



## TRABALHO 01

### INDEXAÇÃO

Prazo para entrega: 16/12/2024 – 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

Você, jovem gafanhoto da Computação, estava conversando com seus amigos sobre seus antigos *hobbies* e percebeu como muitos de vocês aproveitaram intensamente a gloriosa época do Minecraft. Tinham tantos minigames, como “Sky Wars”, “Bed Wars” e o mais icônico, aquele que marcou gerações, o “Hunger Games” – posteriormente renomeado para “Hardcore Games”.

Junto com os seus amigos, vocês decidiram reviver essas memoráveis experiências criando um servidor de Minecraft. Nesse servidor, inspirado em competições e desafios épicos, cada jogador pode se cadastrar e participar de batalhas intensas contra outros membros da comunidade. Além disso, o servidor permite a conquista de prêmios que atestam o progresso de cada participante, tornando as disputas recompensadoras. Eis que nasce o inigualável **UFSCraft**!

As regras são simples, cada uma das partidas conta com 12 jogadores que são colocados em um cenário específico e enfrentam-se até que apenas um sobreviva (*battle royale*). Cada um dos jogadores conta com um *kit* que contém um poder único que pode ser usado de diversas maneiras. Para participar dessa atividade, um aluno deve se cadastrar como um jogador e adicionar saldo (ou aumentar o seu saldo com prêmios). Com o saldo, o jogador é capaz de “comprar” *kits* diferentes e utilizá-los em partidas, auxiliando-os nos seus intensos duelos.

A princípio, a organização estava sendo feita manualmente em uma planilha do excel, porém, com o crescimento das atividades do *UFSCraft*, notou-se a necessidade de armazenar os dados de maneira conveniente e de fácil manipulação. Dessa forma, você se reuniu novamente com os seus amigos, que infelizmente não eram programadores, para criar um MVP (*Minimum Viable Product* – Produto Mínimo Viável), capaz de administrar digitalmente os dados das partidas. Por você ser o único capaz de programar, a escrita do código ficou sobre sua responsabilidade. Dessa forma, você precisará empregar suas habilidades na linguagem C para criar e manipular de forma eficiente as bases de dados dos jogadores, dos *kits*, das partidas e dos resultados.

## 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 <999999999999>.<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: 446795959704271437630365037521605938046212365765399352491004621999910046219957100462199551004621995110046219950100462199501004621000;

## 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 (000000000000.00). 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 6](#). 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 corrida, 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 por jogadores, kits e partidas deverão ser implementadas. *Em todas elas, será necessário utilizar a busca binária e mostrar o caminho percorrido nos índices da seguinte maneira:*

Registros percorridos:    3 2 0 1

No exemplo acima, os números representam o RRN dos registros que foram percorridos durante a busca até encontrar o registro de interesse ou esgotar as possibilidades.

**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., 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 4.2) 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., 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).



### 3.9.2 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 <AAA-AMDDHHMM>. 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 Criação dos índices

Para que as buscas e as listagens ordenadas dos dados sejam otimizadas, é necessário criar e manter índices em memória (que serão liberados ao término do programa). *Todas as chaves devem ser armazenadas na forma canônica, em caixa alta.*

Pelo menos os seguintes índices deverão ser criados:

### 4.1 Índices primários

- **jogadores\_idx**: índice primário que contém o ID do jogador (chave primária) e o RRN do respectivo registro no arquivo de dados, ordenado pelo ID do jogador (**id\_jogador**);
- **kits\_idx**: índice primário que contém o ID do *kit* (chave primária) e o RRN do respectivo registro no arquivo de dados, ordenado pelo ID do *kit* (**id\_kit**);
- **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**);
- **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**).

### 4.2 Índices secundários

- **preco\_kit\_idx**: índice secundário que contém os *kits* ordenados por preço e a chave primária (**id\_kit**) do *kit* específico.
- **data\_idx**: índice secundário que contém as *partidas* ordenadas pela data e horário de início e a chave primária (**id\_partida**) da *partida* específica.
- **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		Jogador kits secundário	
ID jogador	próx. registro	Nome	registro
0	4	KANGAROO	0
4	-1	CANIBAL	1
4	7	STOMPER	3
4	5	RDB	2
3	-1	...	...
7	6		
5	-1		
5	-1		
...	...		

No exemplo acima, a tabela de jogador *kits* secundário possui o nome 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 modelo. 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 tal *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).

Deverá ser desenvolvida uma rotina para a criação de cada índice. Eles 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`.

## 5 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;;#####
#####51004621995;subaru;
202411200137;000000000000;0000000500.20;;#####
#####11004621995;emilia;202411200245;000000000
000;0000000330.00;;#####
#####01004621995;rem;202411192142;000000000000;0000002050.00;;#####
#####010046
21000;reinhard;202411191737;000000000000;0000000099.99;;#####
#####
```

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  $\times$  11 *bytes* cada).

Exemplo de arquivo de dados de partidas com 2 registros:

```
0000000020241120183002203500056667959597042714376303650375216059380462123657653  
9935249100462199991004621995710046219955100462199511004621995010046219950100462  
1000000000012024111914300220350005666795959704271437630365037521605938046212365  
7653993524910046219999100462199571004621995510046219951100462199501004621995010  
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:

```
4467959597000000000000100010220100005910046219990000000000000202100500039100462  
1995000000000000100030201090001010046210000000000000010004015515000051004621995000  
0000000100050142100000427143763030000000000000060132520000650375216050000000000  
000070124170000576539935240000000000000008011442000001004621995000000000001000901  
05220000710046219950000000000000010005414000011004621995000000000000011004244000  
09380462123600000000000100120031220000
```

## 6 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).

## 7 Inicialização do programa

Para que o programa inicie corretamente, deve-se realizar o seguinte procedimento:

1. Inserir o comando `SET ARQUIVO_JOGADORES TO '<DADOS DE JOGADORES>'`; caso queira inicializar o programa com um arquivo de jogadores já preenchido;
2. Inserir o comando `SET ARQUIVO_KITS TO '<DADOS DE KITS>'`; caso queira inicializar o programa com um arquivo de kits já preenchido;
3. Inserir o comando `SET ARQUIVO_PARTIDAS TO '<DADOS DE PARTIDAS>'`; caso queira inicializar o programa com um arquivo de partidas já preenchido;
4. Inserir o comando `SET ARQUIVO_RESULTADOS TO '<DADOS DE RESULTADOS>'`; caso queira inicializar o programa com um arquivo de resultados já preenchido;
5. 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)`.

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

- Estruturas de dados adequadas para armazenar os índices na memória principal;
- 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 de 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 Dicas

- **ATENÇÃO:** Recomendamos que você comece o trabalho o mais cedo possível, 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;
- 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.

---

“Diante de uma larga frente de batalha, procure o ponto mais fraco  
e, ali, ataque com a sua maior força”  
— Sun Tzu