

AEDII Arquivos



1 Pré-Árvore B

Estrutura interna de arquivos de dados

→ Armazenamento não volátil também conhecidos como memória secundária, são aqueles que em que o arquivo gravado não se perde quando "desligado". Alguns exemplos de memória secundária são: HD's externo, DVD, Pen-drives entre outros. São de capacidade maior, porém lentas e baratas.

→ Armazenamento volátil são aqueles que em que o que nele está se perde quando o computador é desligado. São os registradores, memória cache, e memória primária. Possuem capacidade menor, são rápidas e caras.

→ Seeking: posiciona a cabeça de L/E no local desejado; o conteúdo de todo um cilindro poder ser lido com apenas 1 seeking; movimento mais lento da operação de leitura/escrita; deve ser reduzido ao mínimo.

→ O acesso do seek é direto (aleatório ou randômico), pois não há a necessidade de ler todos os dados em sequência. - Por outro lado, fitas precisam de um acesso sequencial.

→ Paginação: capacidade de processar parte dos dados que estavam em memória secundária na memória principal (moldura de memória). Subconjunto do disco em memória principal.

→ Memória virtual

- paginação
- mapeamento de endereços
- reposição de páginas
 - LRU - Menos Recentemente Utilizada
 - LFU - Menos Frequentemente Utilizada
 - FIFO - Primeiro a Entrar Primeiro a Sair

→ Organização de arquivos na memória secundária

- Bloco: unidade de transferência de dados entre memória principal e a memória secundária
 - um bloco é formado por uma ou
 - É comum considerar que em cada bloco pode haver vários registros. Se R (tamanho fixo do registro, para simplificar) e B (tamanho do bloco) e $R \leq B$:
fator de blocagem $fb = \text{floor}(B/R) = \text{número de registros inteiros que cabem em um bloco mais páginas}$
- Organização espalhada: os blocos são totalmente preenchidos; se um registro não cabe inteiramente na parte vazia do bloco, coloca o que couber e um ponteiro para o próximo bloco
- Organização não espalhada: registros não podem ser divididos. Cada bloco pode conter até fb registros.

- Alocação de Blocos:

Blocos alocados sequencialmente

- Leitura fácil (leitura sequencial é ótima, e na leitura aleatória depende da facilidade de localização do deslocamento do registro dentro do arquivo)
- Expansão complicada: se não houver espaço disponível até o próximo arquivo tem que ser removido para outro local
- Fragmentação externa (buracos entre os arquivos): maior ou menor dependendo da política de alocação - o disco pode ficar fragmentado, isto é, com vários trechos disponíveis intercalados por trechos utilizados
- Métodos de ajuste sequencial
 - * Primeiro ajuste: seleciona o primeiro trecho encontrado (a partir do início da lista) grande o suficiente - *é o mais eficiente: balanço entre tempo de achar um bloco de tamanho suficiente (retorna assim que achar o primeiro) e fragmentação (não deixa sobrar sistematicamente o menor ou maior trecho)*
 - * Próximo ajuste: seleciona o próximo trecho grande o suficiente (a partir do índice “corrente”, ajustado após a última alocação) - *similar ao primeiro ajuste, mas chega mais rápido ao fim do heap*
 - * Melhor ajuste: seleciona o menor trecho dentre os trechos grandes o suficiente - *pode ser o pior: gasta tempo analisando tudo e, a menos que o ajuste seja perfeito, deixa sobrar normalmente trechos pequenos que não podem ser reutilizados*
 - * Pior ajuste: seleciona o maior trecho de todos - *tenta evitar esse desperdício, deixando sobrar trechos maiores que podem ser ainda utilizados, e assim posterga a criação de blocos pequenos*

Blocos alocados sequencialmente ordenado

- Leitura ordenada eficiente (sequencial) – $O(b)$: O próximo registro pode estar no mesmo bloco
- Mínimo / Máximo estão no cabeçalho do arquivo – $O(1)$: Isto podemos fazer para todos os tipos de alocação
- Busca: dá para usar busca binária (baseada nos blocos!)

