CS206

Priority Queues

Performance of Trees

	Complete Tree	Worst Tree
search		
insert		
remove		

Create a dataset to make the worst possible tree

Priority Queue

- A queue that maintains order of elements according to some priority
  - Removal order, not general order
    - the rest may or may not be sorted
- Types of PQs
  - min PQ — the element with smallest key is removed first
  - max PQ — the largest is removed first
- Consider a PQ in which priority is based on insertion time
    - min PQ == ??
    - max PQ== ??

Key

- Priority queues are ordered by some key, which may be:
  - derived from the data element
    - one field
    - combination of fields
  - independent of data element
    - for example: insertion time
- best practice is to define relation between keys using `compareTo`
- Changing `compareTo` allows changing the priority queue ordering while changing nothing else

Key-Value Pair

- Typically think of PQ as containing a pair
  - (Key, Value)
    - Key defines priority
    - Value is data the objects store
- KV pairs are frequently used
- Ideally keys are unique
  - how to handle duplicate keys?
- Ideally keys have a natural ordering.
  - Using `compareTo` allows arbitrary comparisons
- Values need not be numerical or unique

Example - minPQ

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Interface

```
public interface PriorityQueueInterface<E
extends Comparable<E>> extends
BinaryTreeInterface<E> {
  E getRootElement();
  int size();
  boolean isEmpty();
  boolean contains(E element);
  void insert(E element);
  boolean remove(E element);
  E peek(); // look at min/max; do not remove
  E poll(); // removeMin/removeMax;
}
```

Note that extending BinaryTreeInterface does not require that PQ is built on a Binary Tree

How do we implement it?

- Efficiency depends on implementation

	Unsorted array	Unsorted list	Sorted array	Sorted list
peek				
poll				
insert				
remove				

- Remove may apply to any element, poll just to the "first"

Priority Queue Sort

- Sorting using a priority queue
  1. Insert with a series of `insert` operations
  2. Remove in sorted order with a series of `poll` operations
- Efficiency depends on implementation and runtime of `insert` and `poll`

Selection Sort

- Selection-sort:
  - select the min/max and swap with 0
- priority queue is implemented with an unsorted sequence
- $O(n^2)$

Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(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)	()

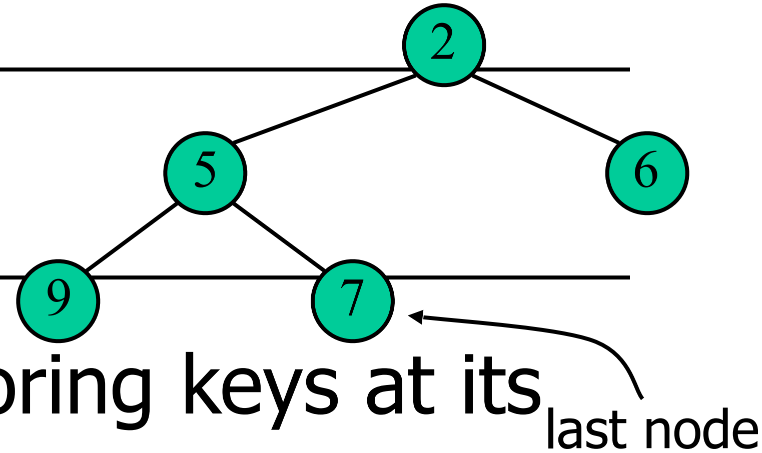
Insertion Sort

- Insertion-sort:
  - insert/swap the element into the correct sorted position
- Priority queue is implemented with a sorted sequence
- $O(n^2)$

Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

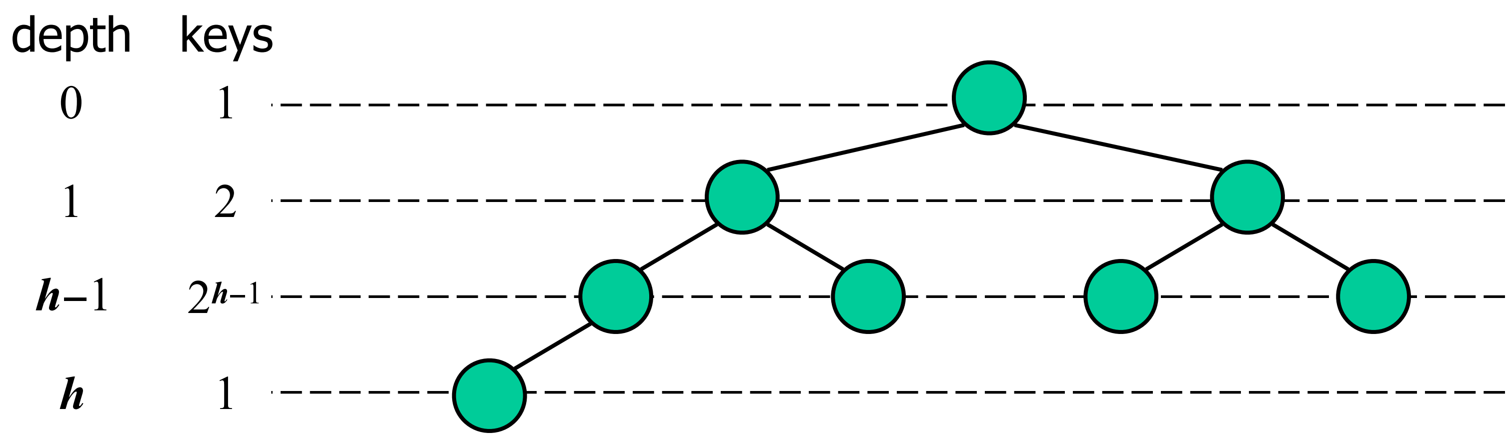
Binary Heap



- 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
    - there are 2^i nodes of depth i, $0 \leq i \leq h-1$
    - at depth h, the leaf nodes are in the leftmost positions
    - last node of a heap is the rightmost node of max depth

