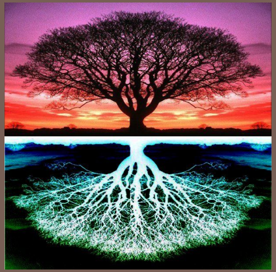
Object-oriented programming and data-structures



CS/ENGRD 2110 SUMMER 2018

Lecture 10: Heaps & Priority Queues

http://courses.cs.cornell.edu/cs2110/2018su

Recall: Data Structures

- List (ArrayList, LinkedList)
- Set (HashSet, TreeSet)
- Map (HashMap, TreeMap)
- Queue (LinkedList)
- PriorityQueue

Recall: Data Structures

- List (ArrayList, LinkedList)
- Set (HashSet, TreeSet)
- Map (HashMap, TreeMap)
- Queue (LinkedList)
- PriorityQueue

Different abstract data-structures expose different functionality

Set :

```
add(E e)
contains(E e)
remove(E e)
```

List:

```
add(E e)
add(int i, E e)
remove(int i)
get(int i)
contains (E e)
```

Recall: Data Structures

- List (ArrayList, LinkedList)
- Set (HashSet, TreeSet)
- Map (HashMap, TreeMap)
- Queue (LinkedList)
- PriorityQueue

Different implementations have different complexity

HashSet :

add(E e) O(1)
contains(E e) O(1)
remove(E e) O(1)

TreeSet :

add(E e)   O(lg n)
contains(E e) O(lg n)
remove(E e) O(lg n)

Priority Queues

- Priority Queues allow you to receive the “next” element in the queue efficiently
  - Where each element has a **priority order (or key)**
    - (ex: Could be defined by compareTo() in Java)

- Two types of priority queues:
  - Min-Queues
  - Max-Queues

Max Priority Queues

- Supports the following operations:
  - insert(e,k) inserts the element e into the queue
  - maximum() returns the element with the largest key
  - extract-max() removes and returns the element with the largest key
  - increase-key(e,k) increases the value of element e's key to the new value k, which is assumed to be at least as large as e's current key value

Min Priority Queues

- Supports the following operations:
  - insert(e,k) inserts the element e into the queue
  - minimum() returns the element with the largest key
  - extract-min() removes and returns the element with the largest key
  - decrease-key(e,k) decreases the value of element e's key to the new value k, which is assumed to be smaller than e's current key value

Why priority queues

- Used for event-driven simulations (Emergencies, casualties)
- Graph searching (Djikstra's algorithm ...)
- Operating Systems (Load balancing, interrupts)
- Video games
- AI algorithms (A* search algo)
- Compression (Huffman Coding)

Priority Queues in Java

```
interface PriorityQueue<E> {
 boolean add(E e) {...} //insert e.
 E poll() {...} //remove/return min elem.
 E peek() {...} //return min elem.
 void clear() {...} //remove all elems.
 boolean contains(E e)
 boolean remove(E e)
 int size() {...}
 Iterator<E> iterator()
}
```

Can we implement priority queues?

- Queues are an **abstract data type**
  - Can we already implemented them using what we've learnt?

Can we implement priority queues?

- What about a linked list?

Can we implement priority queues?

- What about a linked list?
  - add() - put new element at the front O(1)

Can we implement priority queues?

- What about a linked list?
  - add() - put new element at the front O(1)
  - poll() - must search the list - O(n)

Can we implement priority queues?

- What about a linked list?
  - add() - put new element at the front O(1)
  - poll() - must search the list - O(n)
  - peek() - must search the list - O(n)

Can we implement priority queues?

- What about a linked list?
  - add() - put new element at the front O(1)
  - poll() - must search the list - O(n)
  - peek() - must search the list - O(n)

- What about an **ordered** list?

Can we implement priority queues?

- What about a linked list?
  - add() - put new element at the front O(1)
  - poll() - must search the list - O(n)
  - peek() - must search the list - O(n)

- What about an **ordered** list?
  - add() - must search the list - O(n)

Can we implement priority queues?

- What about a linked list?
  - add() - put new element at the front O(1)
  - poll() - must search the list - O(n)
  - peek() - must search the list - O(n)

- What about an **ordered** list?
  - add() - must search the list - O(n)
  - poll() - min element at front - O(1)

Can we implement priority queues?

- What about a linked list?
  - add() - put new element at the front O(1)
  - poll() - must search the list - O(n)
  - peek() - must search the list - O(n)

- What about an **ordered** list?
  - add() - must search the list - O(n)
  - poll() - min element at front - O(1)
  - peek() - min element at front - O(1)

Can we implement priority queues?

- What about a linked list?
  - add() - put new element at the front O(1)
  - poll() - must search the list - O(n)
  - peek() - must search the list - O(n)

- What about an **ordered** list?
  - add() - must search the list - O(n)
  - poll() - min element at front - O(1)
  - peek() - min element at front - O(1)

- What about a **red-black tree**?

Can we implement priority queues?

- What about a linked list?
  - add() - put new element at the front O(1)
  - poll() - must search the list - O(n)
  - peek() - must search the list - O(n)

- What about an **ordered** list?
  - add() - must search the list - O(n)
  - poll() - min element at front - O(1)
  - peek() - min element at front - O(1)

- What about a **red-black tree**?
  - add()/poll()/peek() - must search the tree & rebalance O(log n)

Can we do better?

- Goals:
  - efficiently find the head of the queue
    - Can we do constant time?
  - efficiently insert an element in the queue

- Non-goals:
  - find an element that is not the head of the queue

Introducing Heaps

- A *heap* is a binary tree that satisfies two properties
  - **Completeness.** Every level of the tree (except last) is completely filled.
  - **Heap Order Invariant.**

    Every element in the tree is
    - Smaller or equal than its parent (*min-heap*)
    - Greater or equal than its parent (*max-heap*)

Introducing Heaps

- A *heap* is a binary tree that satisfies two properties
  - **Completeness.** Every level of the tree (except last) is completely filled.
  - **Heap Order Invariant.**

    Every element in the tree is

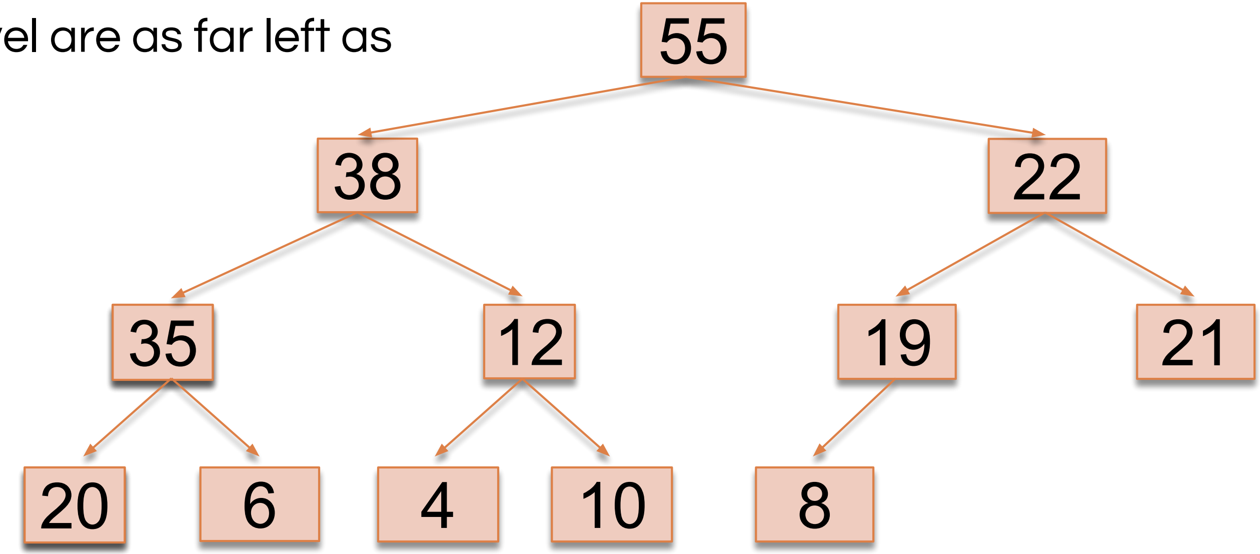
    - Smaller or equal than its parent (*max-heap*)
    - Greater or equal than its parent (*min-heap*)

Do not confuse with heap memory, where a process dynamically allocates space–different usage of the word heap.

Completeness Property

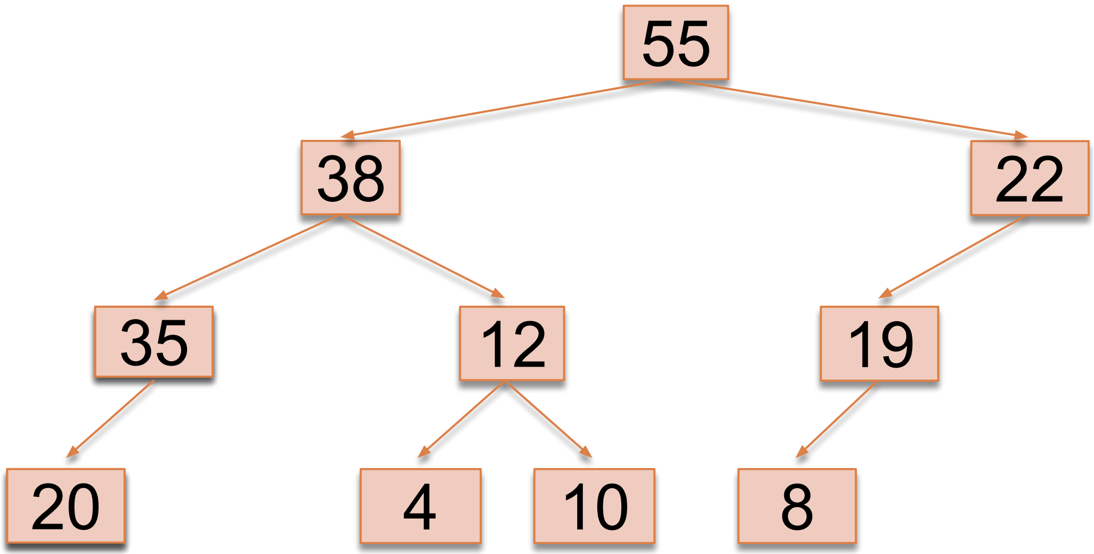
Every level (except last) completely filled.

Nodes on bottom level are as far left as possible.



Completeness Property

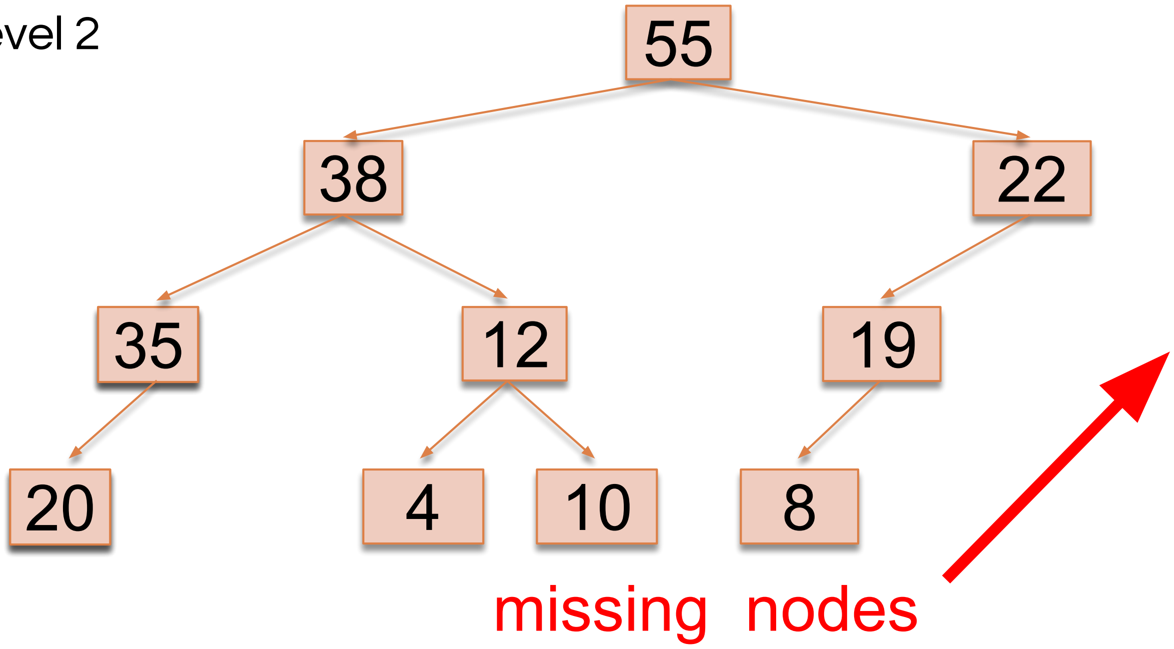
Not a heap because:



Completeness Property

Not a heap because:

