INSTITUTO INFNET

ESCOLA SUPERIOR DE TECNOLOGIA

GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

Projeto de Bloco: Ciência da Computação

TP1

Alunos: Daniel Gomes Lipkin

15 de nov. de 2024

Lógica e Funcionamento

Geração de Arquivos

Foi utilizado o comando *tree* (Listagem de arquivos e diretórios em formato de árvore) com as opções *-Rf* (Recursivo + Caminho do arquivo/diretório aparente) e direcionado à um arquivo com > *out/output.txt*. O arquivo possui 10009 linhas e será usado como entrada para todas as partes da pesquisa.

```
├── ./linuxpb_tp1_993.txt
├── ./linuxpb_tp1_994.txt
├── ./linuxpb_tp1_995.txt
├── ./linuxpb_tp1_996.txt
├── ./linuxpb_tp1_997.txt
├── ./linuxpb_tp1_998.txt
├── ./linuxpb_tp1_999.txt
├── ./linuxpb_tp1_99.txt
├── ./linuxpb_tp1_9.txt
├── ./out
│   ├── ./out/out_bubble.txt
│   ├── ./out/output.txt
│   └── ./out/sort.py
├── ./output.txt
├── ./sub1
│   ├── ./sub1/sub1_0.txt
│   ├── ./sub1/sub1_1000.txt
│   ├── ./sub1/sub1_1001.txt
│   ├── ./sub1/sub1_1002.txt
```

(Um trecho do arquivo)

Algoritmos de Ordenação

Preliminarmente à definição dos algoritmos, temos :

```python
OUT_PATH = "./output.txt"

def swap(arr, a, b):
    #Troca posição a <-> b no vetor arr
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp


def getTime():
    #Tempo em segundos
    return time.perf_counter()
```

Primeiramente temos o Bubble Sort

```python
def bubbleSort(arr):
    noswap = False
    while not noswap:
        noswap = True
        for i in range(1, len(arr)):
            if arr[i] < arr[i-1]:
                swap(arr, i, i-1)
                noswap = False
    return arr
```

Que fica em um loop externo até que todas as execuções do loop interno não tenham efetuado a operação de trocar índices do vetor (arr), cujo acontece quando o índice (i-1) anterior é maior que o presente (i).

Depois temos o Selection Sort

```python
def selectionSort(arr):
    for i in range(len(arr)):
        cur_min = arr[i]
        for j in range(i+1, len(arr)):
            if arr[j] < cur_min:
                swap(arr, i, j)
            else:
                cur_min = arr[j]
    return arr
```

Que de um índice (i), percorre todo vetor para achar elementos menor que o menor elemento (cur_min), trocando ele de lugar com (i) e assinalando um novo elemento menor.

E finalmente temos o Insertion Sort

```python
def insertionSort(arr):
    for i in range(len(arr)):
        k = arr[i]
        j = i-1

        while (j >= 0 and arr[j] > k):
            arr[j+1] = arr[j]
            j -= 1

        arr[j + 1] = k
    return arr
```

Observe que é a variante in-place já que altera o próprio vetor de entrada. O loop externo percorre todo o vetor, enquanto um loop interno com índice (j) percorre o vetor desde (i) na ordem contrária. No loop interno o elemento atual (k ou arr[i]) vai sendo comparado com os elementos antecessores até que seja achado um menor que ele, onde trocará de lugar com o elemento posterior a esse (j+1).

Posteriormente temos a função helper para análise :

```python
def doSortFuncOnFile(path, func, out, times=10, limit=-1):
    f = open(path, "r") #arquivo entrada
    fo = open(out, "w") #arquivo saida
    time_avg = [] #tempos calculados
    for i in range(times):
        start_time = getTime()
        fo.write("".join(func(f.readlines()[:limit])))
        #executa a função func nos conteudos de f até a linha limit
        time_avg.append(getTime() - start_time)
    print(sum(time_avg)/len(time_avg), " segundos") #média de tempo
    f.close()
    fo.close()
```

```python
print("Bubble Sort (10k)")
doSortFuncOnFile(OUT_PATH, bubbleSort, "./out_bubble.txt")
print("Bubble Sort (5k)")
doSortFuncOnFile(OUT_PATH, bubbleSort, "./out_bubble2.txt", 10, 5000)
print("Bubble Sort (2k)")
doSortFuncOnFile(OUT_PATH, bubbleSort, "./out_bubble3.txt", 10, 2000)
```

Estrutura de Dados

Preliminarmente à definição das estruturas, temos :

```
OUT_PATH = "./output.txt"
TIMES = 40


def getTime():
    return time.perf_counter()


def getMem():
    return psutil.Process().memory_info().rss
```

(rss = Resident Set Size ou a memória RAM utilizada pelo processo corrente do Python)

Primeiramente temos a Pilha :

```
class DanStack:
    top = 0
    e = []

    def get(s, k):
        if k < s.top and k > -1:
            return s.e[k]
        return None

    def add(s, el):
        s.e.append(el)
        s.top += 1

    def remove(s, i=-1):
        i = min(i, max(0, s.top))
        if i < 0:
            i = s.top
        i = max(i - 1, 0)
        s.e = s.e[:i] + s.e[i+1:]
        s.top -= 1
```

- O top acompanha o tamanho da pilha
- remove(i) não permite índices negativos ou além do topo, truncando a entrada.