- Inserção: cara! $O(b)$
 - * Tem que achar a posição certa : $O(\lg b)$
 - * Tem que abrir espaço para o registro (deslocar todos os registros com chave maior para frente) : $O(b)$
- Exclusão: cara pelos mesmos motivos! - $O(b)$
- Modificação: busca + atualização

Blocos alocados por lista ligada

Blocos com alocação indexada

- Índices primários
- Índice de clustering
- Índices secundários
- Organização indexada multiníveis

2 Árvore B

→ Com inspiração das árvores binárias de busca e com o dinamismo dos índices multiníveis cria-se a Árvore B.

→ Definição:

- 1. Cada nó x contém os seguintes campos
 - $n[x]$, o número de chaves atualmente armazenadas no nó x ;
 - as $n[x]$ chaves, armazenadas em ordem não decrescente, de modo que $key_1[x] \leq key_2[x] \leq \dots \leq key_n[x]$;
 - $leaf[x]$, um valor booleano indicando se x é um folha (*true*) ou um nó interno (*false*);
 - se x é um nó interno, x contém $n[x]+1$ ponteiros $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$ para seus filhos.

- [2.] As chaves $key_i[x]$ separam as faixas de valores armazenado em cada subárvore: denotando por key_i uma chave qualquer armazenada na subárvore com nó $c_i[x]$, tem-se:

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \dots \leq key_{n[x]}[x] \leq k_{n[x]+1}$$

- [3.] Todas as folhas aparecem no mesmo nível, que é a altura da árvore, h .
- [4.] Há um limite inferior e superior no número de chaves que um nó pode conter, expressos em termos de um inteiro fixo $t \geq 2$ chamado o *grau mínimo* (ou *ordem*) da árvore.
 - Todo nó que não seja a raiz deve conter pelo menos $t - 1$ chaves.
 - Todo nó interno que não seja a raiz deve conter pelo menos t filhos.
 - Todo nó deve conter no máximo $2t - 1$ chaves (e portanto todo nó interno deve ter no máximo $2t$ filhos). Dizemos que um nó está *cheio* se ele contiver exatamente $2t - 1$ chaves.
- [Altura.] Teorema: Para toda árvore B de grau mínimo $t \geq 2$ contendo n chaves, sua altura h máxima será:

$$h \leq \log_t \frac{(n+1)}{2}$$

↓

$$n \geq 1 + (t - 1) * \sum_{i=1}^h 2t^{i-1}$$

$$n \geq 1 + 2(t - 1) * (1 + t + t^2 + \dots + t^{h-1})$$

$$n \geq 1 + 2(t - 1) * \left(\frac{t^h - 1}{t - 1} \right)$$

$$n \geq 1 + 2(t^h - 1)$$

$$n \geq 2t^h - 1$$

$$\frac{(n+1)}{2} \geq t^h$$

$$t^h \leq \frac{(n+1)}{2}$$

Aplicando log dos dois lados:

$$h \leq \log_t \frac{(n+1)}{2}$$

A altura de **pior caso** de uma árvore B é quando em todos os seus nós há um **número máximo de chaves**. Ela estaria na **altura máxima**.

A altura de **melhor caso** de uma árvore B é quando em todos os seus nós há um **número mínimo de chaves**. Ela estaria na **altura mínima**.

→ Criação de uma árvore vazia

```
1: BTreeCreate(T)
2:  $i \leftarrow AllocateNode()$ 
3:  $leaf[x] \leftarrow true$ 
4:  $n[x] \leftarrow 0$ 
5:  $DiskWrite(x)$ 
6:  $root[T] \leftarrow x$ 
```

→ *B-Tree-Search* toma como entrada um ponteiro para o nó de raiz x de uma subárvore e uma chave k que deve ser procurada nessa subárvore. Se k está na B-árvore, *B-Tree-Search* retorna o par ordenado (y, i) , que consiste em um nó y e um índice i tal que $key_i[y] = k$. Caso contrário, o procedimento retorna *NIL*. Assim, a chamada de nível superior é da forma *B-Tree-Search*($root[T]$, k).

```
1: BTreeSearch(x, k)
2:  $i = 1$ 
3: while  $i \leq n[x]$  e  $k > key_i[x]$  do
4:    $i = i + 1$ 
5: end while
6: if  $i \leq n[x]$  e  $k = key_i[x]$  then
7:   return  $(x, i)$ 
8: else if  $leaf[x]$  then
9:   return NIL
```

```

10: else
11:   DiskRead( $c_i[x]$ )
12: end if
13: return BTreeSearch( $x.c_i, k$ )

