Universidade Federal de São Carlos
Bacharelado em Ciência da Computação
Organização e Recuperação da Informação
Prof. Tiago A. Almeida <talmeida@ufscar.br>

Trabalho 02
Árvore-B

Prazo para entrega: 13/01/2025 – 23:59

Atenção

- **E/S:** tanto a entrada quanto a saída de dados devem ser de acordo com os casos de testes abertos;
- **Identificadores de variáveis:** escolha nomes apropriados;
- **Documentação:** inclua comentários e indente corretamente o programa;
- **Erros de compilação:** nota **zero** no trabalho;
- **Tentativa de fraude:** nota **zero na média** para todos os envolvidos. Fraudes, como tentativas de compras de soluções ou cópias de parte ou de todo código-fonte, de qualquer origem, implicará na reprovação direta na disciplina. Partes do código cujas **ideias** foram desenvolvidas em colaboração com outro(s) aluno(s) devem ser devidamente documentadas em comentários no referido trecho. Contudo, isso **NÃO** autoriza a cópia de trechos de código, a codificação em conjunto, compra de soluções, ou compartilhamento de tela para resolução do trabalho. Em resumo, você pode compartilhar ideias em alto nível, modos de resolver o problema, mas não o código;
- Utilize o código-base e as mensagens pré-definidas (`#define`).

1 Contexto

Assim como esperado, o **UFSCraft** ganhou uma tremenda popularidade e se tornou um grande sucesso com o passar do tempo. Agora, milhares de pessoas estão querendo jogá-lo e muitos novos *kits* estão sendo criados para atender à demanda dos jogadores. Consequentemente, a quantidade de jogadores, *kits*, partidas e resultados vem crescendo de maneira astronômica.

Acompanhando métricas de qualidade do seu sistema, como quantidade de requisições e tempo de resposta das operações, além das métricas do banco de dados, você percebeu que algumas operações começaram a ficar lentas. Cadastros de jogadores, *kits* e partidas são rápidos, mas a manutenção dos índices tem ficado demasiadamente lenta; buscas por *kits* demoram; listagens estão demandando muito tempo, entre outros problemas de lentidão do sistema. Após fazer uma profunda análise da situação, você percebeu que o gargalo deve-se ao fato de índices simples (listas ordenadas) não serem mais adequados, pois não cabem mais na memória RAM, demandando muitos acessos ao disco para realizar as operações.

Felizmente, nem tudo está perdido! Com as habilidades adquiridas nas aulas de ORI, você rapidamente chegou à conclusão que a estrutura mais adequada para se utilizar no armazenamento de índices de grande volume de dados é a *Árvore-B*. Portanto, como você ainda é o único programador entre os seus amigos que estão gerenciando esse gigantesco projeto, cabe a você atualizar o sistema para fazer uso de *Árvores-B* ao invés de índices simples.

2 Base de dados da aplicação

O sistema será composto por dados dos jogadores, dos *kits*, das partidas e dos resultados, conforme descrito a seguir.

2.1 Dados dos jogadores

- **id_jogador**: identificador único de um jogador (chave primária), composto por 11 dígitos. Não poderá existir outro valor idêntico na base de dados. Ex: `57956238064`;
- **apelido**: apelido do jogador. Ex: `Badast`;
- **cadastro**: data em que o usuário realizou o cadastro no sistema, no formato `<AAAA><MM><DD><HH><MM>`. Ex: `202411191020`;
- **premio**: última data em que o jogador recebeu uma premiação, no formato `<AAAA><MM><DD><HH><MM>`. Ex: `202411191920`;
- **saldo**: saldo que o jogador possui na sua conta, no formato `<9999999999>.<99>`. Ex: `0000004605.10`;
- **kits**: até 10 *kits* adquiridos pelo jogador, sendo que cada *kit* é identificado por 3 dígitos. Ex: `001002005004012003010007008009`;

2.2 Dados dos *kits*

- **id_kit**: identificador único de cada *kit*, composto por 3 dígitos, no formato `<999>`. Não poderá existir outro valor idêntico na base de dados. Ex: `524`;
- **nome**: nome do *kit*. Ex: `Kangaroo`;
- **poder**: poder do *kit*. Ex: `Causa dano ao cair sobre o inimigo`;
- **preco**: preço do *kit* no formato `<99999999999>.<99>`. Ex: `00000006000.00`;

2.3 Dados das partidas

- **id_partida**: identificador único da partida, composto por 8 dígitos, no formato `<99999999>`. Não poderá existir outro valor idêntico na base de dados. Ex: `12345678`;
- **inicio**: início da partida, no formato `<AAAA><MM><DD><HH><MM>`. Ex: `202411191020`
- **duracao**: tempo de duração da partida, no formato `<HH><MM><SS>`. Ex: `012044`;
- **cenario**: identificador de cenário da partida com 4 dígitos, no formato `<9999>`. Ex: `0136`;
- **id_jogadores**: lista com até 12 jogadores que participam da partida. Ex: `44679595970427143763036503752160593804621236576539935249100462199991004621995710046219955100462199511004621995010046219950100462199501004621000`;

2.4 Dados de resultados

- **id_jogador**: ID do jogador;
- **id_partida**: ID da partida;
- **id_kit**: ID do *kit* que o jogador utilizou na partida;
- **colocacao**: posição do jogador na partida;
- **sobrevivencia**: tempo de sobrevivência do jogador na partida, no formato `<HH><MM><SS>`. Ex: `000512`;
- **eliminacoes**: quantidade de eliminações do jogador na partida;

Garantidamente, nenhum campo de texto receberá caracteres acentuados.

2.5 Criação das bases de dados em SQL

Cada base de dados corresponde a um arquivo distinto. Elas poderiam ser criadas em um banco de dados relacional usando os seguintes comandos SQL:

```
CREATE TABLE jogadores (
  id_jogador  varchar(11) NOT NULL PRIMARY KEY,
  apelido     text NOT NULL,
  cadastro    varchar(12) NOT NULL,
  premio      varchar(12) NOT NULL DEFAULT 0,
  saldo       numeric(12, 2) NOT NULL DEFAULT 0,
  kits        varchar(30) NOT NULL DEFAULT '{}'
);
CREATE TABLE partidas (
  id_partida    varchar(8) NOT NULL PRIMARY KEY,
  inicio        varchar(12) NOT NULL,
  duracao       varchar(6) NOT NULL,
  cenario       varchar(4) NOT NULL,
  id_jogadores  varchar(132) NOT NULL
);
CREATE TABLE kits (
  id_kit  varchar(3) NOT NULL PRIMARY KEY,
  nome    text NOT NULL,
  poder   text NOT NULL,
  preco   numeric(12,2) NOT NULL
);
CREATE TABLE resultados (
  id_jogador     varchar(11) NOT NULL,
  id_partida     varchar(8) NOT NULL,
  id_kit         varchar(3) NOT NULL,
  colocacao      numeric(2, 0) NOT NULL,
  sobrevivencia  varchar(6) NOT NULL,
  eliminacoes    numeric(2, 0) NOT NULL,
);
```

3 Operações suportadas pelo programa

Os dados devem ser manipulados através do console/terminal (modo texto) usando uma sintaxe similar à SQL, sendo que as operações a seguir devem ser fornecidas.

3.1 Cadastro de jogadores

```
INSERT INTO jogadores VALUES ('<id_jogador>', '<apelido>');
```

Para criar uma nova conta de jogador, seu programa deve ler os campos `id_jogador` e `apelido`. Inicialmente, a conta será criada sem saldo (`00000000000.00`). O campo `cadastro` receberá a data em que o cadastro foi realizado. A função deve falhar caso haja a tentativa de inserir um jogador com um `id_jogador` já cadastrado, ou seja, um CPF que já esteja no sistema. Neste caso, deverá

ser apresentada a mensagem de erro padrão `ERRO_PK_REPETIDA`. Caso a operação se concretize com sucesso, exibir a mensagem padrão `SUCESSO`.

