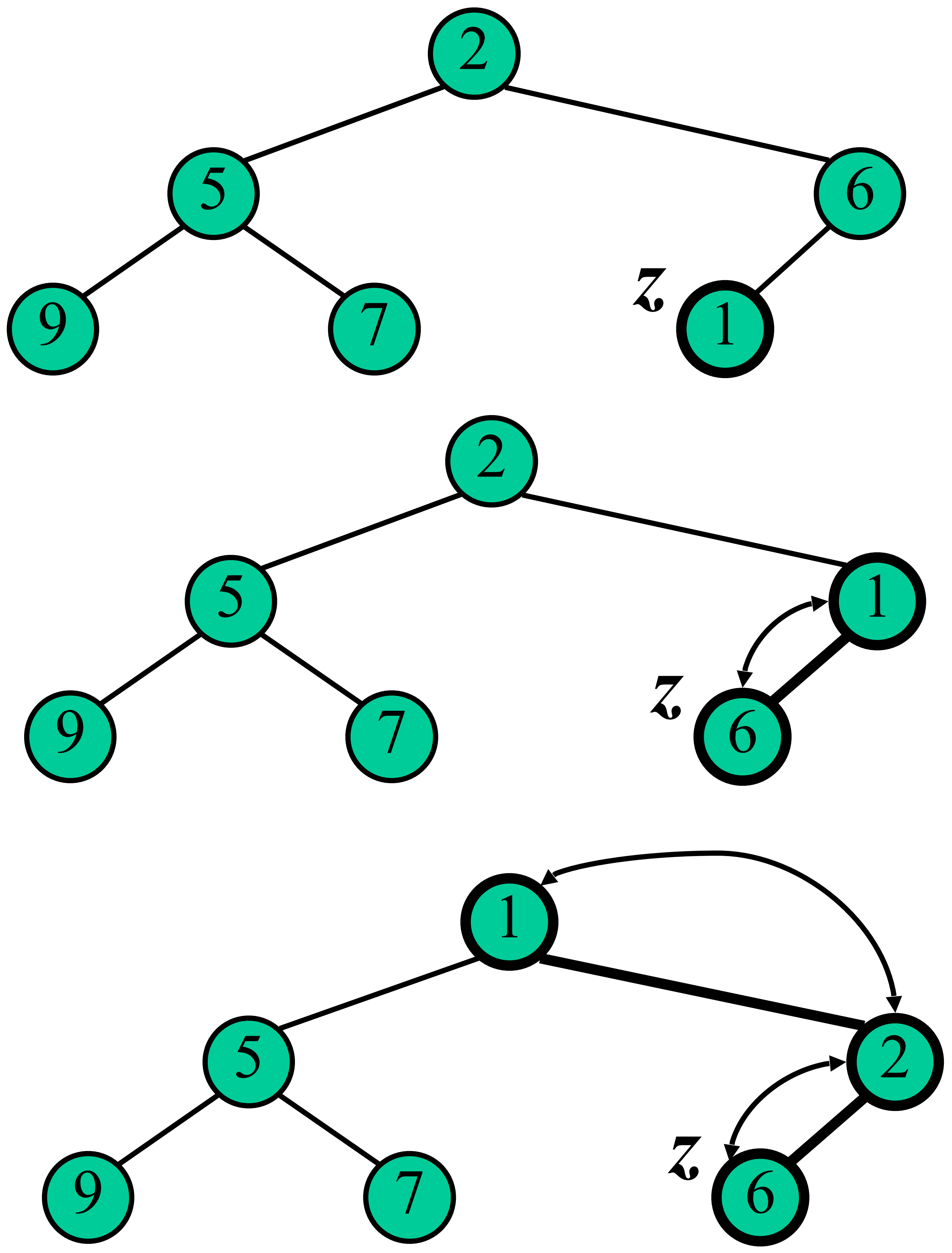
Priority Queues
Sorting

cs206

April 20

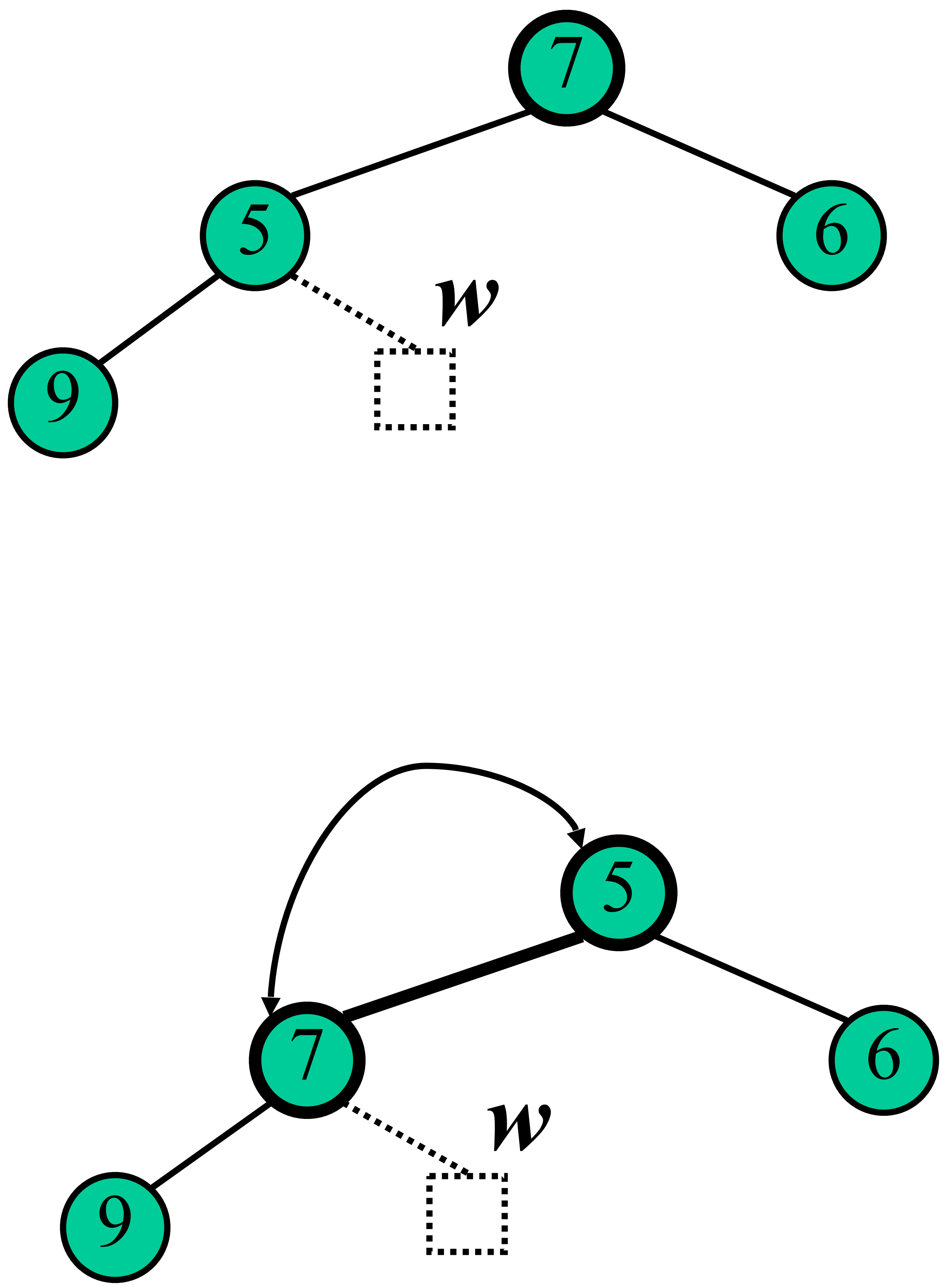
Upheap

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



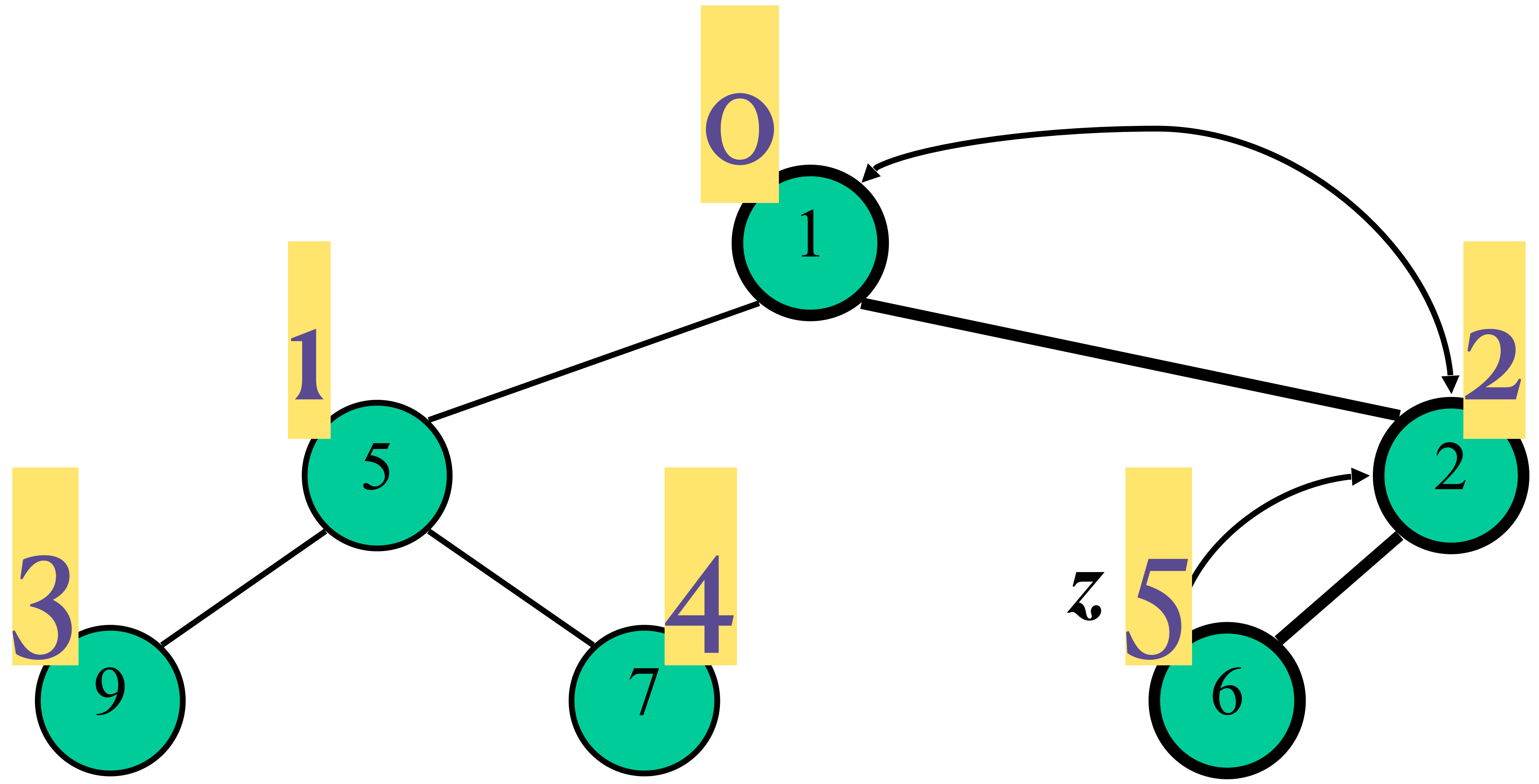
Downheap

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



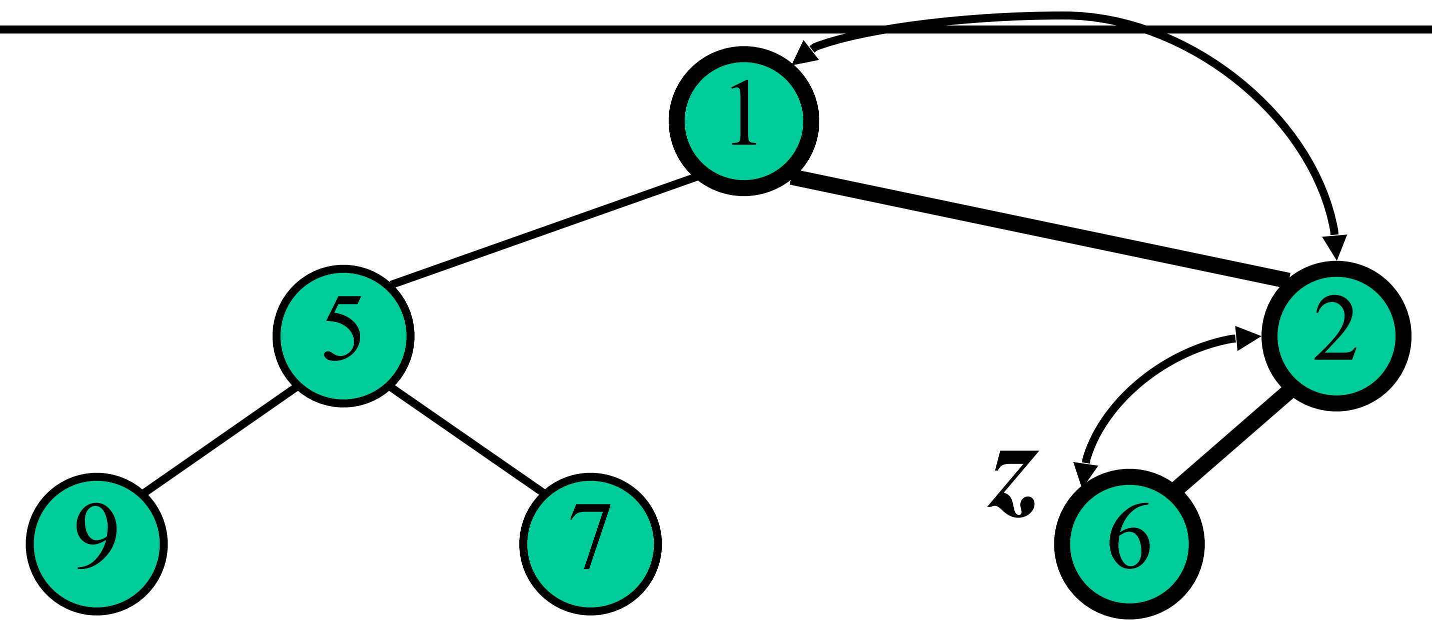
Numbering a Heap

- Can easily associate a number with each position in heap
- Root = 0
- Row 1
  - left = 1
  - right = 2
- Row 2
