Priority Queues

Priority Queue ADT (§ 7.1.3)
- 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 (§ 7.1.1)
- 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 (§ 7.1.2)
- 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:
  - key(): returns the key for this entry.
  - value(): returns the value associated with this entry.
- As a Java interface:
  ```java
  /**
   * Interface for a key-value pair.
   * @interface Entry
   */
  public interface Entry {
      /**
       * Interface for a key-value entry.
       * @interface Entry
       */
      public Object key();
      public Object value();
  }
```
Comparator ADT (§ 7.1.2)

- 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.

The 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`, and `i > 0` if `a > b`; an error occurs if `a` and `b` cannot be compared.

Example Comparator

Lexicographic comparison of 2-D points:

```java
/** 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);
    }
}
```

Priority Queue Sorting (§ 7.1.4)

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 ← priority queue with comparator `C`
while ¬S.isEmpty ()
e ← S.removeFirst ()
P.insert (e, 0)
while ¬P.isEmpty ()
e ← P.removeMin().key()  
S.insertLast(e)
```

Sequence-based Priority Queue

Implementation with an unsorted list

- **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

- **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.

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.
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.