3.2 Remoção de jogadores

```
DELETE FROM jogadores WHERE id_jogador = '<id_jogador>';
```

O usuário deverá ser capaz de remover uma conta dado um ID de um jogador. Caso a conta não exista, seu programa deverá exibir a mensagem padrão `ERRO_REGISTRO_NAO_ENCONTRADO`. A remoção na base de dados deverá ser feita por meio de um marcador, conforme descrito na Seção 5. Se a operação for realizada, exibir a mensagem padrão `SUCESSO`.

3.3 Adicionar saldo na conta

```
UPDATE jogadores SET saldo = saldo + '<valor>' WHERE id_jogador = '<id_jogador>';
```

O usuário deverá ser capaz de adicionar valor na conta de um jogador dado seu ID e o valor desejado. Caso o jogador não esteja cadastrado no sistema, o programa deverá imprimir a mensagem padrão `ERRO_REGISTRO_NAO_ENCONTRADO`. Caso o valor que esteja sendo adicionado seja menor ou igual a zero, o programa deve imprimir a mensagem `ERRO_VALOR_INVALIDO`. Se não houver nenhum desses problemas, o saldo deverá ser atualizado, seguido da impressão da mensagem padrão de `SUCESSO`.

3.4 Comprar um *kit*

```
UPDATE jogadores SET kits = array_append(kits, '<kits>') WHERE id_jogador =
'<id_jogador>';
```

O usuário poderá comprar um *kit* para um jogador dado o ID do jogador e o ID do *kit* desejado, caso possua saldo para comprá-lo. Neste caso, o saldo será descontado da conta do jogador baseado no valor do *kit*. Caso o jogador não possua saldo, a mensagem padrão `ERRO_SALDO_NAO_SUFICIENTE` deverá ser impressa. Caso o jogador ou o *kit* não exista, o programa deverá imprimir `ERRO_REGISTRO_NAO_ENCONTRADO` e, caso o *kit* desejado já esteja presente nos *kits* do jogador, seu programa deverá imprimir `ERRO_KIT_REPETIDO`. Existe um máximo de dez *kits* por jogador e, garantidamente, não haverá nenhuma tentativa de inserir mais de dez por jogador. Caso não haja nenhum erro, o programa deve atribuir o *kit* ao jogador, atualizando todos os índices e arquivos necessários e, então, imprimir a mensagem padrão `SUCESSO`.

3.5 Cadastro de *kits*

```
INSERT INTO kits VALUES ('<nome>', '<poder>', '<preco>');
```

Para um *kit* ser adicionado no banco de dados, seu programa deverá ler os campos `nome`, `poder` e `preco`. O campo `id_kit` segue a ordem de cadastro dos *kits*. Caso a operação se concretize, exiba a mensagem padrão `SUCESSO`.

3.6 Executar partida

```
INSERT INTO partida VALUES ('<inicio>', '<duracao>', '<cenario>', '<jogadores>',
'<kits_jogadores>', '<duracoes_jogadores>', '<eliminacoes>');
```

Para executar uma partida, seu programa deve ler os campos `inicio`, `duracao`, `cenario` e `jogadores`. O campo `id_partida` deverá ser preenchido de acordo com a quantidade de partidas cadastradas no sistema, ou seja, é um valor incremental. Os demais campos informados (`kits_jogadores`, `duracoes_jogadores` e `eliminacoes`) serão utilizados para compor o arquivo `resultados`. A ordem dos ID de jogadores representa a própria colocação deles na partida.

No caso do ID de algum jogador não existir, ou ter sido deletado, exibir a mensagem `ERRO_REGISTRO_NAO_ENCONTRADO`. Caso o jogador não possua o *kit* atribuído a ele na partida, exibir `ERRO_JOGADOR_KIT`. Em qualquer uma dessas situações, a operação de inserção deve ser interrompida. Se a operação for concretizada com sucesso, exibir a mensagem padrão `SUCESSO`.

3.7 Recompensar vencedor

```
RECOMPENSAR_VENCEDOR('<data_inicio>', '<data_fim>'), '<premio>');
```

Ao executar a função, o jogador com o melhor desempenho entre `data_inicio` e `data_fim` deverá ser recompensado com o valor `premio` em seu saldo. O campo `premio`, em `jogadores`, deve armazenar o final do período da premiação mais recente concedida ao jogador. Quando a operação se concretizar com sucesso, exibir a mensagem padrão `CONCEDER_PREMIO`.

Note que registros de jogadores podem ser removidos nesse sistema. Na situação em que o jogador com melhor desempenho no período tiver sido excluído do sistema, exibir a mensagem padrão `ERRO_JOGADOR_REMOVIDO` e encontrar o próximo jogador com melhor desempenho para conceder o prêmio.

Para descobrir qual jogador obteve o melhor desempenho no período, considere as seguintes regras:

(a) O jogador com mais vitórias é aquele que obteve o melhor desempenho;

(b) No caso de dois ou mais jogadores conseguirem obter o mesmo número de vitórias, o de melhor desempenho é aquele que obtiver o maior número de eliminações;

(c) No caso de dois ou mais jogadores com o mesmo número de vitórias e eliminações, o que sobreviveu por mais tempo nessas partidas é aquele com o melhor desempenho;

(d) Por fim, caso dois ou mais jogadores tenham o mesmo número de vitórias, eliminações e o mesmo tempo de sobrevivência, a escolha daquele com melhor desempenho será decidida pelo menor valor `id_jogador`.

3.8 Busca

As seguintes operações de busca deverão ser implementadas. *Em todas elas, será necessário utilizar a busca binária e mostrar o caminho percorrido nos índices.* Antes dos resultados das buscas, deverá ser impresso os RRNs dos nós percorridos, assim como o percurso feito dentro de cada nó.

Exemplo:

```
Registros percorridos:   2 (1 0) 0 (1 0)
```

No exemplo acima, a árvore tem ordem $m = 3$. Fora dos parênteses estão os RRNs dos nós da árvore que foram percorridos e, dentro dos parênteses, o percurso de índices dentro de cada nó, que no caso poderia ser 0, 1 ou 2.

ATENÇÃO: caso o número de elementos dentro do nó seja par (p.ex, 10 elementos), então há 2 (duas) possibilidades para a posição da mediana dos elementos (p.ex., 5a ou 6a posição se o total fosse 10). Neste caso, **sempre** escolha a posição mais à direita (p.ex., a posição 6 caso o total for 10).

3.8.1 Jogador

O usuário deverá poder buscar contas de jogadores pelos seguintes atributos:

(a) ID do jogador:

```
SELECT * FROM jogadores WHERE id_jogador = '<id_jogador>';
```

Solicitar ao usuário o ID de cadastro do jogador. Caso a conta não exista, seu programa deverá exibir a mensagem padrão `ERRO_REGISTRO_NAO_ENCONTRADO`. Caso o cadastro exista, todos os seus dados deverão ser impressos na tela de forma formatada.

3.8.2 *Kit*

O usuário deverá ser capaz de buscar *kits* pelos seguintes atributos:

(a) ID do *kit*:

```
SELECT * FROM kits WHERE id_kit = '<id_kit>';
```

Solicitar ao usuário o ID do *kit*. Caso não exista no banco de dados, seu programa deverá exibir a mensagem padrão `ERRO_REGISTRO_NAO_ENCONTRADO`. Caso contrário, todos os dados do *kit* deverão ser impressos na tela de forma formatada.

3.8.3 Partida

O usuário deverá ser capaz de buscar partidas pelos seguintes atributos:

(a) ID da partida:

```
SELECT * FROM partidas WHERE id_partida = '<id_partida>';
```

Solicitar ao usuário o ID da partida. Caso não exista no banco de dados, seu programa deverá exibir a mensagem padrão `ERRO_REGISTRO_NAO_ENCONTRADO`. Caso contrário, todos os dados da partida deverão ser impressos na tela de forma formatada.

