Lecture 7-1: Priority Queues

Sunghyun Cho
Professor
Division of Computer Science
chopro@hanyang.ac.kr

HANYANG UNIVERSITY

Keywords

- Standby Passenger Queue
- Searching & Sorting

Contents

- **The Priority Queue ADT**
  - Priorities and Total Order Relations
  - The Priority Queue ADT
  - Sorting with a Priority Queue
- **Implementing a Priority Queue with a List**
  - Priority Queue Implementation Using a List
  - Selection Sort
  - Insertion Sort

What is a Priority Queue(P) ?

- Definition
  - An abstract data type for storing a collection of prioritized elements that supports **arbitrary element insertion** but supports **removal of elements in order of priority**
- Features
  - Fundamentally different from the position-based data structures such as stacks, queues, lists, and even trees
  - Previous data structures store elements at specific positions (determined by the insertion and deletion methods performed)
  - The priority queue ADT **stores elements according to their priorities**, and exposes no notion of "position" to the user
  - In a priority queue, the element with **first priority can be removed at any time**

New Terms in Priority Queue

- Key
  - An object that is assigned to an element as a specific attribute for that element, which can be used to identify or weigh that element
  - A property that an element did not originally posses
  - Not necessarily unique and changeable
- Entry
- Total Ordering (Total Order Relation)
- Comparator

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
- ***Total Order Relation***
  - The comparison rule is defined for every pair of keys and it must satisfy the following properties (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: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Entry ADT

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

- As a Java interface:

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

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- 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.

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 ADT

- A priority queue stores a collection of prioritized entries
  - Priorities are assigned to entries
    - Arbitrary elements insertions
    - Removal of elements in order of priority.
  - Different from position-based D.S. (e.g. stack, queue, tree)
    - PQ : No notion of "position"
- An entry is a pair (key, element)
- Key : An object that is assigned to an element as a specific attribute for that element by user or application
  - Used to identify, compare, rank, or weight that element (each element is associated with a key)

PQ ADT (cont.)

- **Main methods of the Priority Queue ADT**
  - **insertItem**(k, o)
    inserts an entry with key k and element o
  - **removeMin**( )
    removes and returns an entry (k,o) having minimal key from the priority queue; return null if the priority queue is empty
- Additional methods
  - minKey(k, o)
    returns, but does not remove, the smallest key of an entry
  - minElement( )
    returns, but does not remove, the element of an entry with smallest key
  - size( ), isEmpty( )
- **Applications:**
  - Standby flyers, Auctions, Stock market

PQ ADT (cont.)

- The effects of a series of operations on an initially empty PQ.

Operation	*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	$\{(9,C)\}$
removeMin()	e_2	$\{\}$
removeMin()	"error"	$\{\}$
isEmpty()	true	$\{\}$

Sorting with a Priority Queue

- 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

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*)

Implementing a Priority Queue

- Implementation of a PQ
  - PQ using sequence
    - ➢ Simple, but not very efficient
    - ➢ Two sorting algorithm
      - Selection sort
      - Insertion Sort
  - PQ based on heap
    - ➢ Efficient implementation
      - Support PQ operations in log() time
      - Heap-sort

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 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

$$O(n + (n-1) + \cdots + 2 + 1) = O\left(\sum_{i=1}^{n} i\right)$$

- 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)
..	..	.. O(n^2)
(g)	() O(n^2)	(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

$$O(1+2+\cdots+(n-1)+n) = O(\sum_{i=1}^{n} i)$$

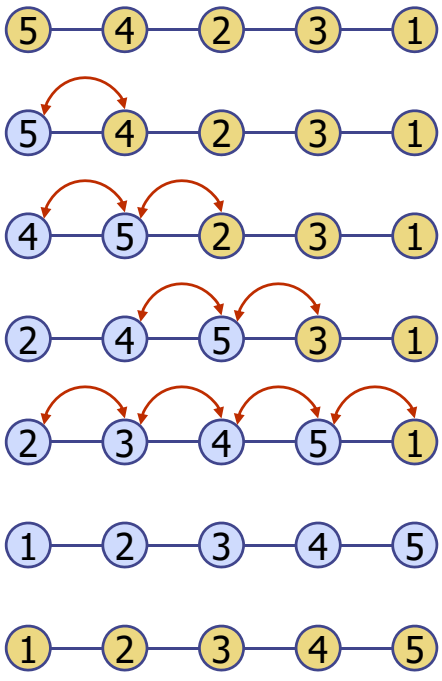
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

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



general 한 정렬 알고리즘보다 장점은 메모리를 더 적게쓴다는 것이다

러닝타임은 O(n^2)

Q & A

한양대학교 ERICA 캠퍼스