Priority Queues

Outline and Reading

- PriorityQueue ADT (§7.1)
- Total order relation (§7.1.1)
- Comparator ADT (§7.1.4)
- Sorting with a priority queue (§7.1.2)
- Selection-sort (§7.2.3)
- Insertion-sort (§7.2.3)

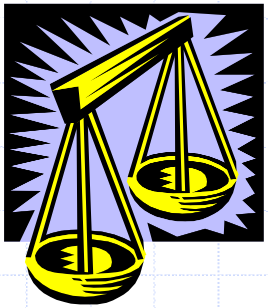
Priority Queue ADT

- A priority queue stores a collection of items
- An item is a pair (key, element)
- Main methods of the Priority Queue ADT
  - insertItem(k, o) inserts an item with key k and element o
  - removeMin() removes the item with the smallest key
- Additional methods
  - minKey(k, o) returns, but does not remove, the smallest key of an item
  - minElement() returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Total Order Relation

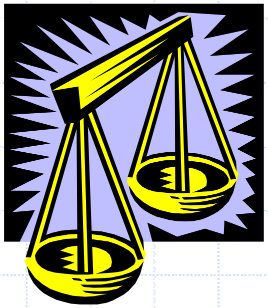
- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct items in a priority queue can have the same key
- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Comparator ADT



- A *comparator* encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses a comparator as a template argument, to define the comparison function (<,=,>)
- The comparator is external to the keys being compared. Thus, the same objects can be sorted in different ways by using different comparators.
- When the priority queue needs to compare two keys, it uses its comparator

Using Comparators in C++



- A comparator class overloads the "()" operator with a comparison function.
- Example: Compare two points in the plane lexicographically.

```
class LexCompare {
public:
    int operator()(Point a, Point b) {
        if       (a.x < b.x) return –1
        else if (a.x > b.x) return +1
        else if (a.y < b.y) return  –1
        else if (a.y > b.y) return +1
        else return 0;
    }
};
```

- To use the comparator, define an object of this type, and invoke it using its "()" operator:
- Example of usage:

```
Point p(2.3, 4.5);
Point q(1.7, 7.3);
LexCompare lexCompare;

if (lexCompare(p, q) < 0)
    cout << "p less than q";
else if (lexCompare(p, q) == 0)
    cout << "p equals q";
else if (lexCompare(p, q) > 0)
    cout << "p greater than q";
```

Sequence-based Priority Queue

- Implementation with an unsorted sequence
  - Store the items of the priority queue in a list-based sequence, in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin, minKey and minElement take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted sequence
  - Store the items of the priority queue in a sequence, sorted by key
- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey and minElement take $O(1)$ time since the smallest key is at the beginning of the sequence

Sorting with a Priority Queue

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insertItem(e, e) 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.remove* (*S. first* ())
    P.insertItem(*e*, *e*)
  while !*P.isEmpty*()
    e ← *P.minElement*()
    P.removeMin()
    S.insertLast(*e*)

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 insertItem 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 insertItem 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 swapElements instead of modifying the sequence