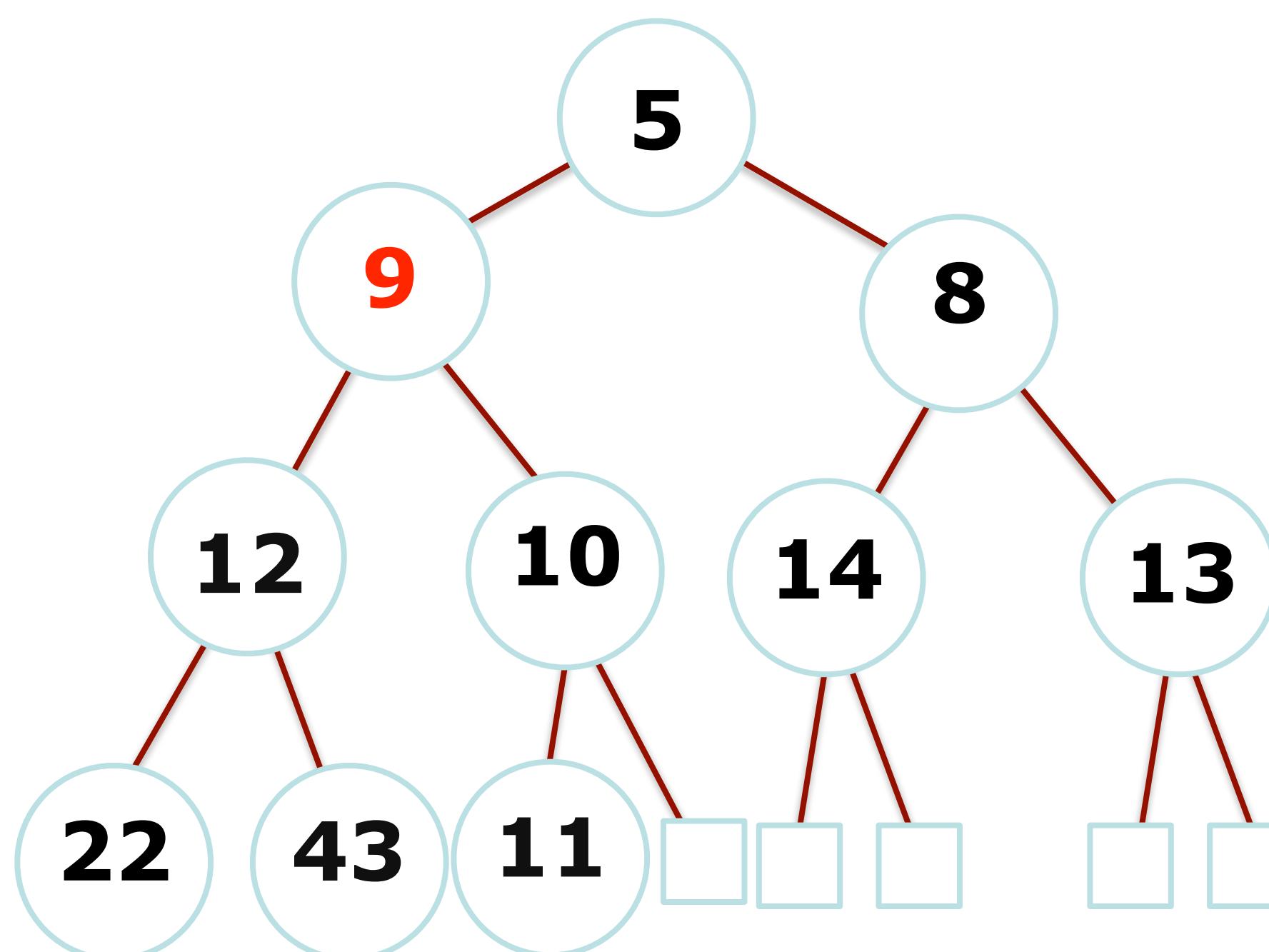


Heap Operations: enqueue(9)



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]

Complexity? $O(\log n)$ - yay!
Average complexity for random inserts:
 $O(1)$, see: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6312854

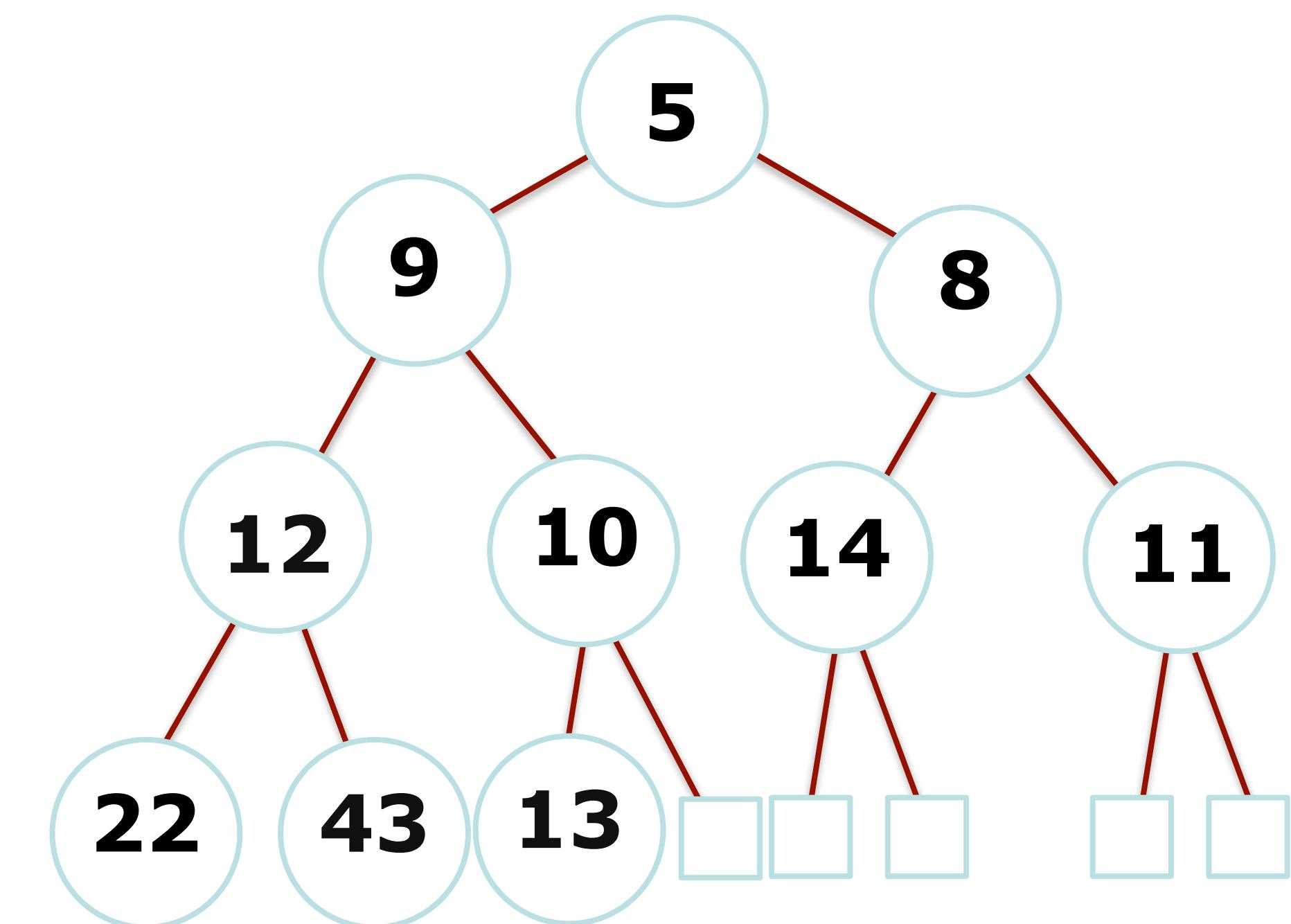


Heap Operations: dequeue()

- How might we go about removing the minimum?

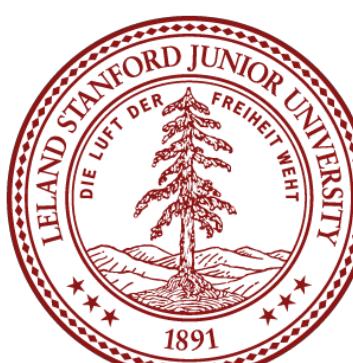
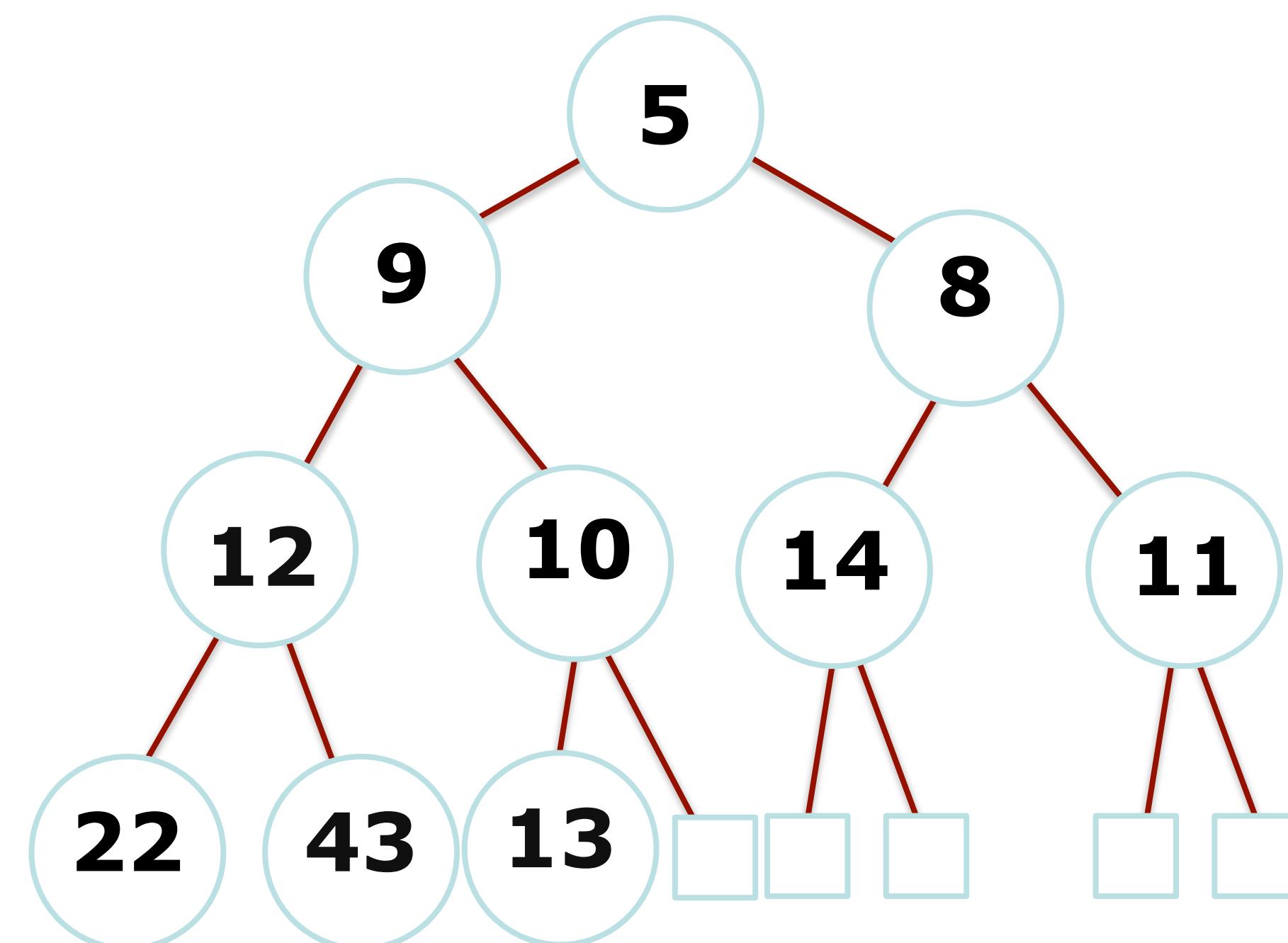
dequeue ()

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

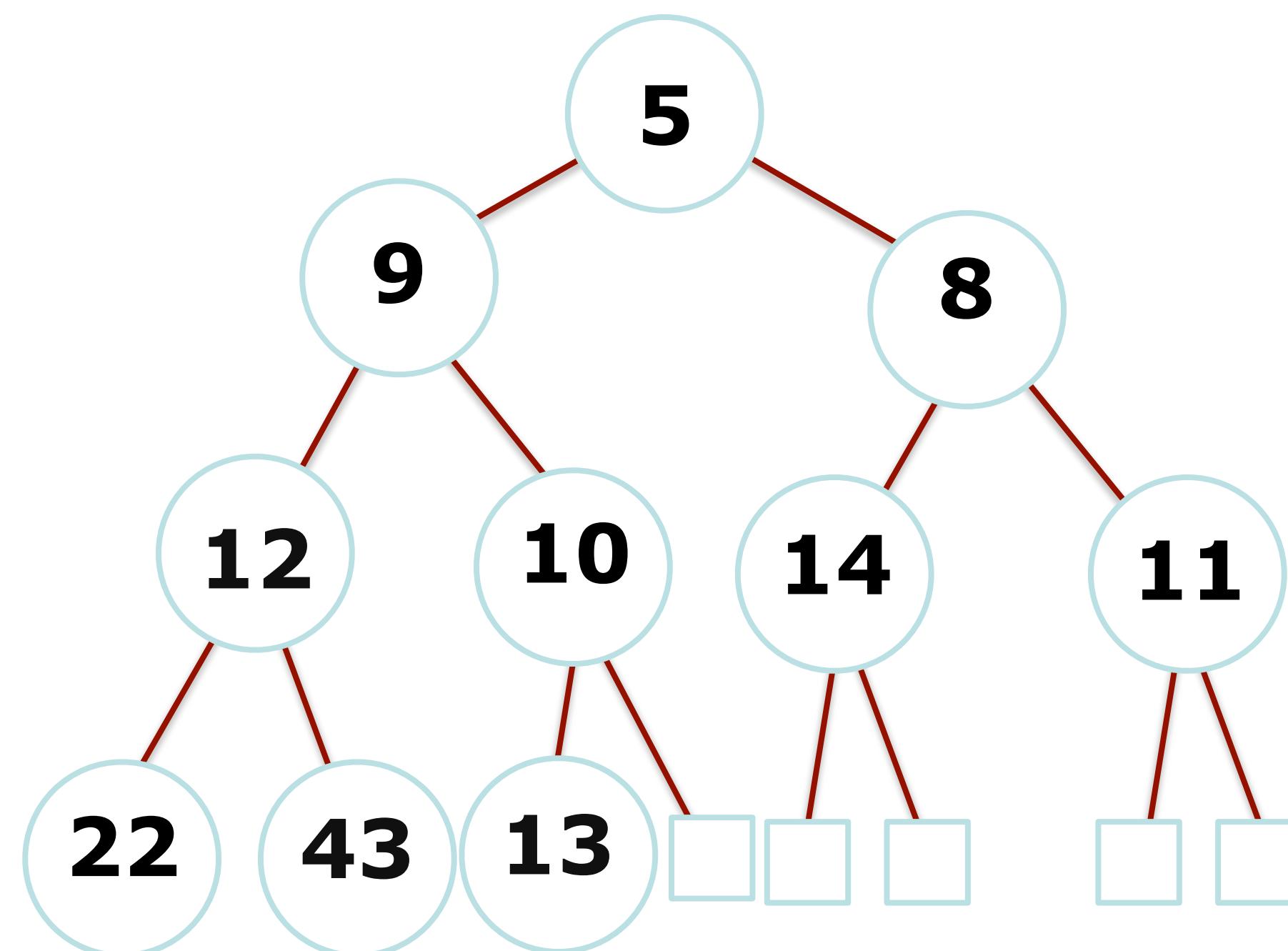


Heap Operations: dequeue()

1. We are removing the root, and we need to retain a complete tree: replace root with last element.
2. **“bubble-down”** or “down-heap” the new root:
 - a. Compare the root with its children, if in correct order, stop.
 - b. If not, swap with smallest child, and repeat step 2.
 - c. Be careful to check whether the children exist (if right exists, left must...)



Heap Operations: dequeue()

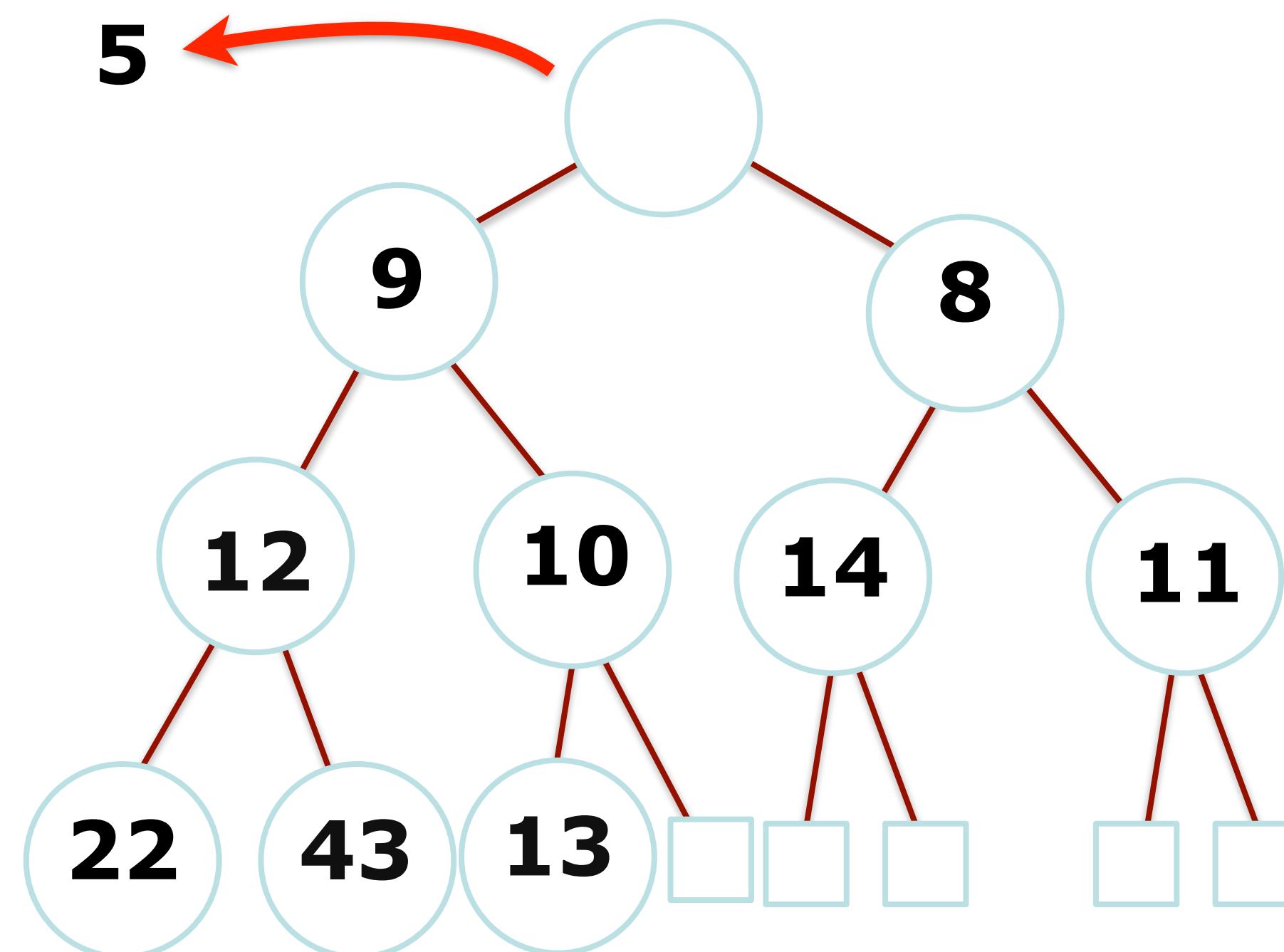


	5	9	8	12	10	14	11	22	43	13		
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



Heap Operations: dequeue()

Remove root (will return at the end)

