Priority Queues

- The priority queue ADT
- Implementing a priority queue with a sequence
- Elementary sorting using a Priority Queue
- Issues in sorting

Priority Queues

- A set of elements each with a given priority

- One can insert in any order

- Removal is performed following the priority order
(the element with highest priority is removed first)

- The elements are stored according to their priorities, and not to their position (like it was for queues, sequences, etc.)

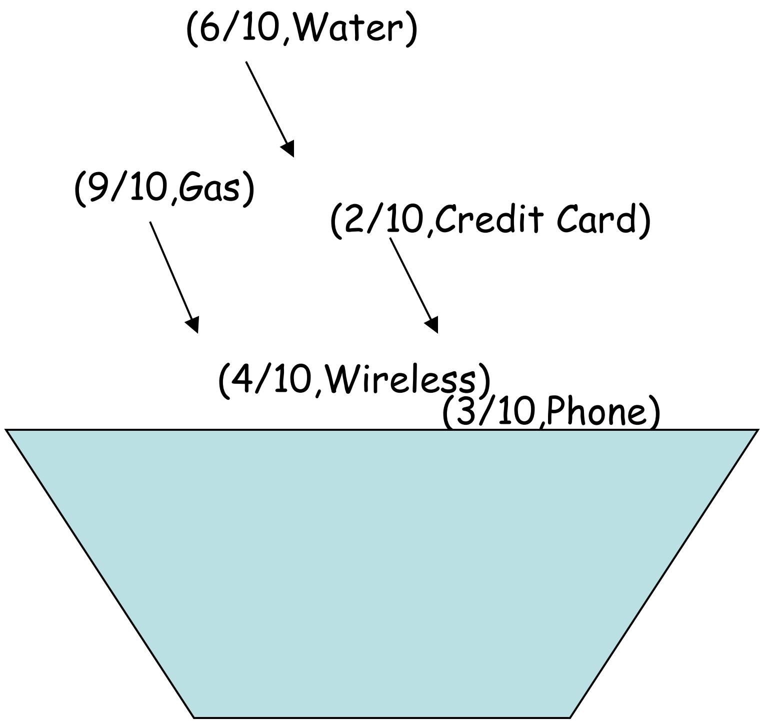
Priority Queue

Queue where we can insert in any order. When we remove an element from the queue, it is always the one with the highest priority.

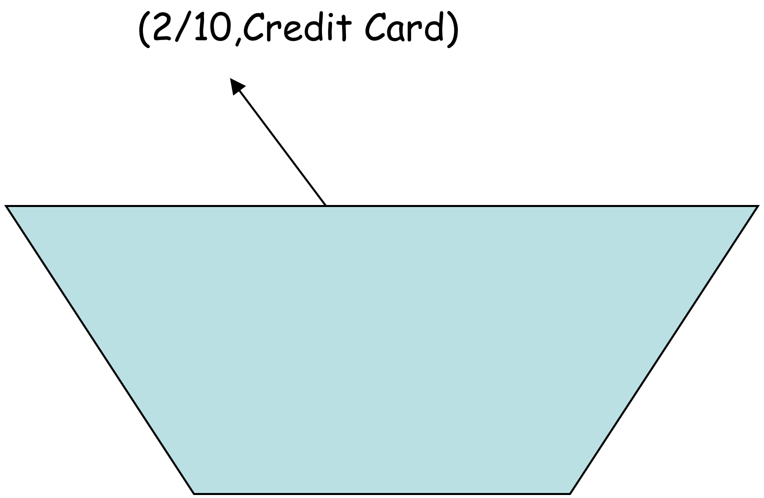
Priority example:

- Deadline to pay a bill
- Deadline to hand in your homework
- A student's mark

insert(key,element)



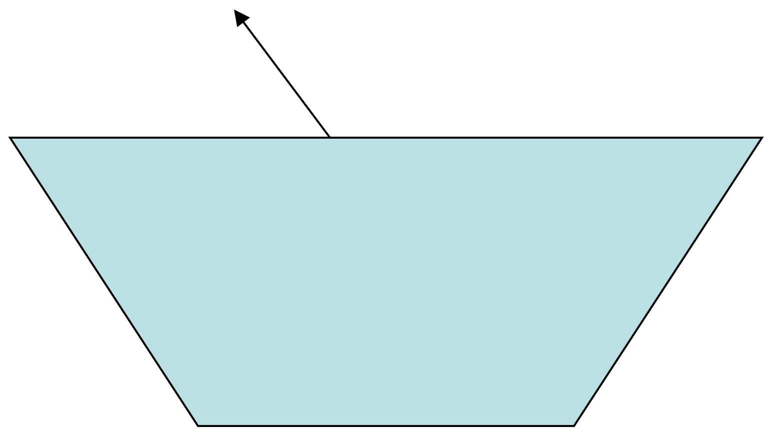
remove



remove

(2/10,Credit Card)

