Priority Queues

Part 2 — Doing it better

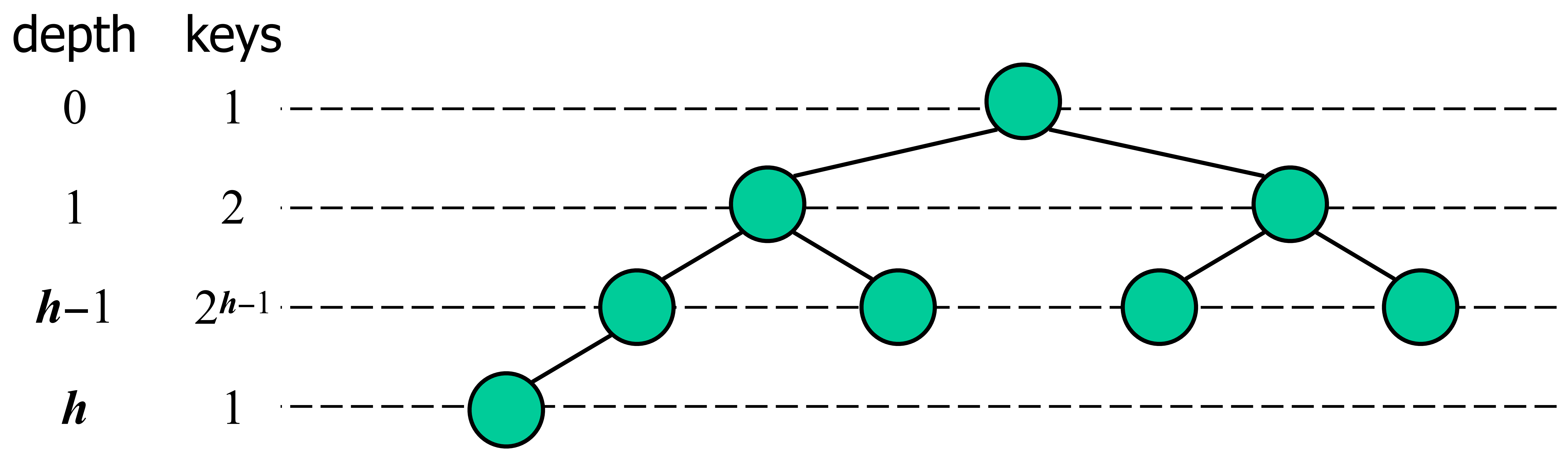
cs206
April 9

Binary Heap

- A heap is a binary tree storing keys at its nodes and satisfying:
  - heap-order: for every internal node v other than root, $key(v) \geq key(parent(v))$
  - complete binary tree: let h be the height of the heap
    - at depth h, the leaf nodes are in the leftmost positions
    - last node of a heap is the rightmost node of max depth