3.9 Listagem

As seguintes operações de listagem deverão ser implementadas.

3.9.1 Jogadores

(a) Pelos IDs dos jogadores:

```
SELECT * FROM jogadores ORDER BY id_jogador ASC;
```

Exibe todos os jogadores ordenados de forma crescente pelo ID. Caso nenhum registro seja retornado, seu programa deverá exibir a mensagem padrão `AVISO_NENHUM_REGISTRO_ENCONTRADO`.

(b) Por nome de *kit*:

```
SELECT * FROM jogadores WHERE nome_kit IN array_append(kits) ORDER BY
id_jogador ASC
```

Exibe todos os jogadores que possuem determinado *kit*, em ordem crescente de ID. Caso nenhum registro for retornado, seu programa deverá exibir a mensagem padrão `AVISO_NENHUM_REGISTRO_ENCONTRADO`.

ATENÇÃO: antes da listagem dos jogadores, o seu programa deverá imprimir os registros do índice da lista invertida (*i.e.*, `jogador_kits_primario_idx`, ver detalhes na Seção 6.2) que foram percorridos na listagem (obs: a busca no índice secundário, contendo os nomes dos *kits*, deverá ser feita por uma busca binária).

Exemplo:

```
Registros percorridos:   3 5 6
```

No exemplo acima, os valores representam o número do índice dos registros que foram percorridos durante a busca até encontrar o registro de interesse ou esgotar as possibilidades no índice secundário.

Atenção: caso o número de elementos seja par (p.ex, 10 elementos), então há 2 (duas) possibilidades para a posição da mediana dos elementos (p.ex., 5^a ou 6^a posição se o total fosse 10). Neste caso, **SEMPRE** escolha a posição mais à direita (p.ex., a posição 6 caso o total for 10).

3.9.2 *Kits* para comprar

(a) De acordo com o saldo do jogador:

```
SELECT * FROM kits WHERE preco <= ('SELECT saldo FROM jogador WHERE id_jogador =
<id_jogador> ');
```

Seu programa deve ler o ID de um jogador e, em seguida, exibir todos os *kits* que o jogador pode comprar, de acordo com o seu saldo. Na situação em que o jogador foi removido, imprimir a mensagem `ERRO_REGISTRO_NAO_ENCONTRADO`. Caso nenhum registro for retornado, seu programa deverá exibir a mensagem padrão `AVISO_NENHUM_REGISTRO_ENCONTRADO`.

3.9.3 Partida

(a) Por data de partida:

```
SELECT * FROM partida WHERE inicio BETWEEN '<data_inicio>' AND '<data_fim>' ORDER
BY inicio ASC;
```

Exibe todas as partidas realizadas em um determinado intervalo (data entre `<data_inicio>` e `<data_fim>`), em ordem cronológica. Ambas as datas estarão no formato `<AAAAMMDDHHMM>`. Para cada registro encontrado na listagem, deverá ser impresso o caminho percorrido. Caso nenhum registro for retornado, seu programa deverá exibir a mensagem padrão `AVISO_NENHUM_REGISTRO_ENCONTRADO`.

ATENÇÃO: antes de imprimir a lista de partidas realizadas no período, primeiro é necessário imprimir o caminho percorrido durante a busca binária para encontrar o registro cujo `<inicio>` seja igual à `<data_inicio>` informada pelo usuário ou data posterior mais próxima.

3.10 Liberar espaço

```
VACUUM jogadores;
```

O `ARQUIVO_JOGADORES` deverá ser reorganizado com a remoção física de todos os registros marcados como excluídos e os índices deverão ser atualizados. A ordem dos registros no arquivo "limpo" não deverá ser diferente do arquivo "sujo". Se a operação se concretizar, exibir a mensagem padrão `SUCESSO`.

3.11 Imprimir arquivos de dados

O sistema deverá imprimir os arquivos de dados da seguinte maneira:

(a) Dados dos jogadores:

`\echo file ARQUIVO_JOGADORES`

Imprime o arquivo de dados de jogadores. Caso estiver vazio, apresentar a mensagem padrão `ERRO_ARQUIVO_VAZIO`;

(b) Dados dos *kits*:

`\echo file ARQUIVO_KITS`

Imprime o arquivo de dados de *kits*. Caso estiver vazio, apresentar a mensagem padrão `ERRO_ARQUIVO_VAZIO`.

(c) Dados das partidas:

`\echo file ARQUIVO_PARTIDAS`

Imprime o arquivo de dados de partidas. Caso o arquivo estiver vazio, apresentar a mensagem padrão `ERRO_ARQUIVO_VAZIO`.

(d) Dados de resultados:

`\echo file ARQUIVO_RESULTADOS`

Imprime o arquivo de resultados. Caso o arquivo estiver vazio, apresentar a mensagem padrão `ERRO_ARQUIVO_VAZIO`.

3.12 Imprimir índices primários

O sistema deverá imprimir os índices primários da seguinte maneira:

(a) Índice de jogadores com `id_jogador` e `rrn`:

`\echo index jogadores_idx`

Imprime as *structs* de índice primário de jogadores. Caso o índice estiver vazio, imprimir `ERRO_ARQUIVO_VAZIO`;

(b) Índice de *kits* com `id_kit` e `rrn`:

`\echo index kits_idx`

Imprime as *structs* de índice primário de *kits*. Caso o índice estiver vazio, imprimir `ERRO_ARQUIVO_VAZIO`;

(c) Índice de partidas com `id_partida` e `rrn`:

`\echo index partidas_idx`

Imprime as *structs* de índice primário de partidas. Caso o índice estiver vazio, imprimir `ERRO_ARQUIVO_VAZIO`;

(d) Índice de resultados com `id_jogador`, `id_partida` e `rrn`:

`\echo index resultados_idx`

Imprime as *structs* de índice primário de resultados. Caso o índice estiver vazio, imprimir `ERRO_ARQUIVO_VAZIO`;

3.13 Imprimir índices secundários

O sistema deverá imprimir os índices secundários da seguinte maneira:

(a) Índice de preço com `preco` e `id_kit`:

`\echo index preco_kit_idx`

Imprime as *structs* de índice secundário de *kits*. Caso o índice estiver vazio, imprimir `ERRO_ARQUIVO_VAZIO`;

(b) Índice de data das partidas com `inicio` e `id_partida`:

`\echo index data_idx`

Imprime as *structs* de índice secundário de datas das partidas. Caso o índice estiver vazio, imprimir `ERRO_ARQUIVO_VAZIO`;

(c) Índice de jogador *kits* secundário:
`\echo index jogador_kits_secundario_idx`

Imprime as *structs* de índice secundário com os nomes dos *kits* (`jogador_kits_secundario_idx`) e o número de índice para o primeiro jogador que possui esse *kit*. Caso o índice estiver vazio, imprimir `ERRO_ARQUIVO_VAZIO`.

(d) Índice de jogador *kits* primário (obs: no escopo deste trabalho, este índice também é secundário):
`\echo index jogador_kits_primario_idx`

Imprime as *structs* de índice secundário com o ID de um jogador que possui o *kit* (`jogador_kits_primario_idx`) e o número do índice para o próximo jogador que tenha esse *kit*. Caso o índice estiver vazio, imprimir `ERRO_ARQUIVO_VAZIO`.

3.14 Finalizar

`\q`

Libera a memória e encerra a execução do programa.