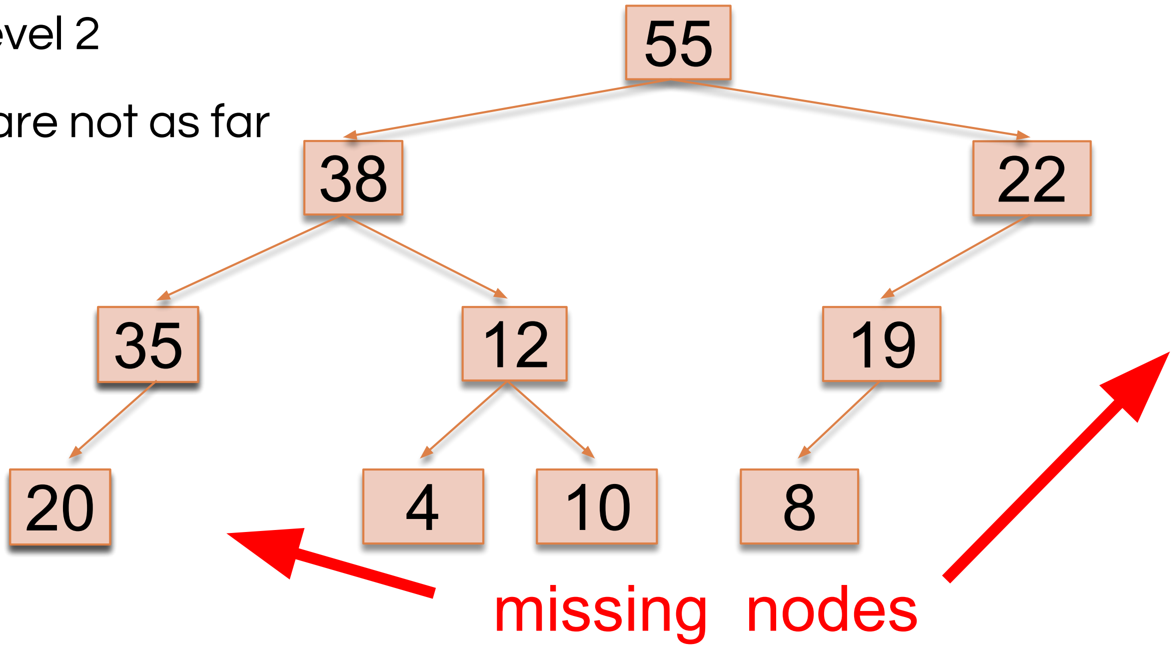
- missing a node on level 2



Completeness Property

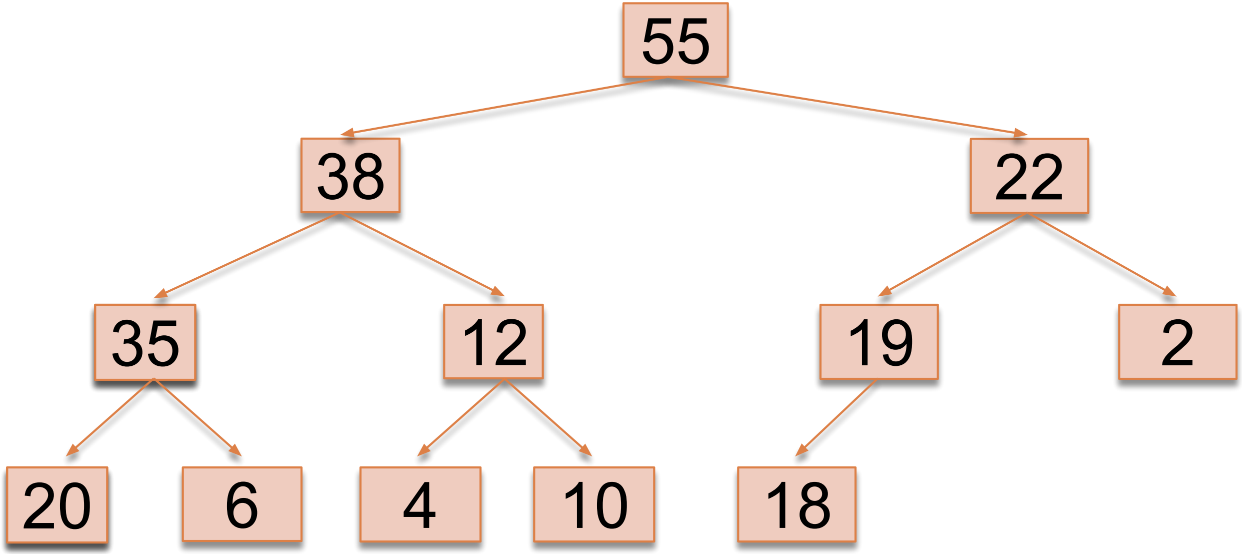
Not a heap because:

- missing a node on level 2
- bottom level nodes are not as far left as possible



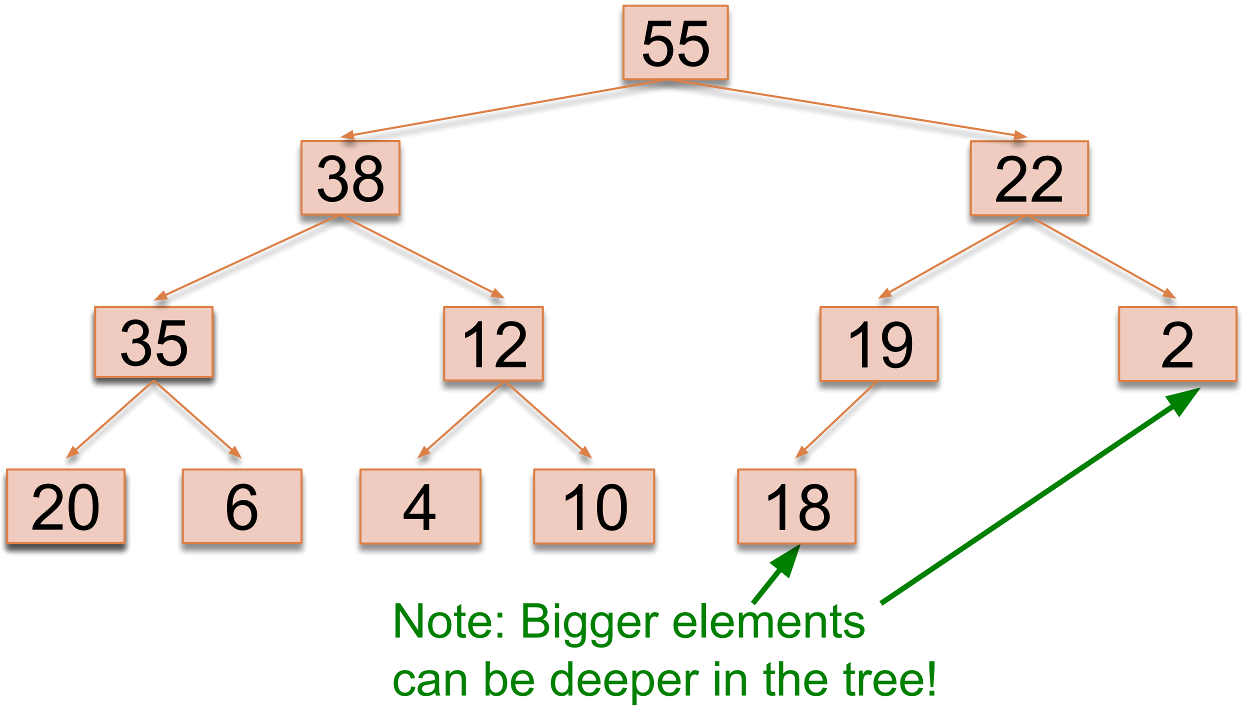
Order Property

Every element is <= its parent



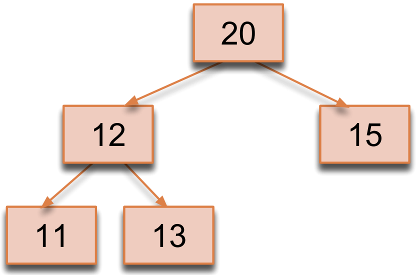
Order Property

Every element is <= its parent

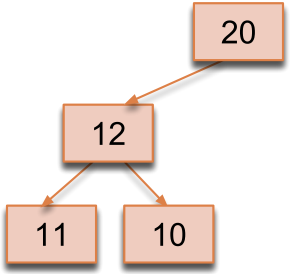


Heap Quiz

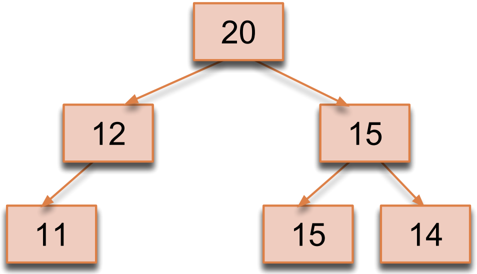
Which of the following are valid heaps?



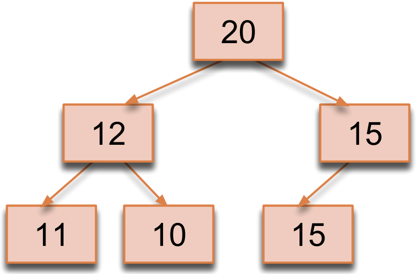
(a)



(b)



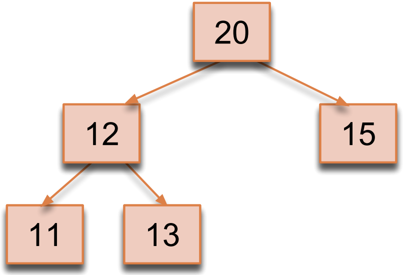
(c)



(d)

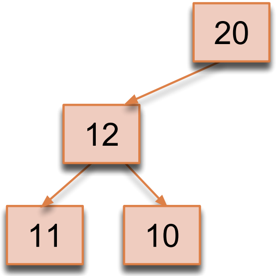
Heap Quiz

Which of the following are valid heaps?



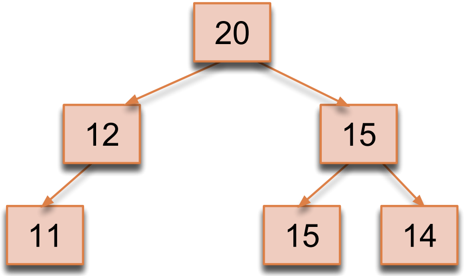
(a)

No



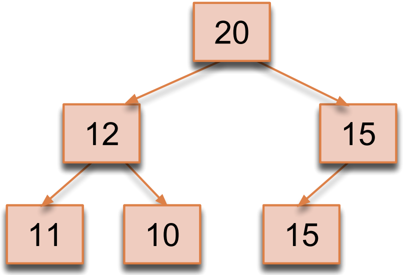
(b)

No



(c)

No



(d)

Ye
s

Heaps

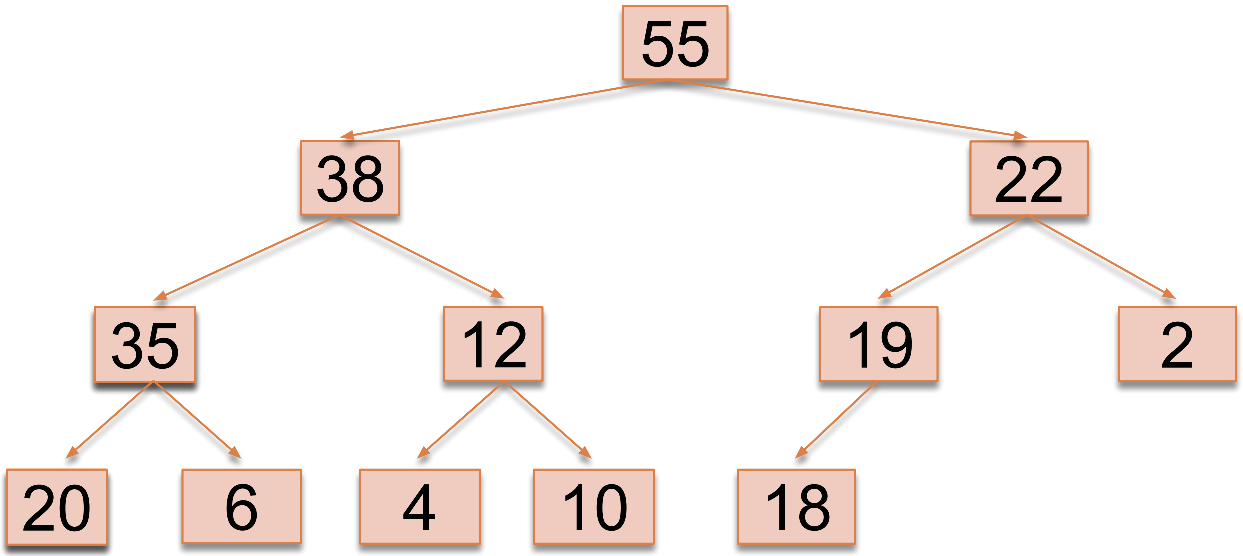
A ***heap*** is a binary tree that satisfies two properties

1) Completeness. Every level of the tree (except last) is completely filled. All holes in last level are all the way to the right.
2) Heap Order Invariant. Every element in the tree is <= its parent

