Roteiro de Desenvolvimento: Hash Table para Variáveis de Ambiente

Este roteiro guiará a criação de uma Hash Table customizada em C, projetada especificamente para armazenar e gerenciar as variáveis de ambiente do seu Minishell. O desenvolvimento é dividido em 4 módulos, começando pela fundação estrutural e evoluindo para operações completas de gerenciamento de memória.

Módulo 1: A Estrutura Fundamental (O Esqueleto)

Objetivo do Módulo: Definir os blocos de construção da Hash Table. Vamos estabelecer como um par chave-valor será representado e como a tabela principal organizará esses pares. Este módulo cobre integralmente a sua issue `1.3`.

Exercício 1: A Estrutura do Par Chave-Valor (O Nó)

Objetivo da Estrutura: Criar a unidade fundamental de armazenamento. Cada nó guardará uma variável de ambiente (ex: `PATH=/bin:/usr/bin`) e um ponteiro para o próximo nó, permitindo a resolução de colisões através de encadeamento (separate chaining).

Tarefa:
Defina a `struct` que representará um item na tabela. Vamos chamá-la de `t_env_item`. Ela deve conter:

- `key`: O nome da variável (ex: "USER").
- `value`: O valor associado (ex: "gmorais").
- `next`: Um ponteiro para o próximo `t_env_item`, para formar uma lista ligada.

```
typedef struct s_env_item {
   char                *key;
   char                *value;
   struct s_env_item   *next;
} t_env_item;
```

Ponto de Reflexão: Por que um ponteiro `next` é essencial nesta estrutura, mesmo antes de sabermos como a tabela funciona? O que isso sugere sobre como lidaremos com múltiplas chaves que podem, eventualmente, ser mapeadas para a mesma posição?

Exercício 2: A Estrutura Principal da Tabela (O Contêiner)

Objetivo da Estrutura: Definir o contêiner principal que irá organizar todos os nós. Esta estrutura conterá um array de ponteiros, onde cada posição (ou "bucket") pode ser o início de uma lista ligada de nós.

Tarefa:
Defina a `struct` da Hash Table, chamada `t_hash_table`. Ela deve conter:

- `size`: A quantidade total de "buckets" (posições) no array.
- `items`: Um array de ponteiros para `t_env_item`. Este será o nosso vetor de listas ligadas.

```
typedef struct s_hash_table {
   int          size;
   t_env_item  **items;
} t_hash_table;
```

Ponto de Reflexão: Por que o membro `items` é um ponteiro para um ponteiro (`t_env_item **`) e não apenas um ponteiro (`t_env_item *`)? O que essa indireção dupla representa em termos de alocação de memória e estrutura de dados?

Módulo 2: O Mecanismo Central

Objetivo do Módulo: Implementar a lógica principal que faz a Hash Table funcionar: a função de hash e a criação da tabela.

Exercício 3: A Função de Hash (O "Endereçador")

Objetivo da Função: Criar uma função que converte uma string (a chave) em um índice numérico. Esse índice determinará em qual "bucket" do array `items` o par chave-valor será armazenado.

Tarefa:
Implemente uma função de hash. O algoritmo "djb2" é um excelente ponto de partida por sua simplicidade e boa distribuição. A função deve receber uma string e retornar um `unsigned long`.

```
// Protótipo da função em, por exemplo, 'include/minishell.h'
unsigned long   hash(char *str);

// Implementação em 'utils/env_hash.c'
unsigned long hash(char *str)
{
   unsigned long hash = 5381;
   int c;

   while ((c = *str++))
       hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
   return (hash);
}
```

Ponto de Reflexão: A função de hash retorna um número grande. Como você garante que o valor retornado seja sempre um índice válido para o seu array `items` de tamanho `size`?

Exercício 4: Criação e Inicialização da Tabela

Objetivo da Função: Escrever uma função que aloque a memória para a `t_hash_table` e para o seu array `items`, garantindo que a tabela comece em um estado limpo e previsível.

Tarefa:
Crie a função `ht_create(int size)`. Ela deve:

1. Alocar memória para a estrutura `t_hash_table`.
2. Alocar memória para o array `items` (que conterá `size` ponteiros).
3. **Crucial:** Inicializar cada um dos `size` ponteiros em `items` com `NULL`.
4. Definir o membro `size` da tabela.

