CS121 Data Structures A, C
Priority Queues

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Priorities

Recall that the queue ADT is a collection of objects that are added and removed according to the *first-in, first-out (FIFO)* principle

In some situations, priorities come into play instead of the FIFO principle

The new ADT known as a **priority queue** is a collection of prioritized elements that allows artibrary element insertion, and allows the removal of the element that has *first priority*

When an element is added to a priority queue, the user designates its priority by providing an associated **key**

The element with the *minimal* key will be the next to be removed from the queue, i.e. key 1 given priority over key 2

The Priority Queue Abstract Data Type

Each **entry** is a pair (key, value), where value is the element and key provides its priority

We define the priority queue ADT to support:

insert(k, v): Creates an entry with key k and value v in the priority queue

min(): Returns (but does not remove) a priority queue entry (k, v) having minimal key; returns `null` if the priority queue is empty

removeMin(): Removes and returns an entry (k, v) having minimal key from the priority queue; returns `null` if the priority queue is empty

size(): Returns the number of elements in the priority queue

isEmpty(): Returns a boolean indicating whether the priority queue is empty

Draws among elements with equivalent keys are resolved arbitrarily

Example

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	{ }

Entries

When implementing a priority queue we must keep track of both an element and its key, even as entries are relocated within a data structure

We use the **composition design pattern** to define an entry as a pair of a key k and a value v as a single object

```
1  /** Interface for a key−value pair. */
2  public interface Entry<K,V> {
3    K getKey( );                    // returns the key stored in this entry
4    V getValue( );                  // returns the value stored in this entry
5  }
```

The Priority Queue API

We use the `Entry` type in the formal interface for the priority queue

```
1  /** Interface for the priority queue ADT. */
2  public interface PriorityQueue<K,V> {
3    int size( );
4    boolean isEmpty( );
5    Entry<K,V> insert(K key, V value) throws IllegalArgumentException;
6    Entry<K,V> min( );
7    Entry<K,V> removeMin( );
8  }
```

To manage technical issues common to all our priority queue implementations, we then define an abstract base class named `AbstractPriorityQueue`

Total Orders

Any type of object can serve as a key, but we must be able to compare keys to each other in a meaningful way

A comparison rule, denoted by $\leq$, must define a **total order** relation, i.e. for any keys k_1, k_2, and k_3:

- **Comparability property:** $k_1 \leq k_2$ or $k_2 \leq k_1$
- **Antisymmetric property:** if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$
- **Transitive property:** if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$

Comparability property implies **reflexive property:** $k \leq k$

If a (finite) set of elements has a total order defined for it, then the notion of a **minimal** key, k_{min}, is well defined, as a key in which $k_{min} \leq k$, for any other key k in our set

The Comparable Interface in Java

A class may define the **natural ordering** of its instances by formally implementing the `java.lang.Comparable` interface

This interface includes a single method, **compareTo**

The syntax a`.compareTo(`b`)` must return an integer i with the following meaning:

- $i < 0$ designates that $a < b$
- $i = 0$ designates that $a = b$
- $i > 0$ designates that $a > b$

For example, the `compareTo` method of the `String` class defines the natural ordering of strings to be **lexicographic**, which is a case-sensitive extension of the alphabetic ordering to Unicode

The Comparator Interface in Java

To compare objects according to some notion other than their natural ordering, we use the `java.util.Comparator` interface

A **comparator** is an object that is external to the class of the keys it compares and provides a `compare`(a, b) method returning an integer, similar to the `compareTo` method just described

```
1  public class StringLengthComparator implements Comparator<String> {
2    /** Compares two strings according to their lengths. */
3    public int compare(String a, String b) {
4      if (a.length( ) < b.length( )) return −1;
5      else if (a.length( ) == b.length( )) return 0;
6      else return 1;
7    }
8  }
```

How does this comparator evaluate strings?

The Comparator Interface in Java

To compare objects according to some notion other than their natural ordering, we use the `java.util.Comparator` interface

A **comparator** is an object that is external to the class of the keys it compares and provides a `compare`(a, b) method returning an integer, similar to the `compareTo` method just described

```
1  public class StringLengthComparator implements Comparator<String> {
2    /** Compares two strings according to their lengths. */
3    public int compare(String a, String b) {
4      if (a.length( ) < b.length( )) return −1;
5      else if (a.length( ) == b.length( )) return 0;
6      else return 1;
7    }
8  }
```

How does this comparator evaluate strings?