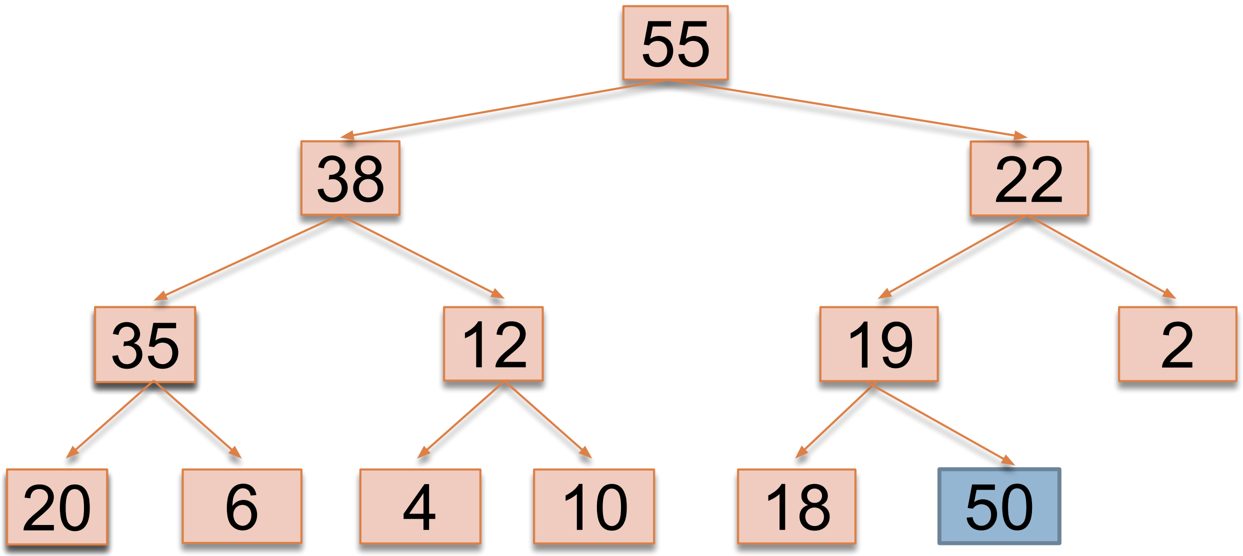
A heap implements three key methods:

3) add(e): adds a new element to the heap
4) poll(): deletes the max element and returns it
5) peek(): returns the max element

add(e)

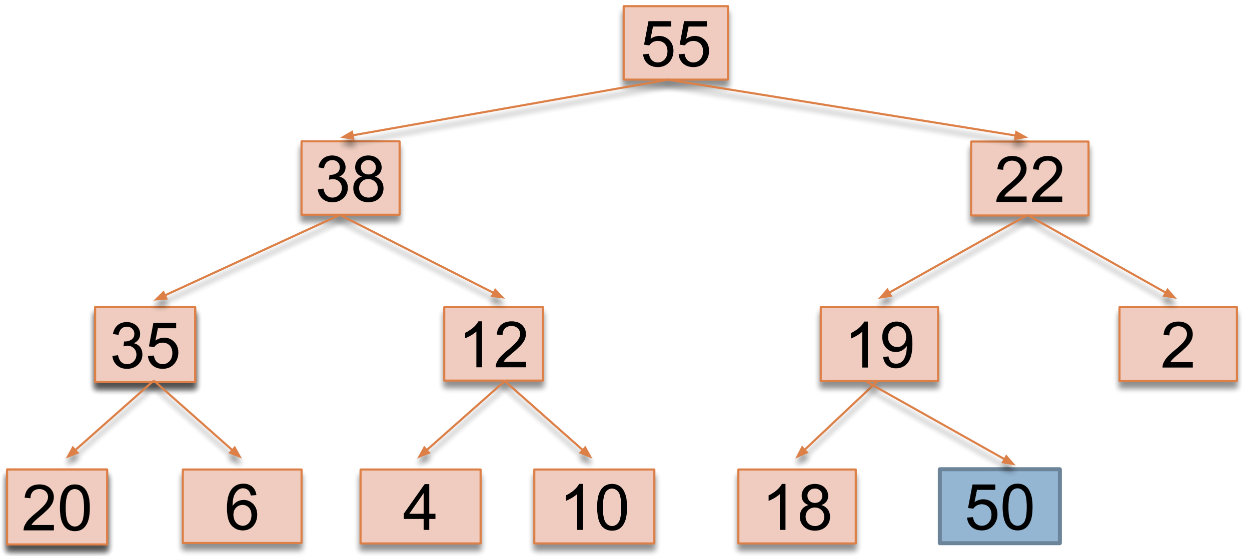


add(e)



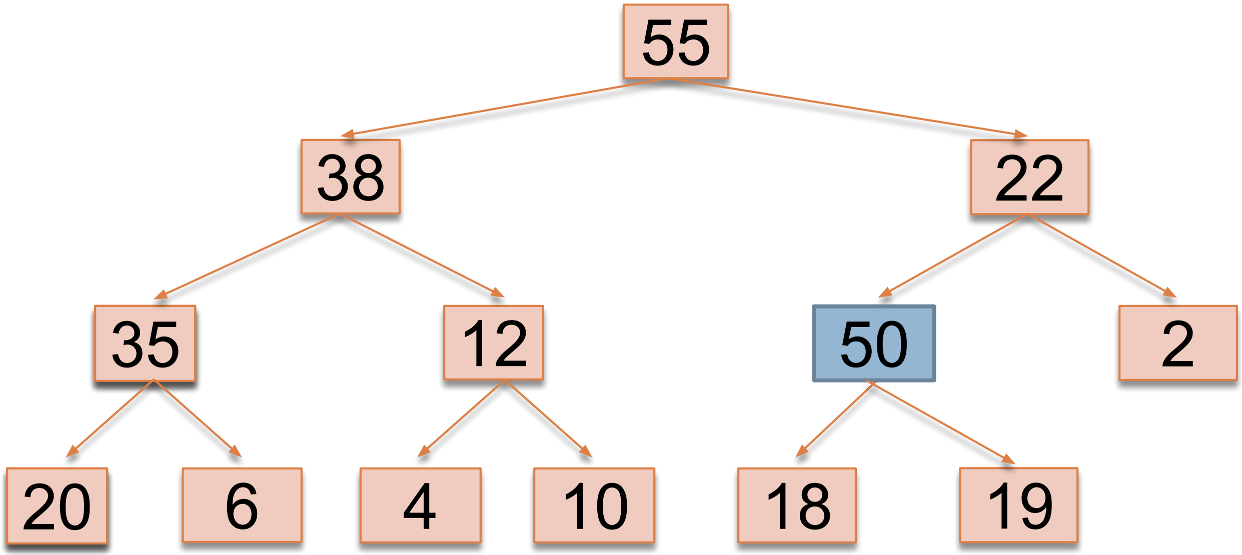
1. Put in the new element in a new node (leftmost empty leaf)

add(e)



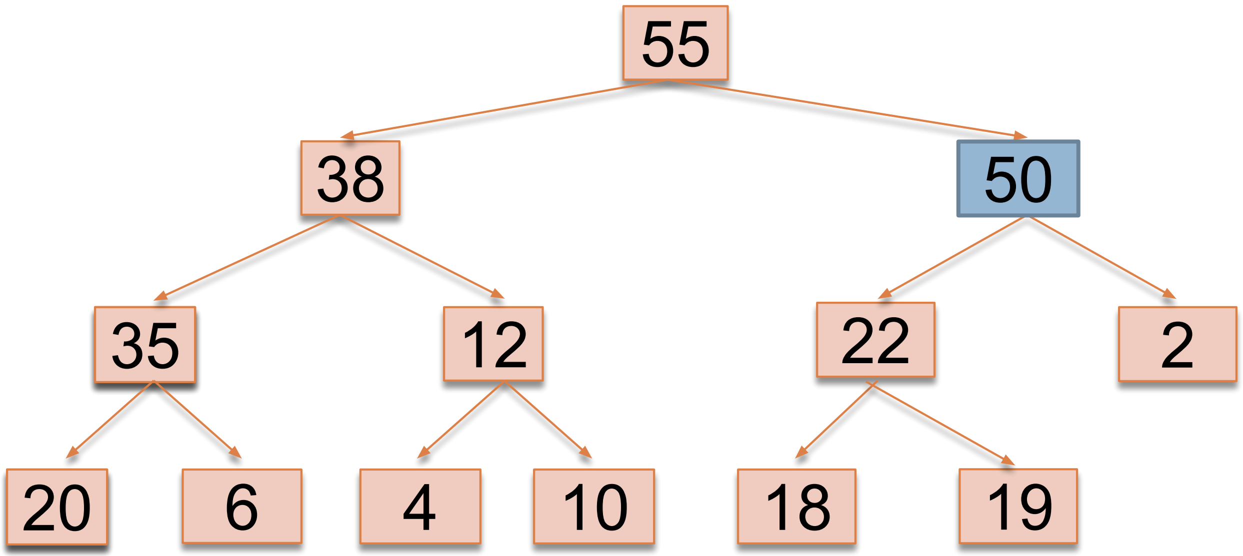
1. Put in the new element in a new node (leftmost empty leaf)
2. Bubble new element up while greater than parent

add(e)



1. Put in the new element in a new node (leftmost empty leaf)
2. Bubble new element up while greater than parent

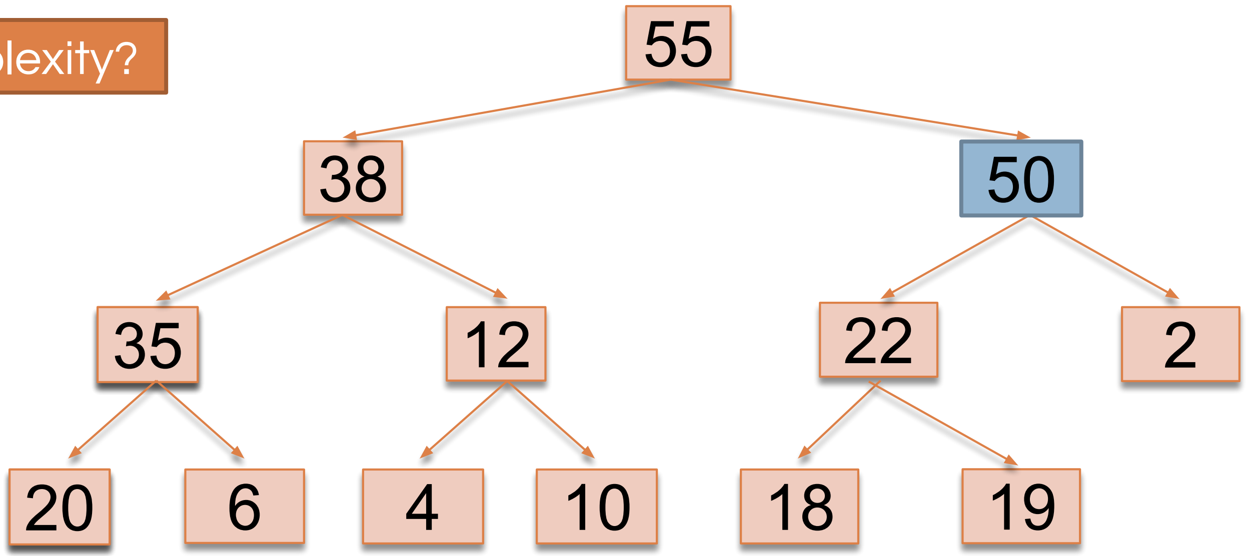
add(e)



1. Put in the new element in a new node (leftmost empty leaf)
2. Bubble new element up while greater than parent

add(e)

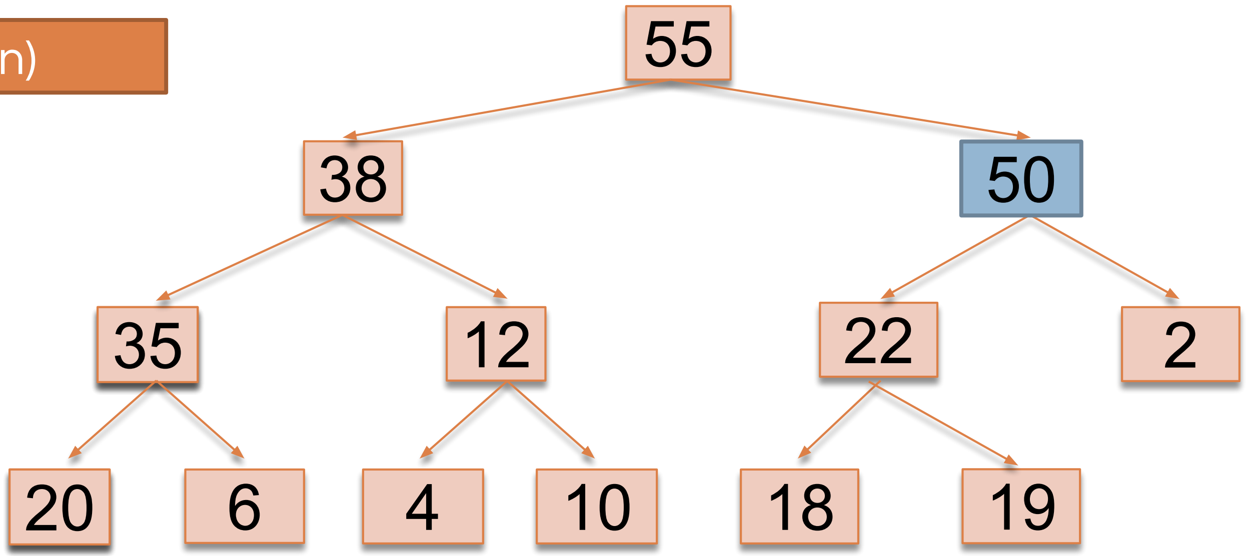
Complexity?