```

→ Inserção ocorrem sempre nas folhas

- Caso o nó em que a folha será inserida estiver cheia, alguma das chaves deverá ser promovida e esse nó se subdividirá.
 - Durante a busca da localização de inserção do nó, no caminho da raiz até uma folha, se achar um nó filho (a ser seguido) cheio, já o subdivide.
 - Vamos assumir que o nó atual (pai do nó cheio) não é cheio, a menos da raiz que deve ser tratada separadamente
-

→ Divisão de um nó na árvore: *BTreeSplitChild*(x, i, y): tem como entrada um nó interno x não cheio, um índice i e um nó y tal que $y = c_i[x]$ é um filho cheio de x . O procedimento divide y em 2 e ajusta x de forma que este terá um filho adicional.

```

1: BTreeSplitChild( $x, i, y$ )
2:  $z \leftarrow \text{AllocateNode}()$ 
3:  $\text{leaf}[z] \leftarrow \text{leaf}[y]$ 
4:  $n[z] \leftarrow t - 1$ 
5: for  $j \leftarrow 1$  to  $t - 1$  do
6:    $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$ 
7: end for
8: if not  $\text{leaf}[y]$  then
9:   for  $j \leftarrow 1$  to  $t$  do
10:     $c_j[z] \leftarrow c_{j+t}[y]$ 
11:   end for
12: end if
13:  $n[y] \leftarrow t - 1$ 
14: for  $j \leftarrow n[x]$  downto  $i$  do
15:    $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$ 
16: end for
17:  $\text{key}_i[x] \leftarrow \text{key}_t[y]$ 

```

```

18:  $n[x] \leftarrow n[x] + 1$ 
19: DiskWrite( $y$ )
20: DiskWrite( $z$ )
21: DiskWrite( $x$ )

```

→→→ Linhas 2-10 Aloca e inicializa um nó para ser filho da chave que será promovida.

→→→ Linha 13 Ajusta o nó que foi dividido.

→→→ Linhas 14-18 Ajusta o nó em que foi parar o nó que saiu do nó dividido.

→ Inserção de uma chave na árvore com raiz T:

```

1: BTreeInsert(T, k)
2:  $r \leftarrow \text{root}[T]$ 
3: if  $n[r] = 2t - 1$  then
4:    $s \leftarrow \text{AllocateNode}()$ 
5:    $\text{root}[T] \leftarrow s$ 
6:    $\text{leaf}[s] \leftarrow \text{false}$ 
7:    $n[s] \leftarrow 0$ 
8:    $c_1 \leftarrow r$ 
9:   BTreeSplitChild( $s, 1, r$ )
10:  BTreeInsertNonFull( $s, k$ )
11: else
12:  BTreeInsertNonFull( $r, k$ )
13: end if

```

→ Inserção de uma chave em uma subárvore cuja raiz x não está cheia:

```

1: BTreeInsertNonFull(x, k)
2:  $i \leftarrow n[x]$ 
3: if  $\text{leaf}[x]$  then
4:   while  $i \geq 1$  and  $k < \text{key}_i[x]$  do
5:      $\text{key}_{i+1}[x] \leftarrow \text{key}_i[x]$ 
6:      $i \leftarrow i - 1$ 
7:   end while
8:    $\text{key}_{i+1}[x] \leftarrow k$ 
9:    $n[x] \leftarrow n[x] + 1$ 

```



```

10:   DiskWrite( $x$ )
11: else
12:   while  $i \geq 1$  and  $k < key_i[x]$  do
13:      $i \leftarrow i - 1$ 
14:   end while
15:    $i \leftarrow i + 1$ 
16:   DiskRead( $c + i[x]$ )
17:   if  $n[c_i[x]] = 2t - 1$  then
18:     BTreeSplitChild( $x, i, c_i[x]$ )
19:     if  $k > key_i[x]$  then
20:        $i \leftarrow i + 1$ 
21:     end if
22:   end if
23:   BTreeInsertNonFull( $c_i[x], k$ )
24: end if

```

→ Remoção em árvores B - *BTreeDelete*(x, k): remoção da chave k da subárvore com raiz x .