(3/10,Phone)



The Priority Queue ADT

A priority queue stores a collection of entries

Each **entry** is a pair (key, value)

or

(key, element)

Keys in a priority queue can be arbitrary objects on which a total order is defined

Two distinct entries in a priority queue can have the same key

Priority Queue ADT

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, v)
    inserts an entry with key k and value v
  - removeMin()
    removes and returns the entry with smallest key, or null if the the priority queue is empty
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Example

- A sequence of priority queue methods:

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	{ }

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key

- Mathematical concept of total order relation $\leq$
  - Comparability property: either $x \leq y$ or $y \leq x$
  - Antisymmetric property: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

Total ordering examples

- $\leq$ for numbers is a total ordering
- $\geq$ for numbers is also a total ordering
- Alphabetical order:

we define $a \leq b$ if 'a' is before 'b' in alphabetical order

- Reverse alphabetical order.
- Orders of pairs:

We can order the co-ordinate pairs

$p=(x_1, y_1)$ and $q=(x_2,y_2)$ by

1. $p \leq q$ if $x1 \leq x2$, and
2. $p \leq q$ if $x1 = x2$ and $y_1 \leq y_2$

Entry ADT

- An entry in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - getKey: returns the key for this entry
  - getValue: returns the value associated with this entry

- As a Java interface:

```
/**
 * Interface for a key-value
 * pair entry
 **/
public interface Entry<K,V> {
    K getKey();
    V getValue();
}
```

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- Primary method of the Comparator ADT
- compare(x, y): returns an integer i such that
  - i < 0 if a < b,
  - i = 0 if a = b
  - i > 0 if a > b
  - An error occurs if a and b cannot be compared.

Example Comparator

- Lexicographic comparison of 2-D points:

```
/** Comparator for 2D points under the
    standard lexicographic order. */
public class Lexicographic implements
    Comparator {
  int xa, ya, xb, yb;
  public int compare(Object a, Object b)
    throws ClassCastException {
      xa = ((Point2D) a).getX();
      ya = ((Point2D) a).getY();
      xb = ((Point2D) b).getX();
      yb = ((Point2D) b).getY();
      if (xa != xb)
            return (xb - xa);
      else
            return (yb - ya);
  }
}
```

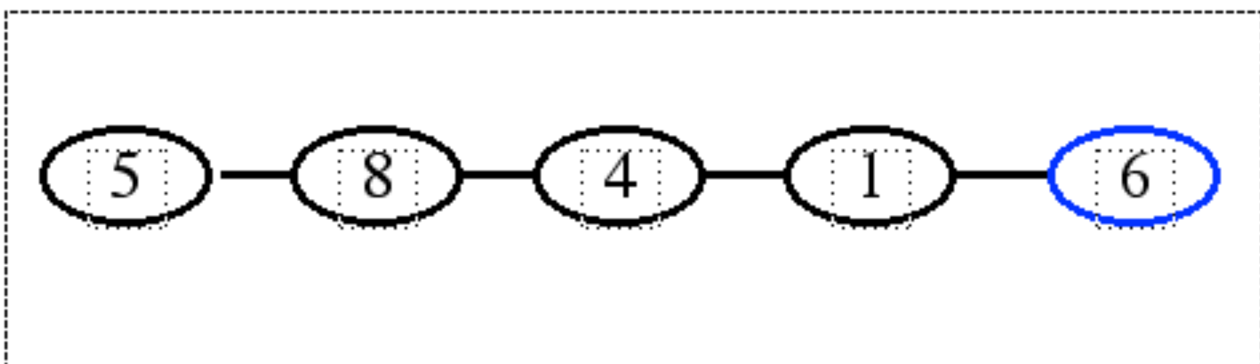
- Point objects:

```
/** Class representing a point in the
    plane with integer coordinates */
public class Point2D {
  protected int xc, yc; // coordinates
  public Point2D(int x, int y) {
    xc = x;
    yc = y;
  }
  public int getX() {
        return xc;
  }
  public int getY() {
        return yc;
  }
}
```

Implementation with an Unsorted Sequence

- The elements of S are a composition of two elements, k, the key, and e, the element.
- insert() = insertLast() in the sequence.

