IS2020 COMP 2540: Data Structures and Algorithms
Lecture 06: Priority Queues

Dr. Kalyani Selvarajah
kalyanis@uwindsor.ca

School of Computer Science
University of Windsor
Windsor, Ontario, Canada

June 22, 2020

Outline

- Introduction
- Priority Queue ADT
- Application of PQ
    - Selection Sort
    - Insertion Sort

Introduction: Priority Queue

- Priority queues are a generalization of stacks and queues. Rather than inserting and deleting elements in a fixed order, each element is assigned a priority represented by an integer.

- We always remove an element with the **highest priority**, which is given by the minimal integer priority assigned.

- Priority queues often have a fixed size.
  - For example, in an operating system the runnable processes might be stored in a priority queue, where certain system processes are given a higher priority than user processes.
  - In a network router packets may be routed according to some assigned priorities.

- Applications:
  - Standby flyers
  - Auctions
  - Stock market
  - "To Do" list
  - Deadline to pay a bill

Priority Queue ADT

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, v) inserts an entry with key k and value v
  - removeMin() removes and returns the entry with smallest key, or null if the the priority queue is empty
- Additional methods
  - min() returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  - size() Returns the number of entries in the priority queue.
  - isEmpty() Returns a boolean indicating whether the priority queue is empty.

Example: A sequence of priority queue methods

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Self Assessment

What does each removeMin call return within the following sequence of priority queue ADT operations: insert(5, A), insert(4, B), insert(7, F), insert(1, D), removeMin(), insert(3, J), insert(6, L), removeMin(), removeMin(), insert(8, G), removeMin(), insert(2, H), removeMin(), removeMin( )?

Implementing a Priority Queue

1. 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

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

Java interface for an entry storing a key-value pair.

```java
/* Interface for the priority queue ADT */
public interface PriorityQueue<K,V> {
      int size();
      boolean isEmpty();
      Entry<K,V> insert(K key, V value) throws
IllegalArgumentException;
      Entry<K,V> min();
      Entry<K,V> removeMin();
}
```

Java interface for the priority queue ADT

2. Comparing Keys with Total Orders Relations

- A Priority Queue ranks its elements by key with a total order relation.
- 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 $\leq$ (a rule for comparing keys)
  - Comparability property: either $\boldsymbol{x} \leq \boldsymbol{y}$ or $\boldsymbol{y} \leq \boldsymbol{x}$
  - Antisymmetric property: $\boldsymbol{x} \leq \boldsymbol{y}$ and $\boldsymbol{y} \leq \boldsymbol{x} \Rightarrow \boldsymbol{x} = \boldsymbol{y}$
  - Transitive property: $\boldsymbol{x} \leq \boldsymbol{y}$ and $\boldsymbol{y} \leq \boldsymbol{z} \Rightarrow \boldsymbol{x} \leq \boldsymbol{z}$

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
- $=$ is not a total ordering since we can't compare any 2 elements with $=$.

More Example of ordering:

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

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(x, y): 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

- Two points on the plane a and b, given by their coordinates $(x_a , y_a)$and and (x_b , y_b) respectively…

  which one is the largest of the two?

- First, they follow a partial order
- But sorting them (lexicographically):
  - from left to right, and then from bottom to top
  - we get a total order

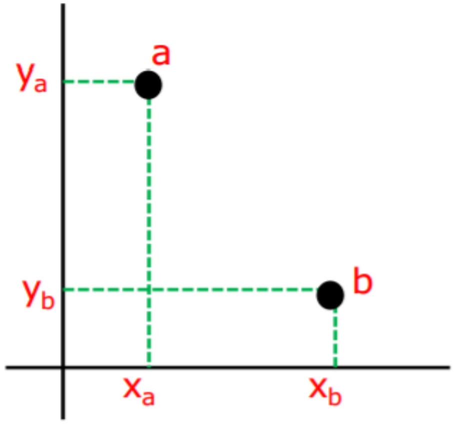
Example Comparator in Java (1)

Lexicographic comparison of 2-D points:

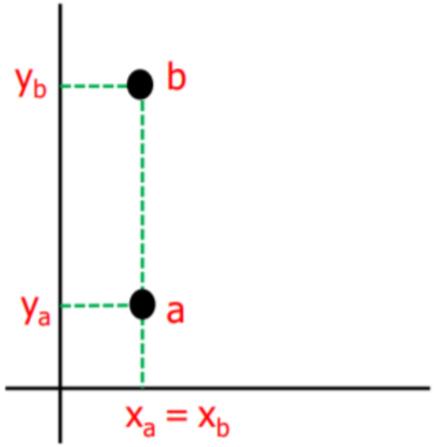
```
/** Comparator for 2D points under the standard lexicographic order. */
public class  Lexicographic  implements  Comparator  {
    int  xa, ya, xb, yb;
    public int  compare(Object a, Object b)  throws  ClassCastException
         {
        xa = ((Point2D) a).getX();
        ya = ((Point2D) a).getY();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



$i = x_a - x_b < 0 \Longrightarrow a < b$



$i = y_a - y_b < 0 \Longrightarrow a < b$

Example Comparator in Java (2)

- A comparator that evaluates strings based on their lengths.

