Relatório de Desempenho – Estruturas de Dados e Ordenação

# 1. Introdução

Este relatório analisa dois programas implementados em Python: algoritmos\_datastructure.py e algoritmos\_sort.py. O primeiro mede tempo de execução e pico de memória para operações em três estruturas de dados (Hashtable, Pilha e Fila). O segundo mede o tempo de execução de três algoritmos clássicos de ordenação (Bubble, Selection e Insertion).

# 2. Lógica e funcionamento

## 2.1 Estruturas de dados

• Hashtable (dict): tabela de dispersão que associa chaves a valores. O acesso, inserção e remoção por chave operam em tempo médio O(1) graças ao hashing e endereçamento em uma tabela de buckets. No programa, usamos chaves 1..N para mapear “posições” aos nomes de arquivo.

• Pilha (list): estrutura LIFO com topo no fim da lista. As operações push (append) e pop no final são O(1) amortizado devido à estratégia de sobrealocação do array dinâmico do Python.

• Fila (deque): fila de duas pontas otimizada para inserções/remoções em ambas as extremidades em O(1). A operação de remoção no início (popleft) é O(1) verdadeiro, ideal para filas reais.

## 2.2 Programas

• algoritmos\_datastructure.py: lê listagem\_completa.txt, constrói as três estruturas, recupera itens em posições específicas (1, 100, 1000, 5000, última) e mede a execução de adição/remoção de 100 itens. Cada operação é envolvida por um medidor que coleta tempo com time.perf\_counter e pico de memória com tracemalloc.

• algoritmos\_sort.py: implementa e mede três algoritmos de ordenação in-place sobre cópias dos mesmos dados para evitar efeito de aquecimento ou ordenação prévia.

# 3. Algoritmos de ordenação – lógica e funcionamento

• Bubble Sort: percorre o array repetidas vezes, trocando pares adjacentes fora de ordem. Após cada passada, o maior elemento “borbulha” para o fim. Uma flag interrompe caso não haja trocas (lista já ordenada).

• Selection Sort: para cada posição i, seleciona o menor elemento do subarray i..n-1 e o troca com a posição i. O número de comparações é determinístico (n(n-1)/2), independentemente da ordem inicial.

• Insertion Sort: varre da esquerda para a direita e insere o elemento atual na posição correta à esquerda, deslocando elementos maiores uma posição para a direita. Em listas já ou quase ordenadas, os deslocamentos são mínimos.

# 4. Complexidade (Big‑O) e “por quê”

• Estruturas: Construção a partir de N itens é O(N) em todas; acesso por posição é O(1) (dict por chave, list por índice, deque com indexação suportada). Inserção/remoção: dict O(1) médio; list O(1) amortizado no final; deque O(1) nas extremidades.

• Bubble Sort: duas laçadas aninhadas em geral (comparações adjacentes) ⇒ O(N²); melhor caso O(N) com a flag, pois nenhuma troca ocorre e a primeira passada encerra.

• Selection Sort: para cada i, varre o restante para achar o mínimo ⇒ N+(N-1)+...+1 ≈ N²/2 comparações; portanto sempre O(N²), com poucas trocas (no máximo N).

• Insertion Sort: no pior/médio caso cada inserção desloca ~N/2 itens ⇒ custo total proporcional a N²; no melhor caso (lista já ordenada) o inner loop quase não executa ⇒ O(N).