  - leftmost = 3
  - rightmost = 6
- Row 3
  - leftmost = $2^3 - 1 = 7$
  - rightmost = $2^4 - 2 = 14$
- Row N
  - leftmost = $2^N - 1$
  - rightmost = $2^{(N+1)} - 2$



Heaps are built on Arrays

Numbering directly leads to storage position in array



Add to Heap here

0	1	2	3	4	5	6	7
1	5	2	9	7	6		

Locations of Parents and children are in strict mathematical relationship

- Parent from child
  - suppose child is at location childLoc in array
    - parentLoc = (childLoc-1)/2
- Child from Parent
  - suppose parent is at parentLoc in array
    - leftChild = parentLoc*2+1
    - rightChild = parentLoc*2+2

- Parent from child
    - child at loc 4 (value 7)
    - parent is at (4-1)/2 = 1 (value 5)
- Child from Parent
  - parent at loc 2 (value 6)
    - leftChild =2*2+1 = 5 (value 1)
    - rightChild = 2*2+2 = 6 (value — not used)

Priority Queue using Heaps

startup

```java
public class PriorityQHeap<K extends Comparable<K>, V> extends AbstractPriorityQueue<K, V>
{

   private static final int CAPACITY = 1032;
   private Pair<K,V>[] backArray;
   private int size;

    public PriorityQHeap() {
        this(CAPACITY);
    }

    public PriorityQHeap(int capacity) {
        size=0;
        backArray = new Pair[capacity];
    }
    @Override
    public int size()
    {
        return size;
    }

    @Override
    public boolean isEmpty()
    {
        return size==0;
    }
```

