| UFRPE | **AVALIAÇÕES 2025**<br>**2ª. VA – 2º. SEMESTRE** | |
|---|---|---|
| | Fundamentos de Problemas Computacionais 1 | |
| | **Curso:** BSI | **Turma:** SI1 |
| **Professor:**<br>**Lucas Fernando da Silva Cambuim** | | |

A segunda verificação de aprendizagem (2ªVA) constará de uma lista de 3 projetos de implementação envolvendo os conteúdos vistos em sala de aula. As soluções deverão ser apresentadas no dia 04/12/2025, quinta-feira, no horário da aula, entre 08:00h e 12:00h. As regras a seguir definem como será a avaliação e como deverá ser resolvida:

(1) Esta lista será resolvida no máximo em dupla. Pode ser também individual.
(2) Esta lista tem peso de 100% na composição da nota da 2ªVA.
(3) No dia da apresentação cada um terá um prazo de 20 minutos no máximo para apresentar todas as soluções conforme a descrição de cada projeto.
(4) Elaboração de slides para explicação de cada solução é opcional.
(5) A nota será determinada de acordo com os seguintes critérios:
(5.1) Corretude: Se a solução está funcionando corretamente (Peso 30% na nota do exercício)
(5.2) Clareza na apresentação: Se foi bem explicado, se respondeu bem as perguntas do professor, etc. (Peso 60% na nota do exercício)
(5.3) Legibilidade de código: Se código é fácil de ser compreendido (Peso 10% na nota do exercício).
(6) A nota da 2ªVA é uma média das notas de cada projeto seguindo os critérios definidos na regra 5.
(7) O professor poderá testar o código e realizar perguntas.
(8) A implementação pode ser realizada em Python, C, C++ ou Java. Recomendo a utilização de Python, uma vez que as aulas têm sido conduzidas nessa linguagem e, adicionalmente, a implementação das soluções tende a ser mais ágil nela.
(9) A solução deverá ser submetida também na plataforma classroom antes do início das apresentações, na aba de exercícios, com o título 2ªVA.
(10) A avaliação é apenas presencial, no dia e horário estabelecidos. O envio pela plataforma destina-se apenas a eventuais análises adicionais que o professor possa precisar realizar posteriormente.
(11) Não utilize o ChatGPT para desenvolver as soluções. Lembre-se de que se trata de uma prova da 2ª VA. Caso eu identifique o uso da ferramenta, a solução não será aceita e os integrantes receberão nota zero no respectivo projeto.
(12) Caso eu identifique plágio, a nota não será considerada para ambas as equipes para o respectivo projeto: tanto para aquela que forneceu o material quanto para aquela que o copiou.
(13) No dia da apresentação haverá uma ata de presença. Portanto, para que recebam a nota, os integrantes **deverão comparecer no dia da prova**.
(14) A implementação envolverá estritamente os conceitos aprendidos em sala de aula. A menos que esteja explicado no projeto, se existir funções, operações ou

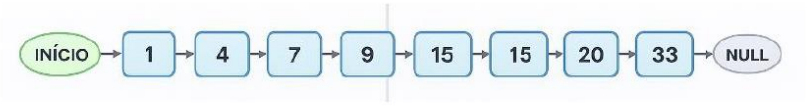
estruturas que não foram contemplados em sala de aula o aluno poderá ter penalização na nota.

Orientações:
(1) No dia da apresentação, o aluno deverá apresentar as soluções em uma máquina que pode ser a da sala de aula ou qualquer outro desktop/notebook de sua preferência. Para reduzir os riscos de não conseguir apresentar as soluções, recomenda-se salvar os arquivos .py em um pendrive.
(2) Para evitar atrasos e eventual comprometimento da nota, no momento da apresentação abra os códigos e realize os testes antes de o professor iniciar a avaliação.
(3) Qualquer dúvida sobre o entendimento do projeto ou sobre a implementação, não hesite em procurar o professor.