1. Put in the new element in a new node (leftmost empty leaf)
2. Bubble new element up while greater than parent

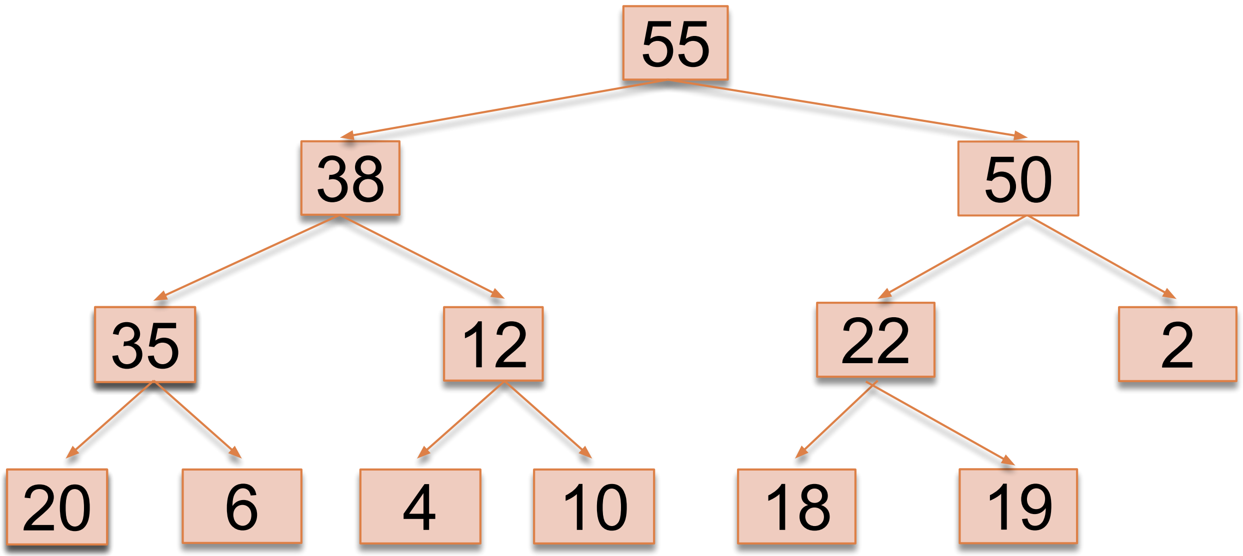
add(e)

O(log n)

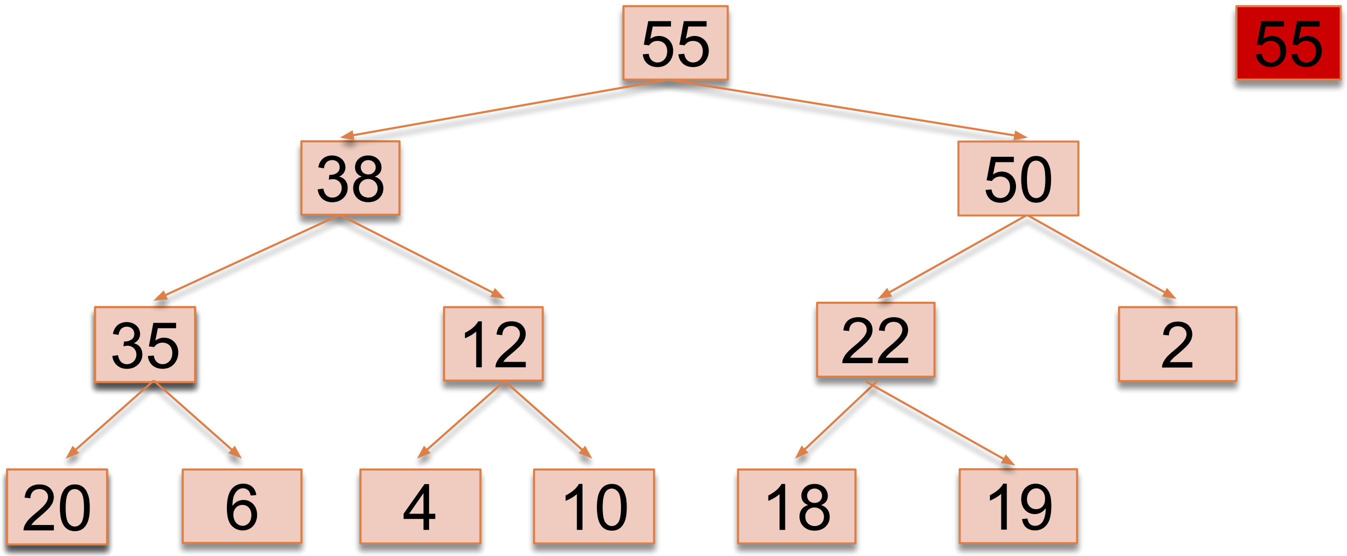


1. Put in the new element in a new node (leftmost empty leaf)
2. Bubble new element up while greater than parent

poll(e)

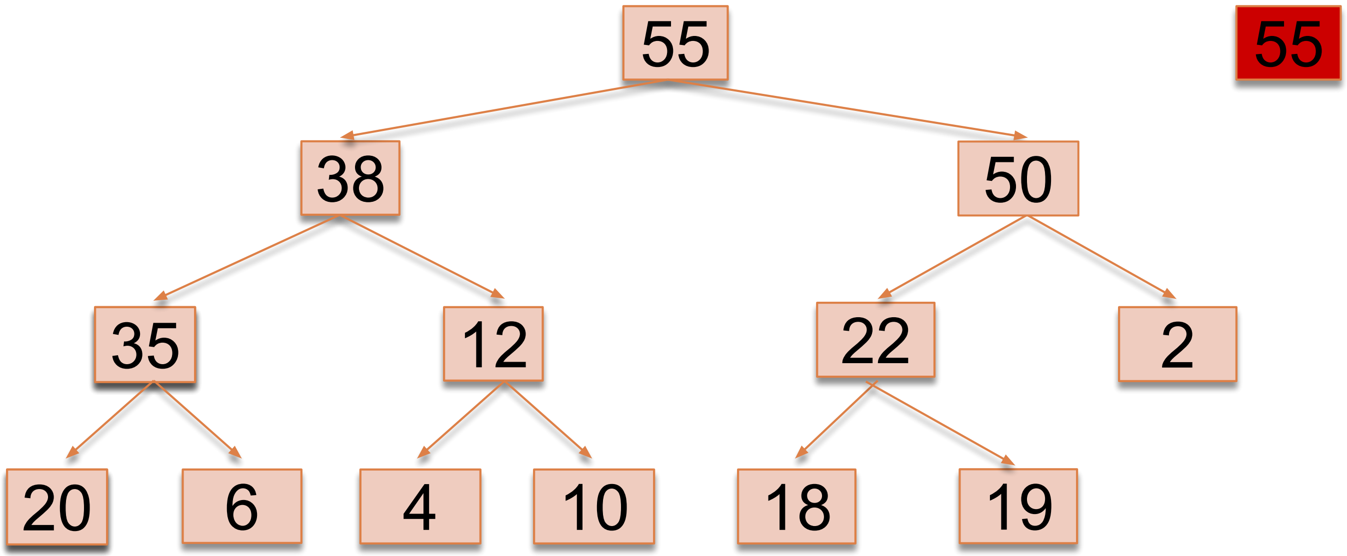


poll(e)



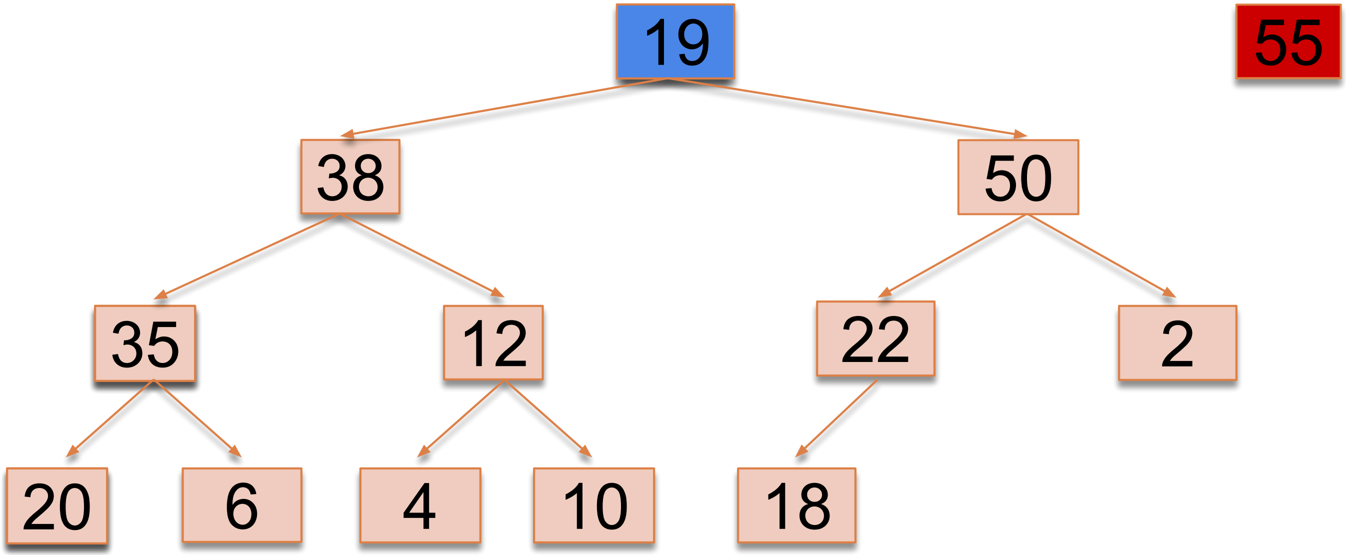
1. Save root element in a local variable

poll(e)



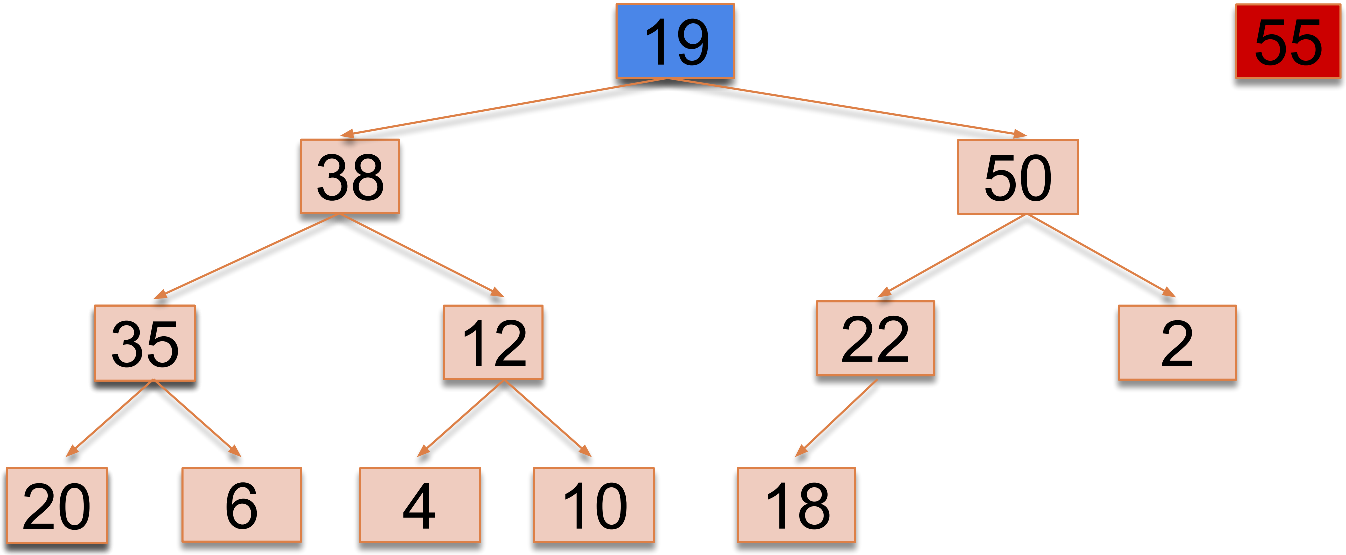
1. Save root element in a local variable
2. Assign last value to root, delete last node.

poll(e)



