Priority Queues

Definition

With queues

- The order may be summarized by ***first in, first out***

If each object is associated with a priority, **we may wish to pop that object which has highest priority**

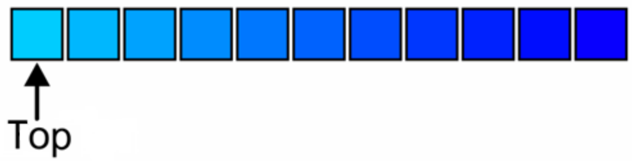
With each pushed object, we will associate a nonnegative integer (0, 1, 2, ...) where:

- **The value 0 has the *highest* priority, and**
- **The higher the number, the lower the priority**

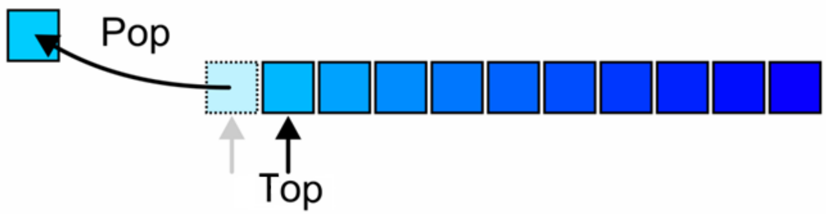
Thus, 5 has lower priority than 3 which has a lower priority than 1.

Operations

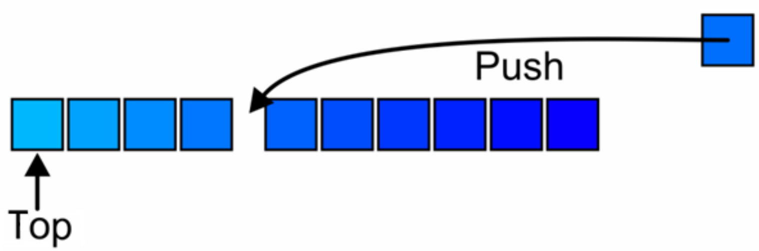
The top of a priority queue is the object with highest priority



Popping from a priority queue removes the current highest priority object:



Push places a new object into the **appropriate** place



Priority Queue

- This is a collection of prioritized elements that allows arbitrary element insertion, and allows the removal of the element that has first priority.
- When an element is added to a priority queue, the user designates its priority by providing an **associated key**.
  - **The element with the minimum key** will be the next to be removed from the queue (thus, an element with key 1 will be given priority over an element with key 2)
  - Although it is quite common for **priorities to be expressed numerically**, any **Python object may be used as a key**, as long as the object type supports **a consistent meaning for the test a < b,** for any instances a and b, so as to define a natural order of the keys.

Lexicographical Priority

Priority may also depend on multiple variables:

- Two values specify a priority: (a, b)
- A pair (a, b) has higher priority than (c, d) if:
  - $a < c$, or
  - $a = c$ and $b < d$

For example,

- (5, 19), (13, 1), (13, 24), and (15, 0) all have *higher* priority than (15, 7)

Common Applications of Priority Queues

- In any flavor of Unix, **the highest priority for any user process is 0** and when you execute any Unix command at the prompt, it automatically runs at priority-level 0.
- **It is possible, however, to lower the priority of a process. ??**
- **Why on earth would anyone want to do that?**

Process Priority in Unix: nice command

nice is a program found on **Unix and Unix-like** operating systems such as Linux.
It directly maps to a kernel call of the same name.

nice is used **to invoke a utility or shell script** with a particular priority, thus giving the process more or less CPU time than other processes.
A niceness of −20 is the highest priority and 19 is the lowest priority. The default niceness for processes is inherited from its parent process and is usually 0.

This is the scheme used by Unix, *e.g.*,

```
% nice +15 ./a.out
```

reduces the priority of the execution of the routine `a.out` by 15

This allows the processor to be used by interactive programs

– This does not significantly affect the run-time of CPU-bound processes

Interactive Process

- Now, why would anyone reduce the priority of a process?

- One possible answer is :

Suppose one user is using an **interactive process such as an editor—this** is a program that has relatively fast responses without significant use of the processor: the user strikes a key, an interrupt is handled, the character is placed into the appropriate location, and the screen is updated.

