Priority Queues

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

Copyright © 2010-2011 Wiley
Copyright © 2010 Michael T. Goodrich, Roberto Tamassia
Copyright © 2011 William J. Collins
Copyright © 2011-2021 Aiman Hanna
All rights reserved

Coverage

- The Priority Queue ADT
- Priority Queue List Implementation
  - Sorting with Priority Queue
- Insertion Sort
- Selection Sort

Priority Queue ADT

- A *priority queue* **(P.Q.)** is an ADT for storing a collection of prioritized elements; the elements are referred to as *values*.
- P.Q. supports arbitrary insertion of elements, however the removal of the elements is made in order of priorities.
- Consequently, a P.Q. is fundamentally different from other position-based ADTs (such as stacks, queues, D.Qs, etc.), where operations are conducted on specific positions.
- P.Q. ADT stores elements according to their priorities and exposes no notion of positions to the user.

Priority Queue ADT

- A *key* can be used to indicate the priority of a value (p.s. value means an element here).
- Each entry in the P.Q. is hence a pair of (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, x): insert an entry with key k and value x into PQ, and return the entry storing them
  - removeMin(): remove and returns the entry with smallest key (smallest key indicates first priority).

Priority Queue ADT

- Additional methods
  - min(): return the entry with smallest key, but do not remove it
  - size(), isEmpty()
- Applications:
  - Auctions
  - Stock market
  - …

Priority Queue ADT

- Example of a P.Q.
  - Notice that the "Priority Queue" column is somewhat deceiving since it shows that the entries are sorted by keys, which is more than required of a P.Q.

Operations	Output	Priority Queue
insert(5, A)	e_1 [= (5, A)]	{(5, A)}
insert(9, C)	e_2 [= (9, C)]	{(5, A), (9, C)}
insert(3, B)	e_3 [= (3, B)]	{(3, B), (5, A), (9, C)}
insert(7, D)	e_4 [= (7, D)]	{(3, B), (5, A), (7, D), (9, C)}
min()	e_3	{(3, B), (5, A), (7, D), (9, C)}
removeMin()	e_3	{(5, A), (7, D), (9, C)}
size()	3	{(5, A), (7, D), (9, C)}
removeMin()	e_1	{(7, D), (9, C)}
removeMin()	e_4	{(9, C)}

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.
- P.Q. needs a comparison rule that will never contradict itself.
- In order for a comparison rule, which we denote by $\leq$, to be robust, it must define a ***total order*** relation.

Total Order Relations

- The comparison rule must be defined for each pair of keys and must satisfy the following properties:
  - **Reflexive property:**
    $$k \leq k$$
  - **Antisymmetric property:**
    $$k_1 \leq k_2 \wedge k_2 \leq k_1 \Rightarrow k_1 = k_2$$
  - **Transitive property:**
    $$k_1 \leq k_2 \wedge k_2 \leq k_3 \Rightarrow k_1 \leq k_3$$
- A comparison rule that satisfies these three properties will never lead to a comparison contradiction.

Entries & Comparators

- Two important questions must be asked:
  - How do we keep track of the associations between keys and values?
  - How do we compare keys so as to determine the smallest key?
- The definition of a P.Q. implicitly makes use of two special kinds of objects, which answer the above questions:
  - The *entry* object
  - The *comparator* object

Entry ADT

- An entry in a priority queue is simply a key-value pair
- That is, an entry object is actually composed of a key and a value objects
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods of Entry ADT:
  - getKey: returns the key for this entry
  - getValue: returns the value associated with this entry

Entry ADT

- As a Java interface:

```
/**
  * Interface for a key-value
  * pair entry
 **/
public interface  Entry<K,V>  {
   public  K getKey();
   public  V getValue();
}
```

Comparator ADT

- It is important to define a way for specifying the total order relation for comparing keys.
- One possibility is to use a particular key type that the P.Q. can compare.
- The problem with such approach is that the utilization of different keys would require the creation of different/multiple P.Qs.
- An alternative strategy is to require the keys to be able to compare themselves to one another.
- This solution allows us to write a general P.Q. that can store instances of a key class that has a well-established ***natural ordering***.

