Sobre o projeto

O projeto consiste em uma demonstração e manipulação de um **TAD de tabela hash**.

Esta tabela hash possui funções de manipulação, e é uma abstração do **Cuckoo Hashing**, ou seja consiste de duas tabelas com funções hash para chaves de indexação diferentes.

A tabela confere colisoes de inserir na primeira sub tabela Hash e empurra elementos da primeira para segunda. Ou seja, **SEMPRE** insere na primeira sub tabela, no entanto, **não é verificado a dupla colisão**, portanto essa implementação pode gerar perda de dados por dupla colisão (tanto na primeira e segunda sub tabela) e também não realiza o "rehashing", comumente feito no Cuckoo Hashing.

Sobre a entrega

- Feito por Gustavo Benitez Frehse (GRR20235087)
- Informática Biomédica
- Matéria: Algoritmos e Estrutura de Dados III

Uso do programa

Compilando

```
# Compilando na raiz
make;

# Usando o programa
./myht
```

Usando

- Existe duas principais operações:
  - `i n` Inserir um valor N na tabela
  - `r n` Deletar um valor N da tabela
- Ao fim do uso ou ao inserir uma operação inválida, o programa imprimira a tabela hash de forma ordenada demonstrando a posição, número e qual tabela cada dados está inserido.

Funcionalidades

- Tamanho estatico da hash: `HASH_TABLE_SIZE` com valor setado em 11

Estrutura de dados em hash.c

- Estrutura de dados para armazenamento da hash (tabela 01 e tabela 02) - `Hash_t`
- Estrutura de dados para armazenamento de cada valor da hash - `__hash_number`
  - Possui valor e bits de manipulacao (tipo deletado ou inicializado)

Implementações na hash.c (TAD puro da tabela Hash):

- `size_t h1(int key)` - função que faz o **cálculo de index (função hash 1)** para a tabela 01
- `size_t h2(int key)` - função que faz o **cálculo de index (função hash 2)** para a tabela 02
- `Hash_t create_hash_table()` - função que cria e retorna uma **tabela hash** estática zerada
- `int search_hash_table(Hash_t t, int k)` - função que **retorna qual tabela** está a chave procurada
- `void insert_hash_table(Hash_t *t, int k)` - **insere um valor** na tabela hash
- `int delete_hash_table(Hash_t *t, int k)` - **deleta um valor** na tabela hash e retorna sua posicao
- `void print_hash_table(Hash_t t)` - **imprime em `stdout` de forma ordenada** a tabela hash

Conclusão

A tabela Hash é muito eficiente para armazenamento, busca e deleção de dados, no entanto é perceptível, principalmente em sua implementação a sua falha em relação a ordenação de dados, que necessita uma iteração de ordenação antes de impressão de dados, o que não é visto em árvores BST.

Além disso, foi perceptível a facilidade e rapidez para fazer uma implementação desta árvore, sem se preocupar com rehashing. Também é visto o quão bom ela é para a memória estática (neste projeto foi implementado desta forma), não necessitado de uma alocação dinâmica de memória para seu uso, comumente visto em diferentes árvores de busca binária. Fora o otimizado tempo de busca e inserção do TAD.

No entanto, principalmente para resumir, são encontrados estes **problemas pontuais no Hash**:

- Alto custo para ordenação (que precisa ordenar do zero os dados)
- Necessidade de dois bits adicionais: inicializado e bit de deleção
- Esta implementação, por não ter hashing, pode gerar perda de dados em colisão dupla
- Desperdicio de espaço por falta de uso (visto a inicial alocação de memória no uso)