Implementación y Análisis de Complejidad de Listas

Juan Sebastián Rueda Segura

Estructuras de Datos

Universidad Nacional de Colombia Sede Bogotá

13 de noviembre de 2024

Índice

1. Introducción

El presente documento tiene como objetivo estudiar la implementación de la estructura de datos **List**, analizando los algoritmos para estudiar su funcionamiento y complejidad mediante Big O. Se presenta la implementación en Python con su respectivo código, que tiene tipadas las variables, como se pidió en clase.

2. Análisis del código

Para esta sección se va a presentar el código correspondiente al método a examinar, los comentarios y las explicaciones pertinentes y finalmente la complejidad de cada uno de ellos dada en notación Big O.

2.1. Librerías a utilizar

```
import time
import matplotlib.pyplot as plt
```

Se trabajará con la librería *time* para medir el tiempo y *matplotlib* para graficar los resultados obtenidos para cada tamaño.

2.2. Inicialización clase Lista

```
class Lista:
    def __init__(self, size: int) -> None:
        self.size: int = size
        self.last_element_index: int = -1
        self.array: list[int] = [None for i in range(0,size)]
```

Se inicializará un arreglo estático de tamaño *size*, que se da al momento de inicializar Lista. A su vez un entero last_element_index que indica la posición del último elemento de la lista. Comienza en -1 ya que simboliza que está vacía la lista.
Complejidad: $O(1)$

2.3. Análisis método *pushBack*

```
def pushBack(self, elem: int) -> None:
        if self.last_element_index == self.size - 1:
            print("Error: Lista llena. No se puede agregar el
    elemento.")
        else:
            self.array[self.last_element_index + 1] = elem
            self.last_element_index += 1
```

Primero se revisa si la lista está llena o no, para saber si se puede insertar un elemento. Se agrega este al final de la lista y se incrementa en 1 el índice del último elemento. Note que este procedimiento se puede hacer en tiempo constante, ya que no hay que hacer modificaciones al arreglo estático.
Complejidad: $O(1)$

2.4. Análisis método *popBack*

```
def popBack(self) -> None:
        if self.last_element_index == -1:
            print("Error: Lista vacia. No se puede eliminar el
    elemento.")
```

```
        else:
            self.last_element_index -= 1
```

Primero se revisa si la lista está vacía o no, para saber si se puede eliminar un elemento. Simplemente se puede reducir el índice del último elemento, puesto que se sobreescribirá del arreglo el elemento al momento de insertar otro. Note que este procedimiento se puede hacer en tiempo constante, ya que no hay que hacer modificaciones al arreglo estático. Finalmente se decrementa el índice del último elemento.
Complejidad: $O(1)$

2.5. Análisis método *popFront*

```
    def popFront(self) -> None:
        if self.last_element_index == -1:
            print("Error: Lista vacia. No se puede eliminar el
    elemento.")
        else:
            for i in range(0, self.last_element_index + 1):
                try:
                    self.array[i] = self.array[i+1]
                except:
                    pass
            self.last_element_index -= 1
```

Primero se revisa si la lista está vacía o no, para saber si se puede eliminar un elemento. En este método, sí se deben hacer cambios al arreglo estático puesto que se deben mover todos los elementos una posición antes. Se deben sobreescribir. En caso de que la lista esté llena, y sea de tamaño n, esto tomará n pasos. Es por esto que la complejidad será lineal y dependerá del tamaño de la lista. Finalmente se decrementa el índice del último elemento.
Complejidad: $O(n)$

2.6. Análisis método *pushFront*

```
    def pushFront(self, elem: int) -> None:
        if self.last_element_index == self.size - 1:
            print("Error: Lista llena. No se puede agregar el
    elemento.")
        else:
            for i in range(0, self.last_element_index + 1):
                self.array[self.last_element_index + 1 - i] = self.
    array[self.last_element_index - i]
            self.array[0] = elem
            self.last_element_index += 1
```

Primero se revisa si la lista está llena o no, para saber si se puede insertar un elemento. Similar al método anterior, se deben mover todos los elementos del arreglo, ya que al insertar al comienzo, se mueven todos los índices. Una vez se mueven, se inserta el elemento en la posición 0 y se incrementa el índice.
Complejidad: $O(n)$

2.7. Análisis método *find*

```
    def find(self, elem: int) -> int:
        for i in range(0, self.last_element_index):
            if self.array[i] == elem:
                return i
        return -1
```

Se recorre la lista buscando una posición donde el elemento en la posición i-ésima coincida con el elemento a buscar. Se retorna el primer índice de esta ocurrencia. En caso de no encontrarlo, devuelve −1.
Complejidad: $O(n)$