A seguir tem-se a descrição dos projetos a serem implementados para a 2ªVA.

**Projeto 1**

Imagine que você trabalha em um grande e-commerce com grande quantidade de produtos armazenados em uma lista encadeada ordenada por ID conforme ilustração abaixo:

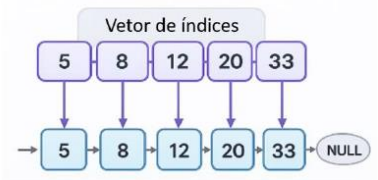



Certamente que cada nó armazenaria informações sobre o respectivo produto, mas para esse projeto não é necessário ter que armazenar essas informações. Somente o ID do produto é necessário.

A lista encadeada foi escolhida no passado porque:
- Inserções eram baratas
- Remoções eram simples
- O sistema era fácil de manter

Porém, agora o sistema cresceu e as buscas por ID ficaram extremamente lentas, pois em uma lista encadeada a busca exige percorrer nó por nó. Com milhões de produtos, isso se torna inviável.

O sistema não pode ser reescrito agora, porque diversas partes dependem da lista encadeada. Então a equipe pensou em criar uma solução intermediária usando um vetor que armazena o ID e a referência para o respectivo nó como mostrado a seguir:



Abaixo, tem-se uma sugestão da estrutura desse vetor de índices:

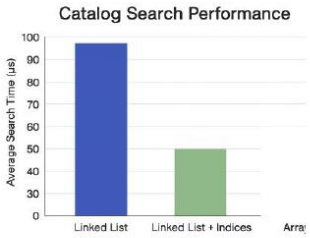
```
[(id1, referencia_para_no1),
 (id2, referencia_para_no2),
 (id3, referencia_para_no3),
            ...
 (idM, referencia_para_noM)]
```

Uma vez que a lista encadeada já está ordenada pelo ID, esse vetor naturalmente estará também ordenado. Então você poderá e deverá usar a busca binária no vetor para localizar o nó correspondente.

Você deverá implementar primeiro a abordagem de lista encadeada simples. Nessa abordagem você deverá implementar as operações de inserção de ID na lista ordenado durante a inserção. O arquivo projeto_1_lista_IDs_entrada.txt contendo 1000 IDs a serem armazenados é disponibilizado. Cada ID é separado por uma quebra de linha. Você também deverá implementar a operação de busca sequencial que deverá retornar 1 se a busca encontrou o número solicitado como parâmetro e -1 caso contrário.

Uma vez que você tem a lista encadeada preenchida você deverá implementar a segunda abordagem que acrescenta o vetor de índices. Nessa abordagem você deverá implementar a busca binária nesse vetor para localizar o respectivo nó da lista encadeada.

Com as duas abordagens implementadas e suas respectivas operações de busca e inserção você deverá compará-los, medindo o tempo total em milissegundos ou microssegundos para a realização de todas buscas em cada abordagem. Um gráfico de barra pode ser apresentado para demonstrar as diferenças de tempo de busca das duas abordagens. Um exemplo dessa comparação é mostrado na figura a seguir:




O arquivo projeto_1_lista_IDs_busca.txt contendo os IDs a serem buscados é disponibilizado. Cada ID é separado por uma quebra de linha, como mostrado no exemplo a seguir:

```
101
55
203
11
98
```

Além disso, você deverá gerar dois arquivos de saída chamados projeto_1_resultado_busca_sequencial.txt e projeto_1_resultado_busca_binaria.txt contendo o sucesso e insucesso da busca para cada ID consultado nas duas abordagens de busca. Esses arquivos precisam ser gerados automaticamente ao executar cada abordagem. Para o sucesso você deverá informar o valor 1 e o insucesso o valor de -1. Cada valor de saída deverá ser separado por quebra de linha. A figura a seguir mostra um exemplo hipotético de um arquivo de saída.

```
1
-1
1
1
-1
```

O arquivo projeto_1_lista_IDs_resultado.txt é disponibilizado contendo as respostas esperadas que servirá para validar a implementação de cada abordagem.

