In [1]:

```
# Importing base classes

from priority_queue_base import PriorityQueueBase    # PriorityQueueBase defines Priority Queue ADT
(abstract data type)
from positional_list import PositionalList           # PostionalList is an implementation of a doubl
y linked list
from exceptions import Empty
```

Unsorted Priority Queue

In our first concrete implementation of a priority queue, we store entries within an ***unsorted list***. Our `UnsortedPriorityQueue` class is given in the following, inheriting from the `PriorityQueueBase` class introduced in `priority_queue_base.py` . For internal storage, key-value pairs are represented as composites, using instances of the inherited `Item` class. These items are stored within a `PositionalList` , identified asthe `_data` member of our class. We assume that the positional list is implemented with a doubly-linked list, as in `positional_list.py` , so that all operations of that ADT execute in *O(1)* time.

We begin with an empty list when a new priority queue is constructed. At all times, the size of the list equals the number of key-value pairs currently stored in the priority queue. For this reason, our priority queue `__len__` method simply returns the length of the internal data list. By the design of our PriorityQueueBase class, we inherit a concrete implementation of the is empty method that relies on a call to our `__len__` method.

Each time a key-value pair is added to the priority queue, via the `add` method, we create a new `_Item` composite for the given key and value, and add that item to the end of the list. Such an implementation takes *O(1)* time.

The remaining challenge is that when min or remove min is called, we must locate the item with minimum key. Because the items are not sorted, we must inspect all entries to find one with a minimum key. For convenience, we define a nonpublic find min utility that returns the *position* of an item with minimum key. Knowledge of the position allows the remove min method to invoke the `delete` method on the positional list. The min method simply uses the position to retrieve the item when preparing a key-value tuple to return. Due to the loop for finding the minimum key, both min and remove min methods run in *O(n)* time, where *n* is the number of entries in the priority queue.

A summary of the running times for the `UnsortedPriorityQueue` class is given as:

Operation	Running Time
`len()`	*O(1)*
`is_empty()`	*O(1)*
`add(o)`	*O(1)*
`o = min()`	*O(n)*
`o = remove_min()`	*O(n)*
space	*O(n)*

In [2]:

```
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."""
```

```python
        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)

    # Custom function to display the entire elements
    def list_elems(self):
        listed_elem = []
        if self.is_empty():                          # is_empty inherited from base class
            return listed_elem
        walk = self._data.first()
        while walk is not None:
            item = walk.element()
            listed_elem.append((item._key, item._value))
            walk = self._data.after(walk)
        return  listed_elem
```

In [3]:

```python
# For testing purpose, assign a random names
import random
names = ["Alfa", "Bravo", "Charlie", "Delta", "Echo", "Foxtrot", "Golf", "Hotel", "India", "Juliett
", "Kilo", "Lima", "Mike", "November", "Oscar", "Papa", "Quebec", "Romeo", "Sierra", "Tango", "Unif
orm", "Victor", "Whiskey", "X-ray", "Yankee", "Zulu"]

Q = UnsortedPriorityQueue()

for name in random.sample(names, 5):
    key = random.randint(0, 100)
    print(key, name)
    Q.add(key, name)
```

```
47 Victor
68 Kilo
25 Zulu
56 Yankee
39 Sierra
```

In [4]:

```python
Q.list_elems()
```

Out[4]:

```
[(47, 'Victor'), (68, 'Kilo'), (25, 'Zulu'), (56, 'Yankee'), (39, 'Sierra')]
```

In [5]:

```python
print(Q.remove_min())
print(Q.remove_min())
```

```
print(Q.remove_min())
print(Q.remove_min())
print(Q.remove_min())
```

```
(25, 'Zulu')
(39, 'Sierra')
(47, 'Victor')
(56, 'Yankee')
(68, 'Kilo')
```

Sorted Priority Queue

An alternative implementation of a priority queue uses a positional list, yet maintaining entries sorted by nondecreasing keys. This ensures that the first element of the list is an entry with the smallest key.

Our `SortedPriorityQueue` class is given in the following code. The implementation of `min` and `remove_min` are rather straightforward given knowledge that the first element of a list has a minimum key. We rely on the first method of the positional list to find the position of the first item, and the delete method to remove the entry from the list. Assuming that the list is implemented with a doubly linked list, operations min and remove min take *O(1)* time.