4 Arquivos de dados

Como este trabalho será corrigido automaticamente por um juiz online que não aceita funções que manipulam arquivos, os registros serão armazenados e manipulados em *strings* que irão simular os arquivos abertos. Para isso, você deverá utilizar as variáveis globais `ARQUIVO_JOGADORES`, `ARQUIVO_KITS`, `ARQUIVO_PARTIDAS` e `ARQUIVO_RESULTADOS`, e as funções de leitura e escrita em *strings*, como `sprintf` e `sscanf`, para simular as operações de leitura e escrita em arquivo. Os arquivos de dados devem ser no formato ASCII (arquivo texto).

`ARQUIVO_JOGADORES`: deverá ser organizado em registros de tamanho fixo de 136 *bytes* (136 caracteres).

Os campos `apelido` (tamanho máximo de 43 *bytes*) e `kits` (tamanho máximo de 39 *bytes*: até 10 valores, sendo cada *kit* representado pelo seu id de 3 *bytes*) devem ser de tamanho variável. O campo multi-valorado `kits` deve ter seus valores separados por '`|`'.

Os demais campos são de tamanho fixo e possuem as seguintes especificações: `id_jogador` (11 *bytes*), `cadastro` (12 *bytes*), `premio` (12 *bytes*), `saldo` (13 *bytes*), totalizando 48 *bytes* de tamanho fixo. Portanto, os campos de tamanho fixo de um registro ocuparão 48 *bytes*. Os campos devem ser separados pelo caractere delimitador '`;`' (ponto e vírgula), e cada registro terá 6 delimitadores (um para cada campo). Caso o registro tenha menos de 136 *bytes*, o espaço restante deverá ser preenchido com o caractere '`#`'. Como são 48 *bytes* fixos + 6 *bytes* de delimitadores, então os campos variáveis devem ocupar no máximo 82 *bytes*, para que o registro não exceda os 136 *bytes*.

Exemplo de arquivo de dados de jogadores:

```
44679595970;Badast;202411191430;000000000000;0000002000.00;001;################
################################################################42714376303;Dragoniste
r;202411191450;000000000000;0000000010.00;000|001;############################
####################################65037521605;Dogenator;202411200230;0000000000
00;0000000000.00;;##############################################################
############93804621236;looz;202411200400;000000000002;0000000000.00;;#########
#########################################################################5765399352
4;rafz;202411200130;000000000000;0000000000.00;;##############################
################################################91004621999;zephyrer;20241119143
0;000000000000;00000004010.00;;################################################
#########################91004621995;El grande padre;202411190730;000000000000;
0000003905.00;;################################################################
###71004621995;Luizerah;202411190430;000000000000;0000000800.00;;##############
############################################################
```

`ARQUIVO_KITS`: o arquivo de *kits* deverá ser organizado em registros de tamanho fixo de `100` *bytes* (*i.e.*, `100` caracteres).

Os campos `nome` (máximo de 20 *bytes*) e `poder` (máximo de 60 *bytes*) devem ser de tamanhos variáveis. Os demais campos são de tamanho fixo e possuem as seguintes especificações: `id_kit` (3 *bytes*) e `preco` (13 *bytes*), totalizando 16 *bytes* de tamanho fixo. Assim como no registro de jogadores, os campos devem ser separados pelo delimitador '`;`', cada registro terá 4 delimitadores para os campos e, caso o registro tenha menos que `96` *bytes*, o espaço restante deverá ser preenchido com o caractere '`#`'. Como são 16 *bytes* de tamanho fixo + 4 *bytes* para os delimitadores, os campos variáveis devem ocupar no máximo `80` *bytes* para que não se exceda o tamanho do registro.

Exemplo de arquivo de dados de *kits*:

```
000;RBD;Ao morrer, permite voltar 10s antes de sua morte;0000000000.00;########
######################001;Stomper;Causa dano ao cair sobre o inimigo;0000055000.
00;########################################002;Viper;Atinge o inimigo com veneno
;0000004000.00;##############################
```

`ARQUIVO_PARTIDAS`: o arquivo de partidas deverá ser organizado em registros de tamanho fixo de 162 *bytes* (162 caracteres). Todos os campos são de tamanho fixo e possuem as seguintes especificações: `id_partida` (8 *bytes*), `inicio` (12 *bytes*), `duracao` (6 *bytes*), `cenario` (4 *bytes*) e `id_jogadores` (132 *bytes*, correspondendo a 12 ID de jogadores × 11 *bytes* cada).

Exemplo de arquivo de dados de partidas com 2 registros:

```
000000002024112018300220350005666795959704271437630365037521605938046212365765
99352491004621999910046219957100462199551004621995110046219950100462100000000
0012024111914300220350005666795959704271437630365037521605938046212365765399352
49100462199991004621995710046219955100462199511004621995010046219950100462100
04621000
```

`ARQUIVO_RESULTADOS`: o arquivo de resultados deverá ser organizado em registros de tamanho fixo de 36 *bytes* (36 caracteres). Todos os campos são de tamanho fixo e possuem as seguintes especificações: `id_jogador` (11 *bytes*), `id_partida` (8 *bytes*), `id_kit` (3 *bytes*), `colocacao` (4 *bytes*), `sobrevivencia` (6 *bytes*) e `eliminacoes` (4 *bytes*).

Exemplo de arquivo de dados de resultados contendo 12 registros:

```
446795959700000000000100010220100005910046219990000000000000020210050003910046219
95000000000001000302010900010100462100000000000001000401551500005100462199500000
0000010005014210000042714376303000000000000060132520000650375216050000000000007
012417000057653993524000000000000008011442000001004621995000000000010009010522
0007100462199500000000000001000541400001100462199500000000000011004244000093804
62123600000000001001200312200000
```

5 Instruções para as operações com os registros

- **Inserção**: cada jogador, *kit*, partida e resultado deverá ser inserido no final de seus respectivos arquivos de dados, e atualizados os índices.

- **Remoção**: o registro de um dado jogador deverá ser localizado acessando o índice primário (`id_jogador`). A remoção deverá colocar o marcador `*|` nas duas primeiras posições do registro removido. O espaço do registro removido não deverá ser reutilizado para novas inserções. Observe que o registro deverá continuar ocupando exatamente `TAM_REGISTRO_JOGADOR` *bytes*. Além disso, no índice primário, o RRN correspondente ao registro removido deverá ser substituído por `-1`.

- **Atualização**: existem dois campos alteráveis: (*i*) o saldo do jogador e (*ii*) data da última premiação recebida pelo jogador. Eles possuem tamanho fixo e pré-determinado. A atualização deve ser feita diretamente no registro, exatamente na mesma posição em que estiverem (em hipótese alguma o registro deverá ser removido e em seguida inserido).

6 Índices

No cenário atual, há um grande volume de dados, e nem mesmo os índices cabem em RAM. Para simular essa situação, a única informação que você deverá manter o todo tempo em memória é o RRN da raiz de cada Árvore-B. Assuma que um nó do índice corresponde a uma página e, portanto, cabe no *buffer* de memória. Dessa forma, trabalhe apenas com a menor quantidade de nós necessários das árvores por vez, pois isso implica em reduzir a quantidade de *seeks* e de informação transferida entre as memórias primária e secundária.

É terminantemente proibido manter uma cópia dos índices inteiros em Tipos Abstratos de Dados (TADs) que não sejam Árvores-B ou, no caso dos *kits* que os jogadores possuem, uma lista invertida.

Todas as árvores terão a mesma ordem (m)

Durante a operação de inserção, caso ocorra *overflow*, **a chave promovida deverá ser sempre a maior chave da página à esquerda** do processo de divisão. Nesse caso, se a ordem (m) for par, a página à esquerda ficará com 1 (uma) chave a mais do que à direita no final do processo de divisão. Todo novo nó criado deverá ser inserido no final do respectivo arquivo de índice.

