a)

```
Algorithm toggle()
output: transform the min-heap to max-heap and vice versa.
for i←(heap.size()-2)/2 to 0 do
        convert(i)
```

```
Algorithm convert(i)
input: the index of the current node
output: according to the sorting rules, change the position of current entry
most←i
if hasLeft(i) ∧ compare(heap.get(left(i)), heap.get(most) < 0) then
        most = left(i)
if hasRight(i) ∧ compare(heap.get(right(i)), heap.get(most) < 0) then
                most = left(i)
if most != i ∧ hasParent(hasChildren(i)) then
        swap(i, most)
        convert(most)
```

```
Algorithm compare(e1, e2)
input: two entries
output: an integer representing the comparison of two entries
if this is a max-heap then
        return compareTo(e1.getK, e2.getV)
else
        return compareTo(e2.getKey, e1.getKey)
```

```
Algorithm remove(e)
input: an entry you want to remove
output: the removed entry
i←e.getIndex
if i=heap.size-1 then
        heap.remove(heap.size()-1)
else
        swap(I, heap.size()-1)
        heap.remove(heap.size()-1)
        bubble(j)
return e
```

Alogrithm replanceKey(e, newk)
input: an entry and a new key
output: the old key
old←e.getKey
e.setKey(newk)
bubble(e.getIndex())
return old

Alorithm replaceValue(e, newV)
input: an entry and a new value
output: the old value
old←e.getValue
e.setValue(newV)
return old

Algorithm bubble(i)
input: the index of the entry
output: restore the heap property according the comparisons
if compare(heap.get(i), heap.get(parent(i))) < 0 then
    upheap(i)
else
    downheap(i)

b)

Since the min-heap and max-heap are identical except the sorting regulations, all the analyses are based on min-heap.

For the toggle() method: O(n)

Assume it is an AFPQ using min-heap now, and we are going to convert it to the format of max-heap. Firstly, we compare the key of the last internal node (i.e., bottommost and rightmost internal node) with the keys of its children nodes to find the entry with the greatest key. Then, we exchange these two entries. Since this is a min-heap, the key of each internal node is smaller than all the keys of its descendants, the times of exchanges are equal to the number of internal nodes $\frac{n}{2}$. Similarly, from the second row from the bottom, these downward-exchanged entries will compare with their current descendants at a one-by-one level.

$$T(n) = \frac{n}{2} + \frac{n}{2^3} \times 1 + \frac{n}{2^4} \times 2 + \frac{n}{2^5} \times 3 + \cdots + \frac{n}{2^k}(k-2) + \cdots + \frac{n}{n}(\log n - 2)$$

$$= \frac{n}{2} + n\left(\frac{1}{2^3} + \frac{2}{2^4} + \frac{3}{2^5} + \cdots + \frac{k-2}{2^k}\right)$$
$$< \frac{n}{2} + n\left(\frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + \cdots + \frac{k}{2^k}\right) \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (C)$$

Assume:

$$S_k = \frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + \cdots + \frac{k-2}{2^{k-2}} + \frac{k-1}{2^{k-1}} + \frac{k}{2^k} \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (A)$$

$$2S_k = \frac{3}{2^2} + \frac{4}{2^3} + \frac{5}{2^4} + \cdots + \frac{k-2}{2^{k-3}} + \frac{k-1}{2^{k-2}} + \frac{k}{2^{k-1}} \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (B)$$

$$(A) - (B) = S_k = \frac{3}{2^2} + \left(\frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \cdots + \frac{1}{2^{k-1}}\right) - \frac{k}{2^k}$$

by Geometric Summation

$$S_k = \frac{3}{4} + \left(\frac{\frac{1}{8}\left(1 - \frac{1}{2^{k-3}}\right)}{1 - \frac{1}{2}}\right) - \frac{k}{2^k} = \frac{3}{4} + \frac{1}{4} - \frac{1}{2^k} - \frac{k}{2^k} = 1 - \frac{k+1}{2^k}$$

$$\lim_{k\to\infty}(1 - \frac{k+1}{2^k}) = 1 \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (D)$$

plug (D) $into$ (C)

$$\mathrm{T(n)} = \frac{n}{2} + n = \frac{3n}{2}; \text{ this is, the time complexity of toggle is O(n)}$$

For the remove(e) method: O(logn)
Since we exchange the passing entry with the last entry, then do the bubble sort. If we consider the worst condition—remove the top element, the time complexity is O(height); this is, O(logn).

For the replaceKey(e, k) method: O(logn)
When replacing the key of the given entry, the worst condition is changing the key of top entry to the maximal; therefor, the time complexity is O(height); this is, O(logn).

For replaceValue(e, v) method: O(1)
As we obtain the position of the given entry by the index straightforward and the heap property will not be influenced, the time complexity is O(1).