TP3 - Documentação

Nome: Rodrigo Ferreira Araújo — Matrícula: 2020006990

Setembro 2021

1 Introdução

Esta documentação trata o problema da organização de nomes e dados binários em uma Árvore Binária de Pesquisa, de modo que os registros chegam linearmente e precisam ser pesquisados, inseridos, removidos adequadamente. Desse modo, com a nova estrutura de dados, visamos obter operações mais eficientes.

Nesse sentido, sob o contexto de download e envio de mentes para corpos sintéticos por meio de servidores da Reallocator CO, o objetivo desta documentação é descrever a implementação do armazenamento e operacionalização em uma Árvore Binária de Pesquisa das consciências, arranjadas individualmente em um campo "nome" e um campo "dados" (número(s) binário que identifica(m) a consciência).

Para a resolução do problema descrito, foi implementado uma Árvore Binária de Pesquisa, em que cada nó contém um registro de consciência, de modo que o nome será a chave a ser comparada (sob a ordem alfabética) nas operações e uma Lista Encadeada com o(s) dado(s) possíveis relacionados à um determinado nome. Desejamos inserir novos registros; inserir um dado à lista encadeada caso o nome de entrada já esteja presente; pesquisar um registro específico, somar todos os seus dados contidos na lista (inteiros decimais), retorná-los à saída e removê-los da árvore; e, por fim, imprimir os nomes da árvore InOrdem. A linguagem de programação usada é C++.

2 Método/Análise de Complexidade

2.1 headers.hpp

Este arquivo contém todos os cabeçalhos das funções implementadas no TP, contidos em classes de acordo com sua estrutura de dados. Ademais, há a definição de um tipo de dado string "TipoDado", que será usado para tipificar as strings de dados binários. Note que tanto os dados quantos os nomes permanecerão como strings ao longo da implementação, uma vez que os dados binários serão convertidos para inteiros apenas quando necessário e a comparação dos nomes quanto à ordem lexicográfica é nativa aos operadores $>$, $<$, $==$ em C++.

2.1.1 Classe TipoCelula

Define uma célula de uma lista encadeada de dados binários de um registro. Contém um atributo dado eu um ponteiro para a próxima célula, determinando um encadeamento simples.

2.1.2 Classe ListaEncadeada

Define a implementação da Lista Encadeada. Contém um atributo para o tamanho, células "primeiro" (entendido como nó cabeça) e "último" e métodos para inserção ao final da lista e para limpar a lista.

2.1.3 Classe TipoNo

Esta classe estrutura o nó da árvore binária de registros. Cada nó possui a chave "nome" (string), a lista encadeada dos dados binários ("dados") e ponteiros para os nós para os filhos da esquerda e direita.

2.1.4 Classe ArvoreBinaria

Classe principal da Árvore Binária de Pesquisa. Possui métodos para inserção, remoção e impressão de nomes InOrdem adequadas para a árvore binária requerida. Possui dois atributos: um nó raiz e uma variável booleana "primeiro_ja_impresso", que, como sugere o nome, checa se o primeiro nome durante a impressão InOrdem já foi impresso. Isto será útil para evitar espaços em branco ao fim da impressão dos nomes.

2.2 headers.cpp

Este arquivo contém as implementações de todas as funções usadas no programa principal ("main.cpp") e métodos necessários para a realização das operações desejadas, organizados nas classes descritas anteriormente.

2.2.1 TipoCelula::TipoCelula

Construtor da classe TipoCelula: inicializa o dado com uma string vazia e o ponteiro para a próxima célula como nulo. A complexidade de tempo e espaço são O(1).

2.2.2 ListaEncadeada::ListaEncadeada

Construtor da classe ListaEncadeada: inicializa o tamanho com 0, inicializa o ponteiro para a célula cabeça e atribui à ela o valor de última. A complexidade de tempo e espaço são O(1).

2.2.3 ListaEncadeada::~ListaEncadeada

Destrutor da classe ListaEncadeada: cria um ponteiro "p" auxiliar na posição da celula após a célula cabeça e, enquanto esse nó não aponta para um NULL, a célula após a cabeça é movida para frente; "p" é deletado e movido para frente. Por fim, o tamanho recebe 0 e a célula cabeça é deletada. A complexidade de tempo desse procedimento é O(k), onde k é o número de dados binários pertencentes à um nome, que pode crescer indefinidamente com os inputs de registros. A complexidade de espaço é O(1), pois apenas um ponteiro auxiliar é criado.

2.2.4 ListaEncadeada::InsereFinal

Procedimento para padronizar a inserção de dados binários à lista encadeada. Nesse caso, a inserção é feita ao final criando-se uma célula auxiliar com o dado enviado, colocando-a na célula após a última da lista e atualizando o ponteiro da última célula para esta. A complexidade de tempo e espaço são O(1), pois apenas uma célula auxiliar é criada e uma célula da lista é manipulada.