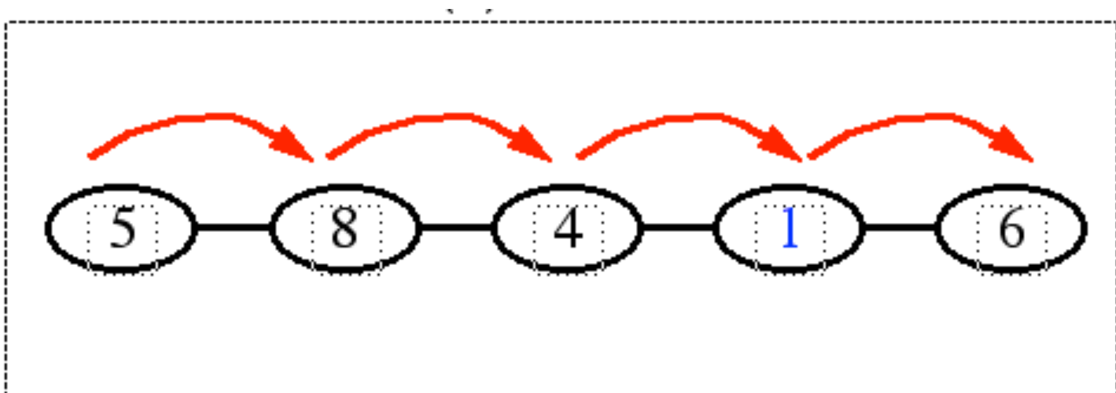
•O(1) time.



Implementation with an Unsorted Sequence (contd.)

- The sequence is not ordered .

For min() and removeMin(),
we need to look at all the elements of S.



O(n) time.

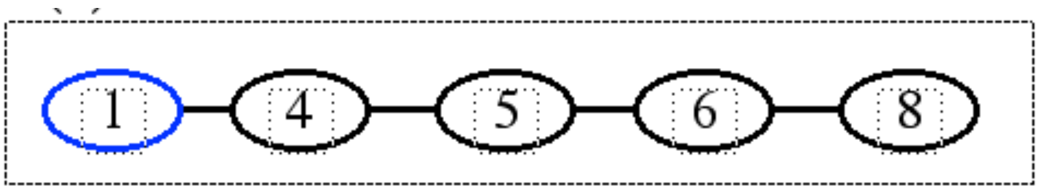
Performance summary

insert()	**O(1)**
min()	**O(n)**
removeMin()	**O(n)**

Implementation with Sorted Sequence

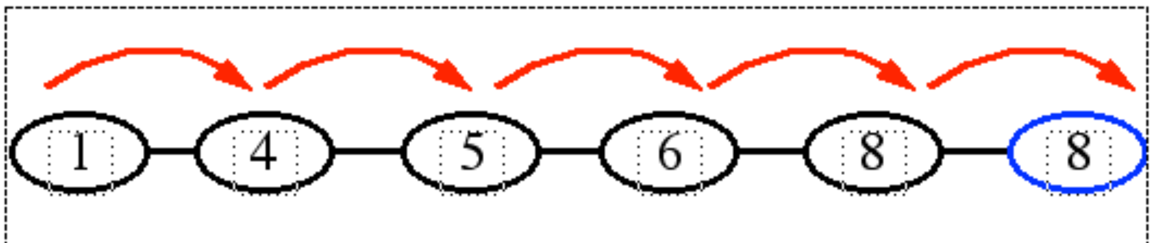
- Use a Sequence S, sorted by increasing keys
- min() and removeMin() take

•O(1) time



Implementation with Sorted Sequence

- However, to implement insertItem(), we must now scan through the entire sequence in the worst case. Thus insertItem() runs in
  - O(n) time



Performance summary

insert()	**O(n)**
min()	**O(1)**
removeMin()	**O(1)**

An observation...

