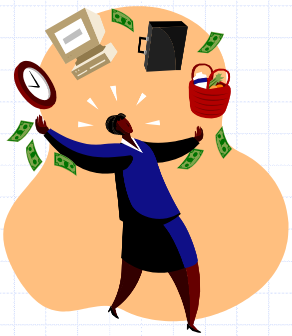
Priority Queues



Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - insert(e) inserts an entry e
  - removeMin() removes the entry with smallest key
- Additional methods
  - min() returns, but does not remove, an entry with smallest key
  - size(), empty()
- 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: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Comparator ADT

- Implements the boolean function isLess(p,q), which tests whether p < q
- Can derive other relations from this:
  - (p == q) is equivalent to
  - (!isLess(p, q) && !isLess(q, p))
- Can implement in C++ by overloading "()"

Two ways to compare 2D points:

```
class LeftRight { // left-right comparator
public:
    bool operator()(const Point2D& p,
         const Point2D& q) const
    { return p.getX() < q.getX(); }
};
class BottomTop { // bottom-top
public:
    bool operator()(const Point2D& p,
    const Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

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 $\neg S.empty$ ()
    $e \leftarrow S.front()$; $S.eraseFront()$
    $P.insert$ (e, $\varnothing$)
  while $\neg P.empty()$
    $e \leftarrow P.removeMin()$
    $S.insertBack(e)$

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

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)	(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)	(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)	()

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

		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)	(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)	(4,5,7,8,9)
	..	..	..
	(g)	(2,3,4,5,7,8,9)	()

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