UFMG

Estruturas de Dados – Trabalho Prático 3

Recebimento e Download das Consciências

Nome: Arthur Pontes Nader

Matrícula: 2019022294

Universidade Federal de Minas Gerais

Belo Horizonte - MG - Brasil

arthurpn@ufmg.br

1. Introdução

Como última tarefa para o Sistema de Realocação Interplanetária (Trabalhos Práticos 1 e 2), foi solicitada a implementação de um mecanismo eficiente para armazenamento e busca de consciências a serem realocadas. Essa atualização permitirá que a transmissão e a consulta de dados ocorra em menor tempo de execução.

Saber lidar com arvores binárias de pesquisa é uma atividade essencial para programadores que necessitam de realizar buscas sobre um grande volume de dados. Essa estruturação de dados é feita de tal forma que, dado um nó qualquer da árvore, todo elemento menor que ele estará localizado a sua esquerda e todo elemento que for maior que ele se encontrará a sua direita.

Dessa maneira, esse trabalho teve como objetivo realizar a implementação de uma árvore binária de tal modo que organizasse melhor os dados e agilizasse a realocação das consciências.

A seção 2 desse relatório detalha a estruturação do programa, enquanto a seção 3 apresenta a complexidade de cada função criada. A seção 4 apresenta uma breve conclusão do trabalho feito. Já a seção 5 mostra as referências bibliográficas utilizadas. Ao final, há um apêndice explicando como se deve compilar e executar o programa.

2. Descrição da implementação

O programa foi desenvolvido utilizando-se três classes: Dado, Consciencia e Arvore. A seguir, há uma breve descrição dos métodos implementados para cada uma dessas classes, informações sobre a “main” e sobre a maneira como o código lida com entradas inválidas.

- **Classe Dado:**

A classe Dado possui como atributo uma string que armazenará um dado e um ponteiro para outro objeto do tipo Dado. Assim, é possível construir uma lista encadeada.

Dado(std::string dadoPassado): inicializa o atributo dado com o valor passado como parâmetro e aponta o ponteiro para “nullptr”.

- **Classe Consciencia:**

A classe Consciencia tem como principal papel o armazenamento dos dados e a instanciação das consciências a serem realocadas. Cada uma dessas consciências possuirão um nome e uma lista encadeada contendo seus dados.

Consciencia(std::string nomePassado, std::string dadoPassado): construtor que cria uma consciência que tem como nome e dados binários os parâmetros passados na chamada do método.

void exibirConsciencia(): imprimi na saída padrão o nome da consciência.

int calcularSomaDados(): converte os dados binários da lista encadeada para o valor decimal, realiza o somatório deles e retorna o valor resultante.

void exibirConscienciaMaisDados(): imprimi na saída padrão o nome da consciência e o valor retornado pelo método calcularSomaDados.

void adicionarDado(std::string dadoNovo): cria um novo objeto do tipo Dado e o adiciona ao final da lista encadeada.

- **Classe Arvore:**

A classe Arvore tem como principal função a estruturação da árvore binária. Ela é criada de tal forma que todo elemento à esquerda de uma consciência seja menor do que ela e todo à direita seja maior. Caso a

consciência a ser removida tenha subárvores tanto à esquerda como à direita, utiliza-se a substituição pelo antecessor para manter as propriedades da árvore. Essa substituição faz com que o maior elemento que seja menor que a consciência a ser removida assuma o seu lugar.

Arvore(): o construtor padrão de objetos da classe Arvore apenas inicializa a raiz apontando-a para “nullptr”.

void adicionarConsciencia(Consciencia*& local, std::string nomeConsciencia, std::string dadoConsciencia): adiciona a nova Consciencia em sua devida posição na árvore. Se ela já estiver presente, adiciona-se o dado passado como parâmetro ao final de sua lista encadeada**.**

void inserirConsciencia(std::string nomeConsciencia, std::string dadoConsciencia): chama o método adicionarConsciencia passando a raiz como parâmetro.

void removerConsciencia(Consciencia*& local, std::string nomeConsciencia): recebe como parâmetro o nome da Consciência a ser removida e, realizando a busca pela árvore, remove-a quando ela for encontrada.

void remove(std::string nomeConsciencia): chama o método removerConsciencia passando a raiz como parâmetro.

void substituirPeloAntecessor(Consciencia* c1, Consciencia*& c2): se a consciência a ser removida tiver outras consciências tanto na esquerda como na direita, ela será substituída pela maior consciência antecessora a ela.

void caminhamentoCentral(Consciencia & local, Consciencia ultima): realiza a impressão do nome das consciências presentes na árvore em ordem.

void realizarCaminhamento(): chama o método caminhamentoCentral passando a raiz como parâmetro.