# 5. Resultados observados

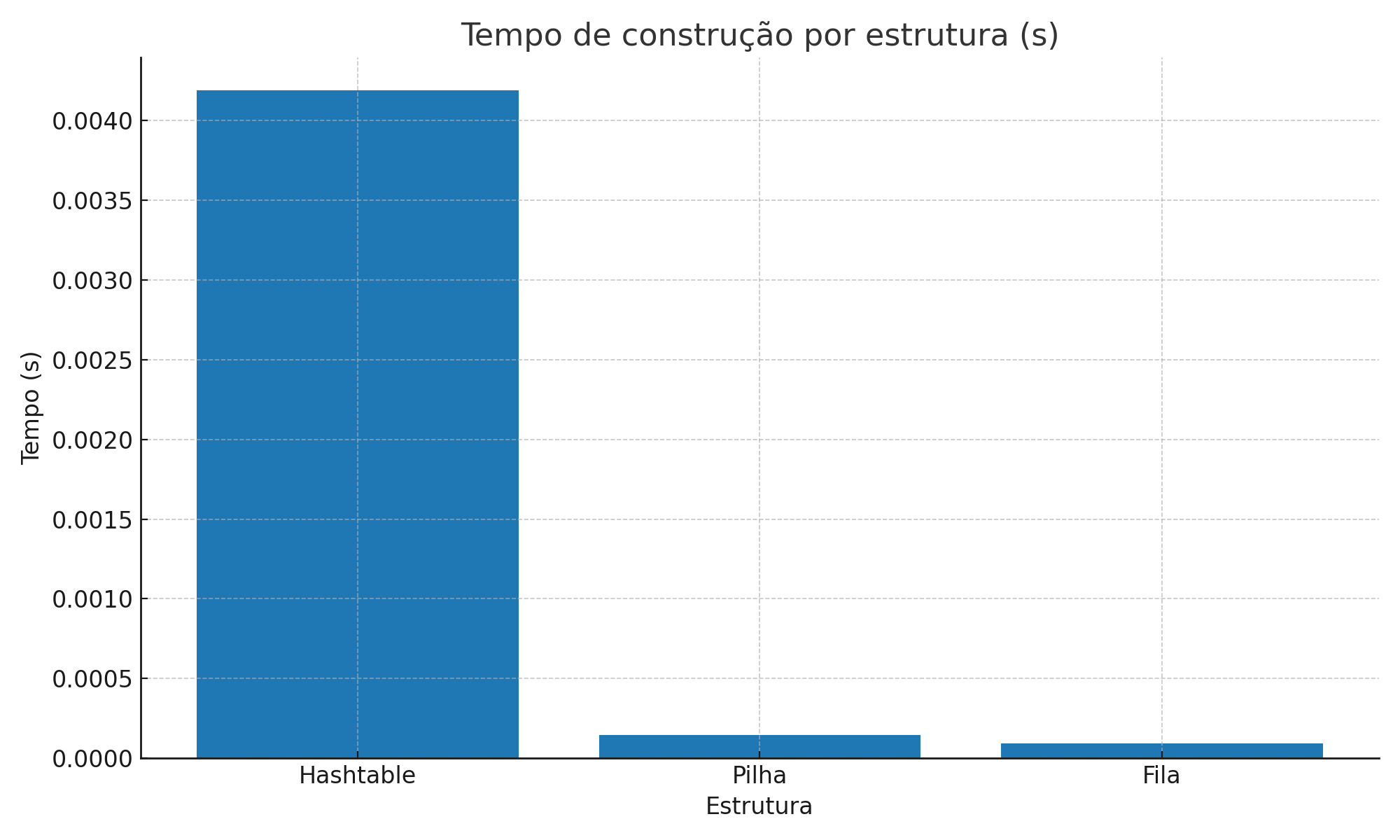
## 5.1 Estruturas de dados

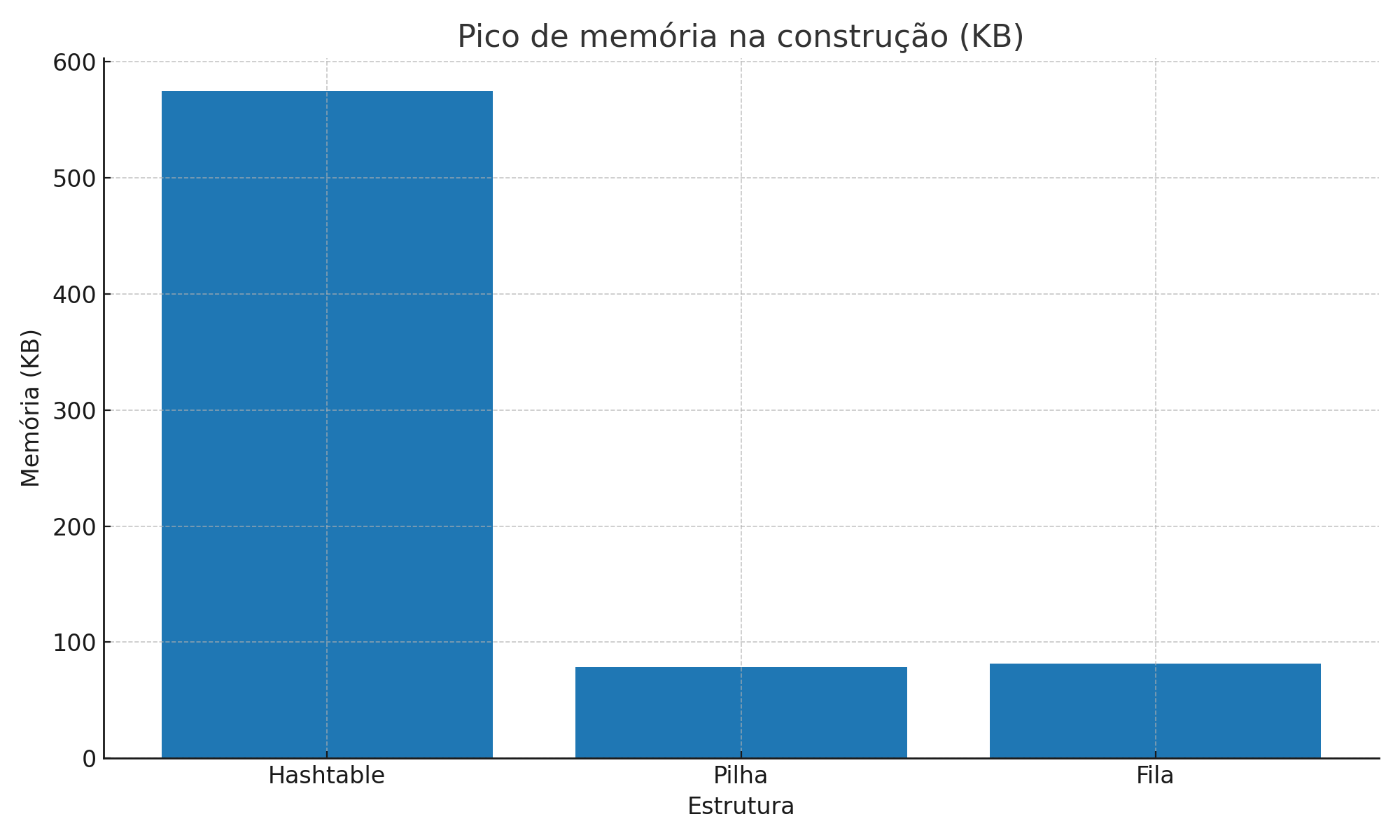
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Métrica | Etapa | Estrutura | Posição | Tempo (s) | Pico Memória (KB) |
| Construção Hashtable | Construção | Hashtable | None | 0.004189 | 574.7 |
| Construção Pilha | Construção | Pilha | None | 0.000147 | 78.6 |
| Construção Fila | Construção | Fila | None | 9.2e-05 | 81.5 |
| Hashtable - get pos 1 | Recuperação | Hashtable | 1 | 2e-06 | 0.4 |
| Hashtable - get pos 100 | Recuperação | Hashtable | 100 | 0.0 | 0.0 |
| Hashtable - get pos 1000 | Recuperação | Hashtable | 1000 | 1e-06 | 0.0 |
| Hashtable - get pos 5000 | Recuperação | Hashtable | 5000 | 0.0 | 0.0 |
| Hashtable - get pos ultima | Recuperação | Hashtable | ultima | 2e-06 | 0.0 |
| Pilha - get pos 1 | Recuperação | Pilha | 1 | 3e-06 | 0.4 |
| Pilha - get pos 100 | Recuperação | Pilha | 100 | 1e-06 | 0.0 |
