**Trabalho Prático 3 – Servidor de e-mails otimizado**

**Carlos Henrique Brito Malta Leão**

**Matrícula: 2021039794**

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte – MG – Brasil

chbmleao@ufmg.br

1. **Introdução**

O objetivo deste trabalho é a implementação do simulador de um servidor de e-mails, em que, além do gerenciamento adequado da memória do sistema, há foco na otimização da pesquisa por usuários e mensagens. Dessa forma, o sistema implementado apresenta suporte à entrega, consulta e remoção de mensagens para usuários, em que o sistema funciona corretamente ao executar as diversas operações do servidor em diferentes situações.

Esta documentação tem como objetivo explicitar como foi realizada a implementação desse sistema, além de realizar uma análise explicativa sobre como o simulador do servidor de e-mails funciona. Ademais, também serão explicitados a robustez, abstração e desempenho do programa, possibilitando um melhor entendimento de como o sistema funciona na prática.

Para a resolução do problema supracitado, foi criado um programa na linguagem C++, que utiliza quatro principais classes: *Email, Node, BinaryTree* e *Hash\_BT*. Essas quatro classes representam em conjunto uma tabela hash de árvores binárias. Dessa forma, a classe *Email* representa um e-mail, que são armazenados em nós, representados pela classe *Node*. Esses nós, fazem parte de uma árvore binária, *BinaryTree*, que são armazenadas em um arranjo na classe *Hash\_BT,* que é utilizada para a realização de pesquisas de e-mails. Por fim, temos a estrutura *memlog,* que auxilia no processo de análise de desempenho do programa e as funções *msgassert,* que são utilizadas de forma a aumentar a robustez do programa.

Por fim, ao decorrer dessa documentação, alguns aspectos sobre o trabalho serão melhor explicados, como a descrição da implementação na seção 2 e a análise de complexidade na seção 3. Em seguida, nas seções 4 e 5 serão explicitadas as estratégias de robustez e a análise experimental. Por fim, teremos a conclusão do trabalho, sumarizando o que foi aprendido durante seu desenvolvimento.

1. **Método**

Durante essa seção, será realizada uma descrição da implementação do programa estruturado na linguagem C++. Dessa forma, serão detalhadas as estruturas de dados, classes e funções implementadas.

**2.1 Email**

Primeiramente, a classe mais simples do programa é a classe *Email*, que simula um objeto de e-mail, armazenando três atributos, sendo a mensagem, o identificador do usuário destinatário e o identificador do e-mail em si. A mensagem é armazenada em um atributo de tipo *string* e os identificadores são armazenadas em inteiros.

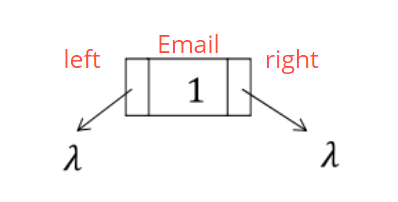
Nesse sentido, essa classe é muito simples e apresenta apenas algumas funções:

* *Email():* Inicializa um e-mail com mensagem vazia e identificadores com -1.
* *Email(emailKey, userKey, message):* Inicializa um e-mail com os atributos recebidos como parâmetro.
* *getEmailKey():* Retorna o identificador de e-mail.
* *getUserKey():* Retorna o identificador do destinatário.
* *getMessage():* Retorna a mensagem do e-mail.
* *print():* Função para depuração que imprime os atributos do e-mail.

**2.2 Node**

A classe *Node* também é muito simples, ela representa uma célula, nó, de uma árvore binária, que será melhor explicada em seguida. Dessa forma, essa classe apresenta apenas três atributos, o e-mail que será armazenado na célula e os endereços das células localizadas na esquerda e na direita, no formato de um nó de uma árvore binária.

Um diagrama esquemático da classe nó pode ser visto na figura a seguir. Nesse sentido, o número 1 representa o identificador de e-mail armazenado e ‘left’ e ‘right’ representam os endereços das células da esquerda e direita respectivamente.