Durante a operação de remoção, caso:

1. **a chave a ser removida não esteja em um nó folha**, então, ela deverá primeiro ser substituída pela **maior chave da subárvore esquerda, ou seja, seu predecessor**;

2. ocorra *underflow*, será necessário emprestar chaves de um de seus nós irmãos (páginas à esquerda ou à direita), sempre que possível, dando **preferência para o nó irmão à direita**, exceto se o número de chaves nesse nó já for o mínimo ou se não houver um nó irmão à direita:

- Se a redistribuição de chaves com um dos nós irmãos for possível, então **somente 1 (uma) chave deverá ser transferida** para o nó onde ocorreu o *underflow*;
- Caso contrário, será necessário concatenar as chaves do nó onde ocorreu o *underflow* e de um de seus nós irmãos (páginas à esquerda ou à direita), sempre que possível, dando **preferência para o nó irmão à direita**, se houver. Nesse caso, a página à esquerda do processo de fusão receberá as chaves da página à direita, deixando-a vazia. O espaço ocupado por páginas vazias **não deverá ser reutilizado** em novas inserções.

A lista invertida de `jogador_kits` também cresceu e, agora, será armazenada em um arquivo.

6.1 Índices primários

6.1.1 `jogadores_idx`

Índice primário do tipo Árvore-B que contém o `id_jogador` (chave primária) e o RRN do respectivo registro no arquivo de dados, ordenado pela chave primária.

Cada registro da Árvore-B para o índice `jogadores_idx` é composto por:

- *Contador de chaves:* 3 bytes para a quantidade de chaves;
- *Chaves ordenadas:* $(m - 1)$ * (11 bytes de chave primária + 4 bytes para o RRN do arquivo de dados) bytes para armazenar as chaves. Para as chaves não usadas, preencha todos os bytes com '`#`';
- *Indicador de folha:* 1 byte para indicar se o nó é folha '`T`' (`True`) ou não '`F`' (`False`);
- *Apontadores para os filhos:* (m * 3 bytes para cada RRN filho) bytes para indicar os RRNs dos nós descendentes do nó atual. Note que esse RRN se refere ao próprio arquivo de índice primário. Utilize '`***`' para indicar que aquela posição do vetor de descendentes é nula.

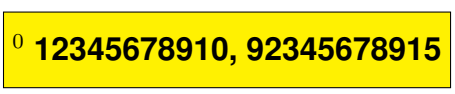
Exemplo da representação da Árvore-B de ordem 3, após a inserção das chaves: **12345678910**, **92345678915**, **09898989999**, **11111111111** e **10111213141** (nesta ordem).

- Inserindo a chave `12345678910`:

Em disco, o arquivo de índice primário seria:

```
001 | 12345678910 0000 | ########### #### | T | *** *** ***
```

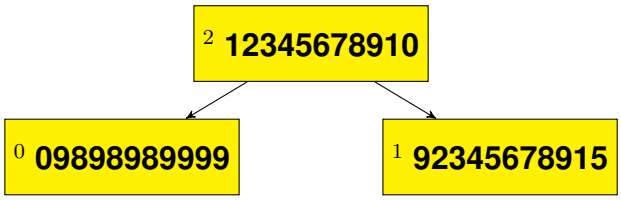
- Inserindo a chave `92345678915`:



Em disco:

```
002 | 12345678910 0000 | 92345678915 0001 | T | *** *** ***
```

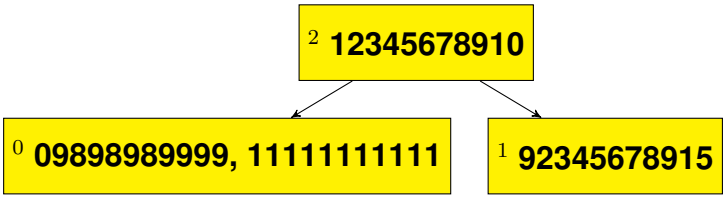
- Inserindo a chave `09898989999`:



Ao inserir a chave iniciada por `09898989999`, ocorre um *overflow* na raiz, sendo necessário criar então, 2 novos nós (de RRN 1 e 2, respectivamente). O primeiro, será para a redistribuição de chaves, e o segundo, para receber a promoção de chave que será a nova raiz. Portanto:

```
001 | 09898989999 0002 | ########## #### | T | *** *** ***
001 | 92345678915 0001 | ########## #### | T | *** *** ***
001 | 12345678910 0000 | ########## #### | F | 000 001 ***
```

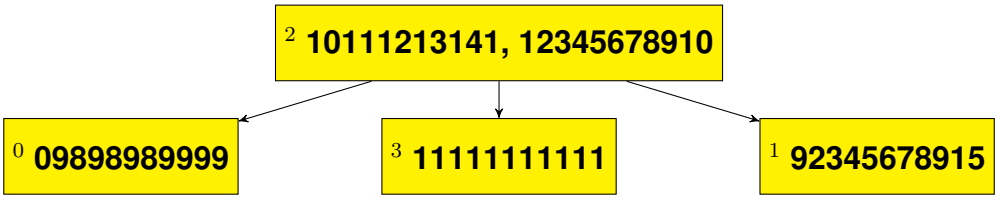
- Inserindo a chave `11111111111`:



Em disco:

```
002 | 09898989999 0002 | 11111111111 0003 | T | *** *** ***
001 | 92345678915 0001 | ########## #### | T | *** *** ***
001 | 12345678910 0000 | ########## #### | F | 000 001 ***
```

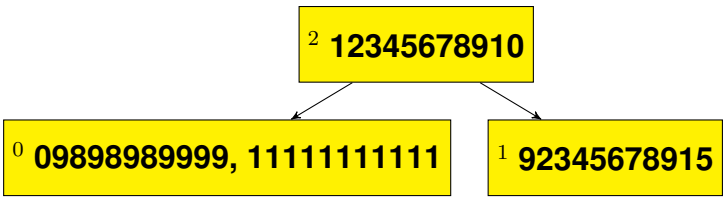
- Inserindo a chave `10111213141`:



Ao inserir a chave iniciada por `10111213141`, ocorre um *overflow* no nó de RRN 0 (página à esquerda do processo de divisão), sendo necessário criar 1 novo nó de RRN 3 (página à direita do processo de divisão) para a redistribuição de chaves e promover a chave `10111213141` para o nó raiz. Portanto:

```
001 | 09898989999 0002 | ########## #### | T | *** *** ***
001 | 92345678915 0001 | ########## #### | T | *** *** ***
002 | 10111213141 0004 | 12345678910 0000 | F | 000 003 001
001 | 11111111111 0003 | ########## #### | T | *** *** ***
```

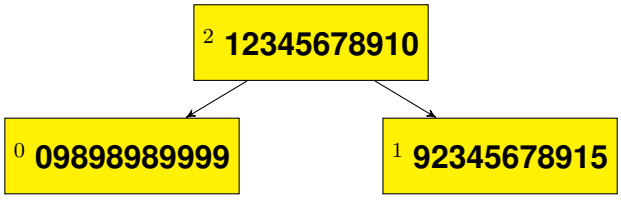
- Removendo a chave `10111213141`:



Ao remover a chave iniciada por `10111213141`, como ela se encontra no nó raiz (de RRN 2) e esse nó não é folha, então, ela é primeiro substituída pela chave iniciada por `09898989999`, que é a sua predecessora, isto é, maior chave do nó de RRN 0 (subárvore esquerda). Em seguida, ela é removida do nó de RRN 0, causando um *underflow* nesse nó, sendo necessário emprestar chaves do nó de RRN 3 (página à direita). Como o número de chaves nesse nó já é o mínimo, então é necessário fundir ambos os nós (de RRN 0 e 3 respectivamente). O primeiro (página à esquerda), será usado para a concatenação de chaves, já o segundo (página à direita), ficará vazio. Isso fará com que a chave iniciada por `09898989999` seja rebaixada do nó raiz (de RRN 2). Portanto:

```
002 | 09898989999 0002 | 11111111111 0003 | T | *** *** ***
001 | 92345678915 0001 | ########## #### | T | *** *** ***
001 | 12345678910 0000 | ########## #### | F | 000 001 ***
000 | ########## #### | ########## #### | T | *** *** ***
```

