CSE 250
Data Structures

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

Orderings and Priority Queues

Examples

How might we order the following?

- (B,10), (D,3), (E,40)
- "A+", "C", "B-"
- Taco Tuesday, Fish Friday, Meatless Monday
- Buffalo Bills, Denver Broncos, Baltimore Ravens
- Aardvark, Baboon, Capybara, Donkey, Echidna

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 first 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 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 B

Is this an ordering??

Ordering Properties

Team A ≤ Team B

Team B won its match against Team A

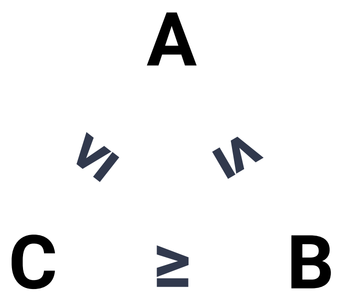
Team B ≤ Team C

Team C won its match against Team B

Team C ≤ Team A

Team A won its match against Team B

Is this an ordering??



Ordering Properties

Team A ≤ Team B

Team B won its match against Team A

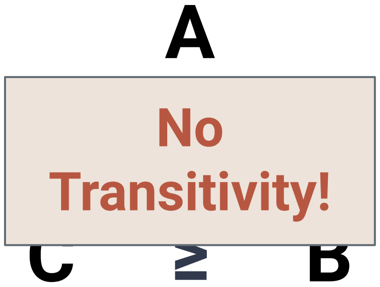
Team B ≤ Team C

Team C won its match against Team B

Team C ≤ Team A

Team A won its match against Team B

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



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

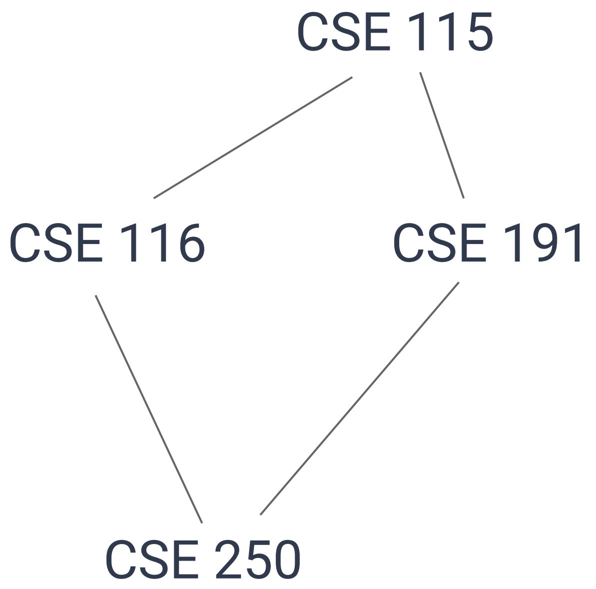
CSE 115 ≤ CSE 116

CSE 116 ≤ CSE 250

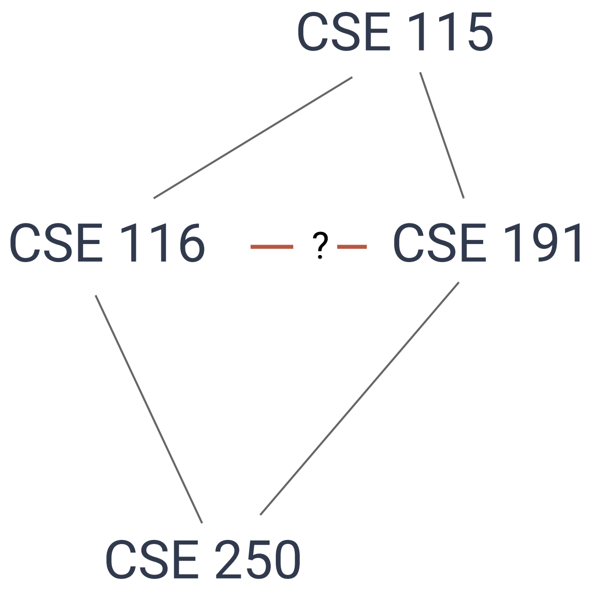
CSE 115 ≤ CSE 191

CSE 191 ≤ CSE 250

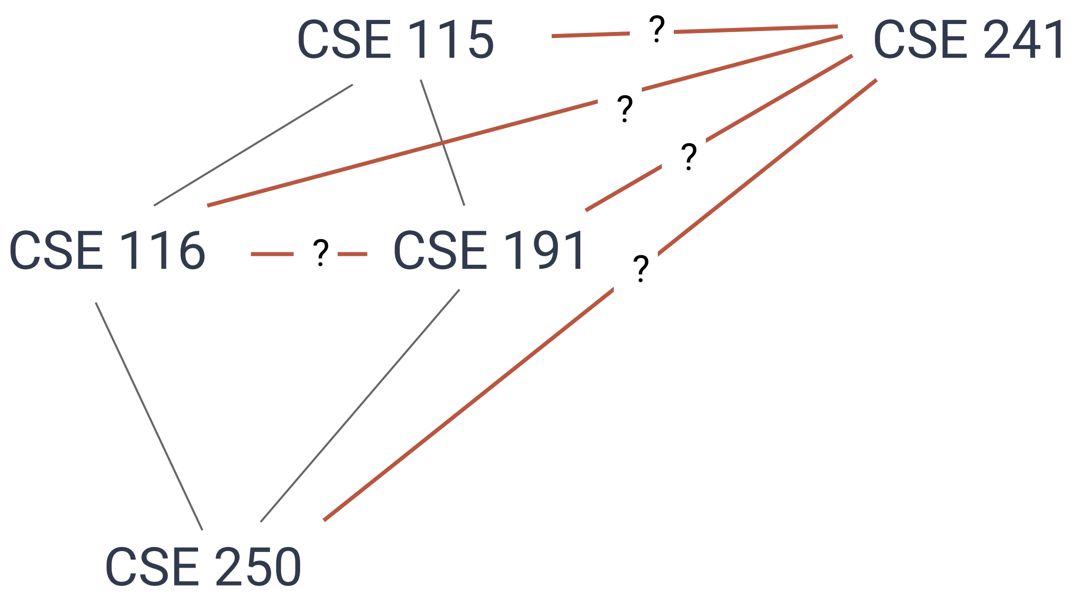
Ordering Properties



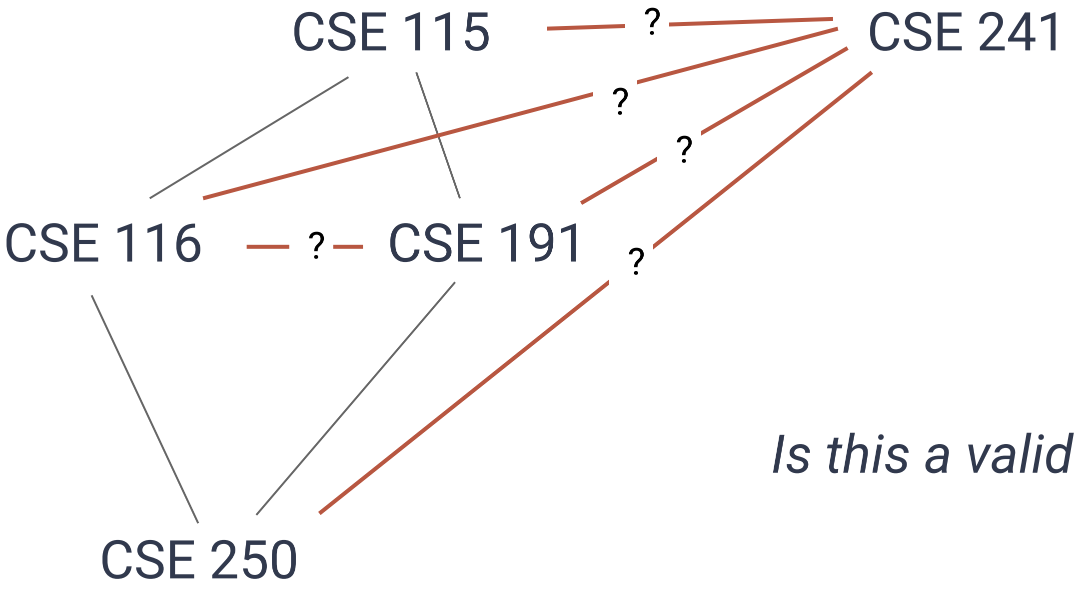
Ordering Properties



Ordering Properties

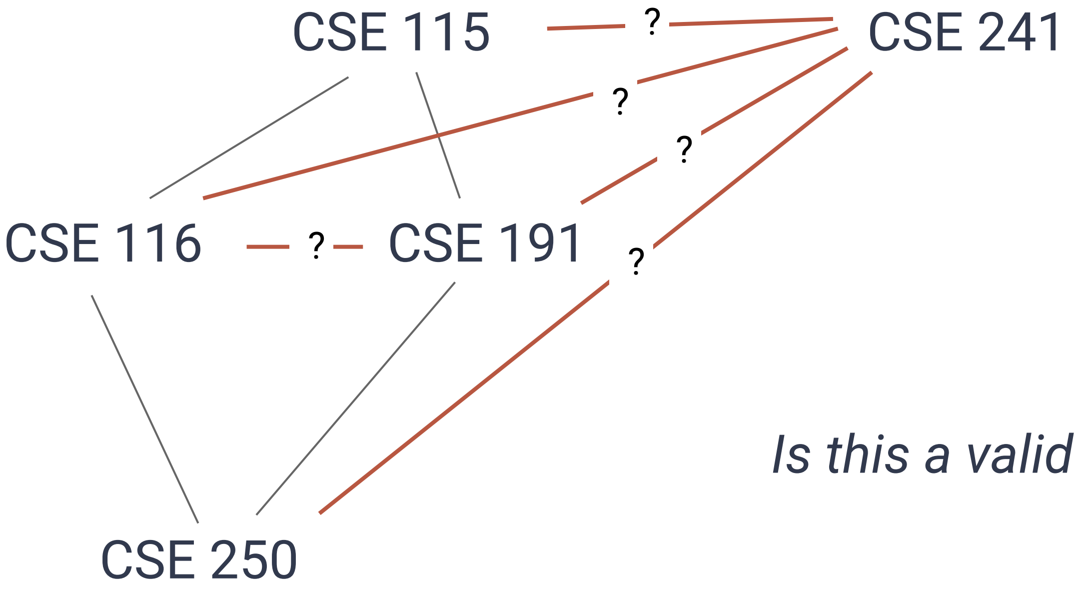


Ordering Properties