Based on their length (rather than their natural lexicographic order)

Comparators and the Priority Queue ADT

For a general and reusable form of a priority queue, we allow a user

1. to choose any key type and
2. to send an appropriate comparator instance as a parameter to the priority queue constructor

The priority queue will use that comparator anytime it needs to compare two keys to each other

For convenience, we also allow a default priority queue to instead rely on the natural ordering for the given keys (assuming those keys come from a comparable class)

```
1  public class DefaultComparator<E> implements Comparator<E> {
2    public int compare(E a, E b) throws ClassCastException {
3      return ((Comparable<E>) a).compareTo(b);
4    }
5  }
```

The AbstractPriorityQueue Base Class in Java

```
1  /** An abstract base class to assist implementations of the PriorityQueue interface.*/
2  public abstract class AbstractPriorityQueue<K,V>
3                                          implements PriorityQueue<K,V> {
4    //---------------------- nested PQEntry class ----------------------
5    protected static class PQEntry<K,V> implements Entry<K,V> {
6      private K k;    // key
7      private V v;    // value
8      public PQEntry(K key, V value) {
9        k = key;
10       v = value;
11     }
12     // methods of the Entry interface
13     public K getKey( ) { return k; }
14     public V getValue( ) { return v; }
15     // utilities not exposed as part of the Entry interface
16     protected void setKey(K key) { k = key; }
17     protected void setValue(V value) { v = value; }
18   } //------------- end of nested PQEntry class -------------
19
```

The AbstractPriorityQueue Base Class in Java (cont'd)

```
20    // instance variable for an AbstractPriorityQueue
21    /** The comparator defining the ordering of keys in the priority queue. */
22    private Comparator<K> comp;
23    /** Creates an empty priority queue using the given comparator to order keys. */
24    protected AbstractPriorityQueue(Comparator<K> c) { comp = c; }
25    /** Creates an empty priority queue based on the natural ordering of its keys. */
26    protected AbstractPriorityQueue( ) { this(new DefaultComparator<K>( )); }
27    /** Method for comparing two entries according to key */
28    protected int compare(Entry<K,V> a, Entry<K,V> b) {
29      return comp.compare(a.getKey( ), b.getKey( ));
30    }
31    /** Determines whether a key is valid. */
32    protected boolean checkKey(K key) throws IllegalArgumentException {
33      try {
34        return (comp.compare(key,key) == 0); // see if key can be compared to itself
35      } catch (ClassCastException e) {
36        throw new IllegalArgumentException("Incompatible key");
37      }
38    }
39    /** Tests whether the priority queue is empty. */
40    public boolean isEmpty( ) { return size( ) == 0; }
41  }
```

Implementing a Priority Queue with an Unsorted List

We consider two concrete implementations of a priority queue:

- storing entries within an *unsorted* linked list
- storing entries within a *sorted* linked list

`UnsortedPriorityQueue` class stores the `PQEntry` entries within a `PositionalList`, implemented with a doubly linked list

Key–value pairs are added to the end of the list; all entries are inspected to find one with a minimal key

Implementing a Priority Queue with an Unsorted List

We consider two concrete implementations of a priority queue:

- storing entries within an *unsorted* linked list
- storing entries within a *sorted* linked list

`UnsortedPriorityQueue` class stores the `PQEntry` entries within a `PositionalList`, implemented with a doubly linked list

Key–value pairs are added to the end of the list; all entries are inspected to find one with a minimal key

Method	**Running Time**
size	$O(1)$
isEmpty	$O(1)$
insert	$O(1)$
min	$O(n)$
removeMin	$O(n)$