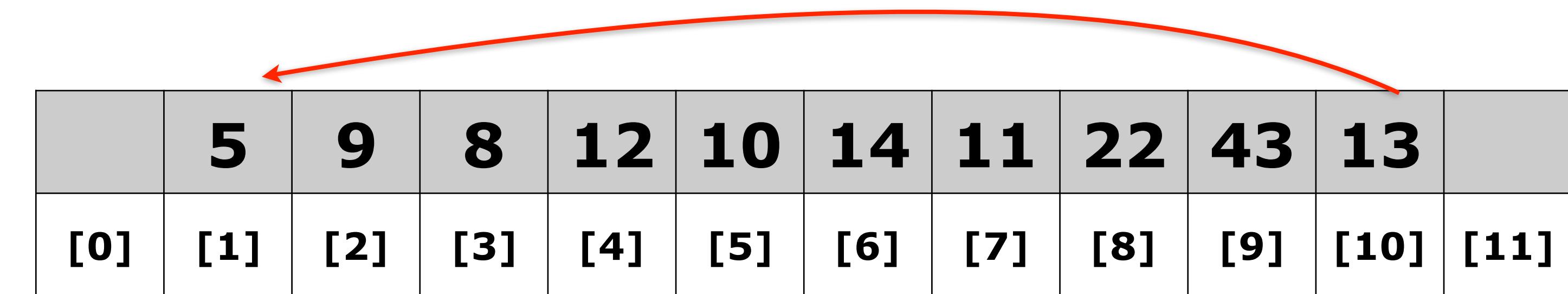
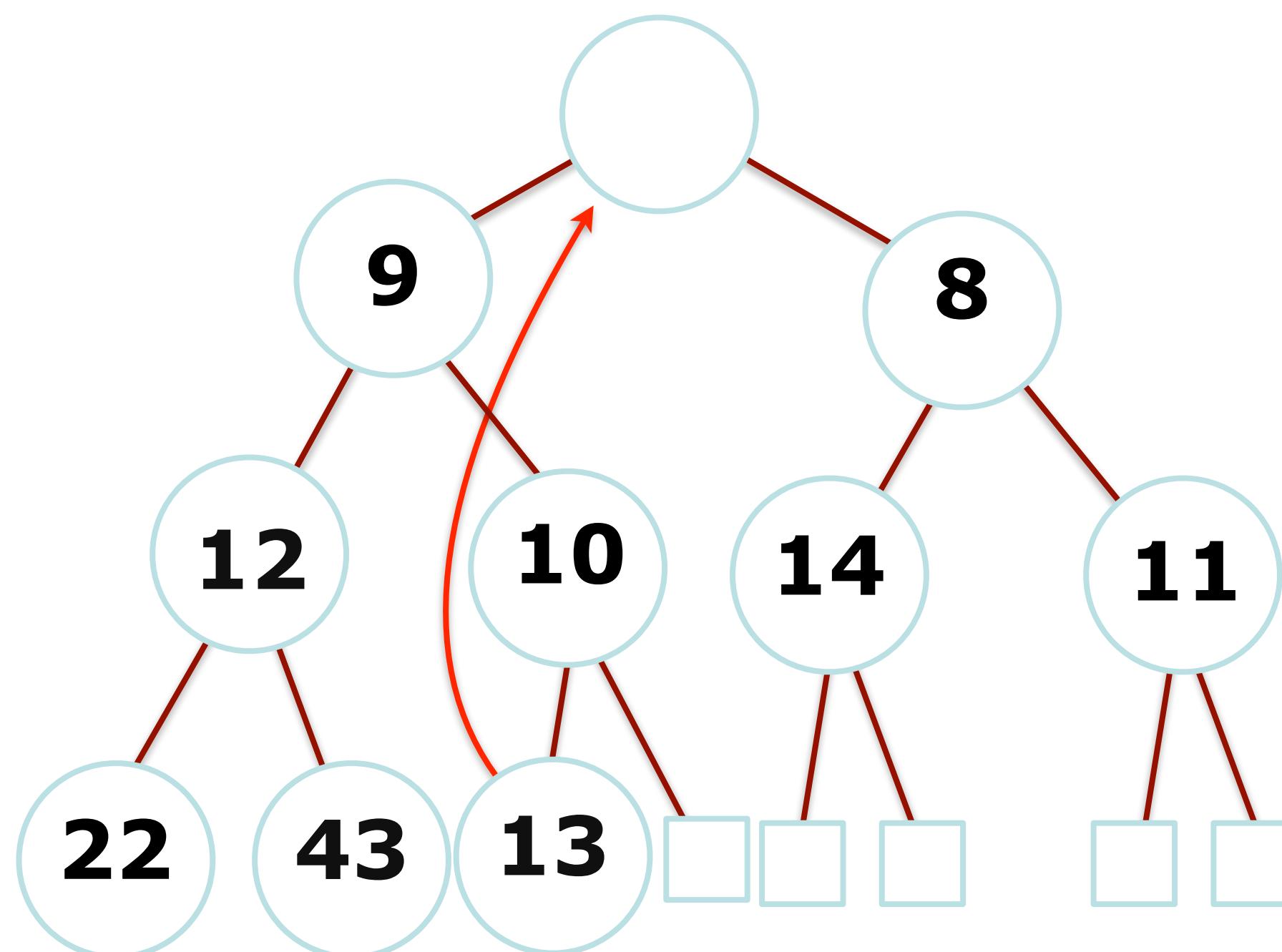


	5	9	8	12	10	14	11	22	43	13		
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



Heap Operations: dequeue()

Move last element (at `heap[heap.size() - 1]`)
to the root (this may be unintuitive!) to begin
bubble-down

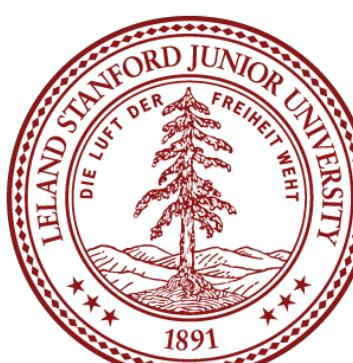
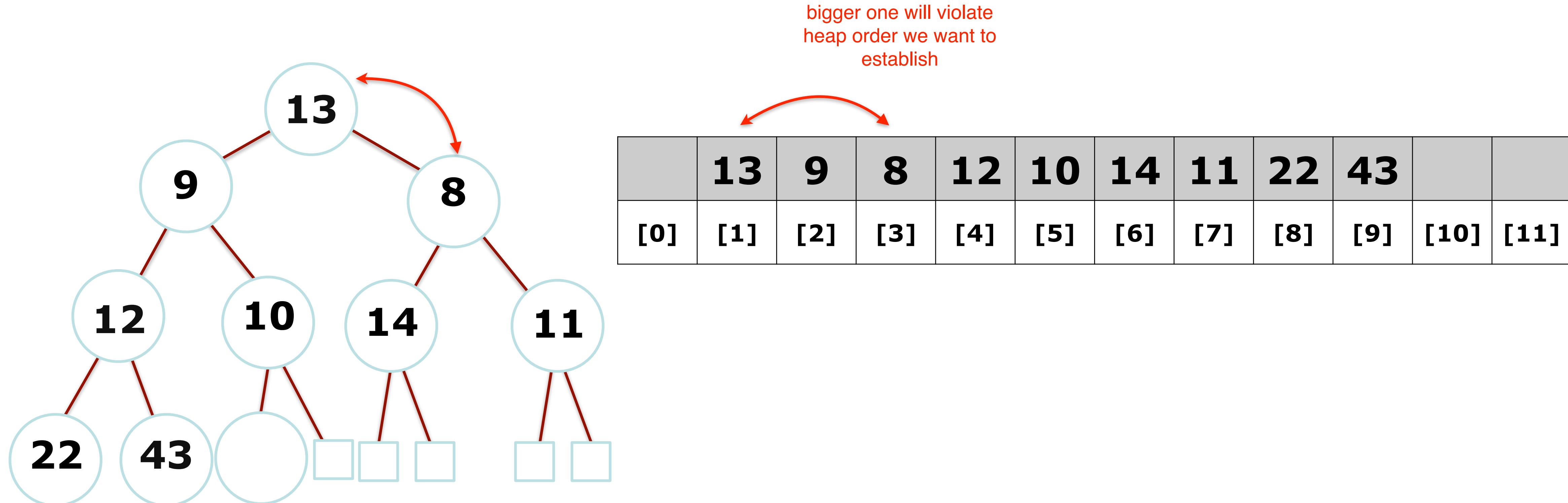


Don't forget to decrease heap size!



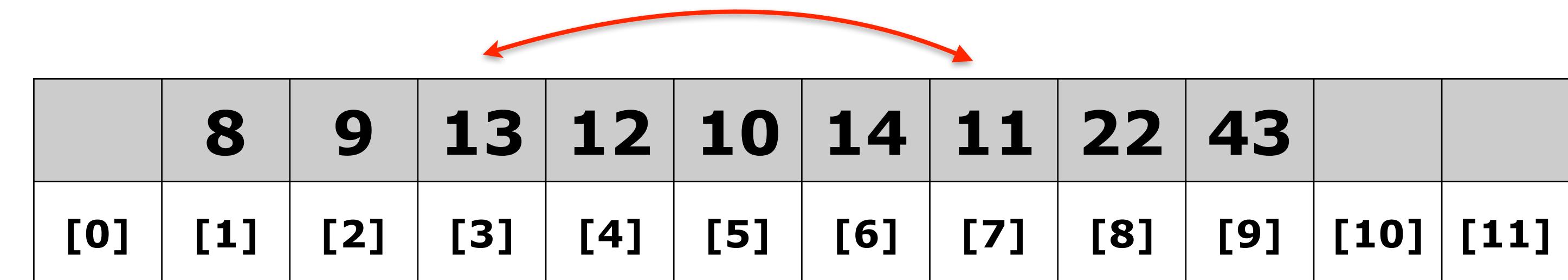
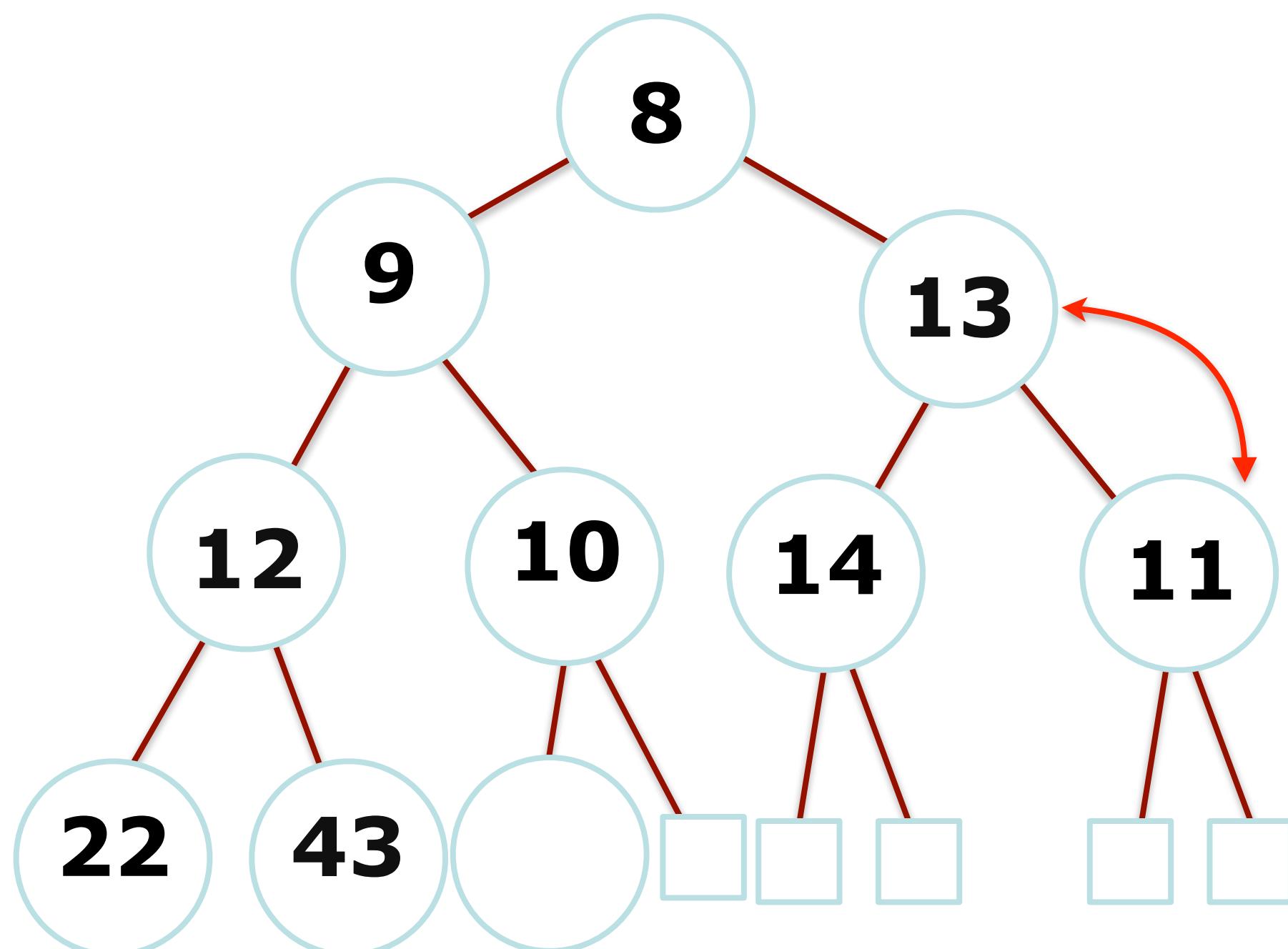
Heap Operations: dequeue()

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



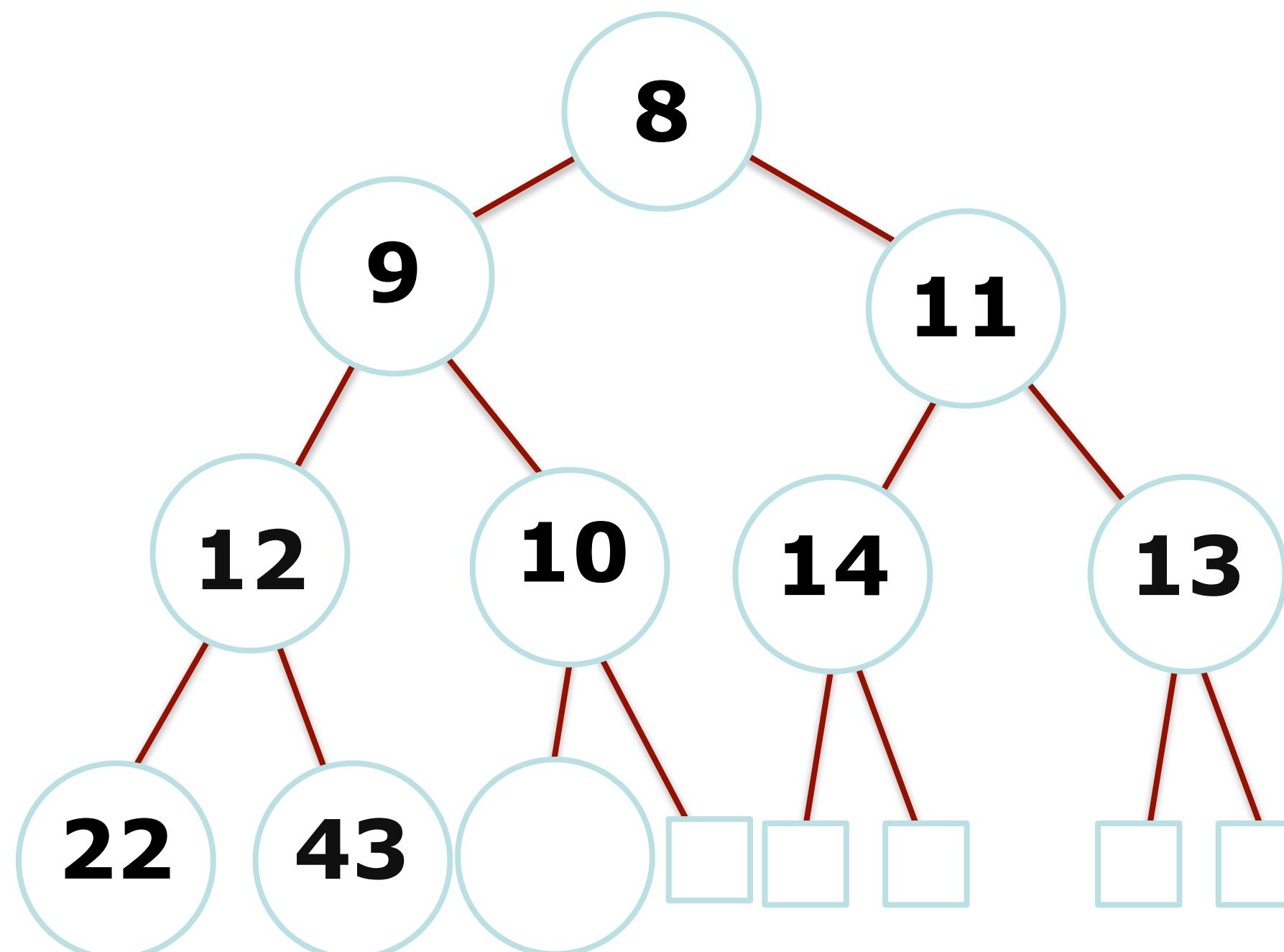
Heap Operations: dequeue()

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



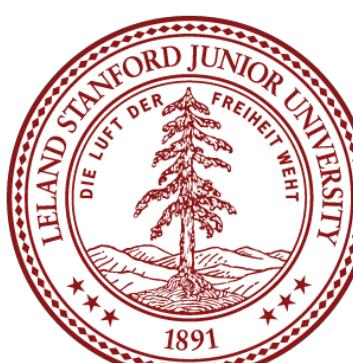
Heap Operations: dequeue()

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



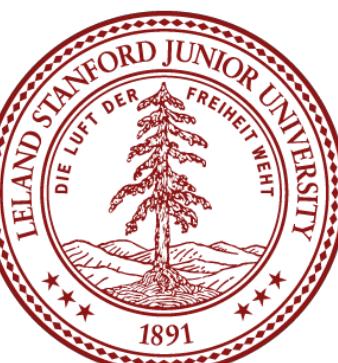
	8	9	11	12	10	14	13	22	43			
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Complexity? $O(\log n)$ - yay!



Heaps in Real Life

- Heapsort (see extra slides)
- Google Maps -- finding the shortest path between places
- All priority queue situations
- Kernel process scheduling
- Event simulation
- Huffman coding



Heap Operations: building a heap from scratch

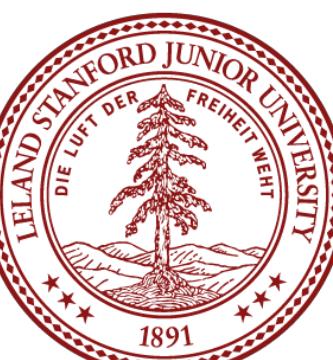
What is the best method for building a heap from scratch (buildHeap())

14, 9, 13, 43, 10, 8, 11, 22, 12

We could insert each in turn.

An insertion takes $O(\log n)$, and we have to insert n elements

Big O? $O(n \log n)$



Heap Operations: building a heap from scratch

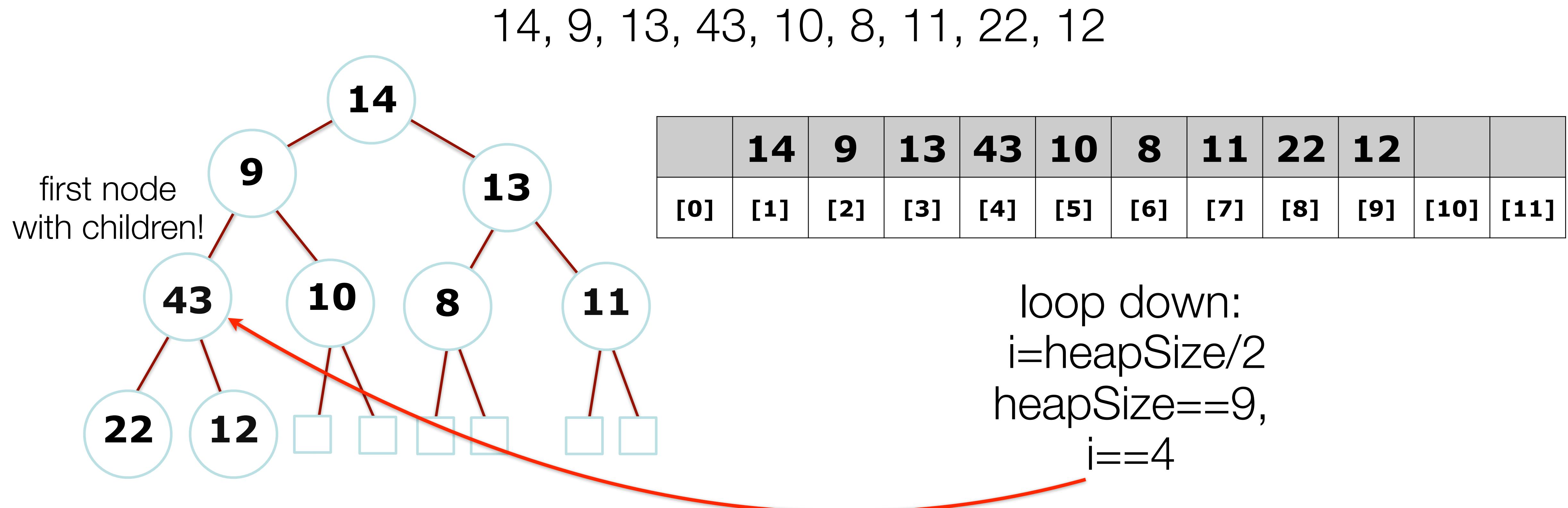
There is a better way: **heapify()**

1. Insert all elements into a binary tree in original order ($O(n)$)
2. Starting from the lowest completely filled level at the first node with children (e.g., at position $n/2$), down-heap each element (also $O(n)$) to heapify the whole tree).

```
for (int i=heapSize/2;i>0;i--) {  
    downHeap(i) ;  
}
```

