Trabalho Prático 3
Estruturas de Dados

Caio Teles Cunha
2020006434
Universidade Federal de Minas Gerais(UFMG)
Belo Horizonte-MG-Brasil
caiot@ufmg.br

1. Introdução

O desafio apresentado foi implementar um programa que recebe um documento de texto com dados e insere esses dados em uma árvore binária de pesquisa. Cada nó desta árvore deve possuir um nome e uma lista encadeada que armazena strings de binários relacionados a cada um dos nomes. Posteriormente, o documento de texto diz quais nomes devem ser tirados da árvore e a soma de todos os dados binários armazenados naquele nó deve ser impressa na tela. Esta documentação tem como objetivo apresentar e explicar o código escrito.

Para ajudar no armazenamento dos dados foi criado uma classe TipoItem com duas strings, nome e dado, entre outras classes necessárias para a implementação da lista encadeada e da arvore binária.

2. Implementação

2.1 Estrutura de Dados

O programa foi implementado tendo como base a estrutura de dados árvore binária de pesquisa. Esta estrutura foi escolhida devido à exigência do trabalho. Uma árvore binária de pesquisa possui nós com ponteiros que apontam para os seus filhos da esquerda e da direita. Todos os filhos à direita do nó pai são maiores que ele, levando em consideração uma chave qualquer armazenada no nó, e todos os filhos à esquerda do nó pai são menores que ele. Cada nó implementado nessa árvore possui um ponteiro para uma FilaEncadeada, outra estrutura de dados utilizada devido às exigências do trabalho, para armazenar os dados binários de cada nome.

2.2 Classes

Foram implementadas várias classes para auxiliar na construção do programa. Uma delas é a classe TipoItem que possui duas strings, chamadas nome e dado, como atributos públicos. Essa classe foi utilizada para armazenar cada linha do documento de texto e foi passada como parâmetro para a maioria dos métodos do código.

A classe TipoCelula possui como atributos uma string dado, implementada para armazenar os dados binários, e um ponteiro para um TipoCelula chamado prox, com o objetivo de encadear células. A classe FilaEncadeada possui três

atributos, um inteiro que armazena o tamanho da fila, e dois ponteiros para TipoCelula, Head e Tail, que apontam, respectivamente, pra primeira célula da fila e para a última célula da lista.

Outra classe implementada foi a TipoNo, ela possui quatro atributos, uma string nome, que será usada como chave nas buscas na árvore binária, dois ponteiros para TipoNo, que apontam para os filhos da esquerda e da direita do nó caso existam, e um ponteiro para FilaEncadeada, para armazenar os dados binários relacionados ao nome. Por fim, foi implementada a classe ArvoreBinaria que possui como atributo apenas um ponteiro para TipoNo que aponta para o nó raiz da árvore.

2.3 Funções e Procedimentos

A classe FilaEncadeada possui os seguintes métodos além do seu construtor:

-void Enfileira(string dado): Essa função acrescenta uma célula com a string dado passada como parâmetro ao final da fila. Ela utiliza o ponteiro Tail da fila encadeada para ir diretamente até a última posição da fila, tornando desnecessário percorrer a fila item por item.

-void Imprime(): Percorre toda a fila, a partir do ponteiro Head, imprimindo as strings armazenadas em cada célula. Foi utilizada basicamente para testes no programa.

-void Limpa(): Percorre toda a fila, a partir do ponteiro Head, liberando o espaço que foi alocado para cada uma das células da lista.

-int SomaBinarios(): Percorre toda a fila, a partir do ponteiro Head, transformando as strings de binários armazenadas em cada célula em inteiros e somando esses valores. Depois retorna o total dos inteiros somados.

A classe ArvoreBinaria possui os seguintes métodos além do seu construtor:

-void Insere(TipoItem item): Simplesmente chama a função InsereRecursivo passando a raiz da árvore e item como parâmetros.

-void InsereRecursivo(TipoNo* &p, TipoItem item): Essa função compara a string nome armazenada em cada nó com a string nome armazenada em item para descobrir em qual lugar o TipoItem deve ficar na árvore. Essa procura é feita recursivamente, caso o nome de item for menor que o nome armazenado no nó a função é chamada novamente passando como parâmetro o nó filho da esquerda e item, caso contrário o nó passado como parâmetro é o filho da direita. Depois de encontrar a posição correta, a função aloca espaço para um novo TipoNo, armazena o nome de item na string nome do nó e chama a função Enfileira para armazenar a string dado de item na FilaEncadeada do nó.

-void InOrdem(TipoNo *p): Essa função percorre a árvore utilizando da recursão para imprimir as strings nome armazenadas em cada nó na ordem central.

-TipoNo *Procura(TipoItem item): Simplesmente chama e retorna o resultado da função ProcuraRecursivo passando como parâmetros a raiz da árvore e item.