This benefit comes at a cost, however, for method add now requires that we scan the list to find the appropriate position to insert the new item. Our implementation starts at the end of the list, walking backward until the new key is smaller than an existing item; in the worst case, it progresses until reaching the front of the list. Therefore, the add method takes *O(n)* worst-case time, where *n* is the number of entries in the priority queue at the time the method is executed. In summary, when using a sorted list to implement a priority queue, insertion runs in linear time, whereas finding and removing the minimum can be done in constant time.

This table compares the running times of the methods of a priority queue realized by means of a sorted and unsorted list, respectively. We see an interesting trade-off when we use a list to implement the priority queue ADT. An unsorted list supports fast insertions but slow queries and deletions, whereas a sorted list allows fast queries and deletions, but slow insertions.

Operation	Unsorted List	Sorted List
`len()`	*O(1)*	*O(1)*
`is_empty()`	*O(1)*	*O(1)*
`add(o)`	*O(1)*	*O(n)*
`o = min()`	*O(n)*	*O(1)*
`o = remove_min()`	*O(n)*	*O(1)*
space	*O(n)*	*O(n)*

In [6]:

```
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)

    # Custom function to display the entire elements
    def list_elems(self):
        listed_elem = []
        if self.is_empty():                             # is_empty inherited from base class
            return listed_elem
        walk = self._data.first()
        while walk is not None:
            item = walk.element()
            listed_elem.append((item._key, item._value))
            walk = self._data.after(walk)
        return  listed_elem
```

In [7]:

```
Q2 = SortedPriorityQueue()

for name in random.sample(names, 5):
    key = random.randint(0, 100)
    print(key, name)
    Q2.add(key, name)
```

```
70 Juliett
77 Sierra
21 November
28 Charlie
63 Golf
```

In [8]:

```
Q2.list_elems()
```

Out[8]:

```
[(21, 'November'),
 (28, 'Charlie'),
 (63, 'Golf'),
 (70, 'Juliett'),
 (77, 'Sierra')]
```

In [9]:

```
print(Q2.remove_min())
print(Q2.remove_min())
print(Q2.remove_min())
print(Q2.remove_min())
print(Q2.remove_min())
```

```
(21, 'November')
(28, 'Charlie')
(63, 'Golf')
(70, 'Juliett')
(77, 'Sierra')
```

Heap-based priority Queue

We provide a Python implementation of a heap-based priority queue in the following code. We use an array-based representation, maintaining a Python list of item composites. Although we do not formally use the binary tree ADT, the code includes nonpublic utility functions that compute the level numbering of a parent or child of another. This allows us to describe the rest of our algorithms using tree-like terminology of *parent*, *left*, and *right*. However, the relevant variables are integer indexes (not “position” objects). We use recursion to implement the repetition in the `_upheap` and `_downheap` utilities.

The table displays the running time of the methods in different priority queue implementations.

Operation	Unsorted List	Sorted List	Heap-based
`len()`	*O(1)*	*O(1)*	*O(1)*
`is_empty()`	*O(1)*	*O(1)*	*O(1)*
`add(o)`	*O(1)*	*O(n)*	O($\log n$)
`o = min()`	*O(n)*	*O(1)*	*O(1)*
`o = remove_min()`	*O(n)*	*O(1)*	O($\log n$)
space	*O(n)*	*O(n)*	*O(n)*

In [10]:

```
# Copyright 2013, Michael H. Goldwasser
#
# Developed for use with the book:
#
#       Data Structures and Algorithms in Python
#       Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser
#       John Wiley & Sons, 2013
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

from priority_queue_base import PriorityQueueBase
from exceptions 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 roo
        return (item._key, item._value)

    def display_heap(self, j=0): # j: position
        if len(self) == 0:
            return

        height = int((j+1)/2)
        print('+-'*height+str(self._data[j]))

        if self._has_left(j):
            left = self._left(j)
            self.display_heap(left)
        if self._has_right(j):
            right = self._right(j)
            self.display_heap(right)
```