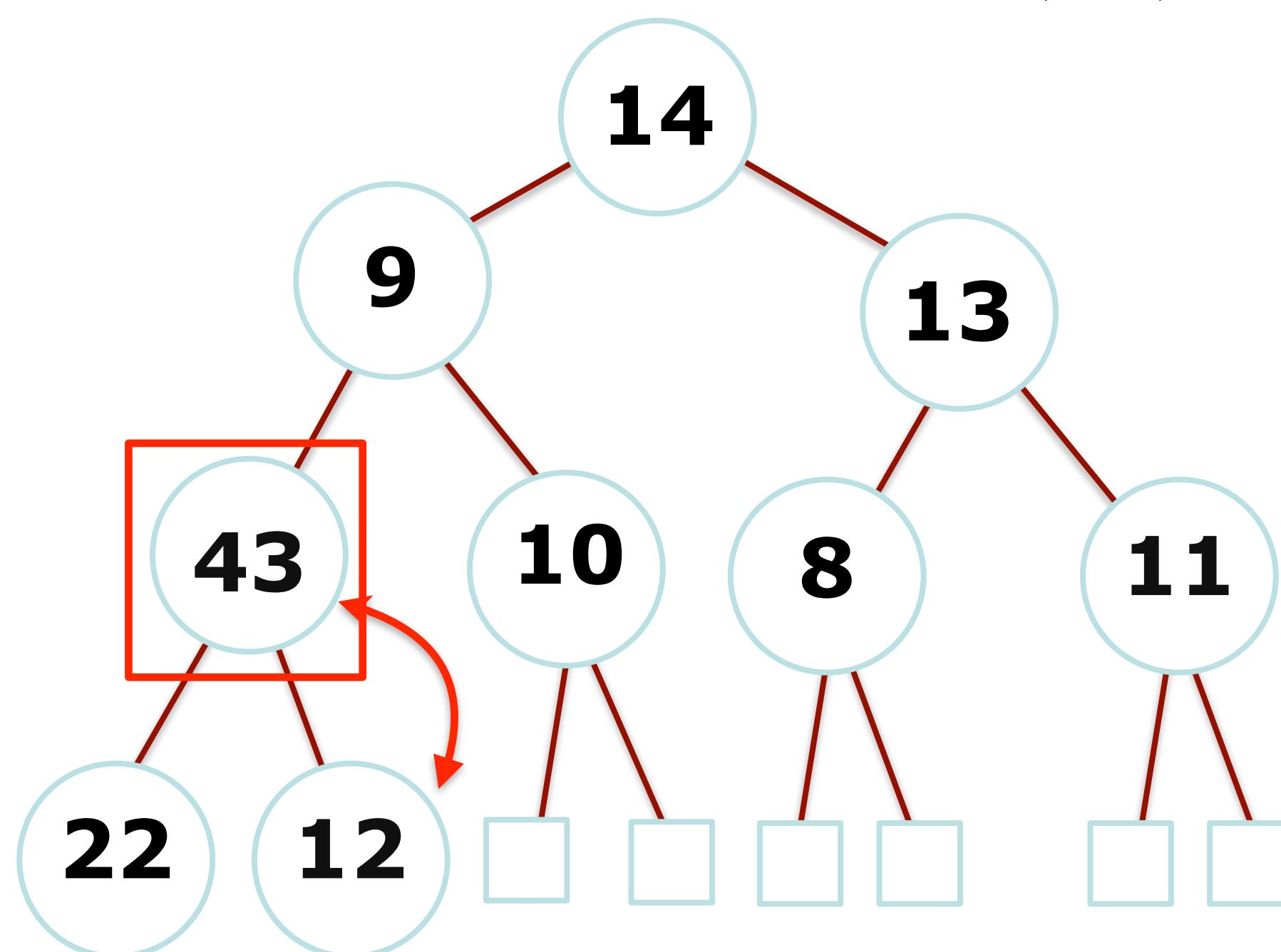


Heap Operations: building a heap from scratch



Heap Operations: building a heap from scratch

14, 9, 13, 43, 10, 8, 11, 22, 12



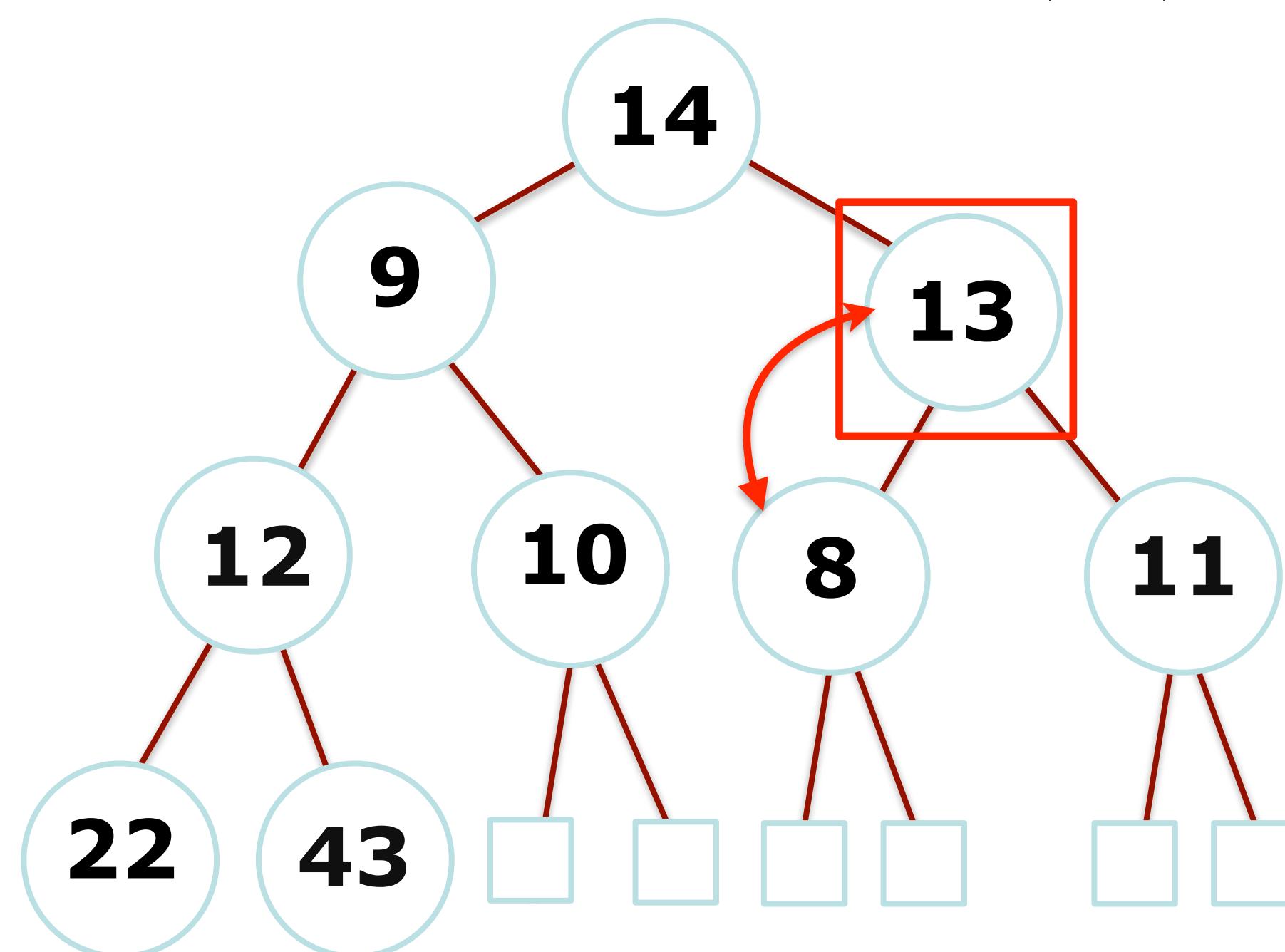
	14	9	13	43	10	8	11	22	12		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

$i=4$



Heap Operations: building a heap from scratch

14, 9, 13, 43, 10, 8, 11, 22, 12

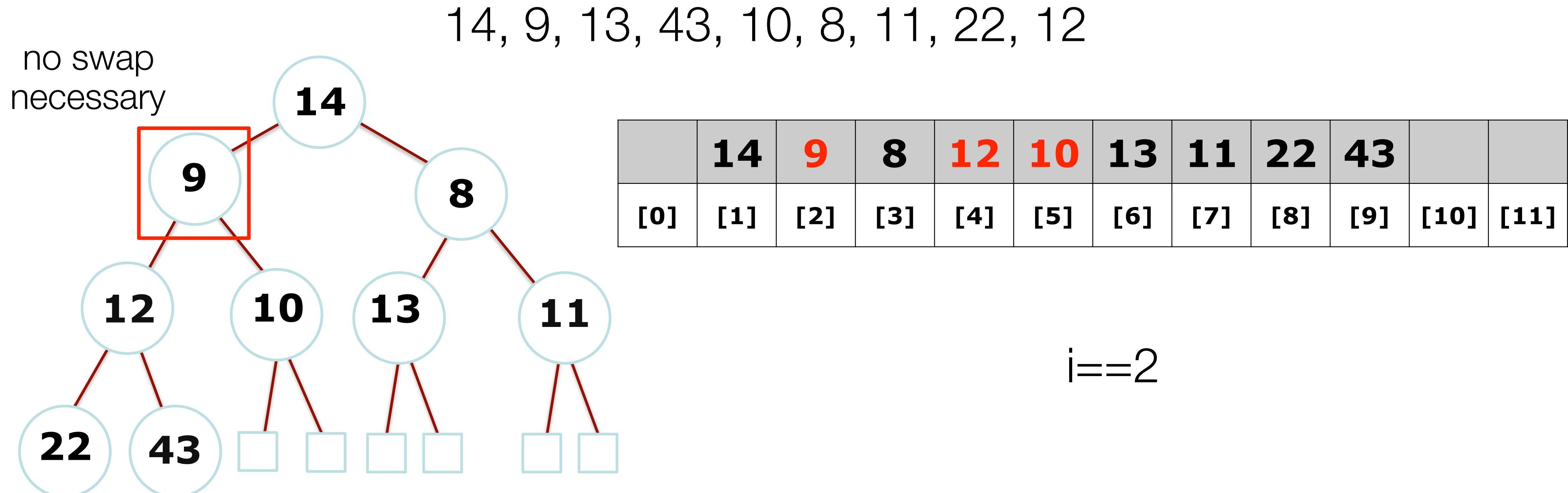


	14	9	13	12	10	8	11	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

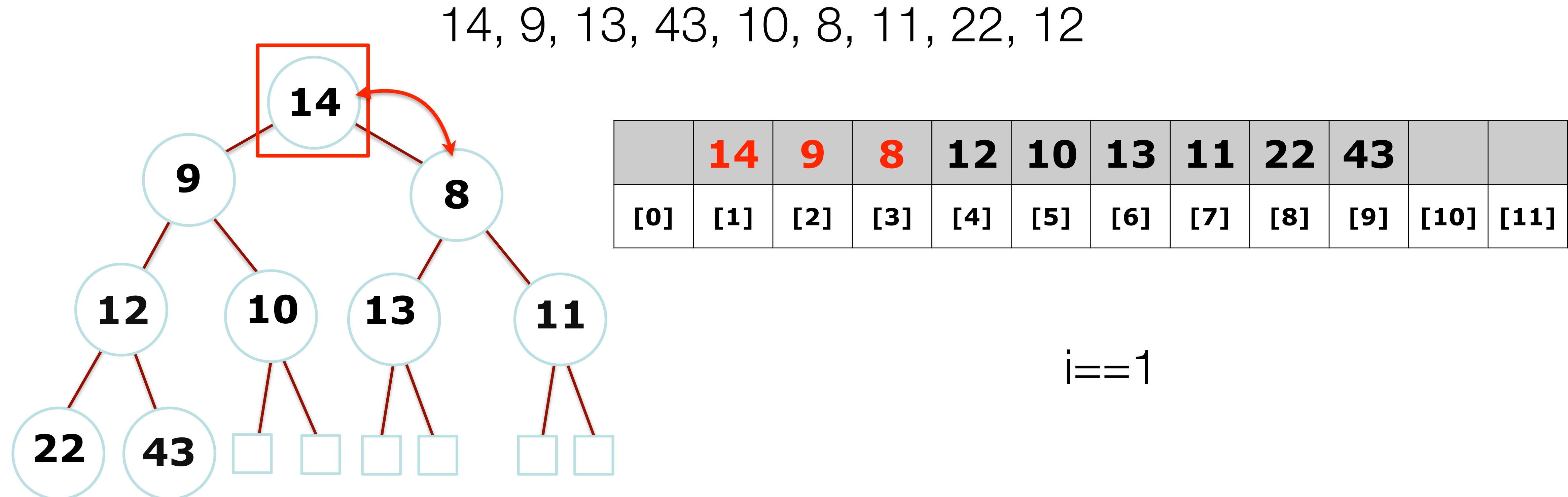
$i == 3$



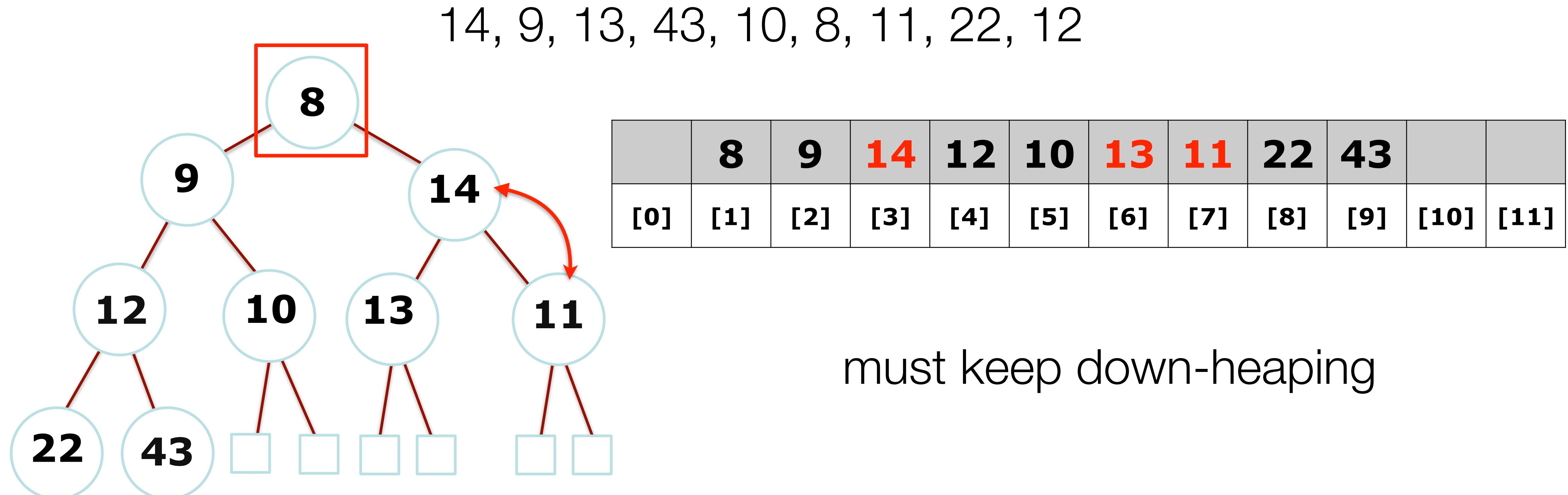
Heap Operations: building a heap from scratch



Heap Operations: building a heap from scratch

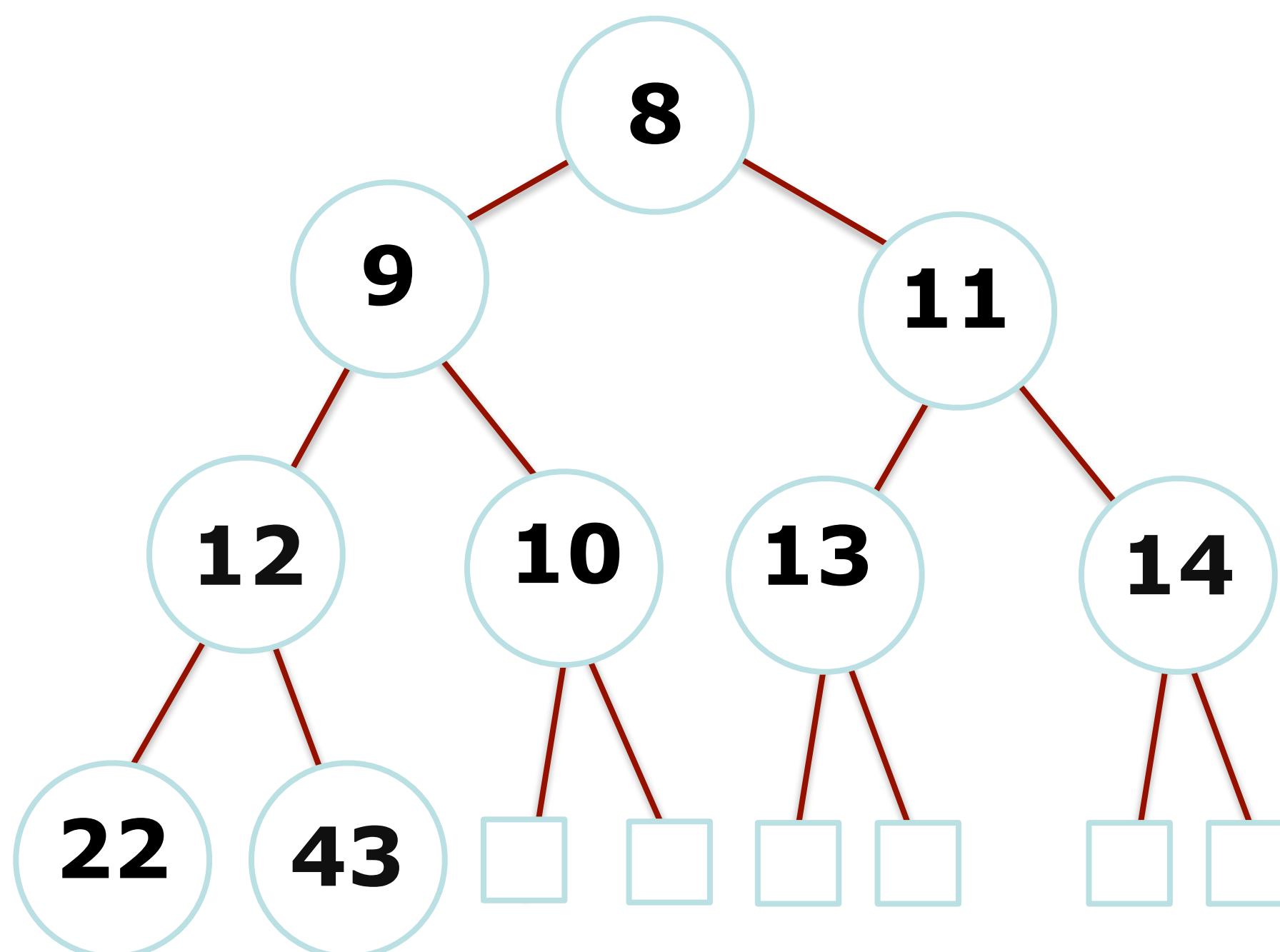


Heap Operations: building a heap from scratch



Heap Operations: building a heap from scratch

14, 9, 13, 43, 10, 8, 11, 22, 12



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

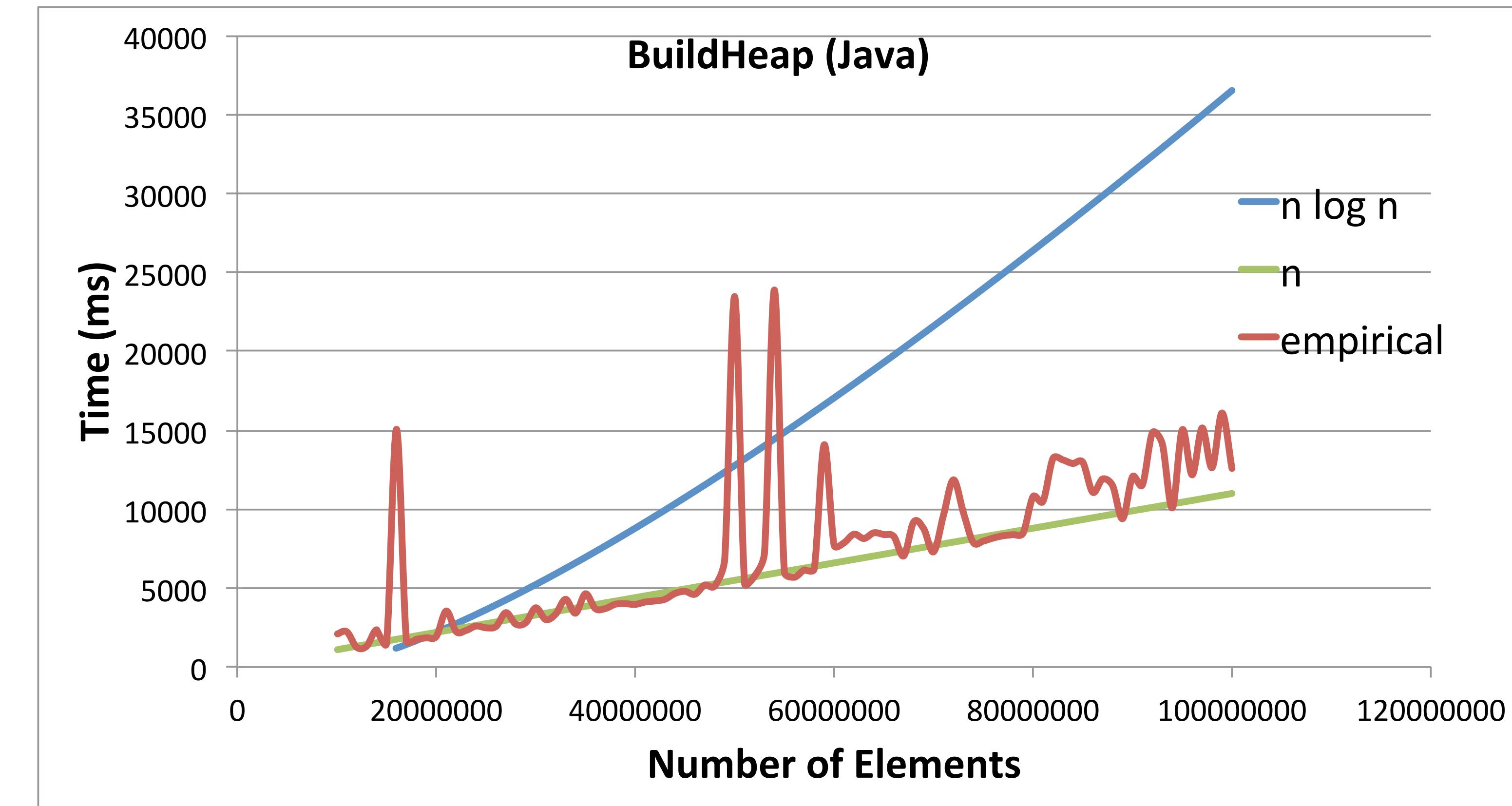
Done!