- 2. - Se a chave k está no nó x e x é um nó interno, faça:
 - a) Se o filho y que precede k no nó x tem pelo menos t chaves, então encontre o predecessor de k de k na subárvore com raiz y . Delete recursivamente k , e substitua k por k' em x .
 - b) Simetricamente, se o filho z imediatamente após k no nó x tem pelo menos t chaves, então encontre o sucessor k' de k na subárvore com raiz z . delete recursivamente k , e substitua k por k' em x .
 - c) Caso contrário, se ambos y e z possuem apenas $t - 1$ chaves, faça a junção de k e todas as chaves de z em y , de forma que x perde tanto a chave k como o ponteiro para z , e y agora contém $2t - 1$ chaves. Então, libere z e delete recursivamente k e y .
- 3. - Se a chave k não está presente no nó interno x , determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k (se k estiver presente na árvore). Se $c_i[x]$ tem apenas $t - 1$ chaves, execute o passo 3a ou 3b conforme necessário para garantir que o algoritmo desça para um nó contendo pelo menos t chaves. Então, continue no filho apropriado de x .

- a) Se $c_i[x]$ contém apenas $t-1$ chaves mas tem um irmão imediato com pelo menos t chaves, dê para $c_i[x]$ uma chave extra movendo uma chave de x para $c_i[x]$, movendo uma chave do irmão imediato de $c_i[x]$ à esquerda ou à direita, e movendo o ponteiro do filho apropriado do irmão para o nó $c_i[x]$.
- b) Se $c_i[x]$ e ambos os irmãos imediatos de $c_i[x]$ contêm $t-1$ chaves, faça a junção de $c_i[x]$ com um de seus irmãos. Isso implicará em mover uma chave de x para o novo nó fundido (que se tornará a chave mediana para aquele nó).
- 1. - Se a chave k está no nó x e x é uma folha, exclua a chave k de x . - (pelos procedimentos anteriores, já sabemos que o nó x tem pelo menos t chaves).
- *** Quando um dos nós for a raiz
 - Ela pode ter menos do que $t-1$ chaves
 - Se ficar com zero chaves precisa desalocar o bloco e atualizar quem é a nova raiz.
 - Precisa de uma camada extra sobre a chamada da deleção:
 - 1: **BTreeDeleteFromRoot**(**T**, **k**)
 - 2: $r \leftarrow \text{raiz}[T]$
 - 3: **if** $n[r] = 0$ **then**
 - 4: **return**
 - 5: **else**
 - 6: $\text{BTreeDelete}(r, k)$
 - 7: **end if**
 - 8: **if** $n[r] = 0$ **and not** $\text{leaf}[r]$ **then**
 - 9: $\text{raiz}[T] \leftarrow c1[r]$
 - 10: $\text{desaloca}(r)$
 - 11: **end if**

2.1 Hashing (Espalhamento): Tenenbaum cap 7.4

- Considere uma conjunto de chaves k e seja I conjunto de índices.
 $\rightarrow I = \{0, 1, 2, \dots, I-1\}$
- Uma função hash, denotada por $h(k)$
 $h : k \rightarrow I$ para cada chave associa-se um único índice. A idéia é que o custo de uma busca seja $O(1)$.

Table 1: Tabela com índices

0
1
2
.
.
.
$I - 1$

– Ex: $h(k) = k \% T$, onde T é o tamanho do conjunto de índices.

$$Colisao = \begin{cases} h(53) = 3 \\ h(13) = 3 \end{cases} \quad (1)$$

*colisão é o fato de duas chaves distintas k_1 e k_2 serem associadas ao mesmo índice.

- Tratamento de Colisões

→ Endereçamento aberto: consiste em uma função chamada de *rehash* – $rh(k)$

→ $rh : I \rightarrow I$

→ Essa função é responsável por “realocar” a chave na qual houve uma colisão.

*há um limite no tamanho da lista e o tempo de busca.