Para medir tempo de execução em Python existem várias funções disponíveis. Você pode, por exemplo, utilizar a função perf_counter da biblioteca time. O trecho de código a seguir mostra um modelo de uso da função perf_counter.

```python
import time

inicio = time.perf_counter()
# código para realização da busca
#
#
fim = time.perf_counter()

# apresentando o tempo final
print("Tempo:", fim - inicio, "s")
```

**Projeto 2**

A empresa de ecommerce do projeto 1 está passando por uma fase de crescimento acelerado.
O catálogo de produtos, que antes tinha em torno de 1000 itens, agora ultrapassa:

- 100 mil produtos ativos,
- milhares de novas variações,
- inserções diárias vindas de fornecedores,
- e consultas simultâneas vindas de clientes navegando pela loja.

Nesse cenário, um recém-contratado do time de tecnologia, empolgado para mostrar serviço, propõe uma ideia ousada: "Por que não usamos uma tabela hash para gerenciar todos os IDs dos produtos? Assim a busca por qualquer item fica praticamente instantânea."

A sugestão surge depois de ele observar que:

- O sistema atual usa uma lista encadeada ordenada para armazenar IDs.

- As consultas para verificar se um produto existe ou não estão ficando cada vez mais lentas.
- O vetor ordenado com busca binária ajuda nas buscas, mas é pouco eficiente para lidar com a enorme quantidade de novos produtos adicionados diariamente.
- Durante promoções, datas sazonais e grandes campanhas, o número de requisições simultâneas explode.

Ele argumenta que uma tabela hash pode:
- inserir novos produtos praticamente em O(1),
- fazer buscas com velocidade constante,
- escalar muito melhor que as outras estruturas,
- manter o catálogo rápido mesmo com milhões de itens,
- melhorar significativamente a experiência do usuário.

Você deverá implementar nesse projeto 2 a abordagem de tabela hash para o desafio de catálogo de produtos. Você deverá implementar a lógica de inserção dos IDs na tabela hash. A implementação deverá contemplar:
- Uma tabela hash de tamanho adequado. Não existe um único tamanho correto. Ele deve ser escolhido conforme o objetivo. A regra geral é: tamanho da tabela ≈ número de elementos esperados / fator de carga. O fator de carga recomendado costuma ser: 0,5 → excelente desempenho; 0,7 → bom desempenho; 0,9 → mais economia de memória, mas aumenta colisões
- Para esse projeto considere o fator de carga de 0,7. Ou seja, o tamanho da tabela será de **7000**.
- Uma função de hash. Como sugestão pode-se utilizar uma função hash simples, como:

```
hash(id) = id % tamanho_tabela
```

- Tratar colisões por uma lista encadeada. Ou seja, cada posição da tabela não guarda um único valor, mas sim uma lista encadeada.

Para cada ID lido do arquivo de entrada o programa deverá:

- Criar o nó (ID + ponteiro).
- Calcular o valor hash.
- Inserir o nó no espaço correspondente.

Você também deverá implementar a lógica de busca de IDs na tabela hash que deverá retornar 1 se a busca encontrou o número passado como parâmetro e -1 caso contrário.

O arquivo projeto_2_lista_IDs_entrada.txt contendo os IDs a serem armazenados será fornecido. Esse arquivo contém os IDs separados por uma quebra de linha, similar ao projeto 1. Você deverá armazenar esses dados na tabela hash.

Uma vez que você tem a tabela hash preenchida você deverá medir o tempo total para a realização de todas buscas solicitadas. O arquivo projeto_2_lista_IDs_busca.txt

contendo os IDs a serem buscados é fornecido. O arquivo contém os IDs separados por uma quebra de linha, similar ao projeto 1.