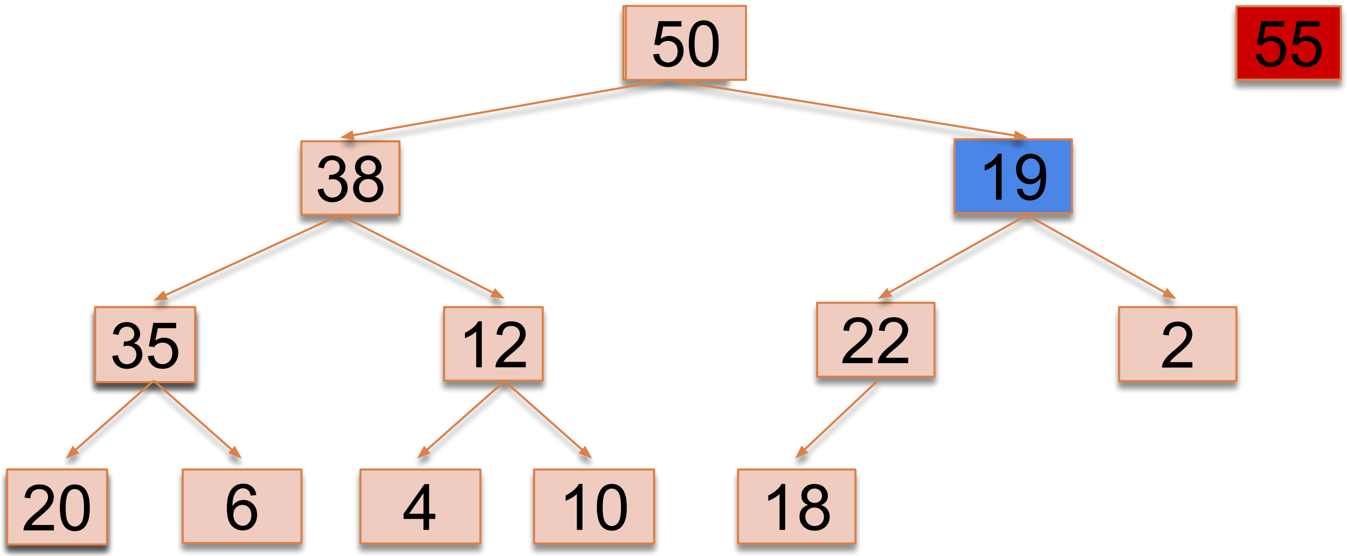
1. Save root element in a local variable
2. Assign last value to root, delete last node.

poll(e)



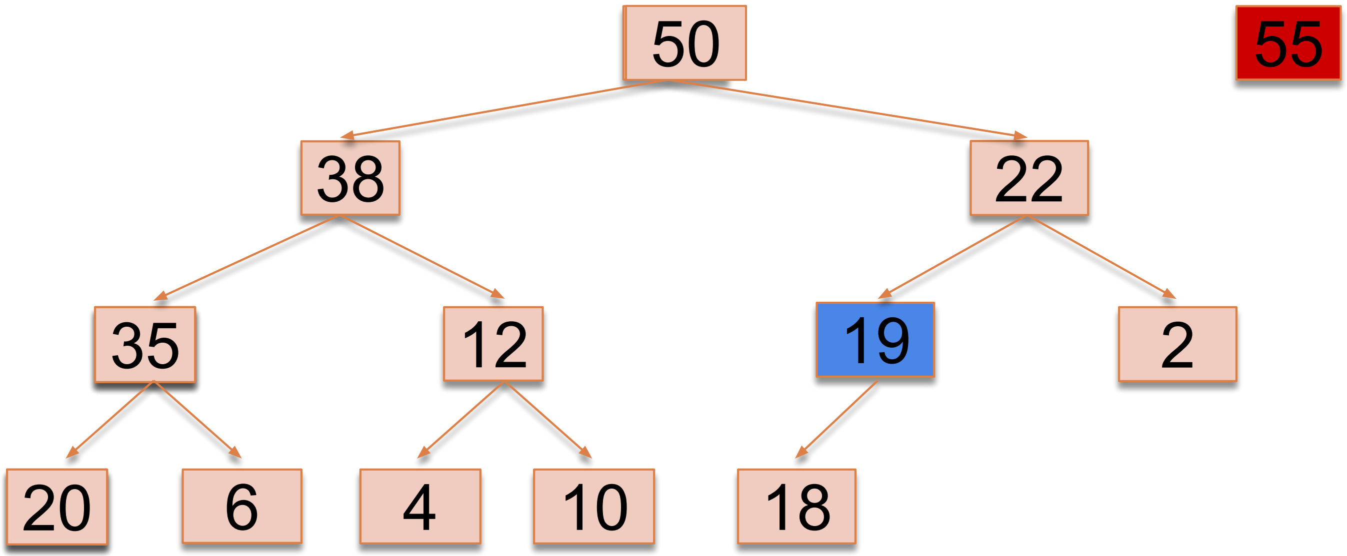
1. Save root element in a local variable
2. Assign last value to root, delete last node.
3. While less than a child, switch with bigger child (bubble down)

poll(e)



1. Save root element in a local variable
2. Assign last value to root, delete last node.
3. While less than a child, switch with bigger child (bubble down)

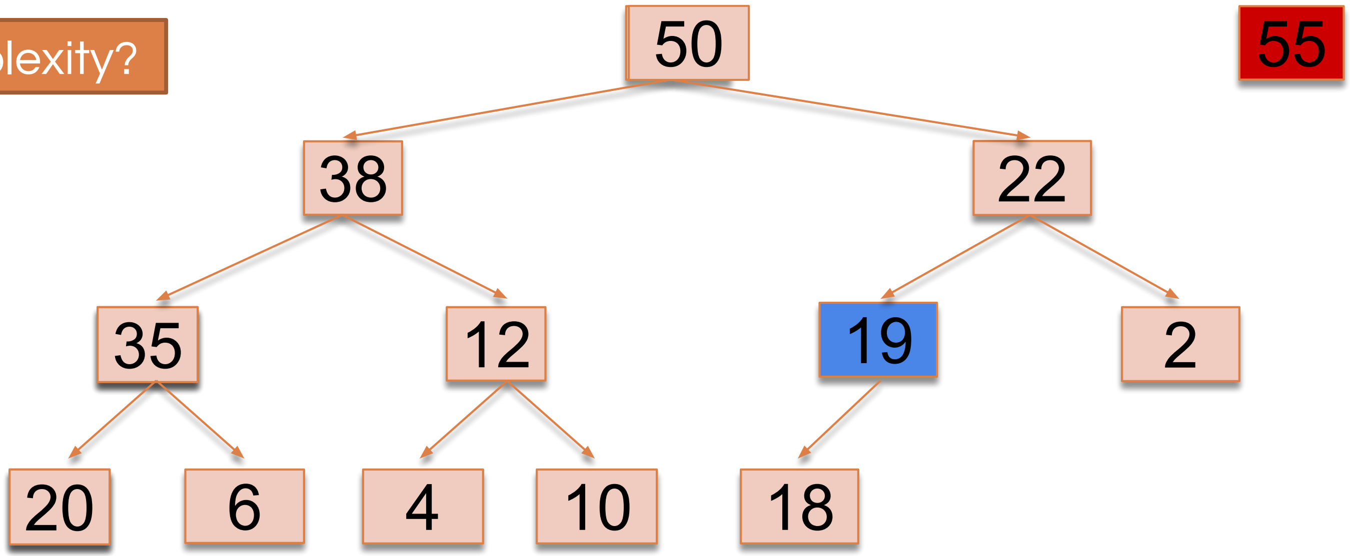
poll(e)



1. Save root element in a local variable
2. Assign last value to root, delete last node.
3. While less than a child, switch with bigger child (bubble down)

poll(e)

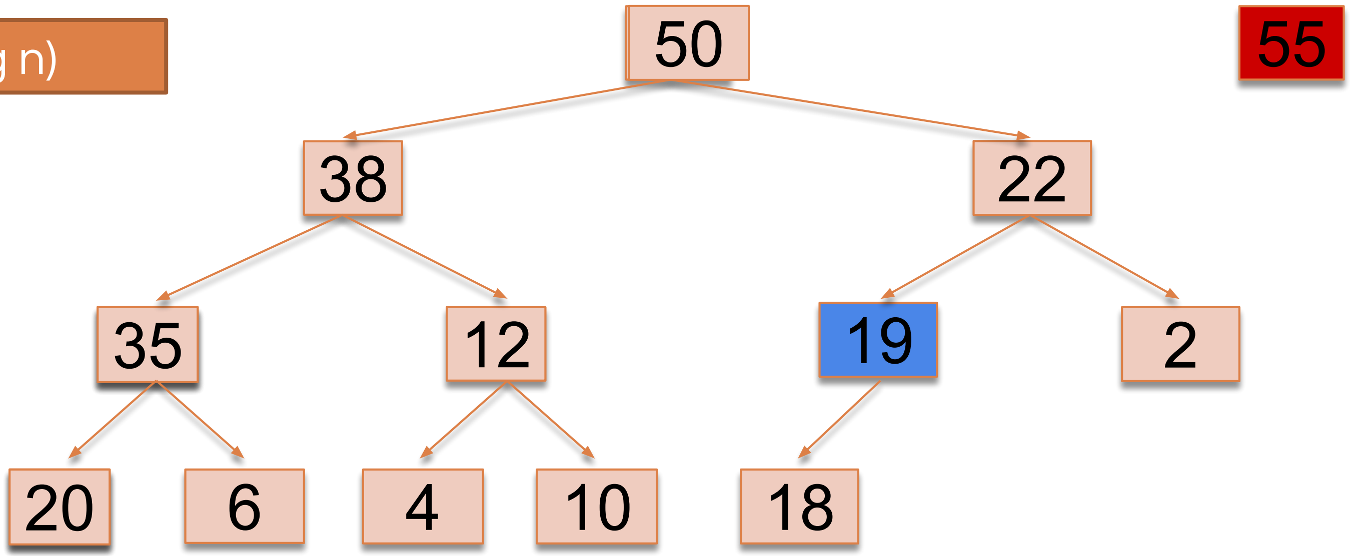
Complexity?



1. Save root element in a local variable
2. Assign last value to root, delete last node.
3. While less than a child, switch with bigger child (bubble down)

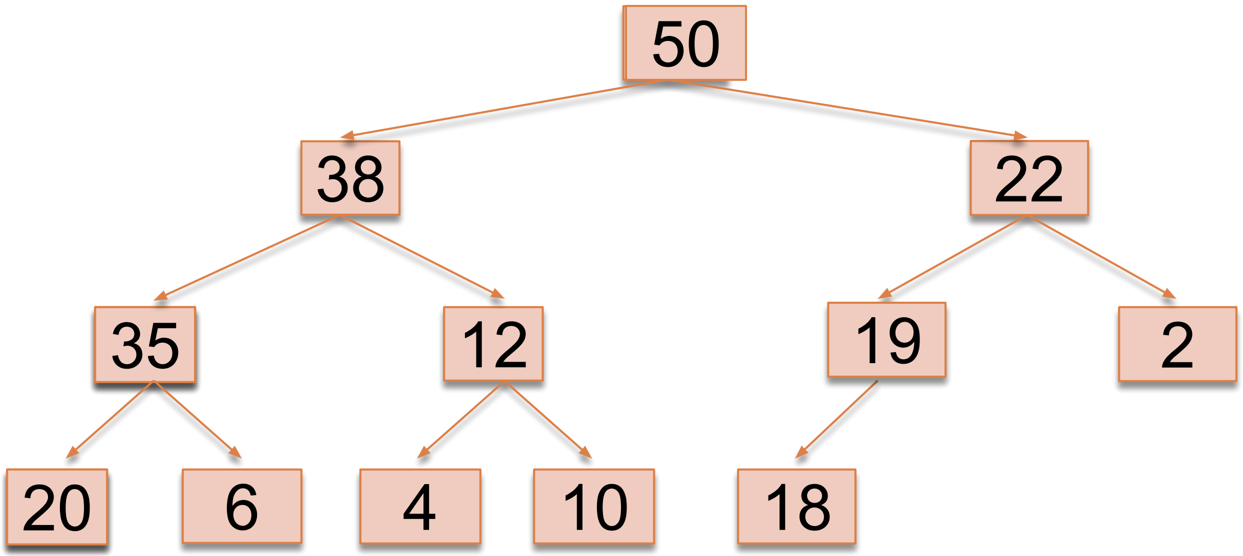
poll(e)

O (log n)



1. Save root element in a local variable
2. Assign last value to root, delete last node.
3. While less than a child, switch with bigger child (bubble down)

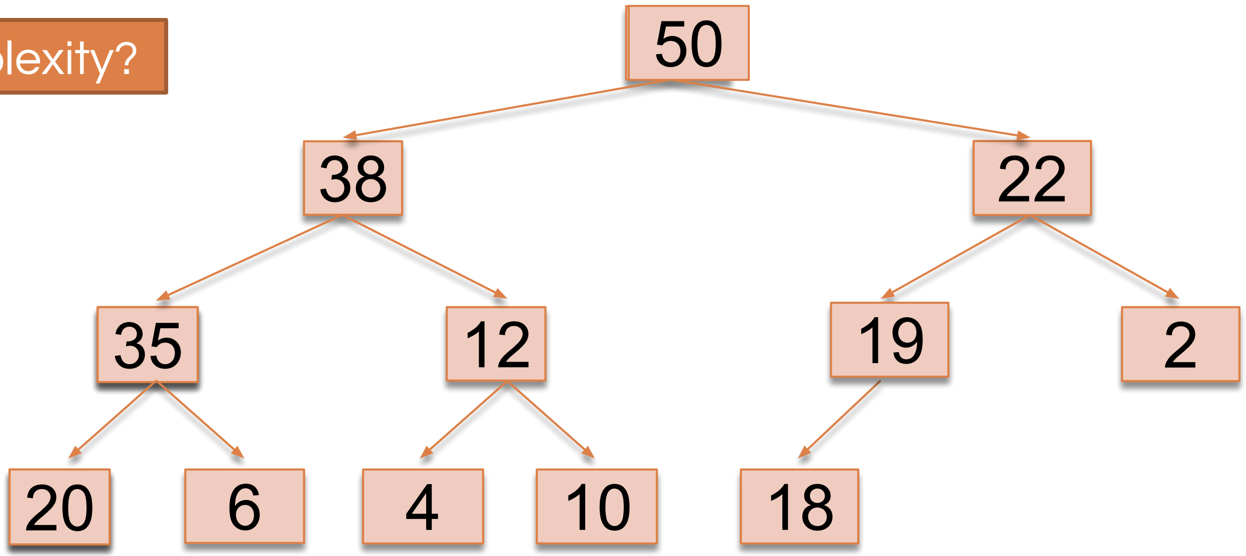
peek(e)



