



► Instituto Infnet

ESTI - ESCOLA SUPERIOR DA TECNOLOGIA DA INFORMAÇÃO
Curso de Graduação em Engenharia de Software

Nome: Kaio Henrique Silva da Cunha

Projeto de Bloco: Ciência da Computação
TP1

Fortaleza - Ceará

KAIO HENRIQUE SILVA DA CUNHA

TP1

TP1 da disciplina de Projeto de Bloco: Ciência da Computação do curso de Graduação em Engenharia de Software.

Parte 1: Manipulação de Arquivos em Linux

1. Utilize um emulador de terminal Linux para acessar um diretório específico fornecido pelo professor (esse diretório terá pelo menos 10000 arquivos, presentes na raiz ou em subdiretórios aninhados).
2. Gere uma listagem completa dos arquivos.

```

cuka088 00:36:16 02/04/25 /shared/projects/kaiohenricunha/pb-ciencia-computacao ✓
cuka088 00:36:16 02/04/25 /shared/projects/kaiohenricunha/pb-ciencia-computacao ✓
cuka088 00:36:16 02/04/25 /shared/projects/kaiohenricunha/pb-ciencia-computacao ✓ ls -laR ../../derailed/ | wc -l
1666
cuka088 00:36:37 02/04/25 /shared/projects/kaiohenricunha/pb-ciencia-computacao ✓ ls -laR ../../derailed/
../../derailed/:
total 0
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:48 .
drwxr-xr-x 1 cuka088 cuka088 4096 Feb  4 00:14 ..
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:56 k9s

../../derailed/k9s:
total 31376
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:56 .
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:48 ..
-rw-r--r-- 1 cuka088 cuka088 296 Jan 31 19:49 codebeatsettings
-rw-r--r-- 1 cuka088 cuka088 121 Jan 31 19:49 dockerignore
drwxr-xr-x 1 cuka088 cuka088 4096 Feb  3 19:57 git
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 github
-rw-r--r-- 1 cuka088 cuka088 176 Jan 31 19:49 gitignore
-rw-r--r-- 1 cuka088 cuka088 4115 Jan 31 19:49 golanci.yml
-rw-r--r-- 1 cuka088 cuka088 2408 Jan 31 19:49 goreleaser.yml
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 kubernetes
-rw-r--r-- 1 cuka088 cuka088 257 Jan 31 19:49 travis.yml
-rw-r--r-- 1 cuka088 cuka088 9 Jan 31 19:49 CNAME
-rw-r--r-- 1 cuka088 cuka088 608 Jan 31 19:49 COPYING
-rw-r--r-- 1 cuka088 cuka088 935 Jan 31 19:49 Dockerfile
-rw-r--r-- 1 cuka088 cuka088 10170 Jan 31 19:49 LICENSE
-rw-r--r-- 1 cuka088 cuka088 1139 Jan 31 19:49 Makefile
-rw-r--r-- 1 cuka088 cuka088 42416 Jan 31 19:49 README.md
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 assets
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 change_logs
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 cmd
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:52 execs
-rw-r--r-- 1 cuka088 cuka088 17202 Jan 31 19:49 go.mod
-rw-r--r-- 1 cuka088 cuka088 186268 Jan 31 19:49 go.sum
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 internal
drwxr-xr-x 1 cuka088 cuka088 3183212 Nov 16 20:37 k9s_linux_amd64.deb
-rw-r--r-- 1 cuka088 cuka088 557 Jan 31 19:49 main.go
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 plugins
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 skids
-rw-r--r-- 1 cuka088 cuka088 4096 Jan 31 19:49 snap

../../derailed/k9s/.git:
total 124
drwxr-xr-x 1 cuka088 cuka088 4096 Feb  3 19:57 .
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:56 ..
-rw-r--r-- 1 cuka088 cuka088 23 Jan 31 19:49 HEAD
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:48 branches
-rw-r--r-- 1 cuka088 cuka088 279 Jan 31 19:49 config
-rw-r--r-- 1 cuka088 cuka088 73 Jan 31 19:48 description
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:48 hooks
drwxr-xr-x 1 cuka088 cuka088 94388 Jan 31 19:49 index
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:48 info
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 logs
drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:48 objects
-rw-r--r-- 1 cuka088 cuka088 25014 Jan 31 19:49 packed-refs