```
public class StringLengthComparator implements Comparator<String> {
    /**
     * Compares two strings according to their lengths.
     */
    public int compare(String a, String b) {
        if (a.length() < b.length()) return -1;
        else if (a.length() == b.length()) return 0;
        else return 1;
    }
}
```

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

Method	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

Priority Queue Sorting

- We can use a priority queue to sort a list 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

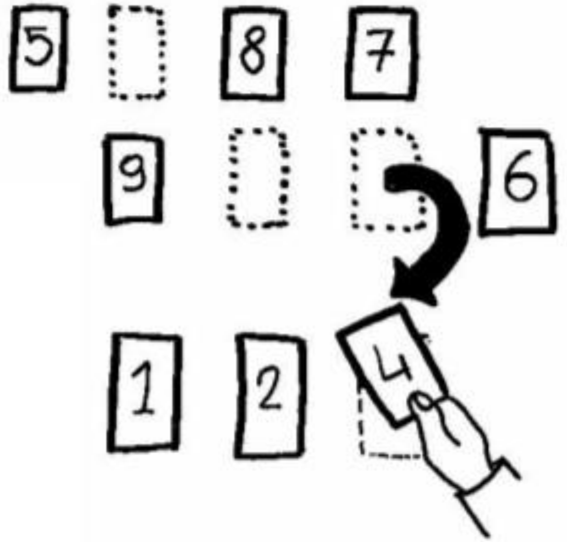
Algorithm *PQ-Sort* (S, C)
    Input list S, comparator C for the elements of S
    Output list S sorted in increasing order according to C
    $P \leftarrow$ priority queue with comparator C
    while $\neg$ *S.isEmpty* ()
        $e \leftarrow$ *S.remove*(*S.first* ())
        P.insert (e, $\varnothing$)
    while $\neg$ *P.isEmpty*()
        $e \leftarrow$ *P.removeMin*().*getKey*()
        S.addLast(e)

1. 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

$$1 + 2 + \ldots + n = {}^{n(n+1)}/_{2}$$

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



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)
(b)	(8,2,5,3,9)	(7,4)
(c)	(2,5,3,9)	(7,4,8)
..	..	..
(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)	()

2. 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

$$1 + 2 + \ldots + n = {}^{n(n+1)}/_{2}$$

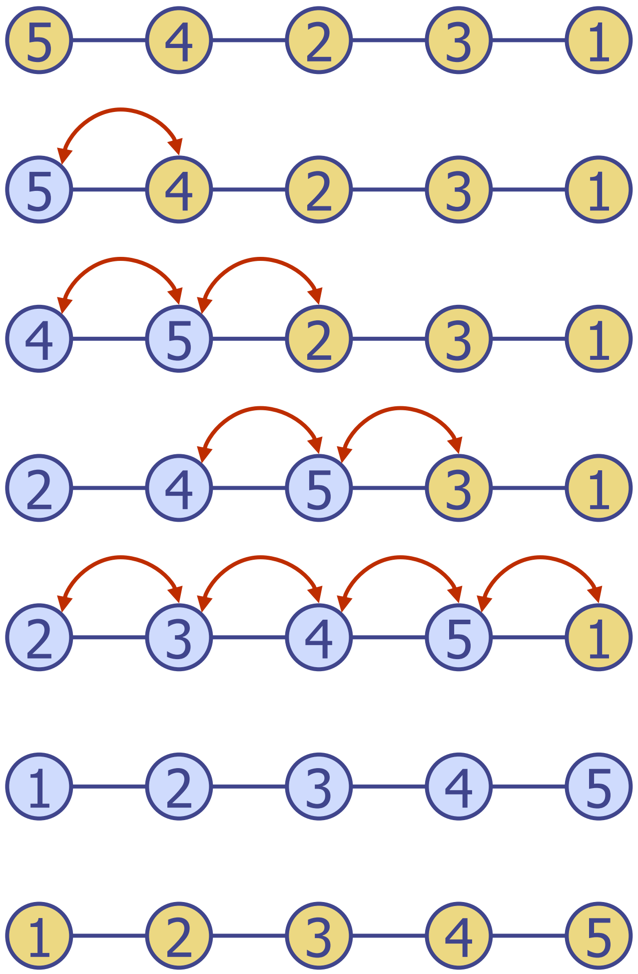
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

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)	()

3. 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



Self Assessment

1. Illustrate the execution of the selection-sort algorithm on the following input sequence: (22, 15, 36, 44, 10, 3, 9, 13, 29, 25).

2. Illustrate the execution of the insertion-sort algorithm on the input sequence of the previous problem.