We now have a proper min-heap.
Asymptotic complexity – not trivial to
determine, but turns out to be $O(n)$.



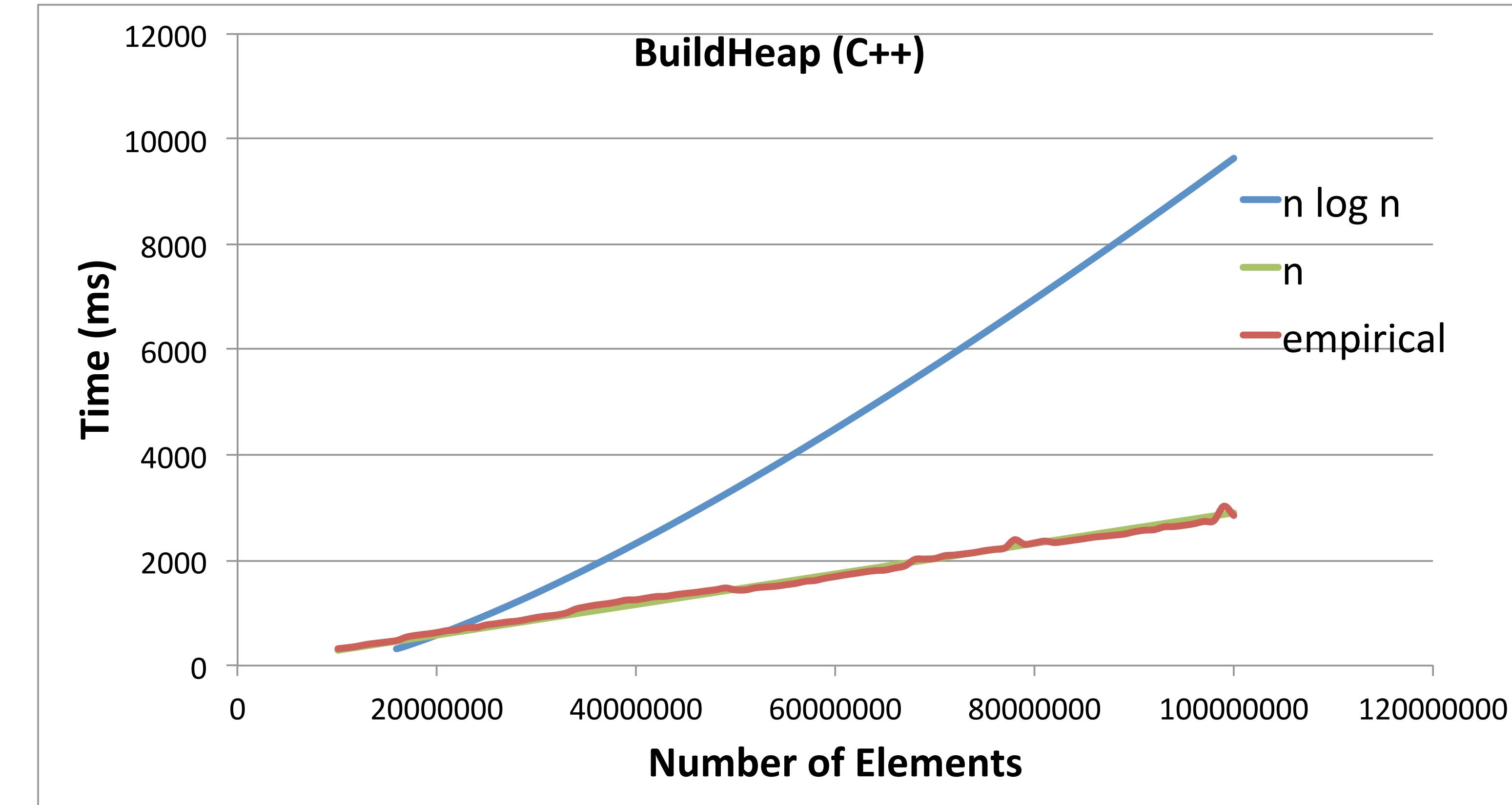
Heap Operations: heaping: empirical

BuildHeap
Empirical Results
(Java)



Heap Operations: heaping: empirical

BuildHeap
Empirical Results
(C++)



References and Advanced Reading

• **References:**

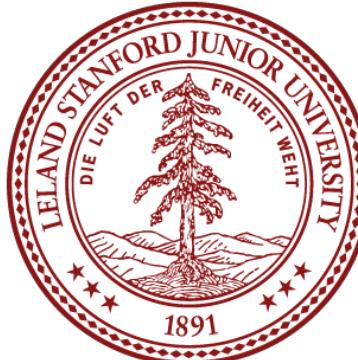
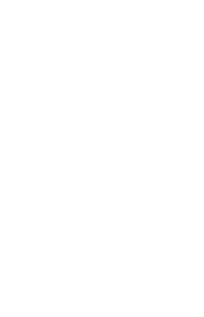
- Priority Queues, Wikipedia: http://en.wikipedia.org/wiki/Priority_queue
- YouTube on Priority Queues: https://www.youtube.com/watch?v=gJc-J7K_P_w
- http://en.wikipedia.org/wiki/Binary_heap (excellent)
- <http://www.cs.usfca.edu/~galles/visualization/Heap.html> (excellent visualization)
- Another explanation online: <http://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html> (excellent)

• **Advanced Reading:**

- A great online explanation of asymptotic complexity of a heap: <http://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf>
- YouTube video with more detail and math: <https://www.youtube.com/watch?v=B7hVxCmfPtM> (excellent, mostly max heaps)



Extra Slides



- We can perform a full heap sort in place, in $O(n \log n)$ time.
 - First, heapify an array (i.e., call build-heap on an unsorted array)
 - Second, iterate over the array and perform dequeue(), but instead of returning the minimum elements, swap them with the last element (and also decrease heapSize)
- When the iteration is complete, the array will be sorted from low to high priority.

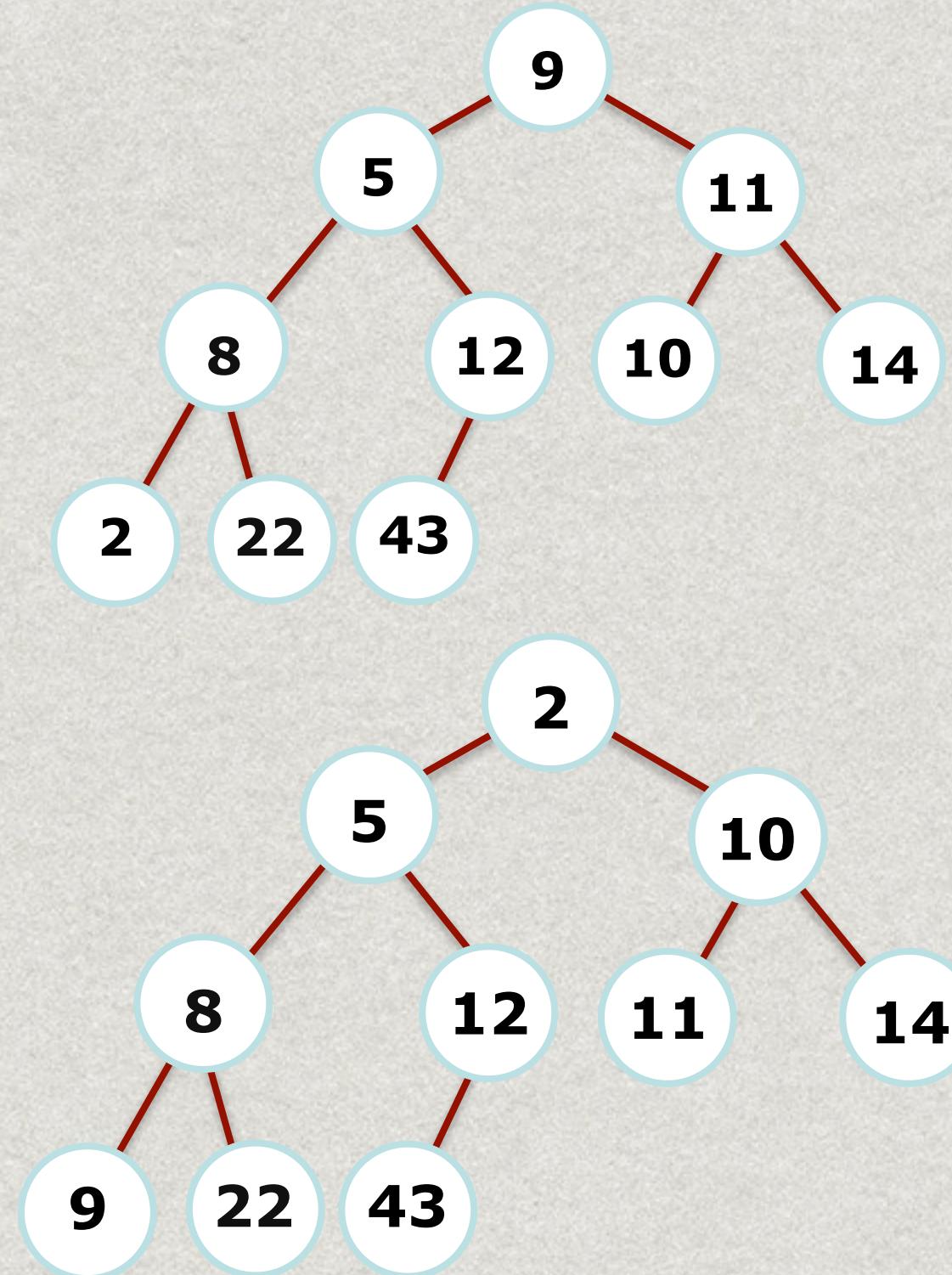
Extras: HeapSort – Heapify first

Unheaped:

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

Heaped:

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

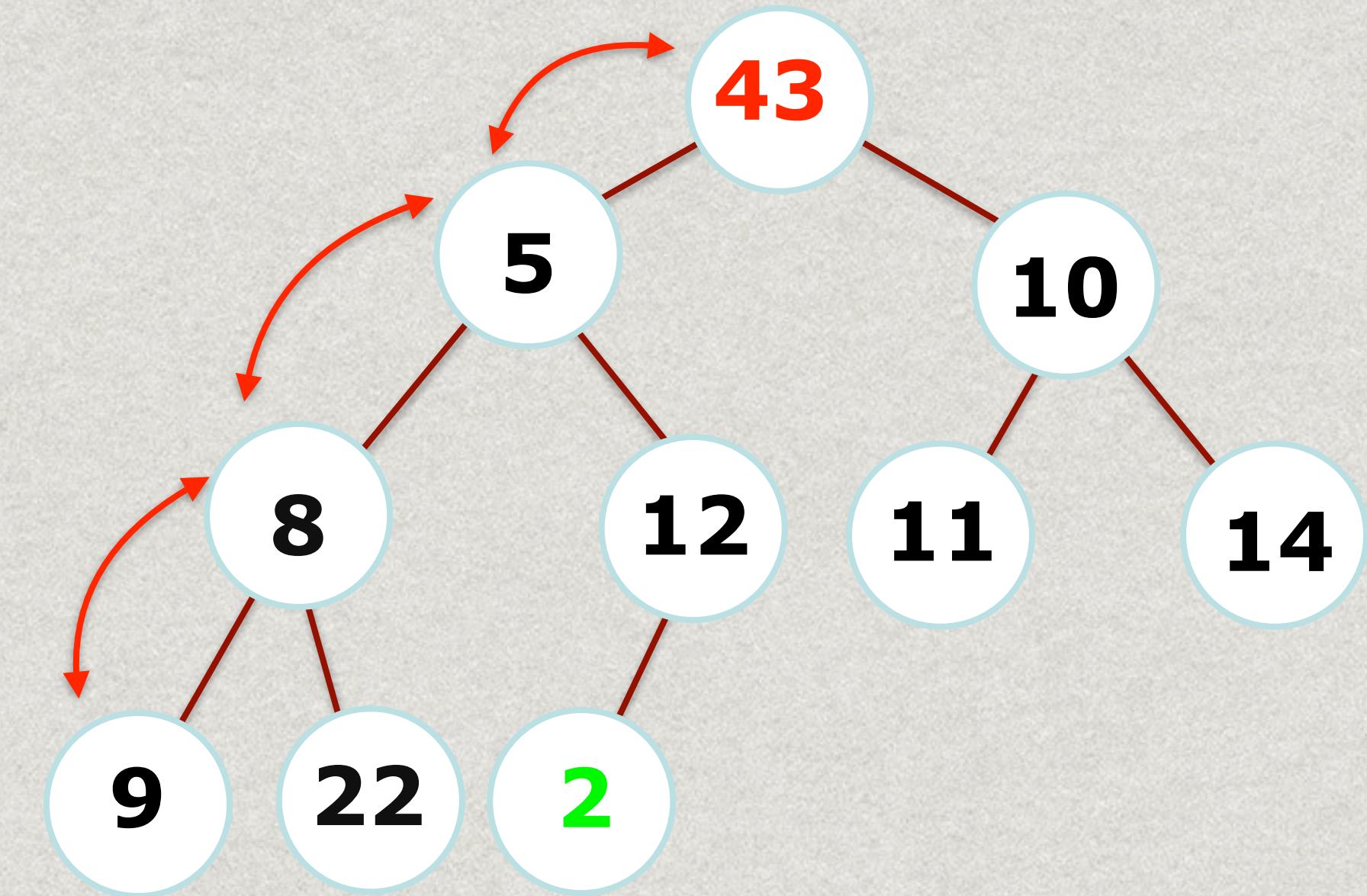


Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

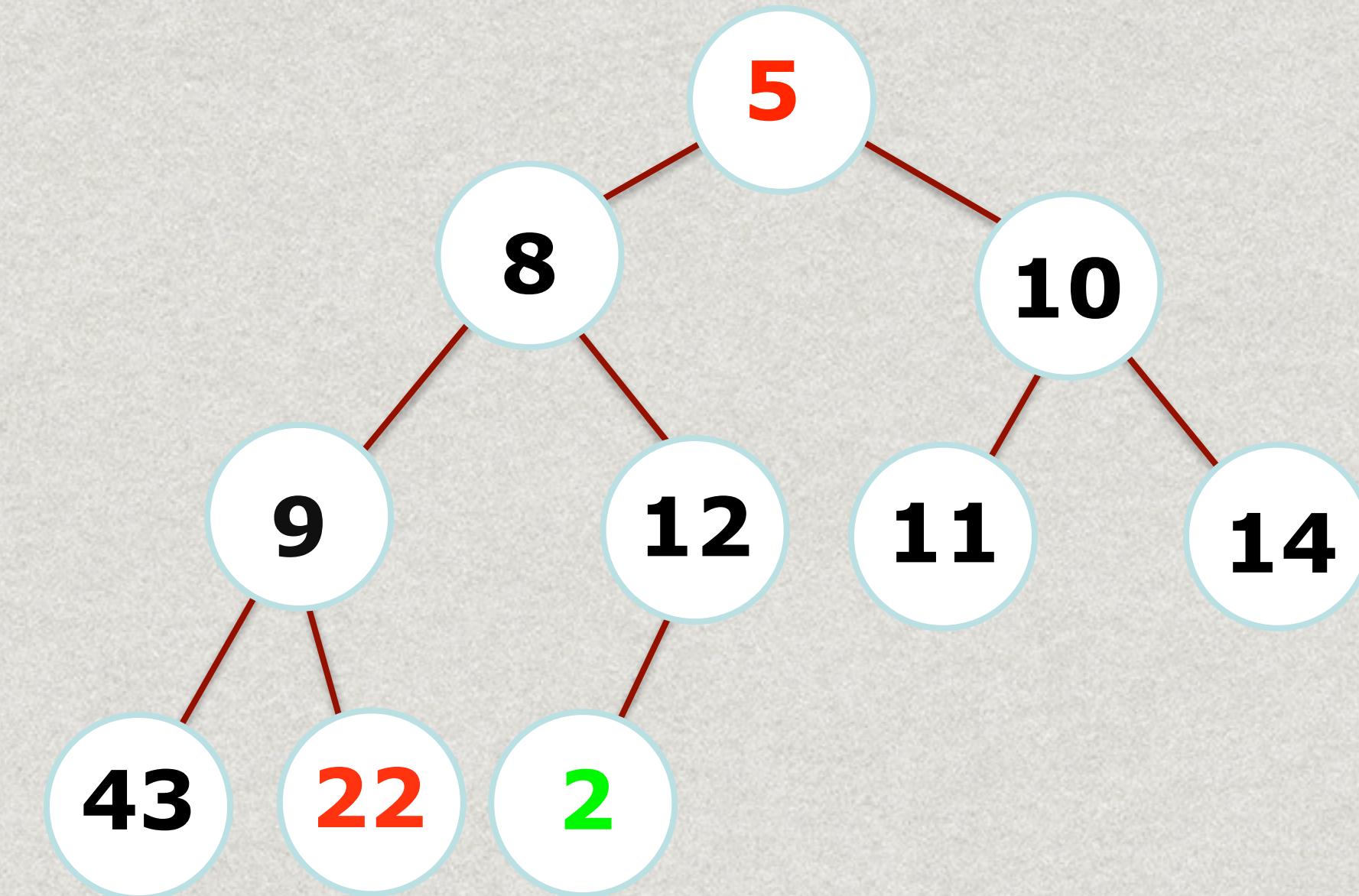
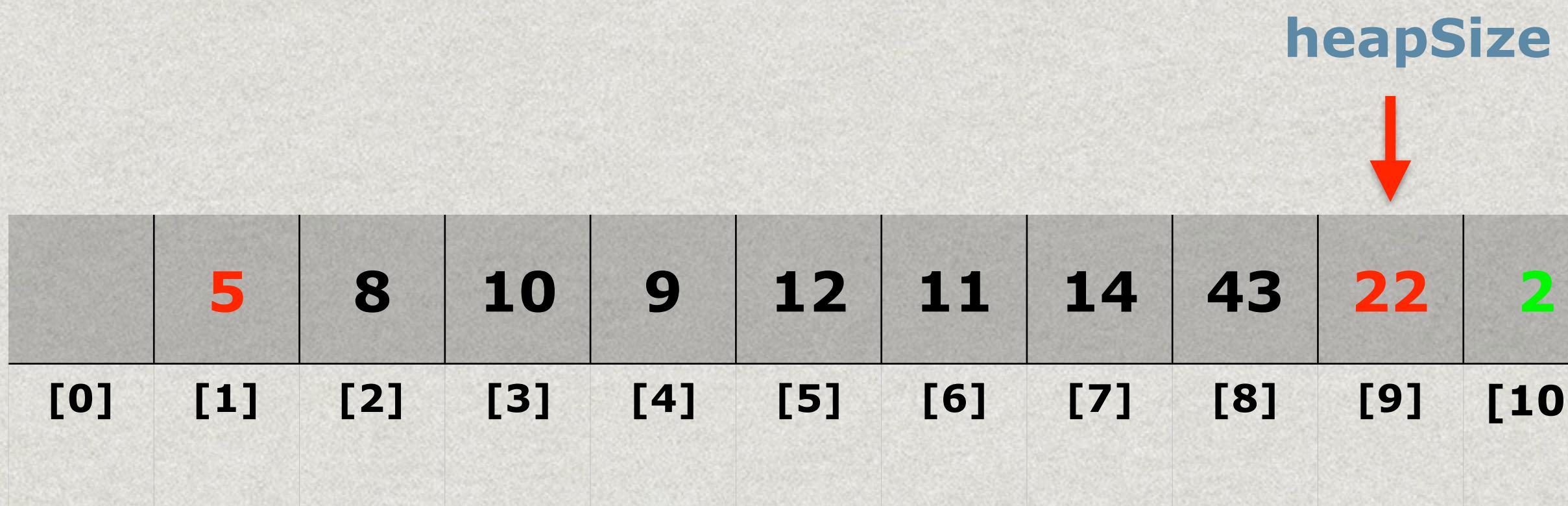
56