Por fim, a função apresenta apenas duas funções, que são seus construtores:

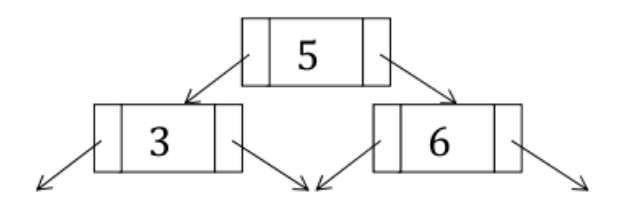
* *Node():* Inicializa um nó com um e-mail vazio e endereços da esquerda e direita nulos.
* *Node(email):* Inicializa um nó com um e-mail recebido como parâmetro e endereços da esquerda e direita nulos.

**2.3 BinaryTree**

A classe *BinaryTree* é mais complexa, ela representa uma árvore binária formada por nós que armazenam e-mails. Nessa árvore binária, cada nó pode ter no máximo dois filhos, que são subárvores da esquerda e da direita. Nesse sentido, considerando um nó em específico, todos os nós que forem armazenados na sua subárvore da esquerda, devem possuir um identificador de e-mail menor que o que ele possui, enquanto a subárvore da direita deve apresentar apenas identificadores de e-mail maiores. Essa lógica serve para todos os nós da árvore, o que garante melhor eficiência para métodos de inserção, remoção e pesquisa de e-mails.

Ademais, a classe apresenta um atributo que armazena a raiz da árvore, a partir desse nó, é possível chegar em qualquer outro nó da árvore binária. Além disso, a árvore não apresenta elementos repetidos, já que cada e-mail enviada e recebido apresenta identificadores distintos.

Um diagrama esquemático da árvore binária implementada pode ser visto na figura a seguir. Nesse sentido, os números representam o identificador de e-mail armazenada em cada célula, enquanto as setas representam os endereços da esquerda e direita armazenados.



Por fim, essa classe apresenta algumas funções específicas, que abstraem e aumentam a robustez do código. Nesse sentido, percebemos que as funções de inserção, remoção e pesquisa são muito semelhantes, devido ao funcionamento da árvore. Nesse sentido, essa funções são recursivas, em que o caso base ocorre quando a função chega em um nó com endereço nulo, isso significa que a árvore chegou em sua extremidade, uma folha, e que não há mais nós para percorrer.

* *BinaryTree():* Construtor da classe que atribui endereço nulo para a raiz.
* *~BinaryTree():* Destrutor da classe que chama a função *clean*.
* *insert(email):* Insere um e-mail na árvore binária chamando a função *insertRecursive* e passando o e-mail e a raiz da árvore como parâmetros.
* *insertRecursive(currentNode, email):* Essa é uma função recursiva para a inserção de e-mails na árvore. O caso base da função ocorre quando o nó atual apresenta endereço nulo, dessa forma é possível inserir um novo nó na árvore. Caso contrário, se a chave do e-mail a ser inserido é menor que a chave do nó atual, a função começa a tratar o nó da esquerda, caso contrário a função chama o nó da direita. Dessa forma, a função percorre os elementos da árvore até chegar em um nó com endereço nulo.
* *printInOrder():* Imprime os e-mails da árvore binária em ordem crescente dos identificadores de e-mail chamando a função *printInOrderRecursive*.
* *printInOrderRecursive(currentNode):* Função recursiva para a impressão dos e-mails da árvore em ordem. O caso base da função ocorre se o nó atual tiver endereço nulo. Caso contrário, a função chama o nó a esquerda, imprime o e-mail, e em seguida chama o nó a direita. Dessa forma, os nós da árvore binária são impressos em ordem.
* *clean():* Libera a memória alocada para os nós da árvore chamando a função *cleanRecursive* e atribuindo o endereço nulo para a raiz.
* *cleanRecursive(currentNode):* Função recursiva que libera a memória alocada para os nós da árvore. O caso base da função ocorre se o nó atual tiver o endereço nulo. Caso contrário, a função percorre a árvore até os nós da extremidade, os excluindo primeiro até chegar a raiz.
* *search(emailKey, userKey):* Pesquisa na árvore um e-mail com os identificadores de e-mail e usuário recebidos como parâmetros chamando a função *searchRecursive*.
* *searchRecursive(currentNode, emailKey, userKey):* Função recursiva que pesquisa um e-mail específico na árvore. O caso base da função ocorre se o nó atual tiver o endereço nulo. Caso contrário, se o identificador do e-mail for menor que o e-mail atual, o nó da esquerda é chamado, caso contrário, se o identificador for maior que o atual, o nó da direita é chamado, caso contrário, se o identificador do usuário for igual ao do e-mail atual, a função retorna o e-mail atual, caso contrário, a função retorna um e-mail vazio.
* *remove(emailKey, userKey):* Remove um nó da árvore com os identificadores de e-mail e usuário recebidos como parâmetros, chamando a função *removeRecursive*.
* *removeRecursive(currentNode, emailKey, userKey):* Função recursiva que remove um nó específico da árvore. O caso base da função ocorre se o nó atual tiver o endereço nulo. Caso contrário, o processo é muito semelhante ao procedimento da função de pesquisa, mas ao invés de retornar o e-mail procurado, o nó procurado tem sua memória liberada. Ademais, ao liberar a memória, se o nó da direita for nulo, o nó atual recebe os valores do nó da direita, caso contrário, se o nó da esquerda for nulo, o nó atual recebe os valores do nó da esquerda. Caso contrário, se os nós da esquerda e direita não sejam nulos, é preciso chamar a função auxiliar *previous*, passando o nó atual e o nó da esquerda. Por fim, caso o nó procurado exista e seja deletado, a função retorna o booleano verdadeiro, caso contrário retorna falso.
* *previous(Node q, Node r):* Função recursiva que, caso necessário, reestrutura a árvore após uma remoção. O caso base da função ocorre quando o nó atual tiver o endereço nulo. Caso contrário, a função é chamada novamente com o endereço do nó a ser retirado e o nó a direita do antecessor. Além disso, a função atribui ao antecessor os valores do nó a esquerda do antecessor, e libera a memória desse antecessor.

