Estruturas de Dados e Introdução à Projeto e Análise de Algoritmos 2024/2

Trabalho Prático T2

January 15, 2026

Leia atentamente **todo** esse documento de especificação. Certifique-se de que você entendeu tudo que está escrito aqui. Havendo dúvidas ou problemas, fale com o professor o quanto antes.

1 Objetivo

O objetivo deste trabalho é relacionado à implementação de árvores B, destinadas a indexação de registros focados em dados armazenados em memória secundária. Para fins de simplificação, esta estrutura deve ser implementada para manipulação e armazenamento de dados em um arquivo binário, que simula o armazenamento secundário. Será necessária a utilização e extensão de estruturas de dados e técnicas de busca e ordenação vistas em aula.

2 O problema de indexação de registros em armazenamento secundário

Em áreas como bancos de dados, por exemplo, a indexação é empregada como meio de otimizar a busca e recuperação de registros, permitindo que consultas sejam realizadas de forma eficiente. O problema da indexação no contexto do armazenamento secundário refere-se a como organizar e acessar rapidamente grandes volumes de dados armazenados de forma permanente, sem sobrecarregar o sistema ou consumir recursos excessivos, e levando em consideração o alto custo envolvido em acessos repetidos a dispositivos de armazenamento secundário, custo este muito mais elevado quando comparado a acesso em memória principal (RAM).
Quando um registro ou conjunto de dados precisa ser acessado em um dispositivo de armazenamento secundário, a operação pode ser muito lenta se não houver um índice adequado e a busca exigir acessos sequenciais e exaustivos nos dados, já que o tempo de acesso à informação pode envolver leituras de grandes quantidades de dados não relacionados ao dados específicamento requisitados. Portanto, uma boa indexação pode diminuir drasticamente o tempo de resposta e aumentar a eficiência do sistema.
O problema de indexação de registros em armazenamento secundário é uma questão fundamental de sistemas de gerenciamento de bancos de dados e sistemas de arquivos, especialmente quando lida-se com grandes volumes de dados que não podem ser mantidos/persistidos na memória principal (RAM). O armazenamento secundário, popularmente representado por dispositivos como discos rígidos (HD), unidades de estado sólido (SSD) e outros dispositivos de armazenamento permanente, tem um desempenho significativamente mais lento do que a memória principal, exigindo soluções eficientes para a indexação e recuperação dos dados.
Quanto a implementação de indexação e acesso a registros organizados em armazenamento secundário, os seguintes desafios devem ser considerados:

- **Tempo de acesso:** A latência associada ao acesso aos dados no disco descreve um grande problema de acesso em armazenamento secundário. Dispositivos desta natureza tem tempos de busca elevados devido ao movimento físico das cabeças de leitura e gravação. Mesmo SSDs, que são mais rápidos, ainda têm um tempo de acesso superior ao da memória principal. Isso significa que, para uma consulta eficiente, é necessário minimizar o número de acessos ao disco.
- **Tamanho dos dados:** Quando os dados não cabem na memória principal e precisam ser armazenados e lidos do armazenamento secundário, o sistema de indexação deve ser capaz de gerenciar grandes volumes de dados. Isso pode exigir o uso de algoritmos eficientes de indexação e estrutura de dados que permitam a busca eficiente para lidar com questões de custo de acesso e manipulação de conjuntos massivos de dados.

- **Estruturas de Dados:** A escolha das estruturas de dados adequadas para indexação é essencial para garantir o desempenho de acesso e armazenamento dos dados. Estruturas como **árvores B**, árvores B+, índices hash e índices baseados em árvores de busca balanceadas são comuns em bancos de dados e sistemas de arquivos, uma vez que a eficiência das operações está diretamente relacionada a aspectos como a distribuição dos dados e a frequência de atualização dos registros implementadas por estas estruturas. A manutenção de índices em uma solução implementada para armazenamento secundário pode ser complexa. A inserção, atualização e exclusão de registros podem exigir modificações frequentes nos índices. Além disso, a reorganização do índice ao longo do tempo (para garantir eficiência) é um processo que deve ser gerido de forma eficiente.

