Priority Queues

Priority Queue ADT

- A priority queue stores a collection of entries
- Each entry is a pair

- 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(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

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	{ }

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

- Mathematical concept of total order(r=or linear 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 $y \leq z \Rightarrow x \leq z$

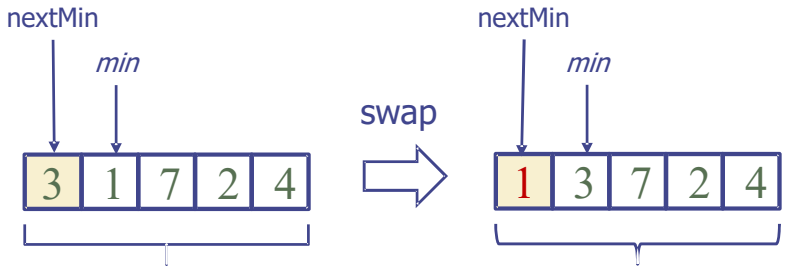
Entry ADT

- 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

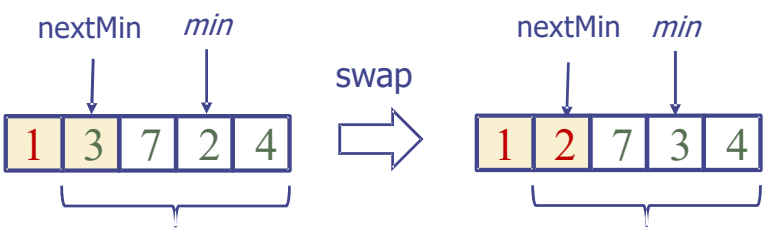
- As a Java interface:

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

Selection Sort

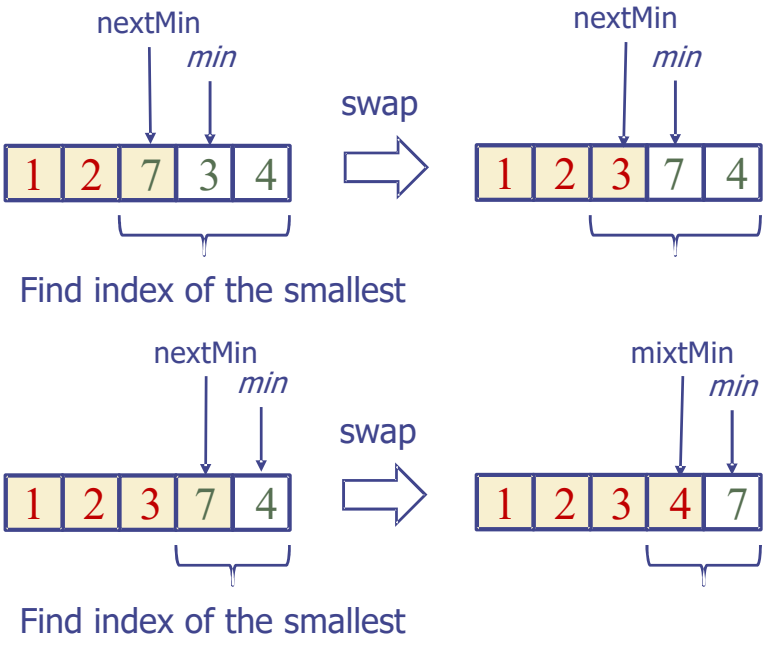


Find index of the smallest



Find index of the smallest

Selection Sort



Selection Sort

```
public static void sort(int[] xs) {
    for (int nextMin = 0; nextMin < xs.length - 1; nextMin++) {
        int min = nextMin;
        for (int i = nextMin + 1; i < xs.length; i++) {
            if (xs[i] < xs[min]) {
                min = i;
            }
        }
        swap(xs, nextMin, min);
    }
}

private static void swap(int[] xs, int i, int j) {
    int temp = xs[i];
    xs[i] = xs[j];
    xs[j] = temp;
}
```

Java.lang.Comparable

```
public interface Comparable<T> {
      int compareTo(T o)
}
```

