Trabalho Prático 3 - Servidor de Emails
Universidade Federal de Minas Gerais

Igor Rahzel Colares Galdino
Matricula 2020096255

Julho 2022

1 Introdução

O Trabalho Prático consiste na implementação de um servidor de emails por meio da combinação de duas estruturas de pesquisa, sendo elas uma tabela Hash e uma árvore binária de pesquisa. Essa implementação deve ser feita utilizando-se pelo menos os seguintes Tipos Abstratos de Dados(TADs) **email**, **árvore binária** e **tabela Hash**.

A seção 2 desta documentação trata sobre os detalhes da implementação do projeto, mostrando como os arquivos do projeto estão divididos, bem como o papel e a funcionalidade de cada um deles para a solução do problema. Já na seção 3 é feita a análise de complexidade das funções. A conclusão está localizada na seção 4. Na seção 5 encontram-se as referências bibliográficas utilizadas, por fim na seção 6 estão as instruções de compilação e execução.

2 Implementação

2.1 Organização do Código

As pastas do projeto estão organizados da seguinte forma:

TP:
- -src:
  - -Email.cpp
  - -Node.cpp
  - -BST.cpp
  - -Hash.cpp
  - -main.cpp
- -include:
  - -Email.h
  - -Node.h
  - -Hash.h
  - -main.h
- -bin:
- -obj:
- -Makefile

2.1.1 Arquivos .cpp

2.1.2 Email.cpp

O arquivo *Email.cpp* implementa apenas getters e setters da classe *Email.cpp* e o método *isEmpty* que verifica se o email está vazio checando se seu id é igual a -1, pois está foi a flag escohida para representar tal caso.

2.1.3 BST.cpp e Node.cpp

No arquivo *BST.cpp* é implementada uma árvore binária de pesquisa com auxílio da classe *Node*, desenvolvida no arquivo *Node.cpp*, a qual possui somente um construtor que inicializa os apontadores como NULL. Os métodos da classe *BST* são: *Insert*, esse método apenas realiza uma chamada para o método *_Insert*, que percorre a árvore binária respeitando a seguinte propriedade, caso o id do email a ser inserido seja maior que o id de um nó deve-se caminhar para direita, caso contrário para esquerda, esse processo é realizado até que se alcance um nó folha da árvore no qual o email é inserido.

O método *Search* apenas chama o método *_Search*, que assim como o método *_Insert* percorre a árvore binária, porém nesse caso isso é feito até se encontrar o email com o id e usuário especificados.

Já o método *Remove*, assim como os demais faz uma chamada para outro método, *_Remove*, que caminha pela árvore binária até encontrar o email desejado e remove-lo. Essa função leva em consideração 3 casos, o primeiro deles é quando se exclui um nó que possui apenas um filho, quando isso acontece coloca-se seu filho em seu lugar, o caso de um nó que não possui filhos está englobado no caso acima, pois basta imaginar que seu filho é um nó NULL, por fim quando o nó a ser removido possui dois filhos chama-se a função *Antecessor* que percorre a subárvore esquerda daquele nó de modo a encontrar o nó mais a direita, quando encontrado substitui-se ele no lugar do nó que desejava-se excluir.

2.1.4 Hash.cpp

Em *Hash.cpp* implementa-se os métodos da tabela Hash. O método *Build* é utilizado para alocar dinamicamente um vetor(tabela) do tamanho do valor passado como parâmetro, além de atribuir esse valor ao parâmetro *size* da classe *Hash*. Já o método *Hash_function* realiza a operação de módulo entre o id do email e o valor do parâmetro *size* e retorna esse valor.

Os métodos *Search, Insert* e *Remove* seguem a mesma lógica de implementação, ambos identificam a posição da tabela a ser analisada apartir do método *Hash_function* e então chama-se algum dos métodos implementados na classe *BST*, para se pesquisar,inserir ou remover um email na árvore binária de pesquisa referente a aquela posição. O arquivo main.cpp é discutido na seção 2.3

2.2 Arquivos de cabeçalho

Na pasta include é onde estão localizados os arquivos de cabeçalho com extensão .h, os quais contém as assinaturas dos métodos e atributos das classes a serem implementadas.

O arquivo *Email.h* define os atributos *int id, int user_id* e *string msg*, esses tem como função registrar o id do email, registrar o id do usuário e armazenar a mensagem enviada respectivamente. Já *Node.h* define o construtor de *Node* e seus atributos, sendo eles: *Email email* que armazena um objeto da classe *Email* e *Node* left* e *Node* right* que apontam para um nó a direita e a esquerda.

O cabeçalho *BST.h* tem como atributo o nó raiz da da árvore binária, o restante do arquivo são assinaturas dos métodos implementados

Hash tem como atributos *int size* utilizado para registrar o tamanho da tabela Hash, *BST* array* utilizado para fazer a alocação dinâmica da tabela Hash de acordo com o valor recebido como parâmetro pelo método *Build*.

2.3 Funcionamento do Programa Principal

O arquivo main.cpp é o arquivo principal do programa. A função *main* realiza a leitura dos argumentos passados pela linha de comando(arquivos de entrada e saída), então é realizada a leitura do arquivo de entrada e de acordo com os comandos fornecidos(*ENTREGA,CONSULTA e APAGA*), são realizadas suas respectivas funções na tabela Hash e árvore binária correspondente a cada posição da tabela. Após a realização de cada comando é escrita uma mensagem no arquivo de saída.