With an unsorted sequence...

removeMin() always takes O(n)

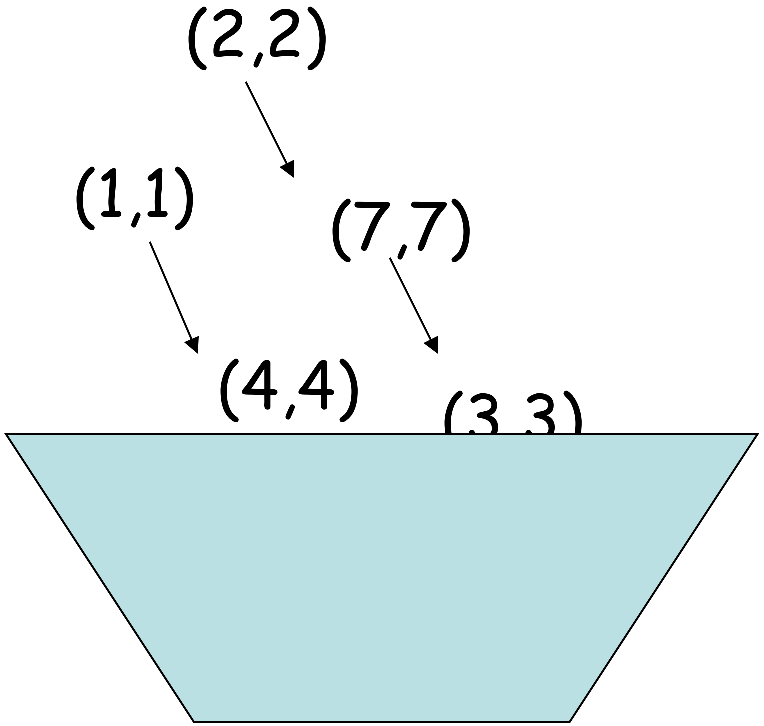
But with a sorted sequence...

insert() takes at most O(n)

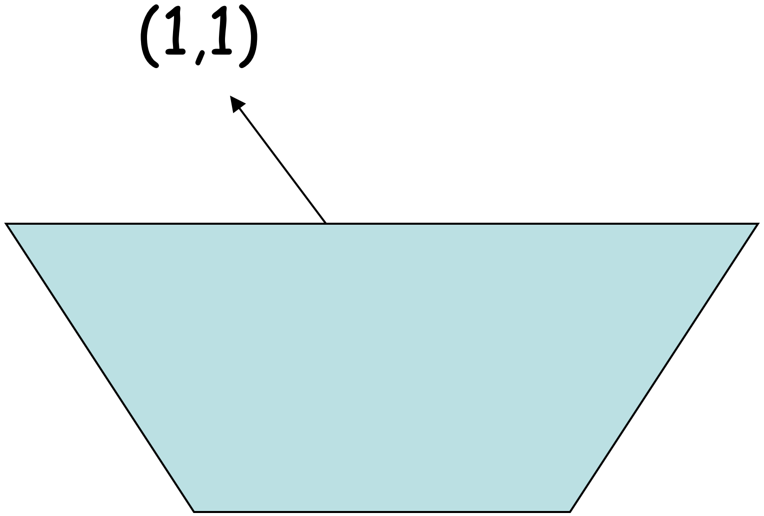
An Application: Sorting

- A Priority Queue P can be used for sorting a sequence S by:
  - *inserting* the elements of S into P with a series of insert(e, e) operations
  - *removing* the elements from P and putting them back into S with a series of removeMin() operations

insert .....



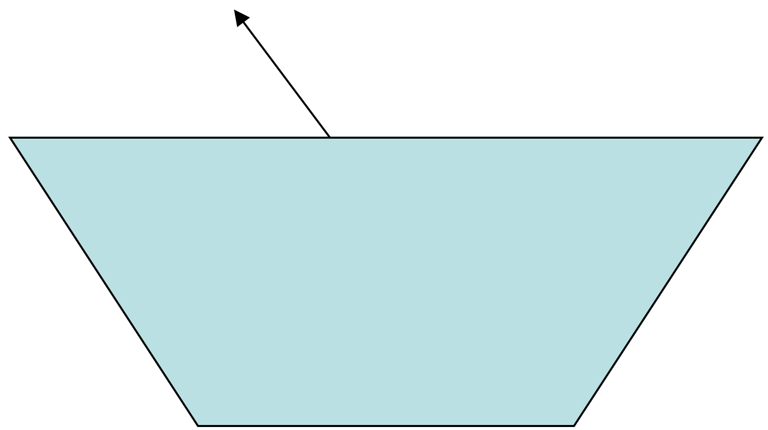
remove



remove

(1,1)

