Heaps and Priority Queues

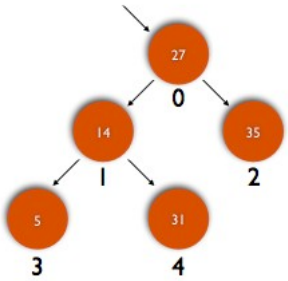
Definitions

- a complete binary tree is a binary tree (not necessarily ordered) that has has each level filled (left to right) before the next level is filled
- a heap is a complete binary tree (not ordered) such that the root node is larger than its two children, and that its children are also heaps

Implementing Complete Binary Trees

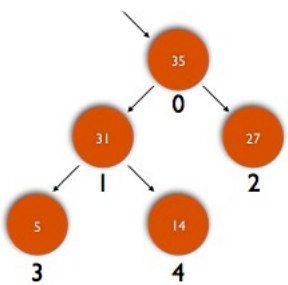
- complete binary trees can be implemented efficiently using arrays — their completeness assures no gaps in the array
- the root node is the value at index 0
- the left child of node i is found at index 2i + 1
- the right child of node i is found at index 2i + 2
- the parent of node i is found at index (i - 1)/2

Complete Binary Tree as an Array



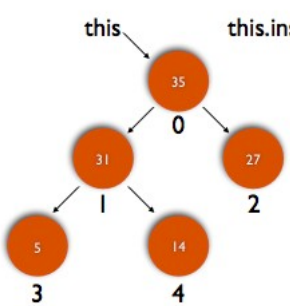
0	27
1	14
2	35
3	5
4	31

Heap as an Array



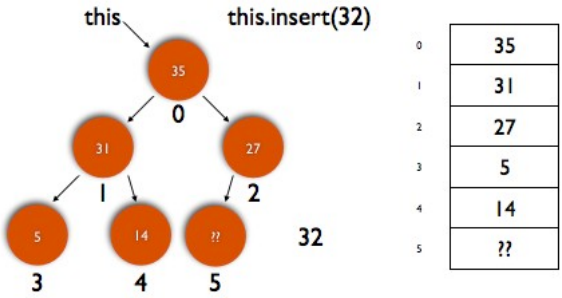
0	35
1	31
2	27
3	5
4	14

Inserting into a Heap in O(log n) time



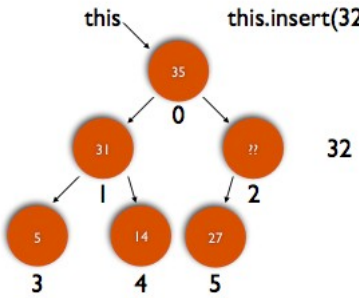
0	35
1	31
2	27
3	5
4	14

Inserting into a Heap in O(log n) time



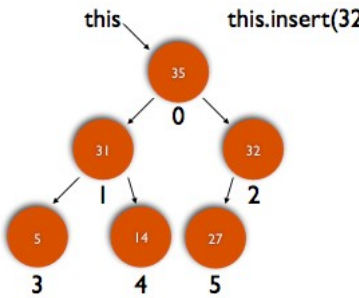
0	35
1	31
2	27
3	5
4	14
5	??

Inserting into a Heap in O(log n) time



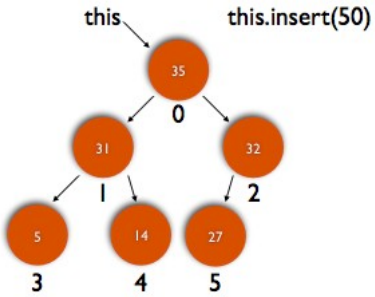
0	35
1	31
2	??
3	5
4	14
5	27

Inserting into a Heap in O(log n) time



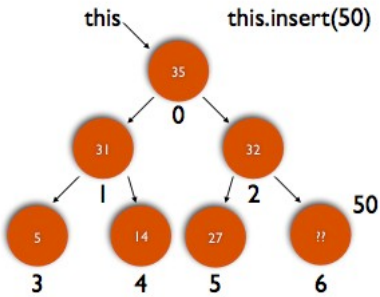
0	35
1	31
2	32
3	5
4	14
5	27

Inserting into a Heap in O(log n) time



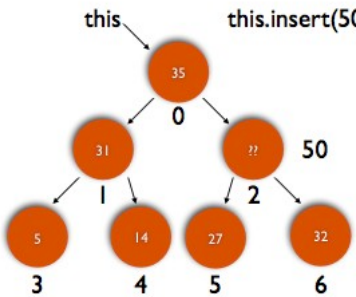
0	35
1	31
2	32
3	5
4	14
5	27

Inserting into a Heap in O(log n) time



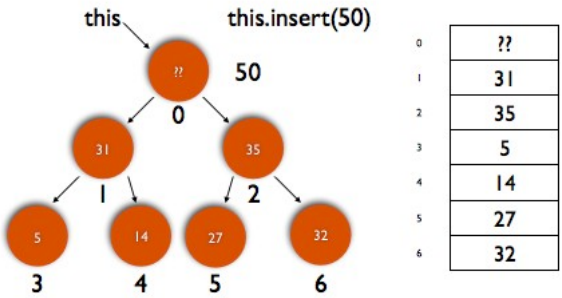
0	35
1	31
2	32
3	5
4	14
5	27
6	??

