Priority Queues

Priority Queue ADT

- A priority queue stores a collection of items
- Each item is a pair (key, value)
- Main methods of the Priority Queue ADT
  - add (k, x) inserts an item with key k and value x
  - remove_min() removes and returns the item with smallest key
- Additional methods
  - min() returns, but does not remove, an item with smallest key
  - len(P), is_empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Priority Queue Example

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min()	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min()	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min()	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min()	(7,D)	{(9,C)}
P.remove_min()	(9,C)	{ }
P.is_empty()	True	{ }
P.remove_min()	"error"	{ }

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: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Composition Design Pattern

- An item in a priority queue is simply a key-value pair
- Priority queues store items to allow for efficient insertion and removal based on keys

```
class PriorityQueueBase:
  """Abstract base class for a priority queue."""

  class _Item:
    """Lightweight composite to store priority queue items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __lt__(self, other):
      return self._key < other._key        # compare items based on their keys

  def is_empty(self):                      # concrete method assuming abstract len
    """Return True if the priority queue is empty."""
    return len(self) == 0
```

Sequence-based Priority Queue

- Implementation with an unsorted list

  4 — 5 — 2 — 3 — 1

- Performance:
  - add takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - Remove_min 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:
  - add takes $O(n)$ time since we have to find the place where to insert the item
  - remove_min and min take $O(1)$ time, since the smallest key is at the beginning

Unsorted List Implementation

```python
class UnsortedPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with an unsorted list."""

  def _find_min(self):                    # nonpublic utility
    """Return Position of item with minimum key."""
    if self.is_empty():                   # is_empty inherited from base class
      raise Empty('Priority queue is empty')
    small = self._data.first()
    walk = self._data.after(small)
    while walk is not None:
      if walk.element() < small.element():
        small = walk
      walk = self._data.after(walk)
    return small

  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = PositionalList()

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair."""
    self._data.add_last(self._Item(key, value))

  def min(self):
    """Return but do not remove (k,v) tuple with minimum key."""
    p = self._find_min()
    item = p.element()
    return (item._key, item._value)

  def remove_min(self):
    """Remove and return (k,v) tuple with minimum key."""
    p = self._find_min()
    item = self._data.delete(p)
    return (item._key, item._value)
```

Sorted List Implementation

```
class SortedPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with a sorted list."""

  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = PositionalList()

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair."""
    newest = self._Item(key, value)              # make new item instance
    walk = self._data.last()       # walk backward looking for smaller key
    while walk is not None and newest < walk.element():
      walk = self._data.before(walk)
    if walk is None:
      self._data.add_first(newest)               # new key is smallest
    else:
      self._data.add_after(walk, newest)         # newest goes after walk

  def min(self):
    """Return but do not remove (k,v) tuple with minimum key."""
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    p = self._data.first()
    item = p.element()
    return (item._key, item._value)

  def remove_min(self):
    """Remove and return (k,v) tuple with minimum key."""
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    item = self._data.delete(self._data.first())
    return (item._key, item._value)
```

Runtime of implementing PQ using Sorted and Unsorted List

Operation	Unsorted List	Sorted List
len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
remove_min	$O(n)$	$O(1)$

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 add operations
  2. Remove the elements in sorted order with a series of remove_min 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.is_empty* ()
  $e \leftarrow$ *S.remove_first* ()
  P.add (e, $\emptyset$)
 while $\neg$*P.is_empty*()
  $e \leftarrow$ *P.removeMin*().*key*()
  S.add_last(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 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