Priority Queues

cs151

Priority Queue

- A queue that maintains order of elements according to some priority
- Contrast to Queue which is FiFo
  - PriorityQueue can implement a stack or a queue
- **PriorityQueues are about the order in which things are removed, NOT the way in which they are stored.**
  - the items may or may not be sorted, or otherwise arranged.
  - Aside: This statement applies to stack and queues also, it is just convenient in those cases to arrange data to make retrieval easy

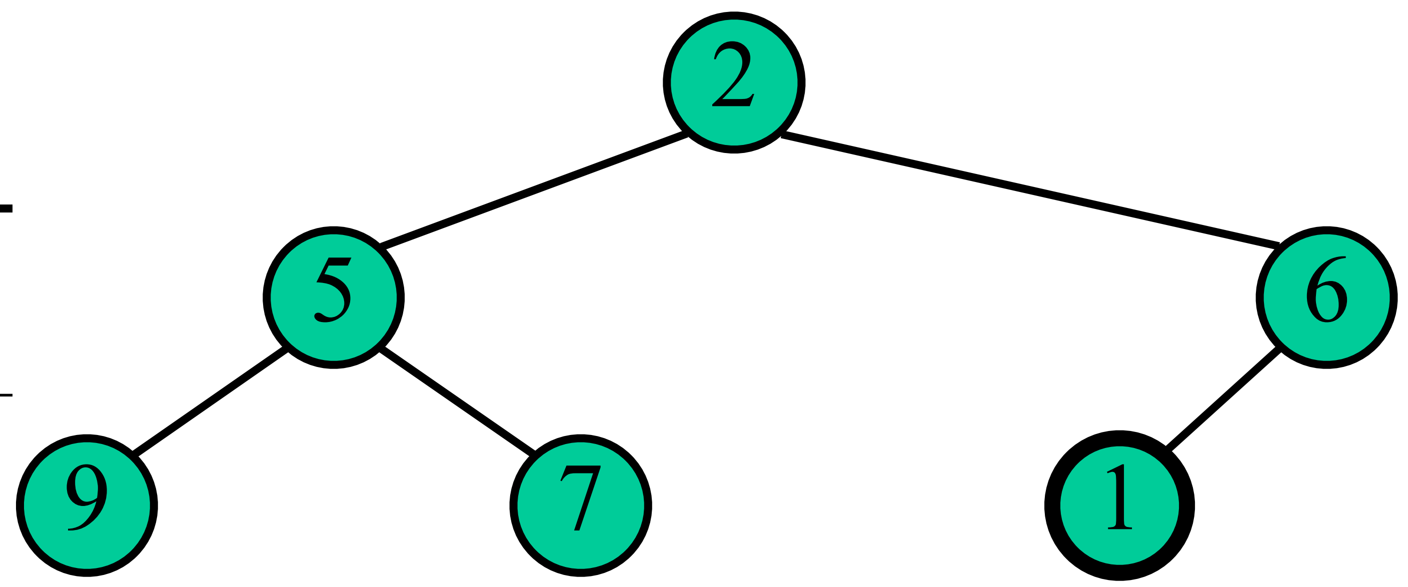
Complexity Analysis

	Unordered	Ordered
offer	O(1)	O(n)
peek	O(n)	O(1)
poll	O(n)	O(1)
Add N items, then Remove N items	Add:O(n) Remove: O(n*n) Overall:O(n*n)	Add:O(n*n) Remove: O(n) Overall: O(n*n)

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, $key(v) \geq key(parent(v))$
  - Heap is filled from top down and within a level from left to right.
    - at depth h, the leaf nodes are in the leftmost positions
    - last node of a heap is the rightmost node of max depth