Inserting into a Heap in O(log n) time



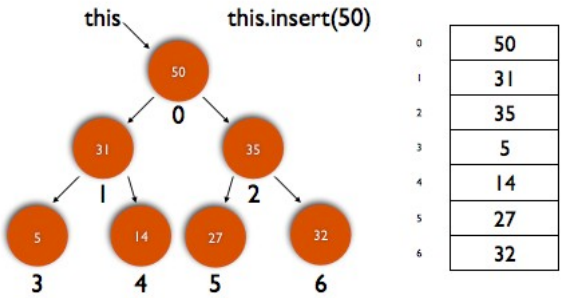
0	35
1	31
2	??
3	5
4	14
5	27
6	32

Inserting into a Heap in O(log n) time



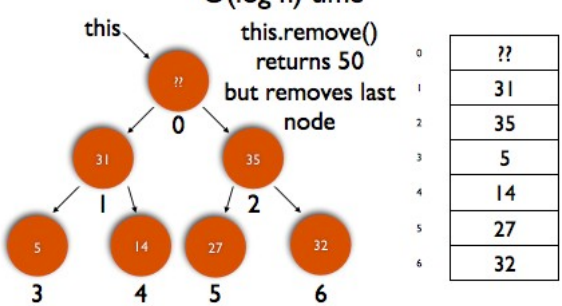
0	??
1	31
2	35
3	5
4	14
5	27
6	32

Inserting into a Heap in O(log n) time



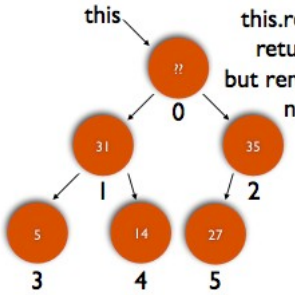
0	50
1	31
2	35
3	5
4	14
5	27
6	32

Removing 'Best' Element of a Heap in O(log n) time



0	??
1	31
2	35
3	5
4	14
5	27
6	32

Removing 'Best' Element of a Heap in O(log n) time

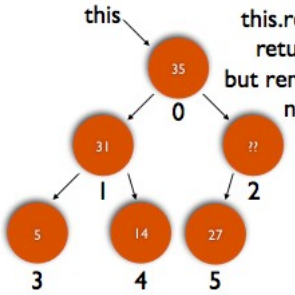


this.remove() returns 50 but removes last node

32?

0	??
1	31
2	35
3	5
4	14
5	27

Removing 'Best' Element of a Heap in O(log n) time

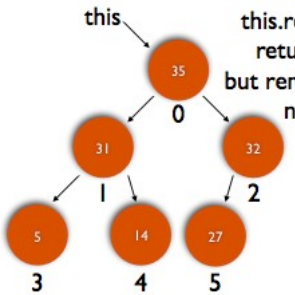


this.remove() returns 50 but removes last node

32?

0	35
1	31
2	??
3	5
4	14
5	27

Removing 'Best' Element of a Heap in O(log n) time



this.remove() returns 50 but removes last node

0	35
1	31
2	32
3	5
4	14
5	27

```
public interface Queue {
    public boolean isEmpty();
    public boolean isFull();
    public Object next();
    public boolean enter(Object o);
    public Object leave();
    public int length();
}
```

```
public class PriorityQueue implements Queue {
    private Comparable[] values;
    private int count;
    private boolean descending;

    private static int left (int i) {
        return 2 * i + 1;
    }

    private static int right (int i) {
        return 2 * i + 2;
    }

    private static int parent (int i) {
        return (i - 1) / 2;
    }
.
.
.
}
```

```
public class PriorityQueue implements Queue {
.
.
.
    public PriorityQueue (int maxsize) {
        this(maxsize, true);
    }

    public PriorityQueue (int maxsize, boolean descending) {
        this.values = new Comparable[maxsize];
        this.count = 0;
        this.descending = descending;
    }
.
.
.
}
```

```
public class PriorityQueue implements Queue {
.
.
.
    public boolean isEmpty () {
        return this.count == 0;
    }

    public boolean isFull () {
        return this.count == this.values.length;
    }

    public int length () {
        return this.count;
    }
.
.
.
}
```

```
public class PriorityQueue implements Queue {
.
.
.
    public Object next () {
        return this.isEmpty() ? null : this.values[0];
    }
.
.
.
}
```

```
public class PriorityQueue implements Queue {
.
.
.
    public boolean enter (Object o) {
       if (this.isFull()) return false;
       Comparable d = (Comparable) o;
       int p;

    // start p at the botton -- that's the next available slot
    // continue while p is not the root AND p's parent belongs below d in the heap
    // each time, move p up after swapping value at p with value at parent

       for (p = this.count;
            p > 0 && ((descending ? 1 : -1) * d.compareTo(values[parent(p)])) > 0;
            p = parent(p))
         this.values[p] = this.values[parent(p)];

    // after loop, p is the position to put the object, and count is incremented

        this.values[p] = d;
        this.count++;

        return true;
    }
.
.
.
}
```

```
public class PriorityQueue implements Queue {
.
.
    public Object leave () {
        int multiplier = this.descending ? 1 : -1;
        if (this.isEmpty()) return null;
        Comparable result = this.values[0];  // to be returned later
        Comparable d = values[--count];      // d is value to be reinserted
        int larger; // index of the larger of the two children
        int p = 0;  // starts at top, moves down to the correct position for d

        // continue as long as p is the index of an internal (non-leaf) node
        while (p < this.count / 2) {
            larger = right(p) < this.count &&
    (multiplier * this.values[right(p)].compareTo(this.values[left(p)])) > 0 ?
                right(p) : left(p);
            // if d belongs at p -- that is, both children belong below --
            // put it there and return
            if ((multiplier * d.compareTo(this.values[larger])) > 0) {
                this.values[p] = d;
                return result;
            } else {
            // otherwise, move the larger (smaller) value up, and move p down
                this.values[p] = this.values[larger];
                p = larger;
            }
        }
        // reached a leaf, put the object there
        this.values[p] = d;
        return result;
    }
.
.
}
```