Why would anyone reduce the priority of their job?

- A **CPU-bound process** is any process that will use the CPU at every single opportunity.
- Examples include **circuit simulations, randomized testing for determining the stability of a system, or any other long-running process the main concern** of which is to perform calculations.
- Suppose there are numerous CPU-bound processes running at the same time as an **interactive process**.
- Most people will be familiar with the results: press a key and wait half a second before it appears on the screen, or type a sentence and a second later, the entire sentence finally appears on the screen. You can fix that **by reducing the priority of the CPU-bound processes, those using interactive processes will not notice a negligible effect**.

Building a Priority Queue ADT

Priority Queue ADT

- A priority queue stores a collection of items
- Each item is a pair (**key**, **value**)
- Main methods of the Priority Queue ADT
  - add (k, x)
    inserts an item with key k and value x
  - remove_min()
    removes and returns the item with smallest key
- Additional methods
  - min()
    returns, but does not remove, an item with smallest key
  - len(P), is_empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Priority Queue Example

The following table shows a series of operations and their effects on an initially empty priority queue P. The "Priority Queue" column is somewhat deceiving since it shows the entries as tuples and sorted by key. Such an internal representation is not required of a priority queue.

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min()	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min()	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min()	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min()	(7,D)	{(9,C)}
P.remove_min()	(9,C)	{ }
P.is_empty()	True	{ }
P.remove_min()	"error"	{ }

Composition Design Pattern

- One challenge in implementing a priority queue is that we must **keep track of both an element and its key**, even as items are relocated within our data structure.

- **Item** class will help that **each element remained paired with its associated count** in our primary data structure.

Item Class

```
#--------------------------- 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)
```

For future apps you can evolve this more , you can add a second component (field) to check or if comparing objects compare with more than one field member

Composition Design Pattern

- For priority queues, we will use composition to store items internally as pairs consisting of a key k and a value v.
- To implement this concept for all priority queue implementations, we provide a **PriorityQueueBase** class

PriorityQueueBase Class
Design stage:
What are the essential methods we need?

Define PriorityQueueBase Class

P.**add**(k, v): Insert an item with key k and value v into priority queue P.

P.**min**(): Return a tuple, (k,v), representing the key and value of an item in priority queue P with minimum key (but do not remove the item); an error occurs if the priority queue is empty.

P**.remove min**():Remove an item with minimum key from priority queue P, and return a tuple, (k,v), representing the key and value of the removed item; an error occurs if the priority queue is empty.

P.is empty( ): Return True if priority queue P does not contain any items

len(P):Return the number of items in priority queue P.

Define the Abstract Class: PriorityQueueBase Class

PriorityQueueBase Class: An Abstract class uses a nested class Item

```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)
```

Comparison is based on key

What are the public behaviors of PriorityQueuBase class?

is_empty

-len

add

min

remove_min

PriorityQueueBase Class:Abstract

```
#----------------------------- 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')
```