```

3. Redirecione a saída dessa listagem para um arquivo de texto usando comandos apropriados no Linux.

```

cuka408 00:38:18 02/04/25 /shared/projects/kaiohenricunha/pb-ciencia-computacao X1
cuka408 00:38:23 02/04/25 /shared/projects/kaiohenricunha/pb-ciencia-computacao ✓ ls -laR ../../derailed/ > listing.txt
head listing.txt
../../derailed/:
total 0
drwxr-xr-x 1 cuka408 cuka408 4096 Jan 31 19:48 .
drwxr-xr-x 1 cuka408 cuka408 4096 Feb  4 00:14 ..
drwxr-xr-x 1 cuka408 cuka408 4096 Jan 31 19:56 k9s

../../derailed/k9s:
total 31376
drwxr-xr-x 1 cuka408 cuka408 4096 Jan 31 19:56 .
drwxr-xr-x 1 cuka408 cuka408 4096 Jan 31 19:48 ..
cuka408 00:38:32 02/04/25 /shared/projects/kaiohenricunha/pb-ciencia-computacao ✓ |

```

Parte 2: Desenvolvimento dos Programas em Python

Crie dois programas em Python (ambos desenvolvidos em um editor de texto não-gráfico do Linux, por linha de comando):

O primeiro deve :

1. Ler a listagem de arquivos do arquivo de texto gerado.
2. Implementar os algoritmos Bubble Sort, Selection Sort, Insertion Sort para ordenar a listagem.
3. Medir e registrar o tempo de execução do algoritmo para cada execução de ordenação.

```
Ubuntu
cuka488 00:42:31 02/04/25 1/3 shared/projects/kaiohenricunha/pb-ciencia-computacao X1 nano sorting.py
cuka488 00:44:39 02/04/25 2/3 shared/projects/kaiohenricunha/pb-ciencia-computacao cat sorting.py
~/usr/bin/env python3
import time

def bubble_sort(data):
    """Implementation of Bubble Sort"""
    arr = data[:] # Make a copy so original isn't modified
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
        return arr

def selection_sort(data):
    """Implementation of Selection Sort"""
    arr = data[:]
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

def insertion_sort(data):
    """Implementation of Insertion Sort"""
    arr = data[:]
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

def measure_sort_time(sort_func, data):
    """Measure execution time of a given sorting function"""
    start_time = time.perf_counter()
    sorted_data = sort_func(data)
    end_time = time.perf_counter()
    return sorted_data, (end_time - start_time)

def main():
    # 1. Read the file listing from the generated text file
    # Change 'listing.txt' if your file has a different name.
    input_file = "listing.txt"

    with open(input_file, "r", encoding="utf-8", errors="ignore") as f:
        lines = [line.strip() for line in f.readlines() if line.strip()]

    # 2. Run Bubble Sort, Selection Sort, Insertion Sort and measure time
    for sort_name, func in [
        ('Bubble Sort', bubble_sort),
        ('Selection Sort', selection_sort),
        ('Insertion Sort', insertion_sort)
    ]:
        _, duration = measure_sort_time(func, lines)
        print(f'{sort_name} took {duration:.6f} seconds to sort {len(lines)} items.')

if __name__ == '__main__':
    main()
```

```
cuka488 00:49:41 02/04/25 cd shared/projects/kaio
kaio-cunha/ kaiohenricunha/
cuka488 00:49:41 02/04/25 cd shared/projects/kaiohenricunha/pb-ciencia-computacao/
/home/cuka488/shared/projects/kaiohenricunha/pb-ciencia-computacao
cuka488 00:50:07 02/04/25 1/3 shared/projects/kaiohenricunha/pb-ciencia-computacao python3
sorting.py
Bubble Sort took 0.109428 seconds to sort 1557 items.
Selection Sort took 0.054516 seconds to sort 1557 items.
Insertion Sort took 0.040924 seconds to sort 1557 items.
cuka488 00:50:16 02/04/25 2/3 shared/projects/kaiohenricunha/pb-ciencia-computacao
```

O segundo deve:

1. Ler a listagem de arquivos do arquivo de texto gerado.
2. Armazenar o conteúdo em três estruturas de dados distintas: Hashtable, pilha e fila.
3. Recuperar o nome dos arquivos presentes nas posições 1, 100, 1000, 5000 e últimas posições da listagem para cada estruturar.
4. Medir e registrar o tempo de execução e memória usada nas tarefas pedidas para cada estrutura testada.
5. O programa também deve executar a adição e remoção de itens nas estruturas de dados, conforme detalhado pelo professor.
6. Também realize a mensuração do tempo de execução e memória dessas tarefas.

```

# cuka088 00:51:13 02/04/25  /shared/projects/kaiohenricunha/ph-ciencia-computacao  cat datastructures.py
#!/usr/bin/env python3
import time
import psutil  # For memory usage (you might need: pip install psutil)
from collections import deque

def measure_memory():
    """Returns the memory usage in MB of the current process."""
    process = psutil.Process()
    mem_info = process.memory_info()
    return mem_info.rss / (1024 * 1024)  # Convert bytes to MB

def main():
    # 1. Read the file listing from text file
    input_file = "listing.txt"

    with open(input_file, "r", encoding="utf-8", errors="ignore") as f:
        lines = [line.strip() for line in f.readlines() if line.strip()]

    # Print an overview
    print(f"Total lines read from file: {len(lines)}")

    # 2. Store content in a hashtable (Python dictionary), stack (list), and queue (deque)
    # (A) HASHTABLE
    # We'll store each line as a key in a dictionary.
    # If lines can repeat, you might store them as (line: count), etc.

    start_time = time.perf_counter()
    start_mem = measure_memory()
    hashtable = {}
    for i, line in enumerate(lines):
        hashtable[line] = i  # store index, or anything else
    end_time = time.perf_counter()
    end_mem = measure_memory()
    print(f"Hashtable creation took {(end_time - start_time):.6f} seconds.")
    print(f"Memory used for hashtable: {end_mem - start_mem:.6f} MB.")

    # (B) STACK (using a Python List)
    start_time = time.perf_counter()
    start_mem = measure_memory()
    stack = []
    for line in lines:
        stack.append(line)  # push
    end_time = time.perf_counter()
    end_mem = measure_memory()
    print(f"Stack creation (push all) took {(end_time - start_time):.6f} seconds.")
    print(f"Memory used for stack: {end_mem - start_mem:.6f} MB.")

    # (C) QUEUE (using collections.deque for efficiency)
    start_time = time.perf_counter()
    start_mem = measure_memory()
    queue = deque()
    for line in lines:
        queue.append(line)  # enqueue
    end_time = time.perf_counter()
    end_mem = measure_memory()
    print(f"Queue creation (enqueue all) took {(end_time - start_time):.6f} seconds.")
    print(f"Memory used for queue: {end_mem - start_mem:.6f} MB.")

# cuka088 00:52:41 02/04/25  /shared/projects/kaiohenricunha/ph-ciencia-computacao  python3 datastructures.py
es.py
Total lines read from file: 1557
Hashtable creation took 0.000000 seconds.
Memory used for hashtable: 0.000000 MB.

Stack creation (push all) took 0.000161 seconds.
Memory used for stack: 0.000000 MB.

Queue creation (enqueue all) took 0.000153 seconds.
Memory used for queue: 0.000000 MB.

Retrieving elements from these positions: [1, 100, 1000, 5000, 1557]

Stack position 1: ../derailed/
Stack position 100: drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 origin
Stack position 1000: -rw-r--r-- 1 cuka088 cuka088 2686 Jan 31 19:49 container_test.go
Stack position 5000: out of range!
Stack position 1557: -rw-r--r-- 1 cuka088 cuka088 790 Jan 31 19:49 snapchat.yaml

Queue position 1: ../derailed/
Queue position 100: drwxr-xr-x 1 cuka088 cuka088 4096 Jan 31 19:49 origin
Queue position 1000: -rw-r--r-- 1 cuka088 cuka088 2686 Jan 31 19:49 container_test.go
Queue position 5000: out of range!
Queue position 1557: -rw-r--r-- 1 cuka088 cuka088 790 Jan 31 19:49 snapchat.yaml

Hashtable position 1: ../derailed/
Hashtable position 100: ../derailed/K9s/git/refs/remotes:
Hashtable position 1000: -rw-r--r-- 1 cuka088 cuka088 1243 Jan 31 19:49 group.go
Hashtable position 5000: out of range!
Hashtable position 1557: out of range!

Stack pop(10) took 0.000111 seconds, memory change: 0.000000 MB.
Queue pop(left(10)) took 0.000104 seconds, memory change: 0.000000 MB.
Hashtable removal of 10 items took 0.000098 seconds, memory change: 0.000000 MB.

Stack push(10) took 0.000070 seconds, memory change: 0.000000 MB.
Queue enqueue(10) took 0.000064 seconds, memory change: 0.000000 MB.
Hashtable insertion of 10 new items took 0.000263 seconds, memory change: 0.000000 MB.
# cuka088 00:53:36 02/04/25  /shared/projects/kaiohenricunha/ph-ciencia-computacao  |
```