Depois temos a Fila :

```
class DanQueue:
    e = []
    size = 0

    def get(s, k):
        if k < s.size and k > -1:
            return s.e[k]
        return None

    def add(s, el):
        s.e = [el] + s.e
        s.size += 1

    def remove(s, i=-1):
        i = min(i, max(0, s.size - 1))
        if i < 0:
            i = s.size - 1
        s.size -= 1
        s.e = s.e[:i] + s.e[i+1:]
```

E finalmente a Hashtable

```
class DanTable:
    def __init__(s, _size=32):
        s.size = _size
        s.e = [None] * s.size

    def get(s, k):
        k = s.getHash(k)
        if k < s.size and k > -1:
            return s.e[k]
        return None

    def getHash(s, obj):
        return hash(obj) % s.size

    def set(s, elk, elv):
        elk = s.getHash(elk)
        if elk < s.size and elk > -1:
            s.e[elk] = elv
            return True
        return False

    def remove(s, elk):
        return s.set(elk, None)
```

- Note que os índices já são inicializados em memória com o valor None
- set(k,v) retorna se a operação teve sucesso.

E posteriormente os testes

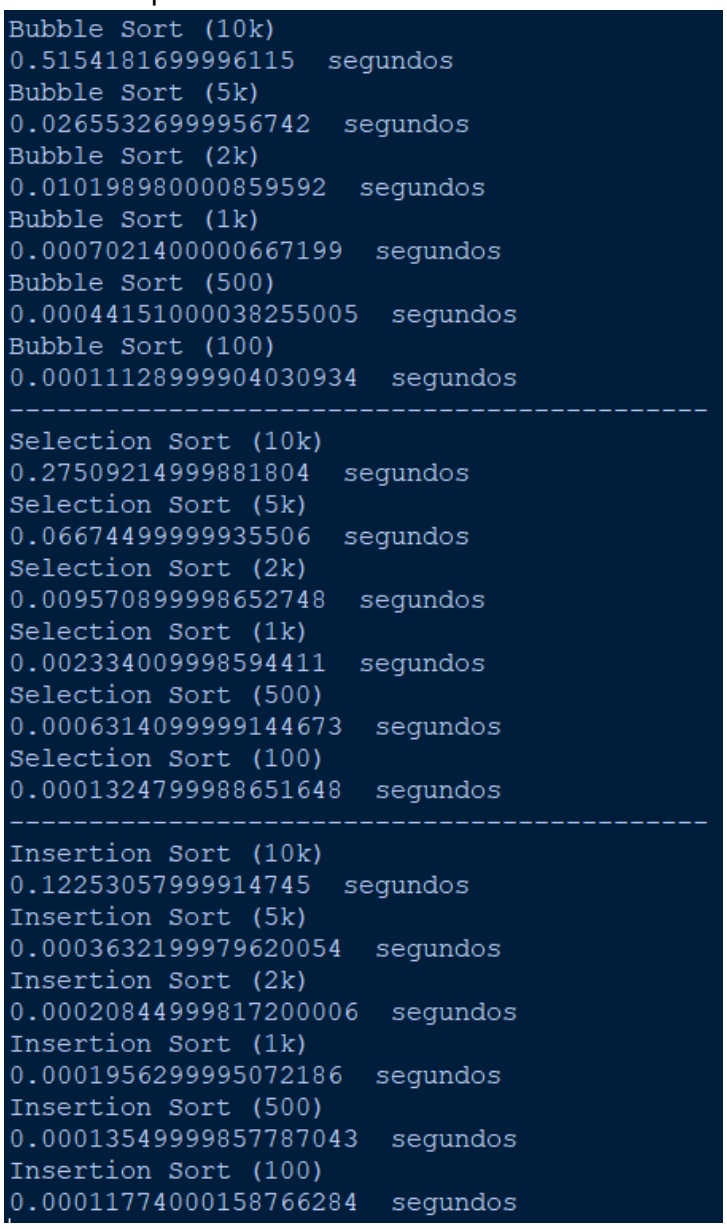
```
f = open(OUT_PATH, "r")
f_str = f.readlines()
f.close()

print("Inicialização de Pilha")
time_avg = []
mem_avg = []
for i in range(TIMES):
    start_time = getTime()
    start_mem = getMem()
    d = DanStack()
    for line in f_str:
        d.add(line)
    d.get(1), d.get(100), d.get(1000), d.get(5000), d.get(d.top-1)
    time_avg.append(getTime() - start_time)
    mem_avg.append(getMem() - start_mem)
print(sum(time_avg)/len(time_avg), " segundos")
print(sum(mem_avg)/len(mem_avg) / 1024, " KB consumidos")
print("-------------------------------")
```

```
# Adição

for n in range(1, 5):
    time_avg.clear()
    mem_avg.clear()
    np = pow(10, n)
    for i in range(TIMES):
        d = DanStack()
        start_time = getTime()
        start_mem = getMem()
        for line in f_str[:np]:
            d.add(line)
        time_avg.append(getTime() - start_time)
        mem_avg.append(getMem() - start_mem)
    print(f"Adição em Pilha ({np})")
    print(sum(time_avg)/len(time_avg), " segundos")
    print(sum(mem_avg)/len(mem_avg) / 1024, " KB consumidos")
print("-------------------------------")
```

