#INTRO

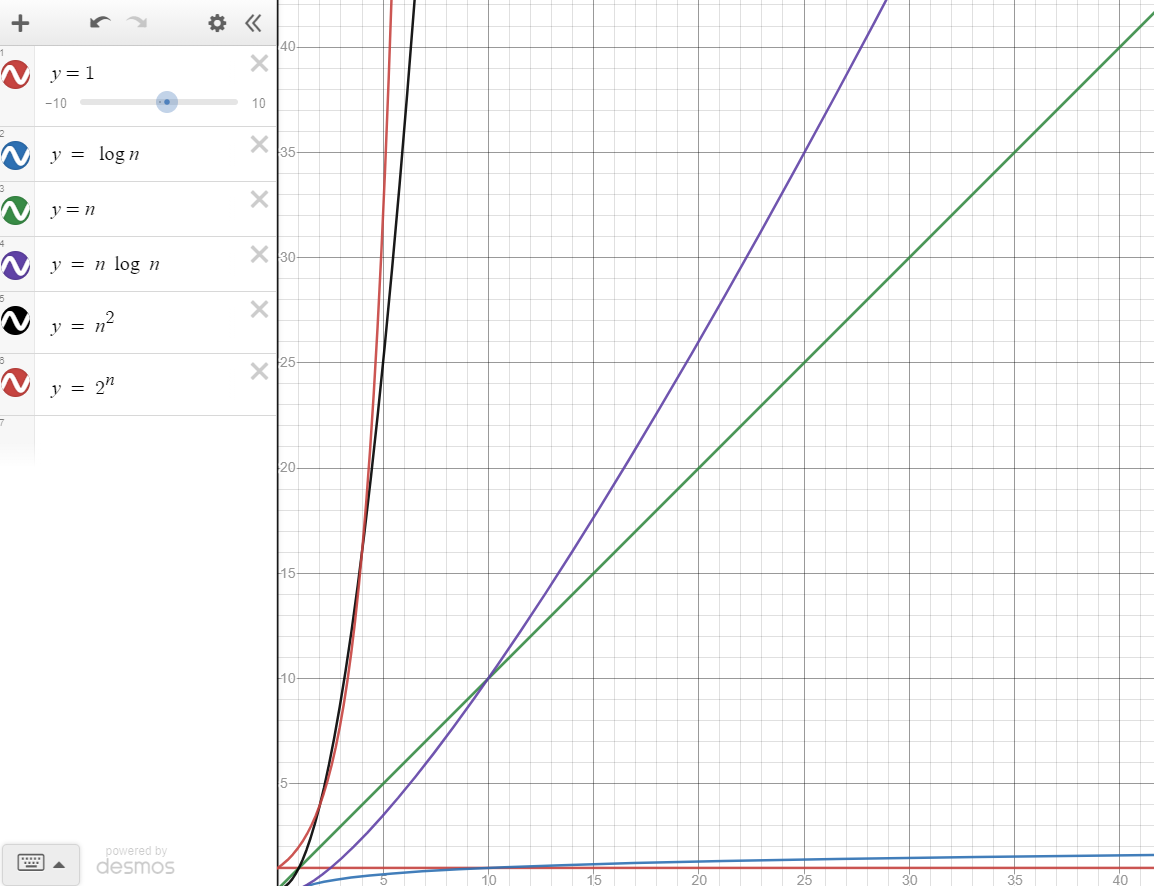
Dynamic array

A dynamic array is common data structure that is versatile in its ability to store and retrieve memory. A major advantage of a dynamic array is its ability to expand automatically. Normal arrays are initiated with a size and cannot expand or contract. The adaptability a dynamic array allows a programmer is great as many programs are create to work with customizable settings.

Big (o) notation

Big O notation is a metric used in software creation to measure the efficiency of a program. Rarely is a program ran with the exact same set of data every time. Most programs are there to perform a specific set of actions or manipulations with many unique data sets. There are many (all) roads that lead to Rome. But we could probably agree this is not the most *efficient* route to Rome from Venice.

