lista 09

Questão 1 (Tabela de Dispersão)**.** Considere o seguinte fragmento de código que define uma classe que é uma tabela de dispersão que usa encadeamento externo para o tratamento de colisões. Complete os métodos de inserção, busca, remoção (apenas retira da tabela sem liberar o espaço) e eliminação (remove da tabela e libera o espaço).

```
#include <iostream>

class NoRegistro {
    public:
        int chave;
        NoRegistro* prox;
};

class TabelaDispersao_EE_Divisao {
    public:
        NoRegistro** tabela;
        int m;   //tamanho da tabela
        int cont;   //número de elementos na tabela

        // método construtor, inicializa a tabela
        TabelaDispersao_EE_Divisao(int tamanho){
            m = tamanho;
            tabela = new NoRegistro*[m]();
            cont = 0;
        }

        int metodo_divisao(int c){
            ...
        }

        // método inserir
        int inserir(NoRegistro* reg){
            ...
        }

        // método buscar
        NoRegistro* buscar(int chave){
            ...
        }

        // método remover, devolve um ponteiro para o nó removido
        // não libera o espaço do nó
        NoRegistro* remover(int chave){
            ...
        }

        // método eliminar, remove o nó liberando o espaço
        void eliminar(int chave){
            ...
        }

        // método destrutor, libera espaço da tabela sem liberar o espaço dos nós
        ~TabelaDispersao_EE_Multiplicacao(){
            ...
        }
};
```

Questão 2 (Método da Multiplicação)**.** Para utilizar outra função de dispersão, basta alterar a função de dispersão. Implemente a classe abaixo que é identica à anterior, exceto pela função de dispersão, que utiliza o método da multiplicação. Modifique a classe anterior para implementar o método da multiplicação (veja a nota de aula para uma explicação do método da multiplicação).

```
#include <iostream>

class NoRegistro {
    public:
        int chave;
        NoRegistro* prox;
};

class TabelaDispersao_EE_Multiplicacao {
    public:
        NoRegistro** tabela;
        int m;   //tamanho da tabela
        int cont;   //número de elementos na tabela

        // método construtor, inicializa a tabela
        TabelaDispersao_EE_Divisao(int tamanho){
            m = tamanho;
            tabela = new NoRegistro*[m]();
            cont = 0;
        }

        int metodo_multiplicacao(int c){
            ...
        }

        // método inserir
        int inserir(NoRegistro* reg){
            ...
        }

        // método buscar
        NoRegistro* buscar(int chave){
            ...
        }

        // método remover, devolve um ponteiro para o nó removido
        // não libera o espaço do nó
        NoRegistro* remover(int chave){
            ...
        }

        // método eliminar, remove o nó liberando o espaço
        void eliminar(int chave){
            ...
        }

        // método destrutor, libera espaço da tabela sem liberar o espaço dos nós
        ~TabelaDispersao_EE_Multiplicacao(){
            ...
        }
};
```

Questão 3 (Velha lista ordenada, agora com ponteiros)**.** Podemos implementar uma lista ordenada usando um vetor de ponteiros. Cada posição do vetor será um ponteiro para um (apenas 1) registro. Complete a classe a seguir que implementa uma lista ordenada usando ponteiros.

```
#include <iostream>

class Registro {
    public:
        int chave;
        int dado;
};

class ListaOrdenada {
    public:
        Registro** L;   //vetor de ponteiros de registros
        int tam_max;   //tamanho máximo da lista
        int cont;   //número de elementos na lista

        // método construtor, inicializa a lista
        ListaOrdenada(int tamanho){
            tam_max = tamanho;
            L = new Registro*[tam_max]();
            cont = 0;
        }

        // método inserir, retorna a posição do registro inserido no vetor
        int inserir(Registro* reg){
            ...
        }

        // método buscar, implementa busca binária
        Registro buscar(int chave){
            ...
        }

        // método remover, retira da lista e devolve ponteiro para o registro removido
        Registro remover(int chave){
            ...
        }
};
```

Questão 4 (Performance)**.** Vamos tentar avaliar a performance das nossas estruturas em relação às operações de inserção, busca e remoção.

