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
Key Value Pairs

- Priority Queues are usually described as being on Key-Value Pairs
  - akin to Hashtables
- Priority queues are ordered by the key, which may be:
  - derived from the Value element (which may be a large, complex object)
    - one field
    - combination of fields
  - independent of Value element
    - for example: insertion time
- best practice is make keys implement Comparable relation between keys using compareTo
- Keys ideally:
  - are unique
    - how to handle duplicate keys?
  - have a natural ordering.
    - Contrast to hashtables in which key ordering is irrelevant
Priority Queues in real world

- Homework
  - $\text{key} = f(\text{due date, difficulty, annoyance})$
- Others items in priority queues
  - what is the key?
PriorityQueue Interface

```java
public interface QueueInterface<Q> {
    boolean isEmpty();
    int size();
    boolean offer(Q q);
    Q poll();
    Q peek();
}

public interface PriorityQInterface<K extends Comparable<K>, V> {
    boolean isEmpty();
    int size();
    boolean offer(K key, V value);
    V poll();
    V peek();
}
```
public abstract class AbstractPriorityQueue <K extends Comparable<K>, V> implements PriorityQInterface<K,V> {
    enum Ordering { ASCENDING, DESCENDING }
    protected Ordering order;
    protected class Pair<L extends Comparable<L>, W> implements Comparable<Pair<L,W>> {
        /** Hold the key */
        final L theK;
        /** Hold the value*/
        final W theV;
        public Pair(L kk, W vv) {
            theK = kk;
            theV = vv;
        }
        @Override
        public int compareTo(Pair<L, W> o) {
            if (Ordering.ASCENDING == order || Ordering.MIN==order)
                return theK.compareTo(o.theK);
            return o.theK.compareTo(theK);
        }
        public String toString() {
            return "{{"+theK+" "+theV+"}}";
        }
    }
}
PQ Implementation

• Questions:
  • How to store keys and values
    • handling of duplicate keys
  • Is the storage:
    • ordered?
    • size bound?
public class PriorityQueue<K extends Comparable<K>, V> extends AbstractPriorityQueue<K,V> {
    private static int CAPACITY = 200;
    private Pair<K, V>[] pqStore;
    private int size;
    public PriorityQueue() {
        this(Ordering.MIN);
    }
    public PriorityQueue(Ordering order) {
        this.order = order;
        pqStore = (Pair<K, V>[]) new Pair[CAPACITY];
        this.size = 0;
    }
    public int size() {
        return size;
    }
    public boolean isEmpty() {
        return size == 0;
    }
    public boolean offer(K newK, V newV) {
        if (size == CAPACITY)
            return false;
        Pair<K, V> entry = new Pair<>(newK, newV);
        pqStore[size] = entry;
        size++;
        return true;
    }
}
peek & poll

```java
public V peek() {
    int lmin = getNext();
    if (lmin<0)
        return null;
    Pair<K, V> entry = pqStore[lmin];
    if (entry==null) return null;
    return entry.theV;
}
```

```java
public V poll() {
    if (size==0) return null;
    if (size==1) {
        Pair<K, V> entry = pqStore[0];
        pqStore[0]=null;
        size=0;
        return entry.theV;
    }
    int lmin = getNext();
    Pair<K, V> entry = pqStore[lmin];
    remove(lmin);
    size--;
    return entry.theV;
}
```
getNext(), remove(lmin)

write them.
Example

```java
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());
```
public class PriorityQueueSAL<K extends Comparable<K>, V> extends AbstractPriorityQueue<K,V> {
    final private SAL<Pair<K,V>> pqStore;
    public PriorityQueueSAL() { this(Ordering.ASCENDING); }
    public PriorityQueueSAL(Ordering order) {
        this.order = order;
        pqStore = new SAL<SAL<Ordering.DESCENDING>>();
    }
    public int size() {
        return pqStore.size();
    }
    public boolean isEmpty() {
        return pqStore.isEmpty();
    }
    public boolean offer(K newK, V newV) {
        pqStore.add(new Pair<>(newK, newV));
        return true; // Note that this always succeeds, so always return true.
    }
    public V poll() {
        if (isEmpty())
            return null;
        Pair<K,V> p = pqStore.getAndRemove(pqStore.size()-1);
        return p.theV;
    }
}
Binary Heap