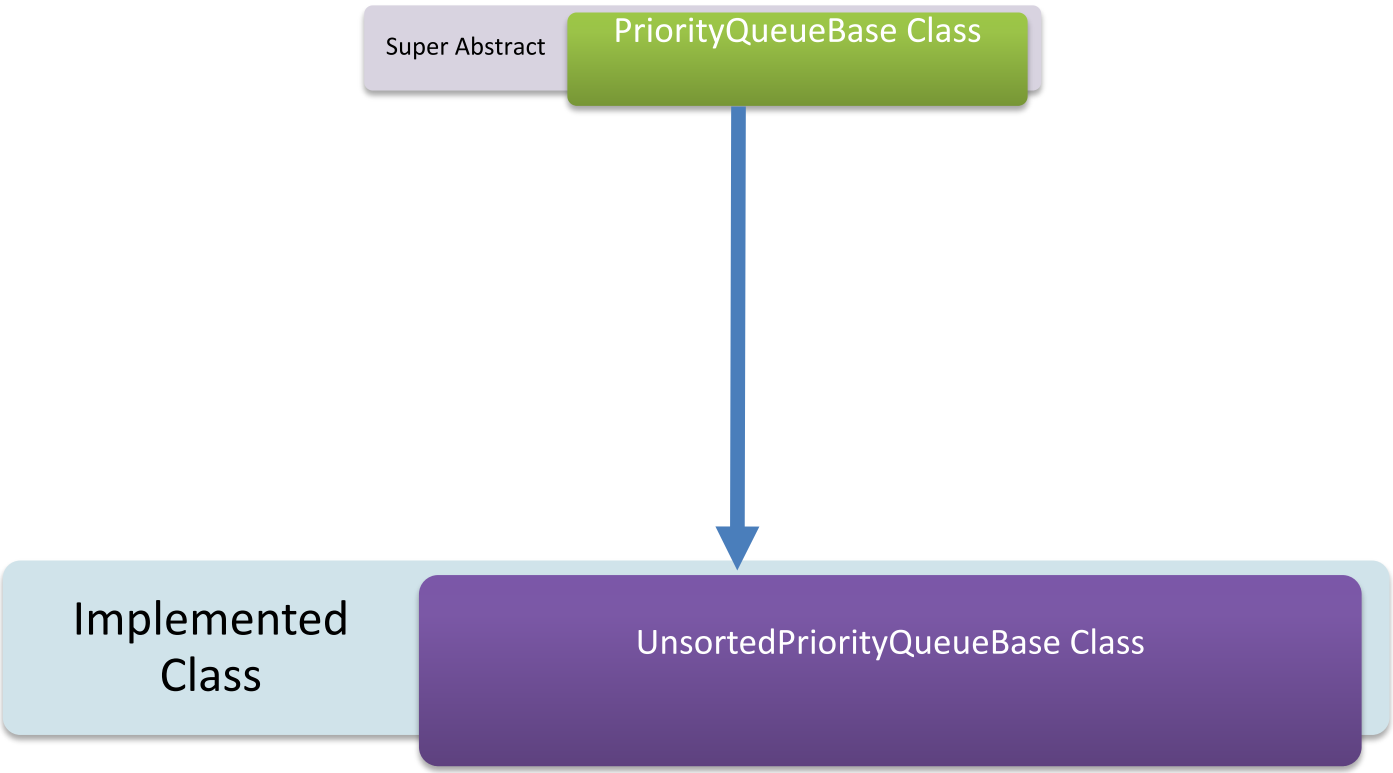
For convenience, we provide a concrete implementation of is empty that is based on a presumed len impelementation.

Now using this Abstract Class we will follow an implementation:
Implementation of Priority Queue with an Unsorted List

Implementation of Priority Queue with an Unsorted List

- Our **UnsortedPriorityQueue** class **inherits** from the **PriorityQueueBase** class
- 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 as the data member of our class.
  - Assumption : The positional list is implemented with a doubly-linked list, so that all operations of that ADT execute in O(1) time. Check out Chapter 7 for review of doubly linked lists.

Inheritance



How will we implement this class?

Implementation of Priority Queue with an Unsorted List

- First : Begin with **an empty list** when a new priority queue is constructed.
- Also assumption: at all times, the **size** of the list equals the number of key-value pairs currently stored in the priority queue.