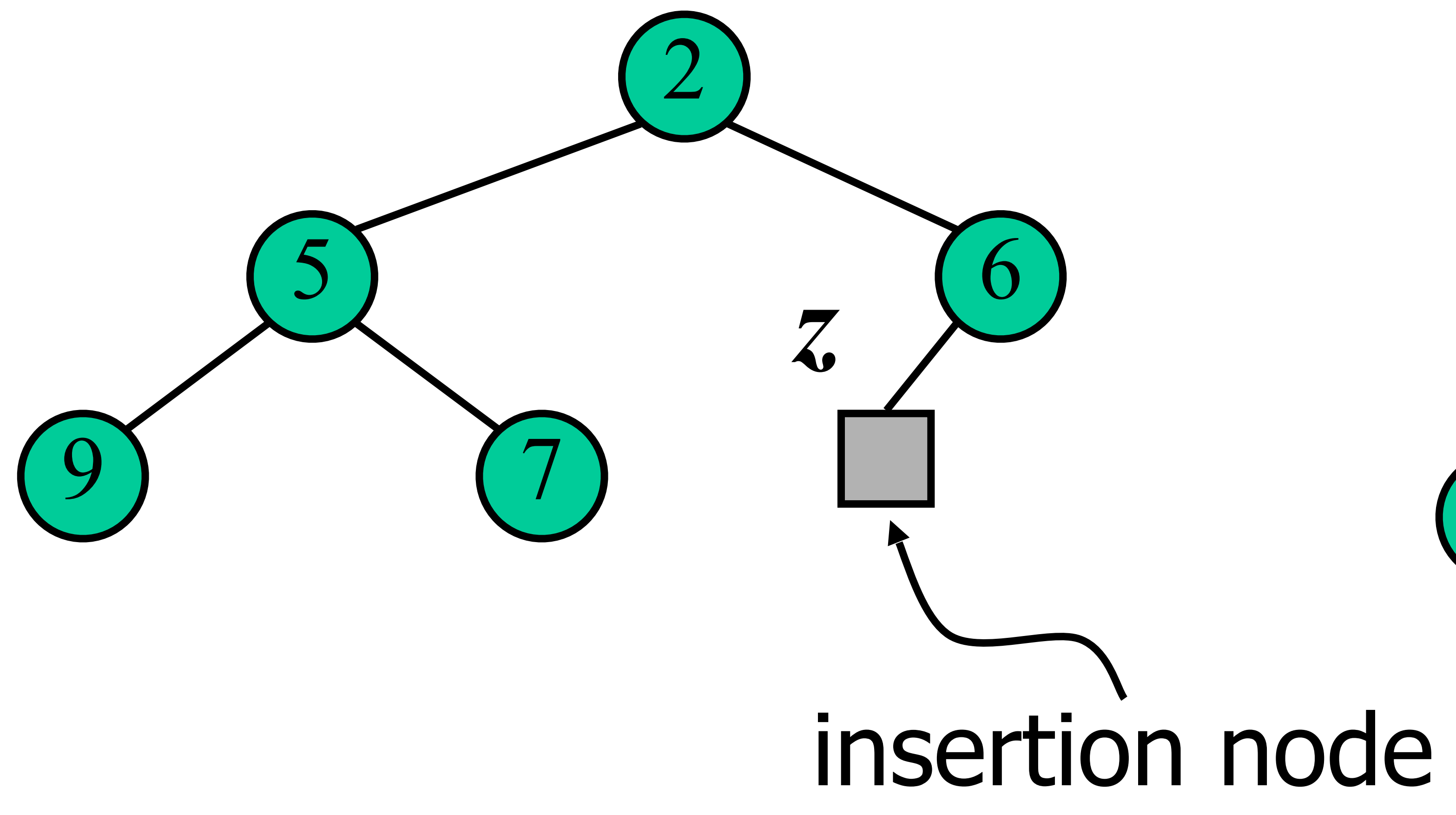
Height of a Heap

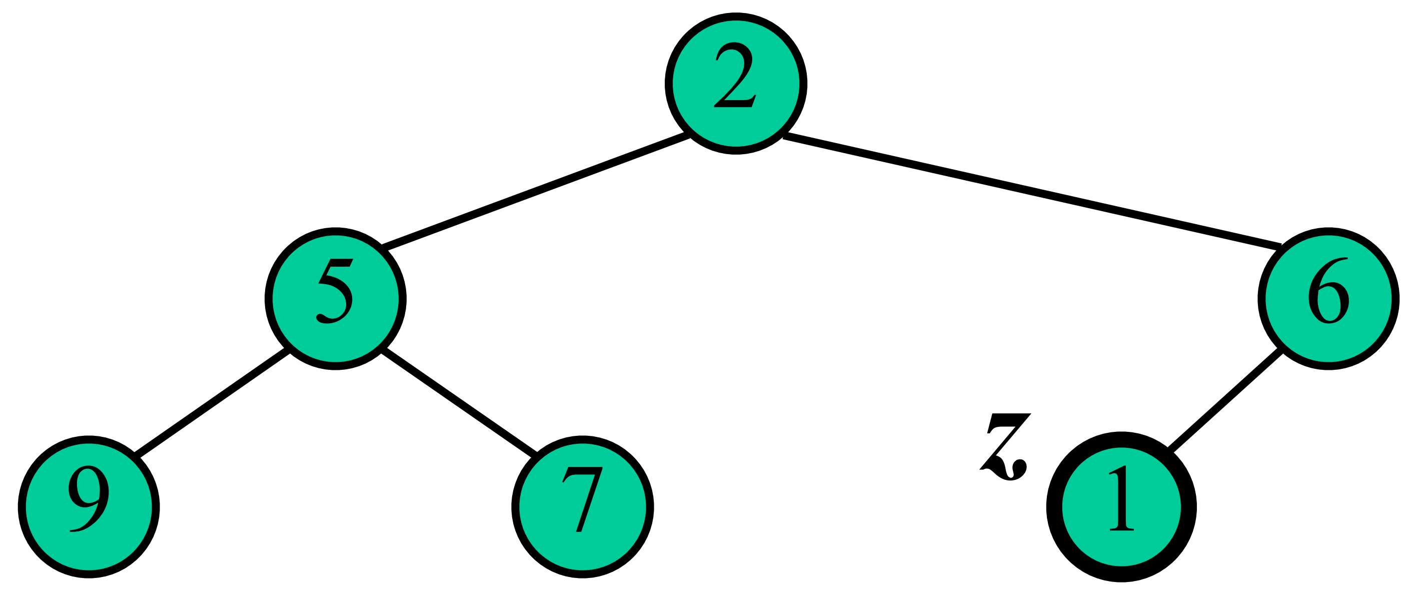
- A heap storing n keys has a height of O(logn)



