**A.**

1. **replaceKey(e, k)**

**Algorithm replaceKey(e, k)**

**Input:** e, an Entry object; k, a new key to replace the existing key of e

**Output:** oldKey, the old key of the entry before it was updated

index 🡨 -1

for i from 0 to size - 1 do

if elements[i] equals e then

index 🡨 i

break

if index is -1 then

signal an error "Entry not found in the priority queue"

oldKey 🡨 elements[index].getKey()

elements[index].setKey(k)

if state (Min heap) then

if k < oldKey then

bubbleUp(index)

else

bubbleDown(index)

else (Max heap)

if k > oldKey then

bubbleUp(index)

else

bubbleDown(index)

return oldKey

**=====HELPER ALGORITHMS FOR replaceKey(e,k) =====**

**Algorithm bubbleUp(index)**

**Input:** index, the index of the entry to bubble up

**Output:** None

while index > 0 do

parentIndex 🡨 getParentIndex(index)

if (state is Min heap and elements[index].getKey() < elements[parentIndex].getKey()) or (state is Max heap and elements[index].getKey() > elements[parentIndex].getKey()) then

swap(elements, index, parentIndex)

index 🡨 parentIndex

else

break

**Algorithm bubbleDown(index)**

**Input:** index, the index from which to start the bubble down operation

**Output:** None

while true do

left 🡨 getLeftChildIndex(index)

right 🡨 getRightChildIndex(index)

smallestOrLargest 🡨 index

if state is Min heap then

if left < size and elements[left].getKey() <

elements[smallestOrLargest].getKey() then

smallestOrLargest 🡨 left

if right < size and elements[right].getKey() <

elements[smallestOrLargest].getKey() then

smallestOrLargest 🡨 right

else (Max heap)

if left < size and elements[left].getKey() > elements[smallestOrLargest].getKey() then

smallestOrLargest 🡨 left

if right < size and elements[right].getKey() > elements[smallestOrLargest].getKey() then

smallestOrLargest 🡨 right

if smallestOrLargest equals index then

break

swap(elements, index, smallestOrLargest)

index 🡨 smallestOrLargest

1. **state()**

**Algorithm state()**

**Input:** None

**Output:** A string indicating the current heap mode

if state is true then

    return "Min heap"

else

    return "Max heap"

1. **peekAt(n)**

**Algorithm peekAt(n)**

**Input:** n, index of the entry to retrieve

**Output:** the nth entry in the priority queue according to the current ordering (Min or Max heap)

if n < 0 or n >= size then

signal an error "Index out of bounds”

tempElements 🡨 new Entry[size]

for i from 0 to size - 1 do

tempElements[i]🡨 elements[i]

return quickSelect(tempElements, 0, size - 1, n)

**=====HELPER ALGORITHMS FOR peekAt(n) =====**

**Algorithm quickSelect(arr, left, right, n)**

**Input:** arr, an array of entries; left, right, the range within the array; n, the index of the element to find

**Output:** the nth element

if left equals right then

return arr[left]

while true do

pivotIndex 🡨 medianOfMedians(arr, left, right)

pivotIndex 🡨 partition(arr, left, right, pivotIndex)

if n equals pivotIndex then

return arr[n]

else if n < pivotIndex then

right 🡨 pivotIndex - 1

else

left 🡨 pivotIndex + 1

**Algorithm medianOfMedians(arr, left, right)**

**Input:** arr, an array of entries; left, right, indices defining the subarray

**Output:** an approximate median index

if right - left + 1 <= 5 then

return findMedianIndex(arr, left, right)

numMedians 🡨 (right - left + 4) / 5

for i from 0 to numMedians - 1 do

subLeft 🡨 left + i \* 5

subRight 🡨 min(subLeft + 4, right)

medianIndex 🡨 findMedianIndex(arr, subLeft, subRight)

swap(arr, left + i, medianIndex)

mid 🡨 left + (numMedians - 1) / 2

return medianOfMedians(arr, left, left + numMedians - 1, mid)

