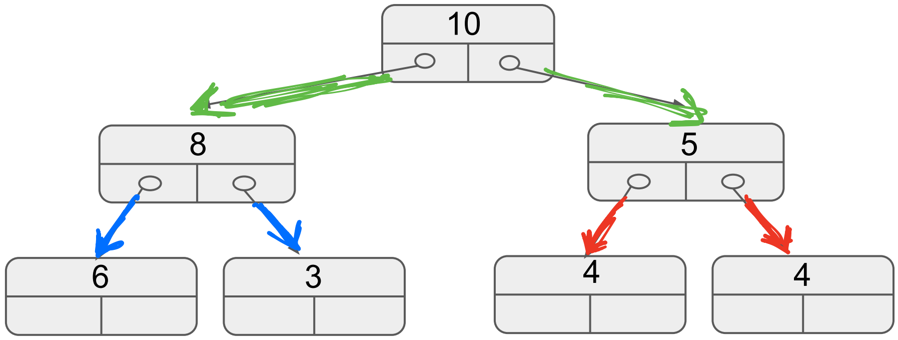
CSE12W19-Mar1-W8-F-20-1

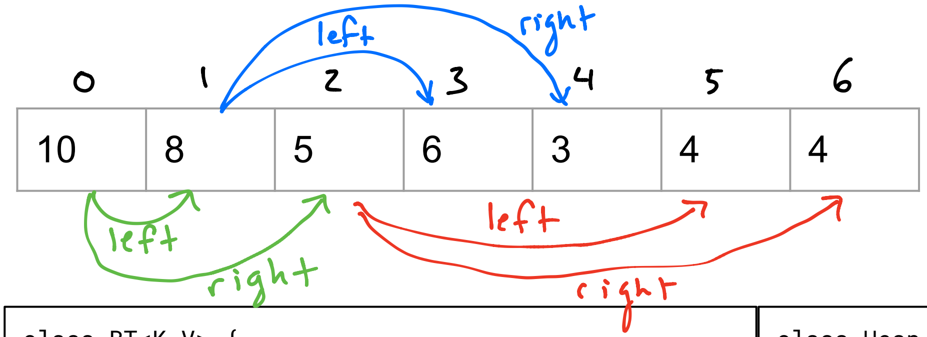
Definition: A tree is a **complete tree** if every level but the last level is completely full, and the last level has its nodes all the way to the **left**.

Property: A complete tree's size and height are related by: *height ~ log(size)*

Definition: A tree is in **max** (min) **heap order** if every node's key is **greater** (less) than or equal to all of its childrens' keys.

Definition: A **max** (min) **heap** is a **complete tree** that is in **max** (min) **heap order**.





