Priority Queues

cs206

lec 19

April 7

Priority Queue

- A queue that maintains order of elements according to some priority
  - Removal order, not general order
    - the rest may or may not be sorted

- Priority Queues in real world
  - Homework
  - More generally, most work
    - key= f(due date, difficulty, annoyance)

Key

- Priority queues are ordered by some key, which may be:
  - derived from the data element
    - one field
    - combination of fields
  - independent of data element
    - for example: insertion time
  - Part (or all) of the data element
- best practice is make keys implement `Comparable` relation between keys using `compareTo`

Key-Value Pair

- Typically think of PQ as containing a pair
  - (Key, Value)
    - Key defines priority
    - Value is data the objects store
- KV pairs are frequently used
- Keys ideally:
  - are unique
    - how to handle duplicate keys?
  - have a natural ordering.
    - Using `compareTo` allows arbitrary comparisons
- Values need not be numerical or unique

PriorityQueue Interface

```
public interface QueueInterface<Q> {
    boolean isEmpty();
    int size();
    boolean offer(Q q);
    Q poll();
    Q peek();'
    // also add, remove, element
}
```

```
public interface PriorityQInterface<K extends Comparable<K>, V> {
    boolean isEmpty();
    int size();
    boolean offer(K key, V value);
    V poll();
    V peek();
}
```

PQ Implementation

- Questions:
  - How to store keys and values
    - handling of duplicate keys
  - Is the storage:
    - ordered?
    - size bound?
  - Can my PQ do MAX or MIN (or both)?
    - if BOTH, how?

Impelementation start

```
public class PriorityQueue<K extends Comparable<K>, V> implements PriorityQInterface<K,V> {

    enum Ordering { ASCENDING, DESCENDING, MIN, MAX}

    protected class Entry {
        /** Hold the key */
        final K theK;
        /** Hold the  value*/
        final V theV;
        /**
         * Create an Entry instance
         * @param kk the key
         * @param vv the value
         */
        public Entry(K kk, V vv) {
            theK = kk;
            theV = vv;
        }
    }

    final private ArrayList<Entry> pqStore;

    final private Ordering order;
```

Constructors & simple methods

```java
public PriorityQueue() {
    this(Ordering.MIN);
}
public PriorityQueue(Ordering order) {
    this.order=order;
    pqStore = new ArrayList<>();
}
@Override
public int size() {
    return pqStore.size();
}
@Override
public boolean isEmpty() {
    return pqStore.isEmpty();
}
@Override
public boolean offer(K newK, V newV) {
    Entry entry = new Entry(newK, newV);
    pqStore.add(entry);
    return true;
}
```

peek & poll

```
@Override
public V poll() {
    Entry entry = getNext();
    if (entry==null) return null;
    pqStore.remove(entry);
    return entry.theV;
}

@Override
public V peek() {
    Entry entry = getNext();
    if (entry==null) return null;
    return entry.theV;
}
```

getNext()

```java
private Entry getNext() {
        if (isEmpty())
            return null;
        Entry cmin = pqStore.get(0);
        for (int i=0; i<pqStore.size(); i++) {
            Entry curr = pqStore.get(i);
            switch (order) {
                case MIN:
                case ASCENDING:
                    if (curr.theK.compareTo(cmin.theK) < 0)
                        cmin=curr;
                    break;
                case MAX:
                case DESCENDING:
                default:
                if (curr.theK.compareTo(cmin.theK) > 0)
                        cmin=curr;
                    break;
            }
        }
        return cmin;
    }
```

Example

```
PriorityQueue<Integer, String> pq = new PriorityQueue<>(Ordering.MIN);
        pq.offer(1,"Jane");
        pq.offer(10,"WET");
        pq.offer(5, "WAS");
        System.out.println(pq.poll());
        System.out.println(pq.poll());
        System.out.println(pq.poll());
        System.out.println();

        pq = new PriorityQueue<>(Ordering.MAX);
        pq.offer(1,"Jane");
        pq.offer(10,"WET");
        pq.offer(5, "WAS");
        System.out.println(pq.poll());
        System.out.println(pq.poll());
        System.out.println(pq.poll());
```

Complexity Analysis
for this implementation

- size, isEmpty, offer:
  - O(1) — assuming ArrayList implementation
- peek, poll
  - O(n)

- Suppose keep underlying ArrayList sorted
  - size, isEmpty, peek, poll == O(1)
  - offer == O(n)

Mini-Homework

Recall from the stack lecture, the example:

Method	Return Value	Stack Comtents
push(5)	5	{5}
push(3)	3	{5, 3}
size()	2	{5, 3}
pop()	3	{5}
empty()	FALSE	{5}
pop()	5	{}
empty()	TRUE	{}
pop()	null	{}
push(7)	7	{7}
push(9)	9	{7,9}
peek()	9	{7,9}

Create a similar table for the priority queue implementation discussed in this lecture showing the effect of offer, peek and poll on the underlying data structure and the return values for each. Be slightly more clever than reusing "Jane was wet". The complete code is on the website to aid in this endeavor.