CSE 250

Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 22: Priority Queues

Announcements

- PA2 testing due on Sunday @ 11:59PM
  - No late submissions or grace days accepted
  - AutoLab now open
  - Can get a 6/5 with bonus point

Examples

<u>How might we order the following?</u>

- (B,10), (D,3), (E,40)
- "A+", "C", "B-"
- Taco Tuesday, Fish Friday, Meatless Monday
- Buffalo Bills, Denver Broncos, Baltimore Ravens
- Halloween, Friday the 13th, The Babadook

Ordering

An **ordering (over type A)**, **(*A*, ≤)**:

- A set of things of type ***A***
- A "relation" or comparator, **≤**, that relates two things in the set

Examples

5 ≤ 30 ≤ 999
Numerical order

(E,40) ≤ (B,10) ≤ (D,3)
Reverse-numerical order on the 2nd field

C+ ≤ B- ≤ B ≤ B+ ≤ A- ≤ A
Letter grades

AA ≤ AM ≤ BZ ≤ CA ≤ CD
Compare 1st then 2nd, 3rd…(Lexical order)

Ordering Properties

Team A ≤ Team B

Team B won its match against Team A

Ordering Properties

Team A ≤ Team B

Team B won its match against Team A

Team B ≤ Team C

Team C won its match against Team B

Ordering Properties

Team A ≤ Team B

Team B won its match against Team A

Team B ≤ Team C

Team C won its match against Team B

Team C ≤ Team A

Team A won its match against Team C

Ordering Properties

Team A ≤ Team B

Team B won its match against Team A

Team B ≤ Team C

Team C won its match against Team B

Team C ≤ Team A

Team A won its match against Team C

Is this an ordering??

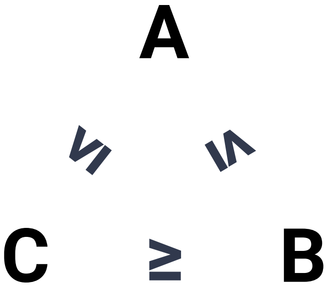
Ordering Properties

Team A ≤ Team B
Team B won its match against Team A

Team B ≤ Team C
Team C won its match against Team B

Team C ≤ Team A
Team A won its match against Team C

Is this an ordering??



Ordering Properties

Team A ≤ Team B

Team B won its match against Team A

Team B ≤ Team C

Team C won its match against Team B

Team C ≤ Team A

Team A won its match against Team C

Is this an ordering?? **NO!**

A

No Transitivity!

C ≤ B

Ordering Properties

An ordering must be…

Reflexive

$x \leq x$

Antisymmetric

If $x \leq y$ and $y \leq x$ then $x = y$

Transitive

If $x \leq y$ and $y \leq z$ then $x \leq z$

Another Example

Define an ordering over CSE Courses:

Course 1 ≤ Course 2 iff Course 1 is a prereq of Course 2

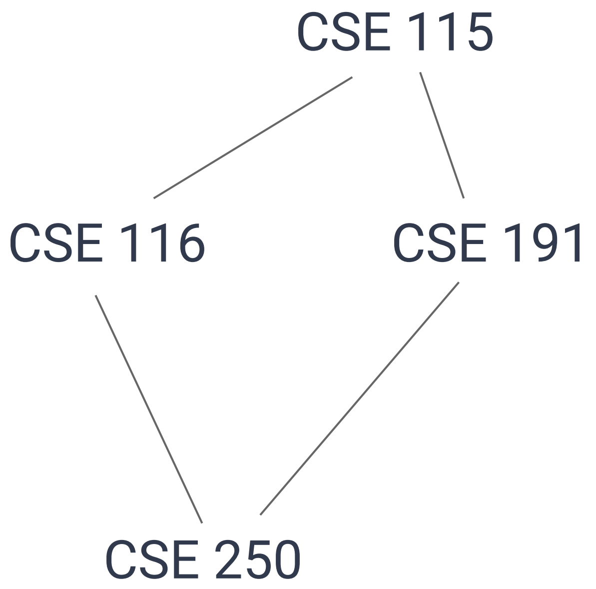
CSE 115 ≤ CSE 116

CSE 116 ≤ CSE 250

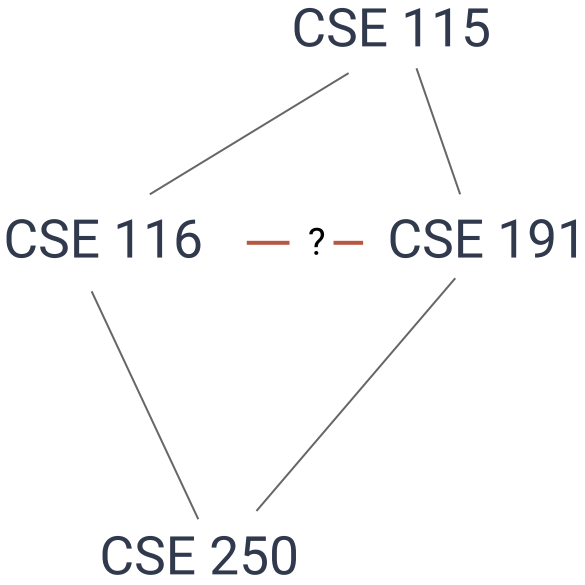
CSE 115 ≤ CSE 191

CSE 191 ≤ CSE 250

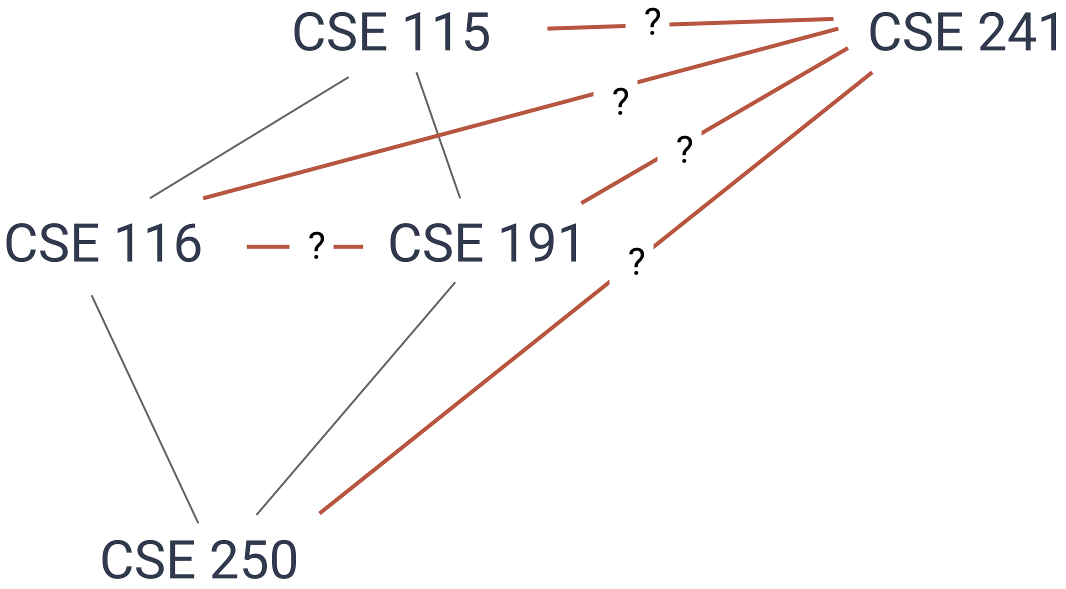
Ordering Properties



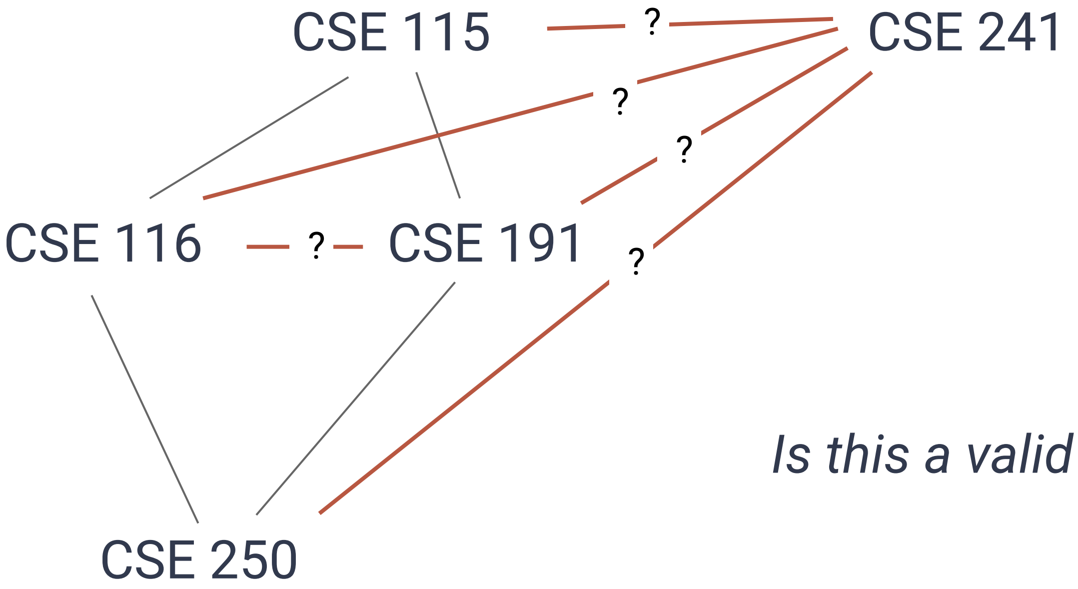
Ordering Properties



Ordering Properties

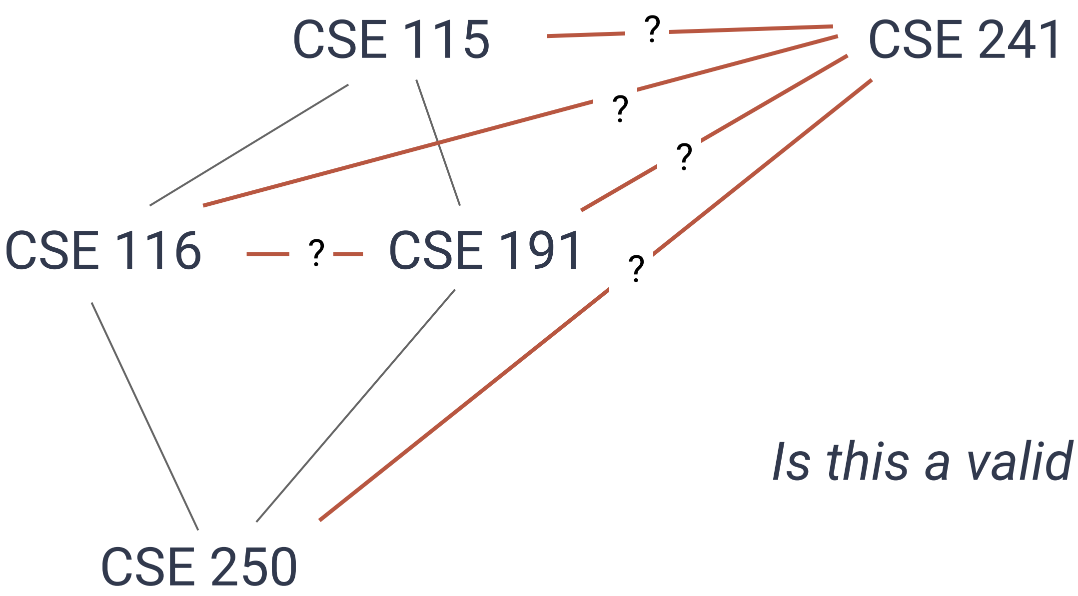


Ordering Properties



Is this a valid ordering?

Ordering Properties



Is this a valid ordering? **YES**

(Partial) Ordering Properties

A **partial ordering** must be…

Reflexive

$x \leq x$

Antisymmetric

If $x \leq y$ and $y \leq x$ then $x = y$

Transitive

If $x \leq y$ and $y \leq z$ then $x \leq z$

(Total) Ordering Properties

An **<u>total ordering</u>** must be…

Reflexive

$x \leq x$

Antisymmetric

If $x \leq y$ and $y \leq x$ then $x = y$

Transitive

If $x \leq y$ and $y \leq z$ then $x \leq z$

Complete

Either $x \leq y$ or $y \leq x$ for any $x,y \in A$

Example

Consider two different ways to "rank" movies:

Halloween, It, Hereditary, Get Out, Descent, Friday the 13th

Example

Consider two different ways to "rank" movies:

Halloween, It, Hereditary, Get Out, Descent, Friday the 13th

I could organize these movies in a tier list based on my preferences:

A-tier: Halloween, Get Out, Friday the 13th

B-tier: It, Descent

C-tier: Hereditary

Example

Consider two different ways to "rank" movies:

Halloween, It, Hereditary, Get Out, Descent, Friday the 13th

I could organize these movies in a tier list based on my preferences:

A-tier: Halloween, Get Out, Friday the 13th

B-tier: It, Descent

C-tier: Hereditary

This is a partial ordering
It is reflexive, antisymmetric and transitive
…but not all pairs are directly order
Consider Halloween and Friday the 13th

Example

Consider two different ways to "rank" movies:

Halloween, It, Hereditary, Get Out, Descent, Friday the 13th

I could also rank these movies based on my preferences:

1. Halloween
2. Get Out
3. Friday the 13th
4. Descent
5. It
6. Hereditary

Example

Consider two different ways to "rank" movies:

Halloween, It, Hereditary, Get Out, Descent, Friday the 13th

I could also rank these movies based on my preferences:

1. Halloween
2. Get Out
3. Friday the 13th
4. Descent
5. It
6. Hereditary

This is a total ordering
It is reflexive, antisymmetric and transitive
…and every pair can be directly compared

Some Other Definitions

For an ordering **(*A*, ≤)**

The **<u>greatest</u>** element is an element $x \in A$ s.t. there is no ***y*** **in** ***A***, where $x \leq y$

The **<u>least</u>** element is an element $x \in A$ s.t. there is no ***y*** **in** ***A***, where $y \leq x$

Some Other Definitions

For an ordering **(A, ≤)**

The **greatest** element is an element $x \in A$ s.t. there is no y **in** A, where $x \leq y$

The **least** element is an element $x \in A$ s.t. there is no y **in** A, where $y \leq x$

*A **partial** ordering may not have a **unique** greatest/least element*

Describing an Ordering

≤ can be described **explicitly**, by a set of tuples:

$\{(a,a),(a,b),(a,c),\ldots,(b,b),\ldots,(z,z)\}$

Describing an Ordering

≤ can be described **explicitly**, by a set of tuples:

{(a,a),(a,b),(a,c),…,(b,b),…,(z,z)}

If (x,y) is in the set, then x ≤ y

Describing an Ordering

≤ can be described by a **mathematical rule**:

$$\{(x,y) \mid x, y \in \mathbb{Z},\ \exists\ a \in \mathbb{Z}^{+} \cup \{0\} : x + a = y\}$$

Describing an Ordering

≤ can be described by a **mathematical rule**:

$\{(x,y) \mid x, y \in \mathbb{Z},\ \exists\ a \in \mathbb{Z}^{+} \cup \{0\} : x + a = y\}$

x ≤ y iff ***x,y*** are integers and there is a non-negative integer ***a*** s.t. ***x+a=y***

Multiple Orderings

Multiple Orderings can be defined for the same set

- RottenTomatoes vs Metacritic vs Box Office Gross
- "Best Movie" first vs "Worst Movie" first
- Rank by number of swear words, killcount, etc

Multiple Orderings

Multiple Orderings can be defined for the same set

- RottenTomatoes vs Metacritic vs Box Office Gross
- "Best Movie" first vs "Worst Movie" first
- Rank by number of swear words, killcount, etc

We use subscripts to separate orderings ($\leq_1$, $\leq_2$, $\leq_3$, …)

Transformations

We can transform orderings:

Transformations

We can transform orderings:

Reverse: If x $\leq_1$ y then define y $\leq_r$ x

Transformations

We can transform orderings:

Reverse: If $x \leq_1 y$ then define $y \leq_r x$

Lexical: Given $\leq_1, \leq_2, \leq_3, \ldots$

- if $x \leq_1 y$ then $x \leq_L y$
- else if $x =_1 y$ and $x \leq_2 y$ then $x \leq_L y$
- else if $x =_2 y$ and $x \leq_3 y$ then $x \leq_L y$
- …

Examples of Lexical Ordering

Names: First letter, then second letter, then third…

Movies: Average of reviews, then number of reviews…

Tuples: First field, then second field, then third…

Sports Teams: Games won, points scored, speed of victory…

Ordering Over Keys

≤ can be described as an **ordering over a <u>key</u> derived from the element:**

$$x \leq_{\mathbf{edge}} y \text{ iff } \mathrm{weight}(x) \leq \mathrm{weight}(y)$$

$$x \leq_{\mathbf{student}} y \text{ iff } \mathrm{name}(x) \leq_{\mathbf{Lex}} \mathrm{name}(y)$$