Insertion into a Heap

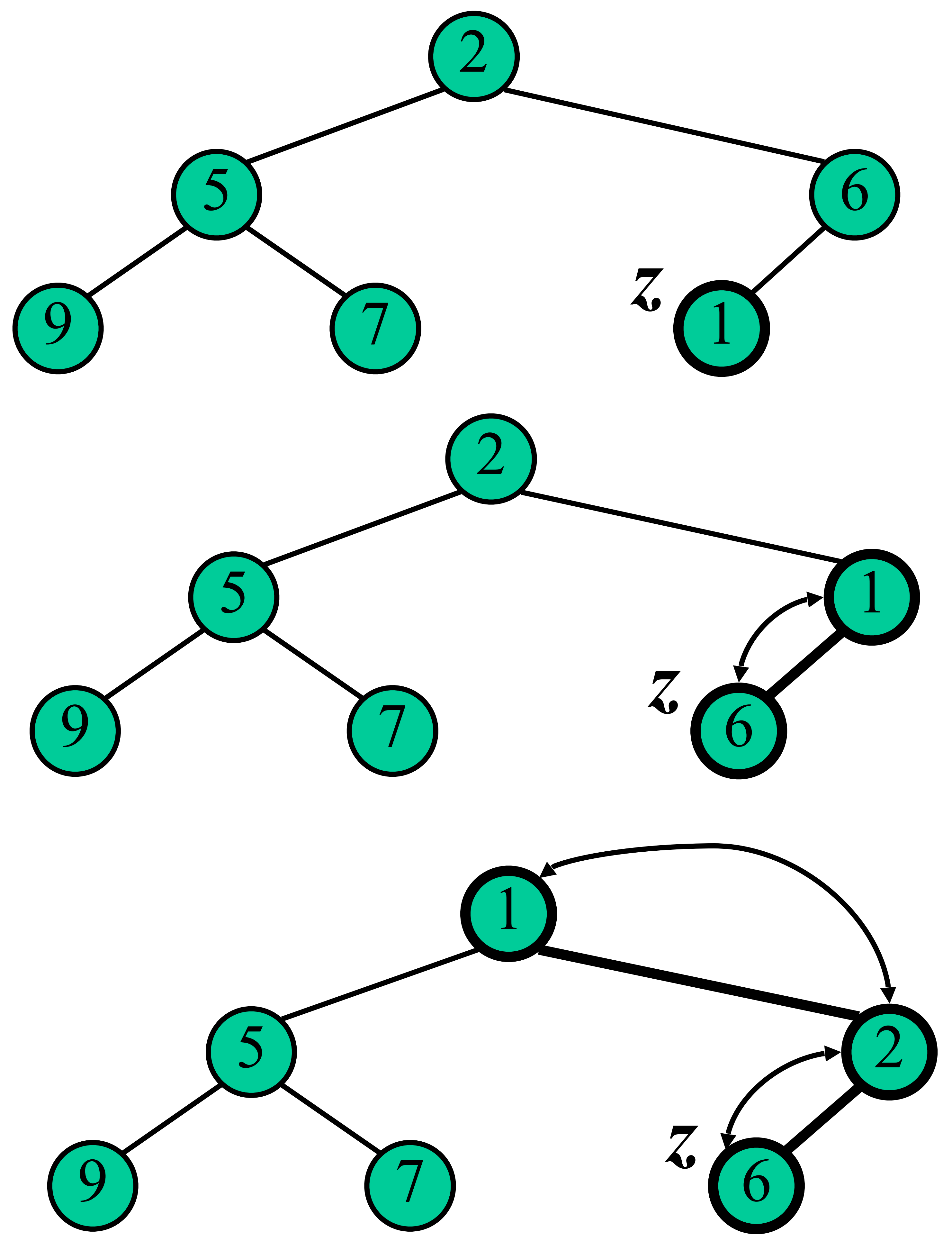
- Insert as new last node
- Need to restore heap order