As such, there are an infinite number of ways to code a solution to a problem. But there are ways to make it more efficient. The most efficient program would take the same amount of time to run for one piece of data as it would for 1 billion pieces of data. This is called O(1) efficiency. There are many different levels of efficiency in big O notation:



**O(1)** – Constant Time: The size of data to has no effect on the time it takes to run a program.

**O(log n)** – Logarithmic Time: The execution time increase decreases with each additional item of data.

**O(n) –** Linear Time: The execution time increases with relation to the size of the data.

**O(n log n)** -- Logarithmic Time

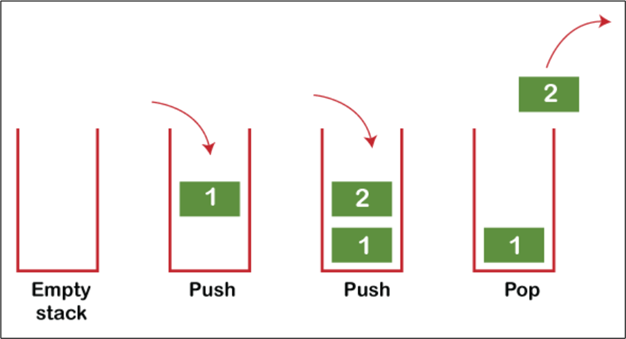
**O(n2)** – Polynomial Time: The execution time increases by a fixed power in relation to the input size.

**O(2n)** – Exponential Time: The execution time increases by a power equal to size of the input.

A picture containing text, clipart

Description automatically generated

Stacks are a great data structure to be able access data quickly and efficiently. A stack stores data inside of a dynamic array. This allows the amount of data stored to increase or decrease as necessary providing great versatility. In a stack, data is always added or removed from the end. In other words, the last piece of data entered a stack, is always the first out (LIFO).



Efficiency –

A fantastic advantage of a stack is the algorithmic efficiency. Adding to or removing from a stack is always O(1). We always add to and remove from the end of the dynamic array. The execution time of this does not depend on the size of the array because none of the elements must shift. Contrarily, if we added to or removed from the beginning, the efficiency would be O(n). This is because each element in the array would be required to shift for each item added or removed.

Real life example of stacks –

Gun magazines – To get to the first bullet entered, you must remove all the bullets that were entered after it. Unless you are a psychopath and cut it open like the guy who made this image. Then you need a course in human decency.

Stack of oreos: According to the Geneva Convention of 1929, you are only allowed to take or add oreos from the top of the pile. Any other method is considereda war crime and punishable by death.

– Abraham Lincon; S*ource: definitely not Wikipedia*

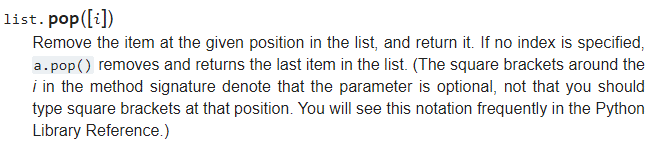
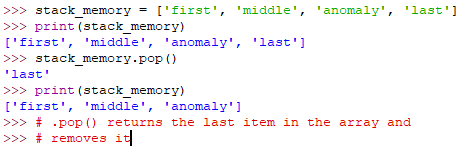
Programming applications –

\*\**The above photos are the only two officially licensed keyboards of Stack Overflow.*

Stacks are a great tool in programming. They are often used in undo/redo features in programs or other applications where quick, ordered memory is to be accessed.

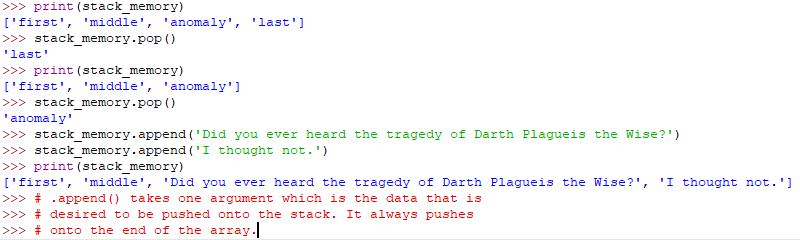
Stack memory is also used while running a program. The computer must remember where is has been, and where it must go to. A great example of this is with functions. Functions are defined throughout code. As a program progresses, it often needs to jump to different parts of the code to run instructions found there. But it must also remember where it was so that it can return to compiling to program in the correct order. Stack memory allows the computer to quickly see where it must go, by adding to the top of the stack, and where it must return to, by removing the item on top to uncover the last known bit of memory, or location to return to in the program.

