Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

The Priority Queue Abstract Data Type

- Consider, for example, an air-traffic control center that has to decide which flight to clear for landing from among many approaching the airport. This choice may be influenced by factors such as each plane's distance from the runway, time spent waiting in a holding pattern, or amount of remaining fuel.
- Is a FIFO policy good for the plane landing decisions?
- In this lecture, we introduce a new abstract data type known as a **priority queue**
- A priority queue is a more specialized data structure than a stack or a queue

The Priority Queue Abstract Data Type

- The Priority Queue is a *collection of prioritized elements* that allows arbitrary 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
- Although it is quite common for priorities to be expressed numerically, any Java object may be used as a key, as long as there exists means to compare any two instances a and b, in a way that defines a natural order of the keys
- With such generality, applications may develop their own notion of priority for each element

The Priority Queue ADT

- We model an element and its priority as a key-value composite known as an ***entry***
- We define the priority queue ADT to support the following methods:
  - **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 entries in the priority queue.
  - **isEmpty()**: Returns a boolean indicating whether the priority queue is empty.
- A priority queue may have multiple entries with equivalent keys, in which case methods *min* and *removeMin* may report an arbitrary choice among those entry having minimal key

Priority Queue Operations

- The following table shows a series ofoperations and their effects on an initially empty priority queue

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	{ }

Implementing a Priority Queue

- One challenge in implementing a priority queue is that we must keep track of both an element and its key, even as entries are relocated within a data structure
- For priority queues, we use composition to pair a key ***k*** and a value ***v*** as a single object
- To formalize this, we define the public interface, **Entry**
- This allows us to return both a key and value as a single object frommethods such as min and removeMin

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

Implementing a Priority Queue

- We use the **Entry** type in the formal interface for the priority queue

```
/** 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( );
}
```

Comparing Keys with Total Orders

- In defining the priority queue ADT, we can allow *any type of object to serve as a key*, but we must be able to compare keys to each other in a meaningful way
- For a comparison rule, which we denote by ≤, to be self-consistent, it must define a ***total order relation***, which is to say that it satisfies the following properties 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 $k_1 \leq k_3$.
- The comparability property states that comparison rule is defined for every pair of keys

The Comparable Interface -1

- Java provides two means for defining comparisons between object types
- The first of these is that a class may define what is known as the *natural ordering* of its instances by formally implementing the ***java.lang.Comparable*** interface, which 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

The Comparable Interface -2

- In some applications, we may want to compare objects according to some notion *other than their natural ordering*
- For example, we might be interested in which of two strings is the shortest, or in defining our own complex rules for judging which of two stocks is more promising
- To support generality, Java defines the ***java.util.Comparator*** interface
- A comparator is an object that is external to the class of the keys it compares
- It provides a method with the signature ***compare(a, b)*** that returns an integer with similar meaning to the compareTo method described above
- A comparator that evaluates strings based on their length (rather than their natural lexicographic order)

```
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;
  }
}
```

Comparators and the Priority Queue ADT

- For a general and reusable form of a priority queue, we allow a user to choose any key type and 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*

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

The AbstractPriorityQueue Base Class - 1

- A nested PQEntry class that implements the public ***Entry interface***
- It provides get and set methods for key and value attributes

```
//---------------- nested PQEntry class ----------------
protected static class PQEntry<K,V> implements Entry<K,V> {
  private K k;    // key
  private V v;    // value
  public PQEntry(K key, V value) {
    k = key;
    v = value;
  }
  // methods of the Entry interface
  public K getKey( ) { return k; }
  public V getValue( ) { return v; }
  // utilities not exposed as part of the Entry interface
  protected void setKey(K key) { k = key; }
  protected void setValue(V value) { v = value; }
} //---------- end of nested PQEntry class ----------
```

The AbstractPriorityQueue Base Class -2

- Abstract class declares and initializes an instance variable ***comp*** that stores the *comparator being used* for the priority queue

```
/** An abstract base class to assist implementations of the PriorityQueue interface.*/
public abstract class AbstractPriorityQueue<K,V>
                                        implements PriorityQueue<K,V> {
  //---------------- nested PQEntry class ----------------
  // instance variable for an AbstractPriorityQueue
  /** The comparator defining the ordering of keys in the priority queue. */
  private Comparator<K> comp;
  /** Creates an empty priority queue using the given comparator to order keys. */
  protected AbstractPriorityQueue(Comparator<K> c) { comp = c; }
  /** Creates an empty priority queue based on the natural ordering of its keys. */
  protected AbstractPriorityQueue( ) { this(new DefaultComparator<K>( )); }
  /** Method for comparing two entries according to key */
  protected int compare(Entry<K,V> a, Entry<K,V> b) {
    return comp.compare(a.getKey( ), b.getKey( ));
  }
```