Na linguagem C++ existe a função `clock()` da biblioteca `ctime` que retorna o tempo **aproximado** decorrido desde o início do programa em uma certa unidade de tempo (*clock ticks*). A constante `CLOCKS_PER_SEC` diz quantas dessa unidades há em 1 segundo. Se quisermos saber o tempo **aproximado** em segundos cosumidos durante a execução de certo trecho de código, podemos fazer assim.

```
#include <ctime>

...

int inicio, fim;

inicio = clock();

// código do qual se deseja estimar o tempo de execução
...

fim = clock();

int duracao = fim - inicio;

float duracao_ms = (float) 1000*duracao/CLOCKS_PER_SEC;
```

Considere o seguinte trecho de código:

```
#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

class NoRegistro { ... };

class TabelaDispersao_EE_Divisao { ... };

//realiza n inserções em uma tabela de tamanho m e retorna o tempo decorrrido
//os registros que serão inseridos estão no vetor Regs
int testeInsercao_TDEED(int m, int n, NoRegistro* Regs) {

    //cria tabela
    TabelaDispersao_EE_Divisao TDEED(m);

    //registra inicio das inserções
    int inicio, fim;

    //registra início do procedimento de inserção
    inicio = clock();

    //insere os nós na tabela
    for (int i = 0; i < n; i++){
        TDEED.inserir(&Regs[i]);
    }

    //registra o final das inserções
    fim = clock();

    return fim - inicio;
```

```
}

void inicializaChavesAleatorias(int n, NoRegistro* Regs){

    //inicializa os registros com chaves aleatórias não repetidas
    srand(time(0));

    for (int i = 0; i < n; i++){
        Regs[i].chave = rand();
    }
}

int main(){

    int n = 10000; //número de registros a serem inseridos
    int m = 5000;  //tamanho da tabela de dispersão

    //cria registros a serem inseridos nas respectivas estruturas
    NoRegistro* Regs = new NoRegistro[n];

    //inicializa chaves aleatórias
    inicializaChavesAleatorias(n, Regs);

    //testa tabela de dispersão com encadeamento externo e método da divisão
    int duracaoTDEED = testeInsercao_TDEED(m, n, Regs);

    cout << "Duração do teste de insersão\n";
    cout << "Tamanho da tabela: " << m << "\n";
    cout << "Número de registros: " << n << "\n";
    cout << "Duração: " << (float) 1000*duracaoTDEED/CLOCKS_PER_SEC << "ms\n";

}
```

O código acima implementa um método

`testeInsercao_TDEED(int m, int n, NoRegistro* Regs)`

que cria uma tabela de dispersão de tamanho `m` com encadeamento externo e usando o método da divisão e realiza a inserção de `n` registros que são passados através do ponteiro `Regs` que aponta para um vetor de registros alocado dinamicamente e com chaves aleatórias.

Complete o código acima com o código das classes `NoResgistro` e `TabelaDispersao_EE_Divisao` que você fez nas questões anteriores e execute o código para estimar o tempo gasto. Varie os valores de `m` e `n` para ver o que acontece.

Questão 5 (Performance II)**.** Adapte e repita o teste de performance acima para a tabela de dispersão usando o método da multiplicação e para a lista ordenada.

Questão 6 (Performance III)**.** Adapte e repita o teste de performance para as 3 estruturas (tabela de dispersão com o método da divisão, tabela de dispersão com o método da multiplicação e lista ordenada) para as operações de remoção e busca.

Note que no caso da lista ordenada, o tamanho do vetor deve ser pelo menos o mesmo número de registros que serão inseridos.

Escolha vários valores de `m` e `n` e monte uma tabela comparando os métodos.