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 **total 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 $x \leq z$

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

Some Other Definitions

For an ordering $(A, \leq)$

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 $y \leq x$

Some Other Definitions

For an ordering **(*A*, ≤)**

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

The **<u>least</u>** element is an element ***x* ∈ *A*** s.t. there is no ***y* in *A***, where ***y* ≤ *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),…,(b,b),…,(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

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

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

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:**

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

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

Ordering Over Keys

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

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

$x \leq_{\textbf{student}} y$ 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[A <:Ordering]`

`enqueue(v: A): Unit`

Insert value ***v*** into the priority queue

`dequeue: A`

Remove the greatest element in the priority queue

`head: A`

Peek at the greatest element in the priority queue

How do we store the following→

How do we store the following→

```
enqueue(5)
```

How do we store the following→

```
enqueue(5)
enqueue(9)
```

How do we store the following→

```
enqueue(5)
enqueue(9)
enqueue(2)
```

How do we store the following→

```
enqueue(5)
enqueue(9)
enqueue(2)
enqueue(7)
```

How do we store the following→

```
enqueue(5)
enqueue(9)
enqueue(2)
enqueue(7)
head        // Should be 9
dequeue     // should be 9
```

How do we store the following→

```
enqueue(5)
enqueue(9)
enqueue(2)
enqueue(7)
head      // Should be 9
dequeue   // should be 9
size      // should be 3
head      // should be 7
```

How do we store the following→

```
enqueue(5)
enqueue(9)
enqueue(2)
enqueue(7)
head       // Should be 9
dequeue    // should be 9
size       // should be 3
head       // should be 7
dequeue    // 7
dequeue    // 5
dequeue    // 2
```

How do we store the following→

```
enqueue(5)
enqueue(9)
enqueue(2)
enqueue(7)
head       // Should be 9
dequeue    // should be 9
size       // should be 3
head       // should be 7
dequeue    // 7
dequeue    // 5
dequeue    // 2
isEmpty    // should be true
```

How do we store the following→

Insertion Order? 5, 9, 7, 2

```
enqueue(5)
enqueue(9)
enqueue(2)
enqueue(7)
head       // Should be 9
dequeue    // should be 9
size       // should be 3
head       // should be 7
dequeue    // 7
dequeue    // 5
dequeue    // 2
isEmpty    // should be true
```

How do we store the following→

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

```
enqueue(5)
enqueue(9)
enqueue(2)
enqueue(7)
head       // Should be 9
dequeue    // should be 9
size       // should be 3
head       // should be 7
dequeue    // 7
dequeue    // 5
dequeue    // 2
isEmpty    // should be true
```

