Trabalho Prático 3
Servidor de Emails Otimizado

Lucas Rocha Laredo

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lucaslaredo@ufmg.br

1. Introdução

A proposta deste trabalho prático foi a implementação de um programa que simula um servidor de emails através de uma estrutura de Tabela Hash, utilizando uma árvore binária para tratar as colisões. O sistema deve ser capaz de armazenar, remover e consultar os emails para diferentes pessoas, com chaves próprias específicas. A fim de solucionar o problema proposto neste trabalho, o projeto é capaz de lidar com arquivos, lendo-os, gerando-os e manipulando-os.

2. Implementação:

O programa foi desenvolvido na linguagem C++, utilizando o compilador G++, da GNU Compiler Collection.

2.1.1. Estrutura de Dados:

A implementação deste programa utiliza uma Estrutura de Dados (ED) para lidar com as possíveis colisões dentro de uma Tabela Hash: A Árvore Binária de Pesquisa, tendo como base o princípio da Programação Orientada a Objetos, possibilitado pela Linguagem C + +. A escolha dessa estrutura foi indicada no enunciado do projeto, entretanto, ela é muito importante para diminuir a complexidade do programa, tanto do ponto de vista de memória quanto de custo computacional. Notadamente, Listas Simples, ou ainda Listas mais complexas, como uma Lista Encadeada, poderia, facilmente, substituir a árvore binária - isso ficará mais claro logo adiante, quando explicarmos qual o uso desses arrays, nesse caso, a

implementação seria mais simples, contudo, a árvore binária de pesquisa promove um melhor tratamento das colisões.

Na implementação deste programa, trabalhamos com um *array* simples de árvores binárias. Ou seja, temos uma outra estrutura de dados, o *array*. Essa outra estrutura é fundamental, pois são necessárias *M* árvores binárias, no caso, M representa o tamanho da Tabela Hash. Logo, ao instanciar-mos uma nova tabela, o método construtor aloca espaço na memória para esse array de árvores de tamanho *M*.

```
Hash_LE::Hash_LE(int M) {
    this->Arvore = new ArvoreBinaria[M];
}
```

Essa ED armazena nos seus nós um Tipo Abstrato de Dados construído especificamente para este programa: os *emails*. Isso será tratado mais adiante com mais amplitude, entretanto, é importante mencionar que a comparação dos nós dentro dos procedimentos de busca, inserção e remoção é feita por meio da chave de cada mensagem, que é atribuída ao atributo *idMensagem*.

A Estrutura de Dados principal desse programa é capaz de lidar com três operações fundamentais, que são essenciais para as necessidades do projeto: Inserção, Busca e Remoção. As Árvores Binárias possuem uma característica própria que auxilia nos processos citados anteriormente. Ela lida com nós e seus filhos, que criam "sub-árvores". As subárvores da esquerda do nó principal sempre serão menores do que o valor chave do nó principal e da direita, sempre maiores do que o valor da chave do nó principal. Essa característica auxilia e lida com remoção, inserção e busca de uma modo mais eficiente do que as listas encadeadas convencionais, como pode ser percebido na figura a seguir:

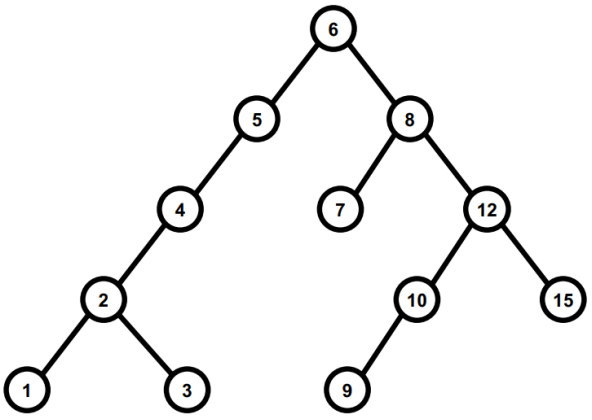


Figura 1. Ilustração de uma Árvore Binária de Pesquisa

Entretanto, é importante mencionar que há também o curto para manter as condições da árvore binária. A cada inserção ou remoção é importante garantir as condições apresentadas anteriormente, e isso representa um custo computacional considerável, da ordem de *O(n)*. Ou seja, ***ainda que a árvore binária seja um aprimoramento de uma lista encadeada convencional, é possível APRIMORAR AINDA MAIS*** **esse programa através da implementação de uma árvore *pseudobalanceada*, como uma árvore AVL.** Essa Estrutura trabalha com uma árvore que não está na sua totalidade balanceada, pois isso oferece um custo computacional considerável, mas com rotações, da ordem de *O(1)*, que deixam a árvore com uma organização favorável aos processos fundamentais: Inserção, Remoção e Busca. Contudo, no caso deste programa, não utilizamos uma estrutura de árvore que balanceia a balanceia a cada execução, que teria custo *O(n)* - outro que é minimizado pela árvore AVL - e isso é negativo do ponto de vista computacional, uma vez que, a medida que a árvore cresce, a busca se torna mais complexa.

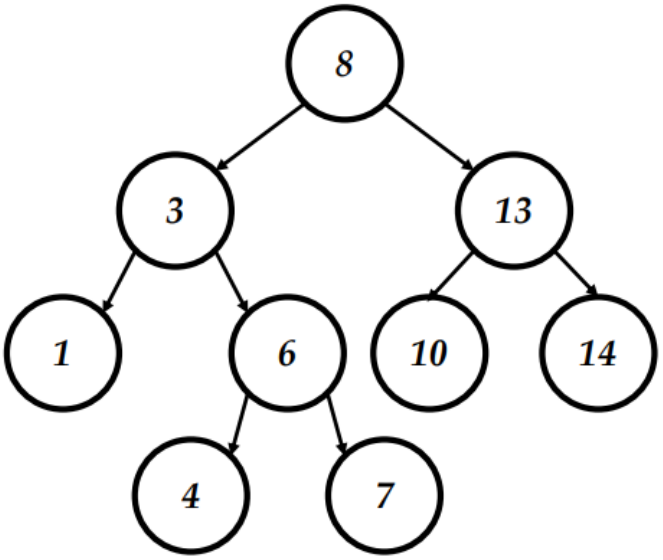


Figura 2. Ilustração de uma Árvore AVL balanceada

2.1.2. Tipos Abstratos de Dados:

A implementação deste programa conta com três Tipos Abstratos de Dados: *email, servidor* e *arvore_binaria* . Esses TADs correspondem respectivamente ao conteúdo do servidor, à Tabela Hash e a Estrutura de Dados fundamental da Tabela: A árvore Binária.

Em princípio, vamos começar com o tipo *email*. Com um nome bastante intuitivo, ele é responsável por armazenar os atributos e métodos necessários para o TAD que irá carregar

o servidor que estamos implementando. Portanto, temos os atributos *idMensagem, idDestinatario, tamanhoMensagem, mensagem* e o nome de saída do arquivo principal, importante para a impressão do programa. No caso, cada email (*mensagem)* possui um identificador próprio, que é importante para calcular a entrada na posição correta da *Tabela Hash*, por meio da operação algébrica *U % M,* sendo que *U* é o *idMensagem.* Além disso, o *idDestinatário* é importante para armazenar o email na posição correta da árvore binária correspondente àquela mensagem. Por fim, o tamanho mensagem e a mensagem correspondem de fato ao conteúdo do email, ou seja, aquilo que deve ser de fato armazenado. A mensagem é um tipo *string*, ou seja, basicamente é um *array* de caracteres.

Em seguida, temos o TAD *servidor.* Esse TAD corresponde a uma Tabela Hash que simula um servidor de emails. ele contém os métodos importantes para o funcionamento dessa tabela, ou seja, métodos de inserção, remoção e pesquisa. Essa tabela é uma tabela de *emails* que utiliza uma árvore binária para tratar as colisões, como já apresentado anteriormente.

Por fim, o TAD *arvore_binaria* corresponde a Estrutura de Dados fundamental do programa, que é utilizada para tratar as colisões dentro da tabela hash. Esse TAD possui uma classe amiga, *tipoNo,* que é equivalente aos nós da árvore. Essa Classe contém ponteiros para esquerda e direita, assim como uma instância do TAD *email*, uma vez que, como abordado anteriormente, é o que será armazenado nos nós das árvores. Os principais métodos da Classe *ArvoreBinaria* correspondem a Inserção, Pesquisa e Remoção, contando com métodos auxiliares. A explicação desses métodos virá adiante, na sessão a seguir.

```
class ArvoreBinaria
{
    public:
        ArvoreBinaria();
        ~ArvoreBinaria();
        void Insere(Email item);
        void Caminha(int tipo);
        Email Pesquisa(Email msg);
        void Remove(Email chave);
        void Limpa();
        void InsereRecursivo(TipoNo* &p, Email item);
        void ApagaRecursivo(TipoNo* p);
        Email PesquisaRecursivo(TipoNo *no, Email msg);
        void RemoveRecursivo(TipoNo* &p, Email chave);
        void Antecessor(TipoNo* q, TipoNo* &r);
    TipoNo *raiz;
};
```