| Pilha - get pos 1000 | Recuperação | Pilha | 1000 | 1e-06 | 0.0 |
| Pilha - get pos 5000 | Recuperação | Pilha | 5000 | 1e-06 | 0.0 |
| Pilha - get pos ultima | Recuperação | Pilha | ultima | 0.0 | 0.0 |
| Fila - get pos 1 | Recuperação | Fila | 1 | 2.3e-05 | 0.4 |
| Fila - get pos 100 | Recuperação | Fila | 100 | 2e-06 | 0.0 |
| Fila - get pos 1000 | Recuperação | Fila | 1000 | 1e-06 | 0.0 |
| Fila - get pos 5000 | Recuperação | Fila | 5000 | 1e-06 | 0.0 |
| Fila - get pos ultima | Recuperação | Fila | ultima | 1e-06 | 0.0 |
| Hashtable - add | Adição/Remoção | Hashtable | None | 8.7e-05 | 8.7 |
| Hashtable - remove | Adição/Remoção | Hashtable | None | 2.8e-05 | 0.6 |
| Pilha - push | Adição/Remoção | Pilha | None | 6.6e-05 | 93.9 |
| Pilha - pop | Adição/Remoção | Pilha | None | 7e-06 | 0.5 |
| Fila - enqueue | Adição/Remoção | Fila | None | 7.6e-05 | 6.5 |
| Fila - dequeue | Adição/Remoção | Fila | None | 1.1e-05 | 0.5 |

