CS 252/ CS242 Data Structures

Priority Queue

Queue

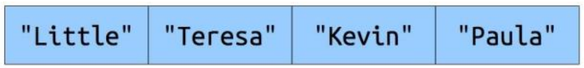
- Order items by when they were placed - first in, first out (FIFO)
- Methods
  - enqueue
  - dequeue
  - first
  - size
  - isEmpty

Priority Queue

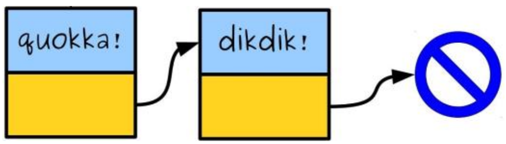
- Order items by rank or key
- Methods
  - enqueue
  - dequeueMin
  - first
  - size
  - isEmpty

Priority Queue Implmentation

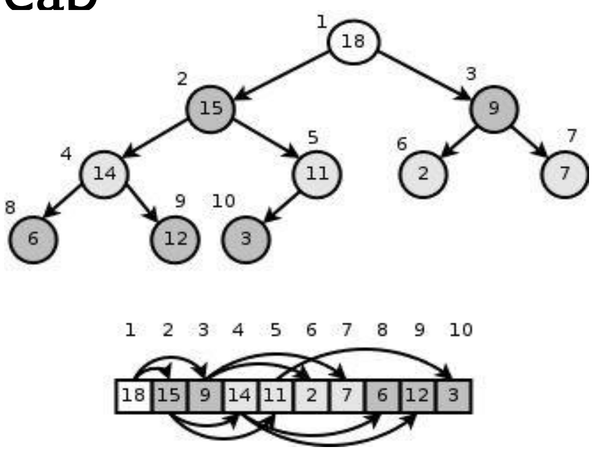
- Unsorted List



- Sorted Singly-Linked List



- Binary Heap

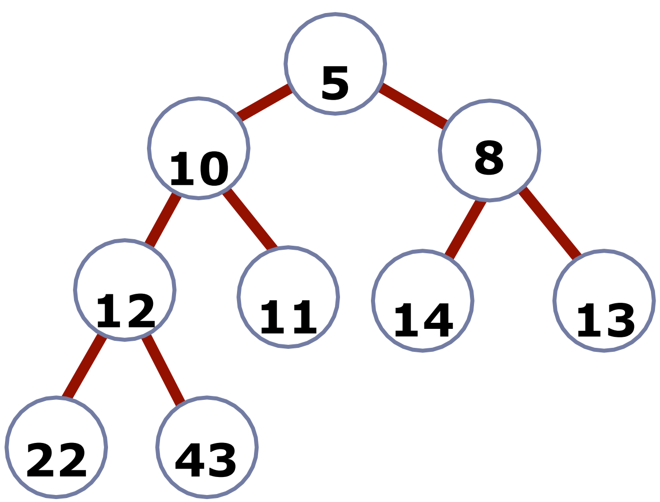


Binary Heaps

- A heap is a tree-based structure that satisfies the heap property:
  - Parents have a higher priority than any of their children.
- Two types of heap

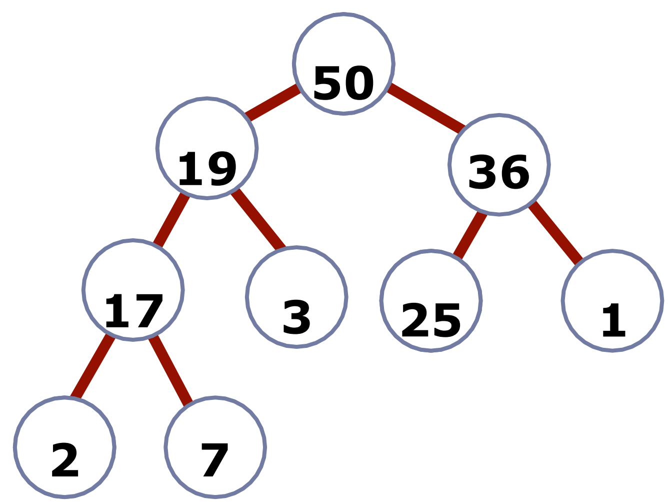
Min Heap

(root is the smallest element)

