ATENÇÃO: É requisito para a execução deste programa, que o arquivo de entrada results.csv esteja em codificação single-byte (ANSI, ISO8859..). O programa não está preparado para leitura de arquivo double byte (tal como o TF8). Se necessário converter, múltiplos programa o fazem, como por exemplo o Notepad++ no menu Enconding. Ou em Linux, o programa iconv. Assim, no ZIP deste programa, inclui um results.csv já no formato que funcionara.

1. Introdução

Neste trabalho prático, buscamos criar um algoritmo que realize uma pesquisa a fim de extrair informações especificadas (ranking em copas do mundo, número confrontos) de um grande volume de dados contendo o histórico completo de jogos entre seleções nacionais de futebol entre os anos de 1872 e 2018.

A fim de atender o primeiro método especificado (Inserção em O(1) e pesquisa em O(n)), optei pelo uso de listas encadeadas. A fim de atender o segundo método (Inserção e pesquisa em O(log n)) optei pelo uso de uma arvore binaria de pesquisa. Já para o terceiro e último método (Inserção e pesquisa em O(1)), escolhi utilizar funções hash.

2. Implementação

Após ler a proposta do trabalho, optei por fazer uma implementação que a priori independe de qual o método de estruturas de dados a ser utilizado (de acordo com a entrada). Isto é, em vista de que as operações a serem realizas nos tres são, em termos de alto nível, as mesmas, (tal como, pesquise, remova, adicione e etc.) escrevi o código de tal forma que o algoritmo principal utilizasse ponteiros para funções em seus loop principais e que então o programa fizesse uma atribuição à esses ponteiros para as funções do método escolhido na entrada.

**Estruturas de Dados:**

Para possibilitar a ideia descrita na introdução deste documento, utilizando tres módulos objeto. Um deles encapsula todas as estruturas e operações necessárias para a implementação de uma lista encadeada (PrepCopaMundoLista.c/.h). A segunda, encapsula as estruturas de operações necessárias para a implementação do algoritmo utilizando arvores binarias de busca (PrepCopaMundoArvBinPesquisa). E por fim, o terceiro TAD encapsula tudo o que foi necessário para implementar o algoritmo utilizando hash.

Em linha com esta organização, os .h contém apenas as declarações necessárias para a comunicação entre esses módulos. A criação destes tres TADs permitiu uma modularização de todas as operações realizadas. Isto é, cada uma dos tres TADs funcionam como módulos de funções a serem chamados pelo arquivo principal do programa de acordo com a entrada.

* **TAD Lista:** Aqui, fez-se necessária a criação de uma struct de células da lista (NodeLista) composto por uma chave, um registro e um pointer para tanto a célula anterior quanto para a próxima a fim de implementar uma lista duplamente encadeada. Aqui estão presentes também as operações típicas de lista, inserção, remoção e busca. Neste método optei pelo uso do mergesort para realizar a ordenação na hora de gerar a saída final do ranking
* **TAD Arvore Binaria de Pesquisa:** Aqui foi necessária a criação de uma struct para compor os vértices da arvore binaria a ser implementada. A árvore binaria foi implementada utilizando alocação dinâmica, reservando em memoria apenas quando a necessidade surgisse ao invés de alocar um grande espaço de memória limitado a um tamanho fixo. Como explicitado em sala, não preocupei com o balanceamento da arvore;
* **TAD Hash:** Implementado com um array de 10 mil posições de ponteiros para as tuplas de hash. Ele utiliza a conhecida função de hash djb2 (http://www.cse.yorku.ca/~oz/hash.html). Utilizei endereçamento aberto para resolver problemas de colisão.

**Funções e Procedimentos:**

**TAD Lista**

* **ListaInsere**( void \* PPInicio, char \* NovaChave, void \* NovoRegistro ): Dada uma chave, a insere na próxima posicao disponivel da lista. Retorna um inteiro correspondente ao status de sucesso ou não da inserção
* **ListaRemove**(void \* PPInicio ): Remove o primeiro elemento da lista e retorna um ponteiro para o registro removido (por exemplo, estatística de pais ou de confronto)
* **ListaBusca**(void \* PInicio, const char \* Chave ):Dada uma chave, verifica sua existência na lista e retorna um ponteiro para o registro encontrado
* **Divide**(NodeLista \*InicioCast):Método da divisão de regiões do mergesort e retorna um ponteiro para o início da sua região da lista
* **Merge**(NodeLista \*Bloco1, NodeLista \*Bloco2) :Método que mescla o conteúdo de duas regiões definidas pelo método merge a fim de aplicar o algoritmo de mergesort. Retorna um ponteiro para o início de sua região na lista
* **MergeSort**( NodeLista \* InicioCast ) :Programa principal do método mergesort. Selecionado para ordenação de uma lista duplamente encadeada. Retorna um ponteiro para o início de sua região na lista
* **ListaOrdena**( void \* PPInicio,int (\*EscolhePrimeiro)(void\*,void\*) ):Método para chamada externa pelo modulo usuário para ordenação na lista