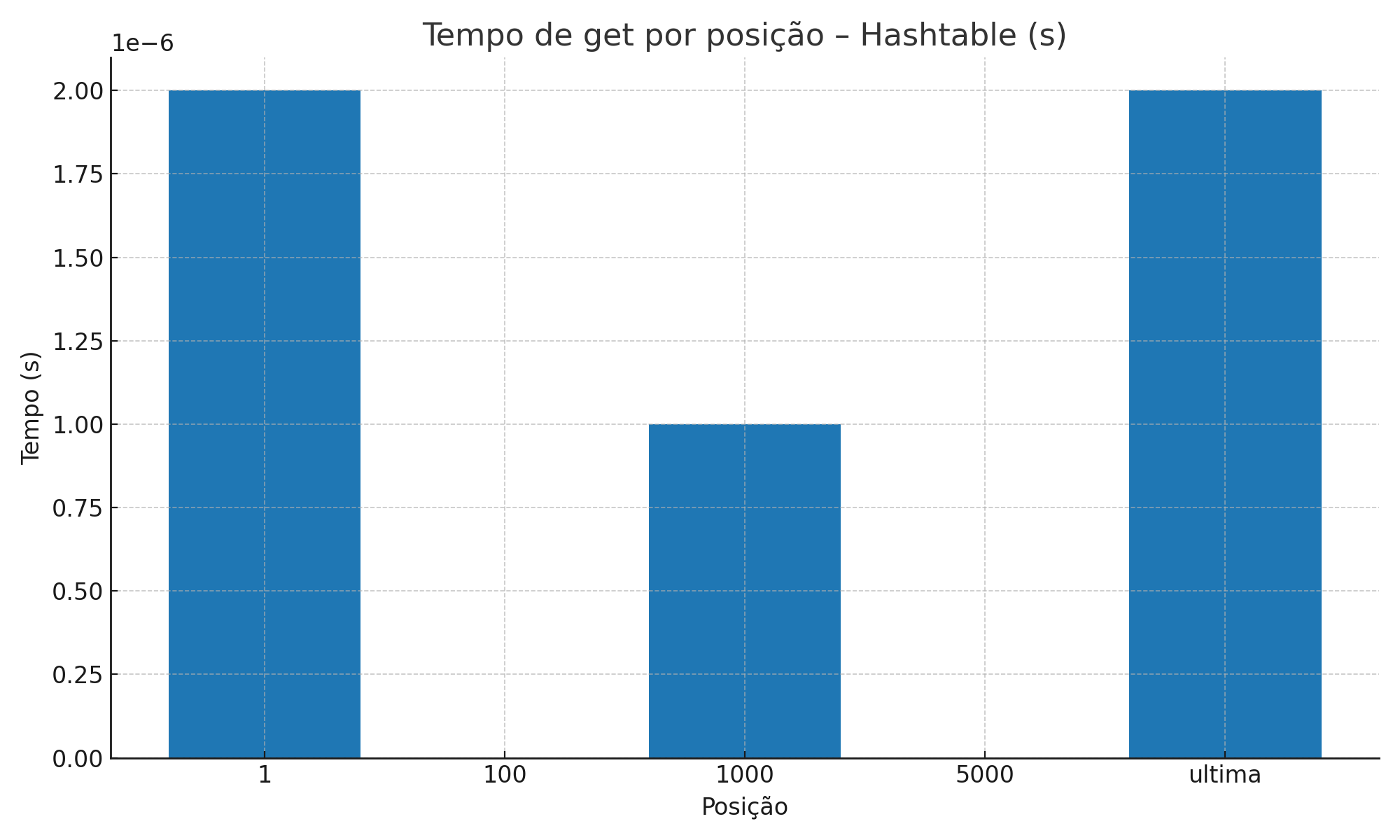
## 5.2 Algoritmos de ordenação

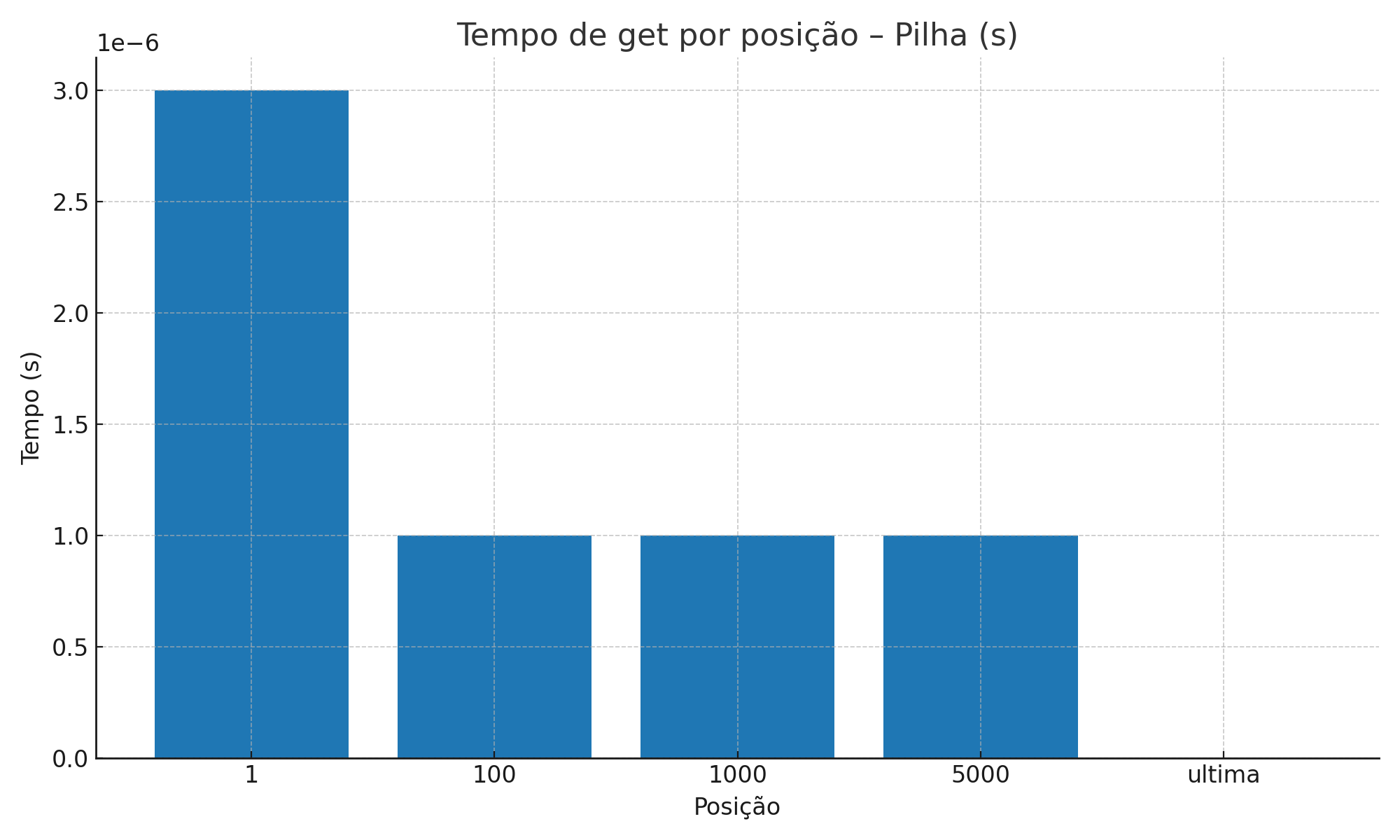
|  |  |
| --- | --- |
| Algoritmo | Tempo (s) |
| Bubble Sort | 0.00072 |
| Selection Sort | 2.620827 |
| Insertion Sort | 0.001232 |