Implementation of Priority Queue with an Unsorted List

- Since the size = number of key value pairs

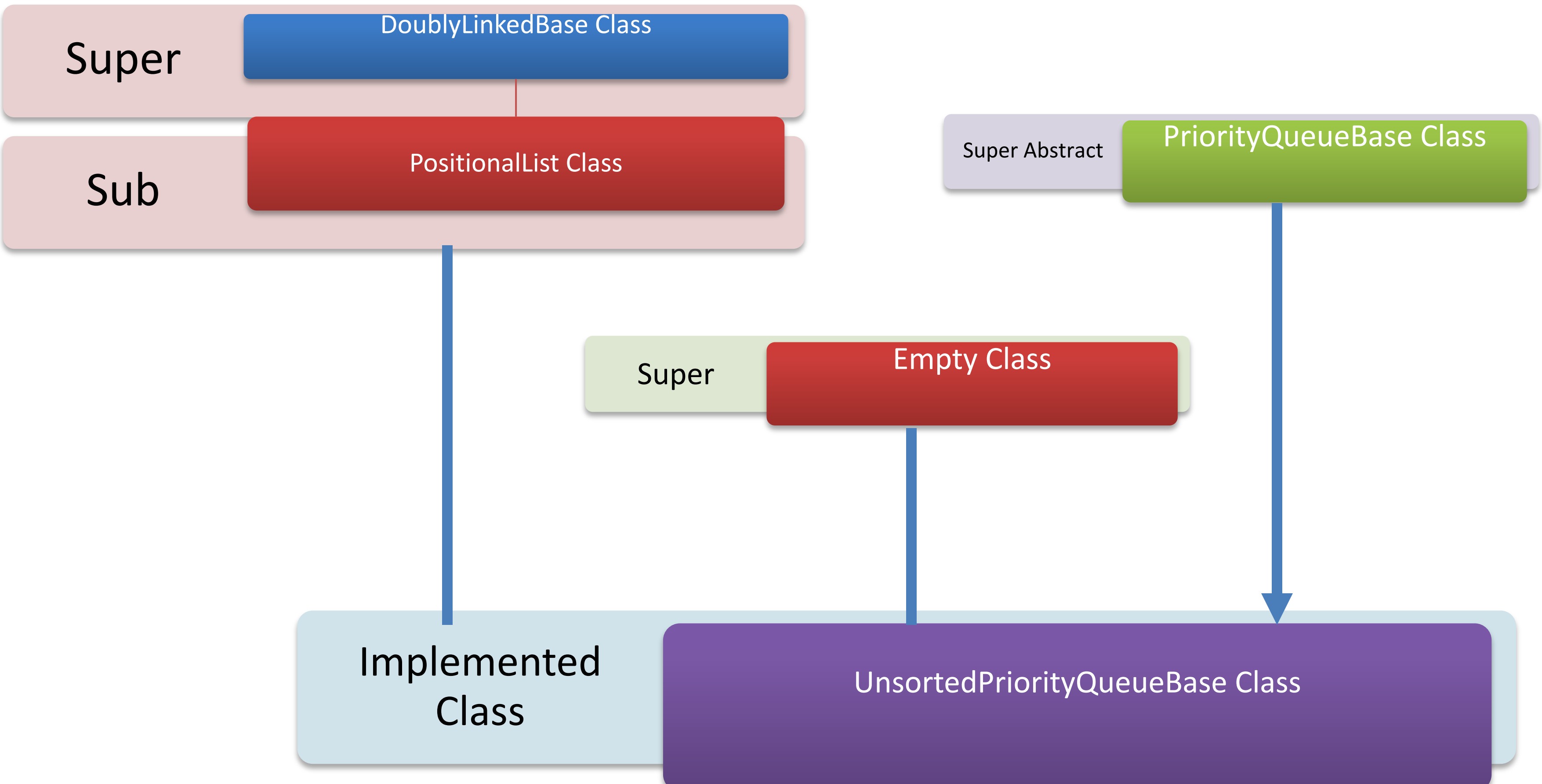
priority queue **__len__** method will return **the length of the internal _data list**.

- 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.

Priority Queue Implementation with an unsorted list

- Implementation with an unsorted list

- Performance concerns:
  - add takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - Remove_min and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key



UnsortedPriorityQueueBase **inherits** from PriorityQueuBase and **uses** Empty, PositionalList classes

```python
from priority_queue_base import PriorityQueueBase
from positional_list import PositionalList
from exceptions 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
```

Inherits from **PositionalList**, these methods (first, after) were implemented in that class
See next slide for reference

Important Side note on PositionalList inherits from DoublyLinked List clas

```
from doubly_linked_base import _DoublyLinkedBase

class PositionalList(_DoublyLinkedBase):
#nested Position class and the utility methods are not shown here,
#check out
#the source folder given to you with these lecture notes.

#------------------------------ accessors ------------------------------
def first(self):
  """Return the first Position in the list (or None if list is empty)."""
  return self._make_position(self._header._next)

def last(self):
  """Return the last Position in the list (or None if list is empty)."""
  return self._make_position(self._trailer._prev)

def before(self, p):
  """Return the Position just before Position p (or None if p is first)."""
  node = self._validate(p)
  return self._make_position(node._prev)

def after(self, p):
  """Return the Position just after Position p (or None if p is last)."""
  node = self._validate(p)
  return self._make_position(node._next)

def __iter__(self):
  """Generate a forward iteration of the elements of the list."""
  cursor = self.first()
  while cursor is not None:
    yield cursor.element()
    cursor = self.after(cursor)
```