Upheap

- Restore heap order
  - swap upwards
  - stop when finding a smaller parent
  - or reach root
- $O(logn)$



Priority Queue using Heaps

startup

```
public class PriorityQHeap<K extends Comparable<K>, V> implements PriorityQInterface<K,V>
{

/** The default size of the heap.  This corresponds to a max depth or 10. */
    private static final int CAPACITY = 1032;
    /** The array that holds the heap. */
    private Entry<K,V>[] backArray;
    /** The number of items actually in he heap. */
    private int size;
    /** The way in which the heap is ordered */
    final private Ordering order;

   public PriorityQHeap() {
        this(Ordering.MIN, CAPACITY);
    }

   public PriorityQHeap(Ordering order, int capacity) {
        this.order=order;
        backArray = new Entry[capacity];
    }
```

Inner class to hold Key-Value pair

moved order based comparison to here

```
protected class Entry<L extends Comparable<L>,W> {
        /** Hold the key */
        final L theK;
        /** Hold the  value*/
        final W theV;
        public Entry(L kk, W vv) {
            theK = kk;
            theV = vv;
        }

       public int doCompare(Entry<L,W> e2) {
            switch (order) {
                case MIN:
                case ASCENDING:
                    return this.theK.compareTo(e2.theK);
                case MAX:
                case DESCENDING:
                default:
                return e2.theK.compareTo(this.theK);
            }
        }
    }
```

