CSI2110

Data Structures and Algorithms

Prof. WonSook Lee

Priority Queues

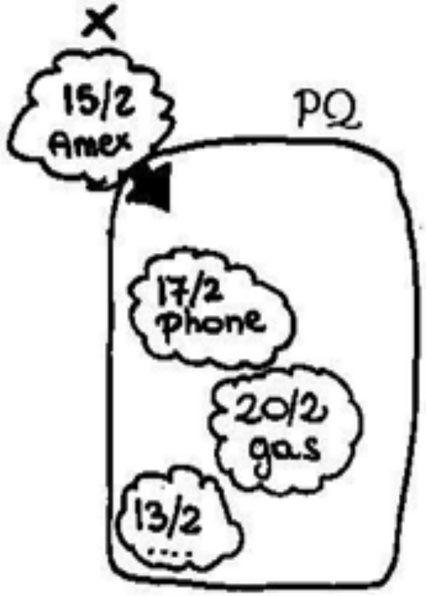
- The priority queue ADT
- Implementing a priority queue with a sequence
- Elementary sorting using a Priority Queue

Priority Queue

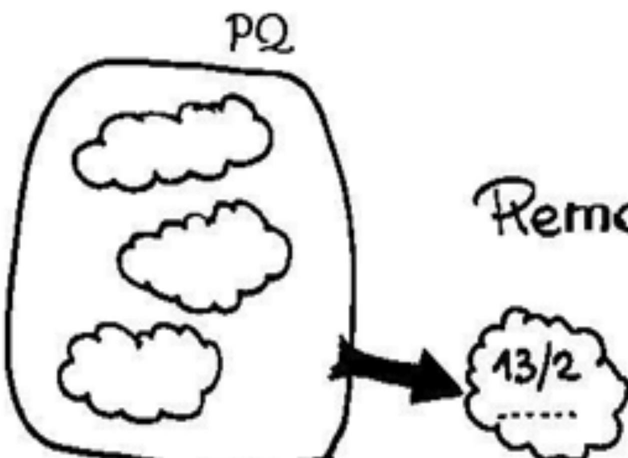
Queue where we can insert in any order. When we remove an element from the queue, it is always the one with the highest priority.

Priority example:

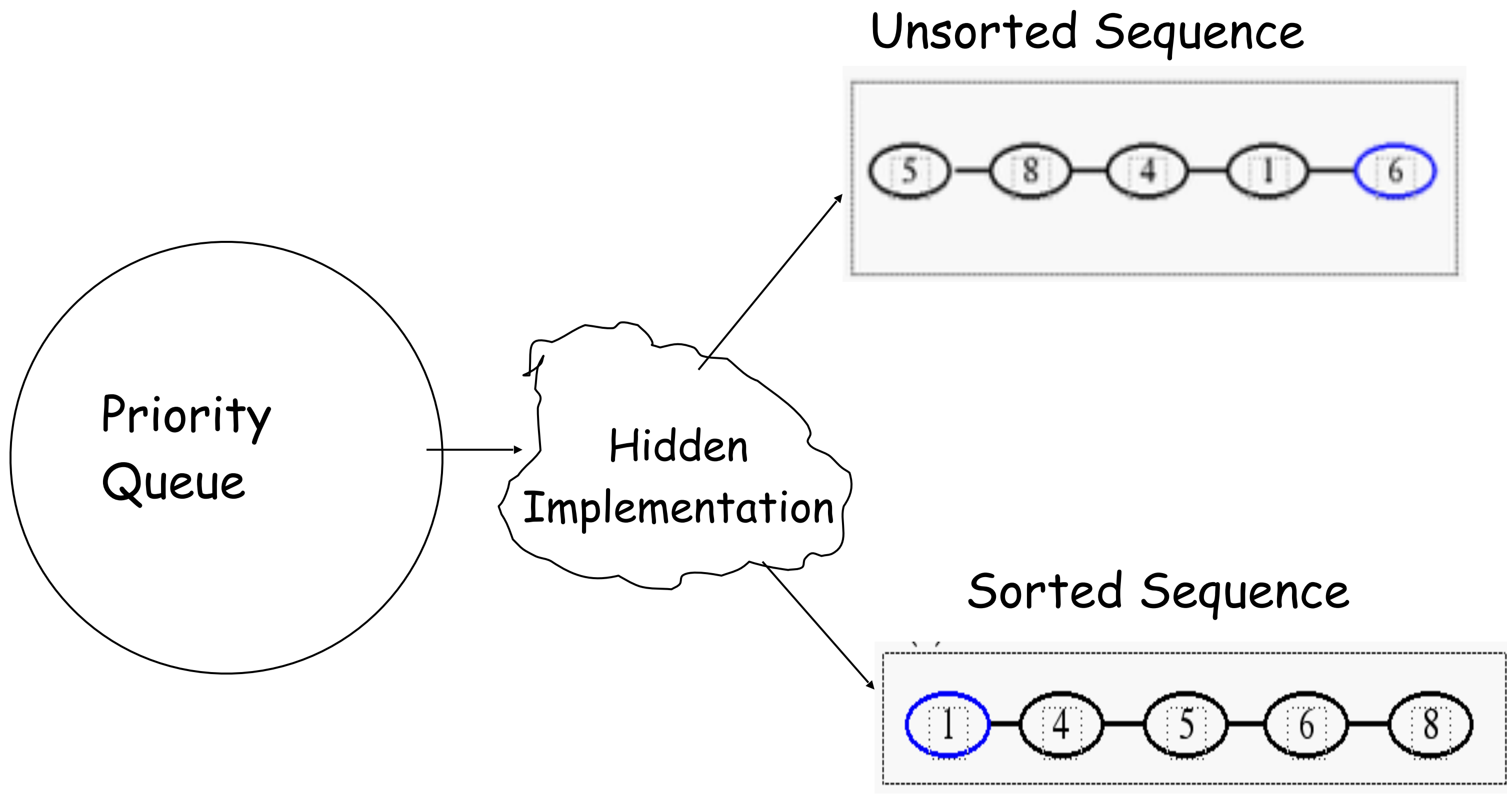
- Deadline to pay a bill
- Deadline to hand in your homework
- A student's mark
- Standby passengers (frequent-flyer status, the fare paid and check-in time, etc)