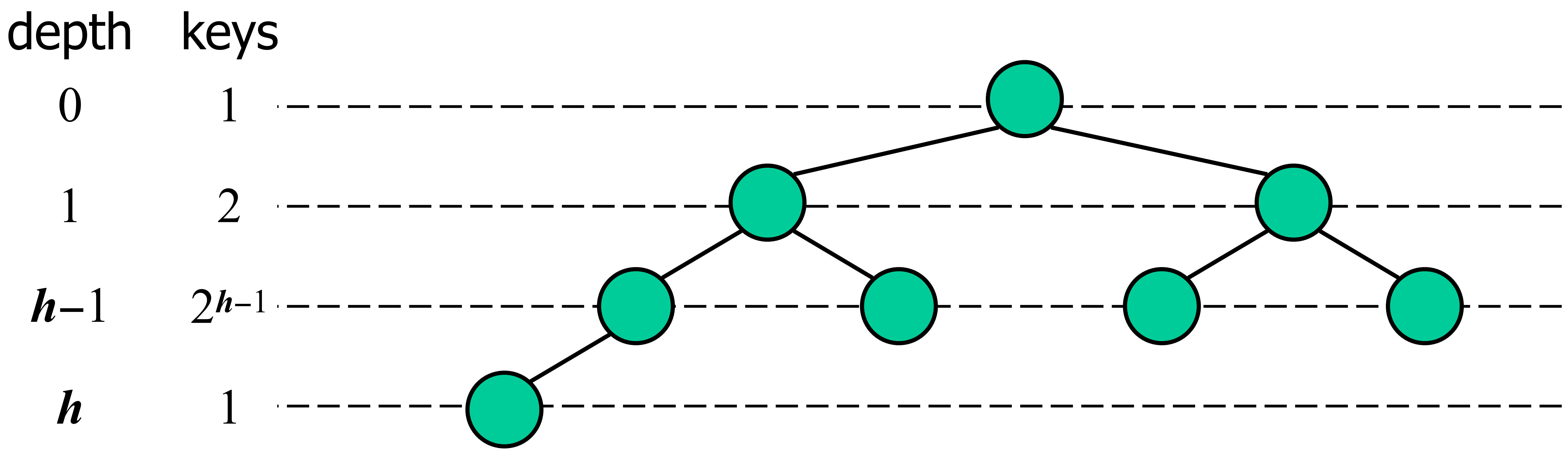
Binary Tree — terms



Term	Definition	
Node	A part of a tree.	2,5,7,9,7,1
Parent	A node that has children	2,5,6
Child	A node that has parents. Child nodes have exactly one parent	
Binary Tree	A structure of nodes such that parent nodes have at at most two children	
Root	The node in a tree that has no parent.	
Leaf	Any node that has no children	
Height	The maximum distance from a the root node to a leaf.	
Subtree	The part of a tree whose root is a given node	