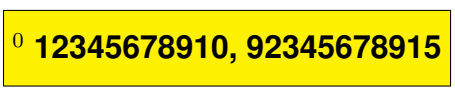
- Removendo a chave `11111111111`:



Em disco:

```
001 | 09898989999 0002 | ########## #### | T | *** *** ***
001 | 92345678915 0001 | ########## #### | T | *** *** ***
001 | 12345678910 0000 | ########## #### | F | 000 001 ***
000 | ########## #### | ########## #### | T | *** *** ***
```

- Removendo a chave `09898989999`:



Ao remover a chave iniciada por `09898989999`, ocorre um *underflow* no nó de RRN 0, sendo necessário emprestar chaves do nó de RRN 1 (página à direita). Como o número de chaves nesse nó já é o mínimo, então, é necessário fundir ambos os nós (de RRN 0 e 1, respectivamente). O primeiro (página à esquerda), será usado para a concatenação de chaves, já o segundo (página à direita), ficará vazio. Isso fará com que a chave iniciada por `12345678910` seja rebaixada do nó raiz (de RRN 2), deixando-o vazio. Por esse motivo, o nó de RRN 0 é definido como sendo o novo nó raiz. Portanto:

```
002 | 12345678910 0000 | 92345678915 0001 | T | *** *** ***
000 | ########## #### | ########## #### | T | *** *** ***
000 | ########## #### | ########## #### | F | *** *** ***
000 | ########## #### | ########## #### | T | *** *** ***
```

- Removendo a chave `92345678915`:

Em disco:

```
001 | 12345678910 0000 | ########## #### | T | *** *** ***
000 | ########## #### | ########## #### | T | *** *** ***
000 | ########## #### | ########## #### | F | *** *** ***
000 | ########## #### | ########## #### | T | *** *** ***
```

- Removendo a chave `12345678910`, a árvore ficará vazia e a representação em disco será:

```
000 | ########## #### | ########## #### | T | *** *** ***
000 | ########## #### | ########## #### | T | *** *** ***
000 | ########## #### | ########## #### | F | *** *** ***
000 | ########## #### | ########## #### | T | *** *** ***
```

Em resumo, um nó da Árvore-B de `jogadores_idx`, de ordem m, possui as seguintes informações:

```
{QUANTIDADE DE CHAVES}
{ID_JOGADOR_1} {RRN_1}
{ID_JOGADOR_2} {RRN_2}
...
{ID_JOGADOR_(m-1)} {RRN_(m-1)}
{FOLHA}
{RRN FILHO_1} {RRN FILHO_2} ... {RRN_FILHO_m}
```

Note que, não há quebras de linhas no arquivo, espaços ou *pipes* (‘|’), Eles foram inseridos apenas para exemplificar a sequência de registros.

6.1.2 `kits_idx`

Índice primário do tipo Árvore-B que contém o `id_kit` do *kit* (chave primária) e o RRN do respectivo registro no arquivo de dados, ordenado pela chave primária.

Cada registro da Árvore-B para o índice `kits_idx` é composto por:

- *Contador de chaves:* 3 bytes para a quantidade de chaves;
- *Chaves ordenadas:* $(m-1)$ * (3 bytes de chave primária + 4 bytes para o RRN do arquivo de dados) bytes para armazenar as chaves. Para as chaves não usadas, preencha todos os bytes com '`#`';
- *Indicador de folha:* 1 byte para indicar se o nó é folha '`T`' (`True`) ou não '`F`' (`False`);
- *Apontadores para os filhos:* (m * 3 bytes para cada RRN filho) bytes para indicar os RRNs dos nós descendentes do nó atual. Note que esse RRN se refere ao próprio arquivo de índice primário. Utilize '`***`' para indicar que aquela posição do vetor de descendentes é nula.

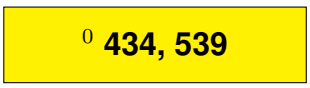
Exemplo da representação da Árvore-B de ordem 3 após a inserção das chaves: `434`, `539` e `394` (nesta ordem).

- Inserindo a chave `434`:

Em disco, o arquivo de índice primário seria:

```
001 | 434 0000 | ####### #### | T | *** *** ***
```

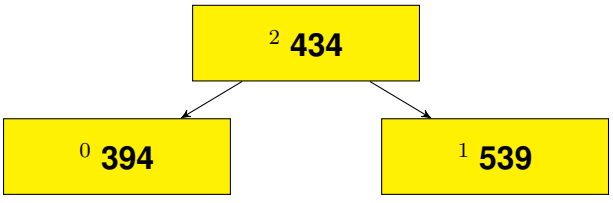
- Inserindo a chave `539`:



Em disco:

```
002 | 434 0000 | 539 0001 | T | *** *** ***
```

- Inserindo a chave `394`:



Ao inserir a chave `394`, ocorre um *overflow* na raiz, sendo necessário criar então, 2 novos nós (de RRN 1 e 2 respectivamente). O primeiro, será para a redistribuição de chaves, e o segundo para receber a promoção de chave que será a nova raiz. Portanto:

```
001 | 394 0002 | ####### #### | T | *** *** ***
001 | 539 0001 | ####### #### | T | *** *** ***
001 | 434 0000 | ####### #### | F | 000 001 ***
```

Em resumo, um nó da Árvore-B de `kits_idx`, de ordem m, possui as seguintes informações:

```
{QUANTIDADE DE CHAVES}
{ID_KIT_1} {RRN_1}
{ID_KIT_2} {RRN_2}
...
{ID_KIT_(m-1)} {RRN_(m-1)}
{FOLHA}
{RRN FILHO_1} {RRN FILHO_2} ... {RRN_FILHO_m}
```

Note que, não há quebras de linhas no arquivo, espaços ou *pipes* (‘`|`’), Eles foram inseridos apenas para exemplificar a sequência de registros.

6.1.3 `partidas_idx`

Índice primário que contém o ID da partida (chave primária) e o RRN respectivo do registro no arquivo de partidas, ordenado pelo ID da partida (`id_partida`);

Cada registro da Árvore-B para o índice `partidas_idx` é composto por:

- *Contador de chaves:* 3 bytes para a quantidade de chaves;
- *Chaves ordenadas:* $(m-1)$ * (8 bytes de chave primária + 4 bytes para o RRN do arquivo de dados) bytes para armazenar as chaves. Para as chaves não usadas, preencha todos os bytes com ‘`#`’;
- *Indicador de folha:* 1 byte para indicar se o nó é folha ‘`T`’ (True) ou não ‘`F`’ (False);
- *Apontadores para os filhos:* (m * 3 bytes para cada RRN filho) bytes para indicar os RRNs dos nós descendentes do nó atual. Note que esse RRN se refere ao próprio arquivo de índice primário. Utilize ‘`***`’ para indicar que aquela posição do vetor de descendentes é nula.

Exemplo da representação da Árvore-B de ordem 3 após a inserção das chaves: `54348768`, `65399951` e `73944063` (nesta ordem).

