Name: Nanako Chung
Net ID (the one in your email): nsc309

Answer each question on a separate page.

As you enter your answers, make sure that the numbering of questions does not change and that each part starts on a new page.

Once you complete all the questions (or as many questions as you have time for before the end of the recitation), download the PDF version (File -> Download As -> PDF document) and submit it to Gradescope.

Make sure to match all pages to appropriate questions when you submit your project to Gradescope.

1. Both classes are generic classes. The PriorityQueue class uses a single generic specifier called E. The HeapPriorityQueue class uses two generic specifiers called K and V. Discuss what each of the generic specifiers represent. Explain, in your own words, why the Open JDK implementation uses only one generic type and the book's class uses two.

The single generic specifier E in the PriorityQueue class represents the type of the elements that will be stored in the priority queue. The two generic specifiers called K and V in the HeapPriorityQueue class represent the type of the keys and the type of the values respectively; the keys tell the values how to order themselves (for example, if the keys were of Age and the values were of type Person, this means that the the Person values should be ordered based on each Person's associated Age).

The Open JDK implementation uses only one generic type because, although both classes rely on the comparator to order the values of the queue, the Open JDK implementation relies only on the comparator. On the other hand, the book's class uses two generic element types because the comparator orders the keys, which are then used to order the values in the queue. In both cases, the comparator is optional to use (if it is null, the order of the keys will be natural).

2. How are the elements stored in each of the implementations. Describe the type of the container used, type of the elements stored and how the size is handled.

In the PriorityQueue class, the elements are stored using the initFromPriorityQueue() method, the initElementsFromCollection() method, or the initFromCollection() method (all of which have private data fields) depending on the type of constructor called. The type of the container used is an array of Objects called queue, which is similar to the array implementation of a queue we created in Project 3. The type of elements stored is E. The size is a private data field that is initially set to 0 and then is incremented each time an element is added and decremented each time an element is removed. If the size exceeds capacity, then the private grow() method is called to create an array with a larger capacity.

In the HeapPriorityQueue class, the elements are stored using the add() method within the ArrayList class. The type of the container used is an ArrayList with elements of type Entry<K,V>. This means that the elements within the priority queue are key-value pairs. The size is handled/obtained through the size() method within the ArrayList class. The ArrayList class also handles the situation of the size exceeding the capacity.

3. Which methods of the PriorityQueue class may be called when adding an element to an object of type PriorityQueue (list all the methods that are called)? Briefly describe what each of these method calls accomplishes (use your own words, do not copy verbatim from the documentation).

The PriorityQueue class's **add(E e)** method, which adds an element to an object of type PriorityQueue, calls:

- **offer(E e)** - this method checks the element e and, if valid, inserts it into the heap
- **grow(int minCapacity)** - this private method is called if the size is equal to or exceeds queue.length (the capacity of the queue) and acts as the equivalent of the makeLarger() method in Project 3; it creates an array with the same elements but a larger capacity; if the queue is empty, the element is simply added to the zeroth index
- **siftUp(int k, E x)** - this private method is called if the size of the current array called is less than the capacity; it inserts element x into position k while following the heap properties
- **siftUpUsingComparator(int k, E x)** - this private method is called if the comparator is not null, which means the elements are not sorted in natural ordering; it uses the compare() method to place the element x in the correct order in the priority queue
- **siftUpUsingComparable(int k, E x)** - this private method is called if the comparator is null, which means the elements are to be placed in natural ordering; it uses the compareTo() method to place the element x in the correct order in the priority queue

4. Which methods of the HeapPriorityQueue and the ArrayList classes may be called when adding an element to an object of type HeapPriorityQueue (list all the methods that are called)? Briefly describe what each of these method calls accomplishes (use your own words, do not copy verbatim from the documentation).