Insert(x, PQ) ;



Remove(PQ) ;


Unsorted Sequence
5 8 4 1 6
Priority Queue
Hidden Implementation
Sorted Sequence
1 4 5 6 8

Keys and Total Order Relations

- A Priority Queue ranks its elements by **key** with a **total order** relation
- Definition of a "Key" : an object that is assigned to an element as a specific attribute for that element, which can be used to identify, rank or weight that element

- Keys: Every element has its own key

  Keys are not necessarily unique

- Total Order Relation, denoted by $\leq$

  Reflexive: $k \leq k$

  Antisymetric: if $k1 \leq k2$ and $k2 \leq k1$, then $k1 = k2$

  Transitive: if $k1 \leq k2$ and $k2 \leq k3$, then $k1 \leq k3$

- Total Order Relation, denoted by ☆

  Reflexive: k ☆ k

  Antisymetric: if k1 ☆ k2 and k2 ☆ k1, then k1 = k2

  Transitive: if k1 ☆ k2 and k2 ☆ k3, then k1 ☆ k3

Total ordering examples

- $\leq$ is a total ordering
- $\geq$ is also a total ordering
- Alphabetical order: we define a $\leq$ b if ‘a’ is before ‘b’ in alphabetical order
- Reverse alphabetical order

But…

- $<$, $>$ are not total orderings since they are not reflexive

Reflexive: $k \leq k$

Antisymetric: if $k1 \leq k2$ and $k2 \leq k1$, then $k1 = k2$

Transitive: if $k1 \leq k2$ and $k2 \leq k3$, then $k1 \leq k3$

The Priority Queue ADT

- A priority queue P supports the following methods:

-size(): Return the number of elements in P

-isEmpty(): Test whether P is empty

-insertItem(k,e): Insert a new element e with key k into P

-minElement(): Return (but don't remove) an element of P with smallest key; an error occurs if P is empty.

-minKey(): Return the smallest key in P; an error occurs if P is empty

-removeMin(): Remove from P and return an element with the smallest key; an error condition occurs if P is empty.