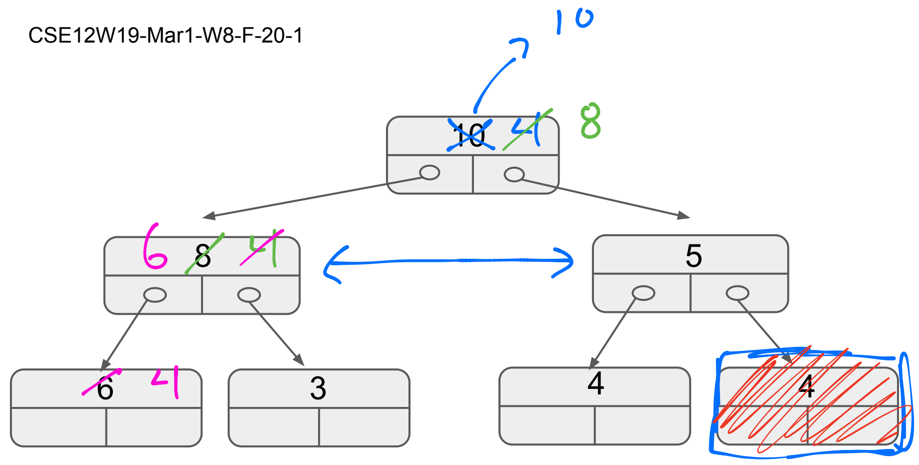
0	1	2	3	4	5	6
10	8	5	6	3	4	4

Key	index	Parent	Left	Right
10	0	N/A	1	2
8	1	0	3	4
5	2	0	5	6
6	3	1	N/A	N/A
3	4	1	N/A	N/A
4	5	2	N/A	N/A
4	6	2	N/A	N/A

```
class BT<K,V> {

Node<K,V> root;

...

V get(Node<K,V> node, K key) {
  if(node == null) { return null; }

  if(node.key.equals(key)) { return node.value; }

  V leftResult = get(node.left, key);
  V rightResult = get(node.right, key);
  if(leftResult != null) { return leftResult; }
  if(rightResult != null) { return rightResult; }
  return null;
}

}
```

```
class Heap<K,V> {

List<Entry<K,V>> entries;

int left(int index) {
   return index * 2 + 1;

}

int right(int index) {
   return index * 2 + 2;

}

V get(int index, K key) {
  if(index >= this.entries.size()) { return null; }
  Entry<K,V> entry = entries.get(index);

  if(entry.key.equals(key)) { return entry.value; }

  V leftResult = get(left(index), key);
  V rightResult = get(right(index), key);
  if(leftResult != null) { return leftResult; }
  if(rightResult != null) { return rightResult; }
  return null;
}

}
```

CSE12W19-Mar1-W8-F-20-1



```
K poll()

// remove and return largest element
```

Θ(height)

Θ(lg(size))

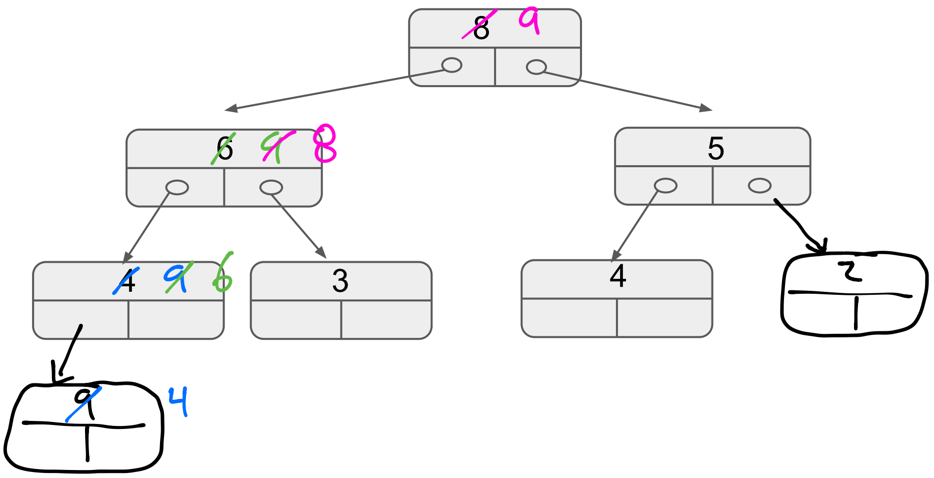
log₂

bubbleDown

swap with larger child

~~10~~ ~~4~~ 8	~~8~~ ~~4~~ 6	5	~~6~~ 4	3	4	~~4~~

```
void add(K k)
// add the element, ensuring heap-ness

add(2)

add(9)
```



Θ(height)

Θ(lg(size))

bubbleUp

~~8~~ 9	~~6~~ ~~9~~ 8	5	~~4~~ ~~9~~ 6	3	4	2	~~9~~ 4

```
void bubbleDown(int index) {
  if(index >= this.entries.size()) { return; }
  int leftIndex = left(index);
  if(leftIndex >= this.entries.size()) { return; }
  int largerChildIndex = leftIndex;
  int rightIndex = right(index);
  if(existsAndGreater(rightIndex, leftIndex)) {
    largerChildIndex = rightIndex;
  }
  if(existsAndGreater(largerChildIndex, index)) {
    swap(index, largerChildIndex);
    bubbleDown(largerChildIndex);
  }
}
```

```
void bubbleUp(int index) {
  if(index <= 0) { return; }
  Entry<K,V> e = this.entries.get(index);
  Entry<K,V> parent = this.entries.get(parent(index));
  int comp = this.comparator.compare(e.key, parent.key);
  if(comp > 0) {
    swap(index, parent(index));
    bubbleUp(parent(index));
  }
  else {
    return;
  }
}
```

```
List<E> {
  void add(E e)
  E get(int index)
}
```

