Chapter 3 – Basic Data Structures (Linear Structures)

We will start our study of Data Structures with Linear Structures (based upon a list), which are the simpler ones. There are 3 main linear ADTs that we will study: Lists, Queues, and Stacks. (The book talks about a Deque (pronounced "Deck"), which we will not do.)

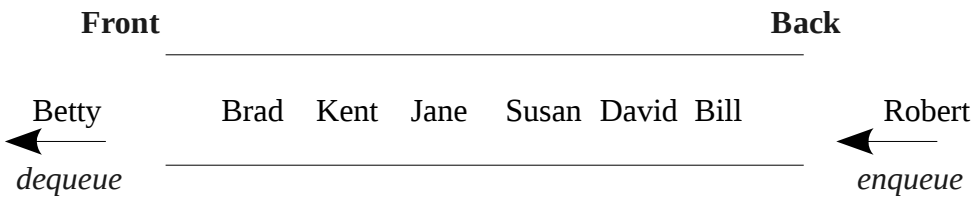
Because Python already has a **list** type, we will do the list last.

Queues

Queue is the British word for a line. When you go to the bank, you get in a *queue*. It is a structure that provides First-Come First-Serve sequencing. It is also known by the property *First-In First-Out (FIFO)*. That means that the first entity that goes into the queue is the first entity that comes out of the queue.

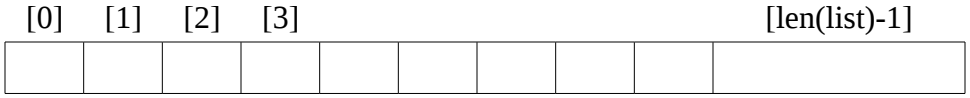
A queue has many uses in Computer Science. It is used anytime there are things that need attention or serviced, but for which they all can't be dealt with immediately. Jobs sent to a printer are queued up to be printed one at a time. Operating Systems use queue extensively for programs that need to use the CPU, requests to load data into memory, requests to read or write to the disks, etc.

A queue ADT can be thought of abstractly as a pipe. It has a front and a back. Entities go in the back and out the front, in the same order they went in. For example:



The operation of putting something into the queue is called *enqueue* and the operation of taking something out is called *dequeue*. Another useful operation is called *peek*, where we get the item at the front, but don't remove it from the queue. Dequeue will remove and return it.

In order to create this ADT, we need to have something by which we store the entities in the queue and in a way so that we know the order and we know which is the front and which is the back. Python's list feature makes this easy. We can simply have a list that stores the values, and we put values into the list in one end and take them out from the other. The question is: **Which end should be the front and which should be the back**?



Remember that a list is indexed start at zero. So the first item in the list is **list[0]**, the second is **list[1]**, and the last one is **list[len(list) – 1]** . Python has several operations that we can use to manipulate lists. The ones that will be of most helpful to us are: **list.insert(i, value)**, **list.append(i)**, **list.pop(i)**. **Insert** will insert a new value at position **i** and shift all of the values after **i** down one position to make room.

Pop will remove and return the value at position **i** and shift all the values to the right of **i** down one position. Since these two operations require moving all the values after position **i**, they both require **O**(*N*-*i*) steps, where *N* is the length of the list. If we insert at the beginning of the list (e.g. **list.insert(0, value)**), then the operation is **O**(*N*). If we pop from the beginning of the list (e.g. **list.pop(0)**), then the operation is **O**(*N*). If pop is used without an argument for **i** (e.g. **list.pop()**), then it pops from the back of the list. Popping from the back of the list is **O**(1) because it doesn't have to shift any values. Similarly, adding to the end of the list (using **append**) does not require shifting values. Therefore, **append** is **O**(1).

As a demonstration, I timed how long it would take to do 1,000,000 consecutive operations on a list. These are the times it took on my laptop:

```
The time to append 1000000 items is 1.0515 seconds.
The time to pop 1000000 items from the front is 1635.7782 seconds.

The time to insert 1000000 items at the front is 1515.8664 seconds.
The time to pop 1000000 items from the end is 1.0315 seconds.
```

If we use the zero position as the front of the queue and the last position as the back, then we could enqueue using append, but we would have to dequeue using **pop(0)**. So enqueue would be **O**(1) whereas dequeue would be **O**(*N*). If we use the zero position as the back of the queue and the last position as the front, then we would enqueue using **insert(0, value)** and dequeue using **pop()**. So enqueue would be **O**(*N*) whereas dequeue would be **O**(1). Either way we choose, one of the two operations is costly (**O**(*N*)) and one is cheap.

Operation	**Complexity if position [0] is the back and position [len(list)-1] is the front**	**Complexity if position [0] is the front and position [len(list)-1] is the back**
enqueue()	**O**(*N*)	**O**(1)
dequeue()	**O**(1)	**O**(*N*)

The textbook chose to use the zero position of the list as the back. So enqueue uses **list.insert(0, value)** and dequeue using **list.pop()**. In the lab, you can use either. The implementation of a queue in Python is shown in Listing 1. Switching the ends of the queue is implemented in Listing 2.

(Note: When we get to creating our own list (by chaining values together), we will be able to do both operations in **O**(1). There is also a way to use a list so as to make both operations **O**(1), but we won't do that now.)