An Application: Sorting

- A Priority Queue P can be used for sorting a sequence S by:
  - **inserting** the elements of S into P with a series of insertItem(k, e) operations
  - **removing** the elements from P in increasing order and putting them back into S with a series of removeMin() operations

```
Algorithm PriorityQueueSort(S, P):
        Input: A sequence S storing n elements, on which a
                total order relation is defined, and a Priority
            Queue P that compares keys with the same relation
        Output: The Sequence S sorted by the total
                        order relation
        while !S.isEmpty() do
                e ← S.removeFirst()
                P.insertItem(e, e)
        while P is not empty do
                e ← P.removeMin()
                S.insertLast(e)
```

Comparators

- Example : Given keys 4 and 11 we have that $4 \leq 11$ if the keys are integer objects (to be compared in the usual manner), but $11 \leq 4$ if the keys are string objects (to be compared lexicographically)

- How to specify the relation for comparing keys?
  - Shall we implement a different priority queue for each key type we want to use and each possible way of comparing keys of such types?
    - NO. This approach is not very general and it requires a lot of similar code.

Comparators

- The most general and reusable form of a priority queue makes use of **comparator** objects.
- Comparator objects are external to the keys to supply the comparison rules.
- A PQ P is given a comparator when P is constructed. We can update the comparator when it is needed.
- When P needs to compare two keys, it uses the comparator it was given to perform the comparison.
- Thus a priority queue can be general enough to store any object.

Comparators

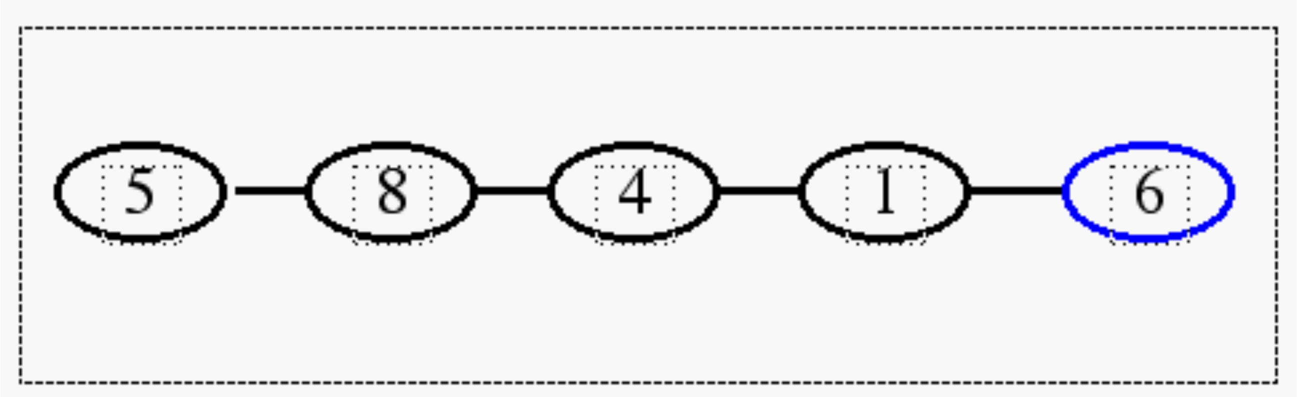
- The comparator ADT includes:
  - isLessThan(a, b)
  - isLessThanOrEqualTo(a,b)
  - isEqualTo(a, b)
  - isGreaterThan(a,b)
  - isGreaterThanOrEqualTo(a,b)
  - isComparable(a)

Implementation!

Implementation with a Sequence for now!

Implementation with an Unsorted Sequence

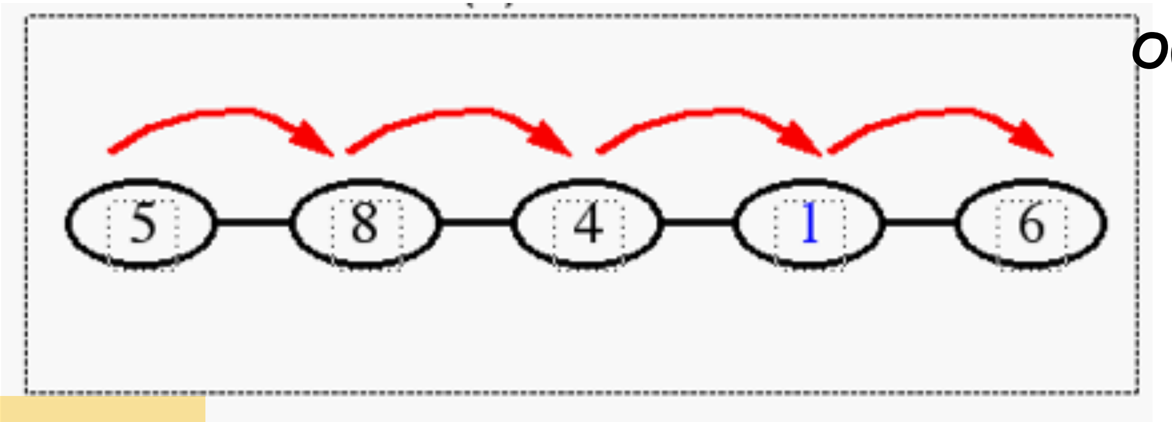
- The elements of S are a composition of two elements;
  - k, the key, and e, the element.
- insertItem() = insertLast() on the sequence. **O(1) time.**



Implementation with an Unsorted Sequence (contd.)