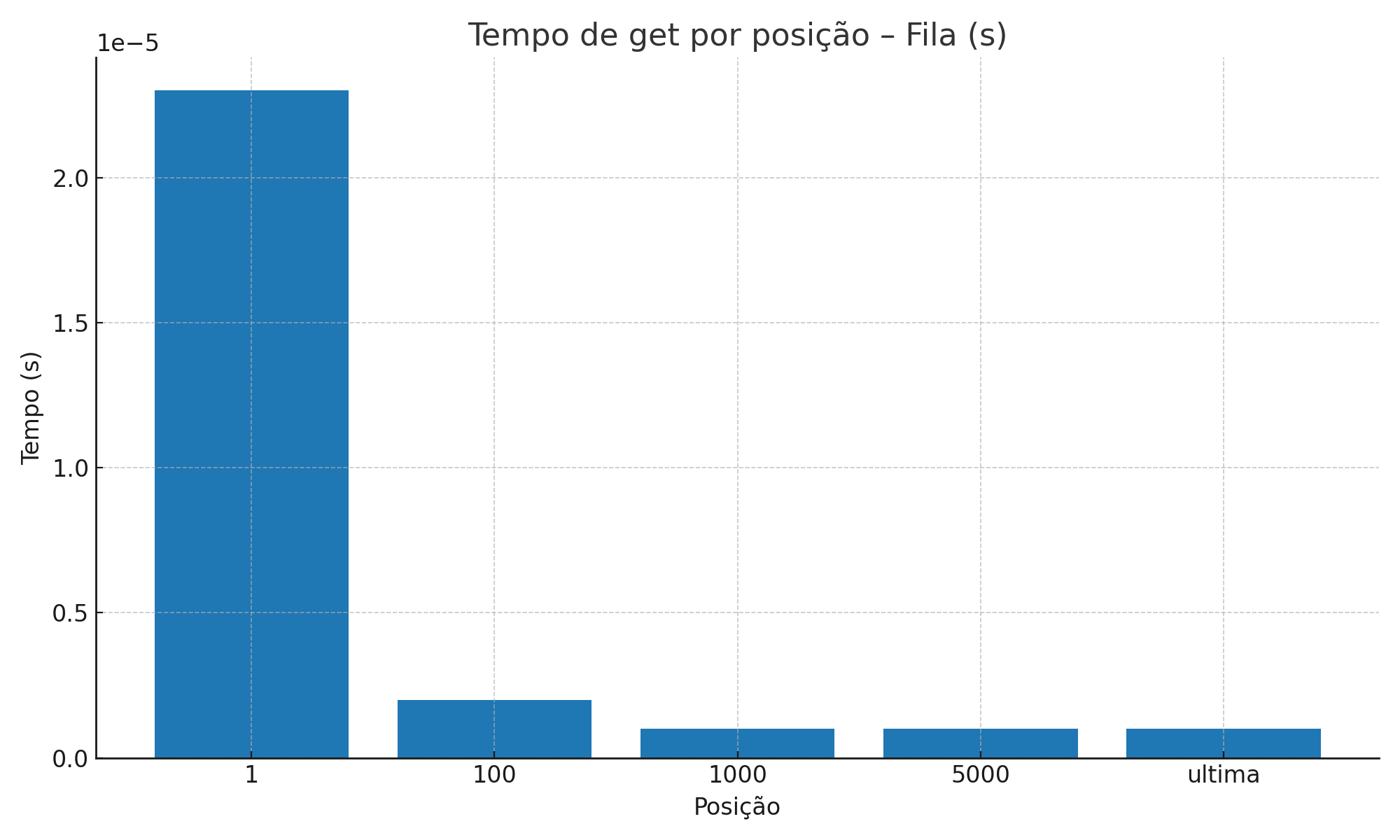
# 6. Gráficos comparativos

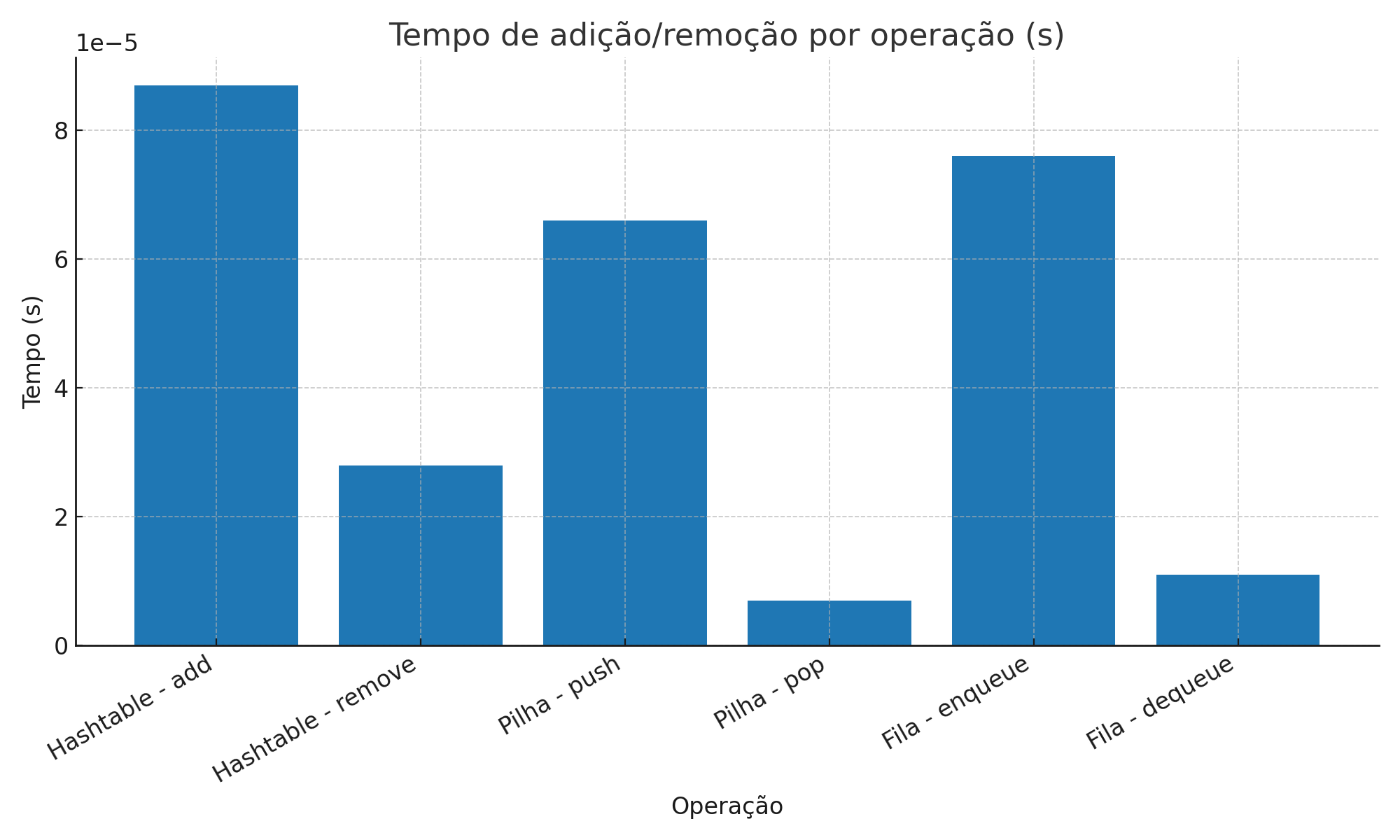


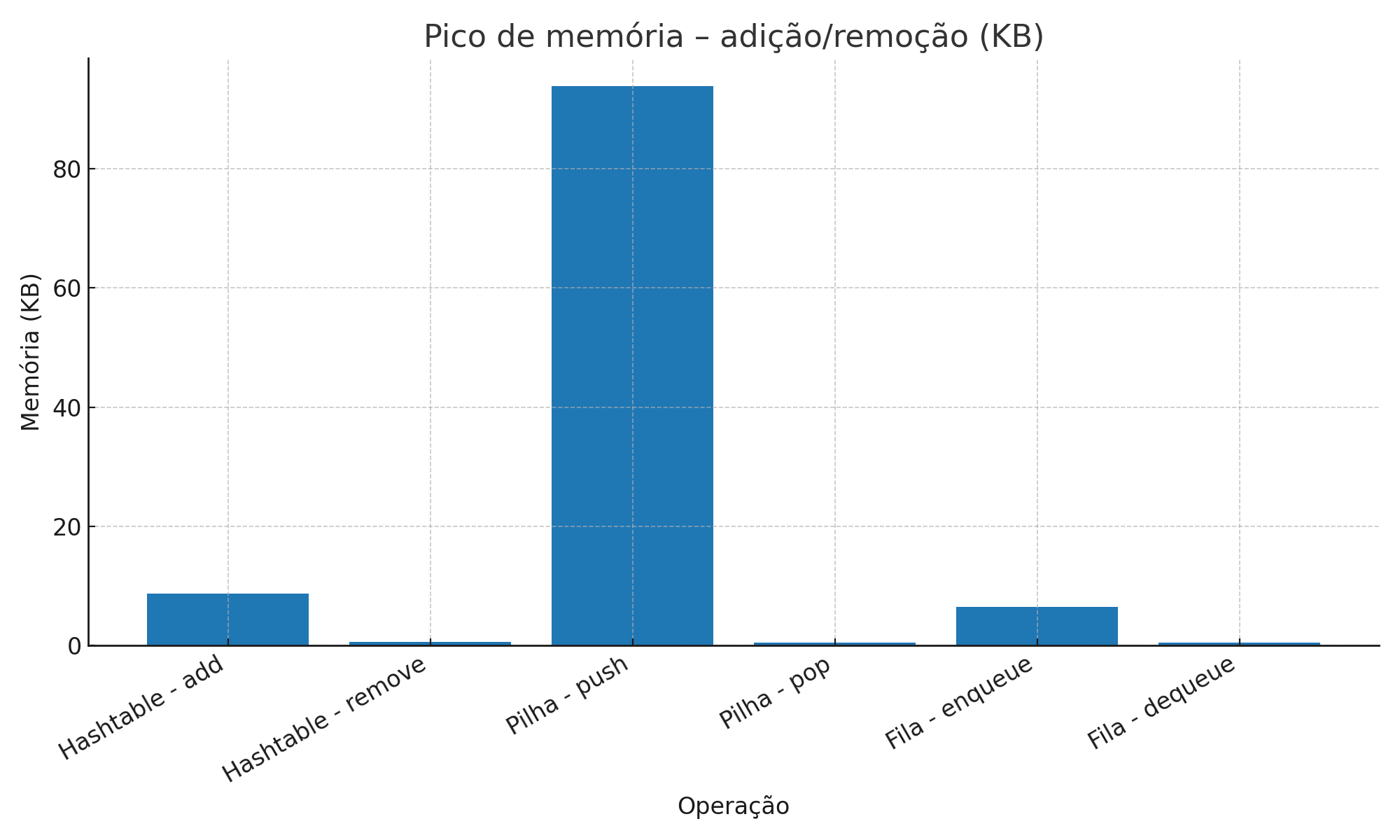


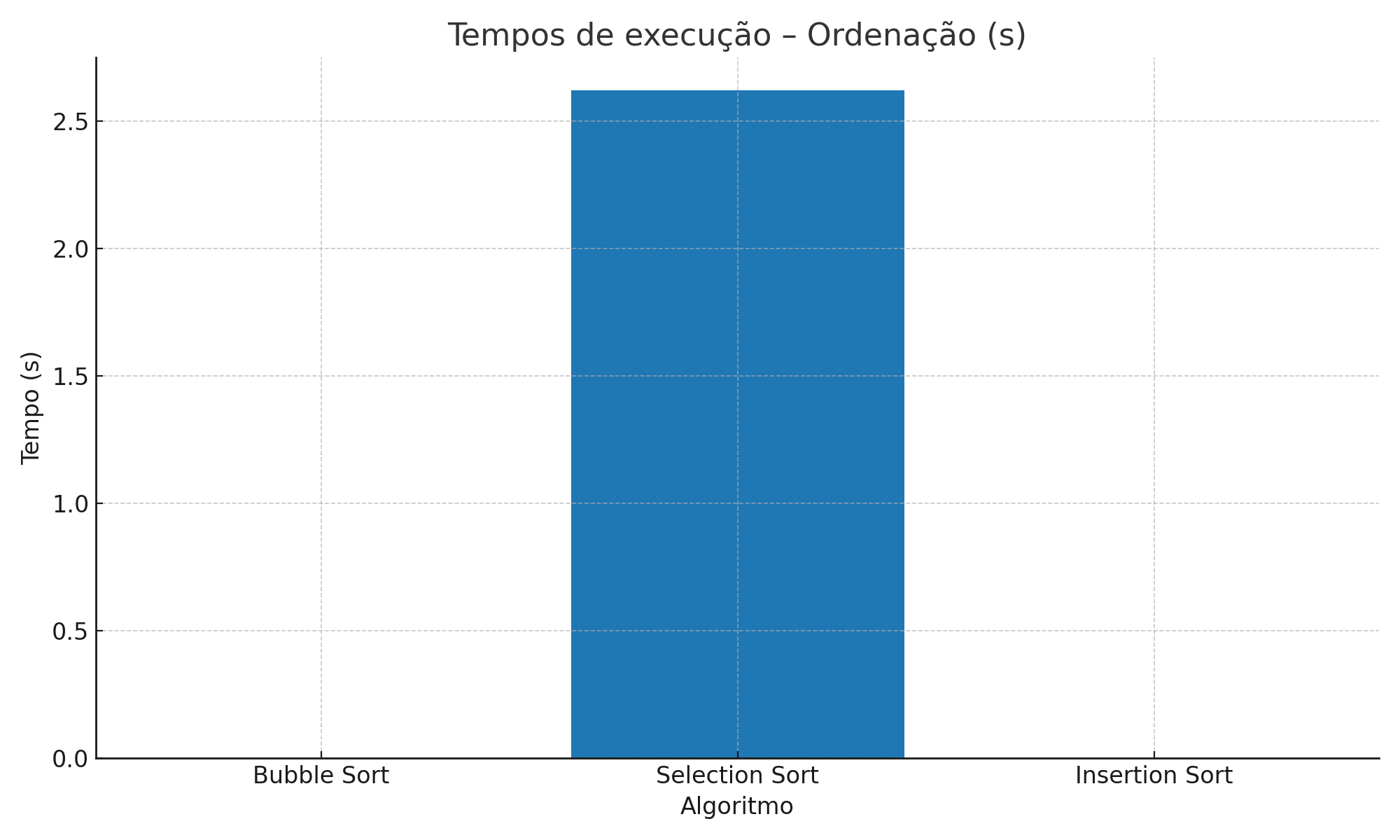












# 7. Discussão: por que os tempos e memórias diferem?

• Hashtable usa tabela de dispersão com slots; isso eleva o pico de memória na construção em relação a list/deque. Além disso, ao inserir novas chaves pode ocorrer realocação/redistribuição da tabela, mas com 100 itens adicionais o crescimento observado foi contido.

• list (Pilha) é um array dinâmico contíguo com sobrealocação: quando a capacidade é excedida, realoca um