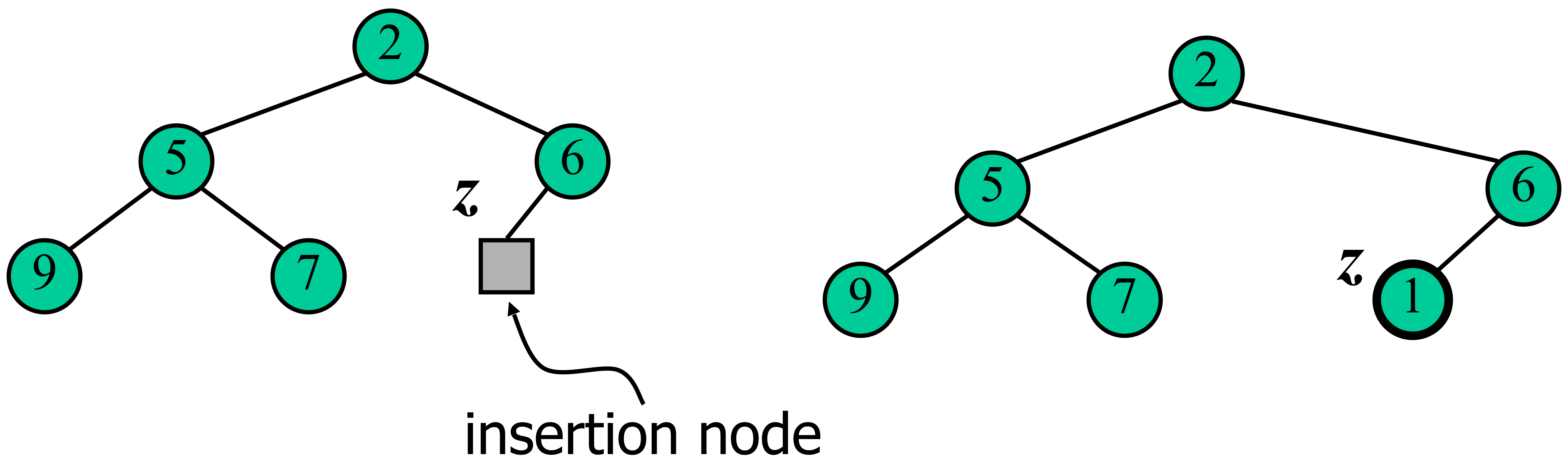
Height of a Heap

- A binary heap storing n keys has a height of $O(\log_2 n)$
- This is NOT true for general binary trees