The constructor starts the **__items** with an empty list, which represents an empty queue. Which ever end of the list is the front of the queue, it is only a queue if enqueue and dequeue work on *opposite* ends of the list.

```
class myQueue:
    def __init__(self):
        self.__items = []

    def enQueue(self, x):
        self.__items.insert(0, x)

    def deQueue(self):
        if (self.isEmpty()):
            return None
        else:
            return self.__items.pop()

    def peek(self):
        if (self.isEmpty()):
            return None
        else:
            return self.__items[self.size()-1]

    def size(self):
        return len(self.__items)

    def isEmpty(self):
        return self.size() == 0
```

Listing 1: Queue implementation (using the zero position as the back)

Notice that the data member **__items** starts with two underscores ("__"). Having a data member start with two underscores makes the data member private and therefore inaccessible by any code that uses this class. This is *information or data hiding*. The only way to manipulate a queue is by using the operations. This allows us to ensure that the queue is not corrupted by the user making a mistake. It also gives us the freedom to change the implementation. We may decide to use our own list instead of Python's list to make both operation **O**(1). If a user wrote a program to used **items** directly, changing the implementation would break that code.

Also notice that the **isEmpty()** operation uses the **size()** operation. We could just as easily implemented **isEmpty()** by returning **len(self.__items) == 0**. It is good programming practice to use other operations as much as possible (although this is a little slower) because it reduces the amount of code that must be debugged. If we have already tested the **size()** operation and we know it works, then the **isEmpty()** operation will work too.

```
class myQueue:
    def __init__(self):
        self.__items = []

    def enQueue(self, x):
        self.__items.append(x)

    def deQueue(self):
        if (self.isEmpty()):
            return None
        else:
            return self.__items.pop(0)

    def peek(self):
        if (self.isEmpty()):
            return None
        else:
            return self.__items[0]

    def size(self):
        return len(self.__items)

    def isEmpty(self):
        return self.size() == 0
```

Listing 2: Queue implementation (using the zero position as the front)

Stacks

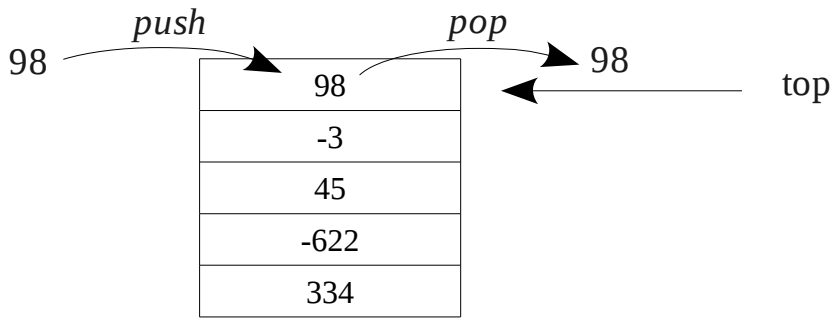
A *stack* is a data structure where, conceptually, items are placed on the top and removed from the top. Think about a stack of dishes. When you clean a dish, you put it on the top (typically). When you want to use a dish, you remove the one from the top. This property is known as *Last-in First-Out (LIFO)*. Items are removed from the stack in the reverse order in which they were put in.

Stacks can be used for things like matching parentheses, brackets, and braces. Parentheses must have a matching closing parenthesis for every opening parenthesis. They can be nested, but they cannot overlap. For example, “({ [ ] { } } ( ))” is valid but “({ [}])” is not, nor is “({ [ ] ”, nor is “ { } [ ])”.

Another problem for which a stack is used is converting an expression in *in-fix* notation to *post-fix* notation. In-fix notation is where the operators are between the operands. Post-fix notation is where the operator follows both operands. For example “2 + 3” means to add 2 and 3 in in-fix notation. That same expression in post-fix is “2 3 +”. HP calculators use to use post-fix notation. What is nice about post-fix is that parentheses are not needed to determine operation order. For example, the expression “(2 + 3) * 4” needs the parentheses in order to force the addition before the multiplication. In post-fix, this can be written as “2 3 4 * +”. Without the parentheses, “2 + 3 * 4” means do the multiplication first, which can be written as “2 3 + 4 *”. The conversion from in-fix to post-fix is easy to do using a stack. Once the expression is in post-fix a stack again can be used to calculate the result.

Another problem for which a stack is used is the implementation of function calls. The return from a sequence of function calls is in the reverse order in which they were called. For example, if the **main()** function calls a function **f()**, and **f()** calls **g()**, and **g()** call **h()**; when **h()** returns, execution goes back to **g()**. When **g()** returns, execution goes back to **f()**, which returns back to **main()**. So storing the return location on a stack allows for this return sequence to happen correctly.

The two main operations on a stack are called "push" (putting a value on the top of the stack) and "pop" (removing and returning a value from the top of the stack). Another useful operation is called *peek*, where we get the item at the top, but don't remove it from the stack. Pop will remove and return it.



With other programming languages, we would likely need to have an instance variable called "top" to keep track of where the top is, but with Python it isn't necessary. Since adding (appending) to the end of the list and popping from the end of the list are both **O**(1), we will make the end of the list the top.

```
class myStack:
    def __init__(self):
        self.__items = []

    def push(self, x):
        self.__items.append(x)

    def pop(self):
        if (self.isEmpty()):
            return None
        else:
            return self.__items.pop()

    def peek(self):
        if (self.isEmpty()):
            return None
        else:
            return self.__items[self.size()-1]

    def size(self):
        return len(self.__items)

    def isEmpty(self):
        return self.size() == 0
```

Listing 3: Stack implementation