CS151 Intro to Data Structures

Priority Queues

Outline

- HW comments
- Review

Announcements

Midterm Wednesday 11/01

Closed note, closed books

HW04 due next Friday 11/10
    Releasing it Wednesday the latest

No lab this Wednesday

Homework Comments

Read instructions!

If we ask for specific things, include them

Start them early

`Comparable` and Generics

`implements Comparable` **vs** `extends Comparable`**?**

Method Summary

All Methods	Instance Methods	Abstract Methods

Modifier and Type	Method and Description
`int`	**`compareTo`**`(T o)` Compares this object with the specified object for order.

`Comparable` and Generics

Method Detail

compareTo

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters:

`o - the object to be compared.`

Returns:

`a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.`

Throws:

`NullPointerException - if the specified object is null`

`ClassCastException - if the specified object's type prevents it from being compared to this object.`

`Comparable` and Generics

`implements Comparable` **vs** `extends Comparable`**?**

A class that `implements Comparable` **supports object comparison**

- `Place implements Comparable`
- `Place implements Comparable<Place>`

Comparable and Generics

A generic data structure (specified via an interface) with `<T extends <Comparable T>>` requires that type parameter (i.e. objects stored in this structure) must support comparison with other instances of its own type

- `public interface BinaryTree<E extends Comparable<E>>`
- `public class LinkedBinaryTree<E extends Comparable<E>> implements BinaryTree<E>`

Comparable and Generics

- `public class LinkedBinaryTree<E extends Comparable<E>> implements Comparable, BinaryTree<E>`

- `public class LinkedBinaryTree<E extends Comparable<E>> implements Comparable<LinkedBinaryTree>, BinaryTree<E>`

- `public class LinkedBinaryTree<E extends Comparable<E>> implements Comparable<LinkedBinaryTree<E>>, BinaryTree<E>`

Queue

First-in First-out

Priority Queue

Priority Queue

A queue that maintains the order of the elements according to some priority

- generally not FIFO
- some other order (although time of insertion COULD be one criteria)

Removal order, not general order

- object with minkey/maxkey in front
- the rest may or may not be sorted (implementation dependent)

Key

Objects have many instance variables

Priority queues are ordered by some key, which may be one of the instance variables, or combinations of many

- what does it mean for a `Place` to be bigger or smaller than another `Place`?

Consistent with `compareTo`

Simply rewrite `compareTo` to rekey/reorder objects

Key-Value Pair

Frequently used pairing in lookup

Keys are unique identifiers

Keys can be easily mapped to numerical values

Values are data the objects store

- not numerical/unique
- data or references to data

Values can be directly used as keys

Entry Interface

```
public interface Entry<K extends Comparable<K>, V> {

  K getKey();
  V getValue();

}
```

Priority Queue ADT

insert(k, v): Creates an entry with key k and value v in the priority queue.

min( ): Returns (but does not remove) a priority queue entry (k,v) having minimal key; returns null if the priority queue is empty.

removeMin( ): Removes and returns an entry (k,v) having minimal key from the priority queue; returns null if the priority queue is empty.

size( ): Returns the number of entries in the priority queue.

isEmpty( ): Returns a boolean indicating whether the priority queue is empty.

Example - minPQ

Method	Return Value	Priority Queue Contents

Example - minPQ

Method	Return Value	Priority Queue Contents
insert(5,A)		

Example - minPQ

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }

Example - minPQ

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

skip

skip

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	{ }

Implementing a Priority Queue

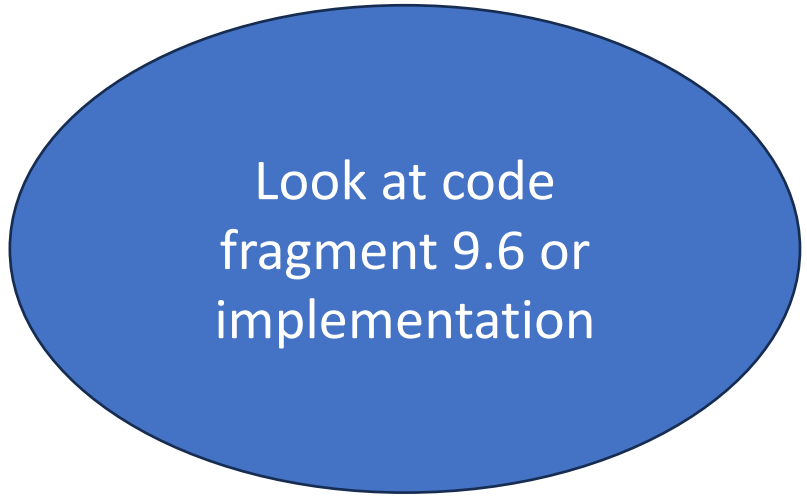
What might be a simple way to implement a priority queue?