(2,2)



Priority Queue Sorting

- We can use a priority queue to sort a list of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

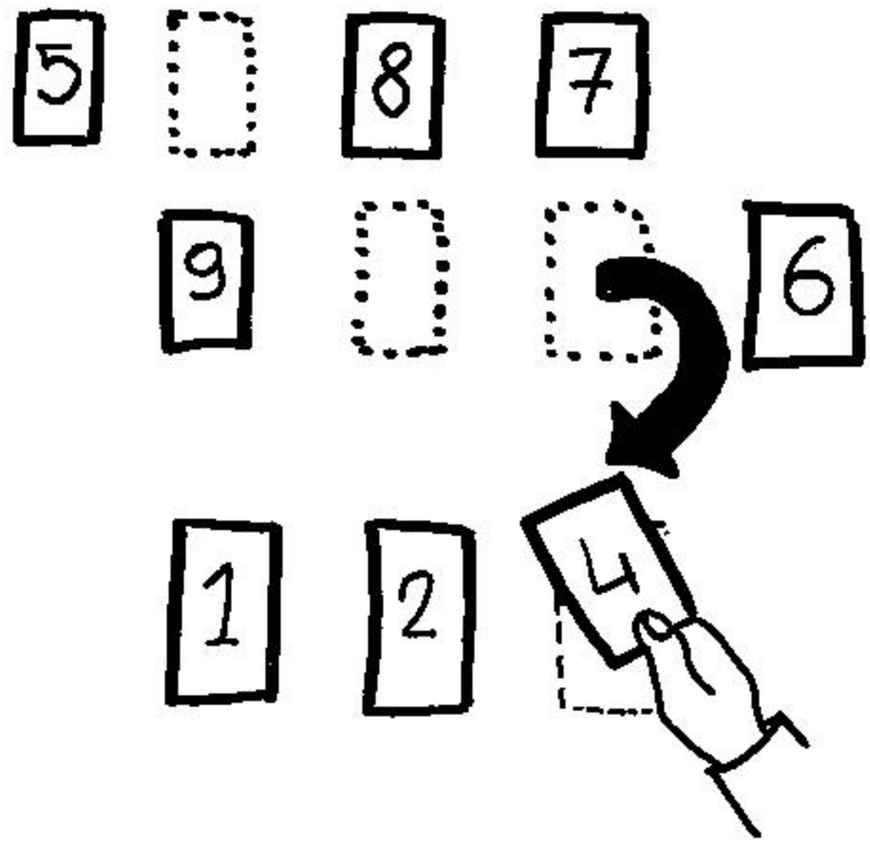
Algorithm ***PQ-Sort***(***S***, ***C***)
  Input list ***S***, comparator ***C*** for the elements of ***S***
  Output list ***S*** sorted in increasing order according to ***C***
  P ← priority queue with
    comparator ***C***
  while ¬***S.isEmpty*** ()
    e ← ***S.remove***(***S.first*** ())
    P.insert (***e***, ∅)
  while ¬***P.isEmpty***()
    e ← ***P.removeMin***().***getKey***()
    S.addLast(***e***)

Selection Sort

- Variation of PriorityQueueSort that uses an unsorted sequence to implement the priority queue P.
  - Phase 1, the insertion of an item into P takes O(1) time
  - Phase 2, removing (selecting) an item from P takes time proportional to the current number of elements in P

Selection Sort

Insert in no specific order



Select in order

Selection Sort Example

		Sequence S		Priority Queue P
Input:		(7,4,8,2,5,3,9)		()
Phase 1				
	(a)	(4,8,2,5,3,9)	insert() == insertLast()	(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)	removeMin()	(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)		()

unsorted sequence

Selection Sort (cont.)