Heap Insertion

Priority Queue offer method

```
public boolean offer(K key, V value)
    {
        if (size>=(backArray.length-1))
            return false;
    // put new item in at end data items
    int loc = size++;
    backArray[loc] = new Pair<K,V>(key, value);
    // up heap
    int upp = (loc-1)/2; //the location of the parent
    while (loc!=0) {
        if (0 > backArray[loc].compareTo(backArray[upp])) {
            // swap and climb
            Pair<K,V> tmp = backArray[upp];
            backArray[upp] = backArray[loc];
            backArray[loc] = tmp;
            loc = upp;
            upp = (loc-1)/2;
            }
        else
        {
            break;
        }
    }
    return true;
    }
```

Peek and Poll

```java
@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

Move end item to the top

There are no children so stop loop

There is one child

There are 2 children, which is smaller

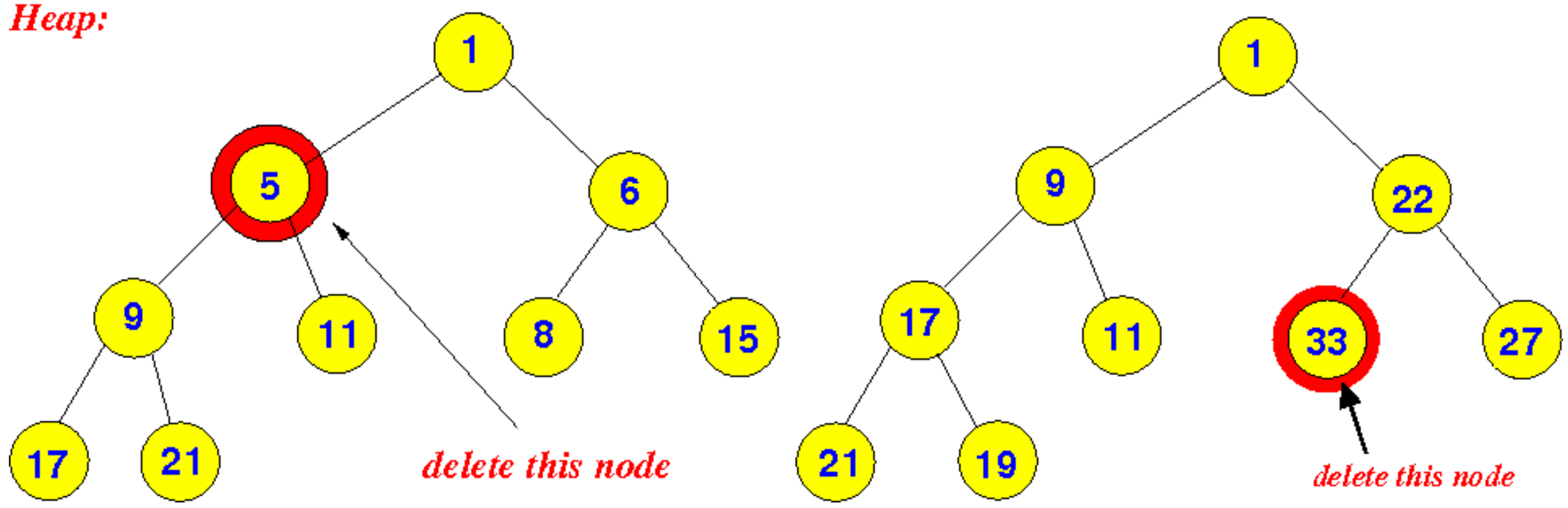
swap parent and smaller child, if appropriate

Else stop loop

```
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].compareTo(backArray[dwn2]);
            if (cmp<=0)  dwn=dwn1;
            else dwn=dwn2;
        }
        if (0 > backArray[dwn].compareTo(backArray[upp]))
        {
            Pair<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



Complexity Analysis

	Unordered	Ordered (using SAL)	Heap Based
offer	O(1)	O(N)	
peek	O(N)	O(1)	
poll	O(N)	O(1)	
Offer N then poll N			

Sorting

- public void sort(Comprable[] arra)
  - change the order of the items in arra
  - All examples will use integers but same statements apply to any Comparable object
  - ideally, do this “in place”.
    - That is do not use any extra memory

- First 3 sort techniques we have already discussed

Sorting

Offer N followed by Poll N is sorting!!!!