insert(k, v):	Creates an entry with key k and value v in the priority queue.
min( ):	Returns (but does not remove) a priority queue entry (k,v) having minimal key; returns null if the priority queue is empty.
removeMin( ):	Removes and returns an entry (k,v) having minimal key from the priority queue; returns null if the priority queue is empty.
size( ):	Returns the number of entries in the priority queue.
isEmpty( ):	Returns a boolean indicating whether the priority queue is empty.

Implementing a Priority Queue

What might be a simple way to implement a priority queue?

Use a list!



insert(k, v)

- Add the new item to the end of the list

min():

- Search through all the elements and find the element with the smallest key

Implementing a Priority Queue w/ a List

insert(k, v)

- Add the new item to the end of the list

min():

- Search through all the elements and find the element with the smallest key

Method	Running Time
size	
isEmpty	
insert	
min	
removeMin	

Implementing a Priority Queue w/ a List

insert(k, v)

- Add the new item to the end of the list

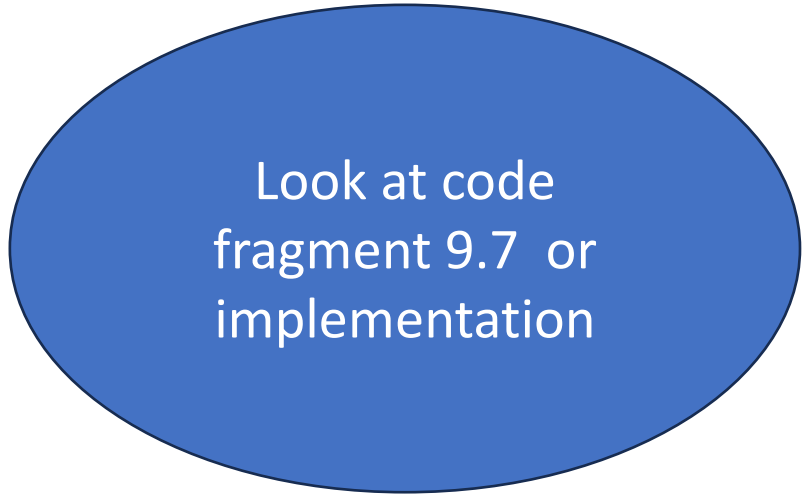
min():

- Search through all the elements and find the element with the smallest key

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
insert	$O(1)$
min	$O(n)$
removeMin	$O(n)$

Implementing a Priority Queue - SortedList

Use a sorted list!



insert(k, v)

- Find where to put the item based on k, then move other items over

min():

- Find the first element in the list

Implementing a Priority Queue - SortedList

Method	Unsorted List	Sorted List
size	$O(1)$	
isEmpty	$O(1)$	
insert	$O(1)$	
min	$O(n)$	
removeMin	$O(n)$	

Implementing a Priority Queue - SortedList

Method	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

Priority Queue

Minimum Priority Queue vs Maximum Priority Queue

- Ascending vs Descending Order

min( ):	Returns (but does not remove) a priority queue entry (k,v) having minimal key; returns null if the priority queue is empty.
removeMin( ):	Removes and returns an entry (k,v) having minimal key from the priority queue; returns null if the priority queue is empty.