→ Dadas funções $h(k)$ e $rh(i)$

```

1: bool search (int k)
2:   int p = h(k)
3:   while ( $T[p]$  is not k) and ( $T[p]$  is not -1)) do
4:     p = r/h(k)
5:   end while
6:   if  $T[p] == -1$  then
7:     return false
8:   else
9:     return true
10: end if
```

o laço entrará num loop infinito caso a tabela esteja totalmente cheia. Uma alternativa é definir como cheia a *tabela* – 1 para garantir

que a busca termine.

```
1: bool insert (int k)
2: int p = h(k)
3: while (T[p] is not k) and (T[p] is not -1)) do
4:   p = rh(p)
5: end while
6: if T[p] == -1 then
7:   T[p] = k
8:   return true
9: else
10:  return false
11: end if
```

```
1: bool remove (int k)
2: int p = h(k)
3: while (T[p] is not k) and (T[p] is not -1)) do
4:   p = rh(p)
5: end while
6: if T[p] == k then
7:   T[p] = ? //o que fazer?
8: end if
```

Atualizar as funções de busca e inserção considerando o marcador de removido.

→ a função de busca não muda

→ a função de inserção precisa considerar o marcador de removido.

↓ insert modificado.

```
1: bool insert (int k)
2: int p = h(k)
3: int temp = -1
4: while (T[p] is not k) and (T[p] is not -1)) do
5:   if T[p] == -2 then
6:     temp = p
7:   end if
8:   p = rh(p)
9: end while
10: if T[p] == k then
```

```

11:   return false
12: else
13:   if temp is not -1 then
14:      $T[temp] = k$ 
15:   else
16:      $T[p] = k$ 
17:     return true
18:   end if
19: end if

```

– Eficiência dos métodos de *rehash*

→ Uma medida de eficiência é contar o número de posições examinadas (probabilidade) antes de encontrar uma chave.

→ O número médio (valores altos de T) de consultas para $\frac{2T-n+1}{2T-2n+2}$ onde n é o número de chaves presentes na tabela.

→ Define-se por *lead factor* $(lf)\frac{n}{T}$ (fração da tabela que está preenchida)

→ Inicialmente todas as posições da tabela tem a mesma probabilidade de serem preenchidas. À medida que a tabela vai enchendo, algumas posições tem um aumento nessa probabilidade.

→ Uma forma de eliminar a aglomeração primária é fazer com que a função $rh()$ dependa de dois argumentos $rh(i, j)$ onde i é o número de vezes que a função $rh()$ for aplicada.

$$rh(i, j) = (i + j) \% T.$$

$$rh(i, 1) = (i + 1) \% T$$

$$rh(i, 2) = (i + 2) \% T$$

$$rh(i, 3) = (i + 3) \% T$$

→ Uma variação dessa técnica é usar uma permutação aleatória de 0 até $T - 1$.

$$(p_0, p_1, p_2, \dots, p_{t+1}) \text{ e define o } rh(j) \text{ como } (h(r) + pj) \% T$$

→ Uma terceira abordagem é $rh(j) = (h(r) + j^2) \% T$.

→ Uma quarta opção é $rh(i, k) = (i + h_{key}) \% T$.

$$\hookrightarrow h_{key} = (1 + h(k)) \% T.$$

– Aglomeração Secundária

→ chaves com o mesmo valor de *hash*, isto é, $k_1 \neq^* k_2$ e $h(k_1) =$

$h(k_2)$ tendem a seguir o mesmo *rehash* até serem inseridas.
→ uma forma de eliminar a aglomeração secundária é chamada de duplo hash e usa duas funções *hash* h_1 e h_2 usada na fase inicial, isto é, para determinar a posição da chave na tabela. Se a posição estiver ocupada o *rehash* fica dessa forma:

$$rh(i, k) = (1 + h_2(k)) \% T$$

2.2 Tabela hash ordenada

A idéia geral é manter as chaves que colidirem na mesma posição em ordem decrescente.

```
1: int insert (int key)
2: int  $i, j, tempKey, nKey$ 
3: bool  $first$ 
4:  $i = h(key)$ 
5:  $nKey = key$ 
6:  $first = true$ 
7: while  $Table[i] > nKey$  do
8:    $i = rh(i)$ 
9: end while
10: int  $tk = Table[i]$ 
11: while  $((tk \text{ is not } -1) \text{ and } (tk \text{ is not } nKey))$  do
12:    $tempKey = Table[i]$ 
13:   if  $tempKey < nKey$  then
14:      $Table[i] = nKey$ 
15:      $nKey = tempKey$ 
16:     if  $first$  then
17:        $j = i$ 
18:        $first = false$ 
19:     end if
20:   end if
21:    $i = rh(i)$ 
22:    $tk = Table[i]$ 
23: end while
24: if  $tk == -1$  then
25:    $Table[i] = nKey$ 
26: end if
27: if  $first$  then
28:   return  $i$ 
29: else
30:   return  $j$ 
31: end if
```

O que permite que as chaves fiquem ordenadas e que a busca seja bem-sucedida é o *if* (13 – 20). Com ele as chaves só serão movidas se o elemento a ser inserido for maior que a chave em que está localizado o seu hash.