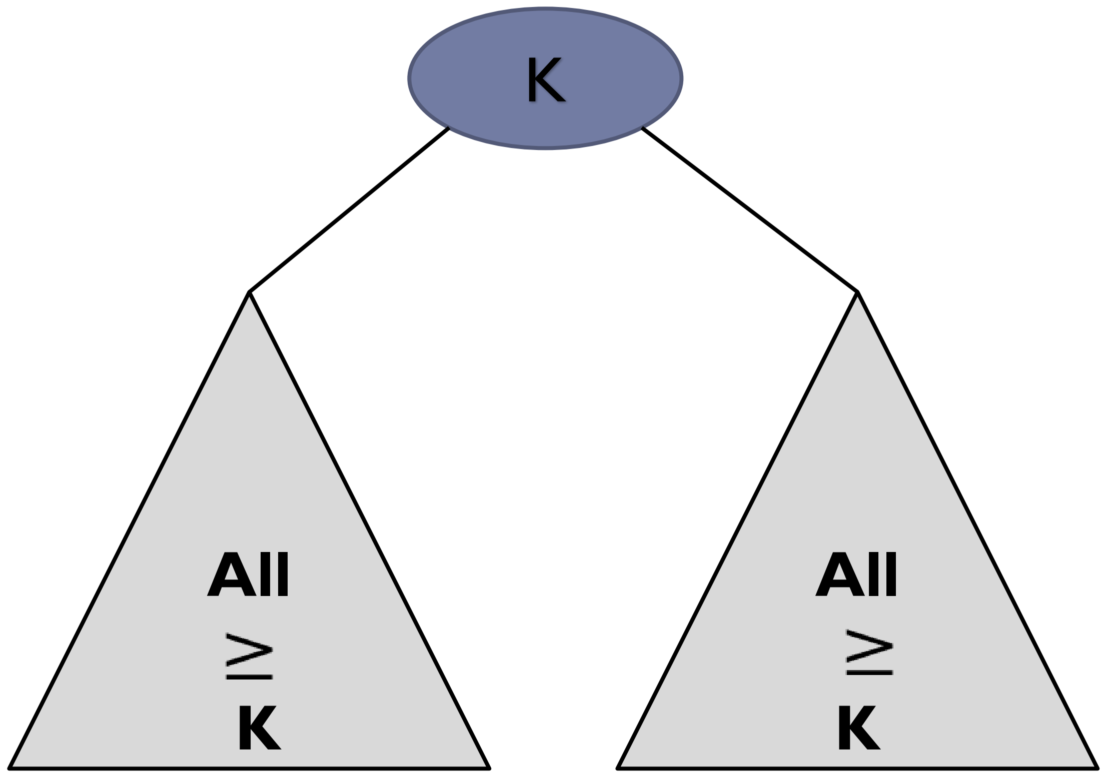


Max Heap

(root is the largest element)



Binary Heaps (Min Heap)