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

heapSize

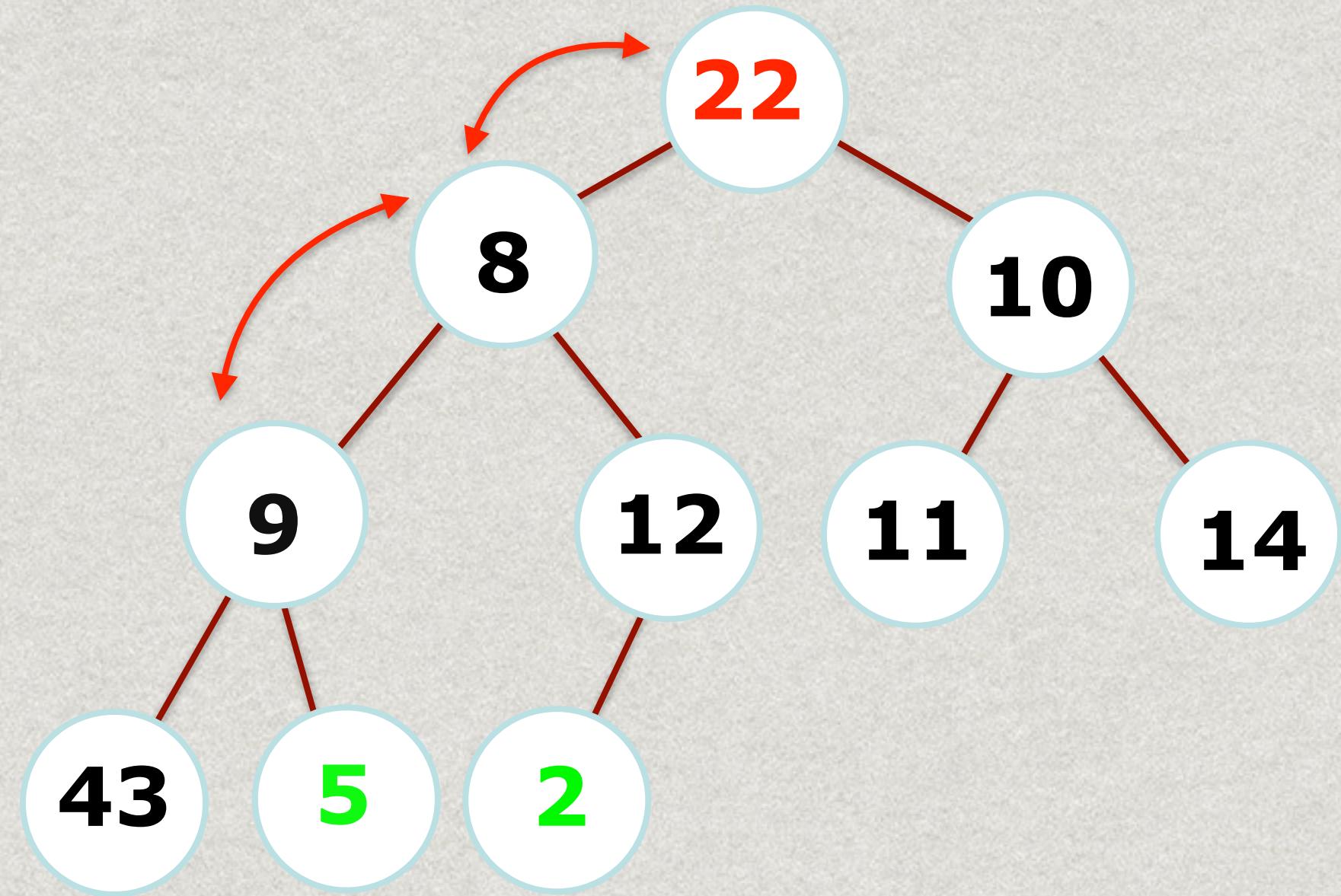
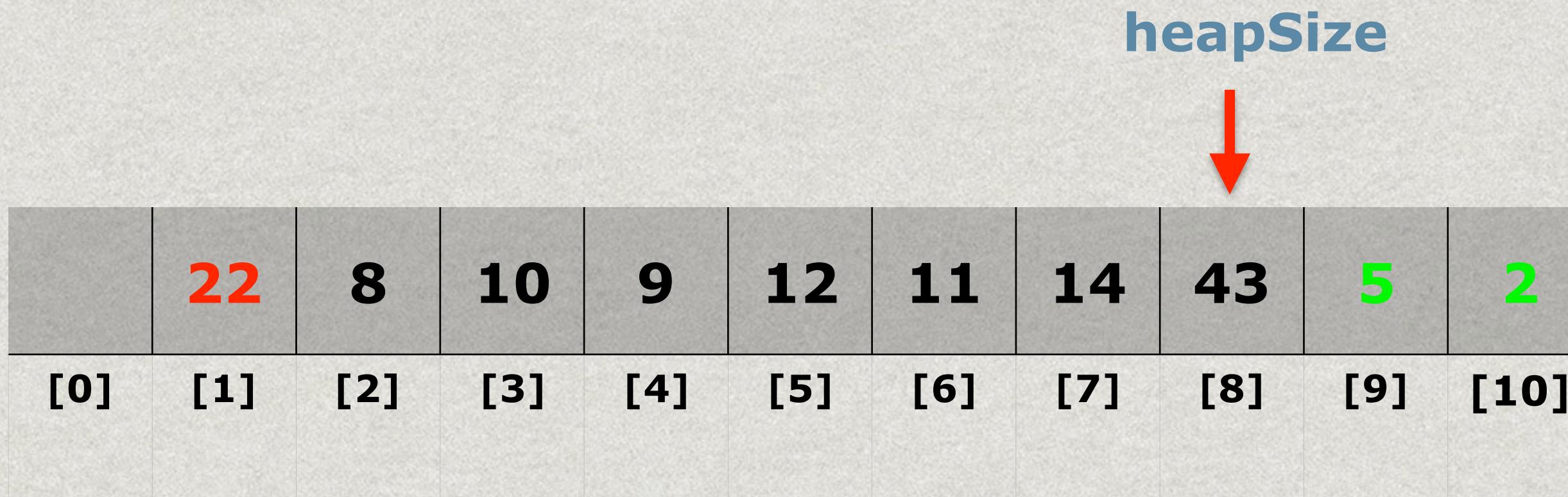


Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping. 57



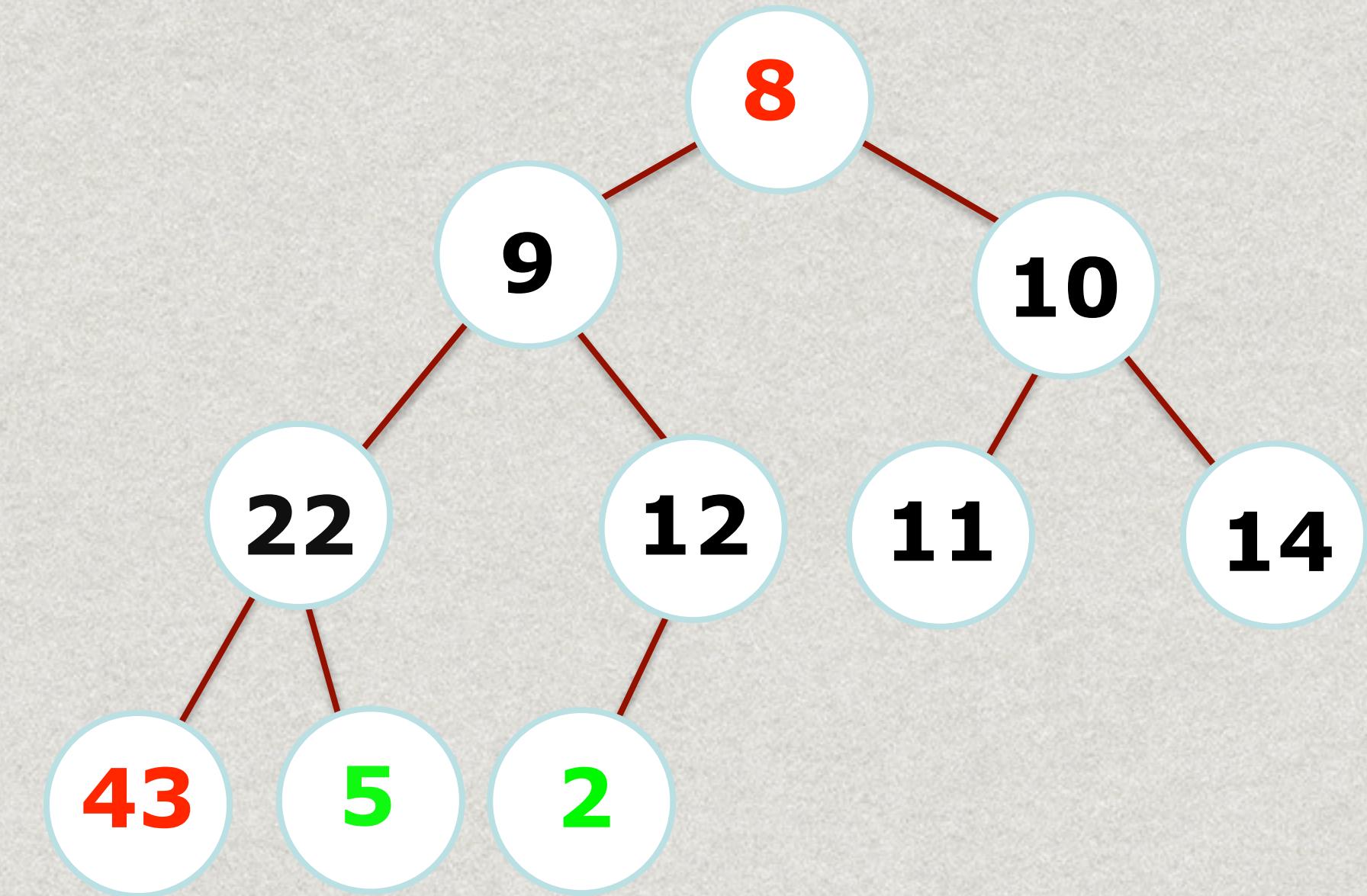
Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

58



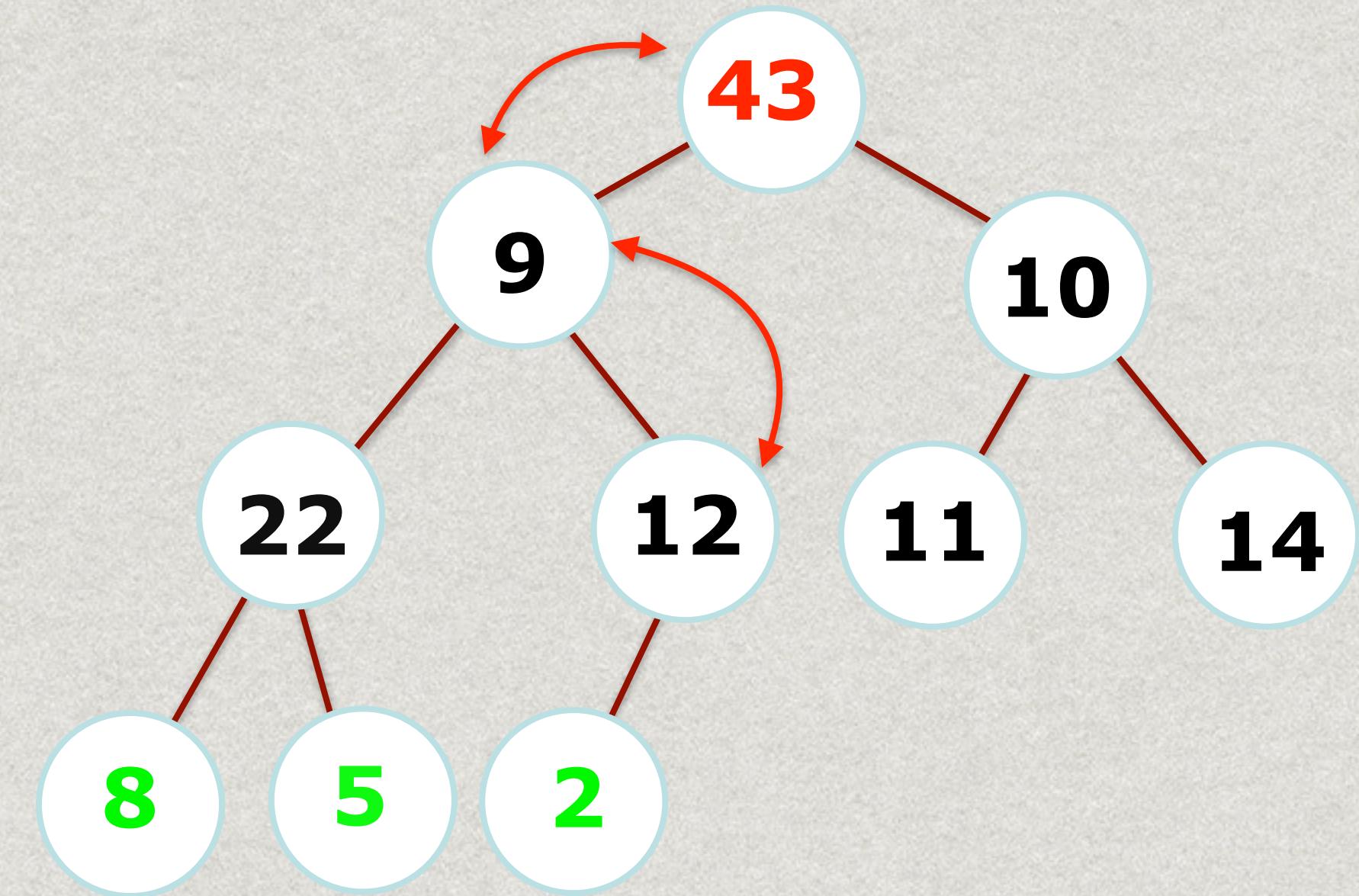
Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

59



Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

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

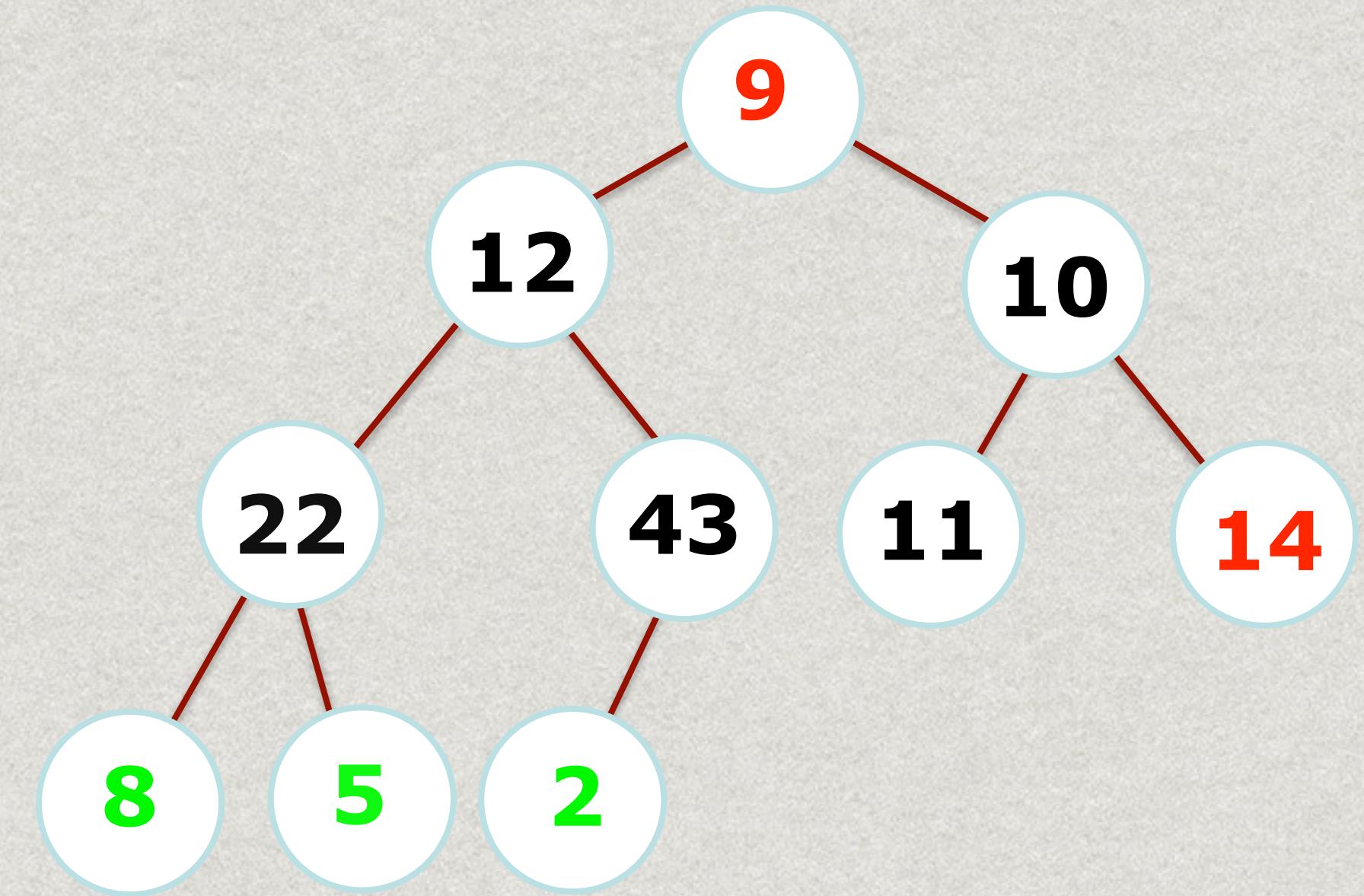


Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

61

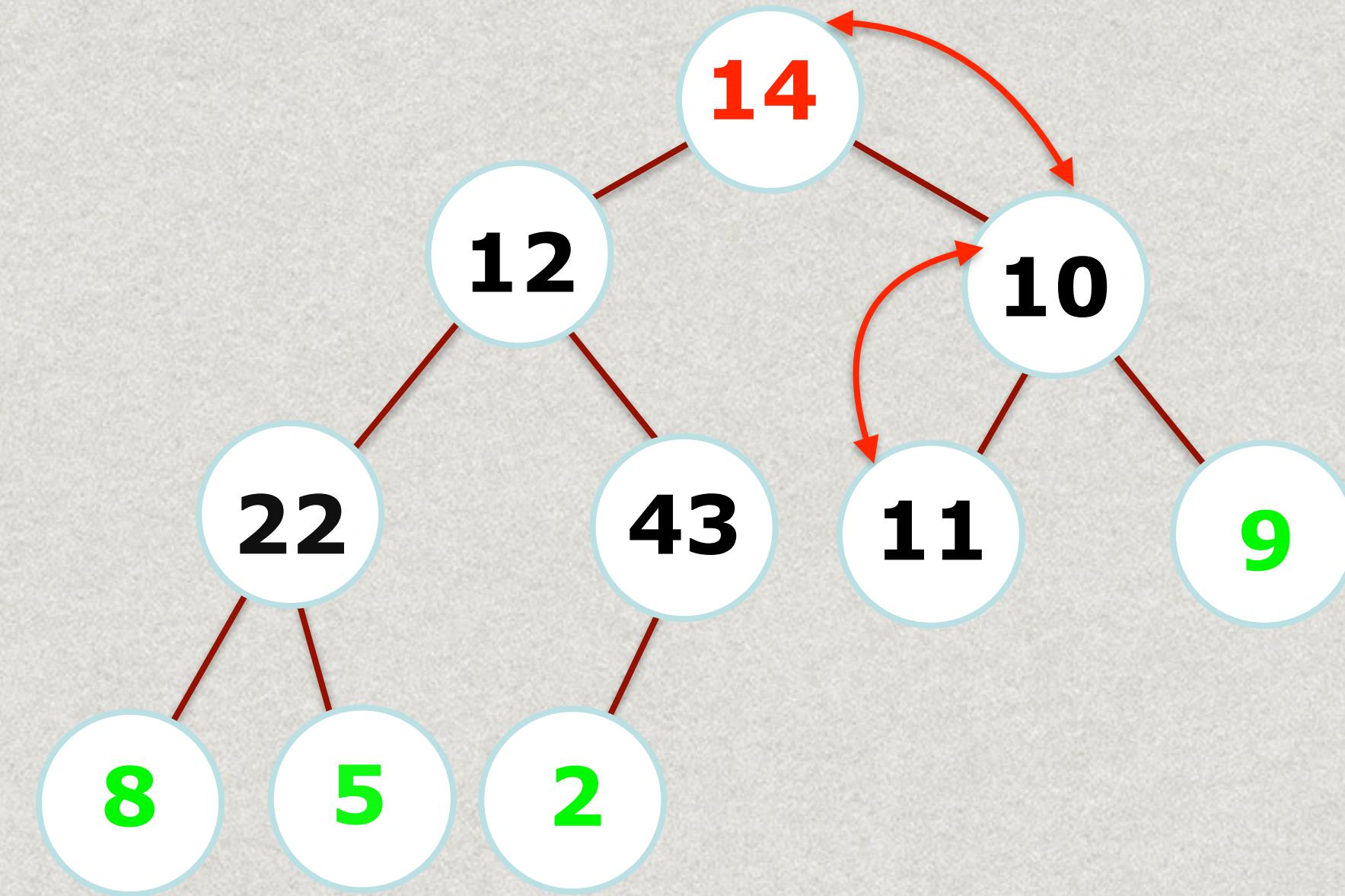
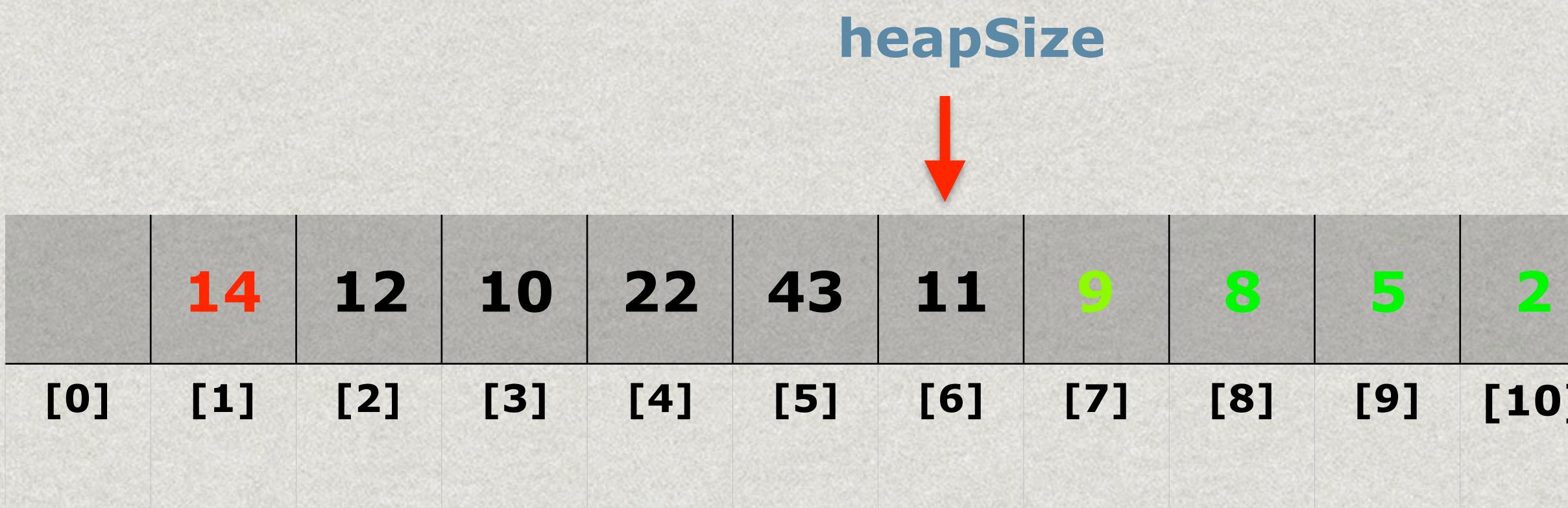
		9	12	10	22	43	11	14	8	5	2	
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	