How do we store the following→

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

```
enqueue(5)
enqueue(9)
enqueue(2)
enqueue(7)
head      // Should be 9
dequeue   // should be 9
size      // should be 3
head      // should be 7
dequeue   // 7
dequeue   // 5
dequeue   // 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

`enqueue(v: A): Unit`

Append ***t*** to the end of the linked list.

`dequeue/head : A`

Traverse the list to find the largest value.

Lazy Priority Queue

Base Data Structure: Linked List

`enqueue(v: A): Unit`

Append ***t*** to the end of the linked list. ***O*(1)**

`dequeue/head : A`

Traverse the list to find the largest value.

Lazy Priority Queue

Base Data Structure: Linked List

```
enqueue(v: A): Unit
```

Append ***t*** to the end of the linked list. ***O*(1)**

```
dequeue/head : A
```

Traverse the list to find the largest value. ***O*(*n*)**

Sorting with Our Priority Queue

```
def pqueueSort[A](items: Seq[A], pqueue: PriorityQueue[A]): Seq[A] =
{
    val out = new Array[A](items.size)
    for(item <- items){ pqueue.enqueue(item) }
    i = out.size - 1
    while(!pqueue.isEmpty) { buffer(i) = pqueue.dequeue; i-- }
    return out.toSeq
}
```

Sorting with Our Priority Queue

```
def pqueueSort[A](items: Seq[A], pqueue: PriorityQueue[A]): Seq[A] =
{
    val out = new Array[A](items.size)
    for(item <- items){ pqueue.enqueue(item) } ← Add all items to pqueue
    i = out.size - 1
    while(!pqueue.isEmpty) { buffer(i) = pqueue.dequeue; i-- }
    return out.toSeq
}
```

Sorting with Our Priority Queue

```
def pqueueSort[A](items: Seq[A], pqueue: PriorityQueue[A]): Seq[A] =
{
    val out = new Array[A](items.size)
    for(item <- items){ pqueue.enqueue(item) } ← Add all items to pqueue
    i = out.size - 1
    while(!pqueue.isEmpty) { buffer(i) = pqueue.dequeue; i-- }
    return out.toSeq                          ^ Pull all items out of pqueue
}
```

Selection Sort

	Seq/Buffer	PQueue
Input	(7,4,8,2,5,3,9)	()

Selection Sort

	Seq/Buffer	PQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)

Selection Sort

	Seq/Buffer	PQueue
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

	Seq/Buffer	PQueue
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

	Seq/Buffer	PQueue
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	[_,_,_,_,_,_,9]	(7,4,8,2,5,3)

Selection Sort

	Seq/Buffer	PQueue
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	[_,_,_,_,_,_,9]	(7,4,8,2,5,3)
Step *n* + 2	[_,_,_,_,_,8,9]	(7,4,2,5,3,9)

Selection Sort

	Seq/Buffer	PQueue
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 $n + 1$	[_,_,_,_,_,_,9]	(7,4,8,2,5,3)
Step $n + 2$	[_,_,_,_,_,8,9]	(7,4,2,5,3,9)
Step $n + 3$	[_,_,_,_,7,8,9]	(4,2,5,3,9)

Selection Sort

	Seq/Buffer	PQueue
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	[_,_,_,_,_,_,9]	(7,4,8,2,5,3)
Step *n* + *2*	[_,_,_,_,_,8,9]	(7,4,2,5,3,9)
Step *n* + *3*	[_,_,_,_,7,8,9]	(4,2,5,3,9)
Step *n* + *4*	[_,_,_,5,7,8,9]	(4,2,3,9)

Selection Sort

	Seq/Buffer	PQueue
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	[_,_,_,_,_,_,9]	(7,4,8,2,5,3)
Step *n* + 2	[_,_,_,_,_,8,9]	(7,4,2,5,3,9)
Step *n* + 3	[_,_,_,_,7,8,9]	(4,2,5,3,9)
Step *n* + 4	[_,_,_,5,7,8,9]	(4,2,3,9)
	…	…
Step 2*n*	[2,3,4,5,7,8,9]	()

Selection Sort

```
def pqueueSort[A](items: Seq[A], pqueue: PriorityQueue[A]): Seq[A] =
{
    val out = new Array[A](items.size)
    for(item <- items){ pqueue.enqueue(item) }
    i = out.size - 1
    while(!pqueue.isEmpty) { buffer(i) = pqueue.dequeue; i-- }
    return out.toSeq
}
```

What is the complexity?

Selection Sort