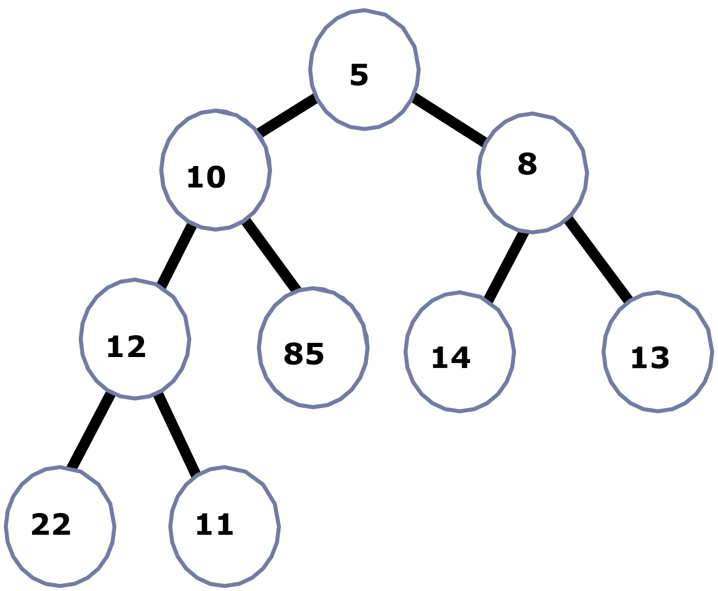


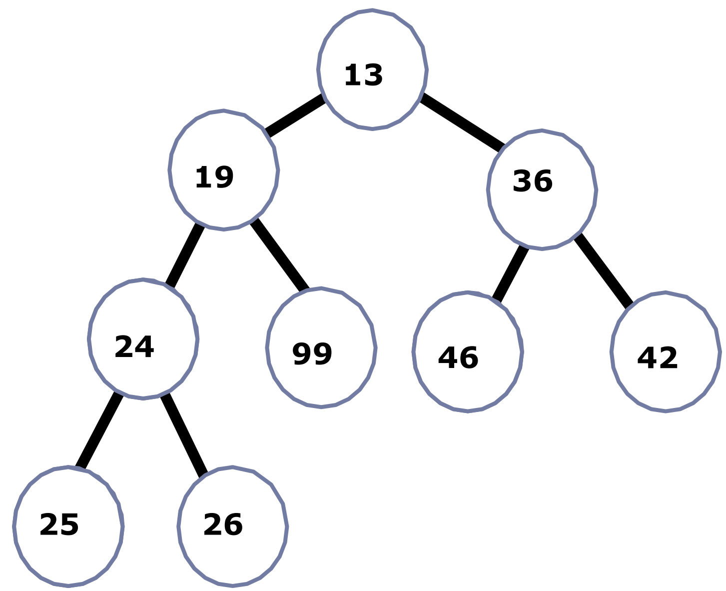
Binary Heaps

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

Binary Heaps

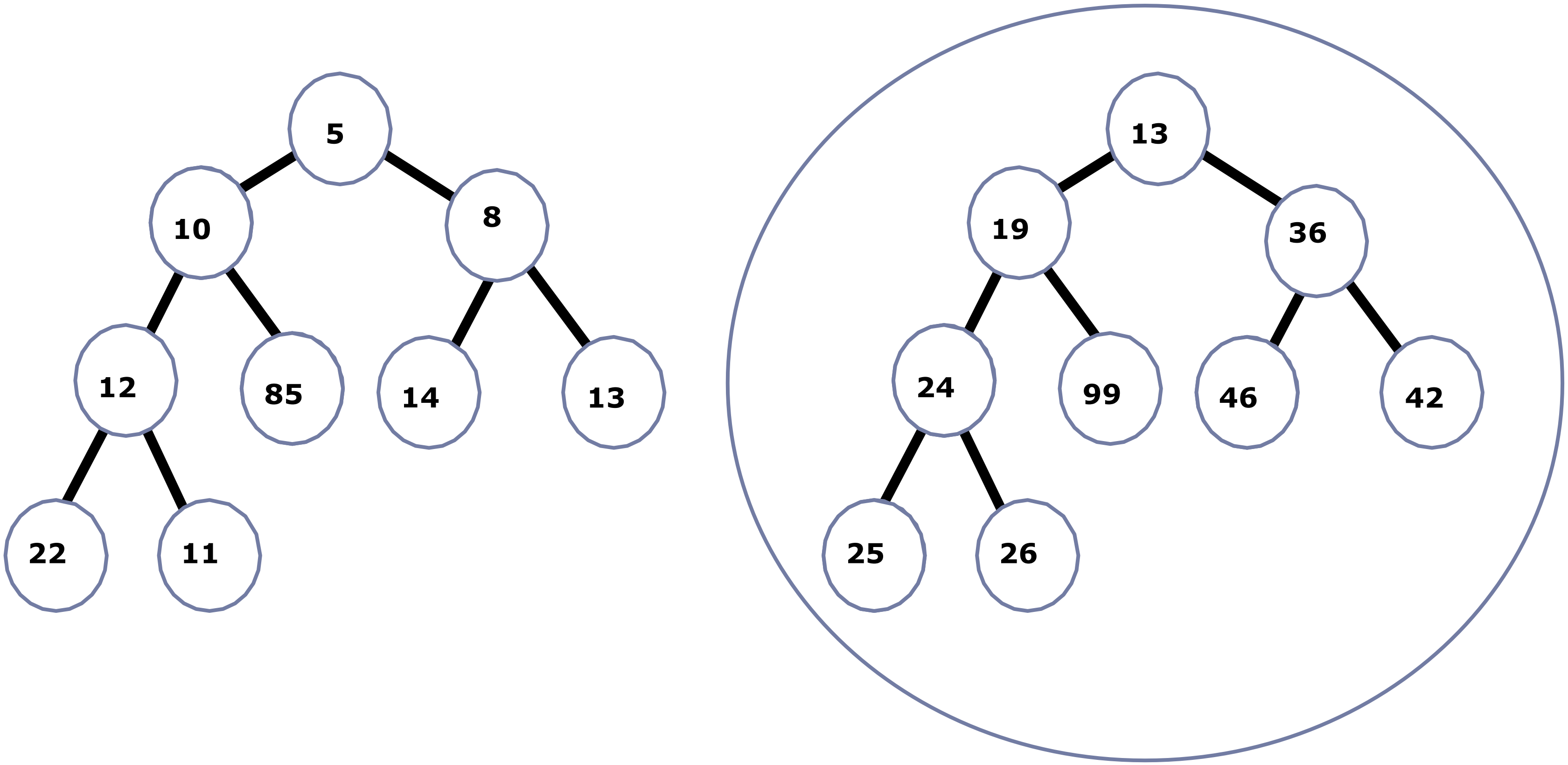
- Circle the min-heap(s)





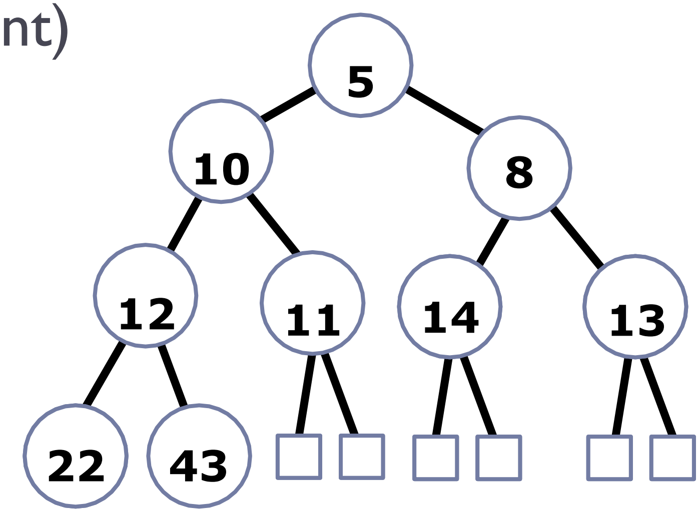
Binary Heaps

- Circle the min-heap(s)



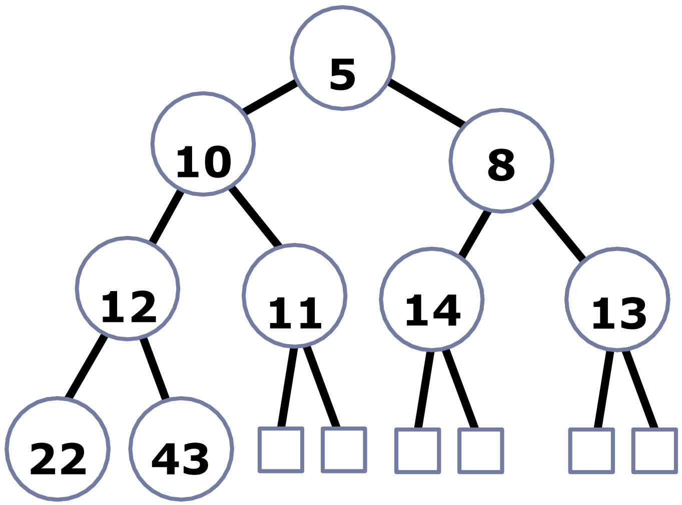
Binary Heaps

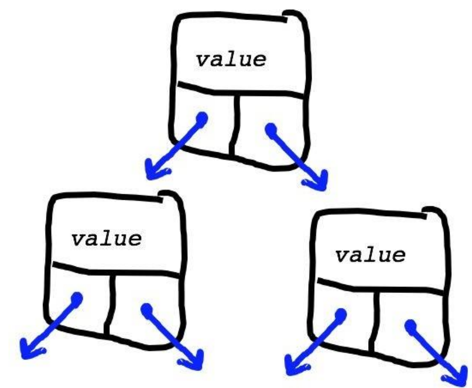
- Heaps are completely filled, with the exception of the bottom level. They are, therefore, "complete binary trees":
  - complete: all levels filled except the bottom
  - binary: two children per node (parent)
  - Height: log(n)



Binary Heaps

- What is the best way to store a heap?