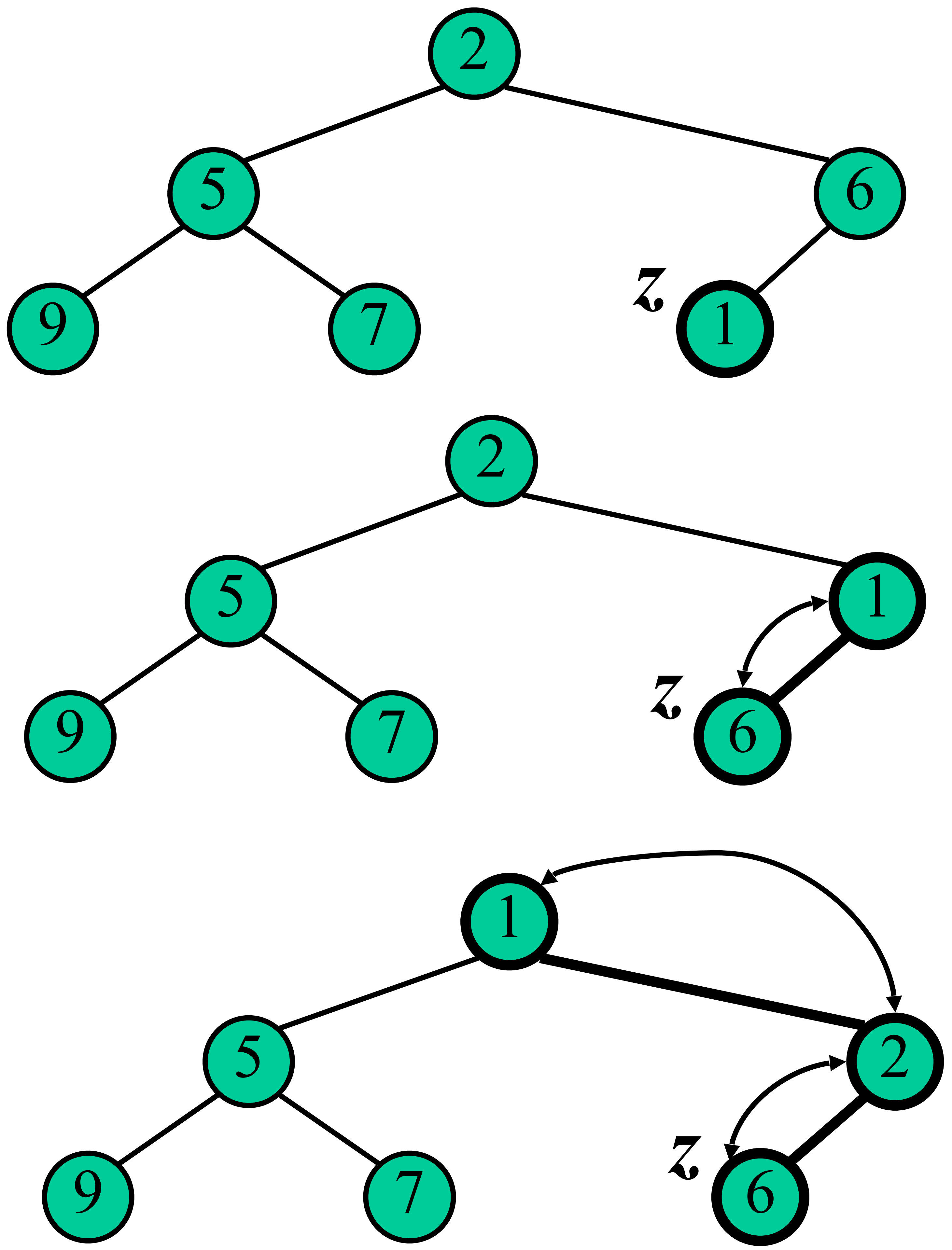
Insertion into a Heap

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



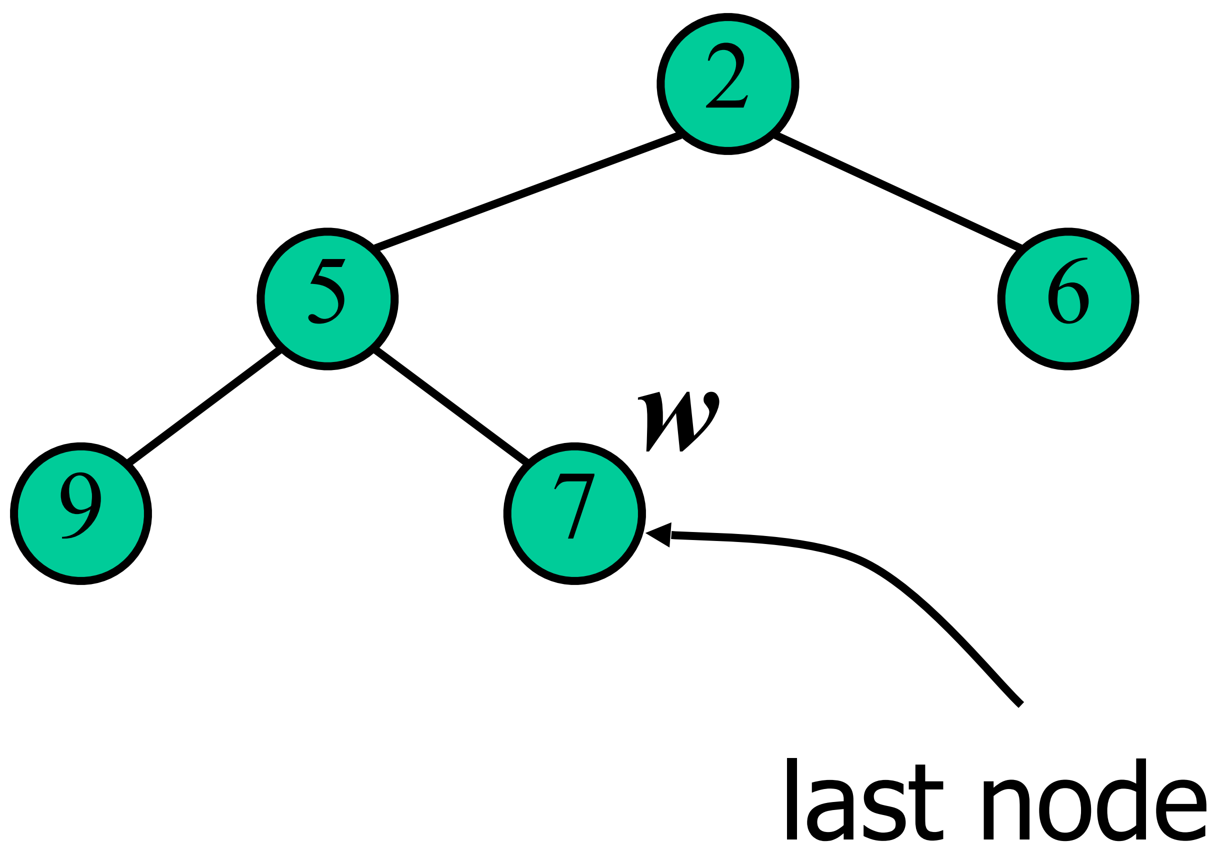