Ordering Over Keys

≤ can be described as an **ordering over a <u>key</u> derived from the element:**

$$x \leq_{\textbf{edge}} y \text{ iff } weight(x) \leq weight(y)$$

$$x \leq_{\textbf{student}} y \text{ iff } name(x) \leq_{\textbf{Lex}} name(y)$$

We say that weight/name are keys

Topological Sort

A **<u>Topological Sort</u>** of *partial* order $(A, \leq_1)$ is *any total* order $(A, \leq_2)$ that "agrees" with $(A, \leq_1)$:

For any two elements x,y in ***A***:

if $x \leq_1 y$ then $x \leq_2 y$

if $y \leq_1 x$ then $y \leq_2 x$

Otherwise, either $x \leq_2 y$ or $y \leq_2 x$

Topological Sort

The following are all topological sorts over our partial order from earlier:

- CSE 115, CSE 116, CSE 191, CSE 241, CSE 250
- CSE 241, CSE 115, CSE 116, CSE 191, CSE 250
- CSE 115, CSE 191, CSE 116, CSE 250, CSE 241

Topological Sort

The following are all topological sorts over our partial order from earlier:

- CSE 115, CSE 116, CSE 191, CSE 241, CSE 250
- CSE 241, CSE 115, CSE 116, CSE 191, CSE 250
- CSE 115, CSE 191, CSE 116, CSE 250, CSE 241

(In this case, the partial ordering is a schedule requirement, and each topological sort is a possible schedule)

And now for an ordering-based ADT…

A New ADT...`PriorityQueue`

`PriorityQueue<T>`

`void add(T value)`

Insert **`value`** into the priority queue

`T poll()`

Remove the highest priority value in the priority queue

`T peek()`

Peek at the highest priority value in the priority queue

A New ADT...`PriorityQueue`

`PriorityQueue<T>`

> **In Java, by default the smallest element has the highest priority**

`void add(T value)`

Insert **`value`** into the priority queue

`T poll()`

Remove the **highest priority** value in the priority queue

`T peek()`

Peek at the **highest priority** value in the priority queue

Sorted Lists

Note this is not the first time we've seen an ordering-based data structure

Consider our SortedList from PA1…

How do we store the following→

How do we store the following→

```
add(5)
```

How do we store the following→

```
add(5)
add(9)
```

How do we store the following→

```
add(5)
add(9)
add(2)
```

How do we store the following→

```
add(5)
add(9)
add(2)
add(7)
```

How do we store the following→

```
add(5)
add(9)
add(2)
add(7)
peek()    // Should be 9
poll()    // should be 9
```

How do we store the following→

```
add(5)
add(9)
add(2)
add(7)
peek()    // Should be 9
poll()    // should be 9
size()    // should be 3
peek()    // should be 7
```

How do we store the following→

```
add(5)
add(9)
add(2)
add(7)
peek()    // Should be 9
poll()    // should be 9
size()    // should be 3
peek()    // should be 7
poll()    // 7
poll()    // 5
poll()    // 2
```

How do we store the following→

```
add(5)
add(9)
add(2)
add(7)
peek()      // Should be 9
poll()      // should be 9
size()      // should be 3
peek()      // should be 7
poll()      // 7
poll()      // 5
poll()      // 2
isEmpty()   // should be true
```

How do we store the following→

Insertion Order?	5, 9, 7, 2
Sorted Order?	2, 5, 7, 9
Reverse Sorted Order?	9, 7, 5, 2

```
add(5)
add(9)
add(2)
add(7)
peek()     // Should be 9
poll()     // should be 9
size()     // should be 3
peek()     // should be 7
poll()     // 7
poll()     // 5
poll()     // 2
isEmpty() // should be true
```

Priority Queues

Two mentalities…

Lazy: Keep everything a mess

Proactive: Keep everything organized

Priority Queues

Two mentalities…

Lazy: Keep everything a mess ("Selection Sort")

Proactive: Keep everything organized

Priority Queues

Two mentalities…

Lazy: Keep everything a mess ("Selection Sort")

Proactive: Keep everything organized ("Insertion Sort")

Lazy Priority Queue

Base Data Structure: Linked List

`void add(T value)`

Append `value` to the end of the linked list.

`T peek()/T poll()`

Traverse the list to find the smallest value.