Figura 3. Ilustração da Classe *ArvoreBinaria*

2.2. Métodos e Procedimentos utilizados e Funcionamento:

A implementação deste programa é levemente robusta, contando com três Tipos de Abstratos de Dados distintos e isso também é refletido na quantidade de métodos e procedimentos existentes. No caso, vamos começar pela função principal.

A função principal - *main* - é onde se encontram as principais chamadas para as funções e métodos que dão sequência ao funcionamento do programa. Dentro da *main* estão os métodos de leitura do arquivo principal, que, a cada linha lida, chamam os métodos correspondentes à operação solicitada dentro da Tabela Hash, como será citado adiante. Além disso, estão também os métodos de leitura dos parâmetros da linha de comando e de *"reset"* do arquivo de entrada.

Dando sequência, passamos para os métodos existentes dentro da *Tabela Hash,* que é o tipo de dados que simula o servidor de emails. Dentro desse TAD está o método construtor, que instancia um *array* de *Árvores Binárias* de tamanho *M*, passado no arquivo de leitura, juntos aos três métodos que correspondem às três operações necessárias: Pesquisa, inserção e remoção. Antes de entrar nesses métodos, é importante reforçar que o tratamento de colisões da tabela hash foi trabalhado através da estrutura Árvore Binária, logo, os três métodos citados anteriormente são implementados seguindo a implementação das mesmas três operações dentro dessa Estrutura de Dados. Logo, seguimos para os métodos.

Em primeiro lugar, o método de Pesquisa da tabela utiliza de um outro método secundário que retorna a posição a ser pesquisada por meio da operação algébrica fornecida no enunciado do projeto. Após descobrir a posição é chamado o método de pesquisa para a árvore de binária equivalente a posição encontrada dentro do array de árvores. O método de pesquisa de uma árvore binária de pesquisa é implementado recursivamente, e confere os valores das chaves dentro dos nós. Caso o valor seja maior, navegamos na subárvore da direita, caso contrário, na subárvore da esquerda. Aqui cabe dizer que não balanceamos a árvore a cada inserção ou remoção, o que é negativo do ponto de vista computacional, uma vez que a medida que a árvore cresce, os procedimentos de inserção, remoção e busca se tornam mais custosos. Logo, a implementação de uma árvore AVL seria positiva, ao passo que com as rotações a árvore seria pseudobalanceada e o método de pesquisa se tornaria mais eficiente, diminuindo a ordem de complexidade do programa, pois as rotações possuem custo *O(1)*. Retornando para o método de busca da tabela, caso a posição buscada seja encontrada, a mensagem é apresentada no arquivo de saída.

O processo de inserção é bastante semelhante, contudo, agora o método da árvore chamado é o método de inserção. Esse método também é recursivo e conta com as mesmas particularidades do apresentado anteriormente. A sua eficiência depende do formato das árvores, ou seja, pode apresentar grandes variações. Entretanto, o mesmo dito anteriormente ainda é válido, as árvores AVL são mais eficientes, pois essa variação não existe, ou seja, minimizamos o pior caso.

Por fim, temos o método de remoção, que, assim como os demais, possui o tratamento de robustez que será apresentado adiante e faz a chamada para o método correspondente dentro da árvore binária. Assim como os demais, há uma pesquisa dentro da árvore para o nó desejado. Uma vez encontrado, ele é removido e um procedimento auxiliar é chamado para reorganizar os ponteiros para os nós restantes, uma vez que, ao remover um nó, o anterior deve apontar para a posição correta que restou na árvore.

Para finalizar, restou o método *limpa* da árvore binária, que funciona como um destrutor. Ou seja, quando queremos desalocar a memória ocupada pela árvore, utilizamos esse método para deletar todos os nós alocados da memória. Essa é uma boa prática que melhora a localidade de referência do programa.