3 A estrutura "Árvore B"

As árvores B são estruturas de dados balanceadas, autoajustáveis e ordenadas, amplamente utilizadas em sistemas de gerenciamento de bancos de dados e sistemas de arquivos para otimizar a busca, inserção e exclusão de dados em grandes volumes. São uma generalização das árvores binárias de busca, adaptadas para permitir que múltiplos elementos (ou chaves) sejam armazenados em cada nó, o que ajuda a reduzir o número de acessos ao disco e a melhorar o desempenho em sistemas de armazenamento secundário.

3.1 Propriedades da Árvore B

Estas estruturas apresentam as seguintes propriedades:

- **Balanceamento:** A árvore B é intrinsicamente balanceada, ou seja, todas as folhas estão localizadas no mesmo nível, garantindo que o tempo de acesso aos dados seja "$O(logn)$", onde n é o número total de elementos. Diferente da degeneração de balanceamento que pode ocorrer em árvores binárias de busca tradicionais que as levam a crescer desproporcionalmente, o desempenho em árvores B é garantido uma vez que a estrutura se mantém sempre balanceada.

- **Nó com múltiplas chaves:** Generalizando as árvores binárias, as árvores B permitem armazenar múltiplas chaves dentro de um único nó. Esta característica reduz o número de acessos ao disco, o que é especialmente importante uma vez que são mais lentos do que acessos a memória principal.

- **Estrutura de árvore d-ária:** As árvores B possuem uma órdem, aqui definida como d, que estabelece os números máximo e mínimo de filhos para cada nó de suas estruturas. Assim, cada nó pode ter até d filhos, no máximo $d-1$ e no mínimo ($\lceil(d/2)\rceil - 1$) chaves (exceto a raíz que pode ter até d filhos, no máximo $d-1$ chaves e no mínimo 1 chave). A árvore garante o balanceamento durante as operações de inserção e remoção de dados por meio de redistribuições e concatenações de registros e pela criação de novos nós elegíveis a folhas, internos ou nova raíz. Os nós internos contêm chaves que atuam como delimitadores, dividindo o intervalo de chaves armazenadas nos seus filhos. Importante: para árvore com ordem 2, um tratamento especial deve ser realizado com relação ao mínimo de chaves permitido para não permitir que a estrutura fique vazia.

- ***Observação:*** **Note que esta formalização de árvore-B refere-se a definição pela ordem, e não pelo grau mínimo t, que normalmente só admite árvores B de grau par. Pode-se alternar entre as duas definições, por comodidade, dependendo da ordem da árvore B.**

3.2 Operações em Árvore B

As operações elementares realizadas em árvores B incluem busca, inserção e remoção:

- **Busca:** Realizada de forma eficiente, a busca explora as chaves dentro de cada nó para decidir em qual subárvore continuar buscando. Percorre-se a árvore de forma similar a uma árvore binária, mas como cada nó pode conter múltiplas chaves, reduz-se a profundidade (e consequentemente, os acessos), reduzindo o número de comparações.

- **Inserção:** A inserção inicia-se pela raiz e desce pela árvore até encontrar um nó folha onde a chave pode ser inserida. Se o nó folha apresenta menos que $d-1$ chaves, a chave é inserida diretamente, e caso contrário, é realizada uma uma divisão de nó onde a chave do meio é promovida para o nó pai, e o nó é dividido em dois. Esta divisão pode se propagar pela estrutura da árvore (de baixo para cima), caso os nós pais também estejam cheios. Todas as inserções acontecem em nós folha.
- **Remoção:** A remoção é realizada de forma semelhante à inserção, mas pode envolver a fusão de nós ou a redistribuição de chaves entre nós irmãos para manter a árvore balanceada. Caso a chave a ser removida esteja em um nó folha e o nó conter chaves suficientes (manter a propriedade de pelo menos $\lceil (d/2) \rceil - 1$ chaves (exceto a raíz que pode ter até d filhos, no máximo $d-1$ chaves e no mínimo 1 chave). A árvore garante o balanceamento durante as operações de inserção e remoção de dados por meio de redistr chaves), basta removê-la diretamente, e caso contrário (o nó ficar com menos do que $\lceil (d/2) \rceil - 1$ chaves após a remoção ou vazio se for a raíz), ele pode se fundir com um irmão, pegar uma chave emprestada de um irmão, ou mesmo pegar uma chave de um dos filhos para manter as propriedades da árvore B.