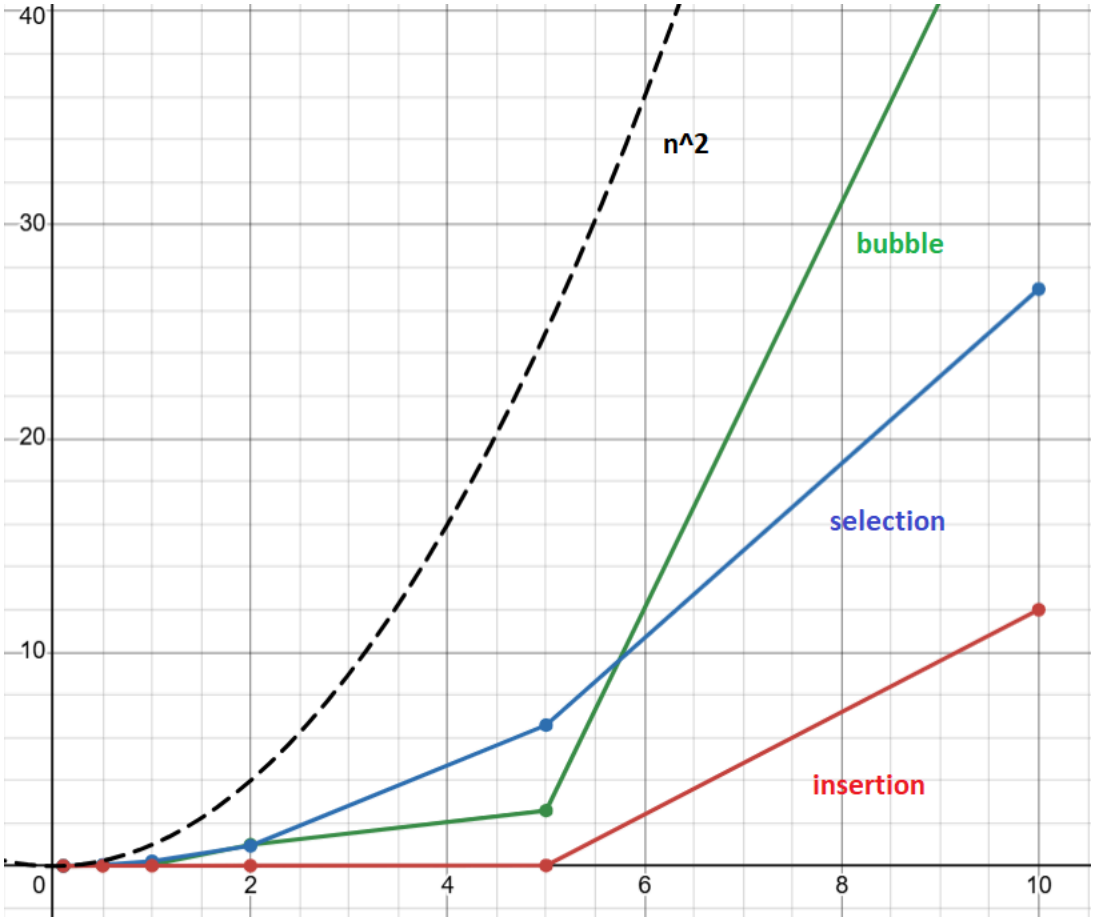
```
# Remoção

for n in range(1, 5):
    time_avg.clear()
    mem_avg.clear()
    np = pow(4, n)
    for i in range(TIMES):
        d = DanStack()
        for line in f_str:
            d.add(line)
        start_time = getTime()
        start_mem = getMem()
        for r in range(np):
            d.remove(np*2)
        time_avg.append(getTime() - start_time)
        mem_avg.append(getMem() - start_mem)
    print(f"Remoção em Pilha ({np})")
    print(sum(time_avg)/len(time_avg), " segundos")
    print(sum(mem_avg)/len(mem_avg) / 1024, " KB consumidos")
print("--------------------------------")
```

