PRIORITY QUEUES

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

PRIORITY QUEUE

- Queue ADT is a collection of objects that are added and removed according to the ***first-in, first-out*** (***FIFO***) principle.
- However, sometimes a FIFO policy does not suffice.
  - Ex> "first come, first serve" policy might seem reasonable, but other priorities also come into play.
- A **priority queue** is a data structure for storing prioritized elements that allows arbitrary insertion, and allows the removal of the element that has **first priority (***minimal* key**)**.
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

PRIORITY QUEUE ADT

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Priority is stored in the key

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

- Main methods
  - 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(), isEmpty()

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	{ }

ENTRY ADT

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

- Java interface:

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

COMPARABLE INTERFACE

- Java provides two means for defining comparisons between object types
  - First, implementing the **java.lang.Comparable interface,** which includes a single method, compareTo.
  - Second, implementing Comparator Interface
- implementing the **java.lang.Comparable interface** for ***natural ordering***
  - *a*.compareTo(*b*)
    - *i* < 0 designates that *a* < *b*.
    - *i* = 0 designates that *a* = *b*.
    - *i* > 0 designates that *a* > *b*.
    - ***Lexicographic*** *for String class*

COMPARATOR ADT

- We may want to compare objects according to some notion
- other than their natural ordering
- A **comparator** encapsulates the action of comparing two objects according to a given total order relation
  - *Comparator* is an object that is external to the class of the keys it compares.
- When the priority queue needs to compare two keys, it uses its comparator
- Primary method
  - 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.

KEYS & TOTAL ORDER RELATIONS

- Keys in a priority queue can be arbitrary objects on which an linear ordering is defined
- Two distinct entries in a priority queue can have the same key

- Mathematical concept of total order relation $\leq$
  - Comparability property: either $x \leq y$ or $y \leq x$
  - Antisymmetric property: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

EXAMPLE COMPARATOR

- Ex> a comparator that evaluates strings based on their length

```
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

- In general and reusable form of a priority queue,
  - Allow a user to choose any key type and
  - Allow to send an appropriate comparator instance as a parameter to the priority queue constructor.
    - The priority queue use that comparator anytime it needs to compare two keys to each other
  - 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

```
/** An abstract base class to assist implementations of the PriorityQueue interface.*/
public abstract class AbstractPriorityQueue<K,V>
                                    implements PriorityQueue<K,V> {
  //---------------- 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 CONT.

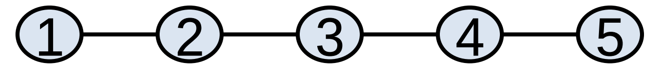
```
// instance variable for an AbstractPriorityQueue
/** The comparator defining the ordering of keys in the priority queue. */
private Comparator<K> comp;
/** Creates an empty priority queue using the given comparator to order keys. */
protected AbstractPriorityQueue(Comparator<K> c) { comp = c; }
/** 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( ));
}
/** 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; }
}
```

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

UNSORTED LIST IMPLEMENTATION

```java
/** 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;
  }
```

UNSORTED LIST IMPLEMENTATION, 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(); }
}
```

SORTED LIST IMPLEMENTATION

```
/** 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); }

  /** 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;
  }
```

SORTED LIST IMPLEMENTATION, 2

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

PRIORITY QUEUE SORTING "SCHEME"

- 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

```
/** Sorts sequence S, using initially empty priority queue P to produce the order. */
public static <E> void pqSort(PositionalList<E> S, PriorityQueue<E,?> P) {
  int n = S.size();
  for (int j=0; j < n; j++) {
    E element = S.remove(S.first());
    P.insert(element, null);        // element is key; null value
  }
  for (int j=0; j < n; j++) {
    E element = P.removeMin().getKey();
    S.addLast(element);             // the smallest key in P is next placed in S
  }
}
```

- The pqSort scheme is the paradigm of several popular sorting algorithms, including selection-sort, insertion-sort, and heap-sort

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

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

- 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)
	⋮	⋮	⋮
	(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)	()

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

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

  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