É importante mencionar que, como será visto adiante, ainda que existam possíveis melhorias para o programa implementado, a Estrutura de Dados utilizada é eficiente e melhor do que outras alternativas existentes. Por exemplo, o custo de inserção de uma lista encadeada é muito maior do que de uma árvore binária. O mesmo é válido para ordenação e remoção, ou seja, a árvore é mais favorável para a aplicação que estamos implementando, especialmente pelo fato de que, em uma lista encadeada existem múltiplos possíveis algoritmos de pesquisa e ordenação que em sua maioria não são tão eficientes, seja por um alto custo computacional, ou por uma complexa implementação. Contudo, ainda assim, a implementação de uma árvore pseudobalanceada, como a AVL, é positiva pelos ganhos citados anteriormente. Tanto a ordenação quanto a pesquisa são menos custosas, o que é positivo na balança final do programa. Portanto, existem possíveis ***aprimoramentos*** a serem implementados, ainda que a implementação esteja boa

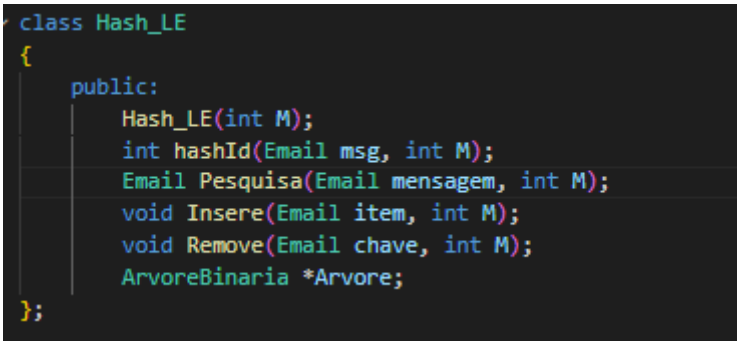


Figura 4. Implementação da Classe equivalente a *Tabela Hash*

3. Análise de Complexidade:

Começaremos a Análise de Complexidade pela função principal: a *main.* Em primeiro lugar, essa função conta com uma série de loops, que aumentam a sua complexidade. O primeiro deles é responsável pela leitura dos parâmetros de entrada padrão do programa *(argc e argv).* No caso, temos um loop *O(n),* sendo *n = argc,* tendo melhor caso *O(1)*. Em seguida, vem a leitura de arquivos, um método simples, mas que pode oferecer um certo custo computacional. A leitura do parâmetro *M* é da ordem *O(1),* pois apenas a primeira linha do arquivo é lida. Em seguida, temos o restante do arquivo, que é lido linha após linha, na ordem em que elas são apresentadas. Ou seja, o loop principal envolve as *n* linhas do arquivo de entrada, logo, em si ele já teria complexidade *O(n)*. Entretanto, caso a operação escolhida seja a *Inserção*, é importante concatenar as *N* palavras na string mensagem, um atributo do TAD email. Logo, no pior caso, a leitura de arquivos é da ordem $O(n^2)$. Sendo assim, a função principal fica com a ordem *O(n)* no melhor caso e $O(n^2)$ no pior caso, assim como a leitura de arquivos.

Em seguida vamos aos Tipos Abstratos de Dados e seus métodos. O TAD *email* é muito pouco custoso computacionalmente, uma vez que possui custo *O(1)*. Entretanto, os demais já são mais custosos. Em primeiro lugar, temos a Tabela Hash. Como a Tabela Hash utiliza as árvores binárias para tratamento de colisão, os custos computacionais desse TAD estão muito associados com os custos da Estrutura de Dados fundamental utilizada por ele. Nesse caso, essas serão apresentadas através da imagem a seguir:

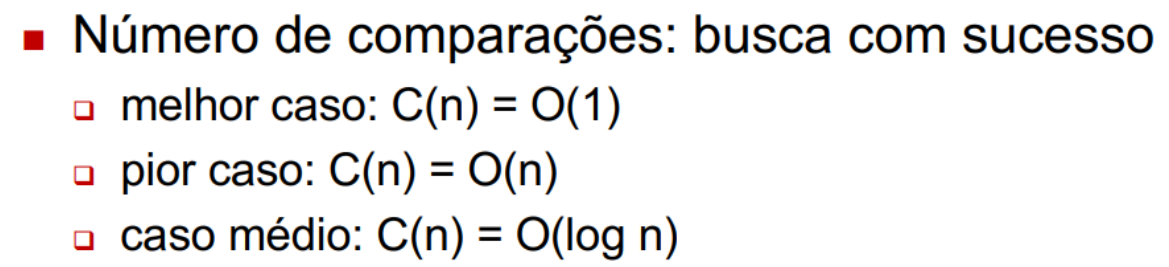


Figura 5. Custo Computacional Árvore Binária de Pesquisa

De modo bastante sucinto, a figura anterior ilustra perfeitamente os custos de uma árvore binária de pesquisa. Dentro dessa Estrutura de Dados, ao fazer uma inserção ou remoção também é necessária uma busca, para encontrar a posição correta do nó a ser inserido ou removido. Ou seja, os custos dos três procedimentos compartilham a mesma *ordem de complexidade*, que foi apresentada anteriormente. ***É* importante mencionar que** a implementação de uma árvore AVL seria um ***aprimoramento***, uma vez que minimizaria os custos computacionais no pior caso, oferecendo sempre um custo da ordem *O(nlogn)* o que é interessante do que o custo computacional anterior. Uma ***desvantagem*** é a maior complexidade de implementação da estrutura de uma árvore pseudo balanceada, sendo uma opção para projetos mais robustos e que definitivamente tratam com um maior volume de dados.

Portanto, concluímos que esse programa oferece um custo computacional aceitável, com uma ordem de complexidade relativamente alta no pior caso, para mensagens muito extensas, entretanto, a busca, inserção e remoção dos elementos da tabela hash não são de ordem de complexidade muito complexa, o que é bastante interessante. É importante mencionar que, como dito anteriormente, essa complexidade poderia ser potencialmente melhorada através de uma outra Estrutura de Dados, como por exemplo uma árvore AVL.

3.2. Espaço:

A complexidade de espaço desse programa pode toda ser pensada em torno da sua estrutura de dados: A árvore binária. No caso, temos, dentro dos servidores, um *array dinâmico* de árvores, que irá possuir tamanho M, ou seja, o tamanho da Tabela Hash, ou seja, possui complexidade *O(M)*. É importante mencionar que o uso das árvores binárias é interessante, pois os seus métodos de ordenação, busca, inserção e remoção não necessitam de nenhuma estrutura externa que ocuparia mais posições na memória. Entretanto, ainda existem os custos de memória dentro das árvores binárias. Essas árvores contam com nós,

que armazenam os TADs *email*. Esses tipo nós são instanciados dinamicamente dentro de cada árvore e possuem ponteiros para os nós da esquerda e direita, consecutivamente. Como uma árvore pode possuir *n* nós, a complexidade de espaço é da ordem *O(n)* no pior caso e *O(1),* se a árvore possuir apenas um nó.

4. Estratégias de Robustez:

Todo programa mais robusto exige mecanismos para evitar bugs. Desse modo, para a depuração foi utilizado o gdb. As estratégias de robustez deste programa estão principalmente em torno das inconsistências nas requisições dentro do servidor de email. Ou seja, essas correspondem a operações com entradas inconsistentes. Por exemplo, para a operação de consulta, caso o *id* do destinatário não exista dentro do servidor é necessário que isso seja tratado, o mesmo vale para a remoção dos emails. Este tratamento é feito diretamente dentro dos procedimentos do servidor que são responsáveis por essas operações. No caso da remoção, por exemplo, caso o identificador da mensagem seja o padrão, -1, uma mensagem de erro é apresentada, caso contrário, a mensagem é corretamente apagada. O mesmo é válido para a consulta, entretanto, neste caso, além do id mensagem, o identificador do destinatário também deve ser o mesmo da mensagem a ser conferida, caso contrário uma mensagem de erro é apresentada. Além disso, também é importante um tratamento para inconsistências dentro dos parâmetros de entrada do programa. É fundamental que existam arquivos de entrada e saída, caso contrário o programa não é executado. Isso é importante pois a própria execução é prejudicada caso essas entradas não sejam fornecidas.