Análise dos Algoritmos

Vejamos o tempo de execução com tamanhos variantes de entrada, executados 10 vezes cada para achar uma média confiável :

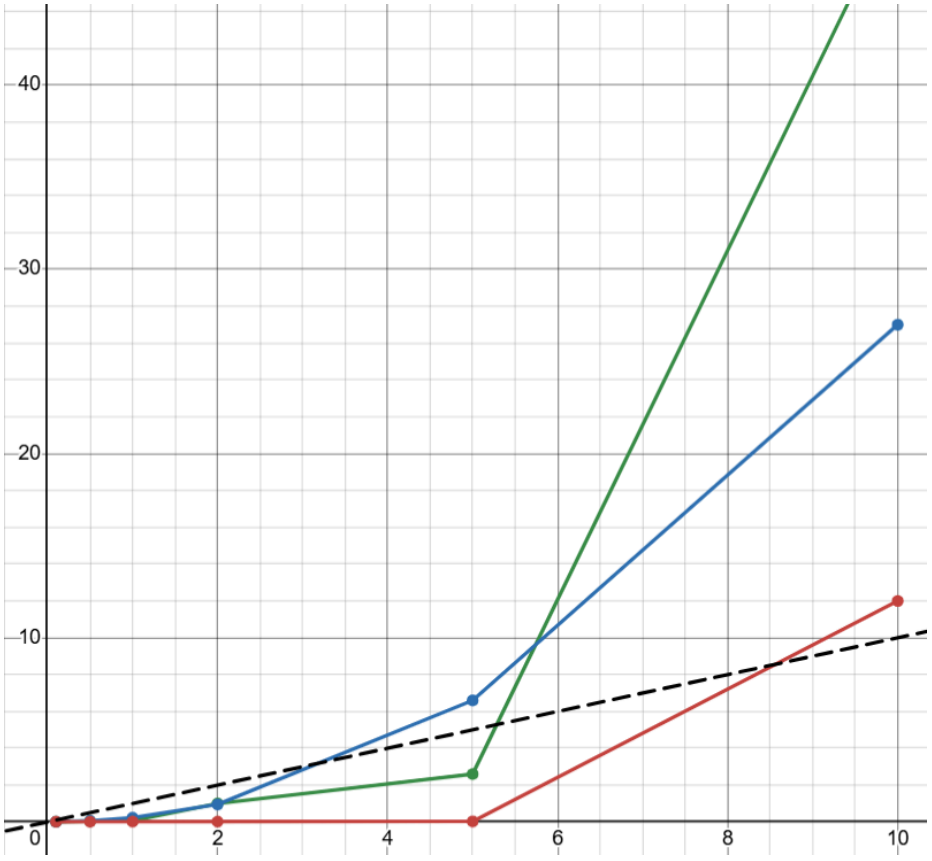


(O número entre parênteses é o número de elementos)



(As entradas foram divididas por 100 e as médias de tempo foram multiplicadas por 100)

Ao colocar em um gráfico os valores, o Bubble, Insertion e Selection Sort parecem muito com uma função quadrática, concluindo-se que possuem um limite assintótico superior de $O(n^2)$. E não só visualmente, mas logicamente já que percorrem o vetor de entrada inteiro a cada elemento do vetor.

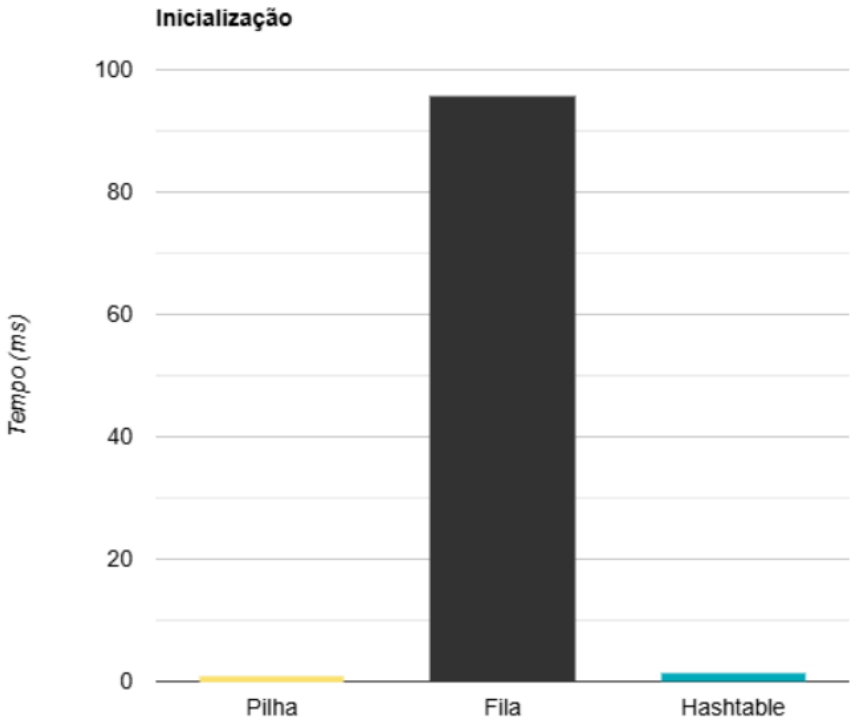


É importante observar que o Bubble Sort, e principalmente o Insertion Sort, ao introduzir uma entrada picotado e com dados praticamente já ordenados, acabam assimilando uma função linear nos valores iniciais, podendo concluir que o limite assimptótico inferior é igual a Ω(n).