5. Retornar um ponteiro para a tabela recém-criada.

Ponto de Reflexão: Por que é absolutamente crítico inicializar todos os ponteiros do array `items` com `NULL`? O que poderia acontecer durante uma operação de inserção ou busca se essa inicialização fosse omitida?

Módulo 3: Operações Essenciais

Objetivo do Módulo: Implementar as funcionalidades básicas de uma Hash Table: inserir e buscar dados.

Exercício 5: Inserindo um Par Chave-Valor

Objetivo da Função: Criar a função `ht_insert(t_hash_table *table, char *key, char *value)` que adiciona uma nova variável de ambiente na tabela.

Tarefa:
Siga estes passos na implementação:

1. Crie um novo `t_env_item` e aloque memória para ele e para suas cópias de `key` e `value` (use `strdup` ou sua própria versão da `libft`).
2. Calcule o índice usando a função `hash()` e o operador módulo (`%`).
3. Recupere o ponteiro que está no `items[index]`.
4. Faça o `next` do seu novo item apontar para este ponteiro.
5. Atualize `items[index]` para que aponte para o seu novo item (inserindo no início da lista).

Ponto de Reflexão: Inserir no início da lista ligada é uma escolha comum. Qual a principal vantagem de performance dessa abordagem em comparação a percorrer a lista para inserir no final?

Exercício 6: Buscando por um Valor

Objetivo da Função: Implementar `ht_search(t_hash_table *table, char *key)`, que retorna o `value` associado a uma `key`, ou `NULL` se a chave não for encontrada.

Tarefa:
A lógica é a seguinte:

1. Calcule o índice da `key` usando a função de hash.
2. Comece com um ponteiro temporário apontando para `table->items[index]`.
3. Percorra a lista ligada (se houver uma) usando o ponteiro `next`.
4. Em cada nó, compare a `key` procurada com a `key` do nó usando `strcmp`.
5. Se encontrar, retorne um ponteiro para o `value` do nó.
6. Se chegar ao fim da lista (`NULL`) sem encontrar, retorne `NULL`.

Ponto de Reflexão: Qual é a complexidade de tempo (Big O) da busca no melhor caso (nenhuma colisão) e no pior caso (todas as chaves caem no mesmo índice)? Como o `size` da tabela influencia isso?

Módulo 4: Removendo

Exercício 7: Removendo um Par Chave-Valor

Objetivo da Função: Implementar a função `ht_delete(t_hash_table *table, char *key)`, que será a base para o comando `unset`. Esta operação é a mais complexa, pois exige a manipulação cuidadosa de ponteiros para remover um nó de uma lista ligada sem quebrar a sequência.

Tarefa:
Crie a função que remove um item da tabela com base na sua chave. A lógica precisa tratar três cenários distintos dentro de um "bucket":
1. O bucket está vazio.
2. O item a ser removido é o **primeiro** da lista ligada.
3. O item a ser removido está no **meio ou no final** da lista.

Passos Sugeridos:
1. Calcule o índice usando a função de hash.
2. Obtenha o ponteiro para o nó inicial do bucket (`t_env_item *item = table->items[index]`).
3. **Caso 1 (Remoção da cabeça da lista):** Se o `item` não for nulo e sua chave

corresponder à procurada:
* Atualize o início da lista no bucket: `table->items[index] = item->next`.
* Libere a memória do `key`, do `value` e da struct `item`.
* Retorne.
4. **Caso 2 (Remoção do meio/fim):** Se o primeiro item não for o alvo, percorra a lista. Você precisará de dois ponteiros: um para o nó `anterior` e outro para o `atual`.
* Inicie um loop enquanto o `atual` não for nulo e sua chave não corresponder.
* Dentro do loop, atualize: `anterior = atual` e `atual = atual->next`.
* Se o loop terminar porque o item foi encontrado (`atual != NULL`), faça o `anterior->next` apontar para `atual->next`, "pulando" o nó a ser removido.
* Libere a memória do `key`, `value` e da struct do nó `atual`.

Ponto de Reflexão: Por que é estritamente necessário manter um ponteiro para o nó `anterior` ao percorrer a lista para uma remoção? O que aconteceria se você tentasse remover um nó do meio da lista tendo apenas um ponteiro para o nó `atual`?