Handy python functions

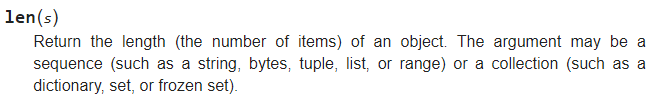
**pop() –** the pop() function is core to stack memory’s function. When this function is called, the last item in the array is returned. It is important to remember that it also removes that item from the array.

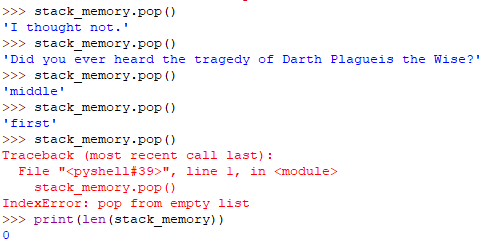
**append() –** to add onto the stack, one must use the append() function in python.





**len() –** the len() function takes one parameter, in this case, the array. Since the pop() function removes data from an array, it is important to make sure there is data present to being with. If one calls the pop() function on an empty array, they will encounter an error.

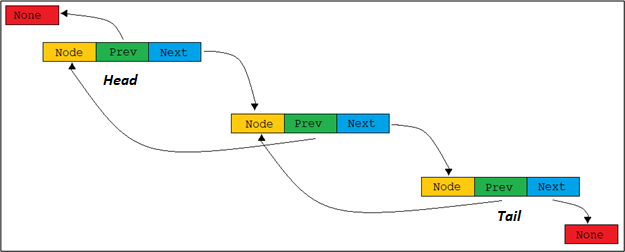




* + 1. Problem 1: mostly solved; student must correct syntax
    2. Problem 2: not solved; student must implement solution

Linked Lists

A linked list is another very useful type of data structure. In Python, an array is called a “list” so one would assume that a linked list would be very similar because they are both “lists”. That is incorrect as the only applicable similarity that a Python “list” (array) shares with a linked list is the ability to store data in memory. A fundamental principle of arrays is that they are contiguous in memory. This is not so with linked lists.

Each item in a linked list can be anywhere in memory. Because of this, each item has “directions” that guide to the next item in the list. These directions are contained in each “link” called a **node**. A node contains two parts, the data stored and a memory address book that contains address for the previous and following nodes.

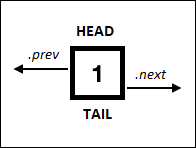
The nature of the linked list makes it impossible to jump directly to an item in the list. One must start from either the front, called the head, or the tail of the list and iterate through each “link” until the desired data is found. Because of this, looking up a value in a linked list is O(n).

**Inserting into a Linked List:**

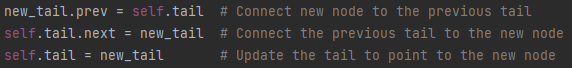
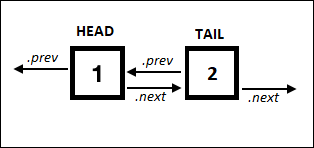
In a dynamic array, inserting or removing data anywhere but the end is O(n) because of the sequential nature of memory used to store the data causes all the elements stored to shift to accommodate the change.

A linked list is always O(1) to insert or remove from the head or tail, but it is O(n) to removing from the middle, because you must iterate through the list to find the node you desire to add/remove. When one adds or removes an element, care needs to be taken in the process since each element contains the addresses of the next and previous elements.