Upheap

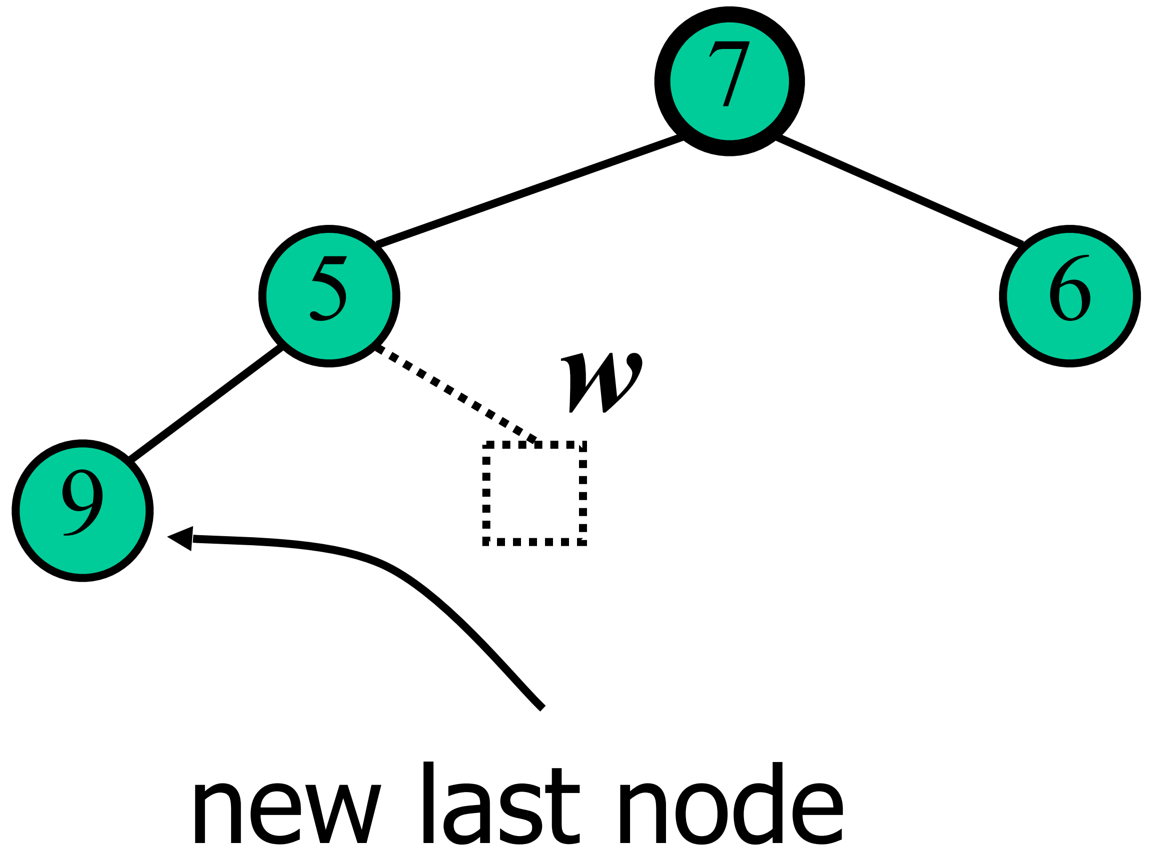
- Restore heap order
  - swap upwards
  - stop when finding a smaller parent
  - or reach root
- $O(\log n)$