5. Testes:

Os testes realizados visam dois objetivos: Estimar a complexidade do programa para entradas de grande porte e o seu funcionamento, do ponto de vista de localidade de referência e depuração de desempenho. Em primeiro lugar, foram utilizadas as entradas fornecidas, para testar o funcionamento do programa. Após finalizar a implementação, essas entradas foram modificadas da seguinte maneira:

O arquivo *entrada_1.txt* foi alterado, ao passo que *n* repetições do texto foram testadas. No caso da localidade de referência, utilizando a biblioteca *memlog*, os testes realizados se basearam em 15 repetições da entrada fornecida. Por outro lado, para estimar o tempo de execução do programa, foram tomadas variações dessas entradas, começando com 220 repetições, dobrando-as até a marca de 1680 repetições. Ou seja, um texto com 220

repetições do texto inicial, um com 440, assim por diante, foram testadas. Esses testes auxiliaram na determinação do custo computacional do programa, assim como o seu tempo de execução. Quanto maior o desafio, mais necessário é um controle em relação à sua estabilidade e segurança de suas respostas. Por esse motivo, foram realizados múltiplos testes, possibilitando assegurar a qualidade do algoritmo.

5.1. Tempo de Execução:

A análise do Tempo de Execução do algoritmo implementado teve como base a variação na entrada dos dados, ou seja, no texto presente no *input*. Como citado anteriormente, foram utilizadas variações da entrada *entrada_1.txt* disponibilizada. Para o teste do tempo de execução, foram testadas variações com 220, 420 e 840 e 1680 repetições do texto, sendo que os testes foram realizados na mesma máquina, sob mesmas condições. O gráfico a seguir apresenta os resultados:

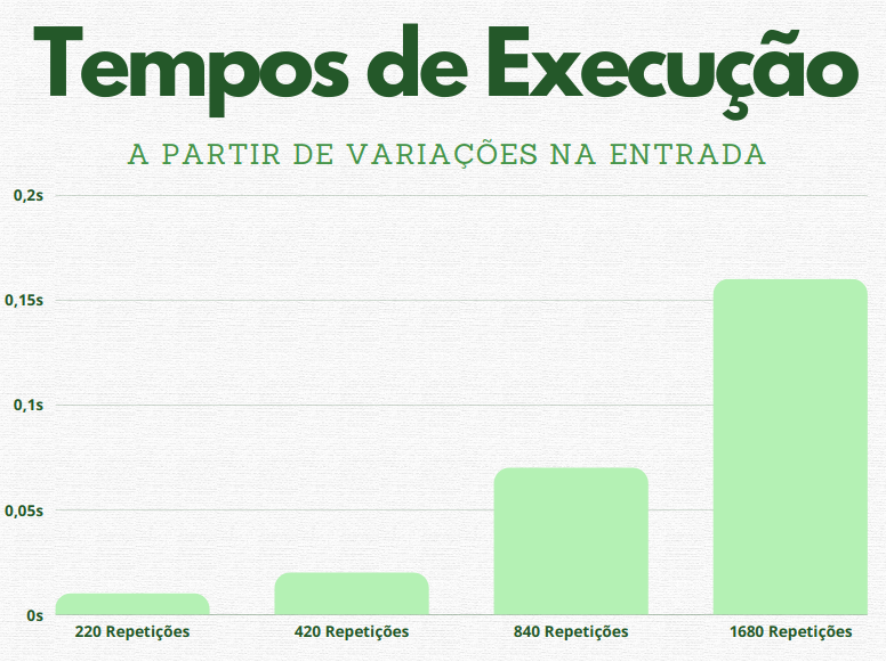


Figura 6. Variação dos tempos de execução

A análise do gráfico disponibilizado anteriormente permite concluir o que já era esperado: A variação nas entradas geram sim diferenças nos tempos de execução, entretanto, como a ordem de complexidade do programa não é elevada, o tempo de execução não necessariamente duplica à medida que as entradas duplicam. Ou seja, a ordem de complexidade do programa não é linear, mas sim próximo à logarítmica. A grande explicação para esse fenômeno é o uso das árvores binárias de pesquisa, que são interessantes justamente por fornecer esse tipo de ordem de complexidade. Entretanto, como dito anteriormente, uma

vez que não usamos árvores completamente balanceadas, ou pseudobalanceadas, ao passo que as árvores crescem, o programa se torna cada vez mais complexo.

6. Análise Experimental:

Agora que os resultados já foram apresentados, faremos a análise. Para facilitar a compreensão, iremos separá-la em subseções.

