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 ADT…

```
class PriorityQueue:
    #Abstract and basic class for PriorityQueue
    #Nested class for the items
    class _Item:
        #efficient composite to store items
        __slots__ = '_key', '_value'

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

        def __lt__( self, other ):
            return self._key < other._key

        def __gt__( self, other ):
            return self._key > other._key

        def __str__( self ):
            return "(" + str( self._key ) + "," + str( self._value ) + ")"
```

Priority Queue ADT

```
def __init__( self ):
    pass

def __len__( self ):
    pass

def __str__( self ):
    if self.is_empty():
        return "[]"
    pp = "["
    for item in self:
        pp += str( item )
    pp += "]"
    return pp
```

```
def is_empty( self ):
    return len( self ) == 0

def min( self ):
    pass

def add( self, k, x ):
    pass

def remove_min( self ):
    pass
```

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

- 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

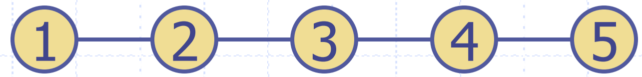
- Mathematical concept of total order relation
  In X under ≤, then properties hold for all x, y and z in X:
  - Antisymmetric property: $x \le y$ and $y \le x \Rightarrow x = y$
  - Transitive property: $x \le y$ and $y \le z \Rightarrow x \le z$
  - Totality property: $x \le y$ or $y \le x$

Sequence-based Priority Queue

- Implementation with an unsorted list

- 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



- 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

```
from PriorityQueue import PriorityQueue
#UnsortedListPriorityQueue
class UnsortedListPriorityQueue( PriorityQueue ):

    def __init__( self ):
        self._Q = []

    def __len__( self ):
        return len( self._Q )

    def __getitem__( self, i ):
        return self._Q[i]

    def is_empty( self ):
        return len( self ) == 0
```

Unsorted List...

```
def min( self ):
    if self.is_empty():
        return False
    #search the min in O(n) on average
    the_min = self._Q[0]
    for item in self:
        if item < the_min:
            the_min = item
    #return the min
    return the_min

def add( self, k, x ):
    #in O(1)
    self._Q.append( self._Item( k, x ) )
```

Unsorted List

```
def remove_min( self ):
    if self.is_empty():
          return False
    #search the index of min in O(n) on average
    index_min = 0
    for i in range( 1, len( self ) ):
        if self._Q[i] < self._Q[index_min]:
            index_min = i
    the_min = self._Q[index_min]
    #delete the min
    del self._Q[index_min]
    #return the deleted item
    return the_min
```

Sorted List Implementation

```
from PriorityQueue import PriorityQueue

#SortedListPriorityQueue
class SortedListPriorityQueue( PriorityQueue ):

    def __init__( self ):
        self._Q = []

    def __len__( self ):
        return len( self._Q )

    def __getitem__( self, i ):
        return self._Q[i]

    def is_empty( self ):
        return len( self ) == 0
```

Sorted List...

```
def min( self ):
    if self.is_empty():
        return False
    #find min in O(1)
    #the min is in Q[0]
    return self._Q[0]
```

```
def remove_min( self ):
    if self.is_empty():
        return False
    #remove min in O(1)
    #the min is in Q[0]
    the_min = self._Q[0]
    del self._Q[0]
    return the_min
```

Sorted List

```
def add( self, k, x ):
    item = self._Item( k, x )
    if self.is_empty():
        self._Q.append( item )
    else:
        #create the extra space in Q
        self._Q.append( item )
        #search for insertion index
        #in O(n) on average
        i = 0
        while item > self._Q[i]:
            i += 1
        #make space for new item
        #in O(n) on average
        for j in range( len( self ) - 1, i, -1 ):
            self._Q[j] = self._Q[j-1]
            print( "Q[", j, "] = Q[", j-1, "]" )
        #insert item at insertion index
        self._Q[i] = item
    #return new item
    return item
```

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 = priority queue with
    comparator *C*
  While not *S.is_empty* ()
    e = *S.remove_first* ()
    P.add (*e*, *e*)
  While not *P.is_empty*()
    e = *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