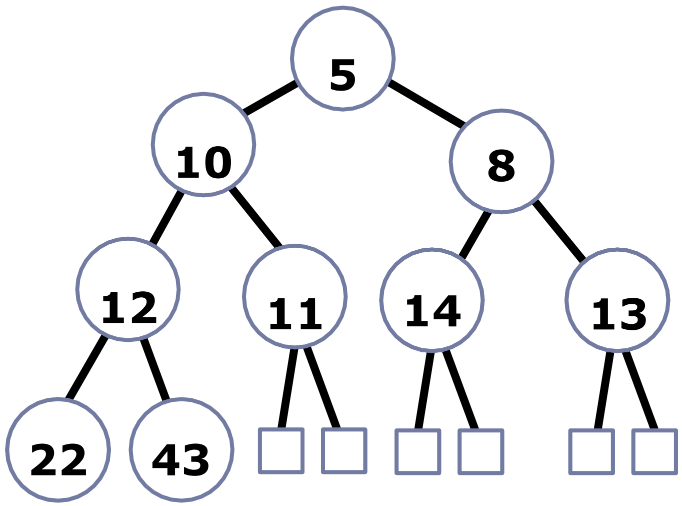




- We could use a node-based solution, but…

Binary Heaps

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

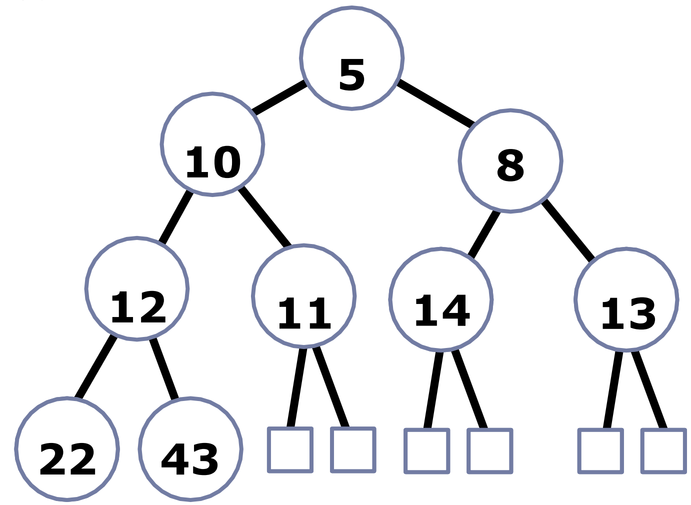


- We will put the root at index 1 instead of index 0 (this makes the math work out just a bit nicer).

heap		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 index i:
  - left child is at 2i
  - right child is at 2i+1
  - parent is at $\lfloor i/2 \rfloor$



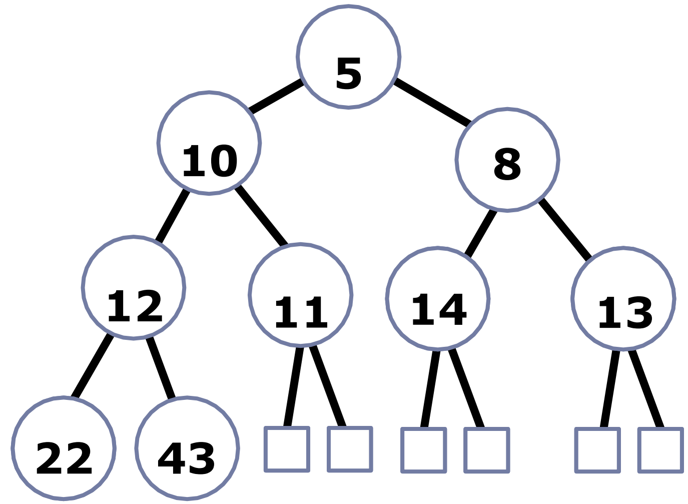
heap		5	10	8	12	11	14	13	22	43		
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap ADT

- **min():** return an element of the heap with the smallest key.
- **insert(e)**: insert element e into the heap.
- **removeMin()**: removes the smallest element from h.
- **size()**: returns number of elements in the heap

Heap Operations: min()

- Just return the root!
  - If(size>0) return heap[1]

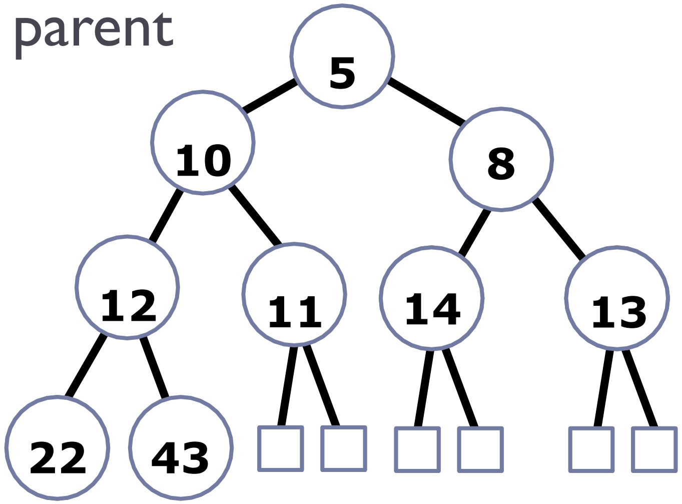


heap

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

Heap Operations: insert(e)