Parte 3: Análise e Relatório

1. Para cada programa implementado, explique sua lógica e funcionamento.

Programa 1: sorting.py

Este programa lê um arquivo de texto (por exemplo, listing.txt) que contém milhares de caminhos de arquivos. Ele armazena esses caminhos em uma lista e, em seguida, aplica três algoritmos de ordenação — Bubble Sort, Selection Sort e Insertion Sort — um após o outro.

Fluxo

1. Leitura do Arquivo:

- Abre o listing.txt e lê cada linha (representando um nome ou caminho de arquivo) em uma lista chamada lines.

2. Funções de Ordenação:

- Bubble Sort: Faz trocas repetidas de elementos adjacentes caso estejam fora de ordem.
- Selection Sort: Encontra o elemento mínimo em cada iteração e o coloca no início da porção ainda não ordenada.
- Insertion Sort: Insere cada novo elemento na posição correta entre os elementos já ordenados.

3. Medição de Tempo:

- Utiliza `time.perf_counter()` (ou função similar) do Python para medir quanto tempo cada função de ordenação leva para ordenar toda a lista.

4. Saída:

- Imprime o tempo decorrido para cada um dos três algoritmos, permitindo comparar o desempenho.

Propósito

- Demonstrar abordagens clássicas de ordenação com complexidade $O(n^2)$.
- Permitir a comparação direta do desempenho prático em um conjunto de dados real.

Program 2: `datastructures.py` (Hashtable, Pilha, Fila)

Este programa lê o mesmo arquivo de texto (`listing.txt`) e armazena as linhas em três estruturas de dados distintas:

- Hashtable (dicionário em Python)
- Pilha (lista em Python, usando `append()` para push e `pop()` para pop)
- Fila (`collections.deque` em Python, usando `append()` para enfileirar e `popleft()` para desenfileirar)

Fluxo

1. Leitura do Arquivo:

- Semelhante ao Programa #1, abre `listing.txt` e lê cada caminho de arquivo em uma lista chamada `lines`.

2. Criação das Estruturas de Dados:

- Hashtable: Chave = caminho do arquivo, Valor = um inteiro ou índice.
- Pilha: Cada caminho de arquivo é empilhado (via `append`).
- Fila: Cada caminho de arquivo é enfileirado (via `queue.append()`).

3. Medição de Tempo e Memória:

- Para cada estrutura, mede quanto tempo leva para inserir todos os itens.
- Mede o uso de memória antes e depois, usando uma biblioteca como `psutil`.

4. Recuperação de Dados:

- Recupera arquivos nas posições 1, 100, 1000, 5000 e a última.
- Para a pilha e a fila, faz acesso direto por índice (embora não seja o uso típico para fila).
- Para a hashtable, converte as chaves em uma lista para simular busca baseada em posição.

5. Adição e Remoção:

- Demonstra como remover itens (pop na pilha, popleft na fila, del no dicionário) e adicionar novos itens.
- Mede novamente o tempo e as mudanças de memória.

6. Saída:

- Imprime todos os intervalos de tempo e diferenças de uso de memória, além dos valores recuperados nas posições especificadas.

Propósito

- Mostrar o uso e as diferenças de desempenho entre três estruturas de dados fundamentais.
- Permitir observar o custo de memória e tempo para inserção, remoção e acesso aleatório/sequencial.