Poll

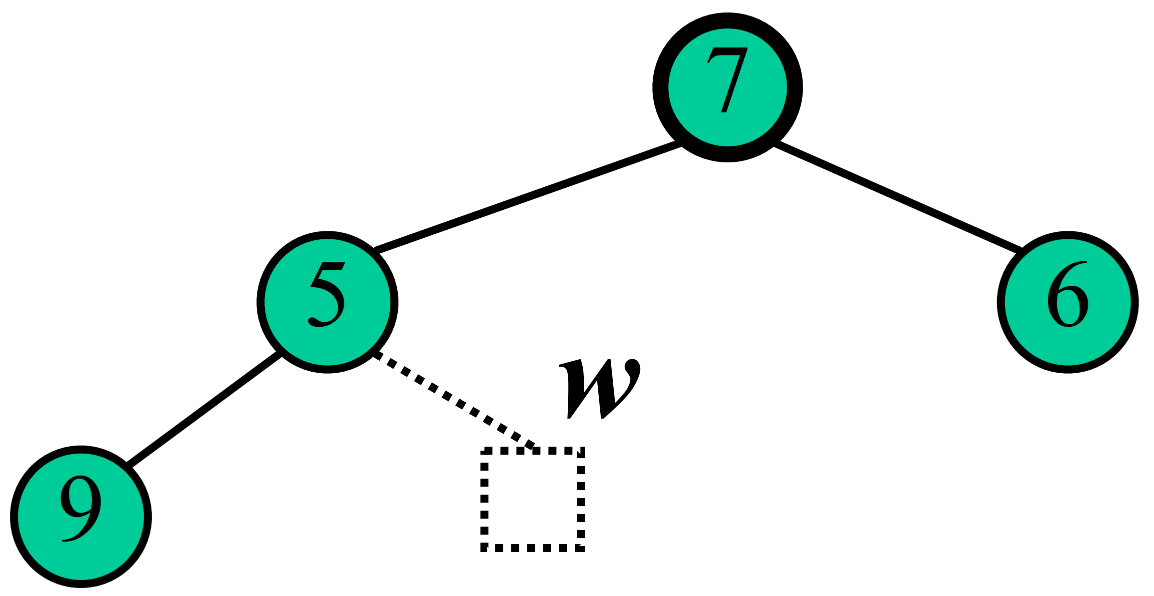
- Removing the root of the heap
  - Replace root with last node
  - Remove last node
  - Restore heap order