- **Main:**

A abertura do arquivo a ser lido ocorrerá na “main” do programa. A instanciação das consciências ocorre por meio de um laço “for” que lê e formata as linhas do arquivo até que se atinja a quantidade informada. Essa formatação é responsável por gerar os parâmetros de instanciação de um objeto da classe Consciencia que será inserido na árvore. Em seguida, realiza-se a impressão dos elementos da árvore em ordem. Após isso, as remoções de consciências são realizadas e imprime-se

mais uma vez os elementos em ordem, mostrando assim a configuração final da árvore. Por fim, a arvore é desalocada, fecha-se o arquivo e o programa é encerrado.

- **Entradas imprevistas:**

Entradas de dados da consciência que não sigam o padrão estabelecido e tentativas de remoção de consciências que não estejam presentes na árvore poderão ocasionar erros de execução.

3. Complexidade de espaço e de tempo

A variável “n” representará o número de consciências instanciadas a partir da leitura do arquivo e a variável “m” será o número de dados presentes em uma lista encadeada de uma consciência. Essas variáveis facilitarão a realização da análise assintótica dos métodos criados.

- **Métodos da Classe Dado:**

Dado(std::string dadoPassado):

Complexidade de tempo: construtor apenas cria um objeto do tipo Dado com o parâmetro passado, portanto, O(1).

Complexidade de espaço: criar um objeto dessa classe tem custo de espaço constante, ou seja, O(1).

- **Métodos da Classe Consciencia:**

Consciencia(std::string nomePassado, **std::string dadoPassado):**

Complexidade de tempo: construtor apenas cria um objeto do tipo Consciencia com os parâmetros passados e atualiza os ponteiros, portanto, O(1).

Complexidade de espaço: nenhuma memória extra é necessária, portanto, O(1).

void exibirConsciencia():

Complexidade de tempo: imprimir uma string tem custo O(1).

Complexidade de espaço: O(1), pois não é necessário memória adicional para execução.

int calcularSomaDados():

Complexidade de tempo: o método deverá realizar a conversão e a soma de todas as strings da lista encadeada, portanto, seu custo é O(m).

Complexidade de espaço: o método não necessita de memória adicional, assim, ele é O(1).

void exibirConscienciaMaisDados():

Complexidade de tempo: como o método chama a função calcularSomaDados, ele também é O(m).

Complexidade de espaço: O(1), pois não é necessário memória extra para execução.

void adicionarDado(std::string dadoNovo):

Complexidade de tempo: apenas insere um dado ao final da lista encadeada. Dessa forma, o método é O(1).

Complexidade de espaço: apenas cria um objeto novo da classe Dado, desse modo, o método é O(1).

- **Métodos da Classe Arvore:**

Arvore():

Complexidade de tempo: construtor apenas aponta a raiz para “nullptr”, portanto, O(1).

Complexidade de espaço: criar um objeto dessa classe tem custo de espaço constante, ou seja, O(1).

void adicionarConsciencia(Consciencia*& local, std::string nomeConsciencia, std::string dadoConsciencia):

Complexidade de tempo: O(log n), pois essa é a complexidade do caso médio para se achar uma consciência ou atingir um nó folha da árvore.

Complexidade de espaço: O(log n), já que essa é a complexidade de espaço do caso médio devido às chamadas do método adicionadas à pilha de execução.

void inserirConsciencia(std::string nomeConsciencia, std::string dadoConsciencia):

Complexidade de tempo: apenas chama o método adicionarConsciencia, portanto, O(log n).

