Queues Overview

In this lecture we will get an overview of what a Queue is, in the next lecture we will implement our own Queue class.

A **queue** is an ordered collection of items where the addition of new items happens at one end, called the “rear,” and the removal of existing items occurs at the other end, commonly called the “front.” As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

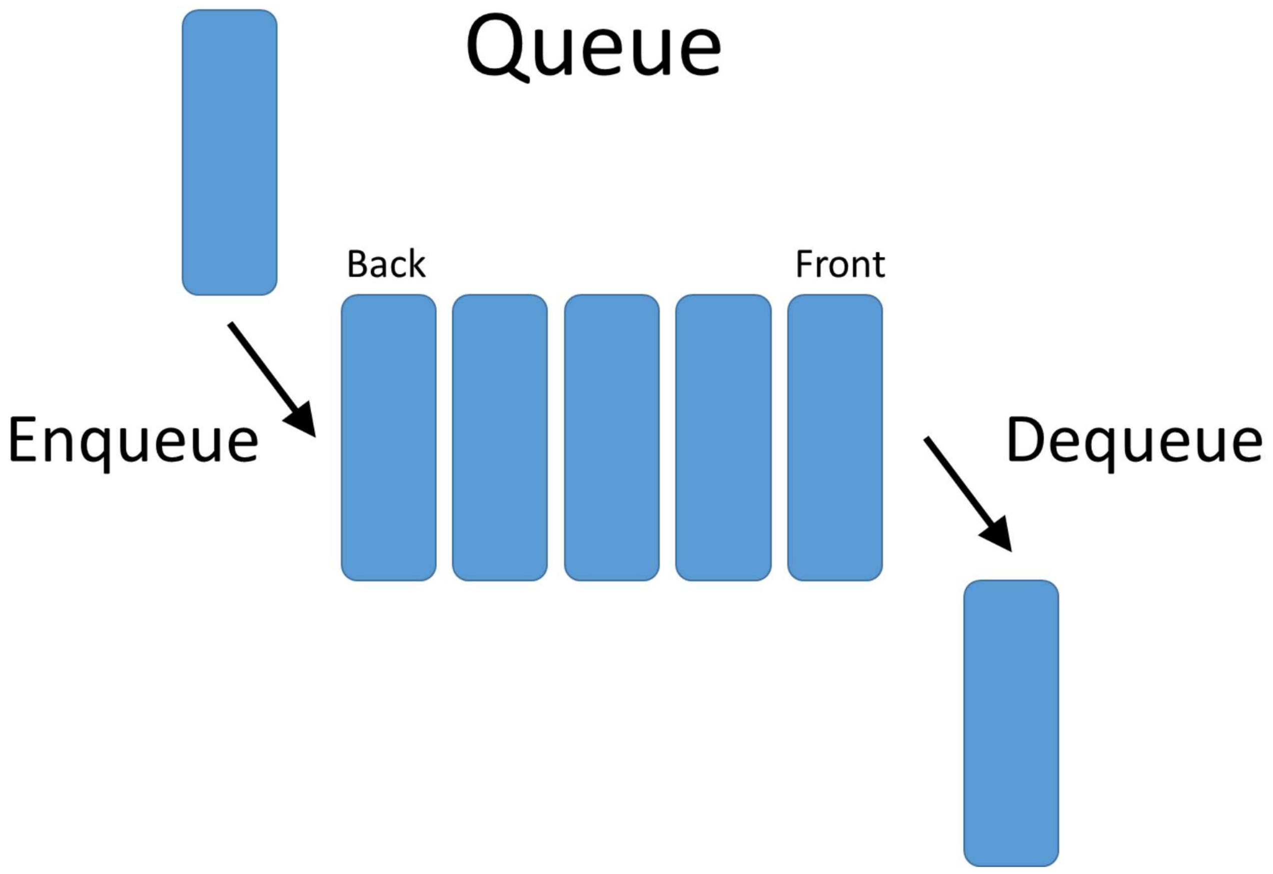
The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called **FIFO, first-in first-out**. It is also known as “first-come first-served.”