2.2.5 ListaEncadeada::SomaDados

Método que soma todos os valores binários, convertidos para decimal, presentes numa lista encadeada. Nesse sentido, todas as células da lista são percorridas e, por meio da função **stoi** da linguagem C++ adaptada para conversão direta de uma string de um número binário para o inteiro correspondente, o valor do inteiro do somatório é retornado pela variável "res". A complexidade de tempo é O(k), onde k é a quantidade de dados binários pertencentes à um nome (semelhante ao destrutor). Para a função **stoi**, a complexidade de tempo e espaço seria constante (O(1)), pois o tamanho da string do dado binário é fixo, o que não afeta a complexidade total de O(k). A complexidade de espaço é O(1), pois apenas um ponteiro célula auxiliar é criado para percorrer a lista.

2.2.6 TipoNo::TipoNo

Contrutor da classe TipoNo. Inicializa a lista encadeada caractuerística do nó da árvore binária dinamicamente (operador **new**) e instancia os nós filhos com **NULL**. A complexidade de tempo e espaço são O(1).

2.2.7 ArvoreBinaria::ArvoreBinaria

Construtor da Árvore Binária: seta a variável "primeiro_ja_impresso" como **false** e inicializa o ponteiro para o nó raiz como **NULL**. A complexidade de tempo e espaço são O(1).

2.2.8 ArvoreBinaria::ApagaRecursivo

Método recursivo usado no destrutor da classe. Recebe a raiz como parâmetro (via chamada da função no destrutor da classe) e deleta Pós-Ordem os ponteiros para os nós. A complexidade de tempo é O(n), onde n é a quantidade de nós da árvore, uma vez que todos os n nós são visitados para a deleção. A complexidade de espaço é O(n), devido às chamadas recursivas.

2.2.9 ArvoreBinaria::∼ArvoreBinaria

Destrutor da classe ArvoreBinaria: chama o processo "**ApagaRecursivo**" descrito com o nó raiz e, por fim, seta o ponteiro para o nó raiz como **NULL**. Complexidade de tempo e espaço: idem "**ApagaRecursivo**".

2.2.10 ArvoreBinaria::InsereRecursivo

Procedimento recursivo para inserção de nós na árvore, tomando os nomes como chaves. Percorre a árvore a partir da raiz e, enquanto não encontra um nó vazio ou um nó com o mesmo nome, realiza chamadas recursivas para o nó esquerdo (nome é menor na ordem lexicográfica) ou para o nó direito (nome é menor na ordem lexicográfica). Caso o nome seja igual, apenas adiciona o dado ao final da lista encadeada do nó, caso o nó seja nulo, cria um novo registro.

A complexidade de tempo é O(h), onde **h** é a altura da árvore (níveis). Para um número **n** suficientemente grande de nós da árvore, h tende a **log n**, ou seja, o caso médio seria O(log n), mas, caso os nomes sejam inseridos em ordem alfabética, teremos uma altura igual ao número de nós, o que gera a complexidade do pior caso: O(n). Para o melhor caso, temos O(log n), pois é o caso de uma árvore binária balanceada. A complexidade de espaço segue o padrão da complexidade temporal, dado as chamadas recursivas.

2.2.11 ArvoreBinaria::Insere

Procedimento usado no programa principal que recebe o nome e o dado do arquivo de entrada e chama o procedimento descrito acima ("**InsereRecursivo**") com o nó raiz. Complexidade de tempo e espaço: idem "**InsereRecursivo**".

2.2.12 ArvoreBinaria::Antecessor

Procedimento usado na remoção de um nó da árvore, que procura um antecessor (filho mais à direita da subárvore da esquerda) para substituir um nó com 2 filhos que está para ser removido. A complexidade de tempo é O(a), onde "**a**" é a altura da subárvore da esquerda (Obs: **a** < quantidade de nós da subárvore da esquerda < quantidade de nós da árvore). A complexidade de espaço é O(1).

2.2.13 ArvoreBinaria::RemoveRecursivo

Procedimento recursivo de remoção e impressão adequada do nome e da soma dos dados binários de um nó da árvore. Procura recursivamente até achar o nó com o nome passado por parâmetro, imprime o nome e a soma dos dados e, por fim, trata os casos de remoção (nó com 0 ou 1 filhos, ou 2

filhos). 0 filhos: apenas remove o nó da árvore; 1 filho: sobe o único filho e remove o nó pai; 2 filhos: remove o nó da árvore e o substitui pelo seu antecessor.