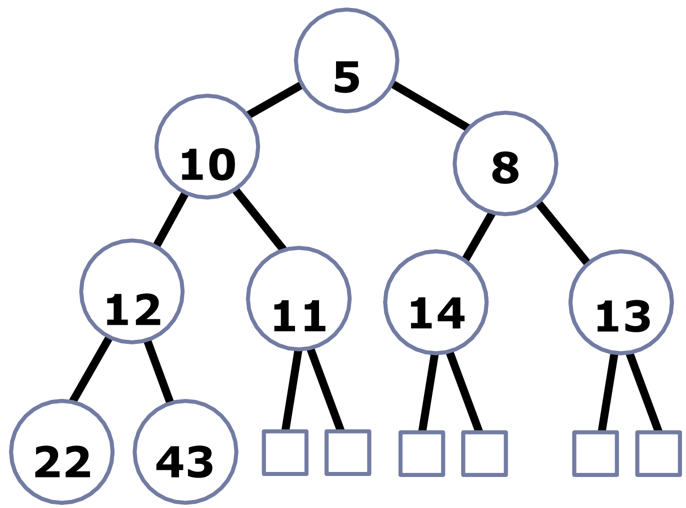
- Insert item at element heap[size()+1]
  - (this probably destroys the heap property)
- Perform a "bubble up" or "up-heap" operation:
  - Compare the added element with its parent
  - if in correct order, stop
  - If not, swap and repeat



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

Heap Operations: insert(9)

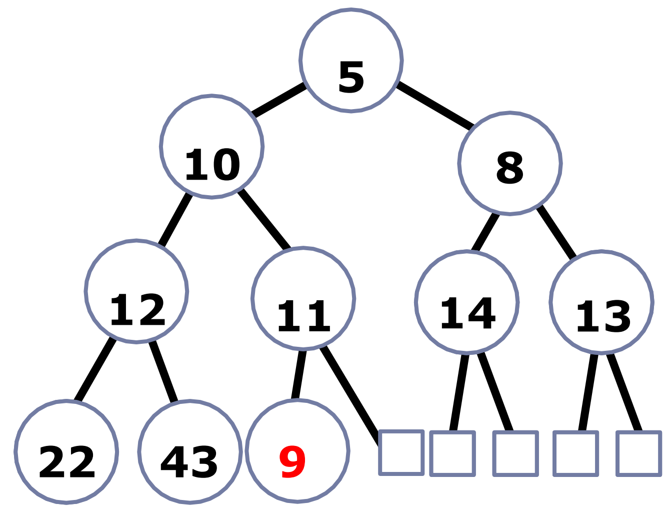
- Start by inserting the key at the first empty position.
  - This is always at index size()+1.



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

Heap Operations: insert(9)

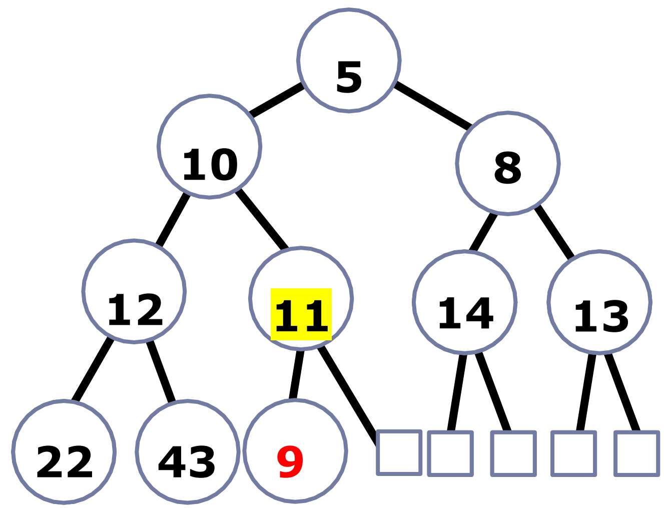
- Start by inserting the key at the first empty position.
  - This is always at index size()+1.



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

Heap Operations: insert(9)

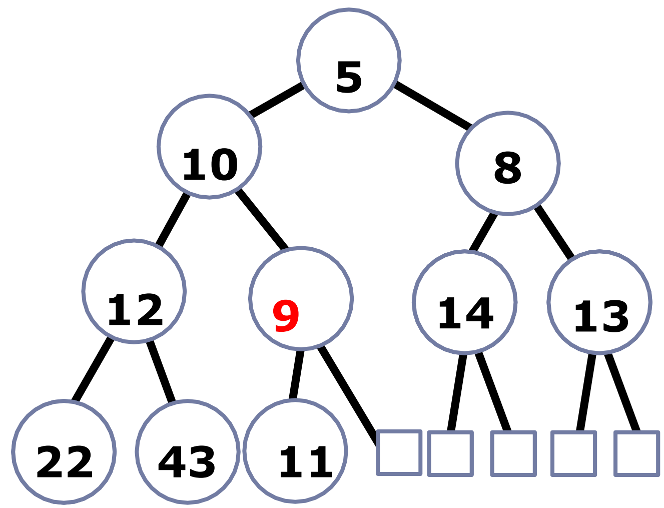
- Look at parent of index 10
  - parent(10)=10/2=5
- Compare: do we meet the heap property requirement?
  - No -- we must swap.



