# <u>Implement a Heap Data Structure</u>

You have been provided with a partial implementation of a heap data structure code. The code already has the following functions implemented:

- $initialize(int\ v[], int\ n)$ – Initialize the heap to store vertices in array $v$ with all their keys set to $\infty$.
- $heapify(int\ i) -$ This operation is required to re-store heap property for implementing certain operations, such as $removeMin$ operation, and $updateKey$ operation. This function is already implemented in given sample skeleton.
- $getKey(int\ data) -$ This operation returns the $key$ value of the given $data$ in the heap.
- $emtpy() -$ Returns $true$ if the heap is empty, otherwise returns false.
- $printHeap()$ - Prints the heap.

Your job is to complete the implementation that support the following heap  operations:

- $buHeapify(int\ i) -$ This operation is called when key value of an existing node is decreased or a new item is inserted at the end of the heap. For example, when the $key$ value of a data item at node index $i$ is decreased, the node may be required to move up the tree so that heap property (parent's $key$ less than or equal to childs' $key$) remains correct. This may be implemented as a recursive function or in iterative manner that starts swapping of nodes at node number $i$, then may call recursively at parent of $i$. In the worse case, the function will end at the root of the tree (all nodes from $i$ up-to the root will be swapped).
- $void\ insertItem(int\ data, int\ key)$ – This function inserts a new $(data, key)$ pair in the heap. The new item is first inserted at the end of the heap which may violate heap property. So $buHeapify$ operation is executed to ensure that heap property is restored. In the worst case, the inserted node may end up moving at the root node of the heap.
- $HeapItem\ removeMin() -$ This operation will return the heap node that has the minimum key value. Must restore heap property by calling $heapify$ after removal.
- $updateKey(int\ data,\ float\ key) -$ This operation updates the  $key$ value of the given $data$ to the given $key$. The function first searches for given $data$ in the heap and then updates the $key$ value. After update, it may happen that the new $key$ violates min-heap property. Hence, after update, a call to $heapify$ or $buheapify$ is required.

Note the following-

***-You must extend this class to implement your own Heap class. You cannot write your own class.***

-Note that a special array $map$ is kept in this M$inHeap$ class to keep track of where each data value is currently stored inside the heap. This is required for searching in the $updateKey(data, key)$ operation, because we need to know in which node the input data is stored.

-Without keeping this $map$ array, you may find where $data$ is stored by a linear search through all heap nodes which will require $O(n)$ time ($n$ is the number of nodes in heap). In this case, $updateKey$ will also require $O(n)$ time which is prohibited.

-However, keeping a $map$ array does the data search in constant time, which gives the overall time of $O(logn)$ for the $updateKey$ operation. *This is very much important for certain algorithms such as Prim's MST algorithm or Dijkstra's shortest path algorithm to ensure their optimal running time.*

-The $map$ array also helps us for $getKey$ operation to return the $key$ value of a vertex in constant time without keeping additional storage.

key is the main thing here . not data .