```
UPDATEKEY(H, i, k)
  old ← H[i]
  H[i] ← k
  if k > old then
    # "spingo verso l'alto finché non ripristino l'heap-property"
    while i > 1 and H[PARENT(i)] < H[i] do
      swap (H[i],H[PARENT(i)])
      i ← PARENT(i)
    end while
  else if k < old then
    #"spingo verso il basso usando Max-Heapify"
    MAX-HEAPIFY(H, i)
  end if

PARENT(i)
    Return FLOOR(i/2)

MAX-HEAPIFY(H, i)
  l ← 2 * i
  r ← 2 * i + 1
  if l ≤ heap_size and H[l] > H[i] then
    largest ← l
  else
    largest ← i
  if r ≤ heap_size and H[r] > H[largest] then
    largest ← r
  if largest ≠ i then
    swap (H[i],H[largest])
    MAX-HEAPIFY(H, largest)
```

Dettagli su PARENT e MAX-HEAPIFY

- `PARENT(i) = ⌊i/2⌋`.
- `MAX-HEAPIFY(H, i)` confronta il nodo in posizione i con i suoi figli (2i e 2i+1), scambia con il più grande se necessario e prosegue ricorsivamente fino a che la sottostruttura non è un heap massimo.

Correttezza

- **Caso k > H[i] (chiave aumentata):** la proprietà di heap può essere violata solo "in alto" (il figlio potrebbe essere più grande del padre), perciò eseguo una serie di scambi verso la radice finché ogni padre è ≥ di tutti i figli.
- **Caso k < H[i] (chiave diminuita):** la proprietà può essere violata "in basso" (il padre potrebbe essere più piccolo di uno dei figli), quindi basta una singola chiamata a `MAX-HEAPIFY (che risolve eventualmente altre violazioni).`

Complessità

- **Aumento di chiave:** in ogni iterazione del ciclo `while` risalgo di un livello nell'albero binario di altezza ≤ $\lfloor \log_2 n \rfloor$, dunque O(log n).
- **Diminuzione di chiave:** `MAX-HEAPIFY` in uno heap di n elementi costa O(log n) nel caso peggiore.

⇒ Nel complesso `UPDATEKEY` è O (log n).