heapSize



Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

62



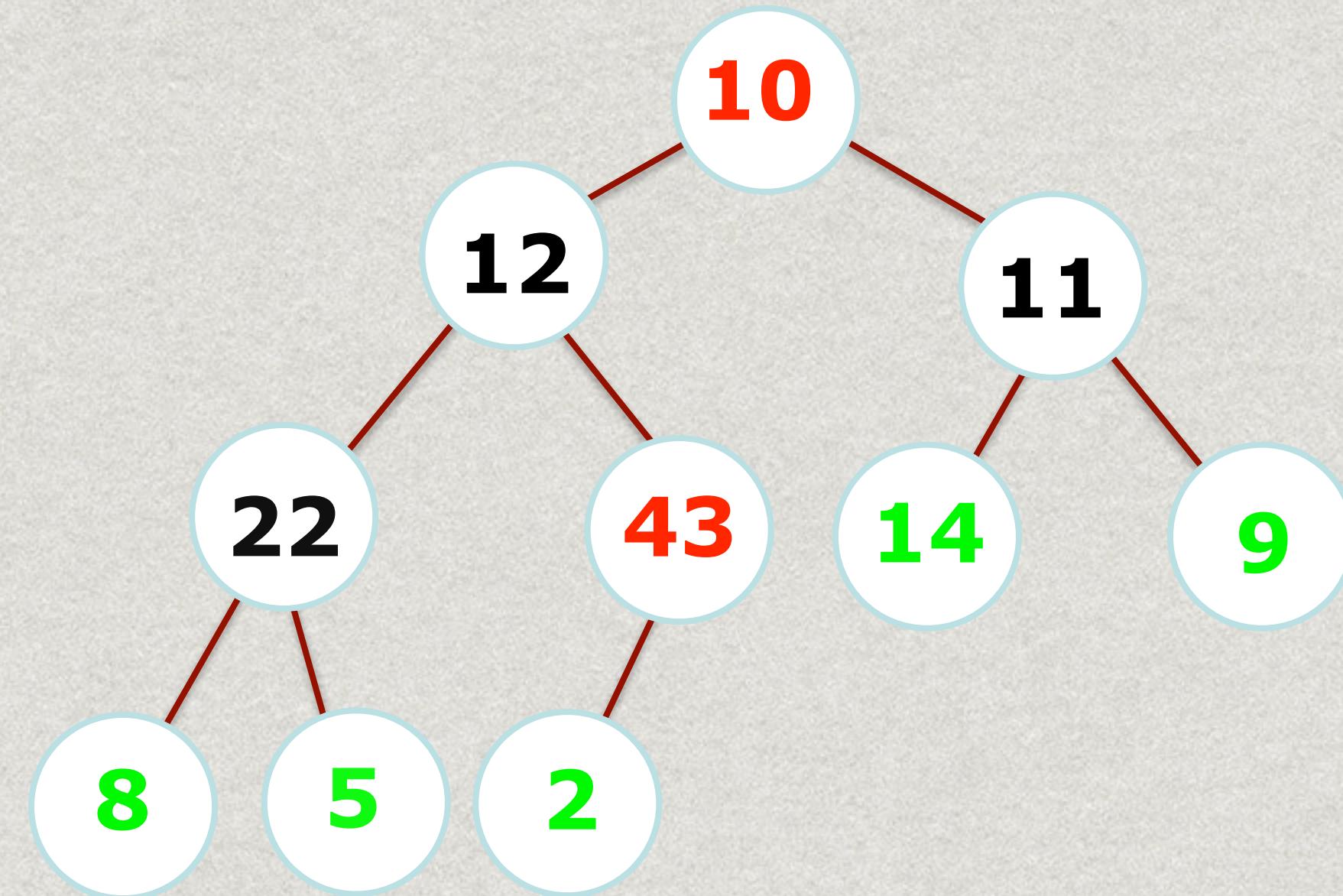
Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

63

heapSize

↓

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

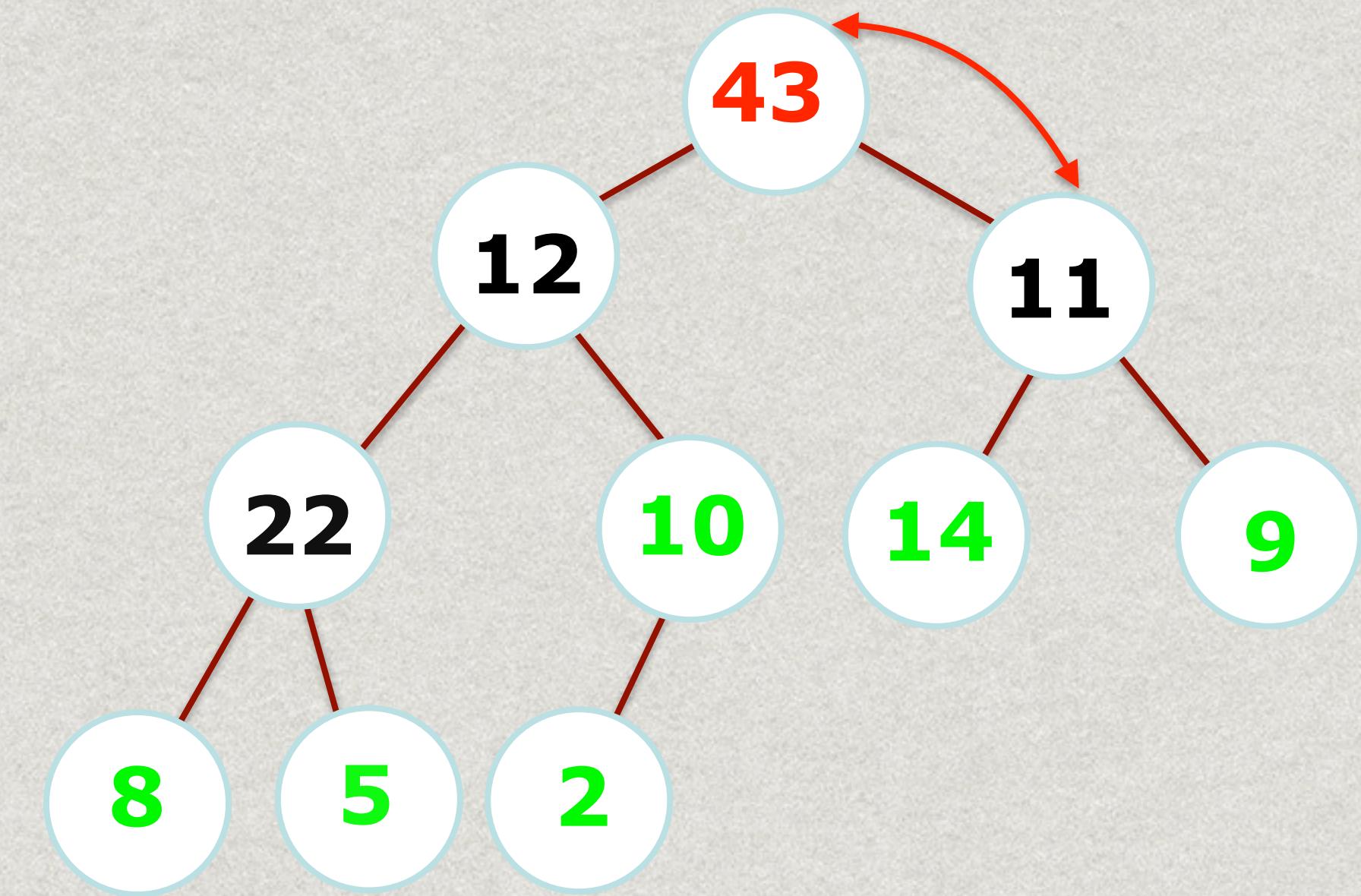


Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

heapSize

↓

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



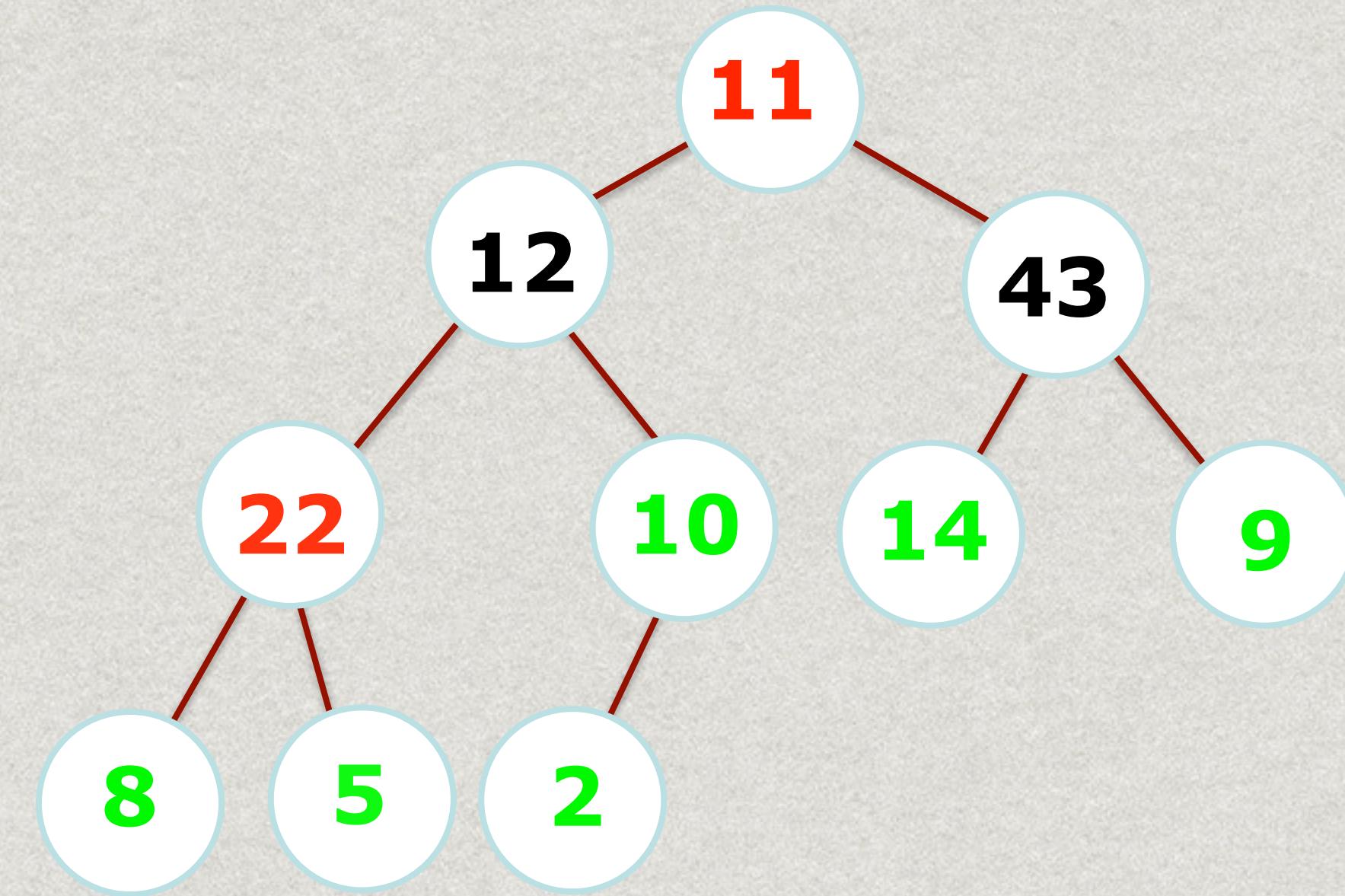
Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

65

heapSize

↓

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



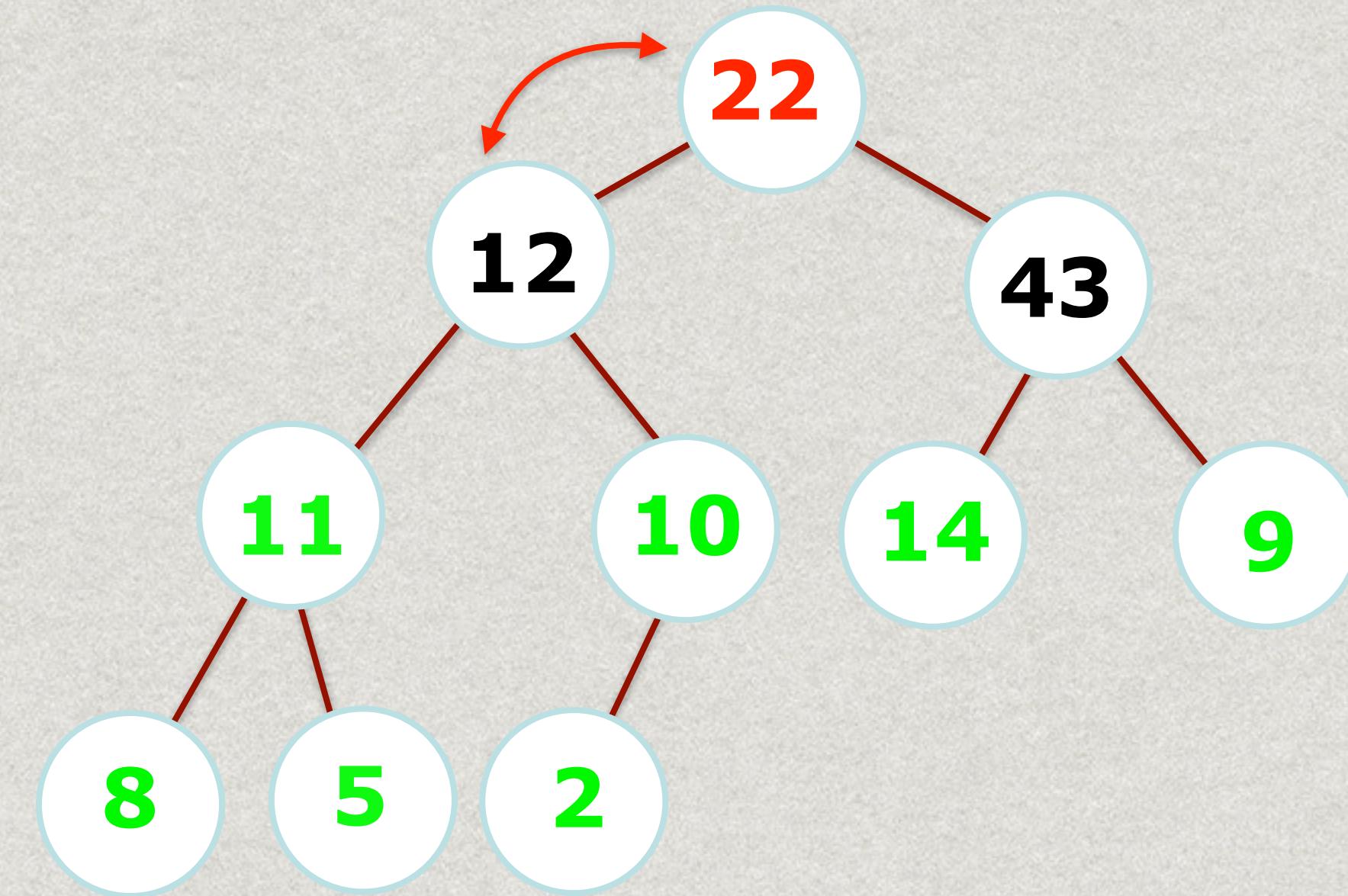
Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