Space usage: $O(n)$, where n is the number of entries in the priority queue

The UnsortedPriorityQueue Class in Java

```
1  /** An implementation of a priority queue with an unsorted list. */
2  public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3    /** primary collection of priority queue entries */
4    private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>( );
5
6    /** Creates an empty priority queue based on the natural ordering of its keys. */
7    public UnsortedPriorityQueue( ) { super( ); }
8    /** Creates an empty priority queue using the given comparator to order keys. */
9    public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }
10
11   /** Returns the Position of an entry having minimal key. */
12   private Position<Entry<K,V>> findMin( ) { // only called when nonempty
13     Position<Entry<K,V>> small = list.first( );
14     for (Position<Entry<K,V>> walk : list.positions( ))
15       if (compare(walk.getElement( ), small.getElement( )) < 0)
16         small = walk;                          // found an even smaller key
17     return small;
18   }
19
```

The UnsortedPriorityQueue Class in Java (cont'd)

```
20    /** Inserts a key–value pair and returns the entry created. */
21    public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
22      checkKey(key);   // auxiliary key–checking method (could throw exception)
23      Entry<K,V> newest = new PQEntry<>(key, value);
24      list.addLast(newest);
25      return newest;
26    }
27
28    /** Returns (but does not remove) an entry with minimal key. */
29    public Entry<K,V> min( ) {
30      if (list.isEmpty( )) return null;
31      return findMin( ).getElement( );
32    }
33
34    /** Removes and returns an entry with minimal key. */
35    public Entry<K,V> removeMin( ) {
36      if (list.isEmpty( )) return null;
37      return list.remove(findMin( ));
38    }
39
40    /** Returns the number of items in the priority queue. */
41    public int size( ) { return list.size( ); }
42  }
```

Implementing a Priority Queue with a Sorted List

We consider two concrete implementations of a priority queue:

- storing entries within an *unsorted* linked list
- storing entries within a *sorted* linked list

`SortedPriorityQueue` class stores the entries within a `PositionalList` (implemented with a doubly linked list) sorted by nondecreasing keys: the first element of the list is an entry with the smallest key

finding an element with a minimal key is straightforward; the list is scanned to find the appropriate position to insert a new entry

Implementing a Priority Queue with a Sorted List

We consider two concrete implementations of a priority queue:

- storing entries within an *unsorted* linked list
- storing entries within a *sorted* linked list

`SortedPriorityQueue` class stores the entries within a `PositionalList` (implemented with a doubly linked list) sorted by nondecreasing keys: the first element of the list is an entry with the smallest key

finding an element with a minimal key is straightforward; the list is scanned to find the appropriate position to insert a new entry

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

Space usage: $O(n)$, where n is the number of entries in the priority queue

Which implementation should be chosen?

The SortedPriorityQueue Class in Java

```
1   /** An implementation of a priority queue with a sorted list. */
2   public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3     /** primary collection of priority queue entries */
4     private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>( );
5
6     /** Creates an empty priority queue based on the natural ordering of its keys. */
7     public SortedPriorityQueue( ) { super( ); }
8     /** Creates an empty priority queue using the given comparator to order keys. */
9     public SortedPriorityQueue(Comparator<K> comp) { super(comp); }
10
11    /** Inserts a key−value pair and returns the entry created. */
12    public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
13      checkKey(key);   // auxiliary key−checking method (could throw exception)
14      Entry<K,V> newest = new PQEntry<>(key, value);
15      Position<Entry<K,V>> walk = list.last( );
16      // walk backward, looking for smaller key
17      while (walk != null && compare(newest, walk.getElement( )) < 0)
18        walk = list.before(walk);
19      if (walk == null)
20        list.addFirst(newest);                    // new key is smallest
21      else
22        list.addAfter(walk, newest);              // newest goes after walk
23      return newest;
24    }
25
```

The SortedPriorityQueue Class in Java (cont'd)

```
26    /** Returns (but does not remove) an entry with minimal key. */
27    public Entry<K,V> min( ) {
28      if (list.isEmpty( )) return null;
29      return list.first( ).getElement( );
30    }
31
32    /** Removes and returns an entry with minimal key. */
33    public Entry<K,V> removeMin( ) {
34      if (list.isEmpty( )) return null;
35      return list.remove(list.first( ));
36    }
37
38    /** Returns the number of items in the priority queue. */
39    public int size( ) { return list.size( ); }
40  }
```

Summary

Reading

Section 9.1 The Priority Queue Abstract Data Type

Section 9.2 Implementing a Priority Queue

Questions?