- compareTo(o): returns an integer *i* such that
  - i < 0 if this < o,
  - i = 0 if this = o
  - i > 0 if this > o
  - An error occurs if "this" and "o" cannot be compared.


```
public class SelectionComparableSort {
    public static <T extends Comparable<T>> void sort(T[] xs) {
        for (int nextMin = 0; nextMin < xs.length - 1; nextMin++) {
            int min = nextMin;
            for (int i = nextMin + 1; i < xs.length; i++) {
                if (xs[i].compareTo(xs[min]) < 0){
                    min = i;
                }
            }
            swap(xs, nextMin, min);
        }
    }

    private static <T> void swap(T[] xs, int i, int j){
        T temp = xs[i]; xs[i] = xs[j]; xs[j] = temp;
    }

    public static void main(String[] args) {
       Integer[] xs = {4, 5, 7, 9, 3, 8, 2};
        sort(xs);
        System.out.println(Arrays.toString(xs));
    }
}
```

Point2D

```java
/**
 * Class representing a point in the plane with integer coordinates
 */
class Point2D {
    private int xc, yc; // coordinates

    public Point2D(int x, int y) {
        xc = x;
        yc = y;
    }

    public int getX() {
        return xc;
    }

    public int getY() {
        return yc;
    }
}
```


Solution 1.

```java
static <T extends Comparable<T>> void sort(T[] xs)
```

```java
class Point2D implements Comparable<Point2D> {
    private int x, y; // coordinates

    public Point2D(int x, int y) {
        this.x = x; this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    /**
     * Compare Point2Ds under the standard lexicographic order
     */
    @Override
    public int compareTo(Point2D other) {
        if (getX() != other.getX())
            return getX() - other.getX();
        else
            return getY() - other.getY();
    }
}
```

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.

Java.util.Comparator

```
public interface Comparator<T> {
     int compare(T o1, T o2)
}
```

- 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.

Selection Sort using Comparator

```
static <T> void sort(T[] xs, Comparator<T> comp) {
    for (int nextMin = 0; nextMin < xs.length - 1; nextMin++) {
        int min = nextMin;
        for (int i = nextMin + 1; i < xs.length; i++) {
            if (comp.compare(xs[i], xs[min]) < 0) {
                min = i;
            }
        }
        swap(xs, nextMin, min);
    }
}

static <T> void swap(T[] xs, int i, int j) {
    T temp = xs[i];
    xs[i] = xs[j];
    xs[j] = temp;
}
```


Solution 2.

```
class LexicoGraphic implements Comparator<Point2D> {
    @Override
    public int compare(Point2D o1, Point2D o2) {
        if (o1.getX() != o2.getX())
            return o1.getX() - o2.getX();
        else
            return o1.getY() - o2.getY();
    }
}

public static void main(String[] args) {
    Point2D[] xs = new Point2D[] {
        new Point2D(3, 5), new Point2D(5, 2),
        new Point2D(3, 3), new Point2D(7, 3)
    };

    sort(xs, new LexicoGraphic());

    System.out.println(Arrays.toString(xs));
```

```
class Point2D {
    private int xc, yc;
    public Point2D(int x, int y) {
        xc = x; yc = y;
    }
    public int getX() { return xc; }
    public int getY() { return yc; }
}
```

More on Comparators

```
Integer[] xs = {4, 5, 7, 9, 3, 8, 2};
sort(xs, Comparator.naturalOrder());    // xs must be Comparable
sort(xs, Comparator.reverseOrder());    // ditto
```

```
Point2D[] xs = new Point2D[] {
    new Point2D(3, 5), new Point2D(5, 2),
    new Point2D(2, 8), new Point2D(3, 3),
};

sort(xs, Comparator.naturalOrder()); // xs must be Comparable
sort(xs, Comparator.reverseOrder()); // ditto
```

안됨, 따로 comparator를 extends해서 implement 해주지 않으면 컴파일 에러 발생함

```
sort(xs, new LexicoGraphic().reversed());
sort(xs, comparingInt(Point2D::getY));
sort(xs, comparing(Point2D::getX).thenComparing(Point2D::getY));
```

sort(xs, comparing(Point2D::getX).thenComparing(Point2D::getY).reverse());

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

```
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
```

Unsorted List Implementation, 2

```
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 }
```

Sorted List Implementation

```
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
```

Sorted List Implementation, 2

```
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 }
```

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 ← priority queue with comparator *C*
  while ¬*S.isEmpty* ()
    e ← *S.remove*(*S.first* ())
    P.insert (*e*, ∅)
  while ¬*P.isEmpty*()
    e ← *P.removeMin*().*getKey*()
    S.addLast(*e*)

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 _____ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

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