66

heapSize

↓

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

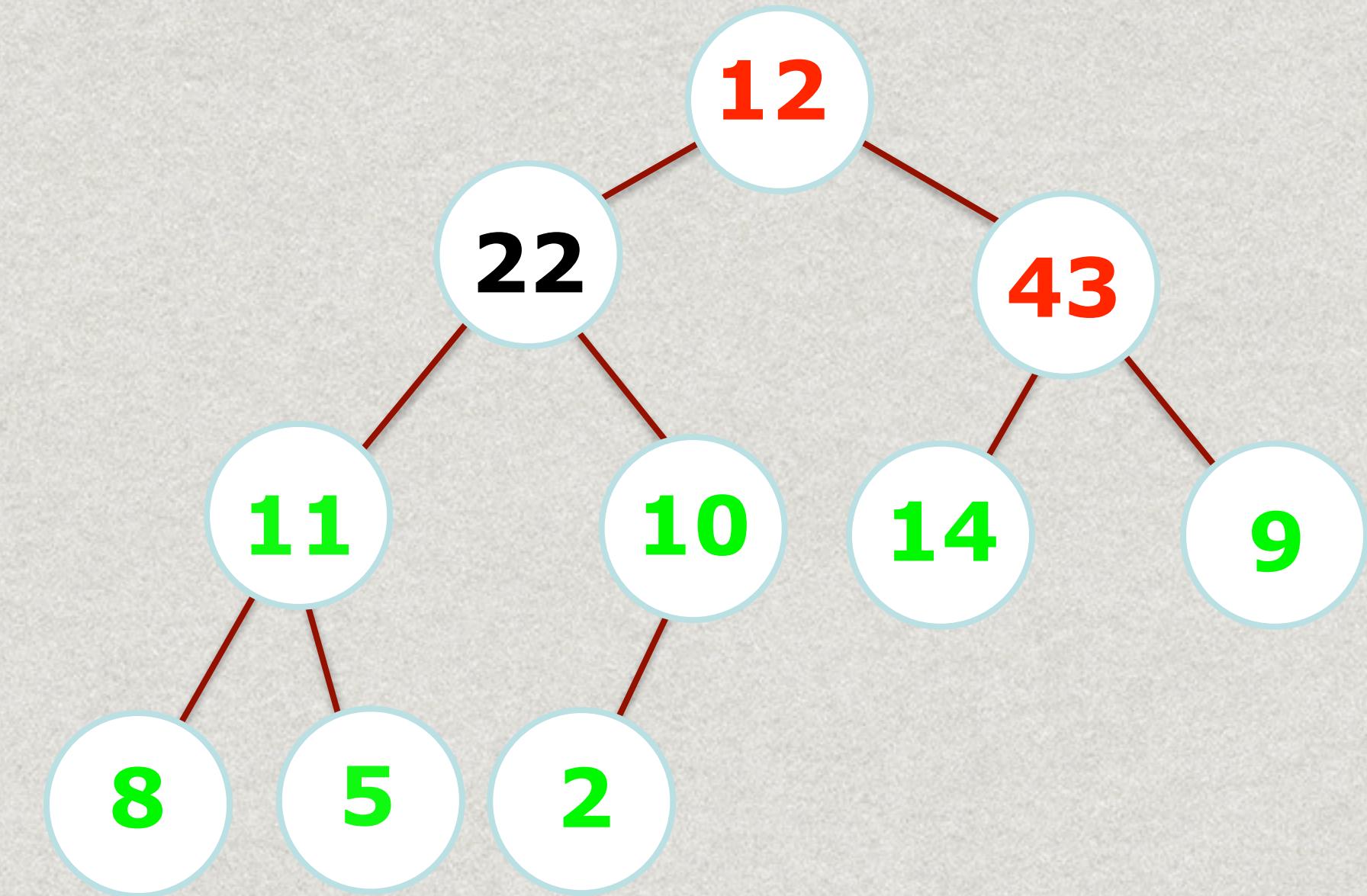


Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping. 67

heapSize

↓

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



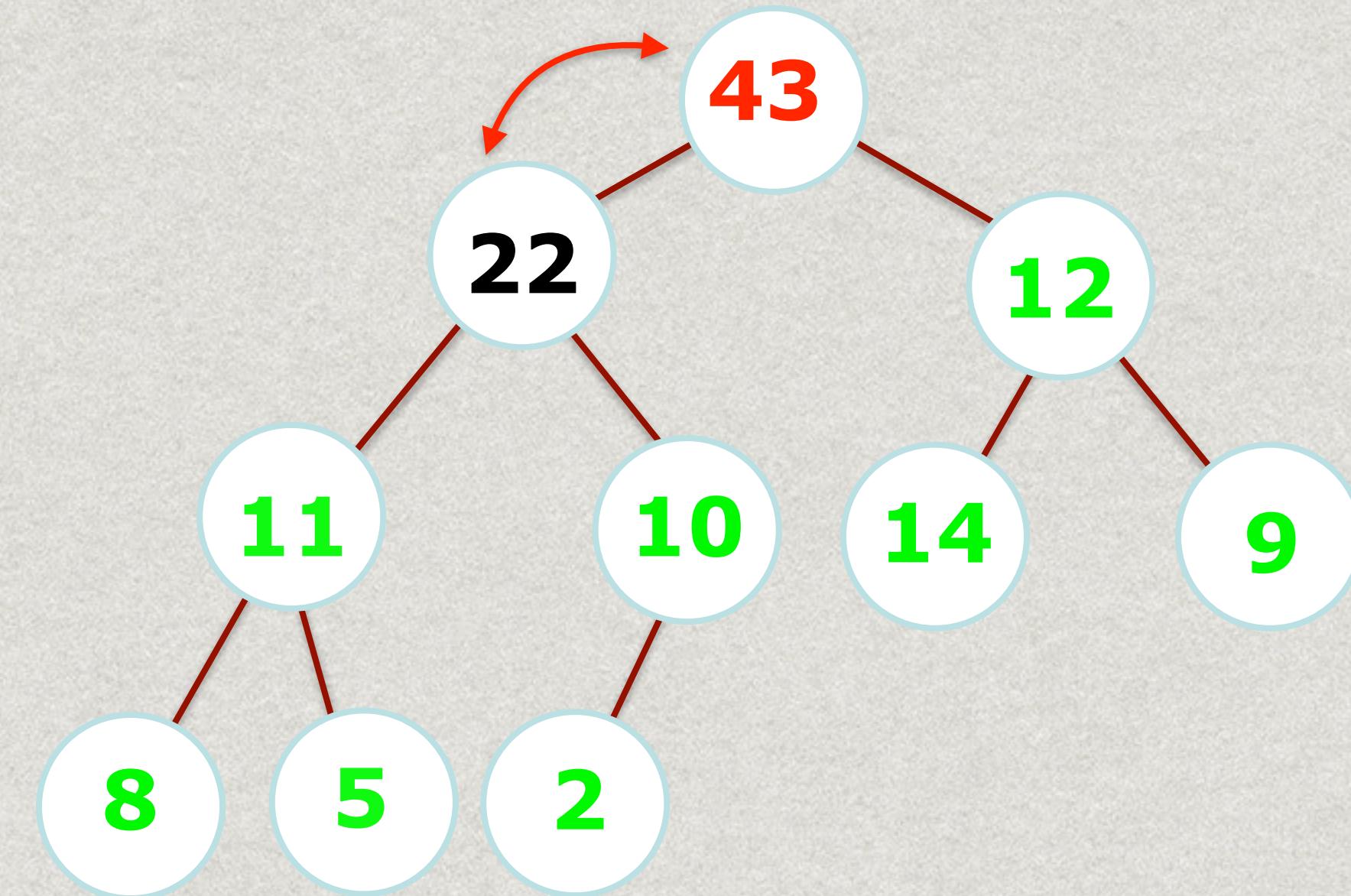
Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

68

heapSize

↓

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



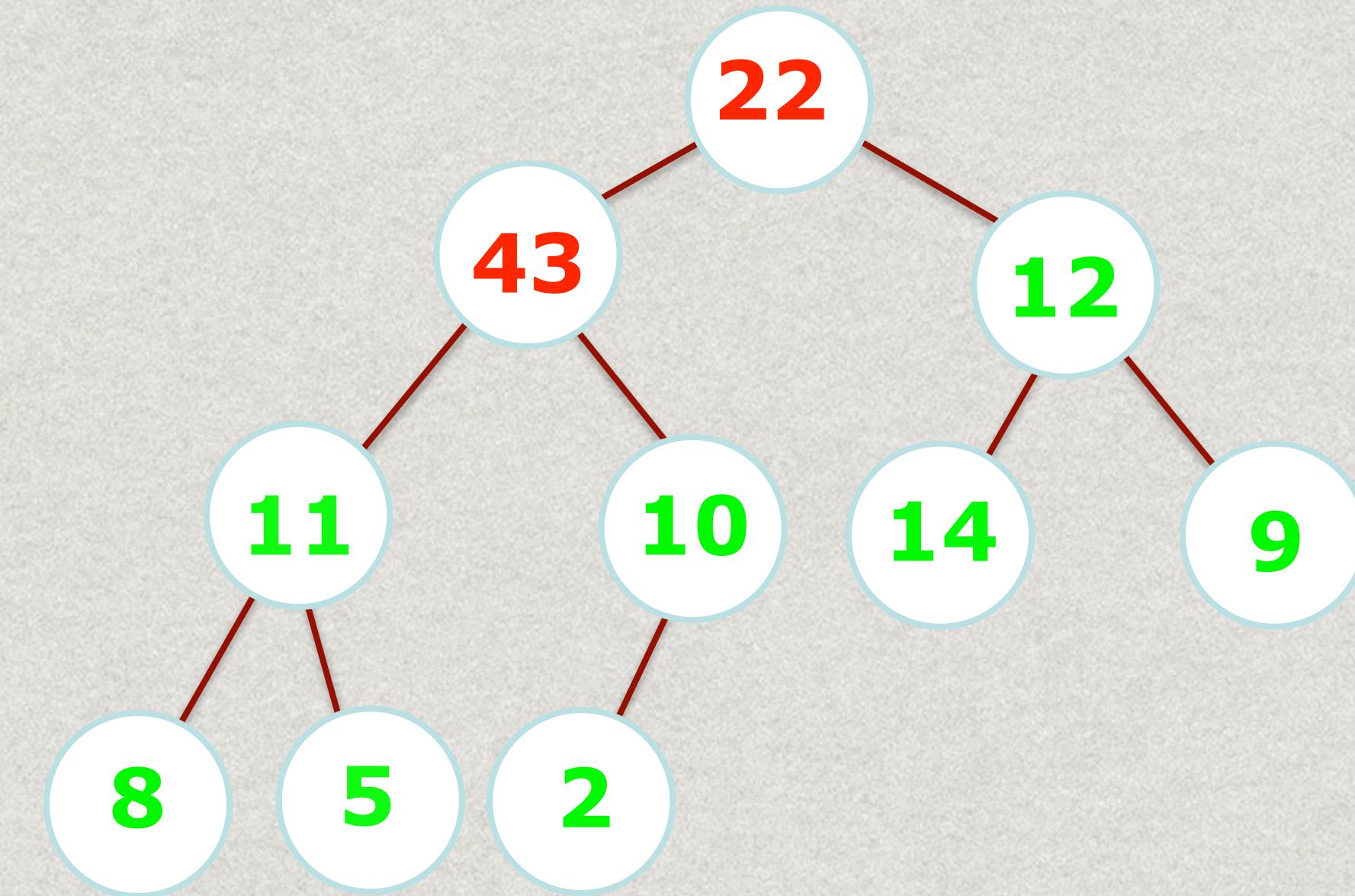
Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping.

69

heapSize

↓

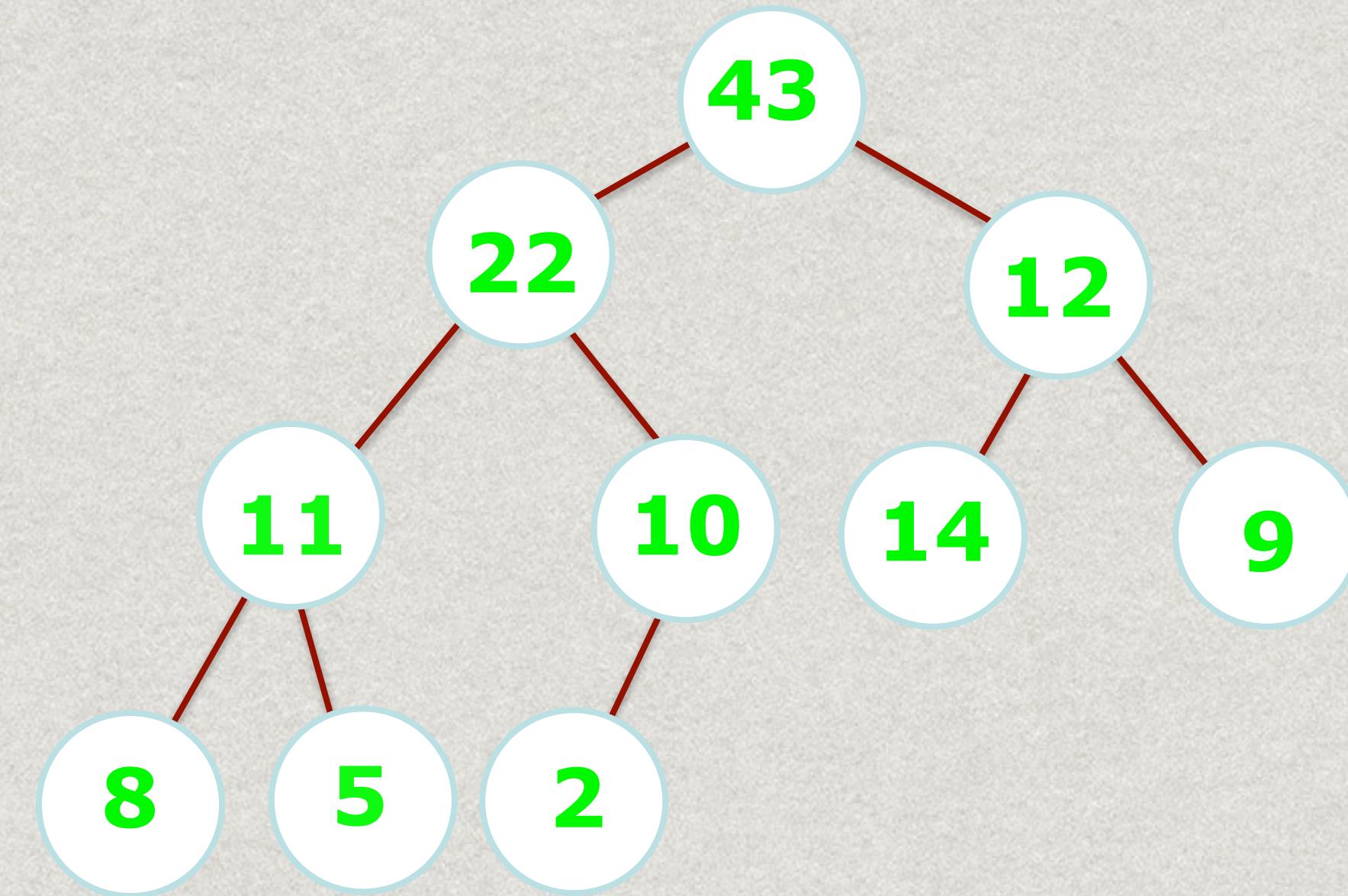
	43	22	12	11	10	14	9	8	5	2
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]



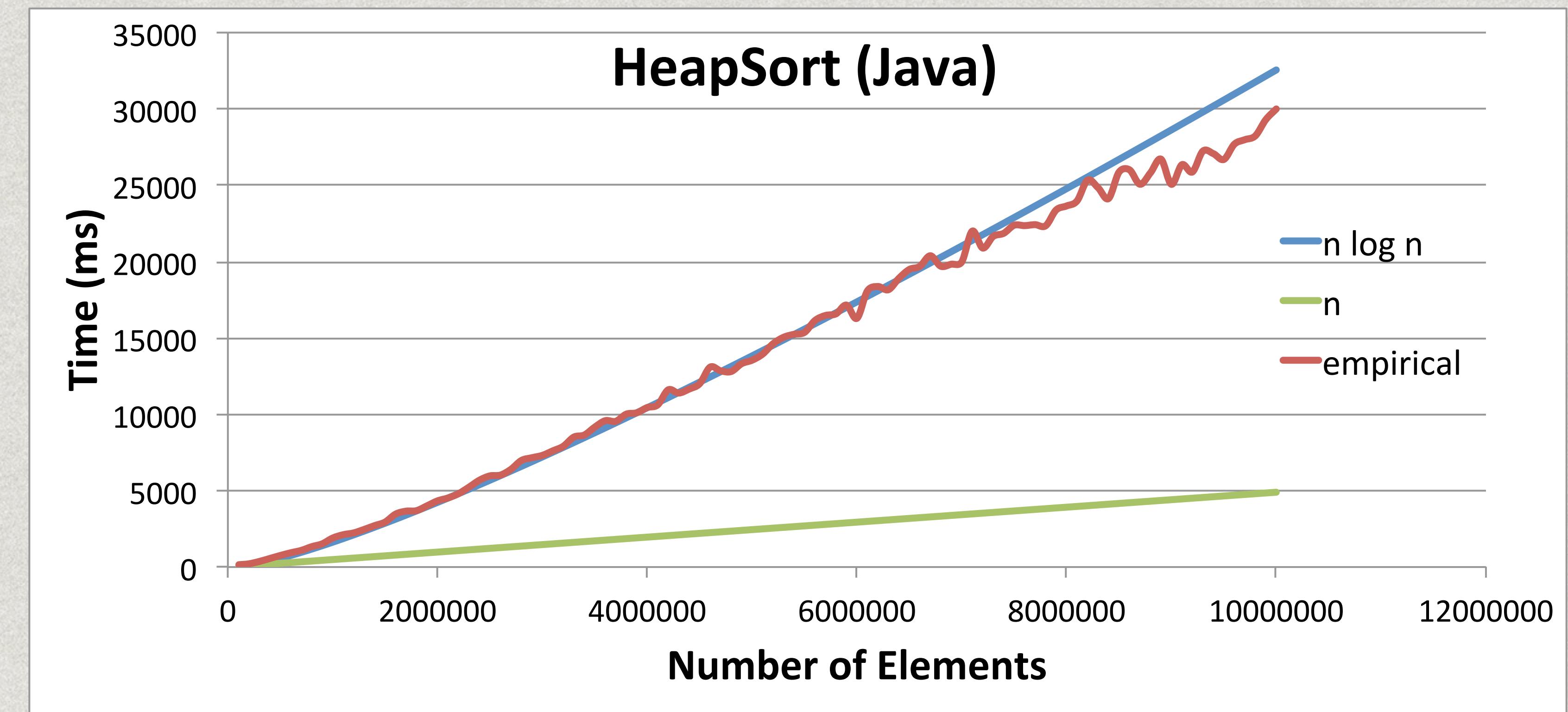
Extras: HeapSort – Iterate and call **dequeue()** , swapping the root with the last element, then down-heaping. 70



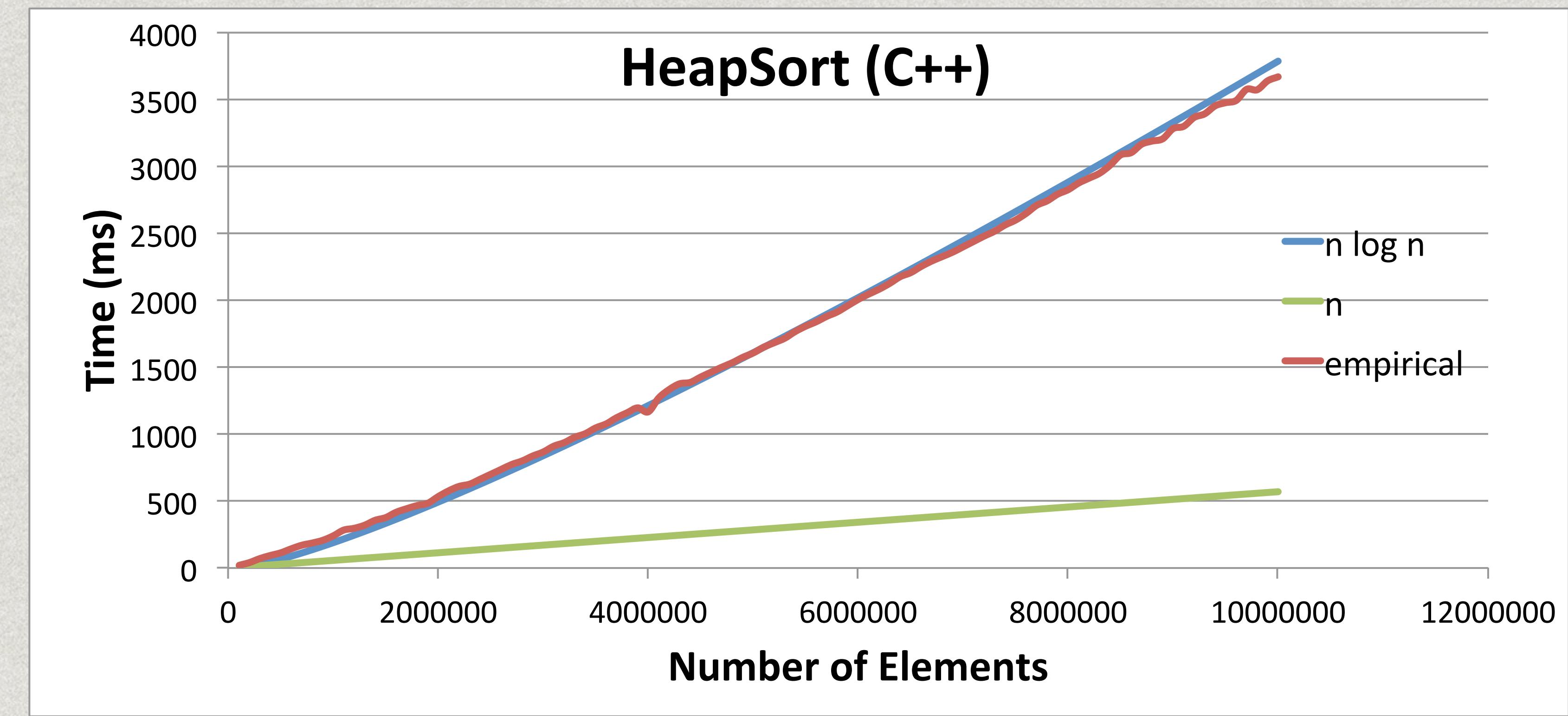
Complexity: $O(n \log n)$



HeapSort Empirical Results (Java)

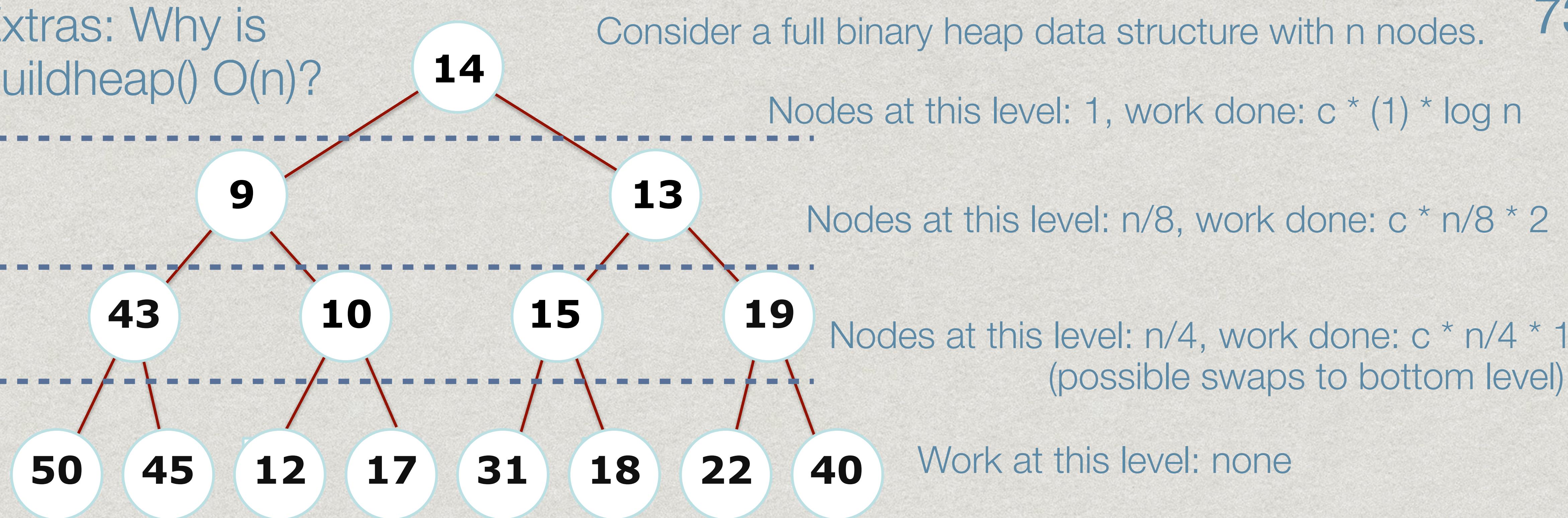


HeapSort Empirical Results (C++)