- The sequence is not ordered .

For minElement(), minKey(), and removeMin(),
we need to look at all the elements of S in the worst case.



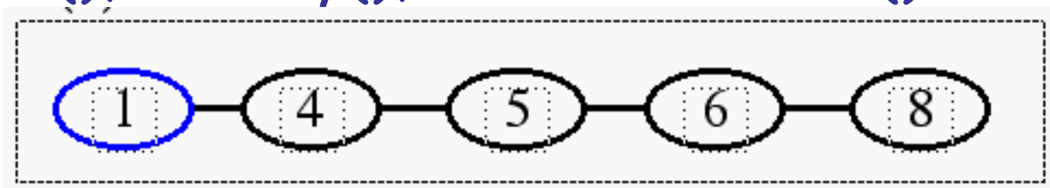
O(n) time.

- Performance summary

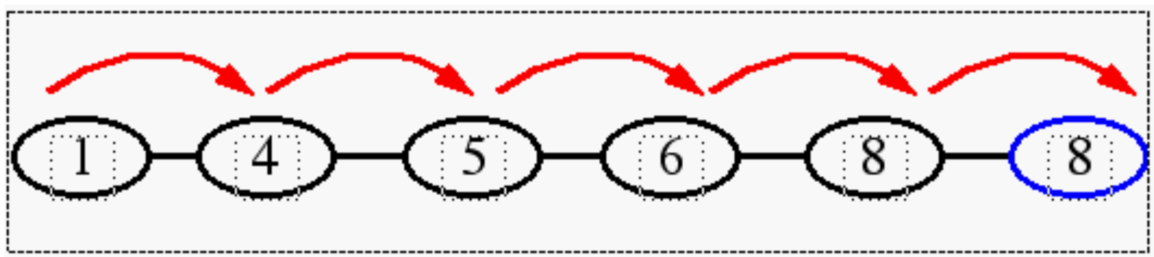
insertItem	$O(1)$
minKey, minElement	$O(n)$
removeMin	$O(n)$

Implementation with Sorted Sequence

- Use a Sequence S, sorted by increasing keys
- minElement(), minKey(), and removeMin() take **O(1) time**



- However, to implement insertItem(), we must now scan through the entire sequence in the worst case. Thus insertItem() runs in **O(n) time**



insertItem	$O(n)$
minKey, minElement	$O(1)$
removeMin	$O(1)$

An observation...

With an unsorted sequence...

removeMin() always takes O(n). That is, these methods runs in Ω(n) time even in the best case. -> Θ(n)

But with a sorted sequence...

insertItem() takes at most O(n)

Application of Priority Queue: Selection Sort

- Variation of PriorityQueueSort that uses an unsorted sequence to implement the priority queue P.
  - Phase 1, the insertion of an item into P takes O(1) time
  - Phase 2, removing an item from P takes time proportional to the current number of elements in P

unsorted sequence

	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)	()

unsorted sequence

Selection Sort (cont.)

- Running time of Selection-sort:

  Inserting the elements into the priority queue with **n** insertItem operations takes **O**(**n**) time

  Removing the elements in sorted order from the priority queue with **n** removeMin operations takes time proportional to

$$n + (n - 1) + \ldots + 2 + 1$$

- Selection-sort runs in $O(n^2)$ time

Application of Priority Queue: Insertion Sort

- PriorityQueueSort implementing the priority queue with a **sorted sequence**

sorted

sequence

	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)	()

sorted

sequence

Insertion Sort(cont.)

Running time of Insertion-sort:

Inserting the elements into the priority queue with **n** insertItem operations takes time proportional to

$$1 + 2 + \ldots + \mathbf{n}$$

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 $\mathbf{O}(\mathbf{n}^2)$ time


I LOVE My
Computer
Because My
Friends
Live In It
More pics on www.imfunny.net