```
class Heap {

    private Entry arr[];
    private int size;
    private boolean readData;

    public Heap() { // default constructor
        arr = new Entry[600]; // Standard array to hold entries. In a
real-world scenario, this would likely be a dynamically sized array
        size = 0;
        readData = false;
    }

    public boolean getRead() { // Gets the status of whether data has
already been read into the database
        return readData;
    }

    public void setRead(boolean read) { // Sets readData value
        this.readData = read;
    }

    public Entry peek() { // Returns the highest priority member of the
heap
        return arr[1];
    }

    public Entry next() { // Returns and removes the highest priority
entry
        Entry Entry = arr[1];
        arr[1] = arr[size];
        size--;
        this.heapify(1);
        return Entry;
    }

    public void remove(int index) { // Removes the entry at a specific
index and heapifies the heap
        arr[index] = arr[this.size];
        this.size--;
```

```
        for (int i = this.size / 2; i > 0; i--) {
            heapify(i);
        }
    }

    public int getSize() { // Returns number of entries
        return this.size;
    }

    public void heapify(int i) { // Reorders the heap starting at the
input index, and working downward to the leafs of the current branch
        int largest = i;

        if (hasLeft(i) && arr[largest].getKey() < arr[left(i)].getKey()) {
// Determines if the root or left child has a higher key
            largest = left(i);
        }
        if (hasRight(i) && arr[largest].getKey() < arr[right(i)].getKey())
{ // Determines if the current largest key or right child has a higher key
            largest = right(i);
        }

        if (largest != i) { // If the largest isn't the root, swap the
largest with the root
            swap(i, largest);
            heapify(largest); // Recursively reheapify the affected entry
and it's children
        }
    }

    public void add(Entry entry) { // Adds an entry to the heap
        size++;
        arr[size] = entry;
        for (int i = this.size / 2; i > 0; i--) { // Reheapifies every
non-leaf entry from bottom up to root entry
            heapify(i);
        }
    }
```

```
    public int contains(Patient patient) { // Compares a set of given info
to see if it matches a patient in the heap
        for (int i = 1; i <= size; i++) {
            if (arr[i].getPatient().equals(patient)) {
                return i;
            }
        }
        return -1;
    }

    public Entry getEntry(int i) { // Returns the entry at a given index
        return arr[i];
    }

    public void updatePriority(int i, String unosStatus) { // Updates the
priority of a patient, and stores the changes into the change history for
that patient

this.arr[i].getPatient().addPastStatus(this.arr[i].getPatient().getUnosSta
tus());

this.arr[i].getPatient().addPastStatusDate(java.time.LocalDate.now().toStr
ing());
        this.arr[i].getPatient().setUnosStatus(unosStatus);
        this.arr[i].updatePriority();
        for (int j = this.size / 2; j > 0; j--) { // Reheapifies all
non-leaf entries to fix any potential violations after a patients status
is updated
            heapify(j);
        }
    }

    private void swap(int i, int j) { // Helper method for heapify, swaps
entries at the given indices
        Entry temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
```

```
    private int parent(int i) { // Returns parent of the entry at a given
index
        return (i / 2);
    }

    private int left(int i) { // Returns left child of the entry at a
given index
        return (2*i);
    }

    private int right(int i) { // Returns right child of the entry at a
given index
        return ((2*i)+1);
    }

    private boolean hasLeft(int i) { // Returns a bool of whether an entry
at a given index has a left child
        if (left(i) > this.size) {
            return false;
        } else {
            return true;
        }
    }

    private boolean hasRight(int i) { // Returns a bool of whether an
entry at a given index has a right child
        if (right(i) > this.size) {
            return false;
        } else {
            return true;
        }
    }
}
```