- PQ on unordered == Selection Sort
- PQ on ordered == Insertion Sort
- PQ on Heap == Heap Sort

Selection Sort

- Selection-sort:
  - in place algorithm given an array with N items:
    - step 1: find the min from 0..(N-1) in array and swap with item in position 0
    - step 2: find min from 1..(N-1) in array and swap with item in position 1.
    - etc
- priority queue implemented with an unsorted array / arrayList / …
- Time:
  - $O(n^2)$
    - In terms of priority Q, can split this into two phases
      - insertion == O(N)
      - polling == $O(N^2)$

Example

Phase 1 — Inserting

(a)	7	(7)	
(b)	4	(7,4)	
….			
(g)	()		(7,4,8,2,5,3,9)

Phase 2 — Polling

(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion Sort

- Insertion-sort
  - in-place algorithm
    - Step 0: start with item in position 0. Now the items in positions 0..0 are sorted
    - Step 1: look at item in position 1. Compare it to item in 0. If p1 is smaller, then swap. the items in position 0..1 are sorted with respect to each other
    - Step 2: determine where item in p2 should go in sorted list 0..N. If needed, For instance, bigger than 0 but smaller than 1. Make a space: save p1 into tmp. Shifting p1 into p2. Then put tmp into p1. Now the item in 0..2 are sorted.
    - Step N:

  - insert/swap the element into the correct sorted position
- Priority queue implemented with a sorted array/ ArrayList / …
- Time:
  - $O(n^2)$
  - In terms of PQ
    - Add:$O(n^2)$
    - Remove: $O(n)$
  - Faster than selection sort

Example

Phase 1 — Inserting

(a)	7	(7)
(b)	4	(4,7)
(c)	8	(4,7,8)
(d)	2	(2,4,7,8)
(e)	5	(2,4,5,7,8)
(f)	3	(2,3,4,5,7,8)
(g)	9	(2,3,4,5,7,8,9)

Phase 2 — polling

(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Heap Sort

- Heap-sort:
  - Insertion — no more than $\log_2(n)$ steps per insertion
  - Deletion — no more than $\log_2(n)$ steps per deletion
- priority queue is implemented with a heap

- Time:
  - Add:O(n * $\log_2(n)$) — under some assumptions O(n).
  - Remove: O(n * $\log_2(n)$)
- Note: with a **lot of work** can do this without an additional array.

Example

Phase 1 — Inserting

(a)	7	(7)
(b)	4	(4,7)
(c)	8	(4,7,8)
(d)	2	(2,4,8,7)
(e)	5	(2,4,8,7,5)
(f)	3	(2,4,3,7,5,8)
(g)	9	(2,4,3,7,5,8,9)

Phase 2 — polling

(a)	(2)	(3,4,7,5,8,9)
(b)	(2,3)	(4,5,7,9,8)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Timing

Table 1

size	selection	Insertion	Insertion	Heap
1000	16	15	11	2
2000	8	12	26	3
4000	24	23	20	5
8000	96	95	81	10
16000	370	378	315	17
32000	1585	1359	1218	36
64000	5771	5590	4605	77
128000	23087	21547	19849	161
256000				345
512000				1128
1024000				1973
2048000				3225
4096000				7577
8192000				18586

10000==1 second

Practice

- Show the array after each step of selection sort, insertion sort and heap sort for the following list
  - for insertion and selection sort, use the in-place algorithms discussed
  - For heap sort, show the array after each addition to the heap and each removal from the heap.
- Count the number of operations you did for each

9,4,1,3,12,17, 2, 8,20,5

Breakpoints

and using them in VSC

- Better than Print statements!!!!!!
- Idea: set a place (or places) when your program will pause.
- When paused:
  - look at variable values
  - look at the stack
    - continue
      - stepwise

Lab

- Open VSC on some program you wrote
- Set a breakpoint
- Run program
- when hit breakpoint
  - get screen to show variable values and the stack
  - take picture and send the picture to me