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

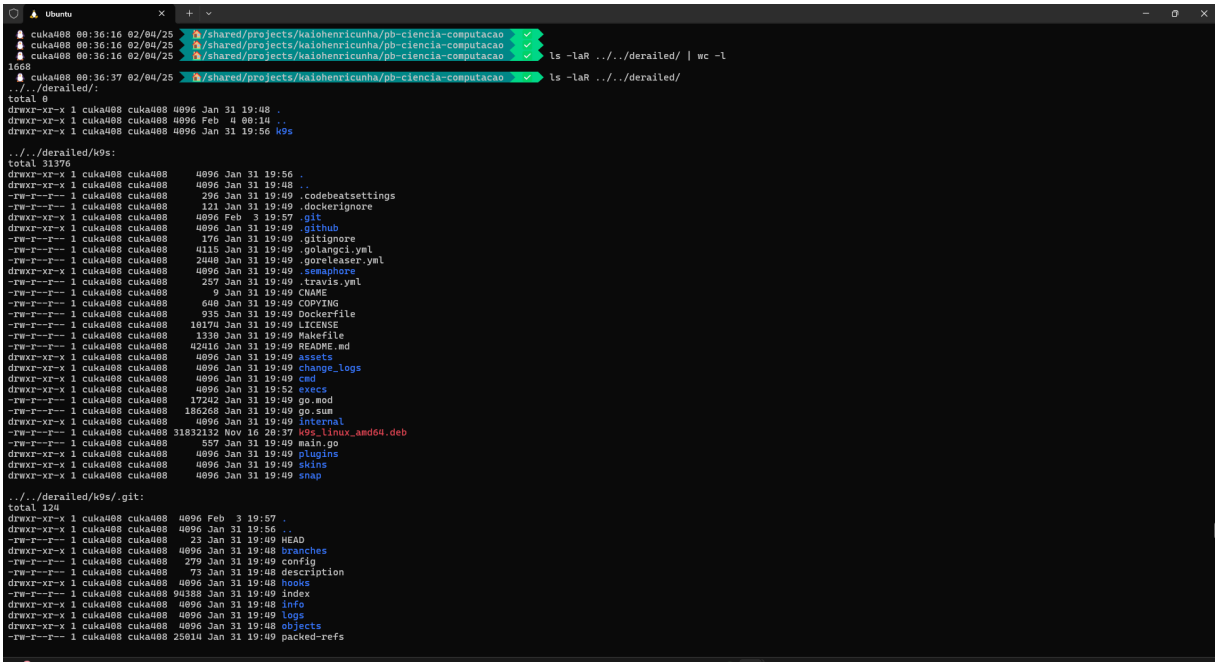
KAIO HENRIQUE SILVA DA CUNHA

TP1

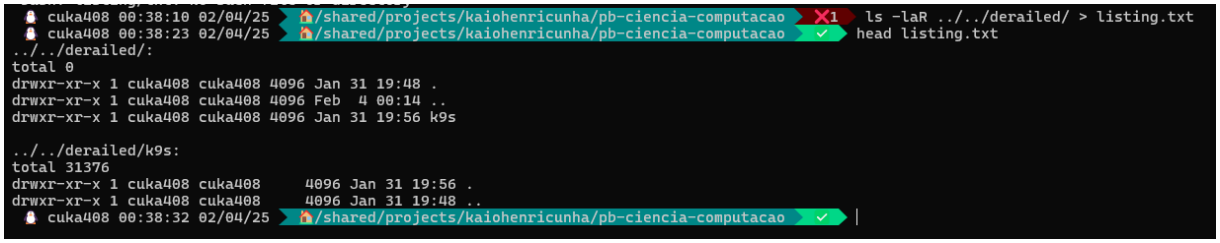
TP1 da disciplina de Projeto de Bloco: Ciência da Computação do curso de Graduação em Engenharia de Software.