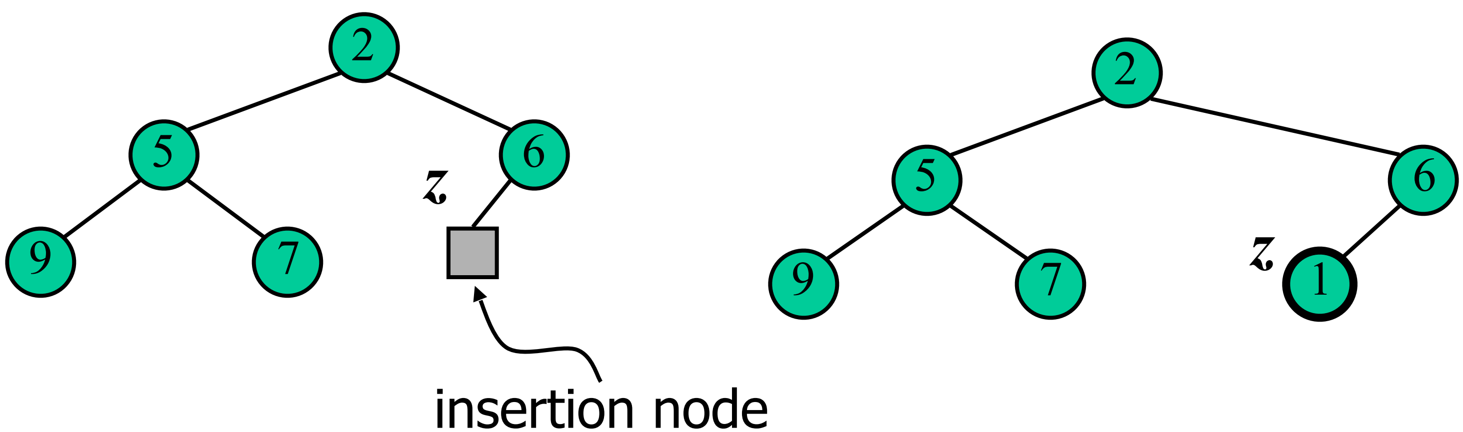
Height of a Heap

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



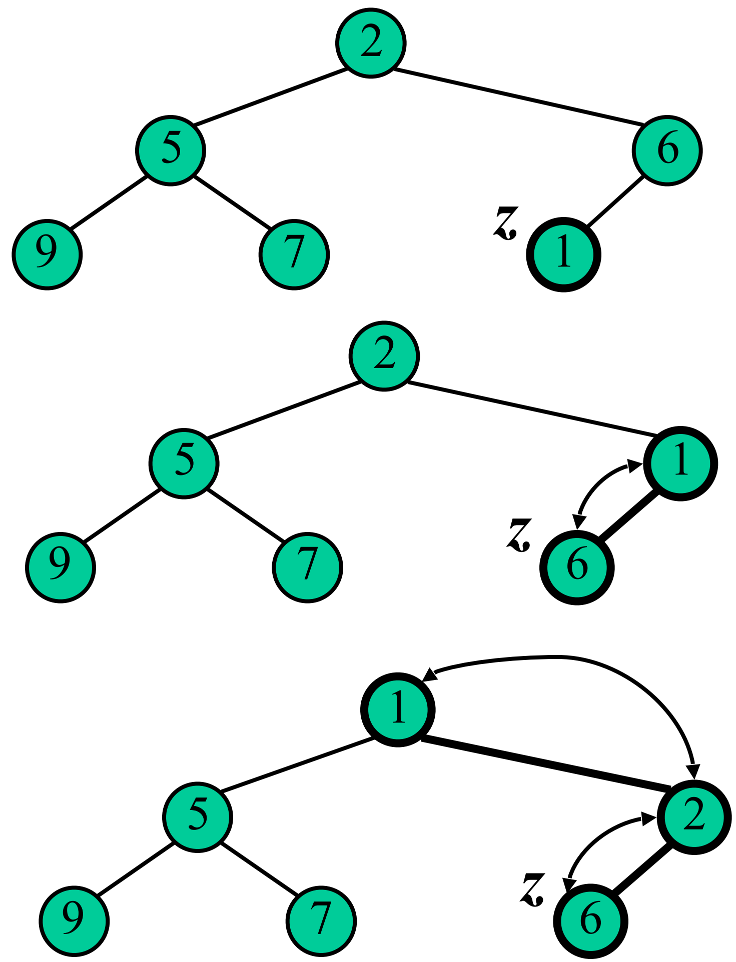
Insertion into a Heap

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



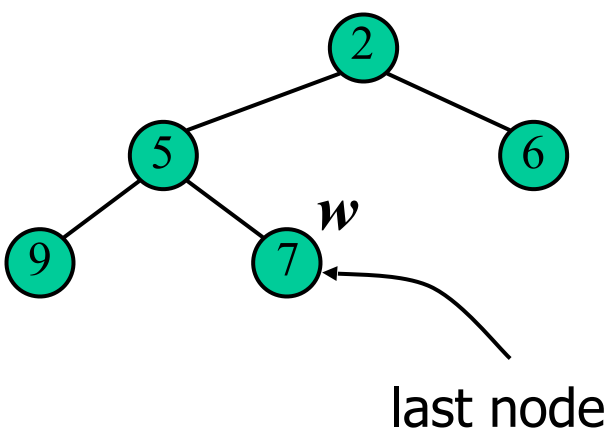
Upheap

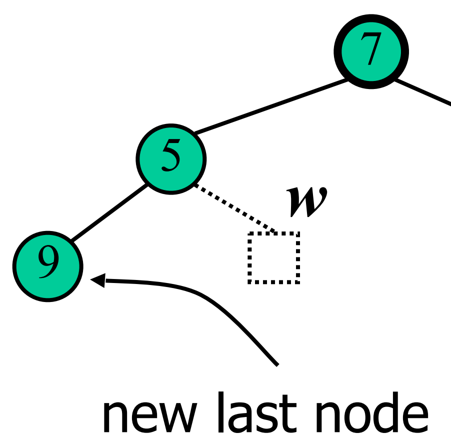
- Restore heap order
  - swap upwards
  - stop when finding a smaller parent
  - or reach root
- $O(logn)$



Poll

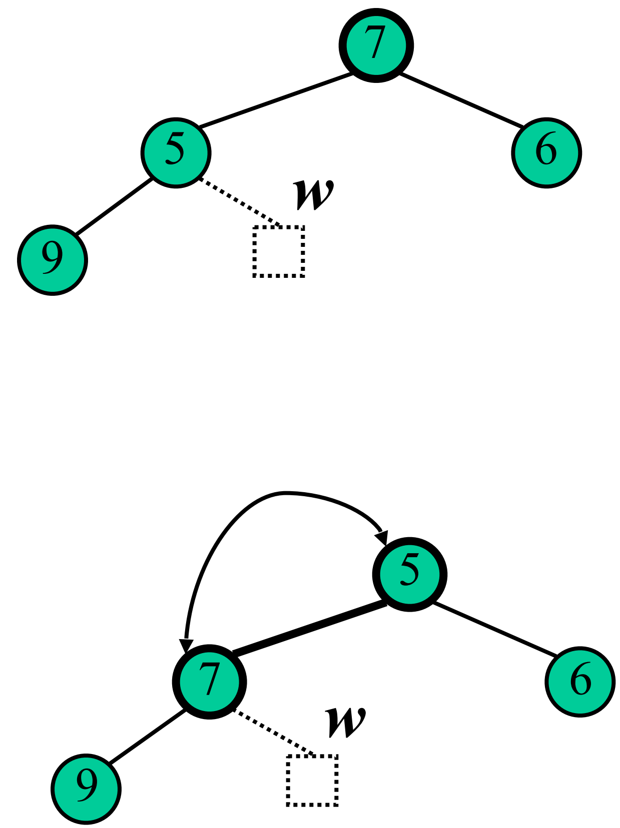
- Removing the root of the heap
  - Replace root with last node
  - Remove last node w
  - Restore heap order





Downheap

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



Heap Sort

- A PQ-sort implemented with a heap
- Space $O(n)$
- insert/poll (each) $O(logn)$
- total time $O(nlogn)$

General Removal

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