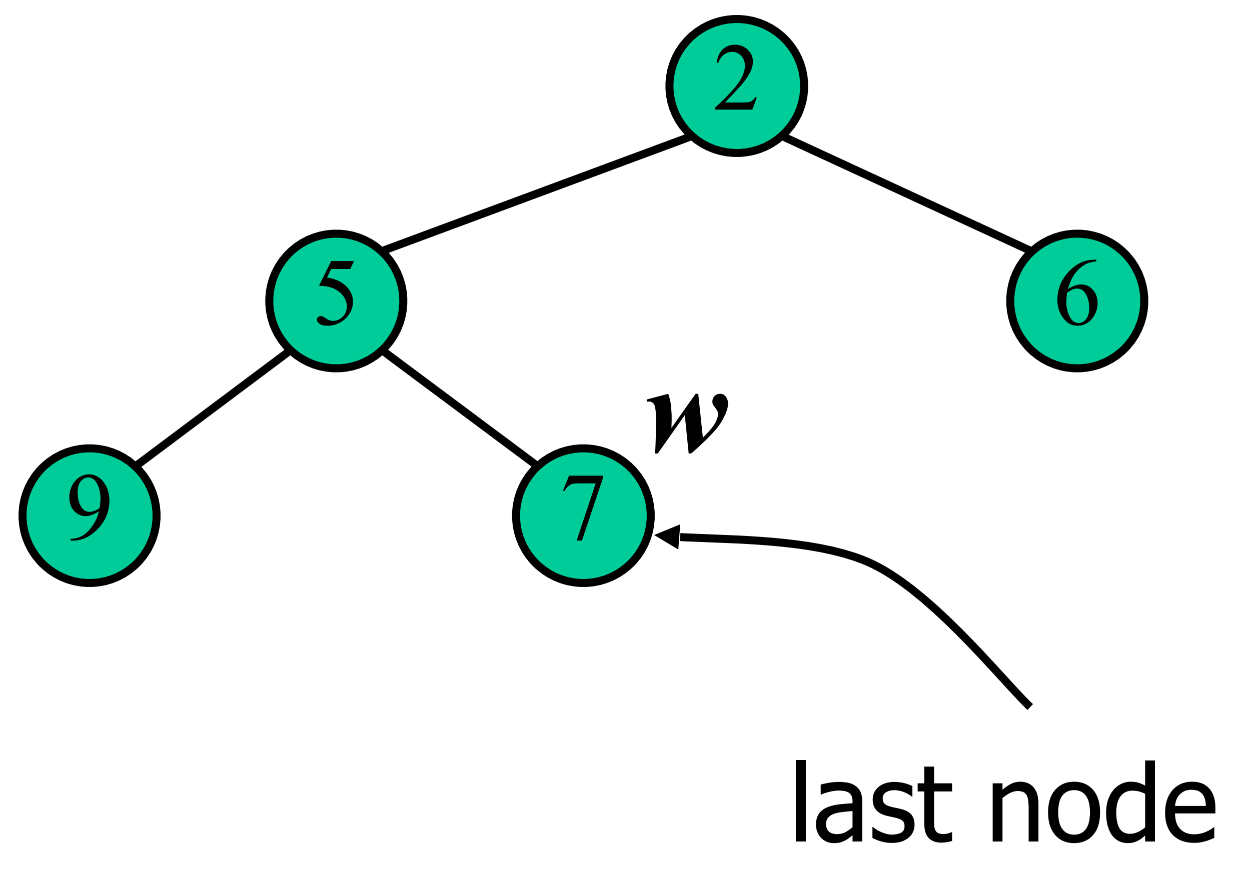
Heap Insertion

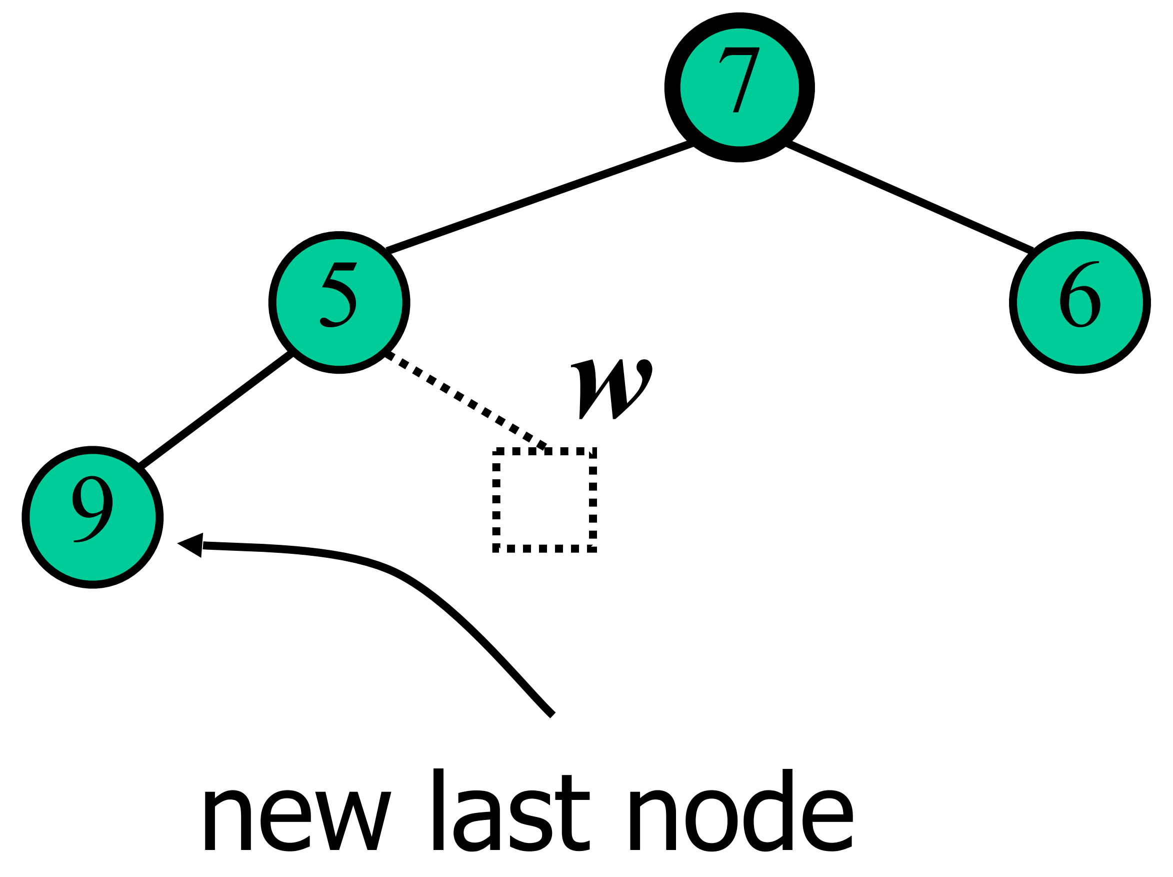
Priority Queue offer method

```
public boolean offer(K key, V value) {
    if (size>=(backArray.length-1))
        return false; // no space in the array
    // put in at end
    int loc = size++;
    backArray[loc] = new Entry<K,V>(key, value);
    // up heap
    int upp = (loc-1)/2;
    while (loc!=0) {
        if (0 > backArray[loc].doCompare(backArray[upp])) {
