Lab 3: Lists, How Do They Work?

In Python we can quite easily append items to a list. It is as simple as using the built in `append` function. However, as we discussed in class, appending to an array is not actually a straightforward task. Unlike linked lists, to increase the size of a list stored in an array you have to allocate an entirely new array of a bigger size and copy over all the existing elements to that new array.

In this lab we will delve deeper into the `append` function in Python to try to figure out what algorithm is uses. Figuring out how a program works is a common task for hackers and cyber professionals called reverse engineering (https://en.wikipedia.org/wiki/Reverse_engineering). There are several ways to reverse engineer a program. One is to get access to the source code and directly examine it. Many times you will not have access to the source code, however if you have access to a binary program you can often decompile (https://en.wikipedia.org/wiki/Decompiler) it to obtain an approximation of the original source code which will also allow you to determine how it works. In this lab we will attempt the hardest type of reverse engineering called *black-box reverse engineering*, where we can only run the program (in this case the `append` function) and see its behavior but do not have any access to its code.

To do this, we will create our own implementation of a list with several different algorithms for the append function. Then, we will benchmark our algorithms against the built-in `append` function to see which one behaves similarly.

The Assignment

Copy the starter code here (labs/sylist.txt) into a file called `sylist.py`. This will be where you do your work. I have started a class for you called `SYList` where you will try to recreate the functionality of the built-in Python list class. I have already made a constructor and get and set functions so that you can use the bracket operator `[]` to read and write elements from the list. Additionally, I have provided a `malloc` function that will allocate a fixed size array for you, like in C.

Important: note that in the constructor there are two class variables related to the size of the list, `length` and `capacity`. The intent of these variables is that you can have a list with an array that has space to store a certain number of items (that would be the `capacity`) while the list itself contains a fewer number of items (the `length`). For example, a list could contain the numbers `[1, 2, 3]` but have an array with capacity 5 that actually looks like this: `[1, 2, 3, None, None]`. This would mean that the array has two "empty" slots which can be filled without having to resize the array.

Your job is to finish the four remaining functions:

1. `__str__()`: The string function should return a string representing all the elements in the list. Note that this might not be the same thing as saying `str(self.array)`, if the capacity and the length are not the same. Considering the above example, the string function should return `[1, 2, 3]` not `[1, 2, 3, None, None]`
2. `append(value)`: This should add one new element to the list. If there is no remaining capacity in the list, i.e. `self.capacity == self.length`, a new array should be allocated, using the `malloc` function I have provided, with size `self.capacity + 1` (one larger than the old array) and everything should be copied to the new array. **Important**: when you make a new array `self.array` should be set to that array after you are done copying everything over.
3. `append5(value)`: It seems rather wasteful that every time a new item is added to the list it has to be entirely resized. What if instead you added some extra "slack" space so that multiple elements can be appended

before it has to be resized? This function should be the same as above, except when the list is out of capacity you should allocate a new array with 5 extra empty slots, increasing the capacity by 5.

4. `appendDouble(value)`: If 5 is good why not more? Same as above, except when the list is out of capacity you should allocate a new array with double the number of slots that the old array had. Note that if the old capacity was 0 the new array should have a capacity of 1.

Note that in all three functions if there is an "empty" slot in the array, i.e. capacity > length, then the item being appended just goes in the first empty slot. The difference in behavior between the three only comes up when the array is full, capacity == length. In that case append adds one new slot, which is immediately filled, append5 adds five new slots, one of which is filled, and appendDouble doubles the slots, one of which is filled. Consider the following execution as an example:

```
>>> from sylist import *
>>> l = SYList()
>>> l.append5(10)
>>> l.append5(5)
>>> print(l[1])
5
>>> print(l)
[10, 5]
>>> print(l.array)
[10, 5, None, None, None]
>>> print(len(l))
2
```

After two items were added to the list, the array had 5 slots (because I used append5 which adds 5 new slots at a time when the array is full) but only 2 of them were used. Three more items could be added to this list before it needed to be resized again.

Benchmarking

To determine which one of these algorithms is closest to what Python's `append` function does, we have to benchmark all of them and compare. I have written a script here (labs/sylistbench.txt) for you to do exactly that. It will run all four functions on inputs from 10,000 up to 100,000 and create a CSV file named benchmark.csv with the amount of time *per call of the append function* that it takes for each algorithm to complete. This script will take a while to run (10-20 minutes depending on your VM resources).

After running this script use Excel or Google Docs (or something else) to graph the data. **Answer the following questions in a file called README.**

1. What do you think is the Big-O runtime of a single call of the function append? Why?
2. What do you think is the Big-O runtime of a single call of the function append5? Why?
3. What do you think is the Big-O runtime of a single call of the function appendDouble? Why?
4. Which of the approaches in do you think is actually used to implement the append function in Python? Why?
5. Python's lists also have an insert function, which inserts an element in the middle of the list. What do you believe would be the runtime of this insert function? Why? Your explanation should say what operations would have to be performed on the array during an insert.

Submission

Please submit your code in a file `sylist.py` as well as your `README` file answering the lab questions.