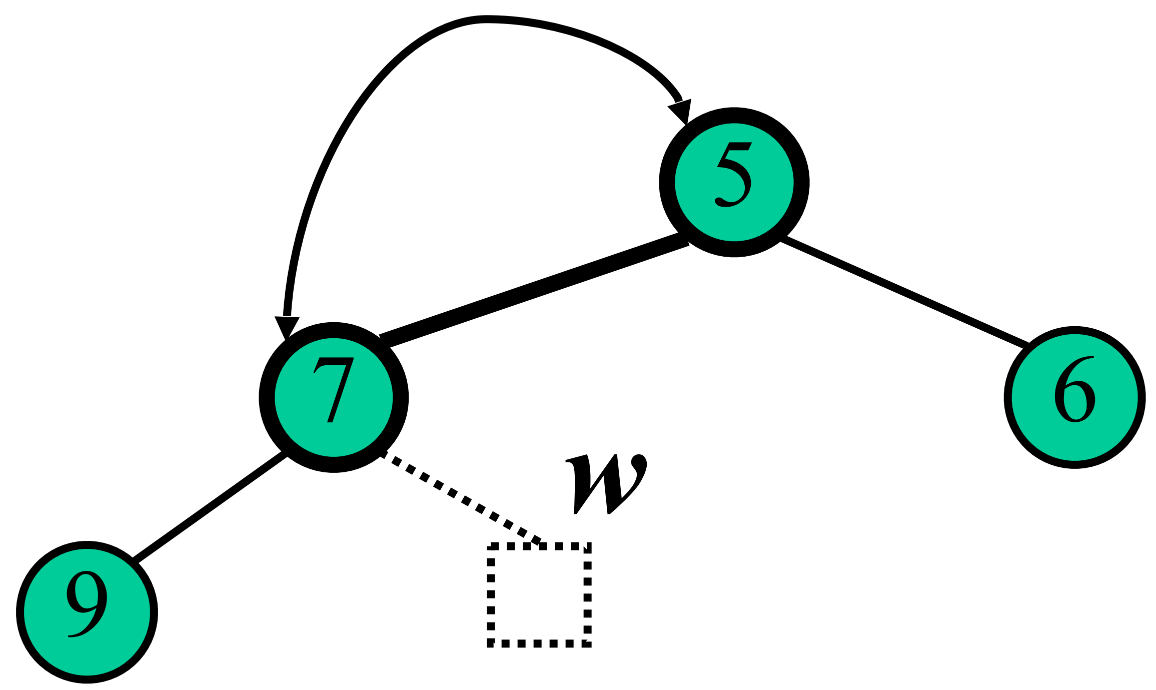




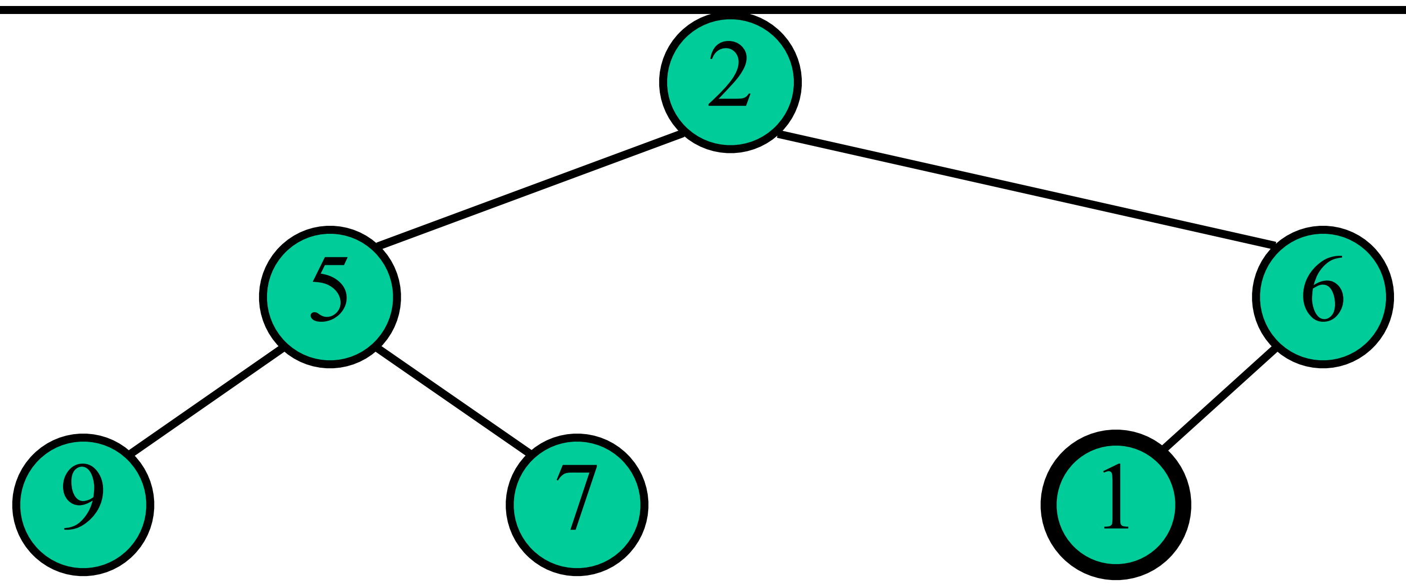
Downheap

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





Heaps are built on Arrays



Add to Heap here

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

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

```
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)
```

1. Ensure there is room — if not return false
2. Add new items to end of heap (low and left viewed graphically) first unoccupied viewed array-wise
3. Repeat
    1. If at root, STOP
    2. Compare with parent
    3. If greater, swap the GoTo 3.1
    4. stop (less -- or equal -- so do not need to keep going up)
4. return true