**TAD Arvore Binaria de Pesquisa**

* **ArvBinPesqInsere**(void \* PPInicio , char \* NovaChave , void \* NovoRegistro ) :Função de interface da inserção de dados na arvore binaria para ativação por programas externos. Faz a conversão de ponteiros void para ponteiros para a estrutura de dados. Retorna o status da operação
* **ArvBinPesqBuscaInterno**( NodeABP\* InicioCast, char \* Chave ):Função que verifica a existência de uma chave na arvore binario, caso encontrado retorna o registro associado a chave
* **ArvBinPesqBusca**( void \* PInicio , char \* Chave ): Função de interface do método de busca para ativação por componentes externos. Faz a conversão de ponteiros genéricos para ponteiros da estrutura de dados da arvore correspondente
* **ArvBinPesqRemoveEmOrdemInterno**( NodeABP\* InicioCast ) :Função que remove o primeiro node em in-order da arvore binaria de busca e retorna um ponteiro para o node removido
* **ArvBinPesqRemoveEmOrdem** ( void \* PPInicio ) : Função de interface da retirada em in-order, retorna um ponteiro para o node removido.
* **(\*EscolhePrimeiroArvore )**(void\*,void\*): Interface da função a ser passada pelo componente usuário que seleciona entre os registros de dois nodes o que vem primeiro na ordem desejada pelo usuário
* **ArvBinPesqInsereNewOrder**(NodeABP \* InicioCast, NodeABP \* NovoNo ):Insere um node na arvore binaria de busca de acordo com o critério de ordenação passado pelo usuário, retorna um pinteiro para a struct node
* **ArvBinPesqOrdena**( void \* PPInicio , int (\*EscolhePrimeiro)(void\*,void\*):Função que reorganiza a arvore binaria de acordo com o critério de ordenação passado na função EscolhePrimeiroArvore

**TAD Hash**

* **CodigoHash**( const char \* Chave ):Função que gera uma posição no array hash a partir da chave informada, retorna a posição na tabela na qual a chave devera ser inserida
* **HashInsere** ( void \* PPInicio , char \* NovaChave , void \* NovoRegistro ):Insere a chave e o registro na tabela hash. Retorna status da inserção.
* **HashBusca**( void \* PInicio , const char \* Chave ):Verifica a existência de um registro com a chave informada na tabela hash. Se encontrada, retorna um ponteiro para o registro associado à chave
* **HashRemove**( void \* PPInicio ):Remove a próxima entrada existente na tabela hash. Retorna um ponteiro para o registro removido
* **HashOrdena**( void \* PPInicio , int (\*EscolhePrimeiro)(void\*,void\*):Reorganiza os pares chave e registro na tabela hash de acordo com a função EscolhePrimeiro passada pelo usuário. Retorna um ponteiro para o primeiro node da lista ordenada

3. Estudo de Complexidade

**Complexidade ->TAD Lista:**

* **ListaInsere:** Tem complexidade O(1) em vista de que é inteiramente composto por atribuições e comparações ao nó no início da lista, sendo desnecessário qualquer tipo de navegação.
* **ListaBusca:** Tem complexidade O(n) pois no pior caso vai passar por todos os elementos da lista
* **ListaRemove:** Tem complexidade O(1) em vista de que é inteiramente composto por atribuições e comparações que não estão em loops e, portanto, apenas executam uma única vez.
* **ListaOrdena:** Implementado utilizando MergeSort, e, portanto, de complexidade O(log n) em todos os casos.