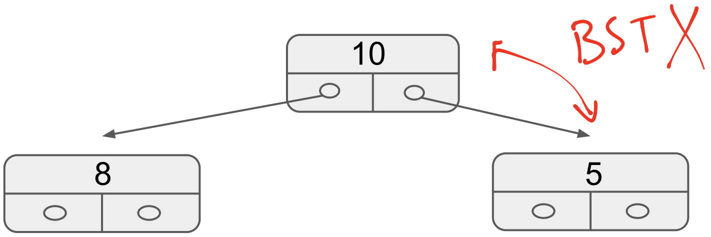
```
Queue<E> {
  void enqueue(E e)
  E dequeue()
}
```

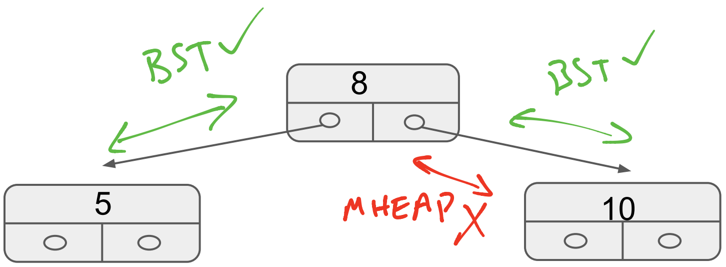
```
Stack<E> {
  void push(E e)
  E pop()
}
```

```
Map<K,V> {
  void set(K k, V v);
  V get(K k);
}
```

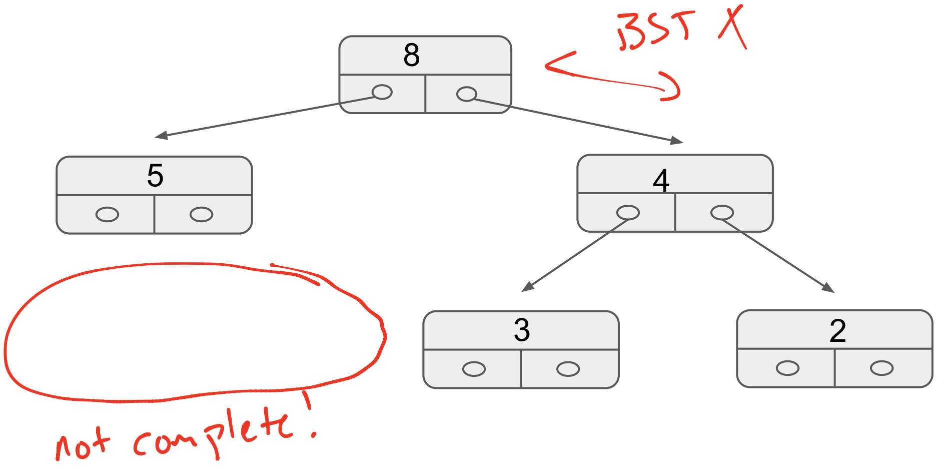
```
PriorityQueue<K,V> {
    void set(K k, V v)
    V  poll();
Entry<K,V> poll();
}
```

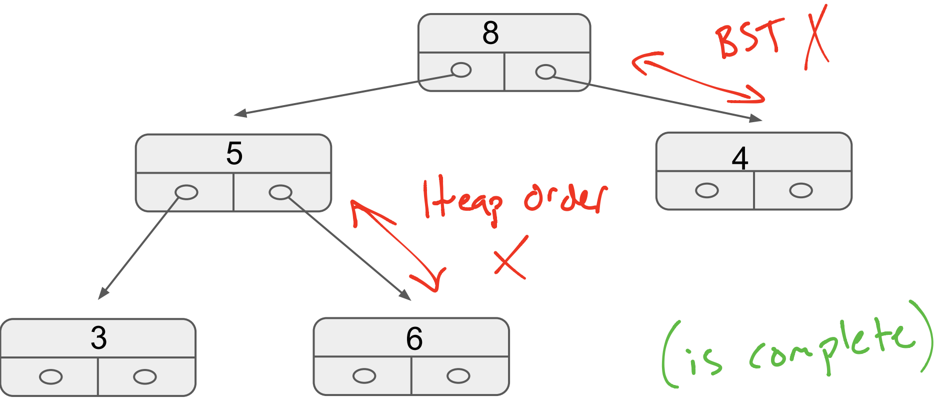
↑ remove and return max/largest key





BST A	Max Heap B	Neither C	Both D
✗	✓	✗	✗
✓	✗	✗	✗





BST	Max Heap	Neither	Both
✗	✗	✓	✗
✗	✗	✓	✗