Running time of Selection-sort:

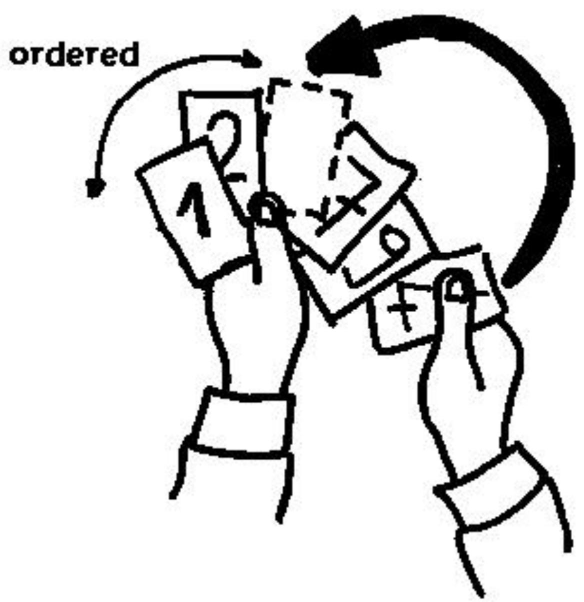
--- Inserting the elements into the priority queue with n insertItem operations takes $O(n)$ time

--- Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$1 + 2 + \ldots + n$

Selection-sort runs in $O(n^2)$ time

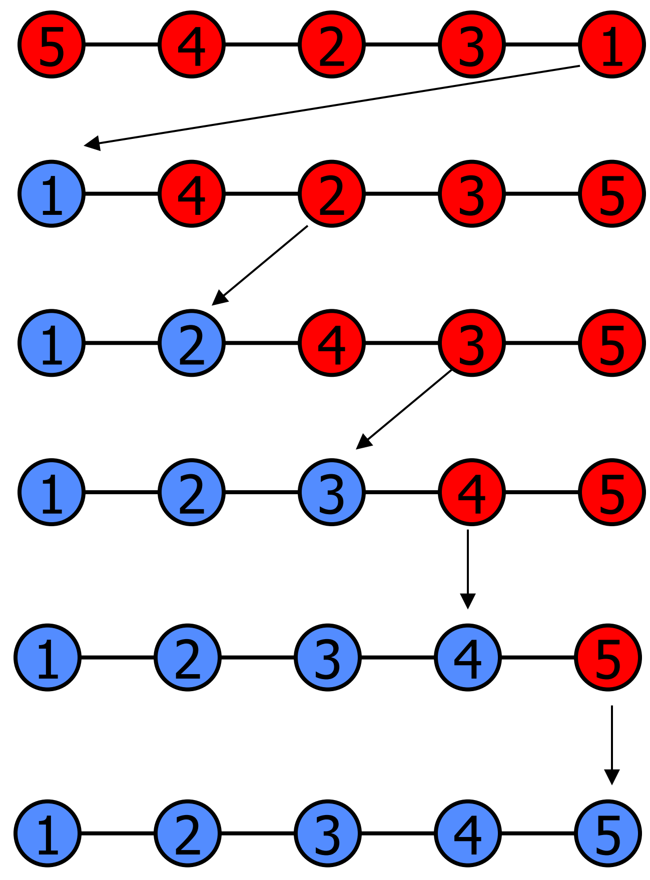
Selection Sort In Place



- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place select-sort
  - keep first part of the sequence ordered, select min, put it at its place

Selection Sort In Place

Sorted part
Non-sorted part



Insertion Sort

- PriorityQueueSort implementing the priority queue with a ***sorted sequence***

Insert in order

Select

Insertion-Sort 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)	insert()	(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)	removeMin()	(4,5,7,8,9)
	(c)	(2,3,4)		(5,7,8,9)
	(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)		()

sorted

sequence

Insertion Sort(cont.)

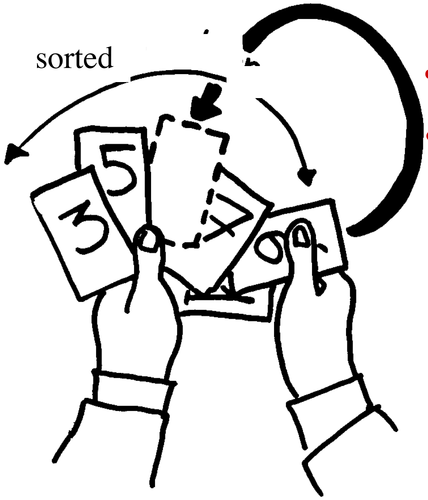
Running time of Insertion-sort:

--- Inserting the elements into the priority queue with n insertItem operations takes time proportional to