`poll`: removeMin() or removeMax()

`peek`: min() or max()

Sorting with a Priority Queue

Algorithm:

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`

	Unsorted array	Unsorted list	Sorted array	Sorted list
poll	$O(n)$	$O(n)$	$O(1)$	$O(1)$
insert	$O(1)^*$	$O(1)$	$O(n)$	$O(n)$

Selection Sort

Select the min/max and swap with first position (0-index position)

A variation of PQ-sort where the priority queue is implemented with an unsorted list

Runtime:

- $O(n^2)$

Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()

Example

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

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)

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		

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)

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

insert/swap the element into the correct sorted position

A variation of PQ-sort where the priority queue is implemented with a sorted sequence

Runtime

- $O(n^2)$

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)	()

Priority Queue – summary

Sorted List

- Inserting is O(n)
- Polling is O(1)

Unsorted List

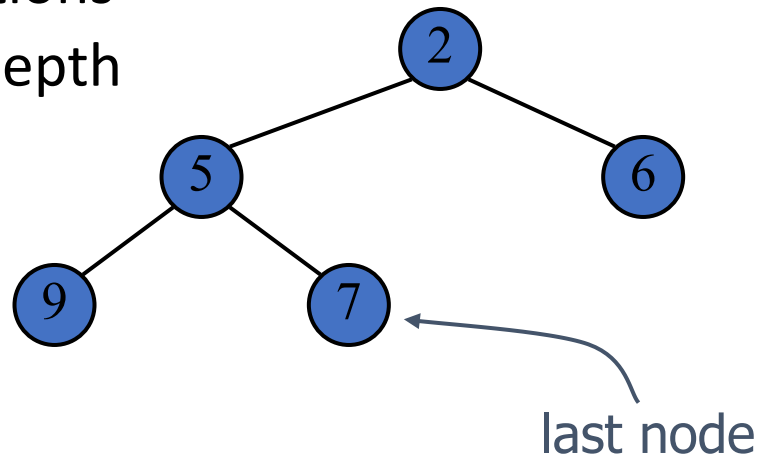
- Inserting is O(1)
- Polling is O(n)

insert(k, v):	Creates an entry with key k and value v in the priority queue.
min( ):	Returns (but does not remove) a priority queue entry (k,v) having minimal key; returns null if the priority queue is empty.
removeMin( ):	Removes and returns an entry (k,v) having minimal key from the priority queue; returns null if the priority queue is empty.
size( ):	Returns the number of entries in the priority queue.
isEmpty( ):	Returns a boolean indicating whether the priority queue is empty.

Binary Heap

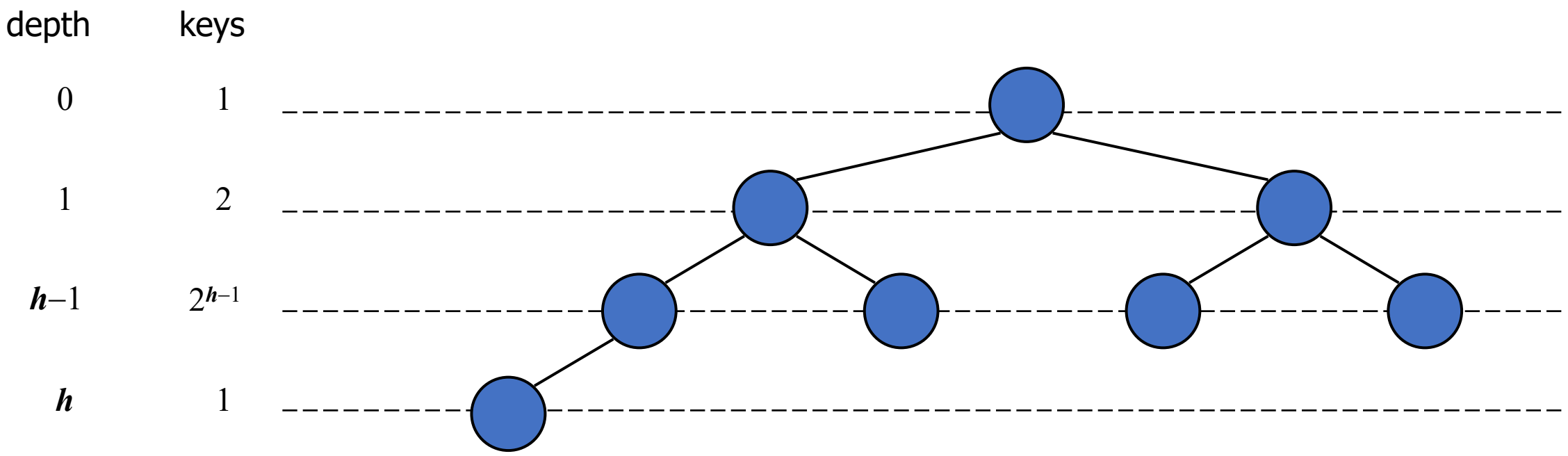
Binary tree storing keys at its nodes and satisfying:

1. heap-order: for every internal node v other than root, $key(v) \geq key(parent(v))$
2. 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 $O(logn)$

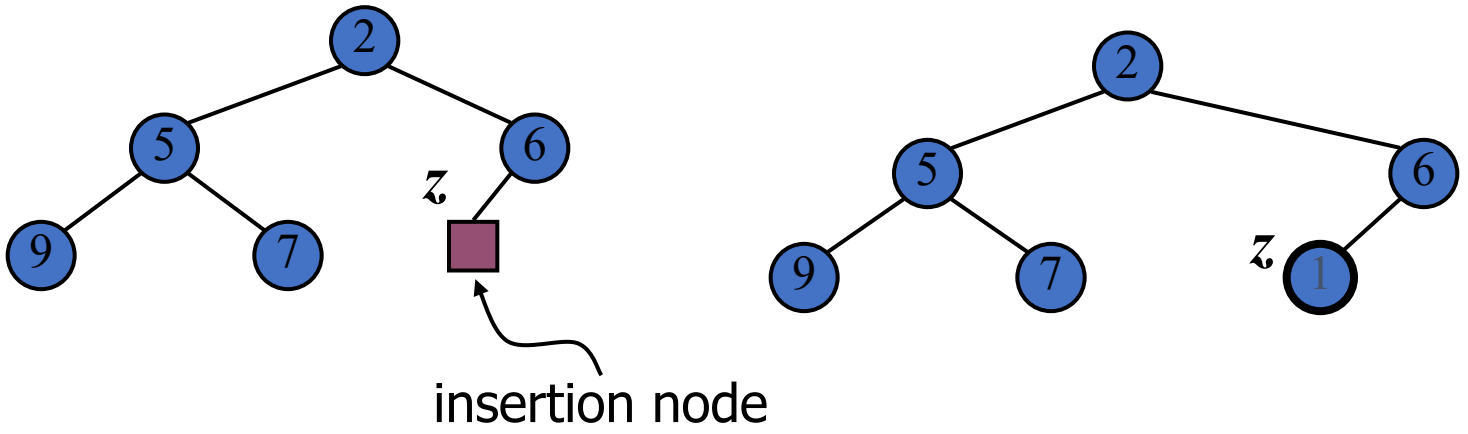


Insertion into a Heap

Insert as new last node

Need to restore heap order

- Up-heap bubbling

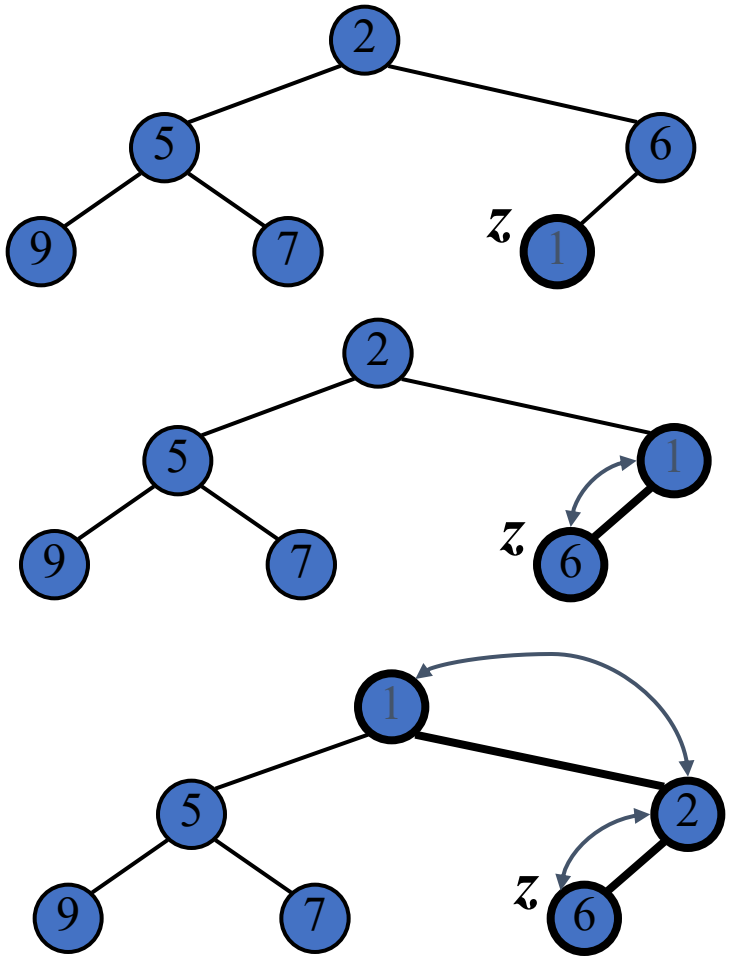


Upheap

Restore heap order

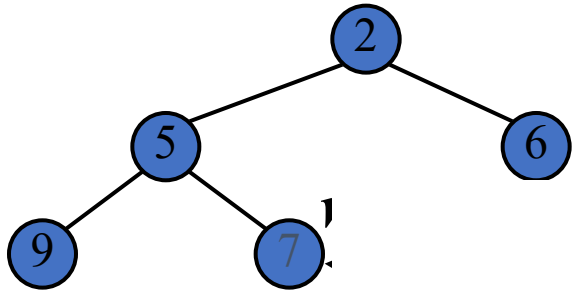
- swap upwards
- stop when finding a smaller parent