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.**



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

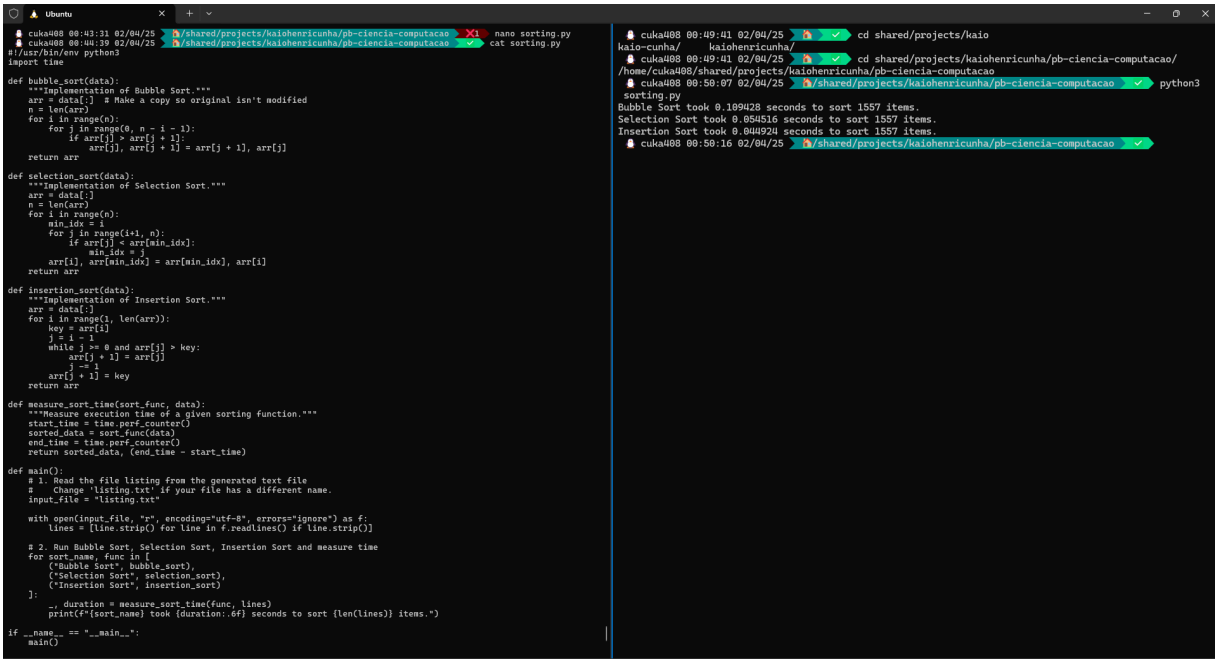


Parte 2: Desenvolvimento dos Programas em Python

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

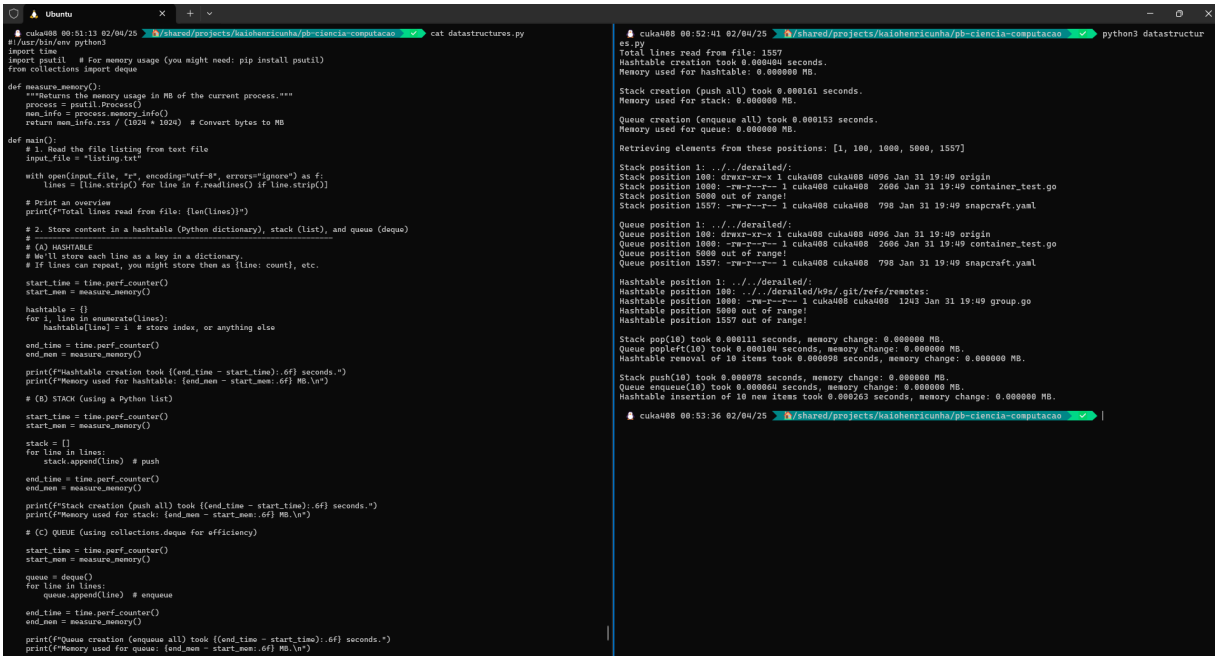
O primeiro deve :