Inherits form Doubly LinkedBase class, which given to you with the lecture notes

Also Check out the code given to you for PositionalList class and its mutators

```
from doubly_linked_base import _DoublyLinkedBase
class PositionalList(_DoublyLinkedBase):
#nested Position class and the utility methods are not shown here,
#check out
#the source folder given to you with these lecture notes.
#------------------------------ mutators ------------------------------
# override inherited version to return Position, rather than Node
def _insert_between(self, e, predecessor, successor):
  """Add element between existing nodes and return new Position."""
  node = super()._insert_between(e, predecessor, successor)
  return self._make_position(node)

def add_first(self, e):
  """Insert element e at the front of the list and return new Position."""
  return self._insert_between(e, self._header, self._header._next)

def add_last(self, e):
  """Insert element e at the back of the list and return new Position."""
  return self._insert_between(e, self._trailer._prev, self._trailer)

def add_before(self, p, e):
  """Insert element e into list before Position p and return new Position."""
  original = self._validate(p)
  return self._insert_between(e, original._prev, original)
```

PositionalList class and its mutators

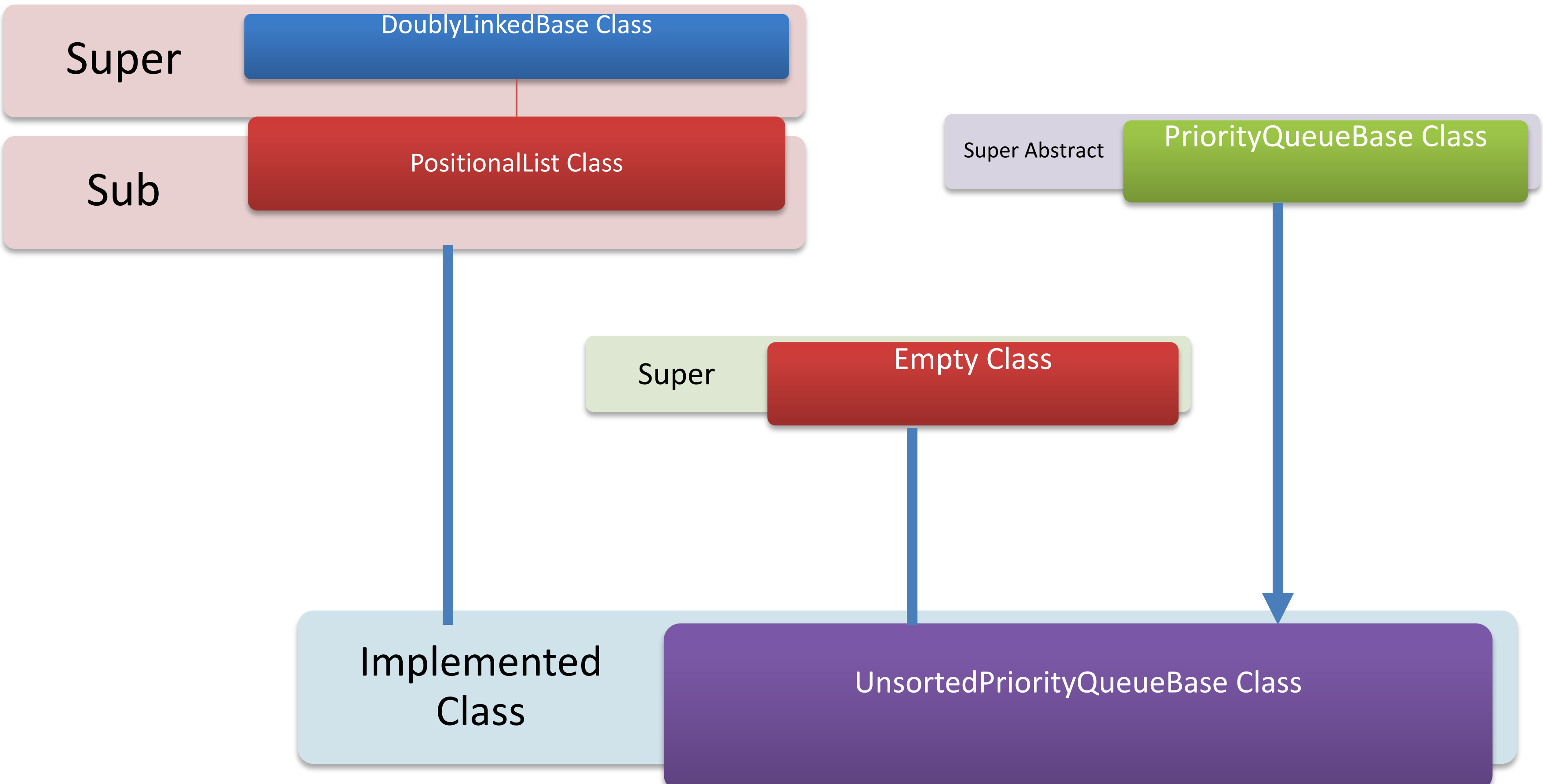
```
from doubly_linked_base import _DoublyLinkedBase
class PositionalList(_DoublyLinkedBase):
#nested Position class and the utility methods are not shown here,
#check out
#the source folder given to you with these lecture notes.
#---------------------------- mutators  cont..----------------------------
# override inherited version to return Position, rather than Node
def add_after(self, p, e):
  """Insert element e into list after Position p and return new Position."""
  original = self._validate(p)
  return self._insert_between(e, original, original._next)

def delete(self, p):
  """Remove and return the element at Position p."""
  original = self._validate(p)
  return self._delete_node(original)   # inherited method returns element

def replace(self, p, e):
  """Replace the element at Position p with e.
  Return the element formerly at Position p.
  """
  original = self._validate(p)
  old_value = original._element      # temporarily store old element
  original._element = e             # replace with new element
  return old_value                 # return the old element value
```