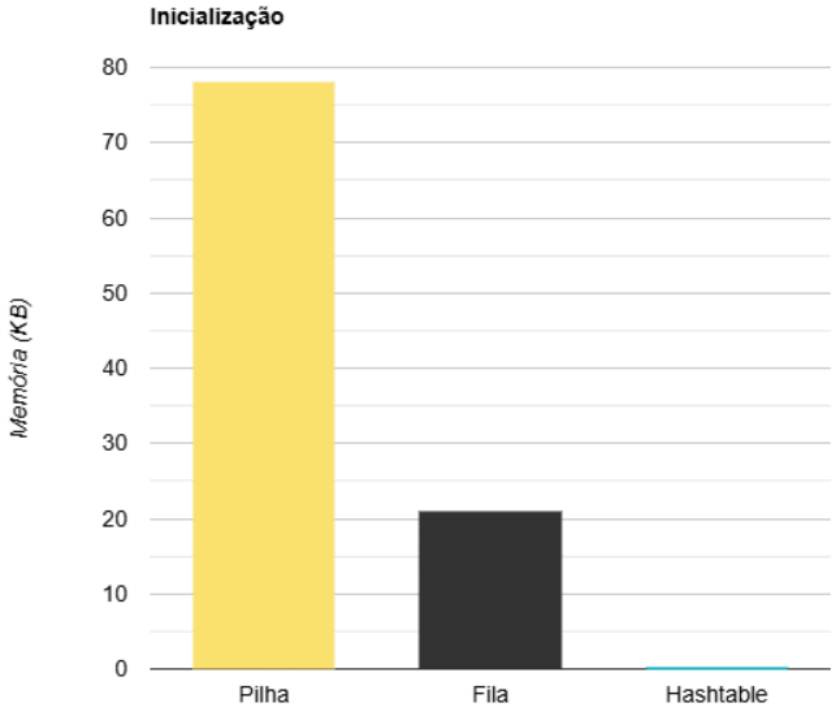
Análise das Estruturas de Dado

Inicialização

```
Inicialização de Pilha
0.0008927775002121052  segundos
78.0  KB consumidos
------------------------------
Inicialização de Fila
0.09579165000031935  segundos
21.0  KB consumidos
------------------------------
Inicialização de Hashtable
0.001259497500177531  segundos
0.0  KB consumidos
------------------------------
```



