Priority Queues
Abstract Classes

cs151
Abstract classes

• A class that should/can NEVER be instantiated.
  • From the Pets example
    • Pet, Dog should be defined as abstract classes
      • The only instances of each of these should be from more specific classes.
      • In taxonomy kingdom, phylum or division, class, order, family, and genus should all be abstract
        • only species should have instance
# Abstract Classes

## Pt 2

<table>
<thead>
<tr>
<th></th>
<th>Interface</th>
<th>Abstract Class</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>stub methods</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>full methods</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Instance Variables</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Multiple inheritance from</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Instantiatable</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Has Constructors</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>May implement interfaces</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>May extend classes</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
public abstract class AbCl {
    private double km;

    public double getKM () {
        return km;
    }

    public double getMiles() {
        return km * 1.62;
    }

    /**
     * A really long comment so that implementers know exactly what to do
     * @param aaa
     * @param bbb
     */
    public abstract void populate(int aaa, int bbb);
}
Priority Queue

• A queue that maintains order of elements according to some priority

• Contrast to Queue which is FiFo

• **Priority Queues are about the order in which things are removed, NOT the way in which they are stored.**
  
  • the items may or may not be sorted, or otherwise arranged.

  • This statement applies to stack and queues also, it is just convenient in those cases to arrange data to make retrieval easy
public abstract class AbstractPriorityQueue <K extends Comparable<K>, V> implements PriorityQInterface<K,V> {

    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;
        /**
         * Create an Entry instance
         * @param kk the key
         * @param vv the value
         */
        public Pair(L kk, W vv) {
            theK = kk;
            theV = vv;
        }
        @Override
        public int compareTo(AbstractPriorityQueue<K, V>.Pair<L,W> o) {
            return theK.compareTo(o.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> {
   /** Default capacity */
   private static int CAPACITY = 200;
   private Pair<K, V>[] pqStore;
   /** The number of items in the priority queue */
   private int size;
   public PriorityQueue() {
      this(CAPACITY);
   }
   @SuppressWarnings("unchecked")
   /**
    * Return an array list of the given capacity
    * @param initialCapacity -- the capacity
    */
   public PriorityQueue(int initialCapacity) {
      pqStore = (Pair<K, V>[]) new Pair[initialCapacity];
      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;
   }
}

(Internally Unordered)
Priority Q
public V peek() {
    if (isEmpty())
        return null;
    int lmin = getNext();
    Pair<K,V> entry = pqStore[lmin];
    return entry.theV;
}

public V poll() {
    if (isEmpty())
        return null;
    int lmin = getNext();
    Pair<K,V> entry = pqStore[lmin];
    remove(lmin);
    return entry.theV;
}
getNext(), remove(lmin)

write them.
```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());

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;
    }
}

(Internally Ordered)
Priority Q
# 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>O(1)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td><strong>peek</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><strong>poll</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td></td>
</tr>
</tbody>
</table>

Unordered PQ == Selection Sort
Ordered PQ = Insertion Sort
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, $key(v) \geq key(parent(v))$
  ▫ 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
Upheap

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

- $O(\log n)$
Poll

• Removing the root of the heap
  ▫ Replace root with last node
  ▫ Remove last node
  ▫ Restore heap order
Downheap

- Restore heap order
  - swap downwards
  - swap with smaller child
  - stop when finding larger children
  - or reach a leaf
- \( O(\log n) \)
Heaps are built on Arrays

- Parent from child
  - suppose child is at location childLoc in array
    - parentLoc = (childLoc-1)/2
- Child from Parent
  - suppose parent is at parentLoc in array
    - leftChild = parentLoc*2+1
    - rightChild = parentLoc*2+2

**Locations of Parents and children are in strict mathematical relationship**

- Parent from child
  - child at loc 4 (value 7)
    - parent is at (4-1)/2 = 1 (value 5)
- Child from Parent
  - parent at loc 2 (value 6)
    - leftChild =2*2+1 = 5 (value 1)
    - rightChild = 2*2+2 = 6 (value — not used)
Priority Queue using Heaps
startup

```java
public class PriorityQHeap<K extends Comparable<K>, V extends AbstractPriorityQueue<K, V>> {

    private static final int CAPACITY = 1032;
    private Pair<K, V>[] backArray;
    private int size;

    public PriorityQHeap() {
        this(CAPACITY);
    }

    public PriorityQHeap(int capacity) {
        size=0;
        backArray = new Pair[capacity];
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean isEmpty() {
        return size==0;
    }
}
```
Heap Insertion
Priority Queue offer method

```java
public boolean offer(K key, V value) {
    if (size >= (backArray.length - 1))
        return false;
    // put new item in at end data items
    int loc = size++;
    backArray[loc] = new Pair<K, V>(key, value);
    // up heap
    int upp = (loc - 1) / 2; // the location of the parent
    while (loc != 0) {
        if (0 > backArray[loc].compareTo(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;
}
```
@Override
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;
}
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; }
    }
}
General Removal

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