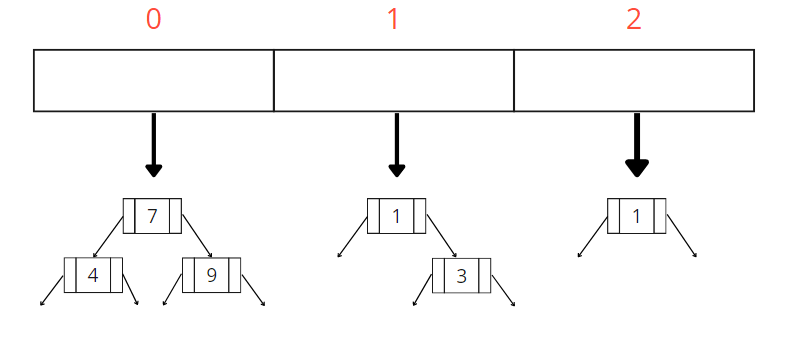
**2.4 Hash\_BT**

A classe *Hash\_BT* implementa um algoritmo de pesquisa do tipo Hashing, que são algoritmos que efetuam uma transformação aritmética sobre a chave de pesquisa para encontrá-la. Nesse sentido, os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa. Além disso, a busca é feita por meio de operações aritméticas que transformam a chave em endereços em uma tabela. A utilização de árvores binárias deve-se ao tratamento de colisões.

Dessa forma, na implementação utilizada, é criada uma tabela de árvores binárias, em que o valor de é especificado no início do arquivo de entrada. Nesse sentido, a operação aritmética utilizada para encontrar o endereço da tabela é o resto da divisão do identificado do usuário pelo tamanho da tabela de árvores, esse resto é a posição da tabela de árvores binárias que o e-mail deve ser inserido.

Porém, considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço da tabela, ou seja, o e-mail de mais de um usuário pode ser armazenado em uma mesma árvore binária. Dessa forma, ao realizar buscas nas árvores binárias, é preciso considerar tanto o identificador do e-mail quanto o do usuário, para não realizar operações equivocadas.

Um diagrama esquemático do Hash de árvores binárias implementado pode ser visto na figura a seguir.



Por fim, essa classe apresenta algumas funções específicas, que abstraem e aumentam a robustez do código. Nesse sentido, a maioria delas segue o mesmo procedimento, primeiro se utiliza a operação aritmética supracitada para encontrar a posição da tabela de árvores binárias a ser utilizada, e em seguida utiliza uma função dessa árvore.