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

Heap Operations: insert(9)

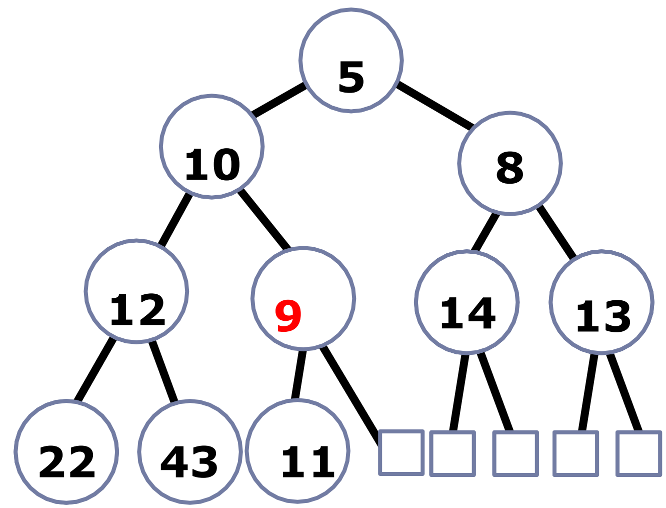
- Look at parent of index 10
  - parent(10)=10/2=5
- Compare: do we meet the heap property requirement?
  - No -- we must swap.



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

Heap Operations: insert(9)

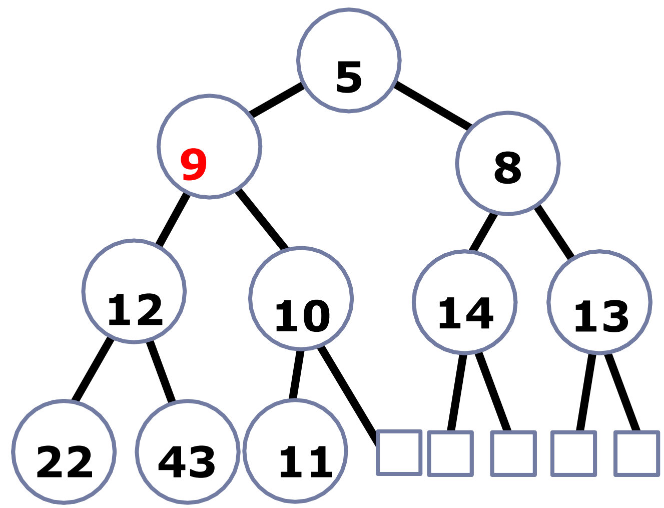
- Look at parent of index 5
  - parent(10)=5/2=2
- Compare: do we meet the heap property requirement?
  - No -- we must swap.



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

Heap Operations: insert(9)

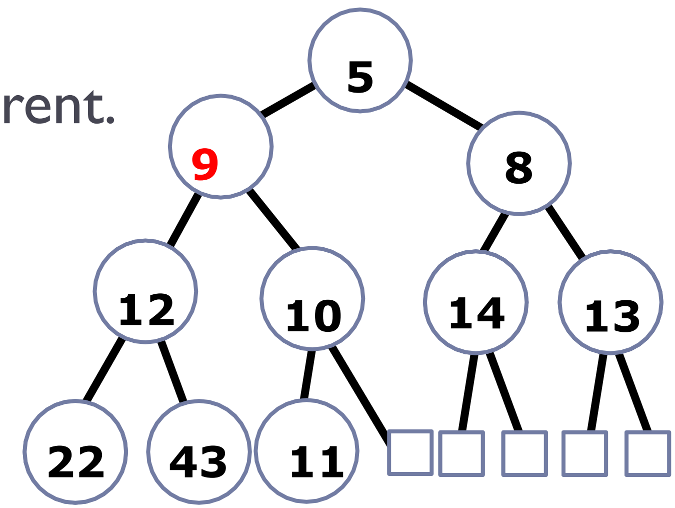
- Look at parent of index 5
  - parent(10)=5/2=2
- Compare: do we meet the heap property requirement?
  - No -- we must swap.



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

Heap Operations: insert(9)

- Look at parent of index 2
  - parent(2)=2/2=1
- Compare: do we meet the heap property requirement?
  - No swap necessary
    between index 2 and its parent.



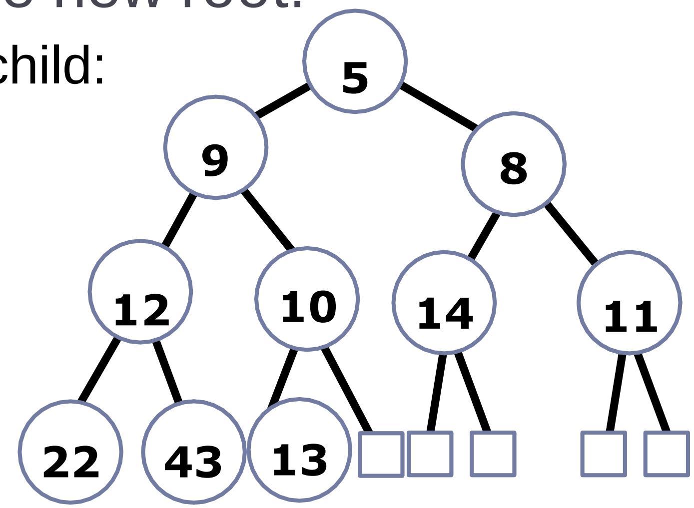
		5	9	8	12	9	14	13	22	43	11	
heap	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Operations: insert(e)

```
insert(e)

    if(heap.length()-1> size())
        heap[size()+1]= e
        size++;
        bubble_up()



bubble_up()

    index=size()
    parent=index/2
    while(index > 1 and  heap[index]< heap[parent])
        swap(index,parent)
        index=parent
        parent=index/2
```

Heap Operations: removeMin()

