Option 1: Insert

```
public static void insert() {
        try (BufferedReader reader = new BufferedReader(new
FileReader(inputFile))) {
            String line;

            while ((line = reader.readLine()) != null) {
                String[] fields = line.split(";");

                if (fields.length != 13) {
                    System.err.println("Invalid record format: " + line);
                    continue;
                }

                // Constructs record from line
                Record record = new Record(
                        fields[0], fields[1], fields[2], fields[3], fields[4],
                        fields[5], fields[6], fields[7], fields[8], fields[9],
                        fields[10], fields[11], fields[12]
                );

                // Inserts record into priority queue
                priorityQueue.insert(record);
            }
            System.out.println("Records successfully inserted into the
priority queue.");
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
        showMenu();
    }

public Record insert(Record record) throws IllegalArgumentException {
        if (size == queue.length) {
            resize(2 * queue.length); // Double the array size if it's full
        }
        queue[size] = record;
        size++;
        upheap(size - 1); // Maintain heap property
        return record;
    }
```

```
public void resize(int capacity) {
        Record[] newArray = new Record[capacity];
        for (int i = 0; i < size; i++) {
            newArray[i] = queue[i];
        }
        queue = newArray; // Update the array reference
    }

protected void upheap(int j) {
        while (j > 0) { // Continue until reaching root (or break statement)
            int p = parent(j);
            if (queue[j].compareTo(queue[p]) >= 0) {
                break; // Heap property verified
            }
            swap(j, p); // Swap the current node with its parent
            j = p; // Continue from the parent's location
        }
      }
```

Breakdown: insert() calls priorityQueue.insert() O(n) times, growing linearly based on how many records need to be inserted. priorityQueue.insert() calls resize() and upheap(). resize is costs O(n) across n insertions, amortizing it to O(1). Upheap has time complexity of O(logn) because it only needs to explore one branch of the tree, and swap is O(1).

Total: O(n) * O(1) * O(logn) * O(1) = **O(nlogn)**

Option 3: nextPatient

```
public static void nextPatient() {
        Record removedRecord = priorityQueue.removeMin();

        if (removedRecord == null) {
            System.out.println("The priority queue is empty. No patient to
remove.");
        } else {
            System.out.println("The patient removed from the heap is as
follows:");
            System.out.println(removedRecord);
        }

        showMenu();
    }
```

```
public Record removeMin() {
        if (isEmpty()) return null;

        Record minRecord = queue[0]; // The root is the min element
        queue[0] = queue[size - 1]; // Move the last element to the root
        queue[size - 1] = null; // Clear the last element
        size--;
        downheap(0); // Restore heap property
        return minRecord;
    }

protected void downheap(int j) {
        Record record = queue[j];
        while (hasLeft(j)) {
            int leftChild = left(j);
            int minChild = leftChild; // Start with the left child
            if (hasRight(j) && queue[leftChild].compareTo(queue[right(j)]) >
0) {
                minChild = right(j); // Choose the right child if it's smaller
            }
            if (record.compareTo(queue[minChild]) <= 0) break; // Heap
property satisfied
            swap(j, minChild); // Move the smaller child up
            j = minChild; // Continue down
        }
        queue[j] = record; // Place the entry at its correct position
    }
```

Breakdown: nextPatient() calls priorityQueue.removeMin() one time, and removeMin() calls downheap() after swapping the root element with the last element. So the time complexity of nextPatient() would be the complexity of downheap(). Because downheap() only needs to traverse one branch, it would be the same as upheap() which is:
O(logn)

Option 4: removePatient

```
public static void removePatient() {
        Record targetRecord = gatherPatientInfo();

        boolean found = priorityQueue.remove(targetRecord);

        if (found) {
```

```
            System.out.println("The requested patient's record has been
removed from the queue.");
        } else {
            System.out.println("The requested patient's record is not
found.");
        }

        showMenu();
    }

public boolean remove(Record targetRecord) {
        for (int i = 0; i < size; i++) {
            if (queue[i].equals(targetRecord)) {
                queue[i] = queue[size - 1]; // Replace with last element
                queue[size - 1] = null; // Clear last element
                size--;
                downheap(i); // Restore heap order
                return true; // Record found and removed
            }
        }
        return false; // No matching record found
    }
```

Breakdown: removePatient() calls gatherPatientInfo() and priorityQueue.remove(). gatherPatientInfo() just reads player input at complexity O(1). remove() linearly searches for the target record (O(n)), then it needs to perform downheap(), O(logn). downheap() shouldn't need to be called a linear amount of times, so in this case O(n) would be the dominating term.

Total: O(1) * (O(n) + O(logn)) = **O(n)**