6.1. Desempenho Computacional:

O desempenho computacional é bastante afetado pela forma que o programa é implementado. Funções otimizadas, ou seja, com complexidades preferenciais, como *O(log n)* oferecem um melhor desempenho computacional ao programa. E isso é interessante, uma vez que o programa foi implementado utilizando uma estrutura que fornece uma ordem de complexidade interessante, como apresentado anteriormente.

A métrica utilizada levou em consideração as análises de complexidades das funções citadas nessa documentação e para análise, foram utilizados arquivos de textos que são variações das entradas disponibilizadas, como anteriormente apresentado. No caso, para essa análise experimental de desempenho computacional levamos em conta a variação com 1680 entradas. É fundamental mencionar que o tempo de execução possui uma leve variação ao apresentado anteriormente, isso é normal, pois, ainda que realizados na mesma máquina, os processos momentâneos, simultâneos ao programa podem influenciar no desempenho.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 81.35      0.26     0.26    15360    16.95    20.36  ArvoreBinaria::InsereRecursivo(TipoNo*&, Email)
  9.39      0.29     0.03 13108795     0.00     0.00  Email::~Email()
  6.26      0.31     0.02 13060795     0.00     0.00  Email::Email(Email const&)
  3.13      0.32     0.01    51840     0.19     0.19  Email::Email()
  0.00      0.32     0.00    86419     0.00     0.00  bool std::operator==<char, std::char_traits<char>, std::allocator<char> >(std
  0.00      0.32     0.00    69121     0.00     0.00  int __gnu_cxx::__stoa<long, int, char, int>(long (*)(char const*, char**, int
  0.00      0.32     0.00    69121     0.00     0.00  std::__cxx11::stoi(std::__cxx11::basic_string<char, std::char_traits<char>, s
  0.00      0.32     0.00    69121     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long (*)(char const*, char**, int), c
  0.00      0.32     0.00    69121     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long (*)(char const*, char**, int), c
  0.00      0.32     0.00    69121     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long (*)(char const*, char**, int), c
  0.00      0.32     0.00    26880     0.00     0.00  Hash_LE::hashId(Email, int)
  0.00      0.32     0.00    15360     0.00    20.36  ArvoreBinaria::Insere(Email)
  0.00      0.32     0.00    15360     0.00     0.00  Email::operator=(Email const&)
  0.00      0.32     0.00    15360     0.00     0.19  TipoNo::TipoNo()
  0.00      0.32     0.00    15360     0.00    20.37  Hash_LE::Insere(Email, int)
  0.00      0.32     0.00    11520     0.00     0.43  ArvoreBinaria::PesquisaRecursivo(TipoNo*, Email)
  0.00      0.32     0.00    11520     0.00     0.43  ArvoreBinaria::Pesquisa(Email)
  0.00      0.32     0.00    11520     0.00     0.00  Email::Email(Email&&)
  0.00      0.32     0.00    11520     0.00     0.00  Email::operator=(Email&&)
  0.00      0.32     0.00     9600     0.00     0.64  Hash_LE::Pesquisa(Email, int)
  0.00      0.32     0.00     1920     0.00     0.00  ArvoreBinaria::RemoveRecursivo(TipoNo*&, Email)
  0.00      0.32     0.00     1920     0.00     0.01  ArvoreBinaria::Remove(Email)
  0.00      0.32     0.00     1920     0.00     0.65  Hash_LE::Remove(Email, int)
  0.00      0.32     0.00       23     0.00     0.00  ArvoreBinaria::ArvoreBinaria()
  0.00      0.32     0.00        1     0.00     0.00  _GLOBAL__sub_I_M
  0.00      0.32     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN6TipoNoC2Ev
  0.00      0.32     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN7Hash_LEC2Ei
  0.00      0.32     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.32     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.32     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.32     0.00        1     0.00     0.00  resetaOut()
  0.00      0.32     0.00        1     0.00     0.00  testaNome(std::__cxx11::basic_string<char, std::char_traits<char>, std::alloc
  0.00      0.32     0.00        1     0.00     0.00  Hash_LE::Hash_LE(int)