4 Formalização e exemplo da aplicação de árvore B

Os nós da árvore B armazenam também os registros de dados, juntamente com as chaves responsáveis pela indexação (diferentemente da árvore B+, onde os nós internos somente servem para indexação, e os registros concentram-se todos nos nós folha da estrutura). Assim, quando um nó é visitado, nele devem estar armazenados não somente as chaves mas também os dados referentes às posições representadas pelos índices. Estas estruturas destinam-se, em grande maioria, a indexação de dados em localizados em armazenamento secundário, todavia, para fins de simplificação, neste trabalho o objetivo é implementar a estrutura e seu funcionamento em Python em armazenamento principal (ou seja, serão utilizados apontamentos simples como os já vistos para árvores binárias). A estrutura de uma árvore B, definida como BT, possui os seguintes atributos:

- A ordem d da árvore
- O ponteiro para a raíz da árvore
- O número de nós da árvore

A árvore B BT pode ser composta por diversos nós. Portanto, a estrutura nó de uma árvore B de ordem d, deve conter os seguintes campos:

- O número de chaves armazenadas no nó (lembrando que para nós internos e folhas, o mínimo de chaves é $\lceil (d/2) \rceil - 1$ e o máximo é $d-1$, e para a raíz, o mínimo de chaves é 1 e o máximo é $d-1$)
- Uma *flag* que indica se o nó é folha ou não
- Um conjunto de chaves armazenados no nó (lembrando que este número está no intervalo $[\lceil (d/2) \rceil - 1, d-1]$ para nós internos e folhas e entre $[1, d-1]$ para a raíz)
- Um conjunto de registros armazenados, cada um indexado por uma chave
- Um conjunto de filhos deste nó (sempre igual ao número de chaves armazenadas + 1)

5 Representação da Árvore B em Bytes e sua manipulação em Python

Formalizada a estrutura de um nó que compõe a árvore B, podem-se ilustrar esta estrutura de forma didática, como pode ser observado na Figura 1. Em sua implementação em memória principal, os filhos dos nós da árvore B são basicamente apontamenos entre os nós, como já foi profundamente explorado para diversas estruturas como árvores binárias de busca. Para a simplificação do armazenamento secundário definido nestre trabalho, cada nó da árvore ocupa uma posição no arquivo binário, sendo estas posições referentes a deslocamentos em bytes do tamanho das estruturas nós que compõem a árvore B. Por exemplo, para uma árvore com três nós, a raíz está na posição $0 * sizeof(node)$, o primeiro filho está na posição $1 * sizeof(node)$ e o terceiro filho está em $2 * sizeof(node)$ do arquivo binário.

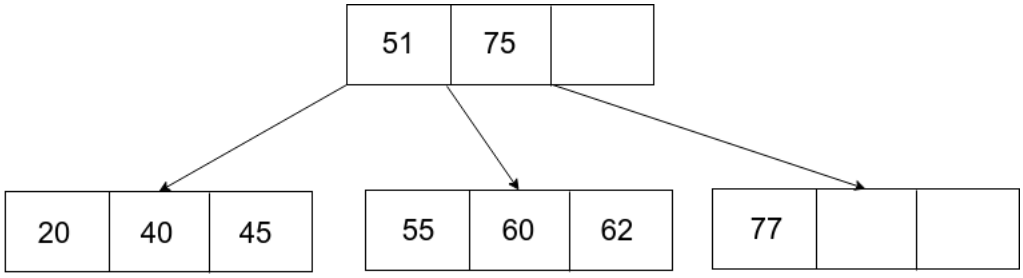

Figure 1: Exemplo de árvore B com ordem 4 e armazenamento em memória principal. A relação entre nós e filhos é feita com ponteiros entre os nós.