$1 + 2 + \ldots + n$

--- Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time

Insertion-sort runs in $O(n^2)$ time



In-place Insertion-sort

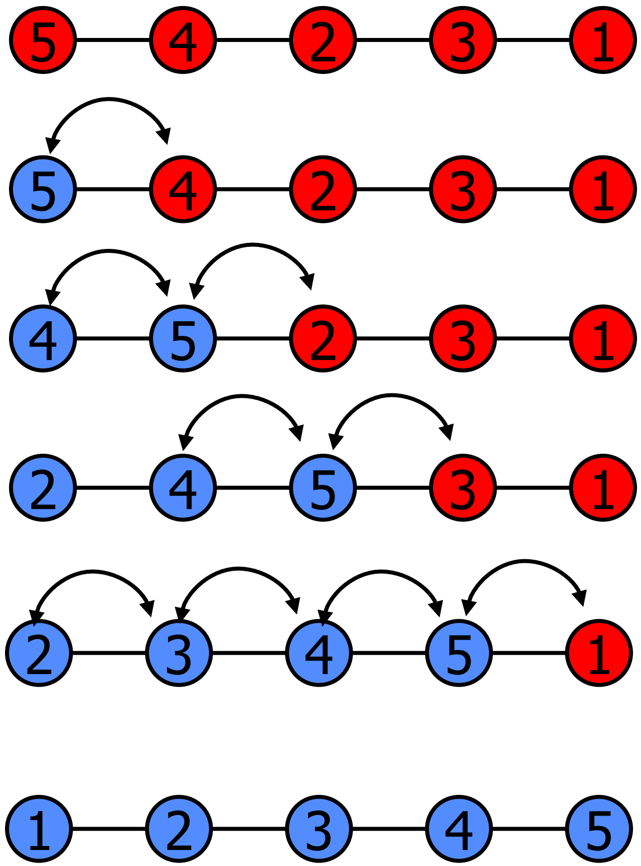
- No external data structure

- A portion of the input sequence itself serves as the priority queue

- We keep sorted the initial portion of the sequence
- We can use swaps instead of modifying the sequence

In-place Insertion-sort

Sorted part
Unsorted part



Comparisons

- Selection sort always performs $O(n^2)$ operations regardless of the input

  removeMin() is always executed in time $O(n)$

- The execution time of Insertion Sort depends on the type of input

  insertItem() is executed in the worst case in $O(n)$

APPENDIX: Reference Java Code

- Priority queues ADT using sorted list data structure
- Priority queues ADT using unsorted list data structure

Unsorted List Implementation

```
/** An implementation of a priority queue with an unsorted list. */
public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public UnsortedPriorityQueue() { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }

  /** Returns the Position of an entry having minimal key. */
  private Position<Entry<K,V>> findMin() {    // only called when nonempty
    Position<Entry<K,V>> small = list.first();
    for (Position<Entry<K,V>> walk : list.positions())
      if (compare(walk.getElement(), small.getElement()) < 0)
        small = walk;          // found an even smaller key
    return small;
  }

```

Unsorted List Implementation, 2

```
  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);      // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    list.addLast(newest);
    return newest;
  }

  /** Returns (but does not remove) an entry with minimal key. */
  public Entry<K,V> min() {
    if (list.isEmpty()) return null;
    return findMin().getElement();
  }

  /** Removes and returns an entry with minimal key. */
  public Entry<K,V> removeMin() {
    if (list.isEmpty()) return null;
    return list.remove(findMin());
  }

  /** Returns the number of items in the priority queue. */
  public int size() { return list.size(); }
}
```

Sorted List Implementation

```
/** An implementation of a priority queue with a sorted list. */
public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public SortedPriorityQueue() { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public SortedPriorityQueue(Comparator<K> comp) { super(comp); }

  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);      // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    Position<Entry<K,V>> walk = list.last();
    // walk backward, looking for smaller key
    while (walk != null && compare(newest, walk.getElement()) < 0)
      walk = list.before(walk);
    if (walk == null)
      list.addFirst(newest);                        // new key is smallest
    else
      list.addAfter(walk, newest);                  // newest goes after walk
    return newest;
  }
```

Sorted List Implementation, 2

```
  /** Returns (but does not remove) an entry with minimal key. */
  public Entry<K,V> min() {
    if (list.isEmpty()) return null;
    return list.first().getElement();
  }

  /** Removes and returns an entry with minimal key. */
  public Entry<K,V> removeMin() {
    if (list.isEmpty()) return null;
    return list.remove(list.first());
  }

  /** Returns the number of items in the priority queue. */
  public int size() { return list.size(); }
}
```