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 Top

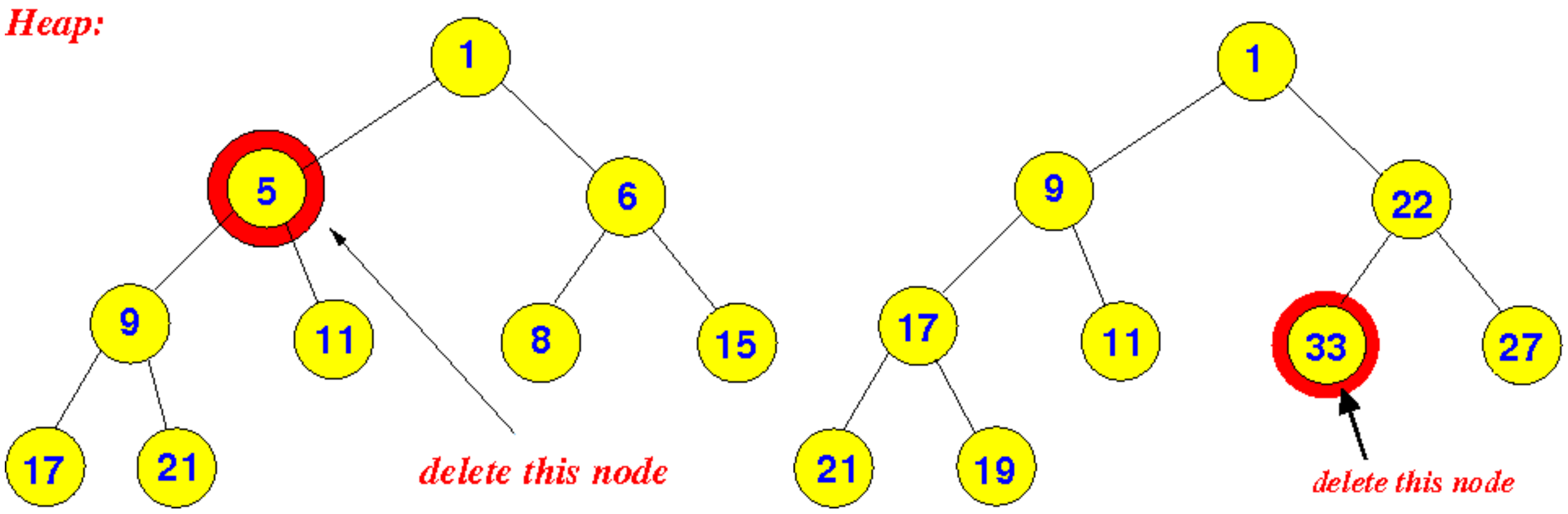
In English

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].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



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;
    }
```

Complexity Analysis

	Unordered	Ordered	Heap Based
offer	O(1)	O(n)	O(lg n)
peek	O(n)	O(1)	O(1)
poll	O(n)	O(1)	O(lg n)
Add N items, then Remove N items	Add:O(n) Remove: O(n*n) Overall:O(n*n)	Add:O(n*n) Remove: O(n) Overall: O(n*n)	Add:O(n * lg n) Remove: O(n * lg n) Overall: (n * lg n)

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 == $O(N^2)$

Selection Sort — Example

Phase 1 Inserting	Inserting		
a	7	(7)	1
b	4	[7,4]	1
…			
g		[7,4,8,2,5,3,9]	
Phase 2	Polling		
a	[2]	[7,4,8,5,3,9]	search=4, shift=3
b	[2,3]	[7,4,8,5,9]	search=5, shift=1
c	[2,3,4]	[7,8,5,9]	search=2 shift=3
d	[2,3,4,5]	[7,8,9]	search=3, shift=1
e	[2,3,4,5,7]	[8,9]	search=1, shift=2
f	[2,3,4,5,7,8]	[9]	search=1, shift=1
g	[2,3,4,5,7,8,9]	[]	search=1

Insertion Sort

- Insertion-sort
  - in-place algorithm
    - public Comparable[] sort(Comprable[] arra)
    - 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:

- Priority queue implemented with a sorted array/ ArrayList / …
- Time:
  - $O(n^2)$
  - In terms of PQ
    - Add:$O(n^2)$
    - Remove: $O(n)$
  - Generally 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 most commonly implemented with a heap

- Time:
  - Add:$O(n * \log_2(n))$ — doable in $O(n)$.
  - 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

size	selection	Insertion	Insertion (improved)	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

anything below 1000
is very noisy