2.8. Análisis método *addBefore*

```
    def addBefore(self, elem: int, insert :int) -> None:
        indx: int = self.find(elem)
        if indx != -1:
            if self.last_element_index == self.size - 1:
                print("Error: Lista llena. No se puede agregar el
    elemento.")
            else:
                for i in range(indx-1, self.last_element_index + 1)
    :
                    self.array[self.last_element_index + 1 - i] =
    self.array[self.last_element_index - i]
                self.array[indx-1] = insert
                self.last_element_index += 1
        else:
            print("Element not in list")
```

Primero se revisa si el elemento al que se le quiere insertar detrás está o no en la lista. En caso que esté, se revisa si la lista está llena y se realiza una inserción como la de pushFirst, pero desde la posición anterior al índice del elemento. Finalmente, se incrementa el índice del último elemento. Note que se deben recorrer dos veces la lista, luego la complejidad es de $O(2n)$ pero esta hace parte de la clase de $O(n)$.
Complejidad: $O(n)$

2.9. Análisis método *addAfter*

```
    def addAfter(self, elem: int, insert :int) -> None:
        indx: int = self.find(elem)
        if indx != -1:
            if self.last_element_index == self.size - 1:
                print("Error: Lista llena. No se puede agregar el
    elemento.")
            else:
                for i in range(indx+1, self.last_element_index + 1)
    :
                    self.array[self.last_element_index + 1 - i] =
    self.array[self.last_element_index - i]
                self.array[indx+1] = insert
```

```
                self.last_element_index += 1
        else:
            print("Element not in list")
```

Similar al método anterior, primero se revisa si el elemento al que se le quiere insertar detrás está o no en la lista. En caso que esté, se revisa si la lista está llena y se realiza una inserción como la de pushFirst, pero desde la posición posterior al índice del elemento. Finalmente, se incrementa el índice del último elemento. Note que se deben recorrer dos veces la lista, luego la complejidad es de $O(2n)$ pero esta hace parte de la clase de $O(n)$.
Complejidad: $O(n)$

2.10. Análisis método *erase*

```
    def erase(self, elem: int) -> None:
        indx: int = self.find(elem)
        if indx != -1:
            if self.last_element_index == -1:
                print("Error: Lista vacia. No se puede eliminar el
    elemento.")
            else:
                for i in range(indx, self.last_element_index + 1):
                    try:
                        self.array[i] = self.array[i+1]
                    except:
                        pass
                self.last_element_index -= 1
        else:
            print("Element not in list")
```

Se llama al método *find* para saber si el elemento está en la lista o no. En este caso, se usa un algoritmo similar al de popFirst, pero desde la posición del elemento. FInalmente se decrementa en 1 el índice del último elemento.
Complejidad: $O(n)$

2.11. Análisis método *printf*

```
    def printf(self) -> None:
        print(f"Elementos en el arreglo: [", end= "")
        for i in range(0, self.last_element_index + 1):
            if i != self.last_element_index:
                print(self.array[i], end=" ")
            else:
                print(self.array[i], end="")
        print("]")
```

Se recorre la lista y se van imprimiendo los elementos uno a uno hasta que llegue al indice del último elemento.
Complejidad: $O(n)$

2.12. Análisis método *empty*

```
    def empty(self) -> None:
        self.last_element_index = -1
```

Se actualiza el índice del último elemento a -1, como cuando se inicializa una lista vacía.
Complejidad: $O(1)$

3. Toma de tiempos

En esta sección analizaremos los tiempos obtenidos al crear listas de diferentes tamaños, y realizar operaciones que, según el análisis presentado anteriormente, son de tiempo constante o de tiempo lineal. Como son varios métodos y tamaños muy grandes como para probarlos todos, se examinaron simplemente dos, *pushBack* y *popFront* con complejidades constante y lineal respectivamente.

3.1. Análisis del código utilizado

```
sizes: List[int] = [100, 1000, 10000, 100000, 1000000, 10000000]
insertion_time: List[float] = []
remove_time: List[float] = []

for size in sizes:
    start = time.time()
    lst = Lista(size)
    for i in range(0, lst.size):
        lst.pushBack(i)
    end = time.time()
    time_measure = (end - start)/size
    insertion_time.append(time_measure)

    start = time.time()
    for i in range(0, lst.size):
        lst.popFront()
    end = time.time()
    time_measure = (end - start)/size
    remove_time.append(time_measure)

plt.plot(sizes, insertion_time, label="pushBack()")
plt.show()

plt.plot(sizes, remove_time, label="popFront()")
plt.show()
```

Se creó una lista con diferentes tamaños los cuales son los que vamos a analizar, y otras dos, donde se almacenarán los valores de inserción y de eliminación por operación , dependiendo del tamaño de la lista. Se mide el tiempo una vez creada la lista con un tamaño. Posteriormente, o se insertan o se eliminan elementos de esta y una vez finalice este proceso se detiene el tiempo. Este simboliza el tiempo

total. Como tanto para llenar la lista como para vaciarla se tienen n pasos, el tiempo por operación será de $\frac{tiempo_{final} - tiempo_{inicial}}{tamaño}$. Una vez tenemos las listas con los tiempos correspondientes al size, se grafican usando las funciones *plt.plot* y se muestran con *plt.show*.