2. Calcule e explique a complexidade de tempo de cada algoritmo usando a notação Big O.

Bubble Sort

- Melhor Caso: $O(n)$ (quando a lista já está ordenada e podemos parar mais cedo em uma versão otimizada).
- Caso Médio: $O(n^2)$.
- Pior Caso: $O(n^2)$.
- Explicação: No Bubble Sort, cada passagem compara e troca elementos adjacentes. Podem ocorrer até $(n-1)$ passagens para n elementos, resultando em aproximadamente $n(n-1)/2$ comparações $\rightarrow O(n^2)$.

Selection Sort

- Melhor Caso: $O(n^2)$.
- Caso Médio: $O(n^2)$.
- Pior Caso: $O(n^2)$.
- Explicação: Para cada posição i , varre-se todo o subarray restante para encontrar o mínimo, exigindo até $(n-i)$ comparações. Somando isso para todos os i , temos $O(n^2)$ em todos os cenários, pois a quantidade de verificações é sempre a mesma.

Insertion Sort

- Melhor Caso: $O(n)$ (se a lista já estiver ordenada).
- Caso Médio: $O(n^2)$.
- Pior Caso: $O(n^2)$.
- Explicação: Cada novo elemento pode ser deslocado por muitas posições no pior caso (lista em ordem inversa). Em média, cerca de metade dos elementos são deslocados a cada inserção, resultando em $O(n^2)$ para dados desordenados na maioria das vezes.

Hashtable (Dicionário)

- Inserção/Acesso: Em média $O(1)$, pior caso $O(n)$.
- Remoção: Em média $O(1)$, pior caso $O(n)$.
- Explicação: Colisões de hash podem degradar o desempenho para $O(n)$ se muitos itens mapearem para o mesmo "bucket". No entanto, funções de hash bem distribuídas mantêm a taxa de colisão baixa, mantendo as operações em $O(1)$ em média.

Stack (List)

- Push (append()): $O(1)$ amortizado.
- Pop (pop()): $O(1)$ amortizado (quando removendo do final da lista).
- Acesso Aleatório: $O(1)$ para indexação direta em listas do Python.
- Explicação: Listas em Python são arrays dinâmicos. Inserir no final costuma ser $O(1)$ amortizado.

Queue (collections.deque)

- Enfileirar (append()): $O(1)$.
- Desenfileirar (popleft()): $O(1)$.
- Acesso Aleatório: $O(n)$ (deques não são otimizadas para indexação aleatória).
- Explicação: Deque é uma fila de duas pontas. Inserir ou remover de qualquer ponta é $O(1)$. Porém, acessar um elemento por índice arbitrário requer percorrer a partir de uma das extremidades.

3. Explique os diferentes tempos de execução e memória encontrados em cada estrutura de dados usados.

As medições dependerão de:

- Tamanho dos Dados: Número de linhas em listing.txt (por exemplo, 10.000).
- Recursos do Sistema: Velocidade da CPU, quantidade de RAM disponível etc.
- Detalhes de Implementação: Versão do Python, overhead do psutil, possíveis realocações etc.

Observações típicas de tempo de execução:

- Hashtable: Consultas/inserções rápidas (próximas de $O(1)$ em média).
- Pilha: Push/pop rápidos no final.
- Fila: Enfileirar/desenfileirar rápido nas extremidades.

Memória:

- Um hashtable pode usar mais memória que uma lista ou deque, pois armazena espaço extra para reduzir colisões (geralmente o load factor é mantido abaixo de um determinado limite).
- Uma pilha ou fila pode ser mais compacta se armazenar apenas referências em uma estrutura contígua ou duplamente encadeada.
- Em Python, há sobrecarga para objetos, referências, hashing de dicionários etc.

4. Compare os tempos de execução observados e relate-os com suas complexidades teóricas.

Program 1 (sorting.py):

- Bubble, Selection e Insertion devem escalar aproximadamente em $O(n^2)$.
- Para $n = 10.000$ linhas, os tempos de execução podem variar:
 - Insertion Sort costuma ter desempenho melhor que Bubble Sort em casos médios, especialmente se os dados estiverem parcialmente ordenados.
 - Selection Sort faz menos trocas, mas ainda varre toda a lista não ordenada a cada iteração; seu desempenho geralmente fica entre ou próximo dos resultados de Bubble e Insertion.
- Compare seus tempos medidos (por exemplo, 4s, 6s, 8s) para ver se algum é consistentemente mais rápido.