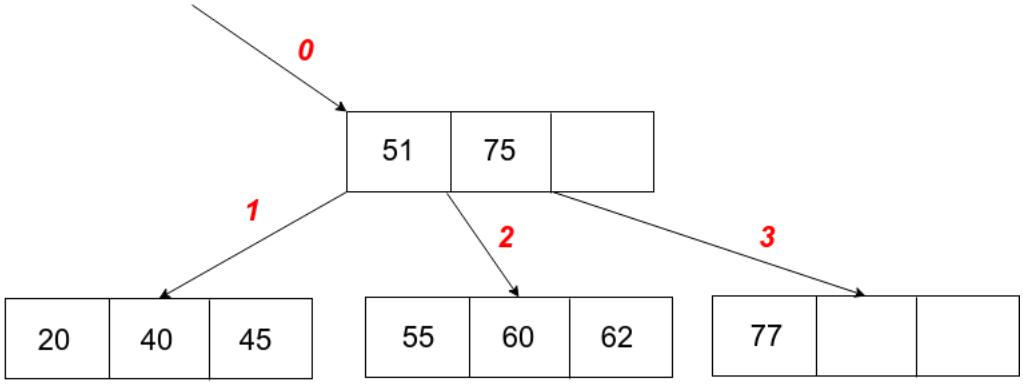

Figure 2: Exemplo de árvore B com ordem 4 e armazenamento em memória secundária. A relação entre nós e filhos é feita por deslocamentos dentro do arquivo binário, considerando o tamanho em bytes de cada nó.

Para a implementação em armazenamento secundário, é importante destacar que o $offset$ deve ser calculado para criação, armazenamento e manipulação dos nós da árvore em ar, sendo este acesso no arquivo binário necessário para manipular cada um dos nós da árvore B (e o comando seek de arquivos deve ser devidamente utilizado para o acesso apropriado dos nós da árvore do armazenamento secundário para a memória principal). A Figura 2 ilustra esta representação da árvore B em arquivo binário. O arquivo binário deve ser criado na criação do primeiro nó da árvore B, deve ser mantido durante todo o processo de crescimento da árvore, e deve ser apagado na finalização da execução do programa.
Como pode ser observado na Figura 2, a cada visita a um dado nó i em BT, uma leitura no arquivo binário deve ser realizada. Para que as alterações sejam refletidas nos dados persistido, remoções e inserções também devem alterar o dado salvo no arquivo binário. Estas manipulações são realizadas com leituras e escritas no arquivo binário, nas posições respectivas dos nós de BT utilizando as operações *f.read* e *f.write* de arquivos.
No **Python** as classes tem layout, ou tamanho em *bytes* variável, e portanto, a representação em arquivos binários pode ser dificultada. Todavia, é possível forçar o mesmo comportamento que a linguagem C apresenta, possibilitando a representação dos nós da árvore B com o mesmo tamanho (a classe, neste caso), exatamente como uma *struct* do C, em termos de tamanhos de *bytes* para todas as instâncias desta classe (neste caso, classe nó que compõe a árvore).

5.1 Manipulando arquivos binários em Python

O código a seguir demonstra as operações de leitura e escrita de dados, referentes a uma classe, no arquivo binário (presente no mesmo diretório do arquivo sendo executado).

```
# leitura binária
arquivo = open("dados.bin", "rb")

# escrita binária
arquivo = open("dados.bin", "wb")
```

```
# leitura e escrita
arquivo = open("dados.bin", "r+b")
```

O mais recomendado é usar with (fecha automaticamente).

```
with open("dados.bin", "rb") as arquivo:
    dados = arquivo.read()
```

As operações de leitura e escrita podem ser observadas nos trechos de código abaixo.

```
# leitura de todo o conteúdo do arquivo
with open("arquivo.bin", "rb") as f:
    conteudo = f.read()

print(conteudo)        # bytes
print(type(conteudo))  # <class 'bytes'>

# leitura de um número específico em bytes
with open("arquivo.bin", "rb") as f:
    primeiros_10 = f.read(10)

# leitura byte a byte
with open("arquivo.bin", "rb") as f:
    while True:
        byte = f.read(1)
        if not byte:
            break
        print(byte)

# escrita em arquivo binário
dados = b'\x01\x02\x03\xFF'

with open("saida.bin", "wb") as f:
    f.write(dados)

# acesso a posições específicas do arquivo binário
with open("arquivo.bin", "rb") as f:
    f.seek(4)        # vai para o byte 4
    byte = f.read(1)
    posicao = f.tell()
```