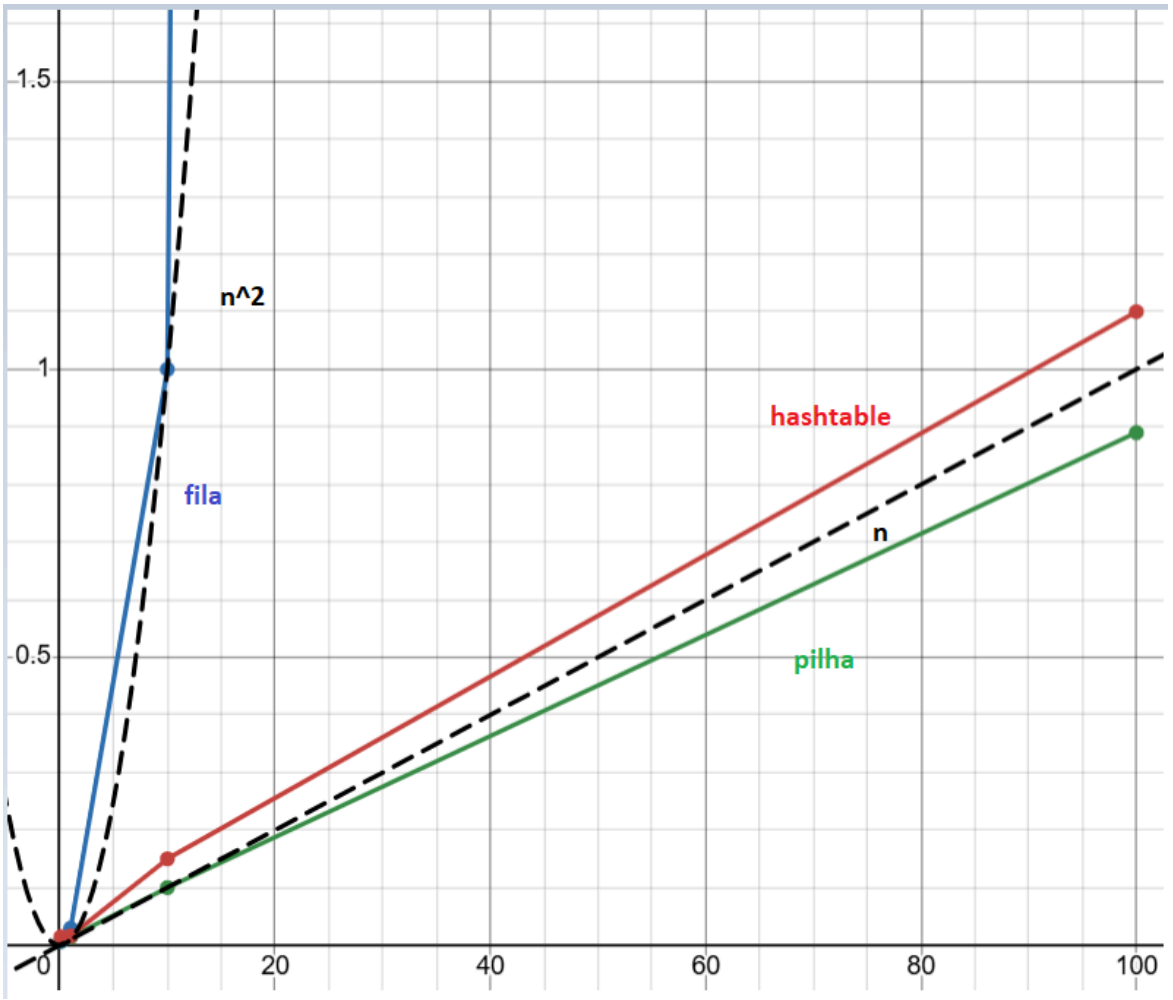
Podemos observar que a Fila em nossa implementação demora muito tempo pois executa a cópia do vetor para cada adição de um elemento, demorando $O(n^2)$ para a entrada de inicialização, enquanto o resto demora O(n) pois executa somente 1 vez a inserção para cada elemento.



Podemos ver que a Pilha ocupa um espaço quase equivalente ao número de elementos, concluindo uma complexidade de espaço O(n). A Fila parece estar próxima do logaritmo de 10000*10 (40), podendo assumir O(log(n)). Finalmente a Hashtable já tem todos seus índices pré-alocados em memória, ocupando somente O(1) de espaço.

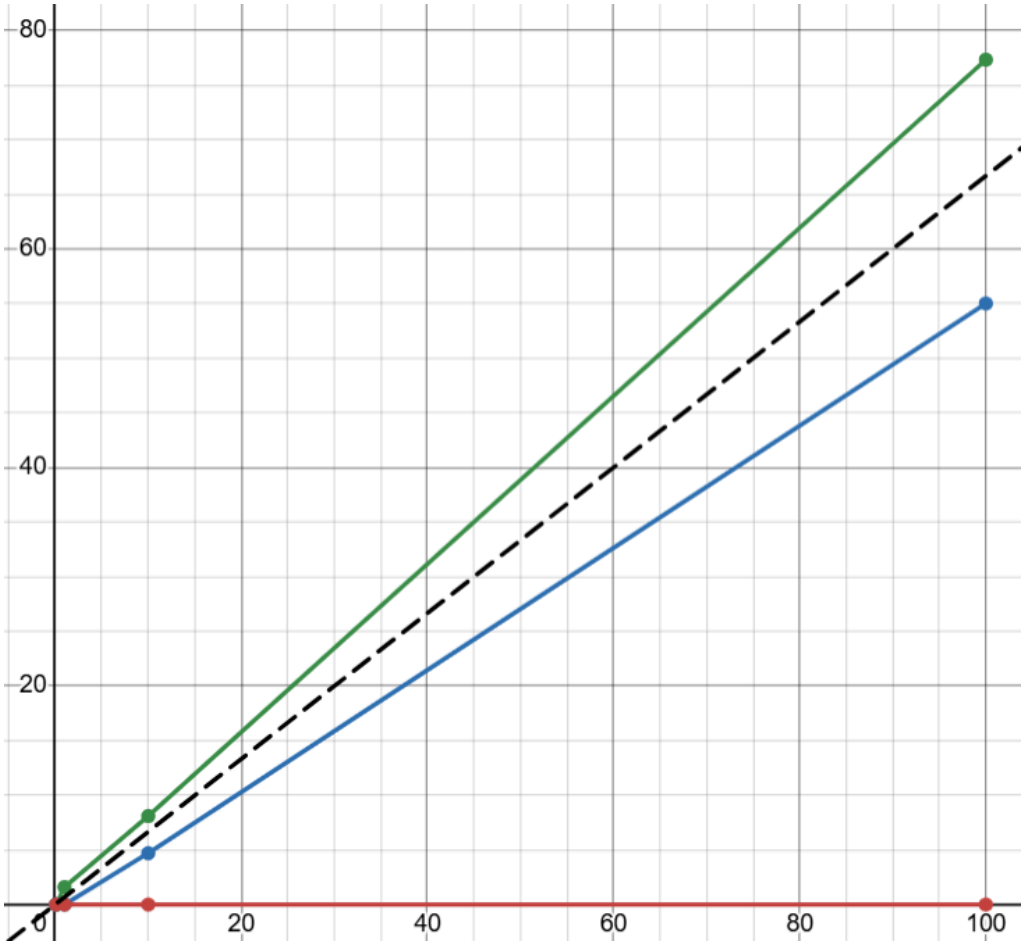
Inserção

```
Adição em Pilha (10)
6.5450003603473306e-06  segundos
0.0  KB consumidos
Adição em Pilha (100)
1.4007500431034714e-05  segundos
1.6  KB consumidos
Adição em Pilha (1000)
0.0001010125004540896  segundos
8.1  KB consumidos
Adição em Pilha (10000)
0.0008957600002759136  segundos
77.3  KB consumidos
------------------------------
Adição em Fila (10)
7.044999802019447e-06  segundos
0.0  KB consumidos
Adição em Fila (100)
2.9337499654502608e-05  segundos
0.0  KB consumidos
Adição em Fila (1000)
0.0010237525009870296  segundos
-4.7  KB consumidos
Adição em Fila (10000)
0.21174784249815276  segundos
55.0  KB consumidos
------------------------------
Adição em Hashtable (10)
1.4882498726365157e-05  segundos
0.0  KB consumidos
Adição em Hashtable (100)
1.675749990681652e-05  segundos
0.0  KB consumidos
Adição em Hashtable (1000)
0.00015737000103399623  segundos
0.0  KB consumidos
Adição em Hashtable (10000)
0.001187819998813211  segundos
0.0  KB consumidos
------------------------------
```