            // swap and climb
            Entry<K,V> tmp = backArray[upp];
            backArray[upp] = backArray[loc];
            backArray[loc] = tmp;
            loc = upp;
            upp = (loc-1)/2;
            }
        else {
            break;
        }
    }
    return true;
    }
```

Poll

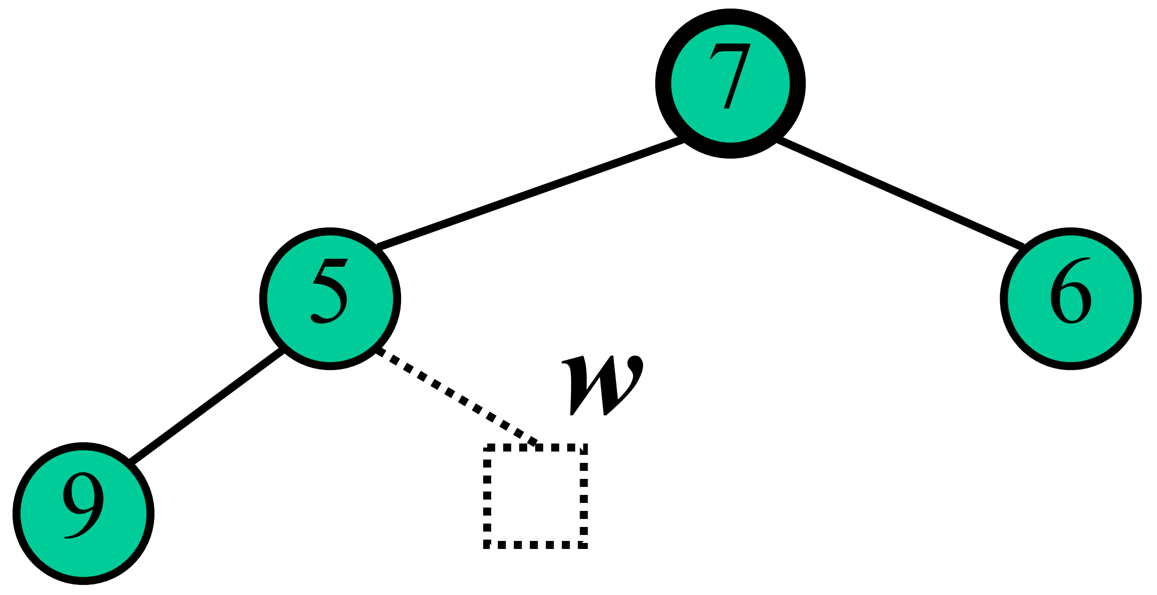
- Removing the root of the heap
  - Replace root with last node
  - Remove last node w
  - Restore heap order