3.2. Resultados obtenidos y análisis

```

pushBack: list[float] =  [0.0, 0.0, 0.0, 1.6487598419189454e-07,
1.0928702354431152e-07, 1.1763114929199219e-07]

popFront: list[float] = [1.768594086754930e-05, 5.09876545678913e
    -05
, 0.000365743902, 0.003546328, 0.00895487365, 0.026463829]
```

Después de esperar varios días a que compilase, se llegó a estos resultados, los cuales son acordes a los esperado; el tiempo en hacer una operación al insertar un elemento en la última posición de una lista es constante, como se ve en los datos al no variar casi unos datos de otros, siendo la diferencia de tamaños significativa. Además, la operación de eliminar el primer elemento sí posee cambios: para mayor tamaño, mayor tiempo se tarda en hacer una eliminación.

3.3. Gráficas asociadas a resultados

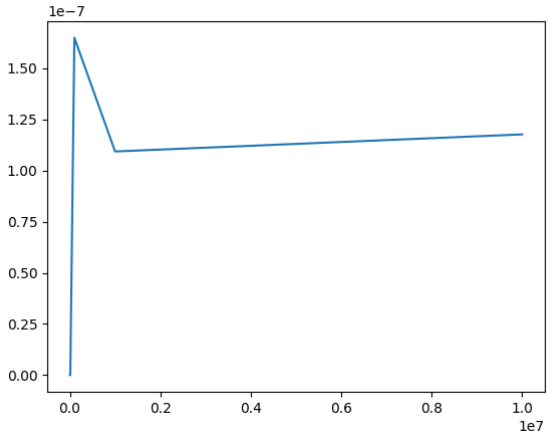

Figura 1: Gráfica de *Matplotlib* para el método *pushBack*

Aunque la Figura 1 tiene una forma extraña, esto se debe al hecho de que los tres primeros elementos de la lista *pushBack*, presentada en la sección anterior son 0.0 y no flotantes con más cifras. De ahí en adelante se puede ver cómo la gráfica tiene un comportamiento constante.

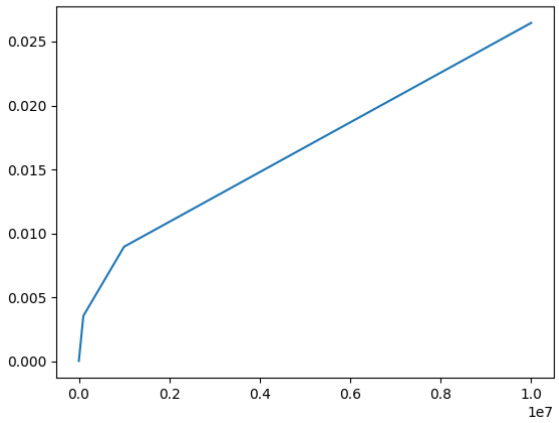

Figura 2: Gráfica de *Matplotlib* para el método *popFront*

En la Figura 2 se puede apreciar claramente el comportamiento lineal con el que crece el tiempo de eliminación a la hora de incrementar el tamaño de la lista *sizes*.

4. Conclusiones

A lo largo de este documento se analizó la implementación de la clase *Lista*, se comprobó su correcto funcionamiento mediante el sometimiento a pruebas que midiesen su eficiencia y a su vez verificasen el análisis de la complejidad computacional que toman los métodos asociados con resultados dados por el propio programa. Sin embargo, por la falta de tiempo para este trabajo, solo se pudieron presentar los gráficos presentes en el documento, aunque estos sumarizan el contenido correctamente.

Un pensamiento al que llegué desarrollando este documento es que me gustaría aprender a usar programación paralela, ya que este paradigma de programación permitiría unos resultados más eficientes a la hora de esperar a que el programa termine de compilar, lo que da un mayor número de pruebas y de tamaños, para tener gráficas aún más precisas y más completas para agilizar futuras entregas donde se deban hacer análisis similares. Además, valdría la pena haber hecho las pruebas con la misma lógica de implementación pero en otro lenguaje más veloz como C++ y contrastar los resultados. En futuras entregas intentaré hacer ese esfuerzo para satisfacer esa curiosidad y ver si realmente es tan notoria la diferencia.