Lazy Priority Queue

Base Data Structure: Linked List

`void add(T value)`

Append `value` to the end of the linked list. $\Theta(1)$

`T peek()/T poll()`

Traverse the list to find the smallest value. $O(n)$

Sorting with Our Priority Queue

```
public List<T> PQueueSort(List<T> input) {
  List<T> out = new ArrayList<>();
  PriorityQueue<T> pq = new PriorityQueue<>();
  for (T item : input) { pq.add(item); }
  while (!pq.isEmpty()) { out.add(pq.poll()); }
  return out;
}
```

Sorting with Our Priority Queue

```java
public List<T> PQueueSort(List<T> input) {
  List<T> out = new ArrayList<>();
  PriorityQueue<T> pq = new PriorityQueue<>();
  for (T item : input) { pq.add(item); }   ← Add everything to a priority queue
  while (!pq.isEmpty()) { out.add(pq.poll()); }
  return out;
}
```

Sorting with Our Priority Queue

```
public List<T> PQueueSort(List<T> input) {
  List<T> out = new ArrayList<>();
  PriorityQueue<T> pq = new PriorityQueue<>();
  for (T item : input) { pq.add(item); }
  while (!pq.isEmpty()) { out.add(pq.poll()); }  ← Remove it all (and add to out)
  return out;
}
```

Selection Sort
(w/"Lazy" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()

Selection Sort
(w/"Lazy" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)

Selection Sort
(w/"Lazy" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(7,4)

Selection Sort
(w/"Lazy" PriorityQueue)

	List	**PriorityQueue**
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(7,4)
	…	…
Step *n*	()	(7,4,8,2,5,3,9)

Selection Sort
(w/"Lazy" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(7,4)
	…	…
Step *n*	()	(7,4,8,2,5,3,9)
Step *n* + 1	[2,_,_,_,_,_,_]	(7,4,8,5,3,9)

Selection Sort
(w/"Lazy" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(7,4)
	…	…
Step *n*	()	(7,4,8,2,5,3,9)
Step *n* + 1	[2,_,_,_,_,_,_]	(7,4,8,5,3,9)
Step *n* + 2	[2,3,_,_,_,_,_]	(7,4,8,5,9)

Selection Sort
(w/"Lazy" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(7,4)
	…	…
Step *n*	()	(7,4,8,2,5,3,9)
Step *n* + 1	[2,_,_,_,_,_,_]	(7,4,8,5,3,9)
Step *n* + 2	[2,3,_,_,_,_,_]	(7,4,8,5,9)
Step *n* + 3	[2,3,4,_,_,_,_]	(7,8,5,9)

Selection Sort
(w/"Lazy" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(7,4)
	…	…
Step *n*	()	(7,4,8,2,5,3,9)
Step *n* + 1	[2,_,_,_,_,_,_]	(7,4,8,5,3,9)
Step *n* + *2*	[2,3,_,_,_,_,_]	(7,4,8,5,9)
Step *n* + *3*	[2,3,4,_,_,_,_]	(7,8,5,9)
Step *n* + *4*	[2,3,4,5,_,_,_]	(7,8,9)

Selection Sort
(w/"Lazy" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(7,4)
	…	…
Step *n*	()	(7,4,8,2,5,3,9)
Step *n* + 1	[2,_,_,_,_,_,_]	(7,4,8,5,3,9)
Step *n* + *2*	[2,3,_,_,_,_,_]	(7,4,8,5,9)
Step *n* + *3*	[2,3,4,_,_,_,_]	(7,8,5,9)
Step *n* + *4*	[2,3,4,5,_,_,_]	(7,8,9)
	…	…
Step 2*n*	[2,3,4,5,7,8,9]	()

Selection Sort (w/"Lazy" PriorityQueue)

```
public List<T> PQueueSort(List<T> input) {
  List<T> out = new ArrayList<>();
  PriorityQueue<T> pq = new PriorityQueue<>();
  for (T item : input) { pq.add(item); }
  while (!pq.isEmpty()) { out.add(pq.poll()); }
  return out;
}
```

What is the complexity?

Selection Sort (w/"Lazy" PriorityQueue)

```
public List<T> PQueueSort(List<T> input) {
  List<T> out = new ArrayList<>();
  PriorityQueue<T> pq = new PriorityQueue<>();
  for (T item : input) { pq.add(item); }
  while (!pq.isEmpty()) { out.add(pq.poll()); }
  return out;
}
```

← **`poll()`** is an ***O*(*n*)** operation

What is the complexity? ***O*(*n*²)**

Proactive Priority Queue

Base Data Structure: Linked List

`void add(T value)`

Insert `value` in ascending sorted order.