- or reach root

Runtime: $O(logn)$



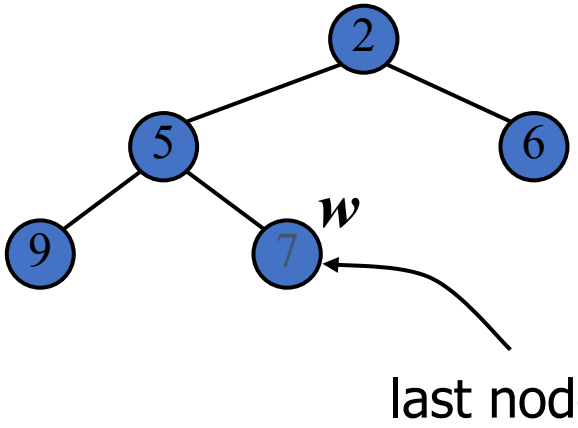
Poll

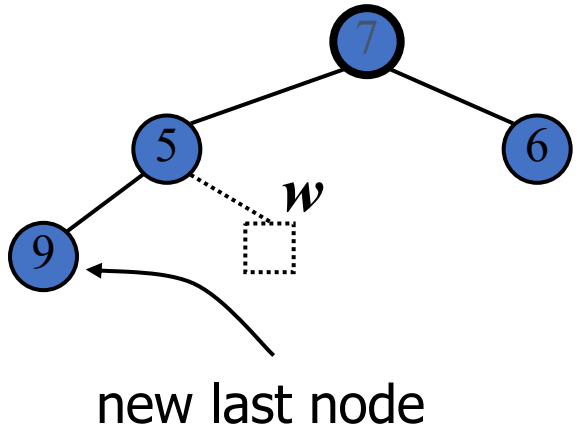
Removing the root of the heap



What becomes the new root?

- Replace root with last node
- Remove last node w
- Restore heap order





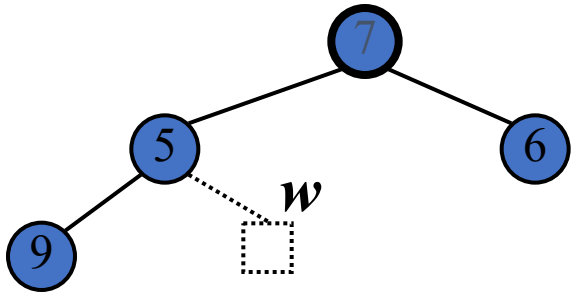
Downheap

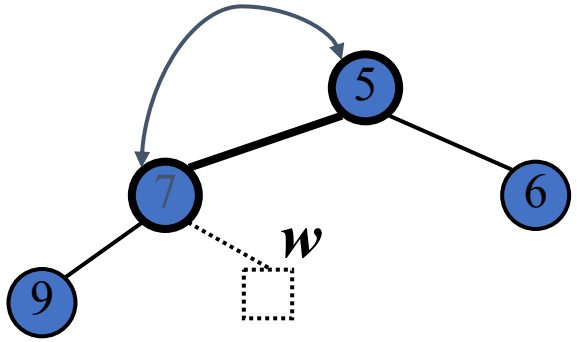
Restore heap order

- swap downwards
- swap with smaller child
- stop when finding larger children
- or reach a leaf