* *Hash\_BT(tableSize):* Construtor da classe que atribui o valor do tamanho da tabela e aloca o espaço na memória para as árvores binárias.
* *hash(key):* Função aritmética do Hashing, retorna o resto da divisão do identificador do usuário pelo tamanho da tabela, ou seja, retorna a posição que deve ser inserida na tabela.
* *search(userKey, emailKey):* Pesquisa um e-mail com os identificadores recebidos como parâmetro. Primeiramente encontra a posição chamando a função *hash*, e, em seguida, chama a função *search* da árvore binária na posição encontrada.
* *insert(userKey, email):* Insere um e-mail na tabela hash. Primeiramente encontra a posição que deve ser inserido o e-mail na tabela chamando a função hash, e, em seguida, chama a função *insert* da árvore binária na posição encontrada.
* *remove(userKey, emailKey):* Remove um e-mail da tabela hash. Primeiramente encontra a posição que deve ser removido o e-mail na tabela chamando a função hash, e, em seguida, chama a função *remove* da árvore binária na posição encontrada.
* *print():* Função utilizada para depuração de código, que imprime todos os e-mails armazenados na tabela hash.

1. **Análise de complexidade**

Esta seção apresenta a análise de complexidade de tempo e espaço para a execução completa do programa, desde a leitura do arquivo até a impressão de todos os comandos. Para um melhor entendimento, realizaremos a análise de complexidade de tempo primeiro e, em seguida, a análise de espaço.

**3.1 Análise de complexidade de tempo**

Para entender melhor a complexidade do programa, primeiramente realizaremos a análise de complexidade de cada função de forma individual, para, em seguida, analisar a complexidade do programa como um todo. Além disso, para uma melhor organização, analisaremos uma classe implementada de cada vez.

**3.1.1 Email**

  A classe *Email*, como já foi supracitado, é uma classe muito simples, que apresenta apenas alguns atributos e funções. Nesse sentido, essas funções lidam apenas com operações constantes, como os construtores que atribuem valores, ou as funções do tipo *get* que retornam atributos da classe. Além disso também existe a função *print*, que foi utilizada exclusivamente para a depuração de código, mas também só realiza operações constantes. Por fim, todas as funções da classe *Email*, apresentam complexidade assintótica .

**3.1.2 Node**

A classe *Node* representa uma célula da árvore binária implementada. Dessa forma, esta função também apresenta um comportamento muito simples, assim como suas funções. Nesse sentido, a classe apresenta apenas duas funções, sendo elas o construtor padrão e o construtor que recebe um e-mail como parâmetro, ambas realizam apenas operações de custo constante, apresentando complexidade assintótica .

**3.1.3 BinaryTree**

A classe BinaryTree representa uma árvore binária de elementos, que no caso são os e-mails enviados e altura . Dessa forma, utilizaremos a quantidade de e-mails para realizar a análise de complexidade, juntamente com a altura da árvore. A seguir, analisaremos a complexidade de cada função da classe de forma separada.

* *BinaryTree:* Construtor da classe, inicializa o atributo com operações constantes com tempo
* *~BinaryTree:* Chama a função clean da classe com tempo .
* *clean:* Chama a função *cleanRecursive*, com tempo *.* Além disso, realiza operações constantes com tempo . Dessa forma, a complexidade assintótica da função será *.*
* *cleanRecursive:* Realiza operações constantes com tempo . Além disso, acessa todos os nós da árvore de forma recursiva, com tempo . Dessa forma, a complexidade assintótica da função será *.*
* *search:* Chama a função *searchRecursive*, com tempo médio *.*
* *searchRecursive:* Realiza operações constantes com tempo *.* Além disso, o tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato das árvores. Dessa forma, o pior caso dessa função ocorre quando a árvore binária está completamente desbalanceada, se tornando praticamente uma lista encadeada, e o e-mail pesquisado se encontra na folha da árvore ou não existe, dessa forma a pesquisa tem tempo . Já o melhor caso, ocorre quando o elemento pesquisado está na raiz da árvore, com tempo . Por fim, podemos tomar como base o caso médio da pesquisa, que é , em que representa a altura da árvore, aproximadamente .
* *remove:* Chama a função *removeRecursive*, com tempo médio .
* *removeRecursive:* Realiza operações constantes com tempo . Além disso, o algoritmo de remoção é muito semelhante ao algoritmo de pesquisa, assim, o pior caso dessa função ocorre quando a altura é igual à quantidade de elementos , portanto tem complexidade . Já o caso médio é muito dependente do valor da altura da árvore e do seu balanceamento, dessa forma, o caso médio dessa função tem tempo, aproximadamente, .
* *insert:* Chama a função *insertRecursive*, com tempo médio .
* *insertRecursive:* Realiza operações constantes com tempo . Além disso, o algoritmo de inserção é muito semelhante aos algoritmos de pesquisa e remoção, assim, o pior caso dessa função ocorre quando a altura é igual à quantidade de elementos , sendo uma árvore muito desbalanceada, portanto tem complexidade . Já o melhor caso ocorre quando a árvore está vazia, com tempo . Por fim, o caso médio é muito dependente do valor da altura da árvore e de seu balanceamento, dessa forma, o caso médio dessa função tem tempo, aproximadamente, .
* *printInOrder:* Chama a função *printInOrderRecursive,* com tempo
* *printInOrder:* Realiza operações constantes com tempo . Além disso, itera recursivamente todos os elementos da árvore, com tempo . Dessa forma, a complexidade assintótica da função é .]