1. Return root value

peek(e)

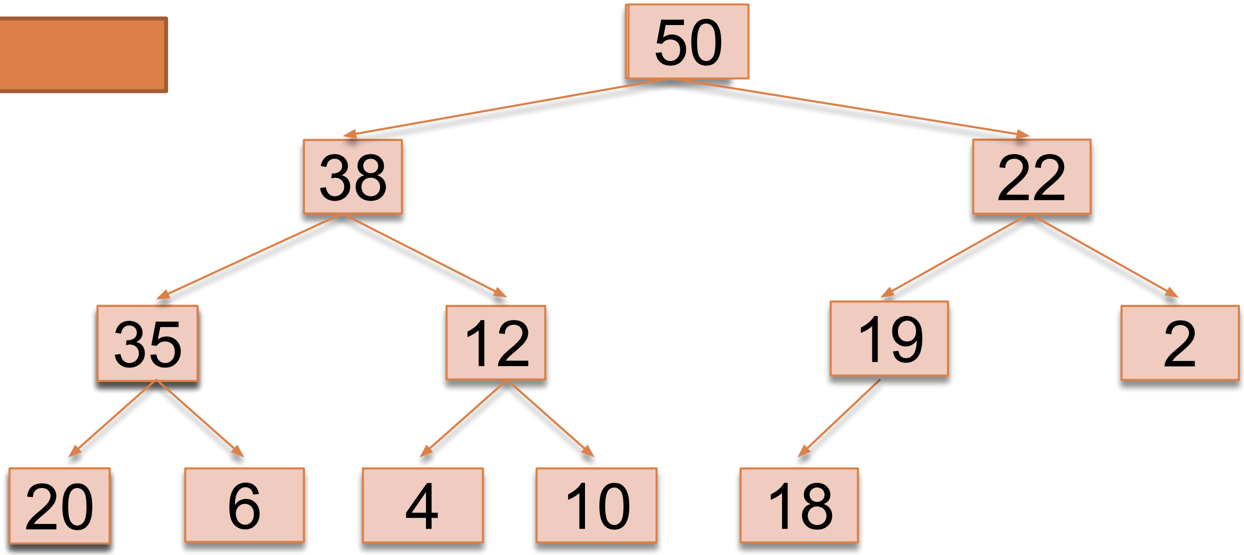
Complexity?



1. Return root value

peek(e)

O(1)



1. Return root value

Implementing Heaps

Implementing Heaps


```
public class HeapNode<E> {
    private E value;
    private HeapNode left;
    private HeapNode right;
    ...
}
```

Implementing Heaps


```
public class HeapNode<E> {
  private E value;
  private HeapNode left;
  private HeapNode right;
  ...
}
```

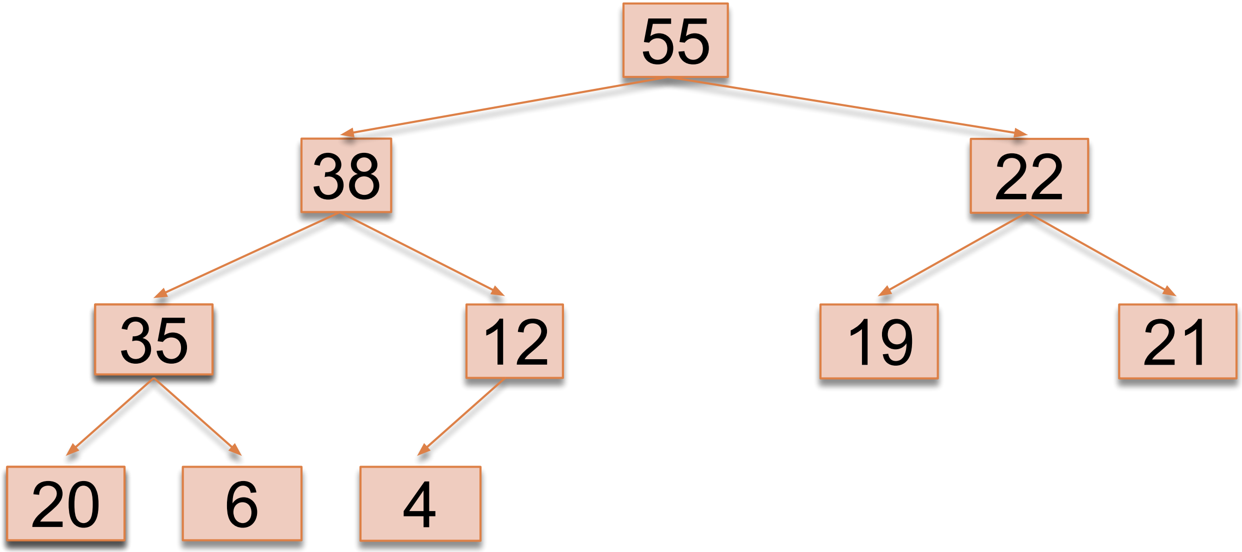
But remember that heaps are complete trees, we can do better!

Implementing Heaps


```
public class HeapNode<E> {
  private E[] value;
  ...
}
```

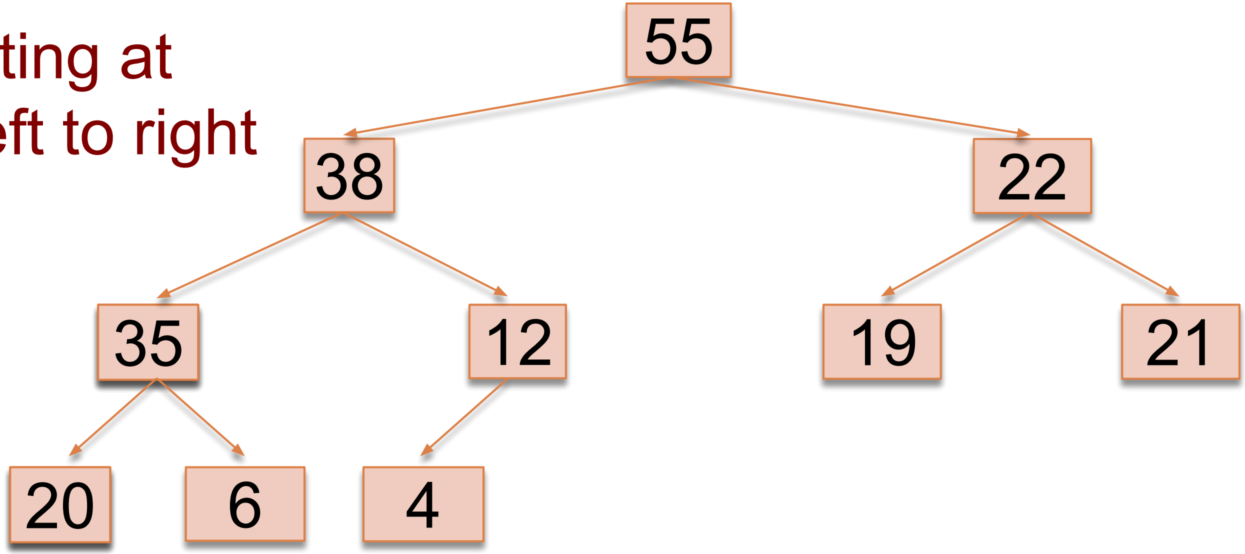
We can use arrays!