PositionalList class and its mutators

insert_between and delete_node are implemented in **doublyLinked** list class

Going back to our implementation of UnsortedPriorityQueue class to list its public functions

Implementing P.Queues using an Unsorted Table



UnsortedPriorityQueueBase **inherits** from PriorityQueuBase and **uses** Empty, PositionalList classes

```
from priority_queue_base import PriorityQueueBase
from positional_list import PositionalList
from exceptions import Empty
 class UnsortedPriorityQueue(PriorityQueueBase):
# ------------------------------ 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)
```

```
from priority_queue_base import PriorityQueueBase
from positional_list import PositionalList
from exceptions import Empty
 class UnsortedPriorityQueue(PriorityQueueBase):
# ---------------------------- 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)
```

Run time analysis

Run time analysis of P.Queues using an Unsorted List

Operation	Running Time
len	O(1)
Is_empty	O(1)
add	O(1)
min	O(n)
remove_min	O(n)

Implementation of a Priority Queue with a **Sorted** List

Priority Queues with a Sorted List

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

Implementation with a sorted list (Priority Queue)

- Implementation with a sorted list

- Performance concerns:
  - add takes $O(n)$ time since **<u>we have to find the place</u>** where to insert the item
  - remove_min and min take $O(1)$ time, since the smallest key is at the beginning

Implementation

- Our SortedPriorityQueue class is given in code segment next.
- The implementation of **min** and **remove min** are rather straightforward given knowledge that the first element of a list has a minimum key.

