Priority Queues
A priority queue stores a collection of entries.

Each entry is a pair (key, value).

Main methods of the Priority Queue ADT:
- `insert(k, x)`: inserts an entry with key k and value x.
- `removeMin()`: removes and returns the entry with smallest key.

Additional methods:
- `min()`: returns, but does not remove, an entry with smallest key.
- `size()`, `isEmpty()`.

Applications:
- Standby flyers
- Auctions
- Stock market
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 \( \leq \):
  - Reflexive property: \( x \leq x \)
  - Antisymmetric property: \( x \leq y \land y \leq x \Rightarrow x = y \)
  - Transitive property: \( x \leq y \land y \leq z \Rightarrow x \leq z \)
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:
```java
/**
 * Interface for a key-value pair entry
 */
public interface Entry<K,V>
{
    public K getKey();
    public 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;
    }
  }

© 2010 Goodrich, Tamassia
Priority Queue Sorting

- We can use a priority queue to sort a set 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 sequence $S$, comparator $C$ for the elements of $S$
Output sequence $S$ sorted in increasing order according to $C$

$P \leftarrow$ priority queue with comparator $C$
while $\neg S.isEmpty()$
    $e \leftarrow S.removeFirst()$
    $P.insert(e, \emptyset)$
while $\neg P.isEmpty()$
    $e \leftarrow P.removeMin().getKey()$
    $S.addLast(e)$
```

© 2010 Goodrich, Tamassia
Sequence-based Priority Queue

- Implementation with an unsorted list
  
  4 5 2 3 1

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list
  
  1 2 3 4 5

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning
Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence.

- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with \( n \) insert operations takes \( O(n) \) time.
  2. 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 Example

<table>
<thead>
<tr>
<th>Input:</th>
<th>Sequence S</th>
<th>Priority Queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7,4,8,2,5,3,9)</td>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>

### Phase 1

<table>
<thead>
<tr>
<th></th>
<th>Sequence S</th>
<th>Priority Queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(4,8,2,5,3,9)</td>
<td>(7)</td>
</tr>
<tr>
<td>(b)</td>
<td>(8,2,5,3,9)</td>
<td>(7,4)</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td></td>
</tr>
<tr>
<td>(g)</td>
<td>()</td>
<td>(7,4,8,2,5,3,9)</td>
</tr>
</tbody>
</table>

### Phase 2

<table>
<thead>
<tr>
<th></th>
<th>Sequence S</th>
<th>Priority Queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(2)</td>
<td>(7,4,8,5,3,9)</td>
</tr>
<tr>
<td>(b)</td>
<td>(2,3)</td>
<td>(7,4,8,5,9)</td>
</tr>
<tr>
<td>(c)</td>
<td>(2,3,4)</td>
<td>(7,8,5,9)</td>
</tr>
<tr>
<td>(d)</td>
<td>(2,3,4,5)</td>
<td>(7,8,9)</td>
</tr>
<tr>
<td>(e)</td>
<td>(2,3,4,5,7)</td>
<td>(8,9)</td>
</tr>
<tr>
<td>(f)</td>
<td>(2,3,4,5,7,8)</td>
<td>(9)</td>
</tr>
<tr>
<td>(g)</td>
<td>(2,3,4,5,7,8,9)</td>
<td>(9)</td>
</tr>
</tbody>
</table>
Insertion-Sort

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

- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with $n$ insert operations takes time proportional to $1 + 2 + \ldots + n$.
  2. 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.
## Insertion-Sort Example

<table>
<thead>
<tr>
<th>Input: (7,4,8,2,5,3,9)</th>
<th>Sequence S</th>
<th>Priority queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Phase 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) (4,8,2,5,3,9)</td>
<td></td>
<td>(7)</td>
</tr>
<tr>
<td>(b) (8,2,5,3,9)</td>
<td></td>
<td>(4,7)</td>
</tr>
<tr>
<td>(c) (2,5,3,9)</td>
<td></td>
<td>(4,7,8)</td>
</tr>
<tr>
<td>(d) (5,3,9)</td>
<td></td>
<td>(2,4,7,8)</td>
</tr>
<tr>
<td>(e) (3,9)</td>
<td></td>
<td>(2,4,5,7,8)</td>
</tr>
<tr>
<td>(f) (9)</td>
<td></td>
<td>(2,3,4,5,7,8)</td>
</tr>
<tr>
<td>(g) ()</td>
<td></td>
<td>(2,3,4,5,7,8,9)</td>
</tr>
</tbody>
</table>

| **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)    |            | ()               |

© 2010 Goodrich, Tamassia
In-place Insertion-Sort

- 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 insertion-sort:
  - We keep sorted the initial portion of the sequence.
  - We can use swaps instead of modifying the sequence.