- A heap is a “binary tree” storing keys at its nodes and satisfying:
  - heap-order: for every internal node \( v \) other than root, \( \text{key}(v) \geq \text{key}(\text{parent}(v)) \)
  - “complete binary tree”: let \( h \) be the height of the heap
    - Heap is filled from top down and within a level from left to right.
      - at depth \( h \), the leaf nodes are in the leftmost positions
      - last node of a heap is the rightmost node of max depth
Height of a Heap

- A binary heap storing $n$ keys has a height of $O(\log_2 n)$
Insertion into a Heap

- Insert as new last node
- Need to restore heap order

![Diagram of heap insertion](image-url)
Upheap

- Restore heap order
  - swap upwards
  - stop when finding a smaller parent
  - or reach root

- $O(\log n)$
public class PriorityQHeap<K extends Comparable<K>, V> implements PriorityQInterface<K, V> {

    /** The default size of the heap. This corresponds to a max depth or 10. */
    private static final int CAPACITY = 1032;
    /** The array that holds the heap. */
    private Pair<K, V>[] backArray;
    /** The number of items actually in the heap. */
    private int size;
    /** The way in which the heap is ordered */
    final private Ordering order;

    public PriorityQHeap() {
        this(Ordering.MIN, CAPACITY);
    }  

    public PriorityQHeap(Ordering order, int capacity) {
        this.order=order;
        backArray = new Pair[capacity];
    }
```java
public boolean offer(K key, V value) {
    if (size>(backArray.length-1))
        return false; // no space in the array
    // put in at end
    int loc = size++;
    backArray[loc] = new Pair<K, V>(key, value);
    // up heap
    int upp = (loc-1)/2;
    while (loc!=0) {
        if (0 > backArray[loc].doCompare(backArray[upp])) {
            // swap and climb
            Pair<K, V> tmp = backArray[upp];
            backArray[upp] = backArray[loc];
            backArray[loc] = tmp;
            loc = upp;
            upp = (loc-1)/2;
        }
        else {
            break;
        }
    }
    return true;
}
```
Poll

- Removing the root of the heap
  - Replace root with last node
  - Remove last node $w$
  - Restore heap order

![Diagram of heap operations](image)
Downheap

• Restore heap order
  ▫ swap downwards
  ▫ swap with smaller child
  ▫ stop when finding larger children
  ▫ or reach a leaf

• $O(\log n)$
```java
@Override
class PeekAndPoll {
    public V poll() {
        if (isEmpty())
            return null;
        Entry<K, V> tmp = backArray[0];
        removeTop();
        return tmp.theV;
    }

    @Override
    public V peek() {
        if (isEmpty())
            return null;
        return backArray[0].theV;
    }
}
```
Remove head item from Heap

```java
private void removeTop()
{
    backArray[0] = backArray[size-1];
    backArray[size-1]=null;
    size--;
    int upp=0;
    while (true)
    {
        int dwn;
        int dwn1 = upp*2+1;
        if (dwn1>size) break;
        int dwn2 = upp*2+2;
        if (dwn2>size) {
            dwn=dwn1;
        } else {
            int cmp = backArray[dwn1].compareTo(backArray[dwn2]);
            if (cmp<=0) dwn=dwn1;
            else dwn=dwn2;
        }
        if (0 > backArray[dwn].compareTo(backArray[upp]))
        {
            Pair<K,V> tmp = backArray[dwn];
            backArray[dwn] = backArray[upp];
            backArray[upp] = tmp;
            upp=dwn;
        } else {
            break;
        }
    }
}
```
# Complexity Analysis

<table>
<thead>
<tr>
<th></th>
<th>Unordered</th>
<th>Ordered (using SAL)</th>
<th>Heap Based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>offer</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>peek</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>poll</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
General Removal

- swap with last node
- delete last node
- may need to upheap or downheap

Heap:

- 1
- 5
- 9
- 11
- 17
- 21
- 6
- 8
- 15
- 17
- 19
- 9
- 11
- 33
- 22
- 27

delete this node
delete this node
# Complexity Analysis

<table>
<thead>
<tr>
<th></th>
<th>Unordered</th>
<th>Ordered (using SAL)</th>
<th>Heap Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>offer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peek</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>poll</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Removal</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>