Relatório Técnico – Implementação de EDs Lineares em Python

1. Estruturas de Dados Utilizadas

Foram implementadas 4 estruturas lineares baseadas na classe abstrata `EstruturaLinear`:

- Pilha (LIFO):
  - Implementada com lista nativa do Python (`_dados`).
  - Operações: `push` (insere topo), `pop` (remove topo), `top` (acessa topo).

- Fila (FIFO):
  - Implementada com lista nativa (`_dados`).
  - Operações: `enqueue` (insere final), `dequeue` (remove início), `front` (acessa início).

- Array Dinâmico:
  - Implementado com lista nativa (`_dados`).
  - Suporta acesso direto por índice e operações em qualquer posição.

- Lista Encadeada Simples:
  - Utiliza nós (`NoSimples`) com `dado` e ponteiro `proximo`.
  - Mantém referências para `cabeca` (início) e `tail` (fim) para otimização.

2. Divisão de Módulos

- Classe Abstrata `EstruturaLinear`:
  - Define 20 métodos abstratos e concretos para operações comuns (inserção, remoção, acesso, ordenação).
  - Métodos como `bubble_sort` e `estrutura_cheia` têm implementação padrão opcional.

- Subclasses Concretas:
  - `Pilha`, `Fila`, `Array`, `ListaEncadeadaSimples` implementam os métodos conforme suas especificidades.
  - Cada classe trata iteráveis no construtor para inicialização flexível.

3. Descrição das Rotinas e Funções

Método	Pilha	Fila	Array	Lista Encadeada
`inserir_topo`	Não aplic…	-	O(n)[1]	O(1)
`inserir_ultimo`	O(1)	O(1)	O(1)	O(1)
`remover_topo`	-	O(n)[2]	O(n)[1]	O(1)
`remover_ultimo`	O(1)	-	O(1)	O(n)[3]
`obter_index`	-	-	O(1)	O(n)
`bubble_sort`	-	-	O(n²)	-

Obs.:

[1]: Deslocamento de elementos em lista.
[2]: `pop(0)` desloca toda a lista.
[3]: Requer busca linear até o penúltimo nó.

Métodos Notáveis:

- `ListaEncadeadaSimples.remover_ultimo`: Busca sequencial até o penúltimo nó (O(n)).
- `Array.bubble_sort`: Implementa ordenação in-place com swap direto.
- `ListaEncadeadaSimples.__iter__`: Iterator para percorrer dados via `yield`.

4. Complexidade de Tempo e Espaço

Operação	Pilha/Fila (Lista)	Array (Lista)	Lista Encadeada
Inserção no Início	-	O(n)	O(1)
Inserção no Fim	O(1)*	O(1)*	O(1)
Remoção no Início	-	O(n)	O(1)
Remoção no Fim	O(1) (Pilha)	O(1)	O(n)
Acesso por Índice	-	O(1)	O(n)
Espaço de Memória	O(n)	O(n)	O(n) + ponteir…

Obs*: Amortizado devido a realocações em listas Python.

Comparação com Implementações Nativas do Python:

- Fila: `dequeue` é O(n) aqui vs. O(1) em `collections.deque`.
- Pilha: `push`/`pop` equivalem à lista nativa (O(1) amortizado).
- Lista Encadeada: Inserção no fim é O(1) (igual a `deque`), mas remover fim é O(n) vs. O(1) em listas duplas.

5. Problemas e Observações

- Fila Ineficiente:

`dequeue` usa `pop(0)` com complexidade O(n) por causa do deslocamento dos elementos.
Solução Sugerida: Usar `collections.deque` ou lista encadeada para O(1).

- Lista Encadeada:

- `remover_ultimo` é O(n) por falta de ponteiro `anterior` (ineficiente para grandes volumes).
- Falta tratamento para concorrência durante iteração.

- Pilha:

`inserir_topo` lança exceção, mas poderia ser mapeado para `push` se "topo" = início.

- Tipagem:

Uso de `Optional` (ex: `proximo: Optional[NoSimples]`) esclarece que valores podem ser `None`, melhorando legibilidade e prevenindo erros.

- Bubble Sort:

Ordenação ineficiente ($O(n^2)$) para `Array`. Em Python nativo, `list.sort()` usa Timsort (O(n log n)).

6. Conclusão

Resultados Obtidos:

- Implementação coesa de estruturas lineares com herança bem aplicada.
- Operações básicas (push/pop, enqueue/dequeue) eficientes, exceto `dequeue` em `Fila`.
- `ListaEncadeadaSimples` funcional, mas com limitações em remoção no fim.

Geral:

O código segue boas práticas de POO e tipagem, mas otimizações em operações críticas (especialmente filas e remoção em listas) são necessárias para cenários de alta performance. O uso de `Optional` e `__slots__` demonstra atenção a detalhes de desempenho e clareza.