(Número de elementos são divididos por 100 e o tempo é multiplicado por 1000)

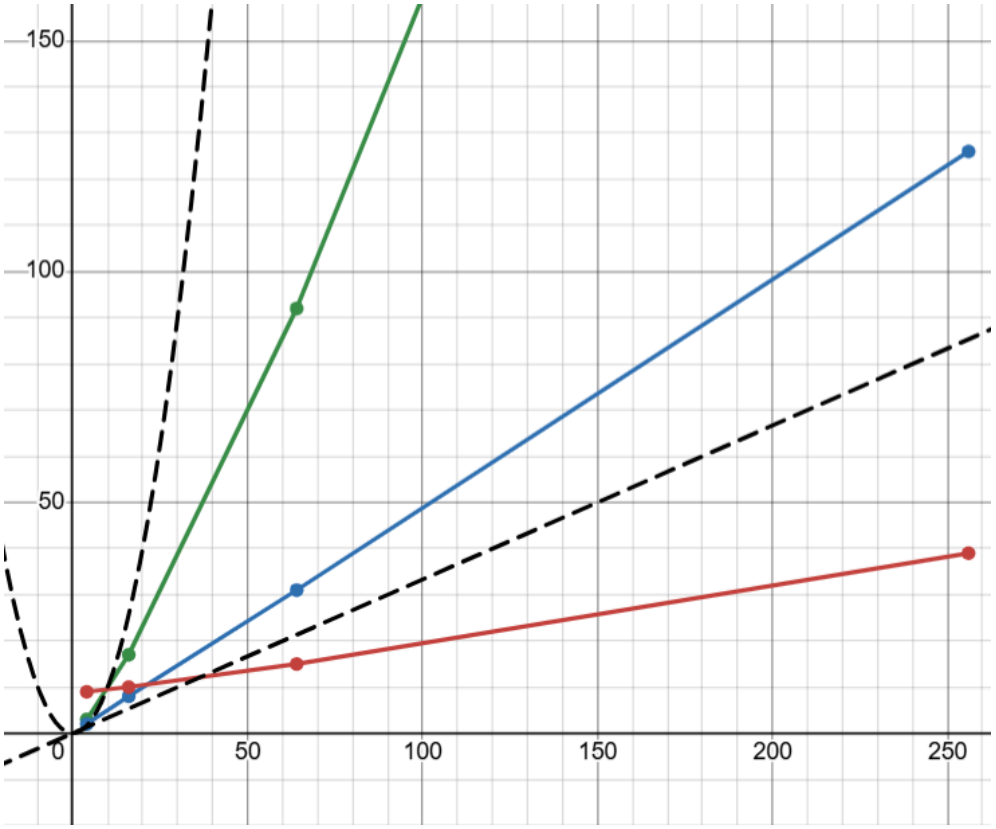
Ao observar o gráfico do complexidade de tempo, ele está coerente com a explicação apresentada sobre a complexidade de tempo na inicialização das estruturas.