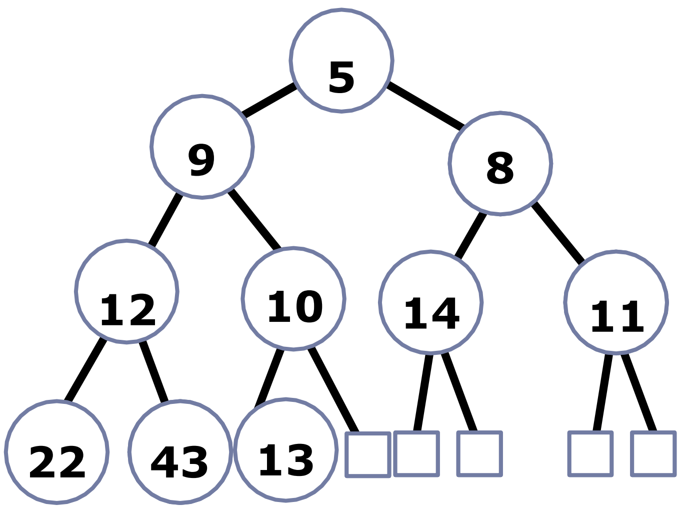
- We are removing the root, and we need to retain a complete tree:
  - replace root with last element.
  - "**bubble-down**" or "down-heap" the new root:
    - Compare the root with its smallest child:
    - if in correct order, stop.
    - if not, swap with **smallest** child and repeat.



heap

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

Heap Operations: removeMin()

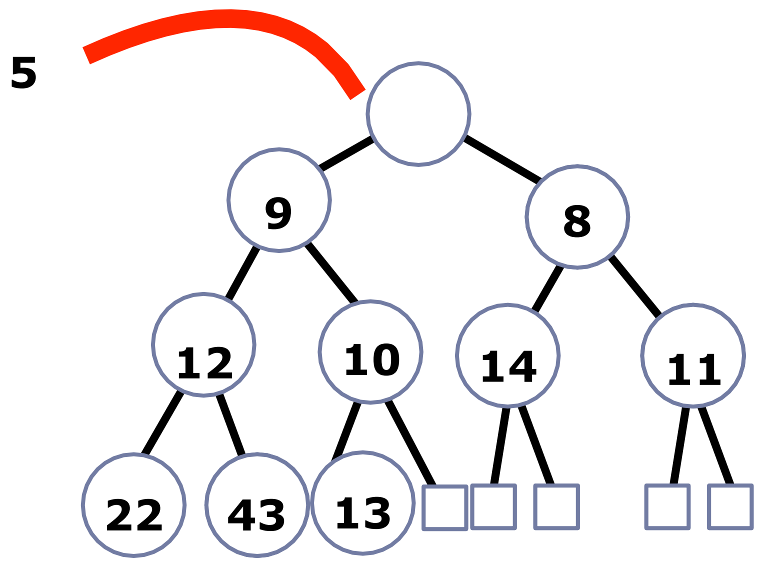


heap

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

Heap Operations: removeMin()

- Remove root (will return at the end)

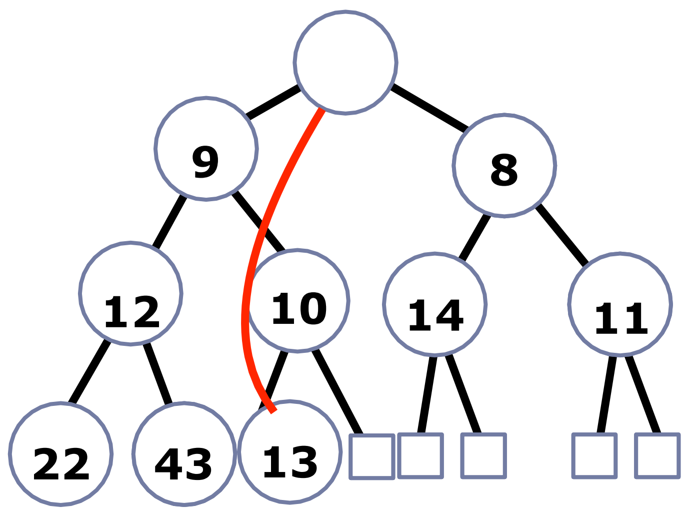


heap

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

Heap Operations: removeMin()

- Move last element (at heap[size()]) to the root.

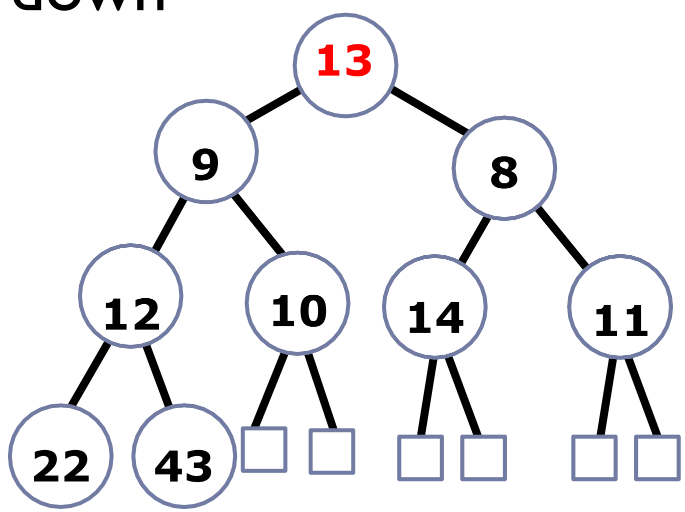


heap

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

Heap Operations: removeMin()

- Move last element (at heap[size()]) to the root.
- Decrease size by 1
- Bubble-down