A complexidade de tempo segue o padrão do método "**InsereRecursivo**", uma vez que o processo de busca para inserção é semelhante ao processo de busca de uma chave (nome) para remoção. Nesse sentido, temos O(h) $\sim$ O(log n) para o melhor caso e caso médio, onde "**h**" é a altura da árvore e "**n**" é a quantidade de nós na árvore e, por fim, para o pior caso temos complexidade O(n) (**h** = **n**). A complexidade de espaço segue o padrão da complexidade temporal, dado as chamadas recursivas.

2.2.14 ArvoreBinaria::Remove_e_Imprime

Procedimento de remoção e impressão adequada (nome e soma dos dados) do nó usado no programa principal, que recebe o nome lido no arquivo de entrada. Chama o procedimento recursivo descrito anteriormente com o nó raiz da árvore. Complexidade de tempo e espaço: idem "**RemoveRecursivo**".

2.2.15 ArvoreBinaria::InOrdem

Método responsável pela impressão InOrdem dos nomes dos nós da árvore binária. Atualiza a variável "primeiro_ja_impresso" para **true** para que impressões sejam realizadas com um espaço à esquerda, assim, não haverá um espaço em branco ao final da impressão. Complexidade de tempo: O(n), onde **n** é igual ao números de nós da árvore, uma vez que todos os nomes serão impressos, já a complexidade de espaço é O(1).

2.2.16 ArvoreBinaria::Imprime_Arvore

Procedimento usado no programa principal que chama a impressão InOrdem a partir do nó raiz. Complexidade de tempo e espaço: idem "**InOrdem**".

2.2.17 ArvoreBinaria::Reset

Método que inverte o sinal lógico da variável que controla os espaços vazios na impressão. É usado no programa principal após a impressão dos registros removidos. Complexidade de tempo e espaço: O(1).

2.3 main.cpp

Programa diretor do trabalho prático, responsável por ler o arquivo de entrada, criar a árvore binária de acordo com as inserções, imprimir InOrdem os nomes com o procedimento "**Imprime_Arvore**", remover os nós com os nomes passados e, por fim, imprimir novamente de acordo com os nós removidos.

Em primeiro lugar, há a criação da árvore binária e a leitura das **N** linhas contendo os nomes seguidos dos dados, onde **N** é um inteiro lido na primeira linha do arquivo de entrada. Nesse loop de leitura, os nomes e os dados são admitidos separadamente sua inserção tratada com o procedimento "**Insere**".

Em seguida, imprimimos InOrdem os nomes contidos na árvore e o resto do arquivo de entrada é lido, uma vez que apenas restaram linhas com nomes a serem removidos da árvore. Nesse contexto, numa mesma iteração do loop de leitura, o nome é admitido e passado como parâmetro para o método "**Remove_e_Imprime**", que trata o caso de remoção do nó e imprime o nome e a soma dos dados binários para um inteiro.

Por fim, chamamos o procedimento "**Reset**" para imprimir adequadamente a árvore binária depois da remoção dos nós com o procedimento "**Imprime_Arvore**".

3 Conclusões

Com este Trabalho prático, realizamos uma aplicação prática de armazenamento e manipulação de registros (no formato de nomes e lista de binários) em uma Árvore Binária de Pesquisa. Com métodos de inserção, remoção,

impressão e soma de dados binários, tornamos essa árvore binária minimamente funcional e capaz de fornecer um relatório simples sobre os nomes e dados removidos e o status atual da árvore.

Portanto, conseguimos visualizar nessa implementação que a Árvore Binária de Pesquisa é uma estrutura de dados flexível e eficiente como um contêiner de registros padronizados, tal como representa uma alternativa inteligente às listas/filas lineares convencionais para armazenamento de informações sob uma chave identificadora (nome).

4 Bibliografia

References

[1] Luiz Chaimowicz e Raquel Prates. *Universidade Federal de Minas Gerais.* [*Material fornecido pela Disciplina de Estrutura de Dados*]. Belo Horizonte, Brasil.

5 Instruções para Compilação e Execução

Para compilar o programa, execute o comando 'make' no terminal na pasta raiz do TP. Em seguida, para executar o 'run.out' gerado na pasta 'bin', vá para este diretório executando o comando 'cd bin/'. Por fim, rode o programa com o comando: ./run.out <arquivo de texto com as entradas (.txt)>.

Como requerido, o(s) arquivo(s) ".txt" não estará(ão) presente(s) na pasta "bin" para execução, ou seja, antes de executar as instruções acima, inclua o(s) arquivo(s) com as entradas padronizadas na pasta 'bin'. Note que, caso o comando 'make clean' seja rodado, apenas o executável e os arquivos ".o" serão apagados, e não o(s) arquivo(s) ".txt".