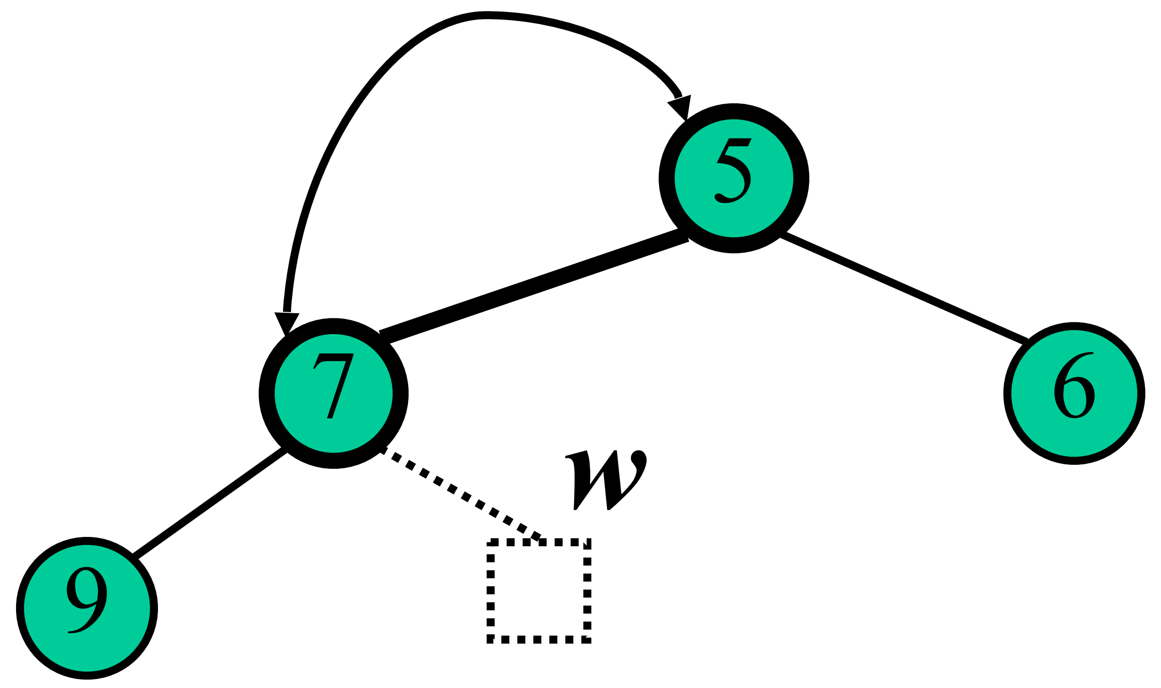




Downheap

- Restore heap order
  - swap downwards
  - swap with smaller child
  - stop when finding larger children
  - or reach a leaf
- $O(\log n)$





