```python
import numpy


def heap_sort(data):
    heapify(data)
    end = len(data) - 1

    while end > 0:
        data[end], data[0] = data[0], data[end]
        end -= 1
        sift_down(data, 0, end)


def heapify(data):
    length = len(data)

    for i in reversed(range(length // 2)):
        sift_down(data, i, length - 1)


# max-heap for heapsort
def sift_down(data, start, end):
    root = start

    while root * 2 + 1 <= end:
        child = root * 2 + 1
        if child + 1 <= end and data[child] < data[child + 1]:
            child += 1
        if data[root] < data[child]:
            data[root], data[child] = data[child], data[root]
            root = child
        else:
            return


def python_heap_sort(data):
    import heapq

    heap = list(data)

    # min-heap
    heapq.heapify(heap)

    for i in range(len(data)):
        data[i] = heapq.heappop(heap)


if __name__ == '__main__':
    nums = [13, 14, 94, -31, 33, 82, 25, 59,
            94, 65, 23, -1, 0, -146, 45, 27,
            73, 25, 39, 10]
    sort_nums = [-146, -31, -1, 0, 10, 13, 14,
                 23, 25, 25, 27, 33, 39, 45, 59,
                 65, 73, 82, 94, 94]

    # using custom

    copy1 = nums[:]

    heap_sort(copy1)
    print(copy1)
    assert numpy.allclose(copy1, sort_nums)

    # using standard library

    copy2 = nums[:]

    python_heap_sort(copy2)
    print(copy2)
    assert numpy.allclose(copy2, sort_nums)
```

```python
from .priority_queue_base import PriorityQueueBase
from ..empty import Empty


class HeapPriorityQueue(PriorityQueueBase):  # base class defines _Item
    """A min-oriented priority queue implemented with a binary heap."""

    # ------------------ nonpublic behaviors ------------------
    def _parent(self, j):
        return (j - 1) // 2

    def _left(self, j):
        return 2 * j + 1

    def _right(self, j):
        return 2 * j + 2

    def _has_left(self, j):
        return self._left(j) < len(self._data)  # index beyond end of list?

    def _has_right(self, j):
        return self._right(j) < len(self._data)  # index beyond end of list?

    def _swap(self, i, j):
        """Swap the elements at indices i and j of array."""
        self._data[i], self._data[j] = self._data[j], self._data[i]

    def _upheap(self, j):
        parent = self._parent(j)
        if j > 0 and self._data[j] < self._data[parent]:
            self._swap(j, parent)
            self._upheap(parent)  # recur at position of parent

    def _downheap(self, j):
        if self._has_left(j):
            left = self._left(j)
            small_child = left  # although right may be smaller
            if self._has_right(j):
                right = self._right(j)
                if self._data[right] < self._data[left]:
                    small_child = right
            if self._data[small_child] < self._data[j]:
                self._swap(j, small_child)
                self._downheap(small_child)  # recur at position of small child

    # ------------------ public behaviors ------------------
    def __init__(self):
        """Create a new empty Priority Queue."""
        self._data = []

    def __len__(self):
        """Return the number of items in the priority queue."""
        return len(self._data)

    def add(self, key, value):
        """Add a key-value pair to the priority queue."""
        self._data.append(self._Item(key, value))
        self._upheap(len(self._data) - 1)  # upheap newly added position

    def min(self):
        """Return but do not remove (k,v) tuple with minimum key.

        Raise Empty exception if empty.
        """
        if self.is_empty():
            raise Empty('Priority queue is empty.')
        item = self._data[0]
        return (item._key, item._value)

    def remove_min(self):
        """Remove and return (k,v) tuple with minimum key.

        Raise Empty exception if empty.
        """
        if self.is_empty():
            raise Empty('Priority queue is empty.')
```

```
        self._swap(0, len(self._data) - 1)  # put minimum item at the end
        item = self._data.pop()  # and remove it from the list;
        self._downheap(0)  # then fix new root
        return (item._key, item._value)
```

```python
class PriorityQueueBase:
    """Abstract base class for a priority queue."""

    # ------------------ nested _Item class ------------------
    class _Item:
        """Lightweight composite to store priority queue items."""
        __slots__ = '_key', '_value'

        def __init__(self, k, v):
            self._key = k
            self._value = v

        def __lt__(self, other):
            return self._key < other._key  # compare items based on their keys

        def __repr__(self):
            return '({0},{1})'.format(self._key, self._value)

    # ------------------ public behaviors ------------------
    def is_empty(self):  # concrete method assuming abstract len
        """Return True if the priority queue is empty."""
        return len(self) == 0

    def __len__(self):
        """Return the number of items in the priority queue."""
        raise NotImplementedError('must be implemented by subclass')

    def add(self, key, value):
        """Add a key-value pair."""
        raise NotImplementedError('must be implemented by subclass')

    def min(self):
        """Return but do not remove (k,v) tuple with minimum key.

        Raise Empty exception if empty.
