Laboratório 9: Tabela de Dispersão

Entrega até domingo, 26/5, às 23:59h

Em sala, discutimos a implementação de uma tabela de dispersão, onde cada posição do array guarda a chave correspondente, o dado associado, e um índice para a próxima chave que gerou uma colisão, pois retorna a mesma posição de acordo com a função de dispersão utilizada, sendo -1 caso não exista uma próxima chave.

A estrutura da tabela de dispersão é dada a seguir:

```
typedef struct {
  int chave;
  int dados;
  int prox;
} ttabpos;

struct smapa {
  int tam;
  int ocupadas;
  ttabpos *tabpos;
};
```

Onde o mapa é uma estrutura que contém o tamanho da tabela, a quantidade de posições ocupadas, e a tabela propriamente dita contendo os nós. Na implementação desse trabalho, deve-se tratar as colisões de maneira semelhante ao que foi visto em sala de aula.

Dois testes foram gerados para você testar a implementação feita. O **testeinterativo.c** permite adicionar, buscar e remover elementos na ordem que desejar, enquanto o arquivo **teste.c** possui um conjunto predefinido de operações chaves.

A partir dos arquivos inicialmente disponibilizado, faça as seguintes tarefas:

1. (3.0) Implemente a função **insere**, inicialmente sem redimensionamento, e inicialmente considerando que toda chave a ser inserida **NÃO** está no mapa. O esqueleto de implementação está criando tabelas com **11** posições. A função insere deve seguir o seguinte algoritmo:
    a. Calcula ***pos*** fazendo o ***hash*** da chave modulo o tamanho da tabela;
    b. Caso ***pos*** esteja livre, realiza inserção e retorna. Caso contrário, procura por ***poslivre***, uma próxima posição livre na tabela.
    c. Caso a chave em ***pos*** tenha o mesmo valor de ***hash*** que a chave a inserir, estamos em um conflito primário: insere a nova chave em ***poslivre*** e encadeia ***poslivre*** na lista atual, como prox de ***pos*** (nova chave sempre será inserida na segunda posição do encadeamento).

d. Caso a chave em ***pos*** tenha valor diferente de ***hash*** que a chave a inserir, estamos em um conflito secundário: move a chave em ***pos*** para ***poslivre*** e corrige a lista encadeada referente aos conflitos deste outro valor de ***hash***. Ao final, insere o valor em ***pos***.

Teste sua implementação usando o programa de testes dado. A função de ***hash*** usada inicialmente gera conflitos para todos os números iguais módulo o tamanho da tabela (definido inicialmente como 11). Use isto para gerar conflitos propositalmente.

2. (2.0) Implemente o redimensionamento da tabela (função ***redimensiona***), chamado quando a tabela atinge 75% de ocupação. Para "liberar" a compilação da função ***redimensiona***, mude de 0 para 1 as linhas do arquivo **mapa.c**

```
#if 0
#endif
```

3. (2.0) Implemente a função de **busca**, e modifique a função **insere** para verificar se a chave que deve ser inserida já existe na tabela, antes de realizar a inserção. Se já existir, a inserção deve ser cancelada.
4. (3.0) Implemente a função de **retirada**. Ao achar a chave que deve ser removida, considere 2 casos de remoção:
    a. A chave é a primeira do encadeamento de chaves com mesmo valor da função de dispersão. Para realizar a remoção, deve-se mover o conteúdo de ***prox*** para ***pos***, e resetar o conteúdo na posição ***prox***. Lembre-se que ***prox*** pode não existir. Nesse caso, apenas o conteúdo de ***pos*** deve ser resetado.
    b. A chave está no meio ou no final do encadeamento, Nesse caso, deve-se remover o conteúdo na posição ***pos***, e corrigir o encadeamento.

Faça upload dos arquivos **mapa.c** no EAD até dia 26 de maio, domingo, às 23:59h. Lembre-se de fazer a entrega mesmo que não tenha chegado ao final do exercício.