**3.1.4 Hash\_BT**

A classe *Hash\_BT* representa uma tabela Hash de árvores binárias de tamanho e que possui registrado e-mails no total, contabilizando os e-mails de todos os usuários. Nesse sentido, a análise de complexidade torna-se um pouco mais complexa, portanto, para generalizar e simplificar a análise, vamos supor que cada árvore binária da tabela possui, em média, elementos, o que tornará essa seção mais objetiva e consistente.

* *Hash\_BT:* Realiza operações constantes de atribuição de valores, com tempo .
* *hash:* Realiza operações constantes, com tempo .
* *search*: Realiza operações constantes com tempo . Além disso, chama a função hash, tempo e a função *search* de uma árvore binária de elementos, com tempo . Dessa forma, a complexidade assintótica é .
* *insert*: Realiza operações constantes com tempo . Além disso, chama a função hash, tempo e a função *search* da classe com tempo . Por fim, chama a função *insert* de uma árvore binária de elementos, com tempo . Dessa forma, a complexidade assintótica é .
* *remove:* Realiza operações constantes com tempo . Além disso, chama a função hash, tempo e a função *remove* de uma árvore binária de elementos, com tempo . Dessa forma, a complexidade assintótica é .
* *print*: Realiza operações constantes com tempo . Além disso, itera todos os elementos da tabela Hash, com tempo . Dessa forma, a complexidade é .

**3.1.5 Main**

O arquivo Main lida com apenas uma função que orquestra outras três, essa função é chamada *readInputFile*. Essa função lê todo o arquivo de entrada, executa as operações requisitadas e imprime o arquivo de saída. As três funções orquestradas são as seguintes:

* *sendEmail:* Realiza operações constantes com tempo . Além disso, realiza uma iteração por todas as palavras da mensagem passada no arquivo de entrada, com complexidade . Além disso, chama a função *insert* da classe Hash, com tempo . Dessa forma, como cada mensagem apresenta no máximo 200 palavras, podemos considerar como 200, assim a complexidade assintótica da função é .
* *consultEmail:* Realiza operações constantes com tempo . Além disso, chama a função *search* da classe Hash, com tempo .
* *removeEmail:* Realiza operações constantes com tempo . Além disso, chama a função *remove* da classe Hash, com tempo .

Dessa forma, a complexidade assintótica de todo o programa, depende de três variáveis, todas dependentes do arquivo de entrada. São elas, o número de árvores binárias na tabela Hash, o número de e-mails enviados e armazenados no sistema, e o número de operações requisitadas, dessa forma, o programa como um todo apresenta tempo médio

A complexidade encontrada, descarta uma série de complicações de casos de entrada, para tornar a análise mais simples, já que se considerássemos todas as possibilidades de entrada, a análise de complexidade se tornaria muito mais complexa. Um exemplo, é a inserção, que muda o valor de e-mails no sistema, ou seja, a primeira inserção apresenta complexidade , enquanto a última apresenta complexidade . Um caso semelhante é a remoção, que diminui o número de e-mails, sendo assim, a primeira remoção apresenta complexidade , enquanto a última tem tempo . Por fim, concluímos que o tempo assintótico do programa é muito inconstante, e depende muito da natureza do arquivo de entrada.

**3.2 Análise de complexidade de espaço**