2.3 Novos métodos para a função *hash*

A função hash que estudamos até agora foi a do método da divisão:

$$h(chave) = chave \% T, T \text{ é o tamanho da Tabela Hash.}$$

Observação: chave é um número inteiro(int)

O que torna a função eficiente? R: Minimizar o número de colisões.

Para os novos métodos e pensando na eficiência deles considere os seguintes critérios:

- Critério 1) Levar em consideração toda a informação contida na chave (não só o último número que era considerado no método da colisão)
- Critério 2) Sensibilidade a permutação (Os números 53 e 33 cairiam no mesmo índice da tabela pelo hash, mesmo sendo números diferentes, para os métodos seguintes não é isso que se deseja).
- Critério 3) Trabalhar com chaves alfa-numéricas.

Métodos:

- 1 - O método da divisão consegue melhorar resultados quando TableSize é primo.
- 2 - Método multiplicativo $h(chave)$ é definido como

$$\text{floor}(m * \text{frac}(c * \text{chave})),$$

onde c é um número real entre 0 e 1, e m é o tamanho da Tabela. Valores de c com boas propriedades:

$$\begin{aligned} c_1 &= 0,6180339887, \\ c_2 &= 0,3819660113. \end{aligned}$$

Este método usa toda a informação da chave e é sensível a permutações.

- 3 - Método do quadrado médio (para TableSize = potência de dez)
O valor da chave é elevada ao quadrado e alguns dígitos do “meio” são usados como índice.

Exemplo:

$$T = 1000 \text{ (índices de 0 a 999)}$$

$$chave = 245$$

$$245^2 = 60025 \rightarrow 6 \text{ } \boxed{002} 5 \rightarrow h(245) = 002.$$

Este método usa toda a informação da chave e é sensível a permutações.

- 4 - Método da dobra
Suponha uma representação binária de uma chave que segue:

$$010111001010110.$$

Dividindo este número em três partes:

$$\underbrace{01011} \quad \underbrace{10010} \quad \underbrace{10110}$$

E empilando para fazer operações com XOR:

$$\begin{array}{r} 01011 \\ 10010 \\ \underline{10110} \\ 01111 \\ \hookrightarrow 15_{10} \end{array}$$

Índice será 15.

Este método usa toda a informação da chave e é sensível a permutações.

- 5 - Chaves alfa-numéricas
Strings; char[50];

chave = “hello”

Podemos interpretar uma string na base 26(alfabeto)

$$\underbrace{h}_{8 \cdot 26^4 +} \underbrace{e}_{5 \cdot 26^3 +} \underbrace{l}_{12 \cdot 26^2 +} \underbrace{l}_{12 \cdot 26^1 +} \underbrace{o}_{15 \cdot 26^0} = 3.752.127$$

Achando o resultado pode-se usar qualquer um dos métodos anteriores, pois com essa etapa já é o suficiente para garantir a eficiência. Este método contempla os três critérios.

```

1: int Quadrados (int c)
2: int cs = c * c
3: if ((cs % 1000) == cs) then
4:     return cs
5: end if
6: int nd = 0
7: int ctemp = cs
8: while ctemp > 0 do
9:     nd ++
10:    ctemp / = 10
11: end while
12: int dt = log(1000) //3
13: int r = nd - dt //saber quantos dígitos retirar
14: r / = 2 //funciona para par e ímpar
15: ctemp = cs
16: for int i = 0; i < r; i ++ do
17:     ctemp / = 10
18: end for
19: return ctemp % 1000 //Pega os três dígitos do final (60025 - 5 já saiu)

```

3 Notas

- O que está de **vermelho com asterisco*** precisa ver se está correto.
- O que está somente de **vermelho** é para incluir no arquivo.