Peek and Poll

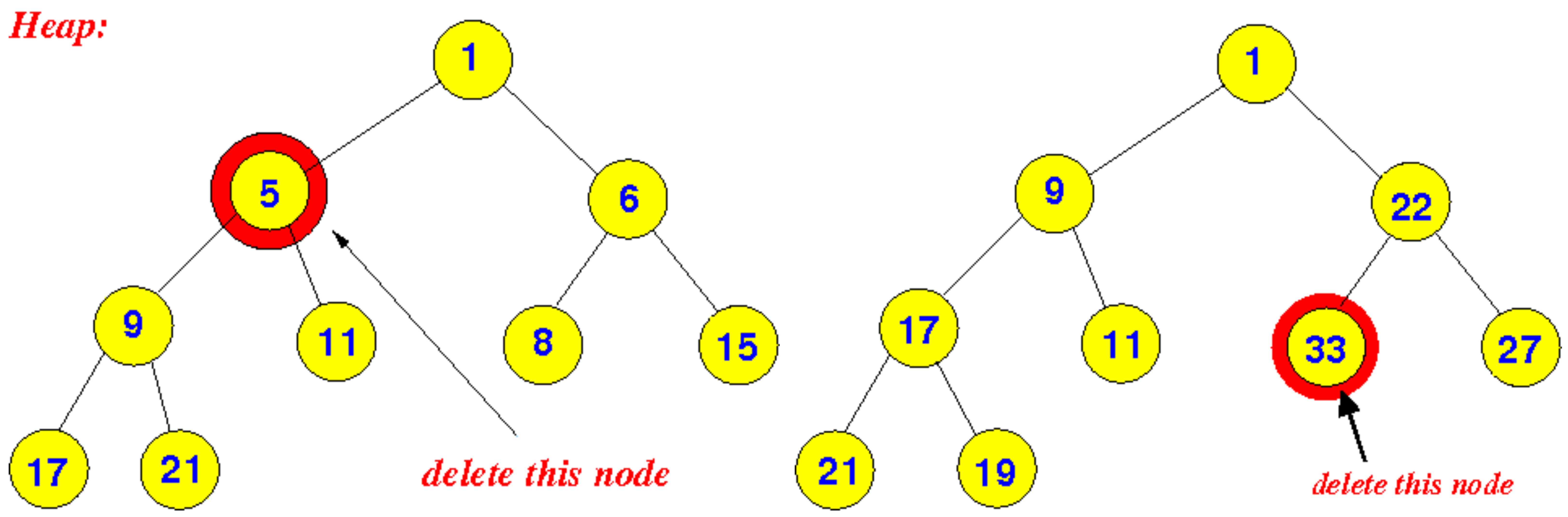
```
@Override
    public V poll() {
        if (isEmpty())
            return null;
        Entry<K,V> tmp = backArray[0];
        removeTop();
        return tmp.theV;
    }

    @Override
    public V peek() {
        if (isEmpty())
            return null;
        return backArray[0].theV;
    }
```

Remove head item from Heap

```
private void removeTop()
{
    backArray[0] = backArray[size-1];
    backArray[size-1]=null;
    size--;
    int upp=0;
    while (true)
    {
        int dwn;
        int dwn1 = upp*2+1;
        if (dwn1>size) break;
        int dwn2 = upp*2+2;
        if (dwn2>size) {  dwn=dwn1;
        } else {
            int cmp = backArray[dwn1].doCompare(backArray[dwn2]);
            if (cmp<=0)  dwn=dwn1;
            else dwn=dwn2;
        }
        if (0 > backArray[dwn].doCompare(backArray[upp]))
        {
            Entry<K,V> tmp = backArray[dwn];
            backArray[dwn] = backArray[upp];
            backArray[upp] = tmp;
            upp=dwn;                  }
        else {  break;                 } } }
```

General Removal

- swap with last node
- delete last node
- may need to upheap or downheap



Mini-Homework

Show the heap after each of the following operations. After the last operation, show the contents of the array in whch the heap is stored.

You may assume that this is a min heap (smallest first) which can store 1,000 items

offer(100, “A”)
offer(200, ”B”)
offer(1000000, “Z”)
offer(300, “C”)
offer(250, “I”)
offer(400, “D”)
offer(50, “E”)
offer(350, “F”)
poll()
offer(60, “G”)
offer(220, “H”)
poll()
poll()
poll()
offer(70, “N)