5.2 Fixando o layout de classes em Python - simulando o comportamento da cláusula *struct* do C

Como citado anteriormente, não é possível de forma nativa a manipulação de *bytes* das classes como em C, mas é possível sim salvar e ler dados de uma classe em arquivo binário, usando outras abordagens. Em C, isso funciona porque a *struct* tem layout fixo na memória e *sizeof()* retorna exatamente o tamanho em *bytes*. Em Python, objetos não têm layout fixo, então não é possível executar um fwrite(&obj, sizeof(obj)).
Para realizar tal operação em Python, deve-se importar a classe *struct*, que possibilita fixar o layout das classes, possibilitando sua manipulação de forma fixa, em número de *bytes*, dentro do arquivo binário. Abaixo, um exemplo de representação de uma classe com a fixação do layout em *bytes* para sua manipulação precisa dentro do arquivo binário:

```
# definição da estrutura e seu layout fixo
import struct
```

```
class Pessoa:
    def __init__(self, idade, altura):
        self.idade = idade      # int
        self.altura = altura    # float

    def to_bytes(self):
        return struct.pack("if", self.idade, self.altura)

    @classmethod
    def from_bytes(cls, dados):
        idade, altura = struct.unpack("if", dados)
        return cls(idade, altura)

# salvar no arquivo binário
p = Pessoa(25, 1.75)

with open("pessoa.bin", "wb") as f:
    f.write(p.to_bytes())

#ler do arquivo binário
with open("pessoa.bin", "rb") as f:
    dados = f.read(struct.calcsize("if")) #calcsize faz extamente o papel do sizeof() do

p2 = Pessoa.from_bytes(dados)
print(p2.idade, p2.altura)
```

Observação: Observe que para a definição do layout fixo da classe em *bytes*, utiliza-se a string "if". Neste caso, ela descreve exatamente os tipos dos dados da classe, que para este exemplo, são um inteiro (i - 4 *bytes*), e um float (f - 4 *bytes*), totalizando 8 *bytes* para cada instância desta classe. Esta *string* utilizada na clausula *struct* está relacionada diretamente aos atributos da classe.
Finalmente, abaixo, um exemplo simples de indexação de dados de uma classe salvos utilizando arquivo binário (é muito similar a manipulação e manutenção dos nós da árvore B no arquivo binário, logicamente, respeitando o comportamento da estrutura). A lógica a ser seguida representa exatamente o process feito em C:

1. Definir registros de tamanho fixo
2. Calcular exatamente o tamanho de cada objeto
3. Calcular o deslocamento: $offset = indice * tamanho\_do\_registro$
4. Usar seek(offset)

```
# definição da estrutura
import struct
from dataclasses import dataclass

@dataclass
class Pessoa:
    id: int
    idade: int
    altura: float

    FORMATO = "iif"  # id, idade, altura
    TAMANHO = struct.calcsize(FORMATO)

    def to_bytes(self):
        return struct.pack(self.FORMATO, self.id, self.idade, self.altura)
```

```
    @classmethod
    def from_bytes(cls, dados):
        return cls(*struct.unpack(cls.FORMATO, dados))

# gravar vários registros no árquivo binário
pessoas = [
    Pessoa(1, 20, 1.70),
    Pessoa(2, 25, 1.80),
    Pessoa(3, 30, 1.65),
]

with open("pessoas.bin", "wb") as f:
    for p in pessoas:
        f.write(p.to_bytes())

# acesso por indexação
def ler_pessoa(indice):
    with open("pessoas.bin", "rb") as f:
        offset = indice * Pessoa.TAMANHO
        f.seek(offset)
        dados = f.read(Pessoa.TAMANHO)
        return Pessoa.from_bytes(dados)

p = ler_pessoa(1)  # segunda pessoa
print(p)
```

