Priority Queues

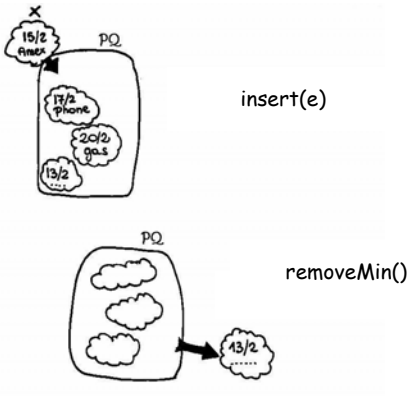
- The priority queue ADT
- Implementing a priority queue with a list
- Elementary sorting using a Priority Queue
- Selection-sort and Insertion-sort

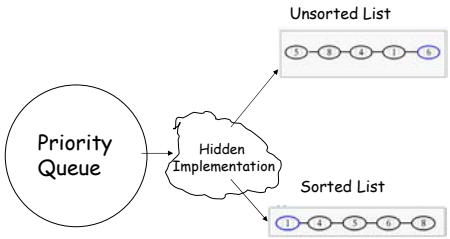
Priority Queue

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





The Priority Queue ADT

A priority queue stores a collection of entries

Each **entry** is a pair (key, value)

or

(key, element)

Keys in a priority queue can be arbitrary objects on which a total order is defined

Two distinct entries in a priority queue can have the same key

Keys and Total Order Relations

- A Priority Queue ranks its elements by **key** with a **total order** relation
- Keys: Every element has its own key
  Keys are not necessarily unique
- 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

- ≤ is a total ordering
- ≥ is also a total ordering
- Alphabetical order: we define a ≤ b if 'a' is before 'b' in alphabetical order
- Reverse alphabetical order

But…

- **<, >** are not total orderings since they are not reflexive
- **=** is not a total ordering since we can't compare any 2 elements with =.
  Given a, b, we do not always have
  a=b or b=a

More examples of ordering

We can order the co-ordinate pairs $p=(x_1, y_1)$ and $q=(x_2,y_2)$ by

1. $p \leq q$ if $x1 \leq x2$
2. $p \leq q$ if $y_1 \leq y_2$
3. $p \leq q$ if $x1 \leq x2$ and $y_1 \leq y_2$

The last one is only a partial ordering!

Entry ADT

- An **entry** in a priority queue is simply a key-value pair (key, value)
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - key(): returns the key for this entry
  - value(): returns the value associated with this entry

Comparator ADT

- The most general and reusable form of a priority queue makes use of **comparator** objects.
- Comparator objects are external to the keys that are to be compared and compare two objects.
- Thus a priority queue can be general enough to store any object.
- The comparator ADT includes:
  - -isLessThan(a, b)
  - -isLessThanOrEqualTo(a,b)
  - -isEqualTo(a, b)
  - -isGreaterThan(a,b)
  - -isGreaterThanOrEqualTo(a,b)
  - -isComparable(a)

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

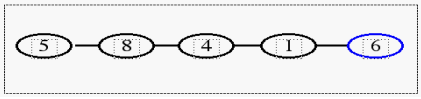
-insert(k,x): Insert into P key k with value x and return entry storing them; error if k is invalid or cannot be compared with other keys

-min(): Return (but don't remove) an entry of P smallest key; an error occurs if P is empty

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

Implementation with an Unsorted List

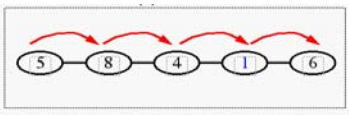
- Store the entries of P in a list S.
- The elements of S are entries (k, x), where the key, and x is the value.
- insert(k,x) on P is like insertLast(e) on S. ***O*(1) time.**



Implementation with an Unsorted List (contd.)

- The sequence is not ordered .

For min(), and removeMin() operation on P, we need to look at all the elements of S in the worst case to find and entry (k,x) of S with minimum k.



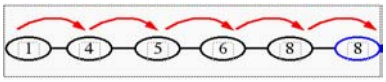
O(n) time.

- Performance summary

insert()	O(1)
min()	O(n)
removeMin()	O(n)

Implementation with Sorted List

- Use a List S to store entries, sorted by increasing keys
- min() and removeMin() on P take **O(1) time** assuming doubly linked list
- However, to implement insert(), we must now scan through the entire list in the worst case. Thus, insert() takes **O(n) time**



An observation…

With an unsorted list…

removeMin() always takes O(n)

→Fast insertions and slow removals

But with a sorted list…

insert() takes at most O(n)

→Fast removals and slow insertions

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 insert() operations -- *Phase 1*
  - ***removing*** the elements from P in increasing order and putting them back into S with a series of removeMin() operations -- *Phase 2*


```
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.insert(e, ø)     {a null value is used}
  while ¬ P.isEmpty() do
       e ← P.removeMin().key()
       S.insertLast(e)    {the smallest key in P is added to end of S}
```

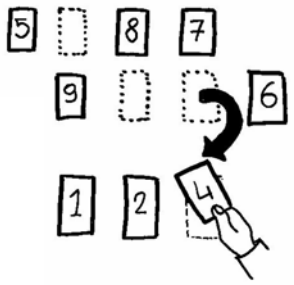

Selection Sort

- 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 (selecting) an item from P takes time proportional to the current number of elements in P

Selection Sort

Insert in no specific order

Select in order



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)
Insert	(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)
Select	(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)	()

Selection Sort (cont.)

unsorted sequence

- Running time of Selection-sort:

  Inserting the elements into the priority queue with n insert operations takes $O(n)$ time

  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

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

  Insert in order

  Select

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)
Insert	(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)
Select	(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 insert operations takes time proportional to

$$1 + 2 + \ldots + n$$

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 $O(n^2)$ time