`T peek()/T poll()`

Get the first value in the list.

Proactive Priority Queue

Base Data Structure: Linked List

`void add(T value)`

Insert `value` in ascending sorted order. $O(n)$

`T peek()/T poll()`

Get the first value in the list. $\Theta(1)$

Insertion Sort
(w/"Proactive" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()

Insertion Sort
(w/"Proactive" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)

Insertion Sort
(w/"Proactive" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(4,7)

Insertion Sort
(w/"Proactive" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(4,7)
Step 3	(2,5,3,9)	(4,7,8)

Insertion Sort
(w/"Proactive" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(4,7)
Step 3	(2,5,3,9)	(4,7,8)
Step 4	(5,3,9)	(2,4,7,8)
	…	…
Step *n*	[_,_,_,_,_,_,_]	(2,3,4,5,7,8,9)

Insertion Sort
(w/"Proactive" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(4,7)
Step 3	(2,5,3,9)	(4,7,8)
Step 4	(5,3,9)	(2,4,7,8)
	…	…
Step *n*	[_,_,_,_,_,_,_]	(2,3,4,5,7,8,9)
Step *n* + 2	[2,_,_,_,_,_,_]	(3,4,5,7,8,9)

Insertion Sort
(w/"Proactive" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(4,7)
Step 3	(2,5,3,9)	(4,7,8)
Step 4	(5,3,9)	(2,4,7,8)
	…	…
Step *n*	[_,_,_,_,_,_,_]	(2,3,4,5,7,8,9)
Step *n* + *2*	[2,_,_,_,_,_,_]	(3,4,5,7,8,9)
Step *n* + *3*	[2,3,_,_,_,_,_]	(4,5,7,8,9)

Insertion Sort
(w/"Proactive" PriorityQueue)

	List	PriorityQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)
Step 2	(8,2,5,3,9)	(4,7)
Step 3	(2,5,3,9)	(4,7,8)
Step 4	(5,3,9)	(2,4,7,8)
	…	…
Step *n*	[_,_,_,_,_,_,_]	(2,3,4,5,7,8,9)
Step *n* + 2	[2,_,_,_,_,_,_]	(3,4,5,7,8,9)
Step *n* + 3	[2,3,_,_,_,_,_]	(4,5,7,8,9)
	…	…
Step 2*n*	[2,3,4,5,7,8,9]	()

Insertion Sort (w/"Proactive" PriorityQueue)

```java
public List<T> PQueueSort(List<T> input) {
  List<T> out = new ArrayList<>();
  PriorityQueue<T> pq = new PriorityQueue<>();
  for (T item : input) { pq.add(item); }
  while (!pq.isEmpty()) { out.add(pq.poll()); }
  return out;
}
```

What is the complexity?

Insertion Sort (w/"Proactive" PriorityQueue)

```
public List<T> PQueueSort(List<T> input) {
  List<T> out = new ArrayList<>();
  PriorityQueue<T> pq = new PriorityQueue<>();
  for (T item : input) { pq.add(item); }
  while (!pq.isEmpty()) { out.add(pq.poll()); }
  return out;
}
```

← **add()** is an $O(n)$ operation

What is the complexity? $\boldsymbol{O(n^2)}$

Priority Queues

Operation	Lazy	Proactive
enqueue	$O(1)$	$O(n)$
dequeue	$O(n)$	$O(1)$
head	$O(n)$	$O(1)$

Priority Queues

Operation	Lazy	Proactive
enqueue	$O(1)$	$O(n)$
dequeue	$O(n)$	$O(1)$
head	$O(n)$	$O(1)$

Can we do better?

Priority Queues

Lazy - Fast Enqueue, Slow Dequeue

Proactive - Slow Enqueue, Fast Dequeue

Priority Queues

Lazy - Fast Enqueue, Slow Dequeue

Proactive - Slow Enqueue, Fast Dequeue

??? - Fast(-ish) Enqueue, Fast(-ish) Dequeue

Priority Queues

Idea: Keep the priority queue "kinda" sorted.

Hopefully "kinda" sorted is cheaper to maintain than a full sort,

but still gives us some of the benefits.