Numbering the nodes



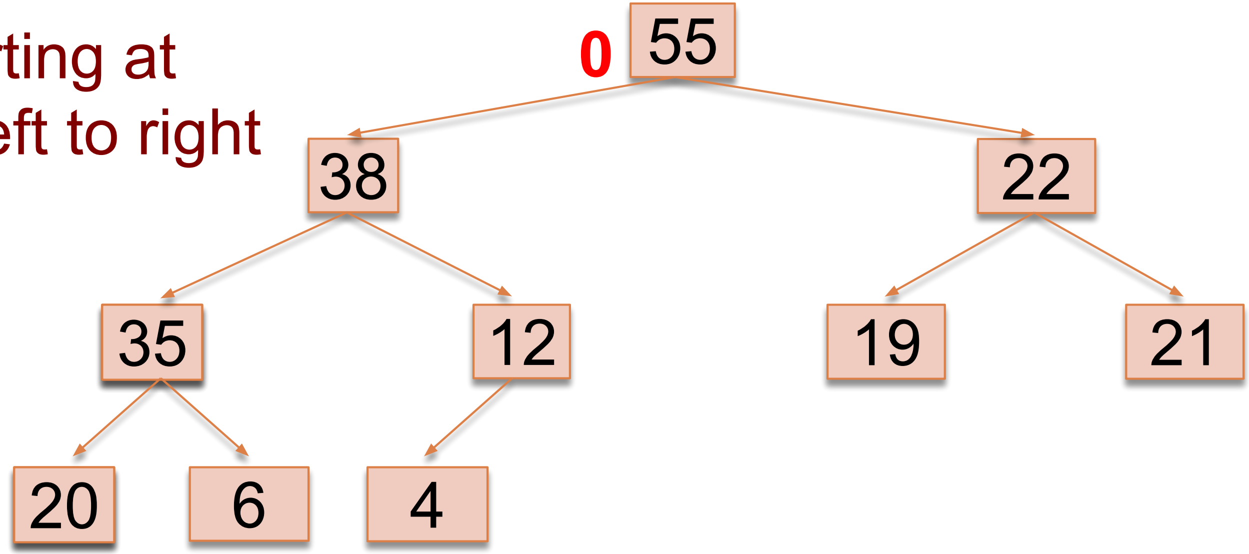
Numbering the nodes

Number node starting at root row by row, left to right



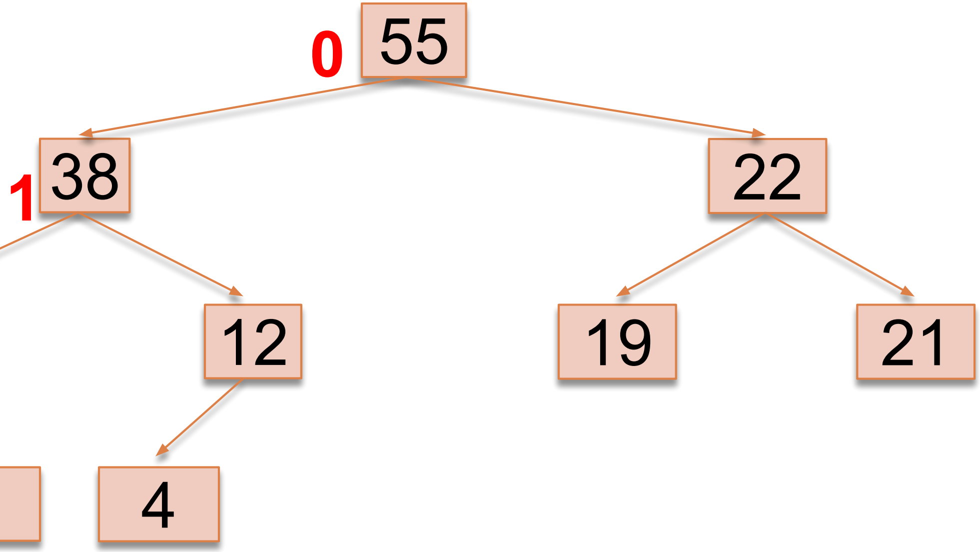
Numbering the nodes

Number node starting at root row by row, left to right



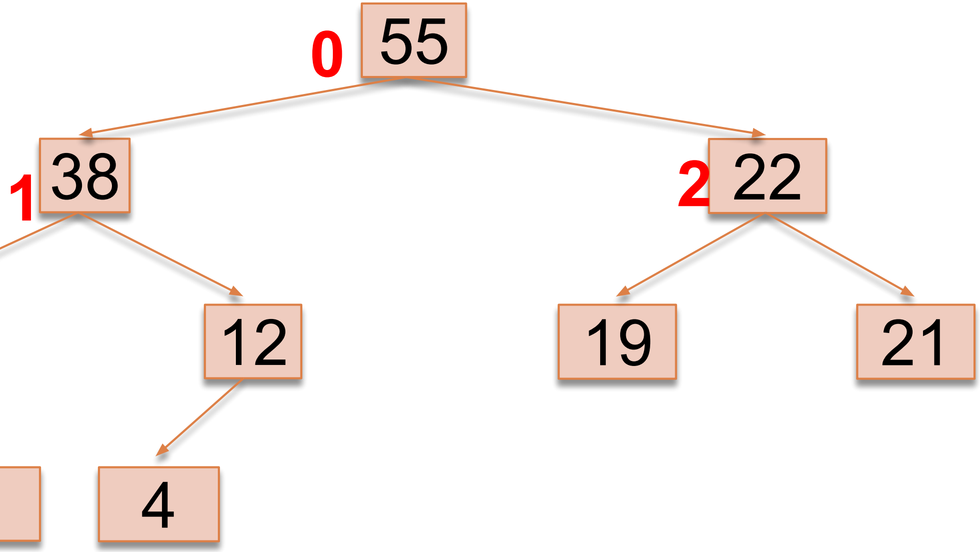
Numbering the nodes

Number node starting at root row by row, left to right



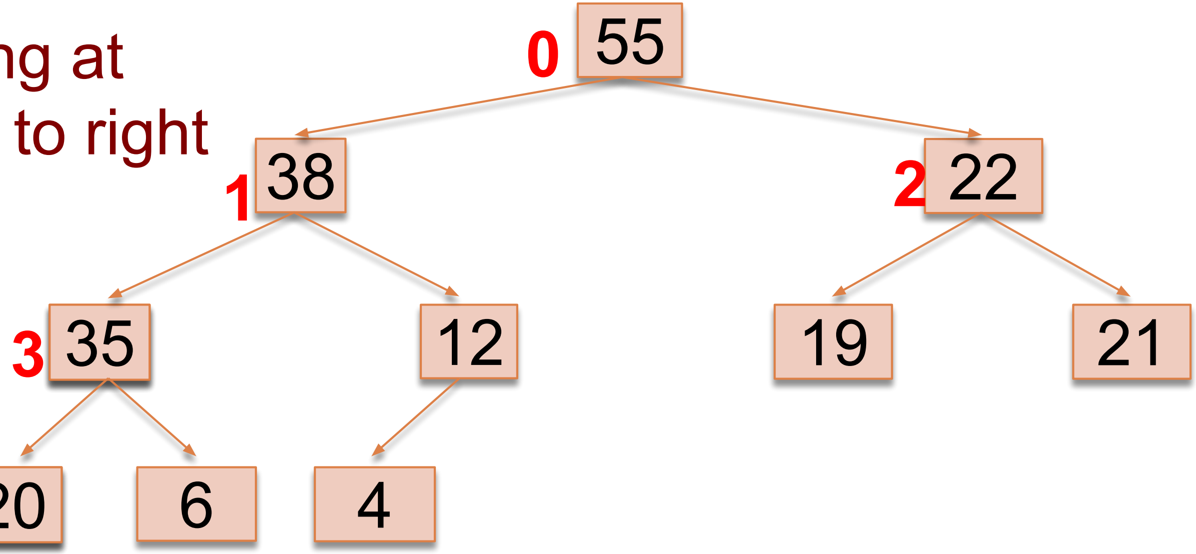
Numbering the nodes

Number node starting at root row by row, left to right



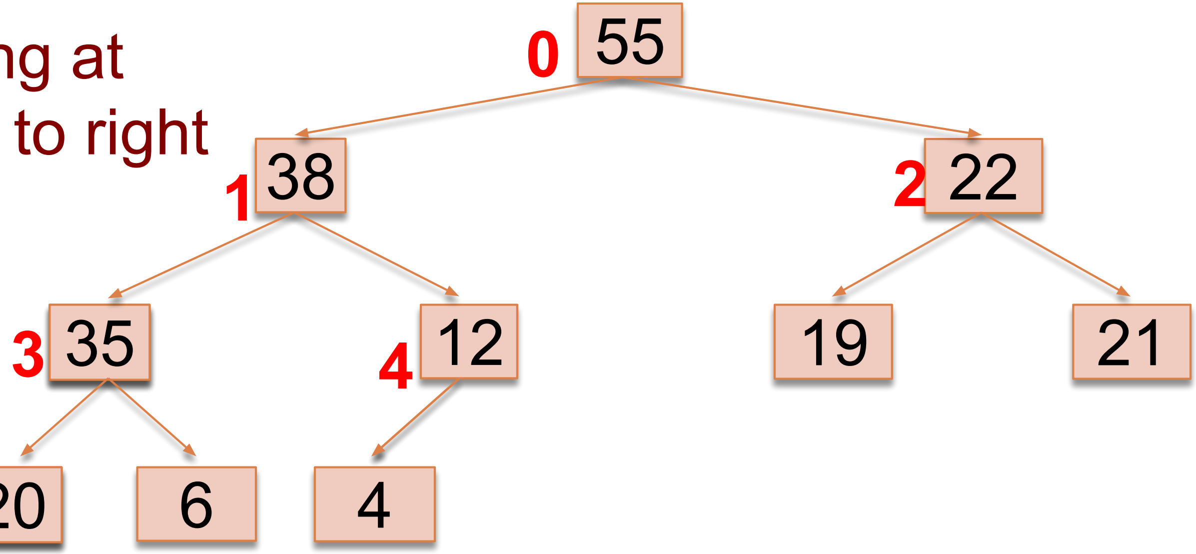
Numbering the nodes

Number node starting at root row by row, left to right



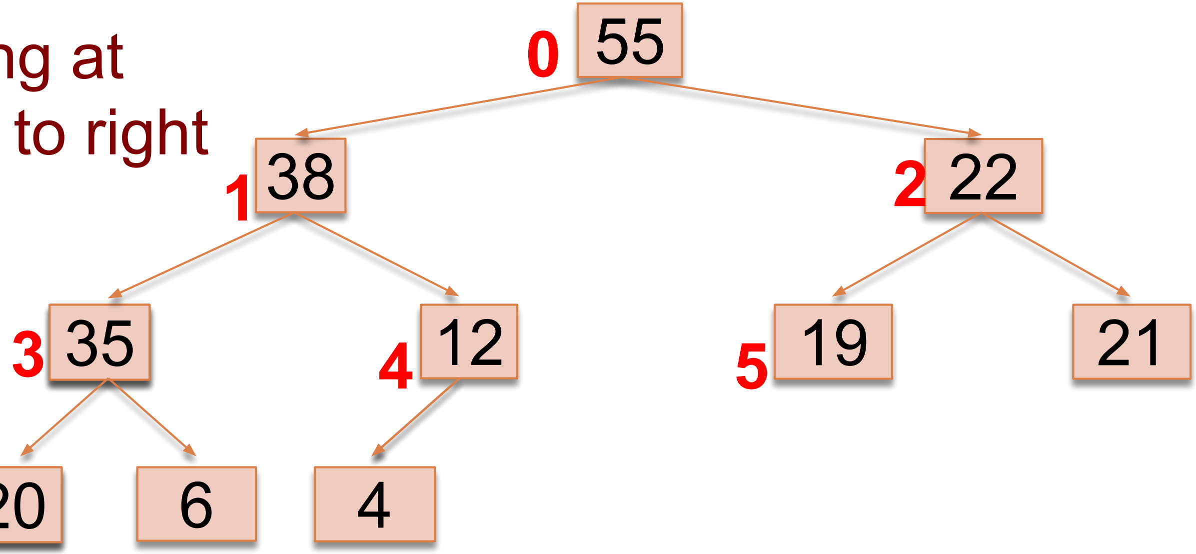
Numbering the nodes

Number node starting at root row by row, left to right



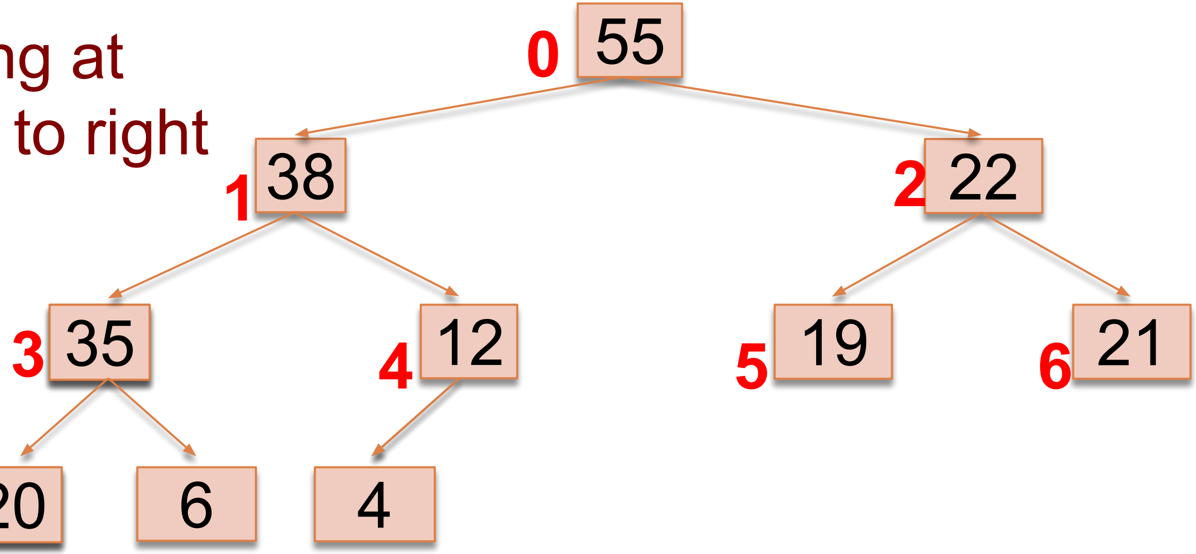
Numbering the nodes

Number node starting at root row by row, left to right



Numbering the nodes

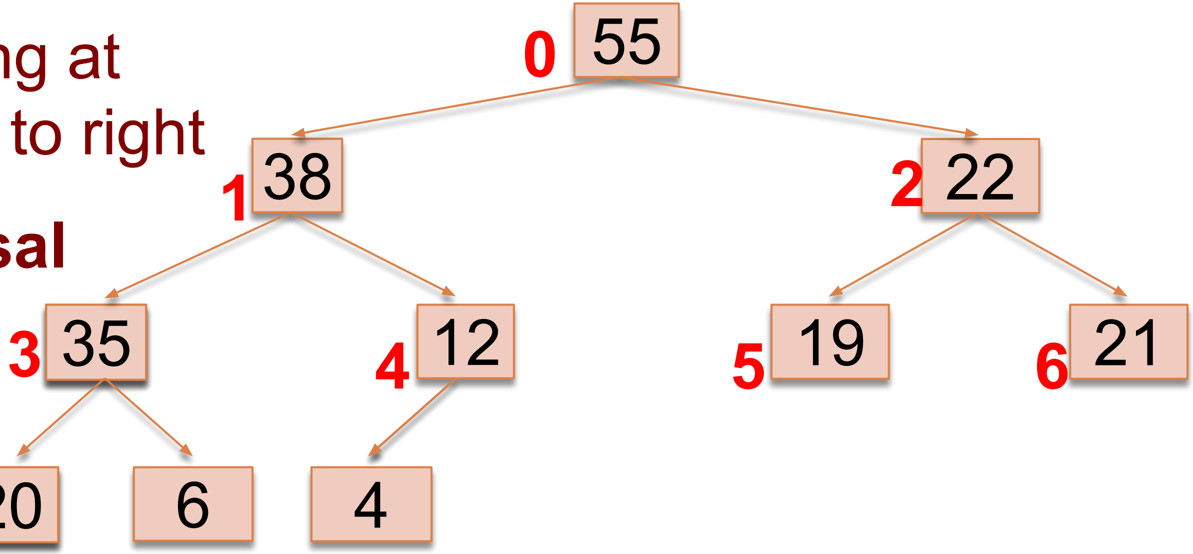
Number node starting at root row by row, left to right