Runtime:

- $O(logn)$





Implementing a Priority Queue – Binary Heap

Method	Running Time
size, isEmpty	
min	
insert	
removeMin	

*amortized, if using dynamic array

Implementing a Priority Queue – Binary Heap

Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

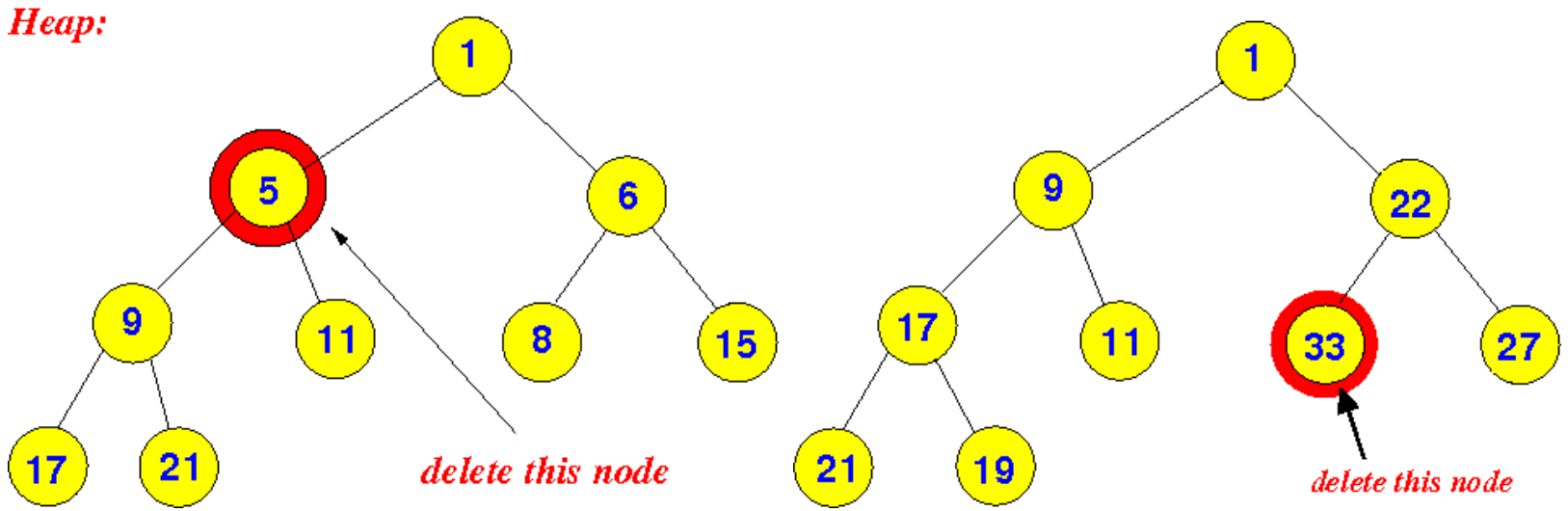
*amortized, if using dynamic array

Heap Sort

- A PQ-sort implemented with a heap
- Space $O(n)$
- insert/poll $O(logn)$
- total $O(nlogn)$
- You have just encountered your first O$(nlogn)$ sorting algorithm!

Remove

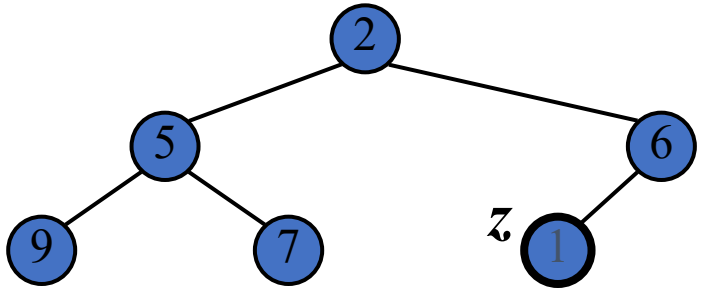
- swap with last node
- delete last node
- may need to upheap or downheap



Find/Search in binary heap

Find 1

- How many elements would we need to search for?



Runtime of search in binary heap?

- $O(n)$