The simplest example of a queue is the typical line that we all participate in from time to time. We wait in a line for a movie, we wait in the check-out line at a grocery store, and we wait in the cafeteria line. The first person in that line is also the first person to get serviced/helped.

Let's see a diagram which shows this and compares it to the Stack Data Structure:

In [1]:
```
from IPython.display import Image
url = 'https://netmatze.files.wordpress.com/2014/08/queue.png'
Image(url)
```

Out[1]:



Note how we have two terms here, **Enqueue** and **Dequeue**. The enqueue term describes when we add a new item to the rear of the queue. The dequeue term describes removing the front item from the queue.

Let's take a look at how pop and push methods would work with a Queue (versus that of a Stack):

In [2]:
```
url2 = 'http://www.csit.parkland.edu/~mbrandyberry/CS2Java/Lessons/Stack_Queue/images/QueuePu
Image(url2)
```

```
---------------------------------------------------------------------------
HTTPError                                 Traceback (most recent call last)
<ipython-input-2-a944ec5ccf87> in <module>
      1 url2 = 'http://www.csit.parkland.edu/~mbrandyberry/CS2Java/Lessons/Stack_Queue/image
s/QueuePushPop.jpg'
----> 2 Image(url2)

~/.local/lib/python3.8/site-packages/IPython/core/display.py in __init__(self, data, url, fil
ename, format, embed, width, height, retina, unconfined, metadata)
   1229         self.retina = retina
   1230         self.unconfined = unconfined
-> 1231         super(Image, self).__init__(data=data, url=url, filename=filename,
   1232                 metadata=metadata)
   1233 

~/.local/lib/python3.8/site-packages/IPython/core/display.py in __init__(self, data, url, fil
ename, metadata)
    635             self.metadata = {}
    636 
--> 637         self.reload()
    638         self._check_data()
    639 

~/.local/lib/python3.8/site-packages/IPython/core/display.py in reload(self)
   1261         """Reload the raw data from file or URL."""
   1262         if self.embed:
-> 1263             super(Image,self).reload()
   1264             if self.retina:
   1265                 self._retina_shape()

~/.local/lib/python3.8/site-packages/IPython/core/display.py in reload(self)
    665             # Deferred import
    666             from urllib.request import urlopen
--> 667             response = urlopen(self.url)
    668             data = response.read()
    669             # extract encoding from header, if there is one:

/usr/lib/python3.8/urllib/request.py in urlopen(url, data, timeout, cafile, capath, cadefaul
t, context)
    220     else:
    221         opener = _opener
--> 222     return opener.open(url, data, timeout)
    223 
    224 def install_opener(opener):

/usr/lib/python3.8/urllib/request.py in open(self, fullurl, data, timeout)
    529         for processor in self.process_response.get(protocol, []):
    530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
    532 
    533         return response

/usr/lib/python3.8/urllib/request.py in http_response(self, request, response)
    638         # request was successfully received, understood, and accepted.
    639         if not (200 <= code < 300):
--> 640             response = self.parent.error(
    641                 'http', request, response, code, msg, hdrs)
    642 

/usr/lib/python3.8/urllib/request.py in error(self, proto, *args)
    567         if http_err:
    568             args = (dict, 'default', 'http_error_default') + orig_args
```

```
--> 569             return self._call_chain(*args)
    570 
    571 # XXX probably also want an abstract factory that knows when it makes

/usr/lib/python3.8/urllib/request.py in _call_chain(self, chain, kind, meth_name, *args)
    500         for handler in handlers:
    501             func = getattr(handler, meth_name)
--> 502             result = func(*args)
    503             if result is not None:
    504                 return result

/usr/lib/python3.8/urllib/request.py in http_error_default(self, req, fp, code, msg, hdrs)
    647 class HTTPDefaultErrorHandler(BaseHandler):
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
    650 
    651 class HTTPRedirectHandler(BaseHandler):

HTTPError: HTTP Error 404: Not Found
```

Conclusion

You should now have a basic understanding of Queues and the FIFO principal for them. In the next lecture we will implement our own Queue class!