Programa 2 (datastructures.py):

- Hashtable (Dicionário): Inserção, acesso e remoção são geralmente estáveis e rápidas ($O(1)$ em média).
- Operações em Pilha (push/pop): Também são $O(1)$ amortizado, então devem ser rápidas.
- Fila (com deque): $O(1)$ para enfileirar/desenfileirar, portanto também é rápida.
- Acesso aleatório em fila: Se você testou indexação aleatória em deque, verá tempos mais lentos ($O(n)$) para índices grandes.
- Uso de memória pode variar. Hashtables geralmente apresentam overhead maior. Deques e listas podem ser mais leves, dependendo dos dados, do modelo de memória do Python e do comportamento de redimensionamento.
- Em um cenário real de medições, você pode ver tempos como:
 - Inserção em Hashtable (10k itens): ~0,01–0,05s
 - Push em Pilha (10k itens): ~0,01–0,05s
 - Enfileirar em Fila (10k itens): ~0,01–0,05s
- Hashtables podem consumir alguns MB a mais do que a pilha/fila.
- As diferenças podem não ser enormes para 10k itens, mas escalar para dados muito maiores (100k ou 1 milhão) normalmente torna esses padrões mais evidentes.

5. Prepare um relatório detalhado apresentando suas descobertas, incluindo tabelas ou gráficos para comparar o desempenho dos algoritmos.

#	Script	Exec	Operação	Itens Process.	Tempo do Script (s)	real(s)	user(s)	sys(s)	Observações
4	sorting.py	#1	Bubble Sort	1557	0.118010	0.404	0.258	0.043	Um pouco mais lento que Selection/Insertion nesta execução
5	sorting.py	#1	Selection Sort	1557	0.059573	0.404	0.258	0.043	~2x mais rápido que Bubble; típico para entrada relativamente pequena
6	sorting.py	#1	Insertion Sort	1557	0.047226	0.404	0.258	0.043	O mais rápido dos três; possivelmente dados parcialmente ordenados
7	sorting.py	#2	Bubble Sort	1557	0.115845	N/A	N/A	N/A	Sem medição em nível de sistema (sem comando "time")
8	sorting.py	#2	Selection Sort	1557	0.053717	N/A	N/A	N/A	Muito próximo do resultado da primeira execução
9	sorting.py	#2	Insertion Sort	1557	0.046422	N/A	N/A	N/A	Consistente com a primeira execução; continua sendo o mais rápido
11	datastructures.py	#1	Criação de Hashtable	1557	0.000529	0.219	0.072	0.042	Tempo mínimo; mudança de memória de 0.000000 MB
12	datastructures.py	#1	Criação de Pilha (push em todos)	1557	0.000136	0.219	0.072	0.042	Muito rápido; empilhando 1557 itens
13	datastructures.py	#1	Criação de Fila (enqueue em todos)	1557	0.000164	0.219	0.072	0.042	Também muito rápido; ligeiramente mais que a criação da pilha
14	datastructures.py	#1	Stack pop(10)	-	0.000113	0.219	0.072	0.042	O(1); mudança de memória desprezível
15	datastructures.py	#1	Queue popLeft(10)	-	0.000212	0.219	0.072	0.042	O(1); ligeiramente mais lento que stack pop(10)
16	datastructures.py	#1	Remoção de 10 itens no Hashtable	-	0.000132	0.219	0.072	0.042	Remoção de dicionário em média O(1); mudança de memória desprezível
17	datastructures.py	#1	Stack push(10)	-	0.000135	0.219	0.072	0.042	O(1) amortizado
18	datastructures.py	#1	Queue enqueue(10)	-	0.000140	0.219	0.072	0.042	O(1); mudança de memória desprezível
19	datastructures.py	#1	Inserção de 10 novos itens no Hashtable	-	0.000067	0.219	0.072	0.042	Inserções rápidas em dicionário; mudança de memória de 0.000000 MB