-TipoNo *ProcuraRecursivo(TipoNo* &p,TipoItem item): Essa função percorre a árvore utilizando chamadas recursivas e retorna um ponteiro para o nó que possui o mesmo string nome igual à string nome armazenada no item passado como parâmetro. Caso, não possua nenhum nó, retorna um ponteiro nulo.

-void Remove(TipoItem item): Simplesmente chama a função RemoveRecursivo passando como parâmetros a raiz da árvore e item.

-void RemoveRecursivo(TipoNo* &no, TipoItem item): Primeiramente, a função procura o nó com a string nome igual à string nome armazenada em item. Em seguida ela avalia se o nó é um nó folha ou se possui filhos. Se o nó é folha ele pode ser deletado sem problemas. Caso o nó possua somente o filho da direita o nó é desencadeado e esse filho assume seu lugar, o mesmo acontece caso o nó possua somente o filho da esquerda. No Caso do nó possuir os dois filhos, precisamos procurar um nó que possa substituí-lo na árvore sem violar as restrições da árvore no programa implementado o nó é substituído pelo seu antecessor, a função Antecessor é chamada passando como parâmetros o nó e o filho da esquerda desse nó.

-void Antecessor(TipoNo *q, TipoNo* &r): Essa função procura o filho mais a direita do filho a esquerda do nó que queremos remover. A função é chamada recursivamente passando o nó que queremos remover e o filho da direita até que encontremos um nó folha. Após encontrar o antecessor, o nó que queremos retirar recebe a string nome do antecessor, chama-se a função limpa para liberar o espaço da lista encadeada armazenada no nó que queremos remover e depois o ponteiro para lista encadeada armazenado no nó que queremos remover passa a apontar para a lista encadeada do nó antecessor.

As classes TipoCelula e TipoNo possuem como método apenas construtores que iniciam seus atributos. O contrutor de TipoNo aloca espaço para uma FilaEncadeada. A Classe TipoItem possui um construtor e uma função Imprime que basicamente imprime as strings dado e nome.

2.4 Programa Principal

Em primeiro lugar, o programa principal cria variáveis auxiliares que serão utilizadas para ler e armazenar os dados na árvore. Além disso cria também a ArvoreBinaria Arvore que será preenchida com os dados do documento de texto. Após isso, o programa abre o arquivo de texto, que foi passado como primeiro argumento na hora da execução, para leitura e pega a primeira linha, que representa o número de linhas que são inserções a serem feitas na árvore, e armazena em uma variável inteira.

No primeiro loop, que faz a inserção dos dados na árvore, cada linha é do arquivo é dividida em duas strings, uma contendo o nome e outra contendo o dado binário, que são armazenadas em um TipoItem chamado Consciencia. Em seguida, utiliza-se a função Procura, passando Consciencia como parâmetro, para saber se existe algum nó na árvore com o mesmo nome daquele armazenado em Consciencia e armazenando o ponteiro para TipoNo retornado em uma variável chamada encontrado. Caso não exista, a função Insere é chamada para colocar

esse Item na árvore. Caso exista, a função Enfileira da fila do nó encontrado é chamada para encadear a string dado de Consciencia. Após o final do loop, a Função InOrdem é chamada para imprimir a árvore.

O segundo loop lê as linhas de comando, que possuem apenas nomes, e armazena a cada iteração o nome lido em na string nome de Consciencia. Em sequência, utiliza-se a função Procura para encontrar o nó que contém o nome que foi passado como parâmetro, então imprime-se o Nome um espaço e soma dos binários na lista encadeada daquele nó, que é retornado pela função SomaBinarios da fila. Posteriormente, utiliza a função Remove para remover o nó com o nome que foi passado como parâmetro. Por fim, depois da finalização do loop, chama a função InOrdem para imprimir a árvore depois das remoções.

2.5 Detalhes Técnicos

O programa foi desenvolvido na linguagem C++, compilado pelo compilador G++ da GNU Compiler Collection. O sistema operacional utilizado foi o Ubuntu na versão 20.04. Em um computador com 16 Gigabytes de memória RAM e processador Intel Core i9.

3. Análise de complexidade

3.1 Tempo

As funções Imprime, tanto de FilaEncadeada quanto de ArvoreBinaria, possuem complexidade de tempo O(n), pois precisam percorrer todos os n elementos, tanto da árvore quanto da fila.

Análise de complexidade das funções mais importantes de FilaEncadeada:

-void Enfileira(string dado): Essa função, devido à presença do ponteiro tail que torna dispensável percorrer toda a fila, possui complexidade de tempo O(1).

-void Limpa(): Por ser necessário percorrer um por um todos os elementos da fila para liberar o espaço ocupado por eles, essa função tem complexidade de tempo O(n).