**Complexidade ->TAD Arvore Binaria de Pesquisa**

* **ArvBinPesqInsere:** Tem complexidade O(log N) pois chama a função **ArvBinPesqBuscaInterno** que por sua vez é chamada recursivamente para as subarvores do node em analise, de forma que na pior das hipóteses, passaremos por log n nodes antes de encontrar a posição correta de inserção
* **ArvBinPesqBusca:** Tem complexidade O(log N) pois chama a função **ArvBinPesqBuscaInterno** que por sua vez é chamada recursivamente para as subarvores do node em analise, de forma que na pior das hipóteses, passaremos por log n nodes antes de obter um resultado para a pesquisa
* **ArvBinPesqRemoveEmOrdem:** Tem complexidade O(log N) pois chama a função **ArvBinPesqRemoveEmOrdemInterno** que por sua vez é chamada recursivamente até que encontre, ou não (caso tenha feito a pesquisa até o fim sem encontrar), o node a ser removido.
* **ArvBinPesqOrdena:** Tem complexidade O(n\*log n) pois realiza a chamada das seguintes funções O(log n) para cada elemento da arvore binaria
  + **ArvBinPesqInsereNewOrder:** Tem complexidade O(lon n) pois chama a si mesmo para o node à esquerda ou à direita do node sendo avaliado. Isto significa que na pior das hipóteses teremos passado por log n nodes.
  + **ArvBinPesqRemoveEmOrdemInterno:** analisado no bullet acima

**Complexidade ->TAD Hash**

**HashInsere:** No caso médio tem comportamento assintótico O(1) apesar de utilizar endereçamento aperto pois utilizei uma tabela hash com muito mais posições do que o número de entradas serão recebidas. Além disso utilizei uma função hash extremamente sensível à variação nas chaves e gerando uma distribuição homogênea das entradas.

**HashBusca:** Tem complexidade O(1) pelos mesmos motivos da função acima. Para encontrar uma chave basta rodar a função hash a utilizando como parâmetro e então, no caso de resultado negativo, analisar as posições imediatamente seguintes à posição encontrada pela função hash (endereçamento aberto)

**HashRemove:** Tem complexidade O(1) pois funciona de maneira análoga à função acima, analisando apenas as posições seguintes àquela encontrada pela função hash para a chave que se deseja remover.

**HashOrdena:** Tem complexidade O(n\*log n) pois executa as operações de preparação de uma arvore binária de busca(O(n\*log n )) utilizando a função **ArvBinPesqInsere** cuja complexidade foi analisada na seção TAD Arvore Binaria de Busca. Reorganização dos nodes da arvore pelo critério de ranking (O(n\*log n )) utilizando a função **ArvBinPesqOrdena**, também já analisada e extração de cada um dos elementos “in-order” (O(n\*log n )) com a função **ArvBinPesqRemoveEmOrdem** já analisada.

**Complexidade->main**

Até a primeira chamada de função (ResultadosCarrega) temos apenas operações de atribuição e comparação, ou seja, O(1). Temos então uma sequência de chamadas de funções que juntas implementam o algoritmo

* Ordena: apenas faz a chamada das duas funções a seguir
  + ColecaoOrdena: variável do tipo função que pode receber diferente funções de ordenação de acordo com o metodo escolhido pelo usuário
    - HashOrdena: já analisado
    - ListaOrdena: já analisado
    - ArvBinPesqOrdena: já analisado
* RankingGera: tem complexidade O(n\*log n), pois para cada país, ele chama a função ColeçãoOrdena, que pode assumir a forma de um dos tres métodos de ordenação já analisados entre os quais a pior performance é o da ABB, que tem sua função ordena em ordem O(log n);
* ConfrontosGera: tem complexidade O(n\*log n), pois para cada país, ele chama a função ColeçãoRemove, que pode assumir a forma de um dos tres métodos de remoção já analisados entre os quais a pior performance é o da ABB, que tem sua função de remoção em ordem O(log n);

Comparações do tempo de execução

A comparação foi feita no caso em que todos os jogos são incluídos(Comando de chamada: time ./tp2 results.csv 1/2/3). Cada método foi executado 25 vezes alternadamente (para efeitos de cache). Os valores a seguir são as médias dos tempos obtidos empiricamente.

|  |  |  |
| --- | --- | --- |
| Método 1(Lista) | Método 2 (ABB) | Método 3 (Hash) |
| 4.504 | 0.557 | 0.550 |

5. Bibliografia

*[CLRS3]  T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein,*Introduction to Algorithms*, 3rd edition,  MIT Press, 2009.*

*Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage, 2017*

*http://www.cse.yorku.ca/~oz/hash.html*