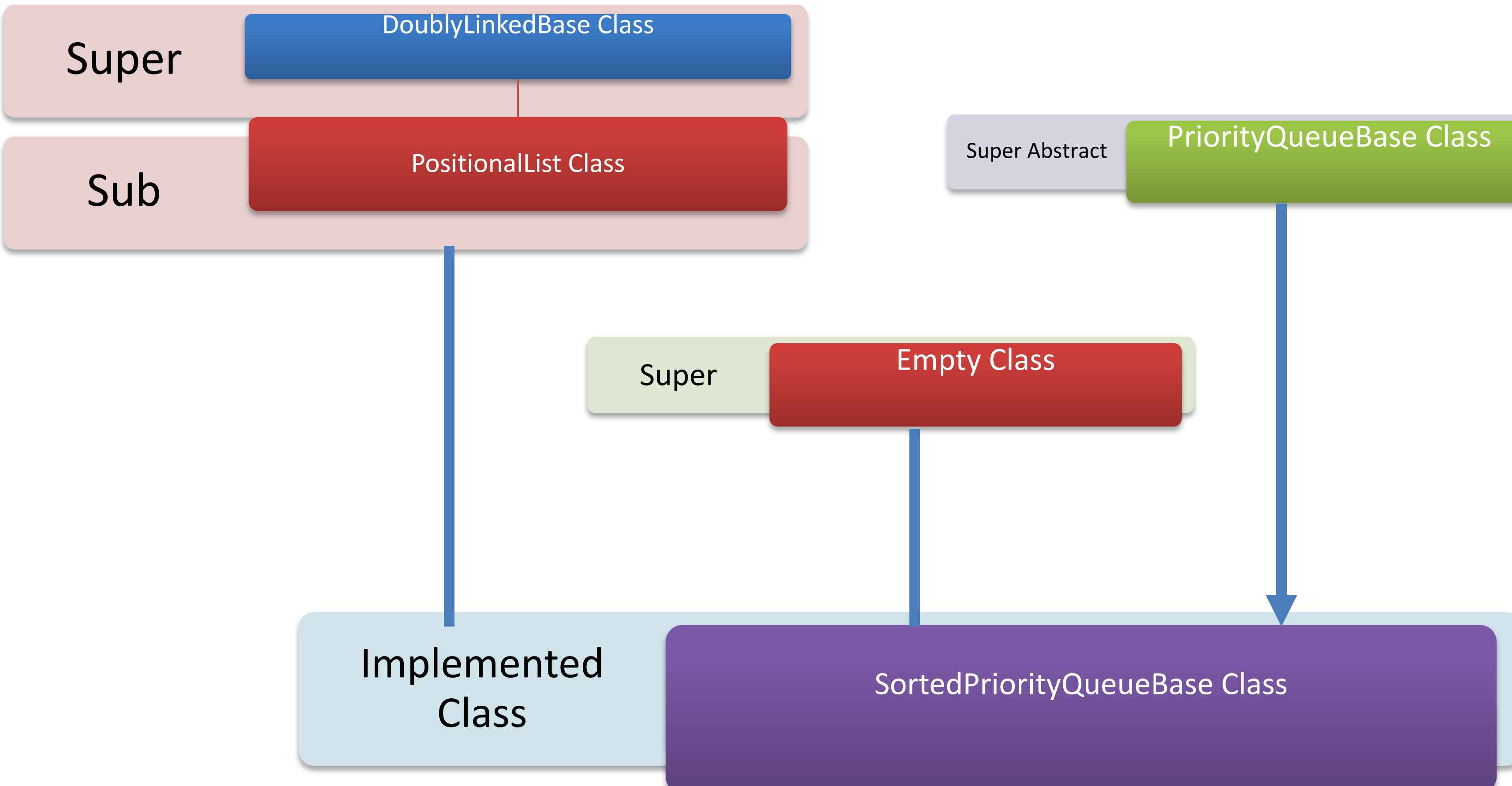
Implementation

- 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.

P.Queue Sorted List Implementation

```
from priority_queue_base import PriorityQueueBase
from positional_list import PositionalList
from exceptions import Empty

class SortedPriorityQueue(PriorityQueueBase): # base class defines _Ite
  """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)
```

inheritance



SortedPriorityQueueBase **inherits** from PriorityQueuBase and **uses** Empty, PositionalList classes

P.Queue Sorted List Implementation

```
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
```

add takes $O(n)$ time since **<u>we have to find the place</u>** where to insert the item

P.Queue Sorted List Implementation

```
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)
```

Since the list is implemented with a doubly linked list, operations **min and remove min** take O(1) time.

P.Queue Sorted List Implementation

Since the list is implemented with a doubly linked list, operations **min and remove min** take O(1) time.

```
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)
```

P.Queue Sorted List Implementation add method concerns

- 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.

Comparing the Two List-Based Implementations

Operation	Unsorted List	Sorted List
len	O(1)	O(1)
is_empty	O(1)	O(1)
add	O(1)	O(n)
min	O(n)	O(1)
remove_min	O(n)	O(1)

Worst-case running times of the methods of a priority queue of size n, realized by means of an unsorted or sorted list, respectively. We assume that the list is implemented by a doubly linked list. The space requirement is O(n).