The HeapPriorityQueue class's **insert(K key, V value)** method, which adds an element to an object of type HeapPriorityQueue, calls:

- **checkKey(key)** - this method checks to see if the key is valid
- **add(Object o)** - this method is called from the ArrayList class, which adds a new element to the ArrayList
- **upheap(int j)** - this method works to move the newest entry up the heap while maintaining the heap properties
- **swap(int i, int j)** - this method is called if a swap between two elements is necessary to fulfill this property; switches the two elements at the specified indices
- **set(int index, Object o)** - this method is called from the ArrayList class, which replaces the current object at the index with o
- **get(int index)** - this method is called from the ArrayList class, which returns the element at the requested index
- **size()** - this method is called from the ArrayList class, which returns the size of the ArrayList

5. Which methods of the PriorityQueue class may be called when removing an element from an object of type PriorityQueue (list all the methods that are called)? Briefly describe what each of these method calls accomplishes (use your own words, do not copy verbatim from the documentation).

The PriorityQueue class's **poll()** method, which is called to remove an element from an object of type PriorityQueue, calls:

- **siftDown(int k, E x)** - this private method places element x at the kth index while upholding the heap properties
- **siftDownUsingComparator(int k, E x)** - this private method is called if the comparator is not null; it removes the element at the k-th index and then adjusts accordingly to satisfy the heap properties using the compare() method (compares to comparator; heap is not in natural ordering)
- **siftDownUsingComparable(int k, E x)** - this private method is called if the comparator is null; it removes the element at the k-th index and then adjusts accordingly to satisfy the heap properties using the compareTo() method (for natural ordering)

6. Which methods of the HeapPriorityQueue and the ArrayList classes may be called when removing an element to an object of type HeapPriorityQueue (list all the methods that are called)? Briefly describe what each of these method calls accomplishes (use your own words, do not copy verbatim from the documentation). For each method specify which class it implemented in.

The HeapPriorityQueue class's **removeMin()** method, which is called to remove an element of an object of type HeapPriorityQueue, calls:

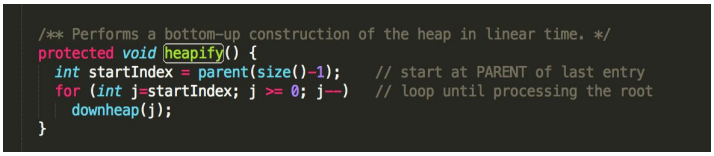
- **isEmpty()** - this method is called from the ArrayList class and checks to see if the queue is empty
- **get(int index)** - this method is called from the ArrayList class, which returns the element at the requested index
- **swap(int i, int j)** - called if a swap between two elements is necessary to fulfill this property; switches the two elements at the specified indices
- **remove(Object o)** - this method within the ArrayList class is called to remove the minimum element from the list
- **downheap(int j)** - this method is called to move the new root down to restore the heap order property
- **parent(int j)** - this method returns the position of the parent of the element at index j
- **left(int j)** - this method returns the position of the left child of the element at index j
- **right(int j)** - this method returns the position of the right child of the element at index j
- **hasLeft(int j)** - this method confirms the existence of the left child of the element at index j
- **hasRight(int j)** - this method confirms the existence of the right child of the element at index j
- **size()** - this method is called from the ArrayList class, which returns the size of the ArrayList

7. The HeapPriorityQueue class has a two parameter constructor as follows:
public HeapPriorityQueue(K[] keys, V[] values)
Consider the code fragment that uses the HeapPriorityQueue class in the question.
Rewrite the main() method so that it performs the same task as the one above, but uses the PriorityQueue class from OpenJDK. Note: you may need to implement additional classes.

```
/**
  * main method
  */
public static void main (String [] args ) {
        String [] words = {"hello", "goodbye", "bye", "good morning", "good evening", "good
        afternoon" };

        //creates an object of type PriorityQueue that takes in String elements
        PriorityQueue<String> pq = new PriorityQueue<String>(new CompareStrings);

        //initializes pq (a heap) with the Strings in the word array
        for (int i=0; i<words.length; i++) {
                pq.add(words[i]);
        }

        //removes the root of the heap and prints the removed root's word with its length
        String e="";
        while (!pq.isEmpty()) {
                e = pq.poll();
                System.out.println(e.length()+", "+e);
        }
}

 /**
  * this class implements the Comparator interface and overrides the compare() method
  */
public class ComparingStrings implements Comparator {

        //overrides the compare method
        public int compare(String word, String word1) {

                //returns the larger string
                return Math.max(word.length, word1.length);
        }
}
```