-int SomaBinarios(): Essa função possui dois loops aninhados. O loop externo é responsável por percorrer cada uma das células da lista, e o loop interno percorre cada um dos caracteres armazenados na string dado da célula. Considerando que toda strings dado das células da lista possuam k elementos e que a lista possua n células. Essa função possui complexidade O(n*k).

Análise de complexidade das funções mais importantes de ArvoreBinaria:

-void InsereRecursivo(TipoNo* &p, TipoItem item): Como nesse trabalho a árvore binária implementada não possui métodos para manter ela balanceada, a função de complexidade de inserção nessa árvore no pior caso é O(n). Que é o caso em que todos os nós só possuem filhos para a direita ou para a esquerda, formando basicamente uma lista. Nesse caso o algoritmo de inserção tem que percorrer toda a árvore. No melhor caso a árvore está balanceada e a complexidade dessa função é O(log n).

-TipoNo *ProcuraRecursivo(TipoNo* &p,TipoItem item): A complexidade da função de procura também depende de como a árvore foi construída. O pior caso acontece quando o elemento procurado não está na árvore e ela é degenerada, assemelha-se a uma lista. Nesse caso a complexidade dessa função é O(n). No caso de uma árvore balanceada a complexidade dessa função é O(log n).

-void Antecessor(TipoNo *q, TipoNo* &r): Essa função procura o antecessor do nó passado como parâmetro. No pior caso, o nó esquerdo passado e todos os seus filhos só possuem filhos da direita. Após achar o antecessor essa função também troca de posição com o outro nó passado, essa troca utiliza a função Limpa de lista encadeada, que tem complexidade igual a O(n) como explicado acima, nesse caso tempos O(n) + O(n) = O(n) que é a complexidade da função.

-void RemoveRecursivo(TipoNo* &no, TipoItem item): Essa função procura o nó que deve ser removido e se necessário chama a função Antecessor. De qualquer forma, a complexidade de tempo, caso fosse possível que tanto a remove recursivo, quanto a Antecessor possuirem complexidade O(n), é O(n) + O(n) = O(n).

Farei a análise da complexidade do programa principal levando em consideração apenas as complexidades no pior caso. No programa principal, o primeiro loop utiliza a cada iteração uma vez a função ProcuraRecursivo e uma vez a função InsereRecursivo, através das funções Insere e Procura, considerando que o arquivo tenha k linhas a serem inseridas a complexidade desse loop é (O(n) + O(n))*k = O(n*k).

Já o segundo loop utiliza uma vez a função ProcuraRecursivo, uma vez a função SomaBinários e uma vez a função RemoveRecursivo a cada iteração. Considerando que existem j linhas de comando, ou seja, o loop será executado j vezes, temos no pior caso, (O(n) + O(n*k) + O(n))*j = O(n*k*j).

3.2 Espaço

A complexidade de espaço das funções de FilaEncadeada:

-void Enfileira(string dado): Realiza todas as operações utilizando estruturas auxiliares unitárias O(1). Então sua complexidade de espaço é O(1).

-void Limpa(): Realiza todas as operações utilizando estruturas auxiliares unitárias O(1). Mesmo com o processo sendo executado n vezes, sendo n o tamanho da lista, a complexidade de espaço desse método é O(1).

-int SomaBinarios(): Realiza todas as operações utilizando estruturas auxiliares unitárias O(1). Então sua complexidade de espaço é O(1).

A complexidade de espaço das funções de ArvoreBinaria, como todas foram feitas utilizando métodos recursivos, requerem mais espaço na memória para as chamadas recursivas, e esse espaço extra é proporcional a altura da árvore. Nesse caso a complexidade de espaço das funções de ArvoreBinaria é O(n).

4. Conclusão

No trabalho foi implementado vários métodos recursivos, bem como a implementação de duas estruturas de dados distintas em conjunto, fila encadeada e árvore binária. Essa implementação foi feita para armazenar os dados que foram passados como parâmetro em uma árvore binária em que cada nó possui uma lista encadeada de dados binários. O trabalho foi desafiador no sentido de conseguir juntar essas duas estruturas de dados e tanto alocar quanto desalocar o espaço reservado para elas durante a execução do programa.

Em resumo, foi colocado em prática algumas das estruturas de dados aprendidas em sala de aula, bem como a prática do pensamento computacional necessário para alterar essas estruturas de modo a atender às exigências do trabalho.

5. Bibliografia

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

6. Instruções para compilação e execução

Para compilar o programa é necessário acessar, através do terminal, o diretório, com as pastas (bin, include, obj, src) que foram enviadas e executar o comando make. Para rodar o programa, basta executar através do terminal o programa run.out passando como primeiro argumento o nome do arquivo de texto com os dados.