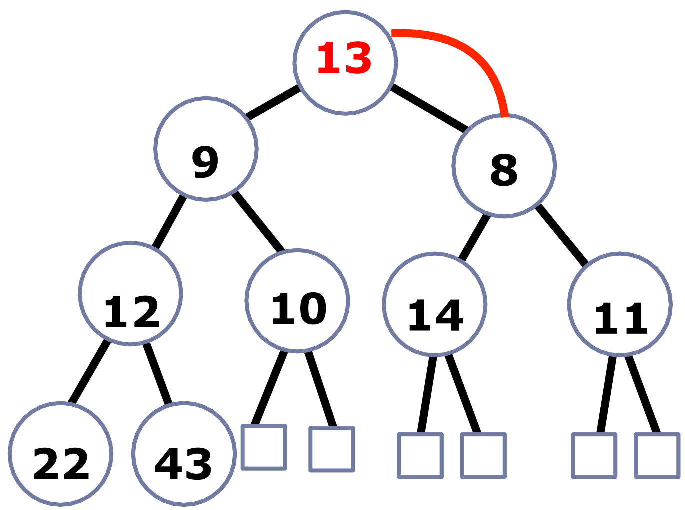



```
removeMin()
   if(size==0)
     return null;
   min=heap[1]
   if(size>1)
     heap[1]=heap[size()];
   size--;
   if(size>1)
     bubble_down(1);
   return min
```

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

Heap Operations: removeMin()

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



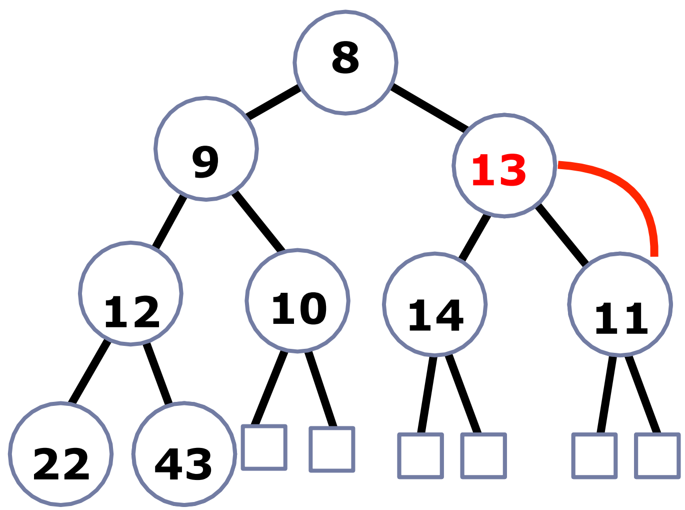

```
left(i)=i*2
right(i)r=i*2+1
```

heap

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

Heap Operations: removeMin()

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



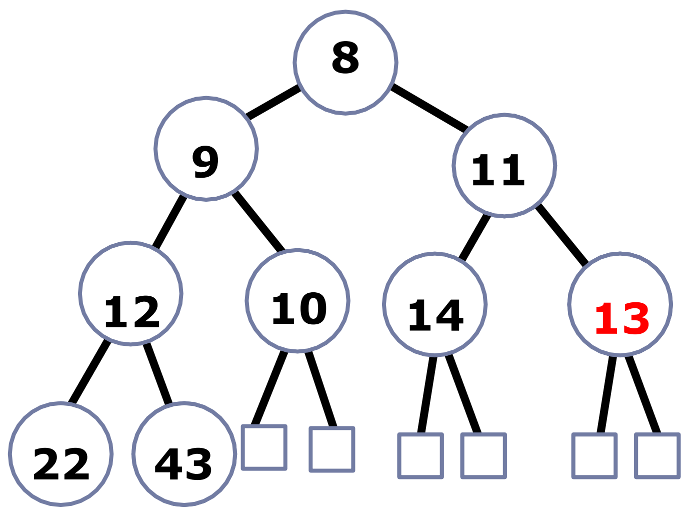

```
left(i)=i*2
right(i)r=i*2+1
```

heap

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

Heap Operations: removeMin()

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



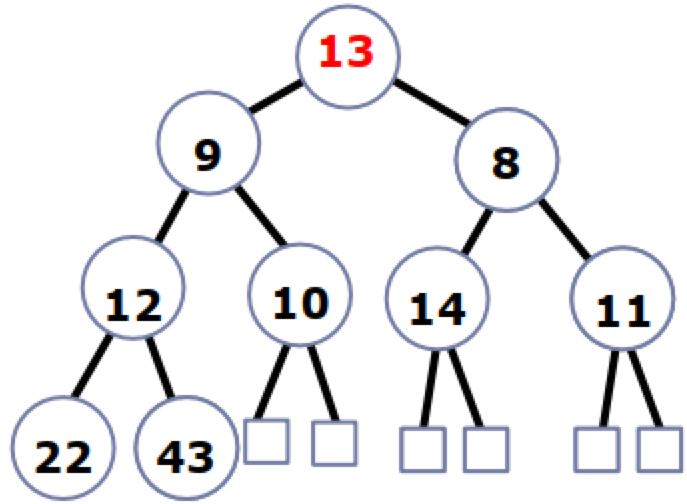

```
left(i)=i*2
right(i)r=i*2+1
```

heap

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

Heap Operations: **bubble_down()**

```
public void bubble_down (int i) {

int l = 2 * i;
int r = 2 * i + 1;

int smallest = i;

if (l < heap.size() && heap[l] < heap[i])
        smallest = l;
    if (r < heap.size() && heap[r] < heap[smallest])
        smallest = r;

    if (smallest != i)
    {
        swap(i, smallest);
        bubble_down(smallest);
    }
}
```



Time Complexity

Method	Binary Heap
`insert`	O(log n)
`removeMin`	O(log n)
`min`	O(1)
`size`	O(1)

Priority Queue

Priority Queue	Binary Heap
enqueue	insert
dequeueMin	removeMin
first	min
size	size
isEmpty	isEmpty

Exercises

- Insert the following elements in sequence into an empty max heap: 6, 8, 4, 7, 2, 3, 9, 1, 5. Draw both the tree and array representations of the heap.

- Write in pseudocode an algorithm for checking that a binary tree satisfies the heap property. Now write the same algorithm but for a heap represented as an array.