Comparator ADT

- It is possible to have comparable objects by implementing the java.lang.Comparable interface.
- The problem with such approach however is that there are cases where the keys will be required to provide more information than they should/expected to, such as their comparison rules.
- For instance, there are two natural ways to compare "7" and "21". "7" is < "21" if the rule is integer comparison, where "21" is < "7" if the rule is lexicographic ordering.
- Such cases would require the keys themselves to provide their comparison rules.

Comparator ADT

- Instead, we can use special comparator objects that are external to the keys to supply the comparison rules.
- A comparator encapsulates the action of comparing two objects according to a given total order relation.
- A generic priority queue uses an auxiliary comparator.
- We assume that a priority queue is given a comparator object when it is constructed. The P.Q. uses its comparator for keys comparisons.

Comparator ADT

- Primary method of the Comparator ADT:
  - compare(a, b): returns an integer i such that
    - $i < 0$ if $a < b$,
    - $i = 0$ if $a = b$
    - $i > 0$ if $a > b$
    - An error occurs if a and b cannot be compared.
- The java.util.Comparator interface correspond to the above comparator ADT.

Example Comparator

- Lexicographic comparison of 2-D points:

```
/** Comparator for 2D points under the
    standard lexicographic order. */
public class Lexicographic implements
    Comparator {
  int xa, ya, xb, yb;
  public int compare(Object a, Object b)
    throws ClassCastException {
     xa = ((Point2D) a).getX();
     ya = ((Point2D) a).getY();
     xb = ((Point2D) b).getX();
     yb = ((Point2D) b).getY();
     if (xa != xb)
            return (xb - xa);
     else
            return (yb - ya);
  }
}
```

- Point objects:

```
/** Class representing a point in the
    plane with integer coordinates */
public class Point2D {
   protected int xc, yc; // coordinates
   public Point2D(int x, int y) {
      xc = x;
      yc = y;
   }
   public int getX() {
          return xc;
   }
   public int getY() {
          return yc;
   }
}
```

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

Priority Queue Sorting

```
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 ¬S.isEmpty ()
        e ← S.removeFirst ()
        P.insert (e, ∅)
    while ¬P.isEmpty()
        e ← P.removeMin().getKey()
        S.addLast(e)
```

Notice that in the above code, the elements of the input sequence *S* serve as keys of the priority queue *P*.

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 algorithm works as follows:
  - Find the minimum value in the collection (list/sequence, P.Q., etc.)
  - Swap it with the value in the first position
  - Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

- Click here to view some illustrative animations

- What is the running time?

Selection-Sort

- Running time of the ***PQ-Sort*(S, C)** Selection-sort; that is when the P.Q. is implemented with unsorted sequence:
  1. (First loop): Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. (Second loop) Removing the elements in sorted order (repeated seclection) from the priority queue with n removeMin operations takes time proportional to
     $$n + n - 1 + n - 2 + \ldots + 3 + 2 + 1$$

  Resulting in a total of $O(n + n^2) \rightarrow O(n^2)$
- ➔ 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 (first loop)		
(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 (second loop)		
(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 algorithm is as follows:
  - Removes an element (possibly arbitrary) from the input data
  - Insert the element into the correct position in the already-sorted list
  - Repeat until no input elements remain.

- Click here to view some illustrative animations

- What is the running time? What is fastest case?

Insertion-Sort

- Running time of the ***PQ-Sort*(*S*, *C*)** Insertion-sort; that is when the P.Q. is implemented with sorted sequence (obtained after phase 1 (first loop) is finished):
  1. (First loop): Inserting the elements into the priority queue with n insert operations takes time proportional to

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

  2. (Second loop): Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time

  Resulting in a total of $O(n^2 + n) \Rightarrow O(n^2)$

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

Insertion-Sort

- Special case:
  - If the sequence is already (by luck) sorted, then

1. (First loop): Sorting the list will take O(n) time
2. (Second loop): Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time

Resulting in a total of $O(n+ n)$ ➔ $O(n)$

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1 (first loop)		
(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 (secondl oop)		
(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