6 Sua tarefa

A sua tarefa será implementar, de forma modularizada e eficiente, a estrutura de dados árvore B para indexação em armazenamento secundário (ou em principal, em caso de problemas). Para isso, você deverá seguir os seguintes passos:

1. Sua entrada será um arquivo texto com: (i) a ordem da árvore B, (ii) um inteiro n que representa o número de operações que serão realizadas (inserções, remoções e buscas) não árvore B. As n linhas de operações podem ser:
    - Iniciadas pelo caractere I, representando a operação inserção, e seguidas por um par $chave, registro$ a ser inserido na árvore B
    - Iniciadas pelo caractere R, indicando a chave a ser removida, e seguida pela $chave$ do registro a ser removido da estrutura
    - Iniciadas pelo caractere B, indicando uma operação de busca, e seguida pela $chave$ do registro a ser buscado na árvore
2. Para os casos de busca de chaves, deve-se imprimir ”O REGISTRO ESTA NA ARVORE!” ou ”O REGISTRO NAO ESTA NA ARVORE!”, seguido de uma quebra de linha.
3. Deve-se implementar a estrutura árvore B tradicional, para indexação em memória principal.
4. Para todos os casos de teste (que serão sempre compostos por ordem, chaves/registros a serem inseridos, chaves dos registros a serem removidos e chaves dos registros a serem buscados), deve-se imprimir a estrutura da árvore ao final da execução do caso de teste como ilustrado na seção ”Entrada e Saída” desta especificação.

7 Entrada e Saída

7.1 Entrada

Todas as informações serão dadas em um arquivo texto: (i) a ordem da árvore B (d), (ii) o número operações na estrurura (n) que incluem (iii) inserções (I) e as chaves/dados a serem inseridos, remoções (R) e as chaves dos dados a serem removidos e buscas (B) e as chaves buscadas na árvore resultante. Para o exemplo das Figuras 1 e 2, o conteúdo arquivo de entrada é o seguinte:

```
4
15
I 20, 20
I 75, 75
I 77, 77
I 78, 78
I 55, 55
I 62, 62
I 51, 51
I 40, 40
I 60, 60
I 45, 45
R 78
B 15
B 40
B 25
B 78
```

7.2 Saída

A saída do trabalho deverá ser salva em um arquivo, também de texto, contendo os resultados das buscas e o estado final da árvore B, impressa em largura. Para o exemplo de entrada da seção anterior, a saída deverá ser como abaixo.

```
O REGISTRO NAO ESTA NA ARVORE!
O REGISTRO ESTA NA ARVORE!
O REGISTRO ESTA NA ARVORE!
O REGISTRO NAO ESTA NA ARVORE!

-- ARVORE B
[key: 51, key: 75, ]
[key: 20, key: 40, key: 45, ][key: 55, key: 60, key: 62, ][key: 77, ]
```

Como pode ser observado, a impressão da árvore B acontece em largura, então em cada linha são impressos os nós e chaves referentes àquele nível. Este padrão deve ser obedecido mesmo para árvores que apresentem profundidade maior do que a apresentada no exemplo. *Importante:* Na inserção de chaves repetidas, deve-se atualizar seu valor, comportamento este esperado para estruturas de armazenamento de registro e de indexação que não permitem a repetição de chaves de busca.

7.3 Execução do trabalho

Para testar seu trabalho, o professor executará comandos seguindo o seguinte padrão.

```
tar -xzvf <nome_arquivo>.tar.gz
python trab2.py <nome_arquivo_entrada> <nome_arquivo_saida>
```

É extremamente importante que este padrão seja seguido. Seu programa não deve solicitar a entrada de nenhum valor e também não deve imprimir nada na tela.
Por exemplo, se o nome do arquivo recebido for `2004209608.tar.gz`, os dados de entrada estiverem em `entrada.txt` e o nome do arquivo de saída for `saida.txt`, o professor executará:

```
tar -xzvf 2004209608.tar.gz
python trab2.py entrada.txt saida.txt
```

8 Sobre o funcionamento de árvores B

A parte central deste trabalho é a implementação da estrutura de dados árvore B em armazenamento secundário. Para tal, deve-se utilizar a aula disponibilizada no Classroom. Para ilustrar o crescimento da estrututura, estado da árvore B após inserções e remoções, recomenda-se a utilização do simulador web disponibilizado pela University of San Francisco[1]. Detalhes do algoritmo estão disponíveis no material sobre árvore B.