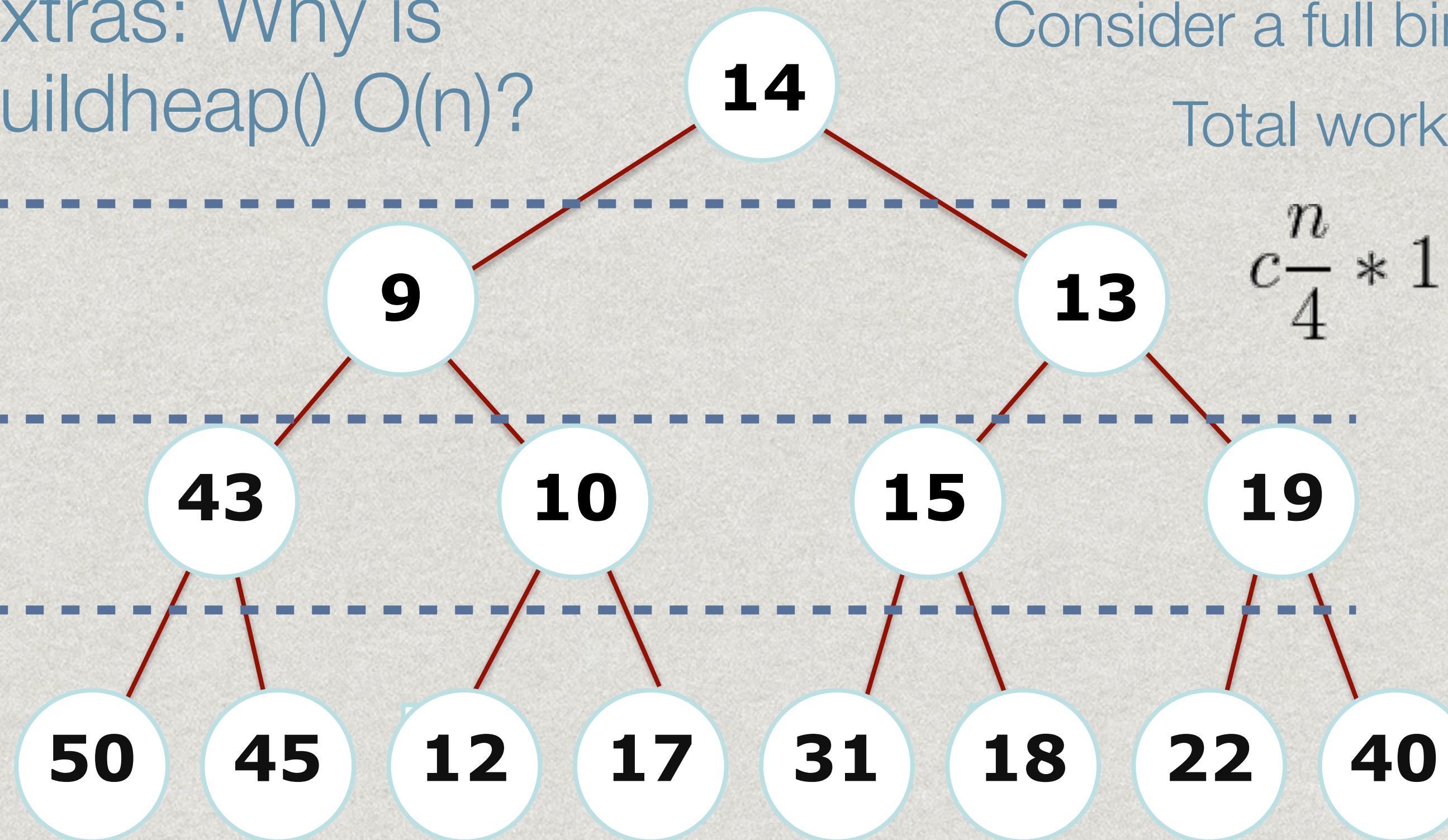


Extras: Why is buildheap() $O(n)$?

73



Extras: Why is
buildheap() $O(n)$?

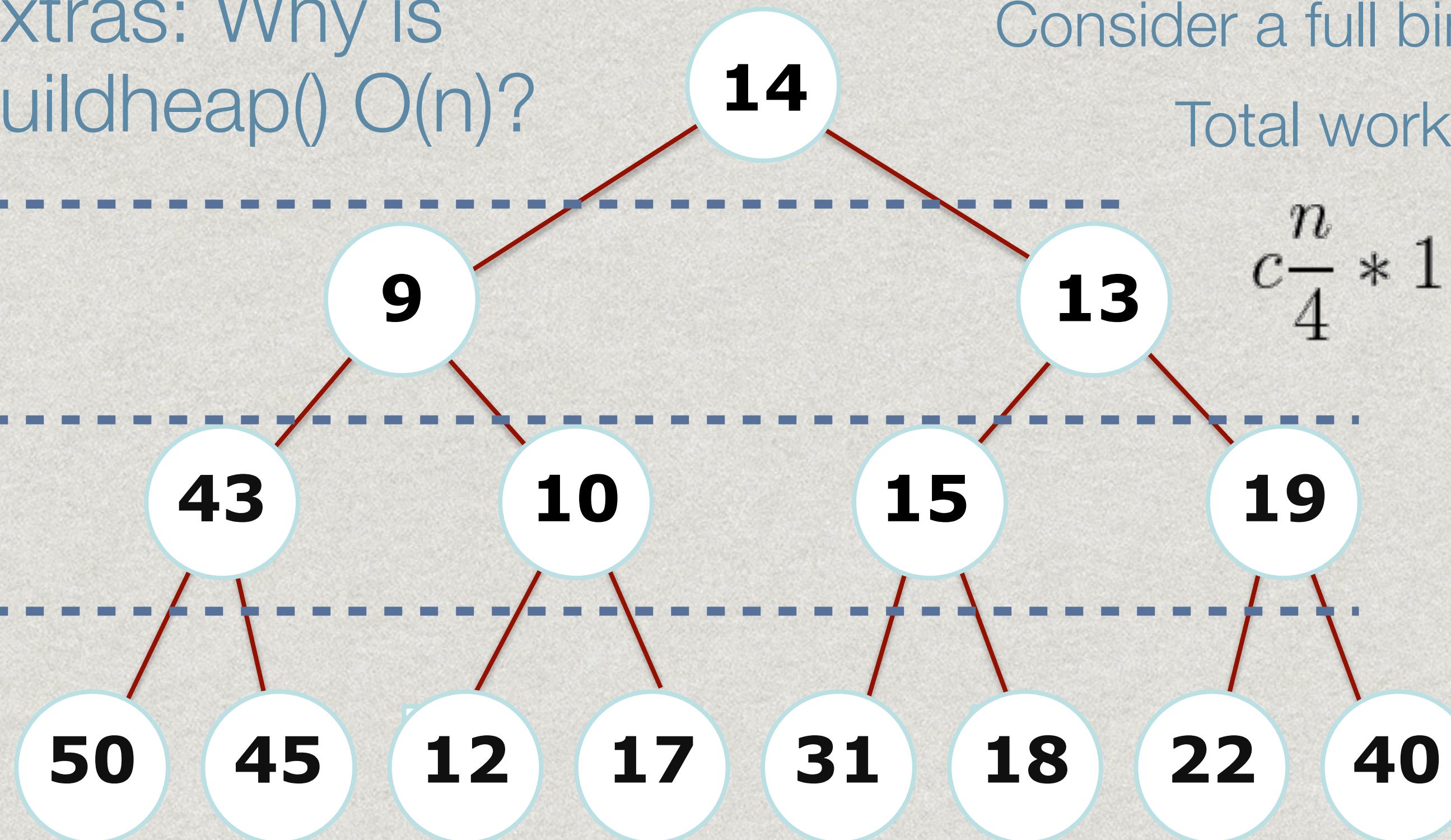


Consider a full binary heap data structure with n nodes.

Total work done:

$$c \frac{n}{4} * 1 + c \frac{n}{8} * 2 + c \frac{n}{16} * 3 + \dots + c(1) * \lg(n)$$

Extras: Why is
buildheap() $O(n)$?



Consider a full binary heap data structure with n nodes.

Total work done:

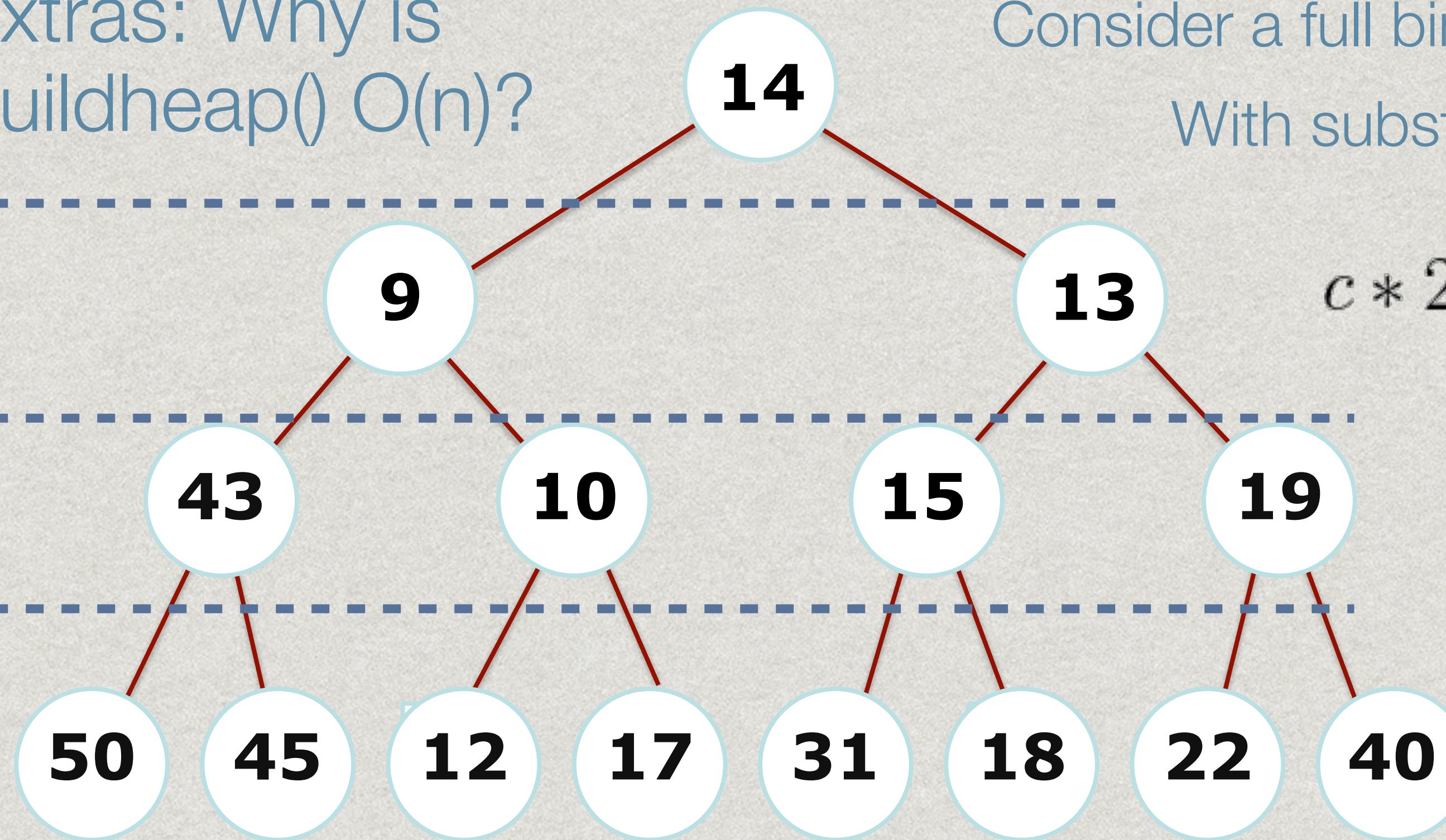
$$c \frac{n}{4} * 1 + c \frac{n}{8} * 2 + c \frac{n}{16} * 3 + \dots + c(1) * \lg(n)$$

Substitution: $\frac{n}{4} = 2^k$

Must do some math for $\lg(n)$:

$$n = 4 * 2^k = 2^2 * 2^k = 2^{k+2}$$
$$\lg(n) = \lg(2^{k+2}) = k + 2$$

Extras: Why is
buildheap() $O(n)$?



Consider a full binary heap data structure with n nodes.

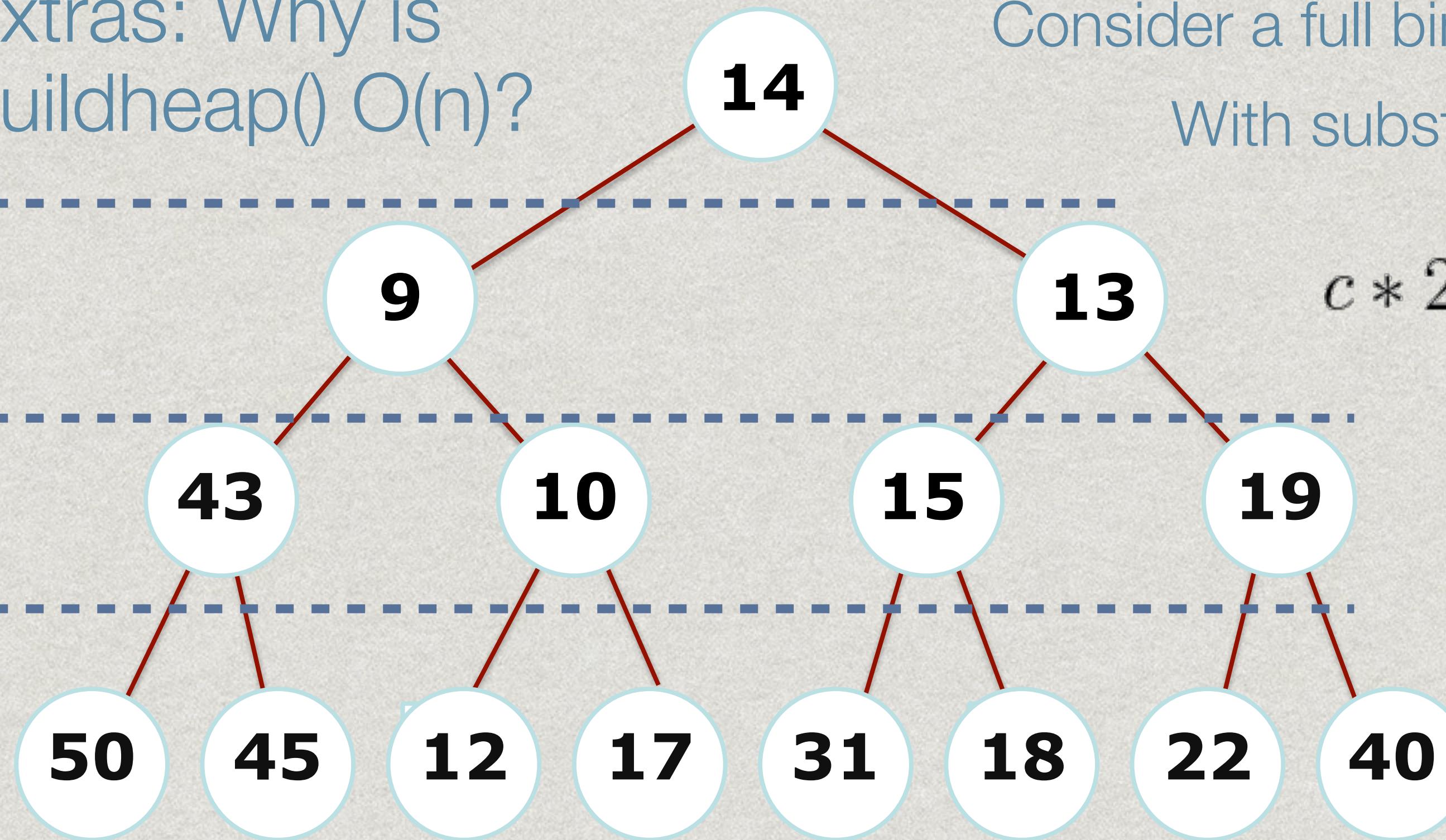
With substitution, and pulling out $c * 2^k$:

$$c * 2^k \left(\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \dots + \frac{k+2}{2^k} \right)$$

Simplify a bit more:

$$\frac{k+2}{2^k} = \frac{k+1+1}{2^k} = \frac{k+1}{2^k} + \frac{1}{2^k}$$

Extras: Why is buildheap() O(n)?



Consider a full binary heap data structure with n nodes.

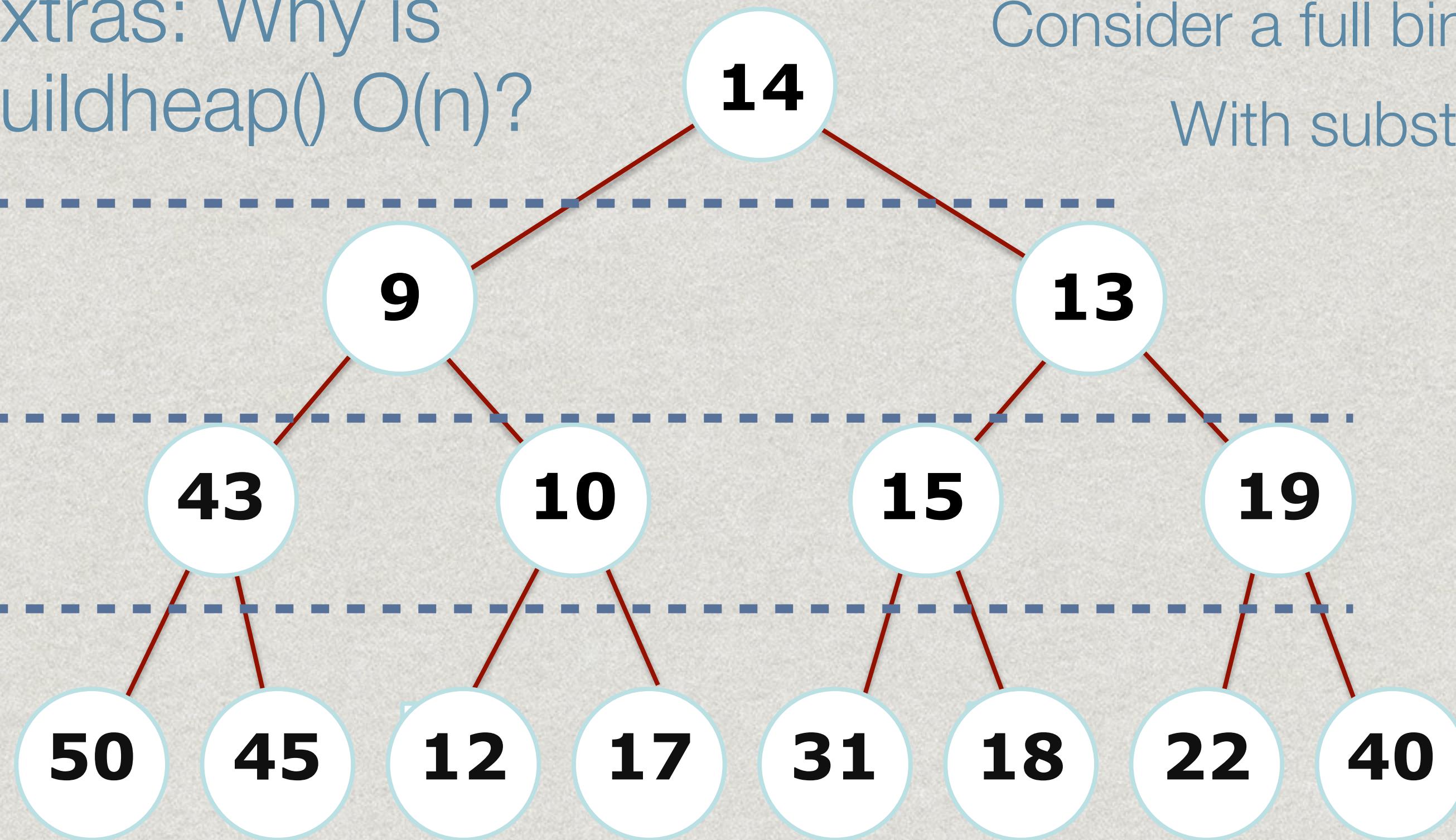
With substitution, and pulling out c^*2^k :

$$c * 2^k \left(\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \dots + \frac{k+2}{2^k} \right)$$

$$c * 2^k \left(\sum_{i=0}^k \frac{i+1}{2^i} + \frac{1}{2^k} \right)$$

$$\sum_{i=0}^k \frac{i+1}{2^i} = 4$$

Extras: Why is buildheap() O(n)?



Consider a full binary heap data structure with n nodes.
With substitution, and pulling out $c * 2^k$:

$$c * 2^k \left(4 + \frac{1}{2^k} \right)$$

$$4c * 2^k + c$$

Substitution: $\frac{n}{4} = 2^k$

$$c * n + c \quad \text{Linear amount of work!}$$