```
def pqueueSort[A](items: Seq[A], pqueue: PriorityQueue[A]): Seq[A] =
{
    val out = new Array[A](items.size)
    for(item <- items){ pqueue.enqueue(item) }
    i = out.size - 1
    while(!pqueue.isEmpty) { buffer(i) = pqueue.dequeue; i-- }
    return out.toSeq
}
```

What is the complexity? ***O(n^2)***

Proactive Priority Queue

Base Data Structure: Linked List

`enqueue(v: A): Unit`

Insert ***t*** in reverse sorted order.

`dequeue/head : A`

Refer to the first item in the list.

Proactive Priority Queue

Base Data Structure: Linked List

`enqueue(v: A): Unit`

Insert ***t*** in reverse sorted order. ***O*(*n*)**

`dequeue/head : A`

Refer to the first item in the list.

Proactive Priority Queue

Base Data Structure: Linked List

```
enqueue(v: A): Unit
```

Insert ***t*** in reverse sorted order. ***O*(*n*)**

```
dequeue/head : A
```

Refer to the first item in the list. ***O*(1)**

Insertion Sort

	Seq/Buffer	PQueue
Input	(7,4,8,2,5,3,9)	()

Insertion Sort

	Seq/Buffer	PQueue
Input	(7,4,8,2,5,3,9)	()
Step 1	(4,8,2,5,3,9)	(7)

Insertion Sort

	Seq/Buffer	PQueue
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)

Insertion Sort

	Seq/Buffer	PQueue
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 3	(2,5,3,9)	(8,7,4)

Insertion Sort

	Seq/Buffer	PQueue
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 3	(2,5,3,9)	(8,7,4)
Step 4	(5,3,9)	(8,7,4.2)

Insertion Sort

	Seq/Buffer	PQueue
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 3	(2,5,3,9)	(8,7,4)
Step 4	(5,3,9)	(8,7,4.2)
	…	…
Step *n*	[_,_,_,_,_,_,_]	(9,8,7,5,4,3,2)

Insertion Sort

	Seq/Buffer	PQueue
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 3	(2,5,3,9)	(8,7,4)
Step 4	(5,3,9)	(8,7,4.2)
	…	…
Step *n*	[_,_,_,_,_,_,_]	(9,8,7,5,4,3,2)
Step *n* + 2	[_,_,_,_,_,_,9]	(8,7,5,4,3,2)

Insertion Sort

	Seq/Buffer	**PQueue**
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 3	(2,5,3,9)	(8,7,4)
Step 4	(5,3,9)	(8,7,4.2)
	…	…
Step *n*	[_,_,_,_,_,_,_]	(9,8,7,5,4,3,2)
Step *n* + *2*	[_,_,_,_,_,_,9]	(8,7,5,4,3,2)
Step *n* + *3*	[_,_,_,_,_,8,9]	(7,5,4,3,2)

Insertion Sort

	Seq/Buffer	PQueue
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 3	(2,5,3,9)	(8,7,4)
Step 4	(5,3,9)	(8,7,4.2)
	…	…
Step *n*	[_,_,_,_,_,_,_]	(9,8,7,5,4,3,2)
Step *n + 2*	[_,_,_,_,_,_,9]	(8,7,5,4,3,2)
Step *n + 3*	[_,_,_,_,_,8,9]	(7,5,4,3,2)
	…	…
Step 2*n*	[2,3,4,5,7,8,9]	()

Selection Sort

```
def pqueueSort[A](items: Seq[A], pqueue: PriorityQueue[A]): Seq[A] =
{
    val out = new Array[A](items.size)
    for(item <- items){ pqueue.enqueue(item) }
    i = out.size - 1
    while(!pqueue.isEmpty) { buffer(i) = pqueue.dequeue; i-- }
    return out.toSeq
}
```

What is the complexity?

Selection Sort

```
def pqueueSort[A](items: Seq[A], pqueue: PriorityQueue[A]): Seq[A] =
{
    val out = new Array[A](items.size)
    for(item <- items){ pqueue.enqueue(item) }
    i = out.size - 1
    while(!pqueue.isEmpty) { buffer(i) = pqueue.dequeue; i-- }
    return out.toSeq
}
```

What is the complexity? ***O(n²)*** → *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?