Complexidade de espaço: o método é O(log n) para a complexidade de espaço, pois apenas chama adicionarConsciencia.

void removerConsciencia(Consciencia*& local, std::string nomeConsciencia):

Complexidade de tempo: uma vez que se deve achar a consciência e realizar a soma de sua lista encadeada, o método será O(log n + m) no caso médio.

Complexidade de espaço: devido à pilha de execução, no caso médio, o método é O(log n) para a complexidade de espaço.

void remove(std::string nomeConsciencia):

Complexidade de tempo: apenas chama removerConsciencia, assim, O(log n + m).

Complexidade de espaço: somente chama removerConsciencia, dessa maneira, o método é O(log n).

void substituirPeloAntecessor(Consciencia* c1, Consciencia*& c2):

Complexidade de tempo: procurar um elemento em uma subárvore tem custo de tempo no pior caso O(log n).

Complexidade de espaço: O(1), pois nenhuma memória extra é necessária para execução.

void caminhamentoCentral(Consciencia & local, Consciencia ultima):

Complexidade de tempo: percorrer a árvore imprimindo o nome das consciências tem custo O(log n).

Complexidade de espaço: as chamadas ao método são desempilhadas durante a execução, por isso, no caso médio, a complexidade de espaço é O(log n).

void realizarCaminhamento():

Complexidade de tempo: esse método somente chama a função caminhamentoCentral, logo, ele é O(log n).

Complexidade de espaço: como o método apenas chama caminhamentoCentral, ele é O(log n).

- **Main:**

Complexidade de tempo: a complexidade de tempo do programa principal será limitada pela função de maior custo computacional. Assim, o programa principal é O(log n).

Complexidade de espaço: o programa principal pode chamar o método que possui a maior complexidade de espaço. Assim, ele será O(log n).

4. Conclusão

O trabalho realizado consistiu na implementação de uma árvore binária para organização das consciências com intuito de aumentar a eficiência do Sistema de Realocação Interplanetária.

A principal dificuldade apresentada durante o desenvolvimento foi na remoção de uma consciência que possuía uma subárvore à direita e outra à esquerda. Entretanto, isso foi solucionado rapidamente observando-se a evolução dos valores das variáveis durante a execução.

Desse modo, a realização do trabalho prático proporcionou uma boa aplicação dos conceitos sobre árvores binárias vistos em aula. Um bom entendimento das funções recursivas foi essencial durante o processo de elaboração dos métodos de inserção e remoção de consciências.

5. Bibliografia

- PRATES, R., CHAIMOWICZ, L. **Árvore Binária de Pesquisa.** Departamento de Ciência da Computação, UFMG, 2020. Apresentação em slide. 52 slides.
- CORMEN, T., LEISERSON, C., RIVEST, R., STEIN, C**. Introduction to Algorithms**. Third Edition, MIT Press, 2009.
- DROZDEK, A. **Data Structures and Algorithms in C++.** Fourth Edition, Cengage Learning, 2013.
- ZIVIANI, N**. Projeto de Algoritmos com Implementações em Pascal e C**. 3ª edição, Cengage Learning, 2011.

Apêndice - Compilação e execução

Para compilação e execução do programa desenvolvido, utilizou-se um computador com 4 GB de memória RAM e que possui como processador um Intel i3 de terceira geração. O sistema operacional usado durante o processo foi o Linux Mint.

Inicialmente, deve-se adicionar os arquivos “.txt” a serem lidos pelo programa na raiz do projeto. Para compilar o programa desenvolvido, é preciso acessar o diretório do programa e digitar “make” no terminal de comando. Aparecerá no terminal que os arquivos do projeto foram compilados com sucesso.

Para execução do código, deve-se digitar no terminal “./bin/run.out ” seguido do nome do arquivo contendo os dados necessários para execução. Ao final do processo, será mostrado no terminal a configuração inicial da árvore binária, as consciências removidas e a soma de seus dados em valor decimal e, por fim, a configuração da árvore binária após as remoções.

Para finalizar o processo de modo correto, digite “make clean” no terminal para limpar os arquivos executáveis gerados.