A complexidade de espaço para ambas a Fila e a Pilha parecem ser O(n), enquanto a Hashtable é somente O(1).

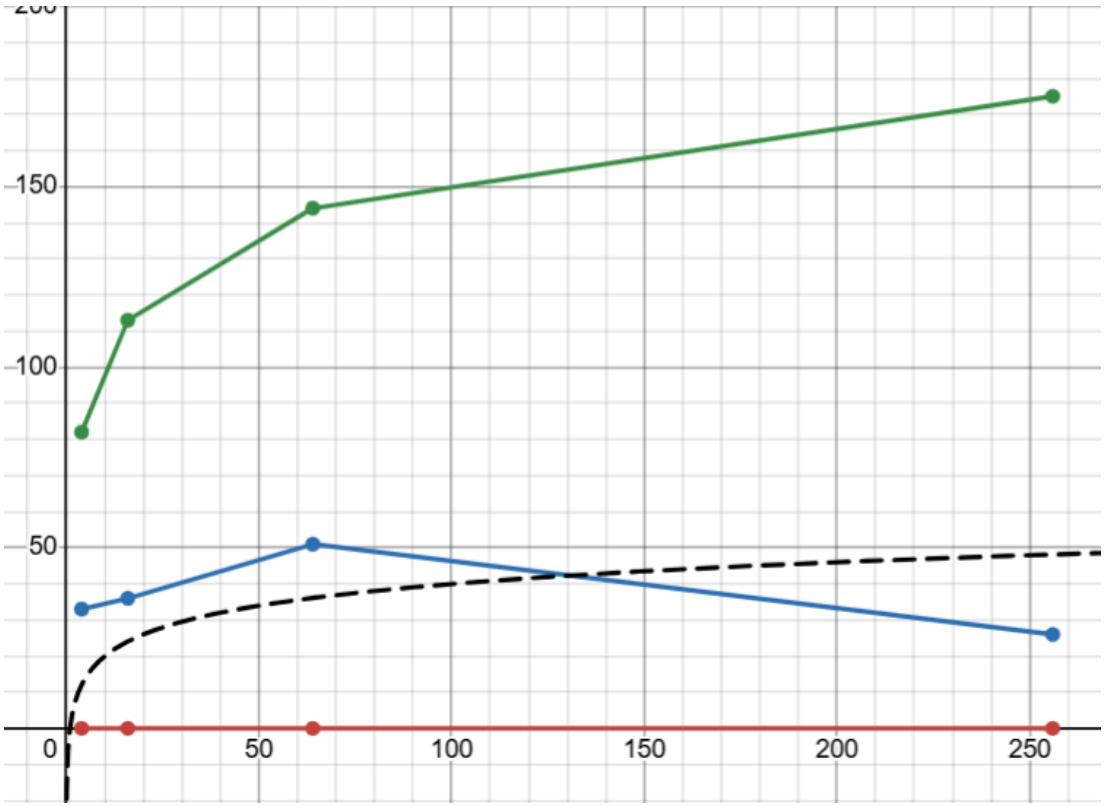
Remoção

```
Remoção em Pilha (4)
0.03038337750149367  segundos
8204.8  KB consumidos
Remoção em Pilha (16)
0.17784905250009614  segundos
11332.4  KB consumidos
Remoção em Pilha (64)
0.9282753250008682  segundos
14460.7  KB consumidos
Remoção em Pilha (256)
4.526448584999161  segundos
17584.3  KB consumidos
------------------------------
Remoção em Fila (4)
0.000237272498634411  segundos
36.6  KB consumidos
Remoção em Fila (16)
0.000829360000352608  segundos
33.8  KB consumidos
Remoção em Fila (64)
0.003195922500162851  segundos
51.3  KB consumidos
Remoção em Fila (256)
0.012687907498548156  segundos
26.0  KB consumidos
------------------------------
Remoção em Hashtable (4)
9.137500819633715e-06  segundos
0.0  KB consumidos
Remoção em Hashtable (16)
9.920001321006567e-06  segundos
0.0  KB consumidos
Remoção em Hashtable (64)
1.56999998580431 55e-05  segundos
0.0  KB consumidos
Remoção em Hashtable (256)
3.9775001278030685e-05  segundos
0.0  KB consumidos
------------------------------
```



(Os valores da pilha são multiplicados por 100, da fila por 10000 e a hashtable por 10^6)

Vemos que ambas a Hashtable e a Fila possuem O(n) complexidade de tempo, enquanto a pilha possui $O(n^2)$.



(Os valores da pilha são divididos por 100)

Dá para observar que a complexidade de espaço da pilha assemelha uma linha logarítmica, concluindo que possui O(log(n)). A fila parece estar entre n e log(n). A hashtable permanece O(1).