9 Detalhes de implementação

A seguir, alguns detalhes, comentários e dicas sobre a implementação. Muita atenção aos usuários do Sistema Operacional Windows.

- O trabalho deve ser implementado em Python. A versão do Python a ser considerada é a presente nos Computadores do LabGrad (Ubuntu).
- **Devem ser implementados no mínimo DOIS (2) TADs (tipos abstratos de dados) opacos.** A modularização é fortemente encorajada, e mais TADs podem ser implementados de acordo com a modelagem do problema a ser implementado. Sugere-se que a estrutura de árvore B e nós a serem seguidas sejam as apresentadas na seção "Formalização e exemplo da aplicação de árvore B", mas outras modelagens em termos de atributos e organização podem ser adotadas, uma vez que garantido o funcionamento adequado da estrutura.
- A proposta inicial deste trabalho é a implementação de árvore B em armazenamento secundário simulado em árquivo binário. Em caso da não simulação em armazenamento secundário (ou seja, o não uso do arquivo binário, mantendo a estrutura 100% em memória principal), a nota será multiplicada por 0,8.

10 Regras para desenvolvimento e entrega do trabalho

- **Data da Entrega**: O trabalho deve ser entregue até as 08:00h do dia 24/02/2026. Não serão aceitos trabalhos após esta data.
- **Como entregar**: Pela atividade criada no Classroom. Envie um arquivo compactado, no formato `.tar.gz`, com todo o seu trabalho. A sua submissão deve incluir todos os arquivos de código. Coloque a matrícula no nome do arquivo do trabalho. O padrão de nomes de arquivos para a entrega, portanto, é: **"T2-matricula-EDPAA-2025-2.tar.gz"**. É obrigatório respeitar este padrão. Também, coloque a matrícula e o nome nos cabeçalhos dos códigos fornecidos.
- **Recomendações**: Modularize o seu código adequadamente. Crie códigos claros e organizados. Utilize um estilo de programação consistente. Comente o seu código extensivamente. Não deixe para começar o trabalho na última hora.

11 Avaliação

- Assim como especificado no plano de ensino, o trabalho vale 10 pontos (implementação com simulação de armazenamento secundário utilizando árquivo binário). Para a implementação somente em memória principal, o trabalho vale, no máximo, 8 pontos.
- A parte de implementação será avaliada de acordo com a fração e tipos de casos de teste que seu trabalho for capaz de resolver de forma correta. Todos os casos de teste serão projetados para serem executados em pouco tempo em uma máquina com 16GB de RAM.

[1] `https://www.cs.usfca.edu/~galles/visualization/BTree.html`

- Trabalhos com erros de execução serão penalizados na nota.
- Evite o uso de variáveis globais. Isto pode implicar em penalização na nota - use a modularização de forma prudente.
- Organização do código e comentários valem nota. Trabalhos confusos e sem explicação sofrerão desconto na nota.
- Caso seja detectada **cópia** (entre alunos ou da Internet), todos os envolvidos receberão nota zero. Caso as pessoas envolvidas em suspeita de cópia discordem da nota, amplo direito de argumentação e defesa será concedido. Neste caso, as regras estabelecidas nas resoluções da UFES serão seguidas.
- A critério do professor, poderão ser realizadas entrevistas com os alunos, sobre o conteúdo do trabalho entregue. Caso algum aluno seja convocado para uma entrevista, a nota do trabalho será dependente do desempenho na entrevista. (Vide item sobre cópia, acima.)
- Cada implementação deve apresentar, **OBRIGATÓRIAMENTE, pelo menos 2 (DOIS) TADs opacos.** O uso de mais TADs é encorajado. Modele bem o problema para garantir uma modularização eficiente e livre de vazamentos de memória.

BOM TRABALHO!