The AbstractPriorityQueue Base Class -3

- Define method checkKey to decide is key is camparable using ***compare*** method

```
  /** Determines whether a key is valid. */
  protected boolean checkKey(K key) throws IllegalArgumentException {
    try {
      return (comp.compare(key,key) == 0);  // see if key can be compared to itself
    } catch (ClassCastException e) {
      throw new IllegalArgumentException("Incompatible key");
    }
  }
  /** Tests whether the priority queue is empty. */
  public boolean isEmpty( ) { return size( ) == 0; }
}
```

Implementing a Priority Queue with an Unsorted List

- Concrete implementation of a priority queue that stores entries within an ***unsorted linked list***
- For internal storage, ***key-value pairs*** are represented as composites, using instances of the inherited *PQEntry class*
- These entries are stored within a *PositionalList* that is an instance variable
- We use the positional list, which is implemented with a doubly linked list that allows all operations of that ADT execute in O(1) time
- At all times, the ***size*** *of the list* equals the *number of key-value pairs currently stored in the priority queue*
- Each time a ***key-value pair is added*** to the priority queue, via the insert method, we ***create a new PQEntry composite*** for the given key and value, and add that entry to the end of the list
- Methods min or removeMin must locate the entry with minimal key and then operate on it
- For convenience, we define a private ***findMin*** utility that returns the position of an entry with minimal key
- Due to the loop for finding the minimal key, both min and removeMin methods run in **O(n) time**

Implementing a Priority Queue with an Unsorted List

- A summary of the running times for the UnsortedPriorityQueue class is:

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

- Worst-case running times of the methods of a priority queue of size n, realized by means of an unsorted, doubly linked list
- The space requirement is O(n)

Implementing a Priority Queue with an Unsorted List -1

```
/** An implementation of a priority queue with an unsorted list. */
public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public UnsortedPriorityQueue() { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }

  /** Returns the Position of an entry having minimal key. */
  private Position<Entry<K,V>> findMin() {    // only called when nonempty
    Position<Entry<K,V>> small = list.first();
    for (Position<Entry<K,V>> walk : list.positions())
      if (compare(walk.getElement(), small.getElement()) < 0)
        small = walk;          // found an even smaller key
    return small;
  }
```

Implementing a Priority Queue with an Unsorted List -2

```
/** Inserts a key-value pair and returns the entry created. */
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
  checkKey(key);      // auxiliary key-checking method (could throw exception)
  Entry<K,V> newest = new PQEntry<>(key, value);
  list.addLast(newest);
  return newest;
}
/** Returns (but does not remove) an entry with minimal key. */
public Entry<K,V> min( ) {
  if (list.isEmpty( )) return null;
  return findMin( ).getElement( );
}
```

```
/** Removes and returns an entry with minimal key. */
public Entry<K,V> removeMin( ) {
  if (list.isEmpty( )) return null;
  return list.remove(findMin( ));
}

/** Returns the number of items in the priority queue. */
public int size( ) { return list.size( ); }
}
```

Implementing a Priority Queue with a Sorted List

- A priority queue maintains entries sorted by ***nondecreasing keys***
- This ensures that the first element of the list is an entry with the smallest key
- The implementation of min and removeMin are rather straightforward given knowledge that the first element of a list has a minimal key
- List is implemented with a doubly linked list, operations ***min*** and ***removeMin*** take **O(1)** time
- This benefit comes at a cost, however, for method insert now requires that we scan the list to find the appropriate position to insert the new entry
- The ***insert method*** takes **O(n) worst-case time**, where n is the number of entries in the priority queue at the time the method is executed

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	$O(n)$	$O(1)$

Implementing a Priority Queue with a Sorted List-1

```
/** An implementation of a priority queue with a sorted list. */
public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public SortedPriorityQueue() { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public SortedPriorityQueue(Comparator<K> comp) { super(comp); }
```

Implementing a Priority Queue with a Sorted List-2

```
/** Inserts a key-value pair and returns the entry created. */
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
  checkKey(key);      // auxiliary key-checking method (could throw exception)
  Entry<K,V> newest = new PQEntry<>(key, value);
  Position<Entry<K,V>> walk = list.last( );
  // walk backward, looking for smaller key
  while (walk != null && compare(newest, walk.getElement( )) < 0)
    walk = list.before(walk);
  if (walk == null)
    list.addFirst(newest);                          // new key is smallest
  else
    list.addAfter(walk, newest);                    // newest goes after walk
  return newest;
}
```

Implementing a Priority Queue with a Sorted List-3

```
  /** Returns (but does not remove) an entry with minimal key. */
  public Entry<K,V> min( ) {
    if (list.isEmpty( )) return null;
    return list.first( ).getElement( );
  }

  /** Removes and returns an entry with minimal key. */
  public Entry<K,V> removeMin( ) {
    if (list.isEmpty( )) return null;
    return list.remove(list.first( ));
  }

  /** Returns the number of items in the priority queue. */
  public int size( ) { return list.size( ); }
}
```