        """
        raise NotImplementedError('must be implemented by subclass')

    def remove_min(self):
        """Remove and return (k,v) tuple with minimum key.

        Raise Empty exception if empty.
        """
        raise NotImplementedError('must be implemented by subclass')
```

```python
from .priority_queue_base import PriorityQueueBase
from ..lists.positional_list import PositionalList
from ..empty import Empty


class SortedPriorityQueue(PriorityQueueBase):  # base class defines _Item
    """A min-oriented priority queue implemented with a sorted list."""

    # ------------------ public behaviors -----------------
    def __init__(self):
        """Create a new empty Priority Queue."""
        self._data = PositionalList()

    def __len__(self):
        """Return the number of items in the priority queue."""
        return len(self._data)

    def add(self, key, value):
        """Add a key-value pair."""
        newest = self._Item(key, value)  # make new item instance
        walk = self._data.last()  # walk backward looking for smaller key
        while walk is not None and newest < walk.element():
            walk = self._data.before(walk)
        if walk is None:
            self._data.add_first(newest)  # new key is smallest
        else:
            self._data.add_after(walk, newest)  # newest goes after walk

    def min(self):
        """Return but do not remove (k,v) tuple with minimum key.

        Raise Empty exception if empty.
        """
        if self.is_empty():
            raise Empty('Priority queue is empty.')
        p = self._data.first()
        item = p.element()
        return (item._key, item._value)

    def remove_min(self):
        """Remove and return (k,v) tuple with minimum key.

        Raise Empty exception if empty.
        """
        if self.is_empty():
            raise Empty('Priority queue is empty.')
        item = self._data.delete(self._data.first())
        return (item._key, item._value)
```

```python
from .priority_queue_base import PriorityQueueBase
from ..lists.positional_list import PositionalList
from ..empty import Empty


class UnsortedPriorityQueue(PriorityQueueBase):  # base class defines _Item
    """A min-oriented priority queue implemented with an unsorted list."""

    # ------------------ nonpublic behavior ----------------
    def _find_min(self):
        """Return Position of item with minimum key."""
        if self.is_empty():  # is_empty inherited from base class
            raise Empty('Priority queue is empty')
        small = self._data.first()
        walk = self._data.after(small)
        while walk is not None:
            if walk.element() < small.element():
                small = walk
            walk = self._data.after(walk)
        return small

    # ------------------ public behaviors ------------------
    def __init__(self):
        """Create a new empty Priority Queue."""
        self._data = PositionalList()

    def __len__(self):
        """Return the number of items in the priority queue."""
        return len(self._data)

    def add(self, key, value):
        """Add a key-value pair."""
        self._data.add_last(self._Item(key, value))

    def min(self):
        """Return but do not remove (k,v) tuple with minimum key.

        Raise Empty exception if empty.
        """
        p = self._find_min()
        item = p.element()
        return (item._key, item._value)

    def remove_min(self):
        """Remove and return (k,v) tuple with minimum key.

        Raise Empty exception if empty.
        """
        p = self._find_min()
        item = self._data.delete(p)
        return (item._key, item._value)
```

```python
from .heap_priority_queue import HeapPriorityQueue


class AdaptableHeapPriorityQueue(HeapPriorityQueue):
    """A locator-based priority queue implemented with a binary heap."""

    # ------------------ nested Locator class ------------------
    class Locator(HeapPriorityQueue._Item):
        """Token for locating an entry of the priority queue."""
        __slots__ = '_index'  # add index as additional field

        def __init__(self, k, v, j):
            super().__init__(k, v)
            self._index = j

    # ------------------ nonpublic behaviors ------------------
    # override swap to record new indices
    def _swap(self, i, j):
        super()._swap(i, j)  # perform the swap
        self._data[i]._index = i  # reset locator index (post-swap)
        self._data[j]._index = j  # reset locator index (post-swap)

    def _bubble(self, j):
        if j > 0 and self._data[j] < self._data[self._parent(j)]:
            self._upheap(j)
        else:
            self._downheap(j)

    # ------------------ public behaviors ------------------
    def add(self, key, value):
        """Add a key-value pair."""
        token = self.Locator(key, value, len(self._data))  # initiaize locator index
        self._data.append(token)
        self._upheap(len(self._data) - 1)
        return token

    def update(self, loc, newkey, newval):
        """Update the key and value for the entry identified by Locator loc."""
        j = loc._index
        if not (0 <= j < len(self) and self._data[j] is loc):
            raise ValueError('Invalid locator')
        loc._key = newkey
        loc._value = newval
        self._bubble(j)

    def remove(self, loc):
        """Remove and return the (k,v) pair identified by Locator loc."""
        j = loc._index
        if not (0 <= j < len(self) and self._data[j] is loc):
            raise ValueError('Invalid locator')
        if j == len(self) - 1:  # item at last position
            self._data.pop()  # just remove it
        else:
            self._swap(j, len(self) - 1)  # swap item to the last position
            self._data.pop()  # remove it from the list
            self._bubble(j)  # fix item displaced by the swap
        return (loc._key, loc._value)
```