In [11]:

```
Qh = HeapPriorityQueue()

for name in random.sample(names, 5):
    key = random.randint(0, 100)
    print("Inserted:", key, name)
    Qh.add(key, name)
    Qh.display_heap()
    print()
```

```
Inserted: 48 Papa
(48,Papa)

Inserted: 54 Tango
(48,Papa)
+-(54,Tango)

Inserted: 77 Charlie
(48,Papa)
+-(54,Tango)
+-(77,Charlie)

Inserted: 79 Lima
(48,Papa)
+-(54,Tango)
+-+-(79,Lima)
+-(77,Charlie)

Inserted: 91 Oscar
(48,Papa)
+-(54,Tango)
+-+-(79,Lima)
+-+-(91,Oscar)
+-(77,Charlie)
```

In [12]:

```
for i in range(5):
    print("Removed:", Qh.remove_min())
    Qh.display_heap()
    print()
```

```
Removed: (48, 'Papa')
(54,Tango)
+-(79,Lima)
+-+-(91,Oscar)
+-(77,Charlie)

Removed: (54, 'Tango')
(77,Charlie)
+-(79,Lima)
+-(91,Oscar)

Removed: (77, 'Charlie')
(79,Lima)
+-(91,Oscar)

Removed: (79, 'Lima')
(91,Oscar)

Removed: (91, 'Oscar')
```

Sorting with priority queue

As our first application of priority queues, we demonstrate how they can be used to sort a collection C of comparable elements. That is, we can produce a sequence of elements of C in increasing order (or at least in nondecreasing order if there are duplicates). The algorithm is quite simple—we insert all elements into an initially empty priority queue, and then we repeatedly call remove min to retrieve the elements in nondecreasing order.

In [13]:

```
def pq_sort(elements, PQ):
    n = len(elements)
    sorted = []
    for elem in elements:
        PQ.add(elem, elem)

    for j in range(n):
        (k, v) = PQ.remove_min()
        sorted.append(v)
```

```
        sorted.append(v)

    return sorted
```

In [14]:

```
import random
import time

unsorted_list = []
for i in range(1000):
    unsorted_list.append(random.randrange(1000))

#print(unsorted_list)
```

In [15]:

```
# Selection Sort

start_time = time.time()
sorted_list = pq_sort(unsorted_list, SortedPriorityQueue())
end_time = time.time()
sorted_time = end_time - start_time

#print(sorted_list)
```

In [16]:

```
# Insertion Sort

start_time = time.time()
sorted_list = pq_sort(unsorted_list, UnsortedPriorityQueue())
end_time = time.time()
unsorted_time = end_time - start_time

#print(sorted_list)
```

In [17]:

```
# Heap Sort
# Improvement: bottom-up heap building
#              in-place heap sort

start_time = time.time()
sorted_list = pq_sort(unsorted_list, HeapPriorityQueue())
end_time = time.time()
heapsort_time = end_time - start_time

#print(sorted_list)
```

In [18]:

```
print(sorted_time, unsorted_time, heapsort_time)
```

```
0.6700878143310547 1.3490607738494873 0.03217196464538574
```

Python's heapq module

`heapq` module provides a convenient **Heap** implementation in Python. https://docs.python.org/3.8/library/heapq.html

In [19]:

```
from heapq import heappush, heappop, heapify

def heapsort(iterable):
    h = []
    for value in iterable:
        heappush(h, value)
    return [heappop(h) for i in range(len(h))]
```

In [20]:

```
print(heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0]))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [21]:

```
items = random.sample(range(100), 10)
print(items)
heapify(items) # This runs in linear time => in-place and bottom-up construction!
print(items)
```

```
[63, 9, 34, 8, 52, 16, 57, 84, 12, 75]
[8, 9, 16, 12, 52, 34, 57, 84, 63, 75]
```

In [22]:

```
h = []
heappush(h, (5, 'write code'))
heappush(h, (7, 'release product'))
heappush(h, (1, 'write spec'))
heappush(h, (3, 'create tests'))
```

In [23]:

```
print(heappop(h))
print(heappop(h))
print(heappop(h))
print(heappop(h))
```

```
(1, 'write spec')
(3, 'create tests')
(5, 'write code')
(7, 'release product')
```

In [ ]:

```

```