Além disso, você deverá comparar a abordagem de tabela hash com as outras duas abordagens do projeto 1, lista encadeada ordenada com busca sequencial e lista encadeada com vetor de índices com busca binária, medindo o tempo total para a realização de todas buscas do arquivo `projeto_2_lista_IDs_busca.txt` nas três abordagens. Cuidado! Você deverá, primeiro, preencher a lista encadeada com os IDs do arquivo `projeto_2_lista_IDs_entrada.txt` para poder realizar as buscas nas duas abordagens do projeto 1. Um gráfico de barra pode ser apresentado para demonstrar as diferenças de tempo de busca das três abordagens, similar ao apresentado no projeto 1.

**Projeto 3**

A mesma empresa de e-commerce dos projetos 1 e 2 está passando por uma reestruturação em um outro módulo, o de listagem de produtos. Um integrante do time de desenvolvimento sugeriu utilizar algoritmos de ordenação implementados manualmente para melhorar o desempenho e permitir maior controle sobre o comportamento das listagens.

Atualmente, os produtos são exibidos com base em uma lógica fixa, mas os gestores querem oferecer ao usuário a opção de ordenar a lista por diferentes critérios:

- Preço
- Estoque
- Popularidade (número de vendas)
- Data de cadastro
- Nome do produto

O time deseja comparar diferentes algoritmos de ordenação para avaliar qual se adapta melhor a diferentes tamanhos e tipos de dados e você foi encarregado de implementar essa função. Para esse projeto você deverá comparar apenas os algoritmos de **inserção** e **intercalação** aprendidos em sala de aula.

A função tem a seguinte assinatura:

```
ordenar_produtos(lista_produtos, criterio, algoritmo)
```
,

que recebe:

- lista_produtos: uma lista de dicionários, onde cada dicionário representa um produto com os seguintes campos e valores:
  - "id" : (string)
  - "nome" : (string)
  - "preco" : (float)
  - "estoque" : (int)
  - "popularidade" : (int)
  - "data_cadastro" : (string no formato "YYYY-MM-DD")
- criterio: uma string indicando o critério de ordenação:
  - "preco"

    - "estoque"
    - "popularidade"
    - "data"
    - "nome"
- algoritmo: string indicando qual algoritmo você deve usar:
    - "insercao"
    - "intercalacao"

A função deve **retornar uma nova lista ordenada** com base no critério e no algoritmo especificado.

A comparação será realizada medindo o tempo de execução para realizar um conjunto de ordenação solicitada para cada algoritmo. O arquivo projeto_3_lista_produtos_entrada.txt contém a lista de produtos e será disponibilizado para que você carregue-o no dicionário. O formato de armazenamento de cada linha é: id | nome | preco | estoque | popularidade | data_cadastro

Depois de carregado você deverá executar as requisições de ordenação a partir do arquivo projeto_3_requisicoes_listagem.txt . Esse arquivo contém a informação do critério de ordenação (ou seja, “preco”, “estoque”, “popularidade”, “data” ou “nome”) em cada linha do arquivo. Você deverá executar todas as requisições para cada algoritmo e medir o tempo total.

Para cada critério você deve gerar um arquivo de saída contendo a lista ordenada seguindo o critério solicitado no mesmo padrão de formatação do arquivo de entrada. O nome do arquivo de saída deverá seguir o fomato “projeto_3_resultado_A_B”, onde A é nome do algoritmo utilizado e B é o nome do critério escolhido. Por exemplo: para o algoritmo “insercao” e o critério preco então o nome do arquivo é “projeto_3_resultado_inserção_preco.txt”.

São disponibilizados também os arquivos de referência contendo a ordem correta esperada para cada critério escolhido para fins de verificação de corretude. Os nomes dos arquivos são:

- projeto_3_resultado_esperado_preco
- projeto_3_resultado_esperado_data
- projeto_3_resultado_esperado_estoque
- projeto_3_resultado_esperado_nome
- projeto_3_resultado_esperado_popularidade