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.**



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.**



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.

   <u>Fluxo</u>

   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(n²).
- 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.**

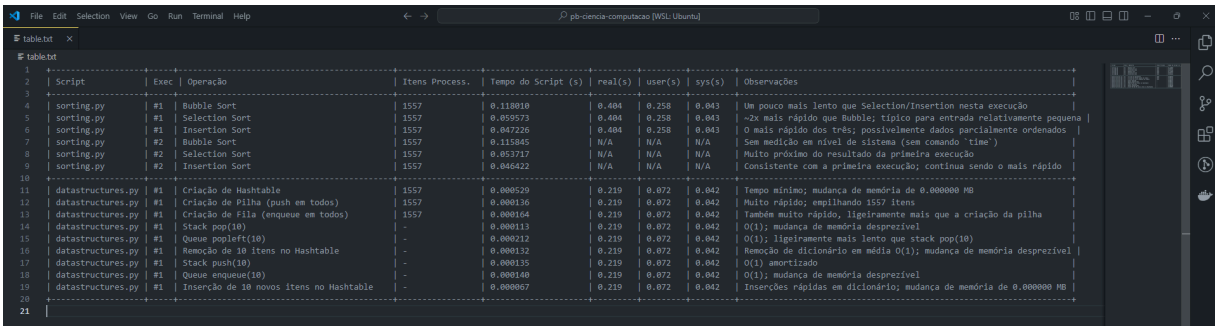
Program 1 (sorting.py):

- 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
sorting.py	#1	Bubble Sort	1557	0.118010	0.404	0.258	0.043	Um pouco mais lento que Selection/Insertion nesta execução
sorting.py	#1	Selection Sort	1557	0.059573	0.404	0.258	0.043	~2x mais rápido que Bubble; típico para entrada relativamente pequena
sorting.py	#1	Insertion Sort	1557	0.047226	0.404	0.258	0.043	O mais rápido dos três; possivelmente dados parcialmente ordenados
sorting.py	#2	Bubble Sort	1557	0.115845	N/A	N/A	N/A	Sem medição em nível de sistema (sem comando `time`)
sorting.py	#2	Selection Sort	1557	0.053717	N/A	N/A	N/A	Muito próximo do resultado da primeira execução
sorting.py	#2	Insertion Sort	1557	0.046422	N/A	N/A	N/A	Consistente com a primeira execução; continua sendo o mais rápido
datastructures.py	#1	Criação de Hashtable	1557	0.000529	0.219	0.072	0.042	Tempo mínimo; mudança de memória de 0.000000 MB
datastructures.py	#1	Criação de Pilha (push em todos)	1557	0.000136	0.219	0.072	0.042	Muito rápido; empilhando 1557 itens
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
datastructures.py	#1	Stack pop(10)	-	0.000113	0.219	0.072	0.042	O(1); mudança de memória desprezível
datastructures.py	#1	Queue popleft(10)	-	0.000212	0.219	0.072	0.042	O(1); ligeiramente mais lento que stack pop(10)
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
datastructures.py	#1	Stack push(10)	-	0.000135	0.219	0.072	0.042	O(1) amortizado
datastructures.py	#1	Queue enqueue(10)	-	0.000140	0.219	0.072	0.042	O(1); mudança de memória desprezível
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