**Algorithm findMedianIndex(arr, left, right)**

**Input:** arr, an array of Entry objects; left, right, indices defining the subarray

**Output:** the index of the median within the subarray

for i from left + 1 to right do

j 🡨 i

while j > left and ((state is true and arr[j].getKey() < arr[j - 1].getKey()) or

(state is false and arr[j].getKey() > arr[j - 1].getKey())) do

swap(arr, j, j - 1)

j 🡨 j - 1

return (left + right) / 2

**Algorithm partition(arr, left, right, pivotIndex)**

**Input:** arr, an array of entries; left, right, indices defining the subarray; pivotIndex, the index of the pivot

**Output:** the final index of the pivot

pivot 🡨 arr[pivotIndex]

swap(arr, pivotIndex, right)

storeIndex 🡨 left

for i from left to right - 1 do

if (state is Min heap and arr[i].getKey() < pivot.getKey()) or (state is Max heap and arr[i].getKey() > pivot.getKey()) then

swap(arr, storeIndex, i)

storeIndex 🡨 storeIndex + 1

swap(arr, storeIndex, right)

return storeIndex

1. **merge(otherAPQ)**

**Algorithm merge(otherAPQ)**

**Input:** otherAPQ, another AdvancedPriorityQueue object to merge with

**Output:** None

newSize 🡨 this.size + otherAPQ.size

if newSize > elements.length then

newElements 🡨 new Entry[newSize]

for i from 0 to this.size - 1 do

newElements[i] 🡨 this.elements[i]

elements 🡨 newElements

for i from 0 to otherAPQ.size - 1 do

elements[this.size + i] 🡨 otherAPQ.elements[i]

this.size 🡨 newSize

buildHeap()

**Algorithm buildHeap()**

**Input:** None

**Output:** None

for i from getParentIndex(size - 1) down to 0 do

bubbleDown(i)

**Algorithm bubbleDown(index)**

**Input:** index, the starting index to bubble down from

**Output:** None

while true do

left 🡨 getLeftChildIndex(index)

right 🡨 getRightChildIndex(index)

largestOrSmallest 🡨 index

if left < size and ((state is true and elements[left].getKey() < elements[largestOrSmallest].getKey()) or (state is false and elements[left].getKey() > elements[largestOrSmallest].getKey())) then

largestOrSmallest 🡨 left

if right < size and ((state is true and elements[right].getKey() < elements[largestOrSmallest].getKey()) or

(state is false and elements[right].getKey() > elements[largestOrSmallest].getKey())) then

largestOrSmallest 🡨 right

if largestOrSmallest = index then

break

swap(elements, index, largestOrSmallest)

index 🡨 largestOrSmallest

**Algorithm swap(i, j)**

**Input:** i, j, indices of the elements to swap in the elements array

**Output:** None

temp 🡨 elements[i]

elements[i] 🡨 elements[j]

elements[j] 🡨 temp

**B.**

**1. toggle()**

**Time Complexity:**

**Explanation:**

* Part: The method begins by flipping the boolean flag, which is a constant-time operation.
* Part: It then calls the helper method buildHeap() (if there is at least one element). Although each call to bubbleDown() (invoked within buildHeap()) can take up to in the worst case, the overall heapify process is known to run in linear time when applied over all non-leaf nodes.

**2. remove(e)**

**Time Complexity**

**Explanation:**

* Part: The method first searches for the element, which requires a linear scan through the heap, resulting in complexity.
* Part: Once the element is found, it needs to be removed, and the heap restructured. The restructuring involves at most a single bubbleDown() or bubbleUp(), each taking time. However, because the search time dominates, the overall complexity remains linear.

**3. peekAt(n)**

**Time Complexity:**

**Explanation:**

* Part: peekAt(n) uses the Quickselect algorithm with the median-of-medians algorithm to find the nth smallest or largest element. Quickselect has an average time complexity of but the use of median-of-medians ensures a worst-case time complexity of as well.

**4. merge(otherAPQ)**

**Time Complexity:**

**Explanation:**

* Part: The method merges two heaps by concatenating their elements, which takes time where is the size of the otherAPQ. After merging, buildHeap() is called on the combined heap, which has a size of The buildHeap() function operates in time to ensure all elements satisfy the heap property.