Inserindo a chave `54348768`:

Em disco, o arquivo de índice primário seria:

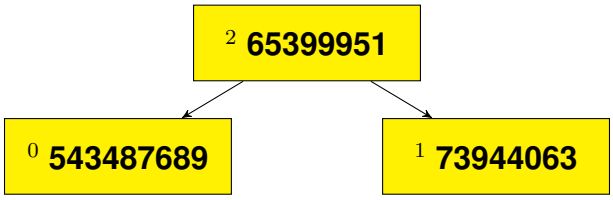
```
001 | 54348768 0000 | ######## #### | T | *** *** ***
```

Inserindo a chave `65399951`:

Em disco:

```
002 | 54348768 0000 | 65399951 0001 | T | *** *** ***
```

Inserindo a chave `73944063`:



Ao inserir a chave `73944063`, ocorre um *overflow* na raiz, sendo necessário criar então, 2 novos nós (de RRN 1 e 2 respectivamente). O primeiro será para a redistribuição de chaves, e o segundo para receber a promoção de chave que será a nova raiz. Portanto:

```
001 | 54348768 0002 | ######## #### | T | *** *** ***
001 | 73944063 0001 | ######## #### | T | *** *** ***
001 | 65399951 0000 | ######## #### | F | 000 001 ***
```

Em resumo, um nó da Árvore-B de compras, de ordem m, possui as seguintes informações:

```
{QUANTIDADE DE CHAVES}
{ID_PARTIDA_1} {RRN_1}
{ID_PARTIDA_2} {RRN_2}
...
{ID_PARTIDA_(m-1)} {RRN_(m-1)}
{FOLHA}
{RRN FILHO_1} {RRN FILHO_2} ... {RRN_FILHO_m}
```

Novamente, não há quebras de linhas no arquivo. Elas foram inseridas apenas para exemplificar a sequência de registros.

6.1.4 `resultados_idx`

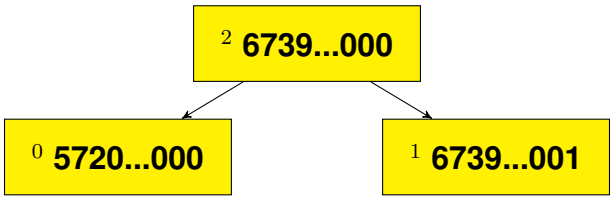
Índice primário que consiste no ID de um jogador, ID de uma partida e o RRN relativo ao registro no arquivo de resultados, ordenado pelo ID do jogador (`id_jogador`) e ID da partida (`id_partida`).

Cada registro da Árvore-B para o índice `resultados_idx` é composto por:

- *Contador de chaves:* 3 bytes para a quantidade de chaves;
- *Chaves ordenadas:* $(m-1)$ * (19 bytes de chave primária [11 bytes para o campo `id_jogador` e 8 bytes para o campo `id_partida`] + 4 bytes para o RRN do arquivo de dados) bytes para armazenar as chaves. Para as chaves não usadas, preencha todos os bytes com '`#`';

- *Indicador de folha:* 1 byte para indicar se o nó é folha '`T`' (True) ou não '`F`' (False);
- *Apontadores para os filhos:* (m * 3 bytes para cada RRN filho) bytes para indicar os RRNs dos nós descendentes do nó atual. Note que esse RRN se refere ao próprio arquivo de índice primário. Utilize '`***`' para indicar que aquela posição do vetor de descendentes é nula.

Exemplo da representação da Árvore-B de ordem 3 após a inserção das chaves: `67392034567 00000000`, `57209482376 00000000` e `67392034567 00000001` (nesta ordem).



Em disco:

```
001 | 57209482376 00000000 0002| ############ ######## #### | T | *** *** ***
001 | 67392034567 00000001 0001| ############ ######## #### | T | *** *** ***
001 | 67392034567 00000000 0000| ############ ######## #### | F | 000 001 ***
```

Em resumo, um nó da Árvore-B de compras, de ordem m, possui as seguintes informações:

```
{QUANTIDADE DE CHAVES}
{ID_JOGADOR_1} {ID_PARTIDA_1} {RRN_1}
{ID_JOGADOR_2} {ID_PARTIDA_2} {RRN_2}
...
{ID_JOGADOR_(m-1)} {ID_PARTIDA_(m-1)} {RRN_(m-1)}
{FOLHA}
{RRN FILHO_1} {RRN FILHO_2} ... {RRN_FILHO_m}
```

Novamente, não há quebras de linhas no arquivo. Elas foram inseridas apenas para exemplificar a sequência de registros.

6.2 Índices secundários

6.2.1 `data_idx`

Índice do tipo Árvore-B que contém as partidas ordenadas por data e a chave primária (`id_partida`) da partida específica.

Cada registro da Árvore-B para o índice `data_idx` é composto por:

- *Contador de chaves:* 3 bytes para a quantidade de chaves;
- *Chaves ordenadas:* ($m - 1$) * (20 bytes [12 bytes do `inicio` + 8 bytes de `id_partida`]) bytes para armazenar as chaves. Para as chaves não usadas ou para completar o tamanho da chave, preencha todos os bytes com '`#`';
- *Indicador de folha:* 1 byte para indicar se o nó é folha '`T`' (True) ou não '`F`' (False);

- *Apontadores para os filhos:* (m * 3 bytes para cada RRN filho) bytes para indicar os RRNs dos nós descendentes do nó atual. Note que, assim como no índice primário, esse RRN se refere ao próprio arquivo de índice secundário. Utilize '`***`' para indicar que aquela posição do vetor de descendentes é nula.

Exemplo de arquivo de índice secundário de partidas, representado por uma Árvore-B de ordem 5, após a inserção da chave '`202302011010 00000007`'.


```
001 | 202302011010 00000007 |
      ############ ######## |
      ############ ######## |
      ############ ######## |
      T | *** *** *** *** ***
```

Em resumo, um nó da Árvore-B de partidas, de ordem m, possui as seguintes informações:

```
{QUANTIDADE DE CHAVES}
{HORARIO_PARTIDA_1} {ID_PARTIDA_1}
{HORARIO_PARTIDA_2} {ID_PARTIDA_2}
...
{HORARIO_PARTIDA_(m-1)} {ID_PARTIDA_(m-1)}
{FOLHA}
{RRN FILHO_1} {RRN FILHO_2} ... {RRN_FILHO_m}
```

Lembre-se, aqui também não há quebras de linhas no arquivo, espaços ou *pipes* ('`|`'), Eles foram inseridos apenas para exemplificar a sequência de registros.

6.2.2 `preco_kit_idx`

Índice secundário do tipo Árvore-B que contém os *kits* ordenados por preço e a chave primária (`id_kit`) do *kit* específico.

Cada registro da Árvore-B para o índice `preco_kit_idx` é composto por:

- *Contador de chaves:* 3 bytes para a quantidade de chaves;
- *Chaves ordenadas:* $(m-1)$ * (16 bytes [13 bytes para o `preco` + 3 bytes para o `id_kit`]) bytes para armazenar as chaves. Para as chaves não usadas, preencha todos os bytes com '`#`';
- *Indicador de folha:* 1 byte para indicar se o nó é folha '`T`' (True) ou não '`F`' (False);
- *Apontadores para os filhos:* (m * 3 bytes para cada RRN filho) bytes para indicar os RRNs dos nós descendentes do nó atual. Note que, assim como no índice primário, esse RRN se refere ao próprio arquivo de índice secundário. Utilize '`***`' para indicar que aquela posição do vetor de descendentes é nula.

Exemplo de arquivo de índice secundário representado por uma Árvore-B de ordem 4, após a inserção das chaves na ordem: '0000000050000 000', '0000000000000 001' e '0000000045000 002'.


```
003 | 0000000000000 001 | 000000045000 002 |
      0000000050000 000 | T | *** *** *** ***
```

Em resumo, um nó da Árvore-B de datas das inscrições, de ordem m, possui as seguintes informações:

```
{QUANTIDADE DE CHAVES}
{PRECO_1} {ID_KIT_1}
{PRECO_2} {ID_KIT_2}
...
{PRECO_(m-1)} {ID_KIT_(m-1)}
{FOLHA}
{RRN FILHO_1} {RRN FILHO_2} ... {RRN_FILHO_m}
```

Novamente, não há quebras de linhas no arquivo, espaços ou *pipes* ('|'), Eles foram inseridos apenas para exemplificar a sequência de registros.