Numbering the nodes

Number node starting at root row by row, left to right

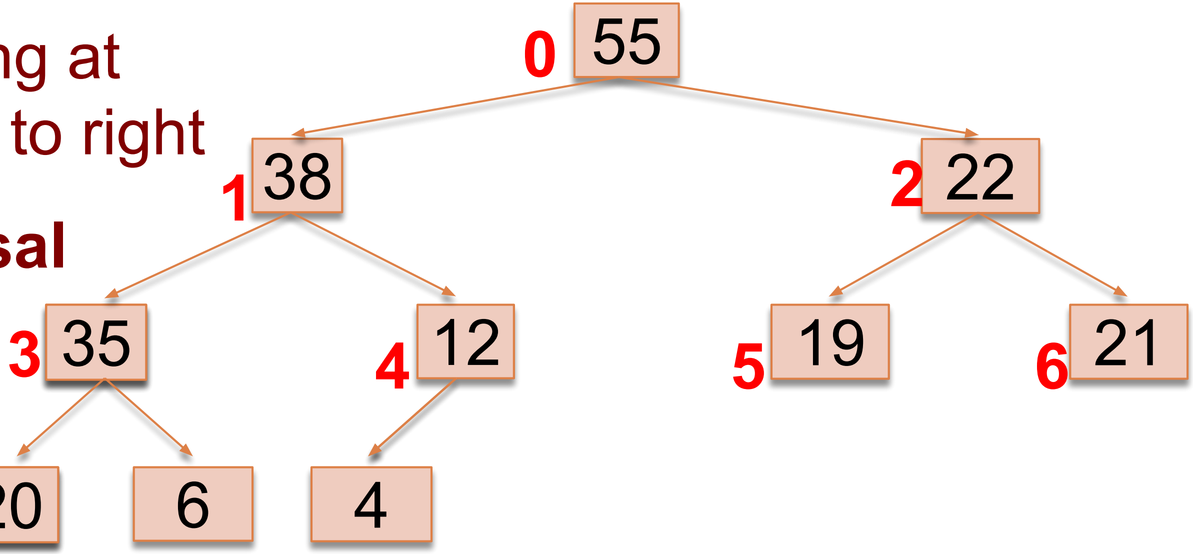
Level-order traversal



Numbering the nodes

Number node starting at root row by row, left to right

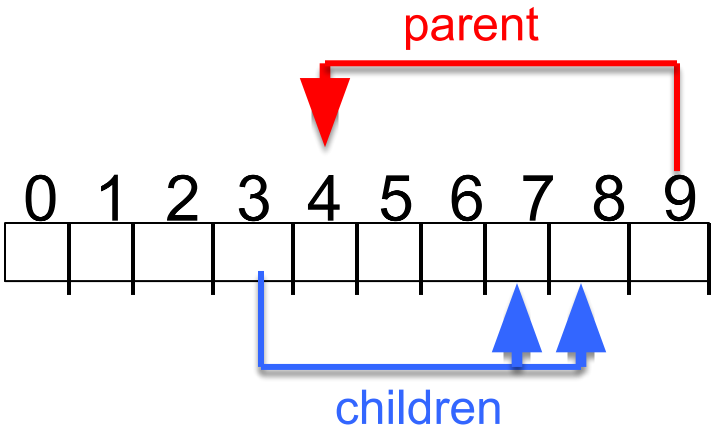
Level-order traversal

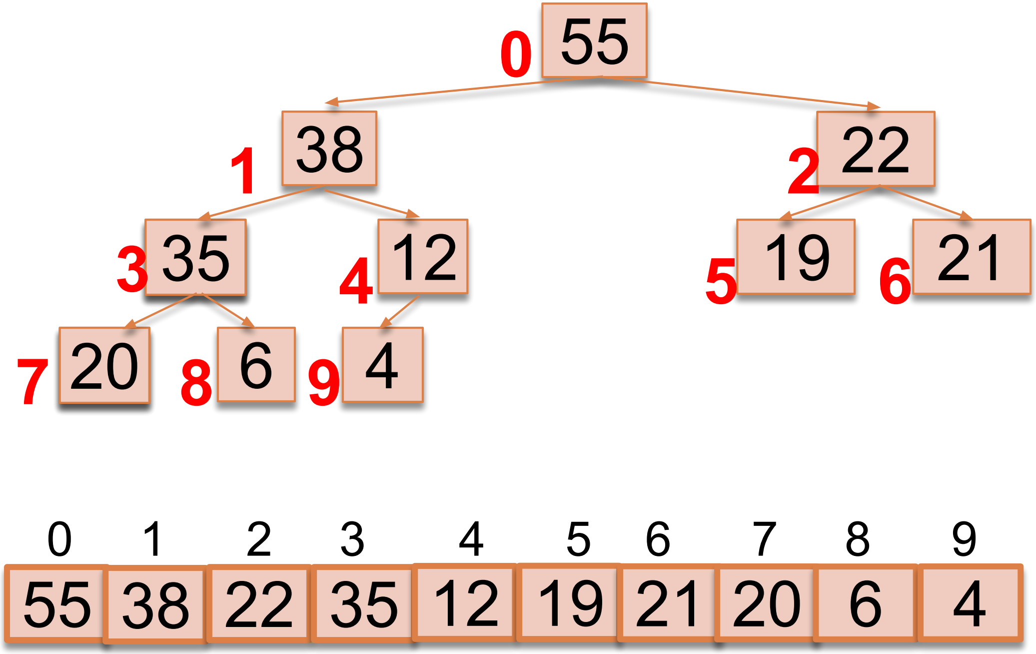


Children of node k are nodes 2k+1 and 2k+2
Parent of node k is node (k-1)/2

Storing a heap in an array

- Store node number i in index i of array b
- Children of b[k] are b[2k +1] and b[2k +2]
- Parent of b[k] is b[(k-1)/2]





add() --assuming there is space

add() --assuming there is space


```
/** An instance of a heap */
class Heap<E> {
  E[] b= new E[50];  // heap is b[0..n-1]
  int n= 0;          // heap invariant is true

  /** Add e to the heap */
  public void add(E e) {
    b[n]= e;
    n= n + 1;
    bubbleUp(n - 1); // given on next slide
  }
}
```

add() -- BubbleUp

add() -- BubbleUp


```
class Heap<E> {
  /** Bubble element #k up to its position.
    * Pre: heap inv holds except maybe for k */
  private void bubbleUp(int k) {
      int p = (k-1)/2
    // inv: p is parent of k and every element
    // except perhaps k is <= its parent
    while (k > 0  &&  b[k].compareTo(b[p]) > 0) {
                swap(b[k], b[p]);
            k= p;
            p= (k-1)/2;
        }
}
```

poll()

poll()


```
/** Remove and return the largest element
  * (return null if list is empty) */
public E poll() {
    if (n == 0) return null;
    E v=  b[0];   // largest value at root.
    b[0]= b[n];   // element to root
    n= n – 1;     // move last
    bubbleDown(0);
    return v;
}
```

poll()

poll()


```
/** Tree has n node.
 *  Return index of bigger child of node k
   (2k+2 if k >= n) */
public int biggerChild(int k, int n) {
   int c= 2*k + 2;     // k's right child
   if (c >= n || b[c-1] > b[c])
      c= c-1;
   return c;
}
```

poll()

poll()


```
/** Bubble root down to its heap position.
    Pre: b[0..n-1] is a heap except maybe b[0] */
private void bubbleDown() {
        int k  = 0;
        int c = biggerChild(k,n) ;
   // inv: b[0..n-1] is a heap except maybe b[k] AND
   //      b[c] is b[k]'s biggest child
   while (c<n && b[k] < b[c]  ) {
        swap(b[k], b[c]);
        k = c;
        c = biggerChild(k,n);
      }
}
```

peek()


```
/** Return the largest element
  * (return null if list is empty) */
public E peek() {
    if (n == 0) return null;
    return b[0];   // largest value at root.
}
```

Let's try it!

Here's a heap, stored in an array: [9 5 2 1 2 2]

What is the state of the array after execution of add(6)? Assume the existing array is large enough to store the additional element.

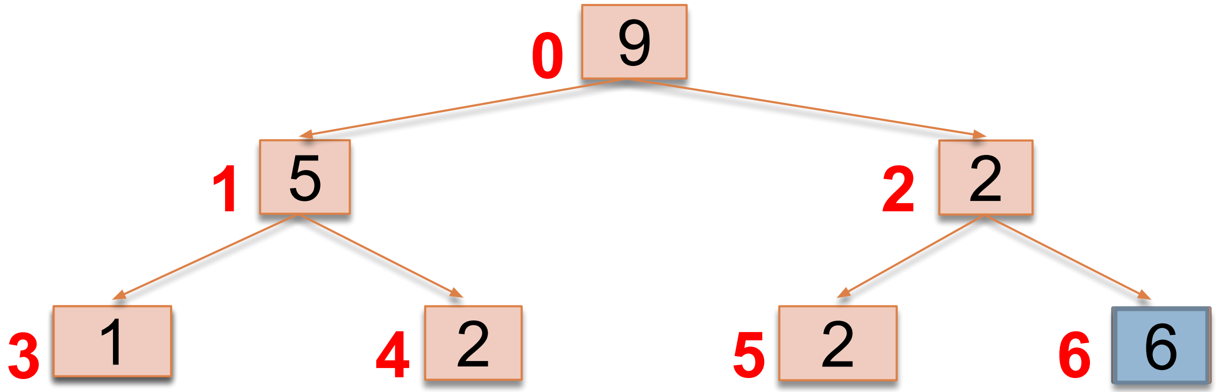
A. [9 5 2 1 2 2 6]
B. [9 5 6 1 2 2 2]
C. [9 6 5 1 2 2 2]
D. [9 6 5 2 1 2 2]

Let's try it!

Here's a heap, stored in an array:

[9 5 2 1 2 2]

Write the array after execution of add(6)

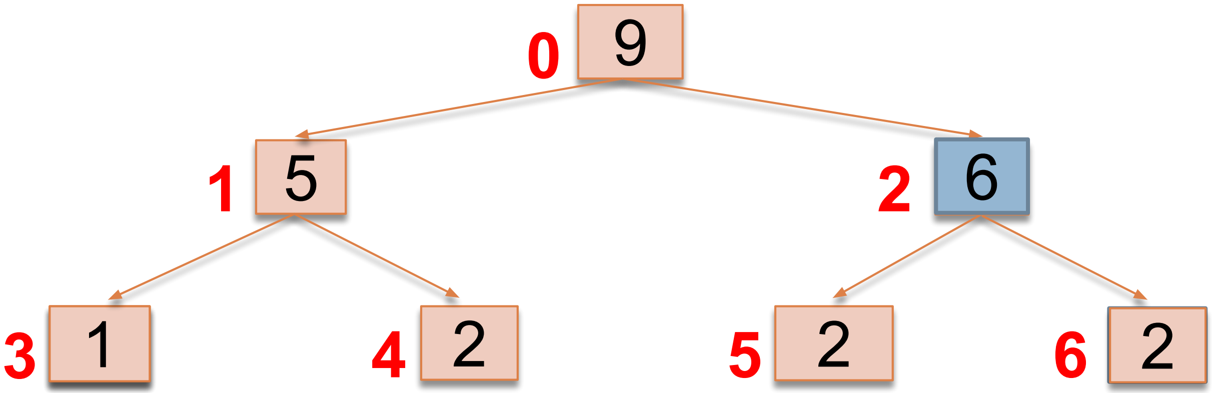


Let's try it!

Here's a heap, stored in an array:

[9 5 2 1 2 2]

Write the array after execution of add(6)



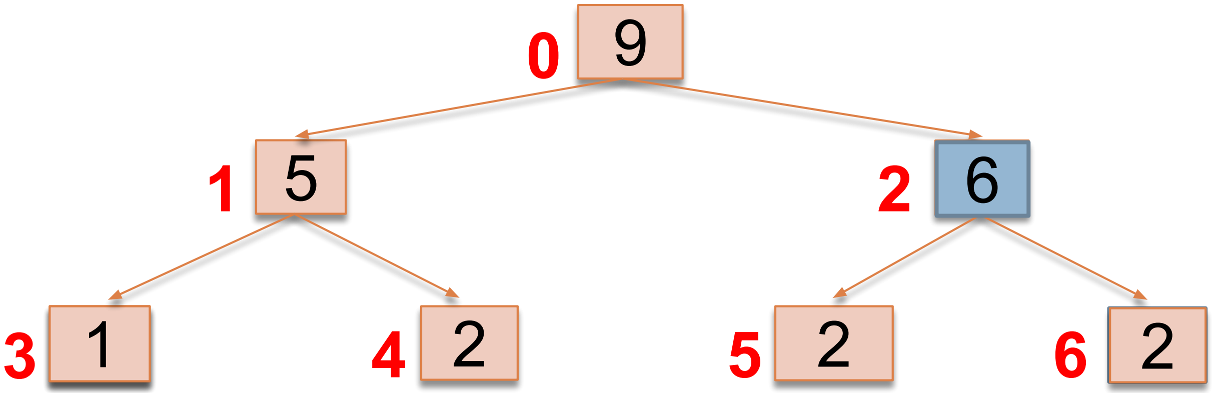
Let's try it!

Here's a heap, stored in an array:

[9 5 2 1 2 2]

Write the array after execution of add(6)

⇒ [9 5 6 1 2 2 2]



Heap Sort

- Can we use a heap to sort an array?

Heap Sort

- Can we use a heap to sort an array?
- We said that heaps weren't great for "finding" an element in the array arbitrarily
- But they're pretty good at finding the minimum/maximum
- What can we do?

Heap Sort

- Create a heap of the n elements in the array
- Repeatedly extract the minimum (or maximum) until the heap is empty

Heap Sort

- Create a heap of the n elements in the array
- Repeatedly extract the minimum (or maximum) until the heap is empty
- What's the cost of creating a heap consisting of n elements?

Heap Sort

- Create a heap of the n elements in the array
- Repeatedly extract the minimum (or maximum) until the heap is empty
- What's the cost of creating a heap consisting of n elements?
  - Equivalent to the cost of inserting n elements into a heap

Heap Sort

- Create a heap of the n elements in the array
- Repeatedly extract the minimum (or maximum) until the heap is empty
- What's the cost of creating a heap consisting of n elements?
  - Equivalent to the cost of inserting n elements into a heap
  - n * log (n)

Heap Sort

- Create a heap of the n elements in the array
- Repeatedly extract the minimum (or maximum) until the heap is empty
- What's the cost of creating a heap consisting of n elements?
  - Equivalent to the cost of inserting n elements into a heap
  - n * log (n)
- What's the cost of extracting the minimum n times?

Heap Sort

- Create a heap of the n elements in the array
- Repeatedly extract the minimum (or maximum) until the heap is empty
- What's the cost of creating a heap consisting of n elements?
  - Equivalent to the cost of inserting n elements into a heap
  - n * log (n)
- What's the cost of extracting the minimum n times?
  - n * log (n)

Heap Sort

- Create a heap of the n elements in the array
- Repeatedly extract the minimum (or maximum) until the heap is empty
- What’s the cost of creating a heap consisting of n elements?
  - Equivalent to the cost of inserting n elements into a heap
  - n * log (n)
- What’s the cost of extracting the minimum n times?
  - n * log (n)
- So n * log (n) + n * log (n): Heapsort is O(log n)!