2.4 Configurações Utilizada

Abaixo são apresentadas as configurações utilizadas para as realizações dos testes do programa:

- Sistema Operacional do Computador: Linux Ubuntu 22.04 LTS
- Linguagem de Programação : C++
- Compilador Utilizado : g++
- Dados do Processador : Intel® Core™ i5-10310U CPU @ 1.70GHz × 8
- Memória RAM : 8 GB Soldado DDR4 2666MHz

2.5 Estratégias de Robustez

As estratégias de robustez adotadas para o programa são a impressão das mensagens: *"CONSULTA U E: MENSAGENS INEXISTENTE"*, *"ERRO: MENSAGEM INEXISTENTE*, onde U representa o id usuário e E representa o id do email, ambas as mensagens são escritas no arquivo de saída quando deseja-se consultar ou apagar uma mensagem inexistente respectivamente.

3 Análise de complexidade

3.1 classe Email

Todos os métodos da classe *Email* tem complexidade de tempo e espaço igual a $O(1)$, pois são apenas getters e setters, ou seja, retornam algum valor de seus atributos ou realizam atribuições.

3.2 classe Node

A classe *Node* possui apenas um construtor que inicializa seus ponteiros como NULL, dessa forma a complexidade de tempo e espaço desse método é $O(1)$.

3.3 classe BST

.

_Insert - complexidade de tempo: Essa função é $O(n)$ e ocorre quando a árvore binária está degenerada, ou seja, quando cada nó possui apenas um filho e eles são inseridos em ordem crescente ou decrescente. Caso a árvore binária esteja balanceada seu pior caso é $O(logn)$

_Insert - complexidade de espaço: Caso a árvore binária esteja degenerada sua complexidade de espaço é $O(n)$, pois fará n chamadas recursivas. Caso ela esteja balanceada sua complexidade será $O(logn)$, pois este será aproximadamente o número de chamadas recursivas realizadas.

Os métodos *_Search e _Remove*, possuem a mesma análise de complexidade de tempo e espaço do método _Insert, pois ambos devem caminhar pela árvore desejado.

3.4 Hash

Build - complexidade de tempo: A complexidade de tempo dessa função é ordem de $O(1)$, pois são realizadas um número fixo de atribuições.

Build - complexidade de espaço: A complexidade de espaço dessa função é $O(n)$, pois é alocado um vetor do tamanho do parâmetro passado para a função.

Hash_function - complexidade de tempo: Esse método é $O(1)$, pois apenas realiza uma operação.

Hash_function- complexidade de espaço: A ordem da complexidade de espaço é $O(1)$, pois apenas realiza um número constante de operações.

Search - complexidade de tempo: A complexidade assintótica desse método é o mesmo da função *Search* da classe *BST*, pois é feita uma série de operações de custo constante e então realiza-se uma chamada para este método.

Search - complexidade de espaço: Pelos motivos descritos na complexidade de tempo, a ordem de complexidade de espaço também será igual ao do método *Search* da classe *BST*.

A vantagem de se utilizar a tabela Hash nessa implementação é que ao invés de se utilizar uma única árvore binária para armazenar todos os emails de todos os usuários, temos uma árvore binária para cada entrada da tabela, ou seja, uma árvore binária que armazena apenas os email dos usuários cujo módulo do seu id pelo tamanho da tabela é igual. Isso faz com que a altura da árvore percorrida para armazenar o email seja menor, embora o número de nós criados(nós de todas as árvores da tabela) seja o mesmo.

Os métodos *Insert e Remove* são casos análogos ao método *Search*, isso ocorre, pois ambos realizam operações de custo constante e depois fazem uma chamada para um método da classe *BST* que irá caminhar em uma árvore binária.

3.5 programa principal

main - complexidade de tempo: A complexidade de tempo da função *main* é $O(logn)$, considerando-se que a árvore não vai estar degenerada, pois a partir dos comandos do arquivo serão feitas inserções,busca e remoções em uma árvore binária de pesquisa e tais operações tem complexidade $O(logn)$ caso a árvore esteja degenerada a complexidade será $O(n)$.

main - complexidade de espaço: A complexidade de espaço é $O(logn)$ caso a árvore não esteja degenerada,caso contrário a complexidade da função main é $O(n)$, a complexidade de espaço nos dois casos é dada pelo número de chamadas recursivas realizadas.

4 Conclusão

O trabalhou tratou do problema da implementação de um simulador de um servidor de emails. A abordagem para esse problema foi a criação da classe *Email*, a qual armazenava o id do usuário, o id da mensagem e o texto da mensagem, foi então implementada a classe *BST*, a qual foi utilizada para criar uma árvore binária de pesquisa. Por fim criou-se a classe *Hash* para realizar a implementação da tabela *Hash*, onde cada posição da tabela continha uma árvore binária de pesquisa, tal implementação permite criar várias árvores de alturas menores de modo a diminuir o número total de nós percorridos ao se realizar as operações de consulta, remoção ou inserção de um email.

O principal desafio do trabalho foi abstrair o que foi solicitado no seu enunciado para a elaboração do programa, outro desafio foi fazer como que os dois algoritmos de pesquisa de forma conjunta.

Através desse trabalho foi possível compreender melhor sobre o funcionamento da tabela Hash e da árvore binária de pesquisa, e principalmente a vantagem da combinação desses dois métodos.

5 Referências

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

https://www.cplusplus.com/reference/fstream/fstream/

6 Instruções de Compilação e Execução

- Pelo terminal acesse o diretório da pasta TP
- Execute o arquivo *Makefile* utilizando o comando *make all*
- Esse comando irá gerar o arquivos objetos .o dos arquivos .cpp
- Proceda para digitar o comando *.\bin\tp3.out*