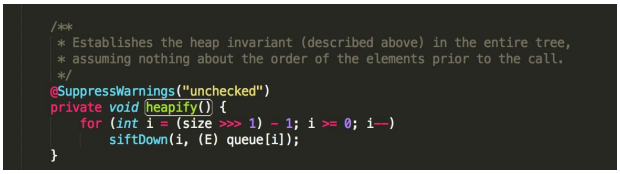
8. Both classes provide the void heapify() methods. Analyze both methods and determine if they perform the same task or not. If they do, establish correspondence of each step. If they do not, the explain to what extent the tasks overlap and how they differ.

HeapPriorityQueue class:



```
/** Performs a bottom-up construction of the heap in linear time. */
protected void heapify() {
  int startIndex = parent(size()-1);    // start at PARENT of last entry
  for (int j=startIndex; j >= 0; j--)   // loop until processing the root
    downheap(j);
}
```

PriorityQueue class:



```
/**
 * Establishes the heap invariant (described above) in the entire tree,
 * assuming nothing about the order of the elements prior to the call.
 */
@SuppressWarnings("unchecked")
private void heapify() {
    for (int i = (size >>> 1) - 1; i >= 0; i--)
        siftDown(i, (E) queue[i]);
}
```

The heapify() method in the HeapPriorityQueue class assumes that a heap with keys and values exists and thus assumes it is dealing with some ordered values. The heapify() method in the PriorityQueue class assumes that it is dealing with an unsorted array. Both methods create a heap using bottom-up construction. Overall, the two methods perform the same task. As seen above, both heapify() methods use a for loop that start at the parent of the last entry into the priority queue (size>>>1 is the same as the index of the parent) and iterate through it until the root is reached. Then, both heapify methods call a method (siftDown() in the PriorityQueue class and downheap() in the HeapPriorityQueue class) that “heapifies” down to restore the root correctly and abide by the heap properties.

9. Both classes use an interface called Comparator. Discuss, in your own words, what the purpose of this interface is, why it is implemented and its similarities and differences to the Comparable interface.

The purpose of the Comparator interface is to redefine a more flexible way to compare two objects; it is implemented so that the compare() method, which will be overridden in the class that implements it, can define specifically how to compare these two objects. The compare() method is typically used to sort things in a non-natural (or natural if desired) order easily.

In this way, the main similarity between the Comparator interface and the Comparable interface is that they are both interfaces that are implemented to compare two objects' references. They are also similar with regards to the PriorityQueue and HeapPriorityQueue classes in that both of their comparison methods return an int value.

The main difference, on the other hand, is that the Comparator's compare() method has a distinct definition of how to compare these two objects that may or may be the same as the Comparable interface's compareTo() method; that is, the compareTo() method compares based on the "natural order" of the Object's type (e.g. 'a' comes before 'c'), but the compare() method is ordered based on a single "key" element which does not necessarily have to follow the "natural order." Further, whereas Comparable interface has only the compareTo() method, the Comparator interface has two methods: equals() and compare(). Lastly, the Comparable interface is found in the java.lang package whereas the Comparator interface is found in java.util package.

10. Analyze and discuss the performance of the remove operation in both classes. Use the big-O notation to describe what the running time is. Explain why it is what you think it is. Make sure your discussion covers both implementations.

In the PriorityQueue class, the poll() method has a running time of O(logN) because it first removes the root and then relies on the siftDown() method, which searches and moves every node accordingly, to move it to maintain the heap properties. This siftDown() method takes O(logN), making the running time of poll() itself O(logN).

In the HeapPriorityQueue class, the removeMin() method has a running time of O(logN). This is because first the minimum value/root is swapped to the end and then removed, which takes O(logN) running time. Then, the downheap() method is called to adjust the new root that has taken its place, which is O(logN). Together, this makes the runtime O(logN).

In conclusion, both classes have remove operations that have a running time of O(logN).