**Creating a new linked list:**

If there are no elements in a linked list, then creating a new list is simple. First, we need to create a new node and set it to be both the head and the tail:

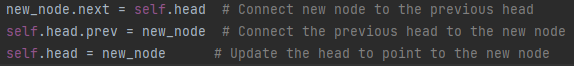
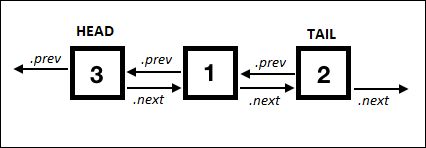
1. Set the new\_node as the head 🡪 self.head = new\_node
2. Set the new\_node as the tail 🡪 self.tail = new node

Now what if we want to add another element at the end and create a new tail?

Let us call the new element *new\_tail.*

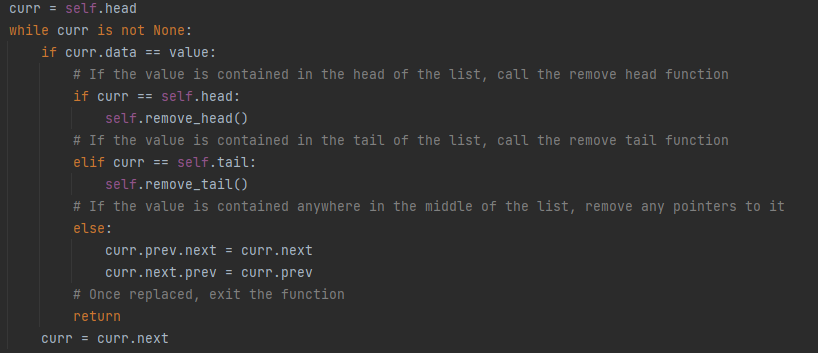
1. We first need to connect the new\_tail to the current tail: *new\_tail.prev = self.tail*
2. Now we need to attach the current tail to the new\_tail: *self.tail.next = new\_tail*
3. Now we need to make new­­­\_tail the current tail: *self.tail = new\_tail*

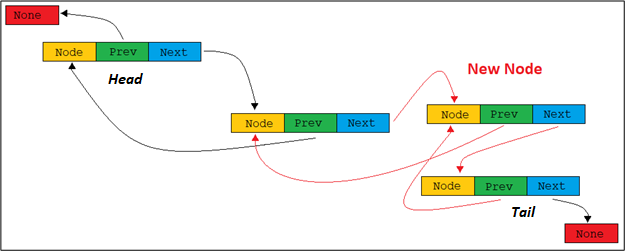
The order here is very important. If these steps are done out of order you could lose the address of the element that had been before the original tail. It is important to attach the new node before removing the existing ties between nodes. This ensures that we still have links to all the data.

What if you wanted to add a new head?

The order is

new\_node.prev = curr # Connect new node to the node containing 'value'  
new\_node.next = curr.next # Connect new node to the node after 'value'  
curr.next.prev = new\_node # Connect node after 'value' to the new node  
curr.next = new\_node # Connect the node containing 'value' to the new node



What about adding in the middle of the linked list? This is a little trickier as the middle of the linked list may not be directly linked to either the head or the tail. We would need to know what the value is that we want to insert it after, so a while loop is very helpful here:

In the above picture, the red arrows show how the new node was inserted into the linked list before the tail. The 2nd node and the tail do not point to each other anymore, rather they now provide an address to the new node.

***self.head*** – this is the syntax that refers to the beginning element in a linked list.

***self.tail*** -- this is the syntax that refers to the last element in a linked list.

***self.next*** – for our purposes, *self.next* is the syntax that points to the next element in the list. *ex. self.head.next* refers to the element immediately following the head.

***self.prev*** *–* for our purposes, *self.prev* is the syntax that points to the previous element in the list. *ex. self.tail.prev* refers to the element immediately before the tail. **Note:** *self.head.prev* and *self.tail.next* should always equal *None* since they are on the ends of the linked list.

***self.data*** – this is the syntax that allows one to access the data stored in the node.

* 1. Problems to solve:
     1. Problem 1: mostly solved; student must correct syntax
     2. Problem 2: not solved; student must implement majority of solution