bloco maior e copia os ponteiros, o que pode gerar picos de memória notáveis em ondas. Isso explica por que o push de 100 itens pode ter apresentado pico maior que o deque e até que o dict.

• deque é composto por blocos (buffers) encadeados, evitando cópias ao crescer nas extremidades; por isso, os picos em enqueue/dequeue tendem a ser menores e o tempo para popleft permanece O(1) real.

• Nas recuperações por posição, todas aparecem próximas de zero por serem O(1), mas o deque pode apresentar leve overhead na indexação por ser segmentado; ainda assim, as diferenças são pequenas e compatíveis com ruído de medição e custo do próprio tracemalloc.

• Para ordenação, Selection mantém O(N²) independentemente da ordem inicial; Bubble (com flag) e Insertion exploram sequências já/quase ordenadas, justificando os tempos visivelmente menores observados nesses dois.

# 8. Conclusões

Pilha (list) é excelente para acesso por índice e operações de topo; deque é a escolha certa para filas com remoção no início; Hashtable (dict) oferece acesso por chave em O(1) médio ao custo de memória superior. Entre os algoritmos didáticos, Insertion e Bubble (com flag) são aceitáveis para listas pequenas/quase ordenadas; no geral, prefira o sort nativo do Python (Timsort) para grandes N.