Relatório: Trabalho 2 - Cuckoo Hashing

Pedro Folloni Pesserl
Departamento de Informática
Universidade Federal do Paraná - UFPR
Curitiba, Brasil
pfp22@inf.ufpr.br

Resumo

Este relatório documenta o software desenvolvido para o segundo trabalho prático da disciplina CI1057 - Algoritmos e Estruturas de Dados 3, do Departamento de Informática da UFPR. O programa é uma implementação em linguagem C da busca, inclusão e exclusão de valores em uma tabela hash de endereçamento aberto, simulando o algoritmo de Cuckoo Hash.

1 Estruturas de dados utilizadas

A tabela hash foi implementada usando as seguintes estruturas, definidas na biblioteca `libchash.h`:

- Entrada da tabela: contém uma chave de busca e uma variável que especifica se a entrada está vazia ou não.

```
struct Entrada {
    int chave, vazia;
};
```

- Tabela Hash: contém dois vetores de Entradas, um para cada tabela do Cuckoo Hash (OBS: `TABLESIZE` é um macro, definido nessa implementação como 11).

```
struct Hash {
    struct Entrada t1[TABLESIZE], t2[TABLESIZE];
};
```

2 Bibliotecas desenvolvidas

A biblioteca desenvolvida para o trabalho foi a `libchash.[hc]`, que define as structs acima, além das seguintes funções (considere, para as funções a seguir, $m =$ `TABLESIZE`):

- `int h1(int k);` – Função hash para a tabela 1, dada por $h_1(k) = k \bmod m$.
- `int h2(int k);` – Função hash para a tabela 2, dada por $h_2(k) = \lfloor m(0,9k - \lfloor 0,9k \rfloor) \rfloor$.
- `int busca_chash(struct Hash *hash, int k);` – Busca uma chave na tabela. Se a posição h_1(`k`) na tabela 1 estiver vazia, checa a posição h_2(`k`) da tabela 2 (veja detalhes sobre esse comportamento na seção 3, adiante). Se a chave não estiver em nenhuma das tabelas, retorna

-1. Se estiver na tabela 1, retorna a posição (um inteiro `i` $\in [0..10]$). Se estiver na tabela 2, retorna a posição mais m (um inteiro `j` $\in [11..21]$). Esses valores foram escolhidos para facilitar a interpretação do resultado em um possível uso da função de busca.

- `int insere_chash(struct Hash *hash, int k);` – Insere uma chave na tabela. Se a inserção ocorrer normalmente, retorna 0. Se houver tentativa de inserir um valor duplicado, não faz nada e retorna 1. Se o valor não puder ser inserido (colisão na tabela 2), não faz nada e retorna 2.

- `int exclui_chash(struct Hash *hash, int k);` – Exclui um valor da tabela hash, seja da tabela 1 ou da tabela 2. Na prática, a posição na tabela é marcada como vazia. Se a exclusão ocorrer normalmente, retorna 0. Se a chave `k` não estiver na tabela, não faz nada e retorna 1.

- `void imprime_chash(struct Hash *hash);` – Imprime as chaves armazenadas na tabela em ordem crescente, juntamente com a tabela na qual a chave `k` se encontra (`T1` ou `T2`), seguido pela posição da chave na tabela (inteiro `i` $\in [0..10]$). A saída segue o formato `k,T[1|2],i`.

  OBS: Para realizar a impressão em ordem, foi usada a função qsort, da biblioteca padrão da linguagem C. Para tanto, foi desenvolvida uma nova estrutura, `Entrada_id`:

  ```
  struct Entrada_id {
      int chave, vazia, tabela, pos;
  };
  ```

  Dessa forma, é criado um vetor com as entradas das duas tabelas, e esse vetor é ordenado pelas chaves das estruturas. Depois, as entradas não vazias são impressas seguindo o formato acima.

3 Decisão pelos campos definidos na estrutura `Entrada`

Inicialmente, a estrutura `Entrada` continha três campos: `chave`, `vazia` e `excluida`. Esse último servia para determinar se uma entrada havia sido excluída após ser inserida. Isso porque, na função de busca, se a posição de uma chave na tabela 1 estivesse vazia, a função retornava -1 imediatamente, mas pode ser necessário checar a tabela 2 também.

Porém, esse campo só era útil na tabela 1, e seria inutilizado na tabela 2; além disso, essa característica só faz diferença no comportamento da função de busca considerando as primeiras vezes que a tabela é utilizada: após várias inserções e remoções, todas as posições da tabela 1 serão marcadas como não-vazias, e a função checará a tabela 2 de qualquer maneira.

Então, em lugar de criar duas estruturas diferentes (uma para a tabela 1, com os três campos, e outra para a tabela 2, com apenas dois), e considerando que a função de busca não perde grande desempenho checando as duas tabelas em toda execução, foi decidido usar apenas os campos `chave` e `vazia`. Isso torna a implementação mais simples e ajuda na legibilidade do código.