6.2.3 `jogador_kits_idx`

Índice secundário do tipo *lista invertida*. Será necessário manter dois índices (`jogador_kits_primario_idx` e `jogador_kits_secundario_idx`), sendo que o primário possui os IDs de jogadores (`id_jogador`) que possuem certo *kit* e o apontador para o próximo jogador que possui o mesmo *kit* nesse mesmo índice primário. Se não houver um próximo jogador, esse apontador deve possuir o valor -1. No índice secundário estão os nomes dos *kits*, assim como a referência do primeiro jogador que possui aquele *kit* no índice primário.

Para simplificar o entendimento, considere o seguinte exemplo:

Jogador *kits* primário	
ID jogador	próx. registro
0	4
4	-1
4	7
4	5
3	-1
7	6
5	-1
5	-1
...	...

Jogador *kits* secundário	
Nome	registro
KANGAROO	0
CANIBAL	1
STOMPER	3
RDB	2
...	...

No exemplo acima, a tabela de jogador *kits* secundário possui o nome do *kit* na primeira coluna, assim como o RRN do primeiro jogador que possui aquele *kit* que foi inserido na tabela de jogador *kits* primário. Na tabela primária, tem-se na primeira coluna o ID dos jogadores que possuem cada *kit*. Note que o jogador com ID = 4 aparece três vezes no exemplo, o que significa que ele possui três *kits* diferentes. Na segunda coluna da tabela primária, temos o RRN para o próximo jogador que possui o mesmo *kit* na própria tabela de jogador *kits* primária, sendo que, RNN = −1, significa que aquele jogador é o último que possui o mesmo *kit*. Vale destacar que o índice primário **não** precisa estar organizado, pois cada registro já possui uma referência direta para o próximo (assim como em uma lista encadeada).

Exemplo de como as tabelas acima ficariam nos arquivos de índices primário e secundário:

```
Índice primário:
00000000 0004 00000004 -001 00000004 0007
00000004 0005 00000003 -001 00000007 0006
00000005 -001 00000005 -001


Índice secundário:
KANGAROO############ 0000 CANIBAL############# 0001
STOMPER############# 0003 RDB################## 0002
```

Lembrando novamente que as quebras de linha e espaços foram apenas adicionados para facilitar a visualização.

Deverá ser desenvolvida uma rotina para a criação de cada índice. Os índices serão sempre criados e manipulados em memória principal na inicialização e liberados ao término do programa. Note que o ideal é que os índices primários sejam criados primeiro, depois os secundários.

Após a criação de cada arquivo de índice, deverá ser impresso na tela a frase padrão `INDICE_CRIADO`.

7 Inicialização do programa

Para que o programa inicie corretamente, o seguinte procedimento deverá ser realizado:

1. Inserir o comando `SET BTREE_ORDER '<ORDEM DAS ÁRVORES-B>';` para informar a ordem das Árvores-B. Caso não informado, é definido como 3 por padrão;
2. Inserir o comando `SET ARQUIVO_JOGADORES TO '<DADOS DE JOGADORES>';` caso queira inicializar o programa com um arquivo de jogadores já preenchido;
3. Inserir o comando `SET ARQUIVO_KITS TO '<DADOS DE KITS>';` caso queira inicializar o programa com um arquivo de *kits* já preenchido;
4. Inserir o comando `SET ARQUIVO_PARTIDAS TO '<DADOS DE PARTIDAS>';` caso queira inicializar o programa com um arquivo de partidas já preenchido;

5. Inserir o comando `SET ARQUIVO_RESULTADOS TO '<DADOS DE RESULTADOS>';` caso queira inicializar o programa com um arquivo de resultados já preenchido;

6. Inicializar as estruturas de dados dos índices.

8 Implementação

Implemente suas funções utilizando o código-base fornecido. **Não é permitido modificar os trechos de código pronto ou as estruturas já definidas**. Ao imprimir um registro, utilize as funções `exibir_jogador(int rrn)`, `exibir_kit(int rrn)`, `exibir_partida(int rrn)` ou `exibir_resultado(int rrn)`.

Note que são seis índices do tipo Árvore-B e uma lista invertida. **Não é necessário implementar rotinas específicas para a manipulação de cada árvore**.

Lembre-se que funções como *bsearch* e *qsort*, por exemplo, são totalmente genéricas e funcionam com qualquer vetor, basta informar o tamanho de cada elemento, o número de posições do vetor e a função de comparação. Use esse mesmo conceito nas funções de manutenção de Árvores-B, lembrando que a única informação que se altera entre elas é o tamanho das chaves.

Implemente as rotinas abaixo com, obrigatoriamente, as seguintes funcionalidades:

- Estruturas de dados adequadas para armazenar os índices em arquivos simulados por meio de *strings*;
- Verificar se os arquivos de dados existem;
- Criar os índices primários: deve refazer os índices primários a partir dos arquivos de dados;
- Criar os índices secundários: deve refazer os índices secundários a partir dos arquivos de dados;
- Inserir um registro: modifica os arquivos de dados e os índices na memória principal;
- Buscar por registro: busca pela chave primária ou por uma das chaves secundárias;
- Alterar um registro: modifica o campo do registro diretamente no arquivo de dados;
- Remover um registro: modifica o arquivo de dados e o índice primário na memória principal;
- Listar registros: listar todos os registros ordenados pela chave primária ou por uma das chaves secundárias;
- Liberar espaço: organizar o arquivo de dados e refazer os índices.

Lembre-se que, sempre que possível, é **obrigatório o uso da busca binária**, com o arredondamento para **cima** para buscas feitas em índices tanto primários quanto secundários.

9 Resumo de alterações em relação ao T01

Para facilitar o desenvolvimento do T02, é possível reutilizar as funções já implementadas no T01, com ênfase nas seguintes alterações:

1. Todos os índices serão do tipo Árvore-B, menos o índice de lista invertida, que também deve ser alterado para ser armazenado em arquivo;
2. A ordem das Árvores-B é informada na inicialização do programa e será a mesma para todas as árvores.

10 Dicas

ATENÇÃO: Comece o trabalho o mais cedo possível, pois você **PRECISA** dedicar várias horas para conseguir concluir o trabalho.

- Ao ler uma entrada, tome cuidado com caracteres de quebra de linha (`\n`) não capturados;
- Você nunca deve perder a referência do começo do arquivo, então não é recomendável percorrer a *string* diretamente pelo ponteiro `ARQUIVO`. Um comando equivalente a `fseek(f, 256, SEEK_SET)` é `char *p = ARQUIVO + 256`;
- Diferentemente do `fscanf`, o `sscanf` não movimenta automaticamente o ponteiro após a leitura;
- O `sprintf` adiciona automaticamente o caractere `\0` no final da *string* escrita. Em alguns casos você precisará sobrescrever a posição manualmente. Você também pode utilizar o comando `strncpy` para escrever em *strings*. Esse comando, diferentemente do `sprintf`, não adiciona o caractere nulo no final;
- Atenção ao uso do `sprintf` e do `strcpy`, pois a *string* de destino deve ter tamanho suficiente para também receber o caractere '`\0`';
- A função `strtok` permite navegar nas *substrings* de uma certa *string* dado o(s) delimitador(es). Porém, tenha em mente que ela deve ser usada em uma cópia da *string* original, pois ela modifica o primeiro argumento;
- É recomendado olhar as funções `qsort` e `bsearch` da biblioteca C. Caso se baseie nelas para a sua implementação, leia suas documentações;
- Para o funcionamento ideal do seu programa, é necessário utilizar a busca binária;
- Utilize as funções auxiliares que estão implementadas no código-base, como por exemplo:
  (a) `int btree_register_size (btree *t)`
  (b) `btree_node btree_node_malloc (btree *t)`
  (c) `void btree_node_free (btree_node no)`
  (d) `char* strpadright (char *str, char pad, unsigned size)`

"Às vezes, as perguntas são complicadas e as respostas são simples"
— L. Lawliet