%         the percentage of the total running time of the
```

Figura 7. Saída *gprof*

A partir da saída *gprof* apresentada anteriormente, podemos fazer uma análise mais aprofundada do desempenho computacional do programa. Em primeiro lugar, percebe-se que a leitura de arquivos, apesar de possuir muitas chamadas, não oferece um grande impacto do tempo de execução do programa. Na verdade, a grande influência nos tempos de execução está em torno especialmente das chamadas de inserção dentro das árvores binárias. No caso da execução anterior, cerca de 81% do tempo de execução esteve em torno das inserções na árvore. Esse alto custo já foi abordado anteriormente, e pode ser explicado pelo fato de que, para toda inserção, é necessária uma busca, ou seja, há um custo considerável. Além disso, a árvore implementada não é balanceada a cada inserção, logo, quanto maior ela se torna, mais complexa se torna a inserção. Esse alto custo é evidenciado na figura anterior, tendo um alto impacto no programa. Em seguida, uma questão relativamente surpreendente é o impacto da instanciação da Classe email no tempo de execução do programa. Ficou muito claro que os métodos construtores e destrutores dessa classe são bastante custosos e possuem o maior número de chamadas dentro do programa. Em especial, os demais procedimentos de pesquisa e remoção não tiveram um grande impacto no programa. Entretanto, o número de chamadas foi consideravelmente inferior ao de inserção. Ainda assim, o impacto foi muito menor, o que também já era esperado.

É fundamental compreender que, apesar do método de inserção na árvore binária ser levemente custoso, a sua eficiência é bastante tolerável. Contudo, fica evidente que a ordenação é sempre um problema complexo, ou seja, ainda que um método de ordenação ***extremamente eficiente*** seja utilizado, fica claro, pela análise gprof, que a comparação entre os elementos ainda é custosa. Novamente, esse tempo pode ser melhorado utilizando estruturas que tornem essa comparação e ordenação mais eficientes.

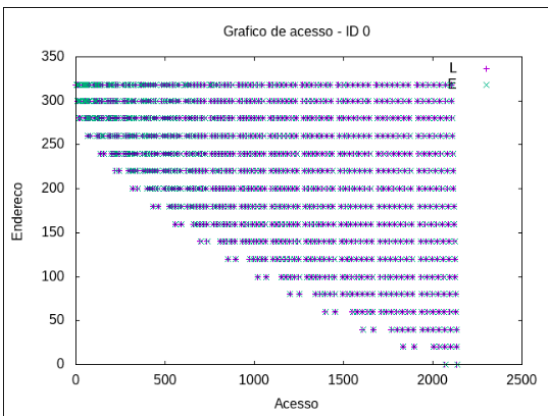
```
void ArvoreBinaria::Insere(Email msg){
    InsereRecursivo(raiz,msg);
}

void ArvoreBinaria::InsereRecursivo(TipoNo* &p, Email msg){
    if(p==NULL){
        p = new TipoNo();
        p->msg = msg;
    }
    else{
        if(msg.idMensagem < p->msg.idMensagem)
            InsereRecursivo(p->esq, msg);
        else
            InsereRecursivo(p->dir, msg);
    }
}
```

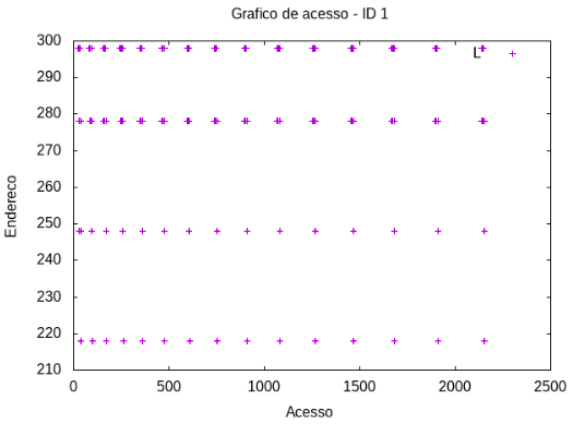
Figura 8. Função de Inserção da Árvore Binária de Pesquisa

6.2. Padrão de acesso à Memória e Localidade de Referência:

Para a análise de acesso à Memória e localidade de referência foi utilizada a entrada *entrada_1.txt*, disponibilizada no enunciado deste TP. Essa escolha busca encontrar um intervalo entre um programa de tamanho razoável, oferecendo desempenho suficiente para análise de localidade.

A análise de localidade de referência parte da divisão do programa em fases. Ou seja, para facilitar a interpretação dos gráficos e a análise do comportamento do programa, ele foi dividido em "partes" que refletem o seu funcionamento. A *fase 0* é o momento de inserção e dos emails na árvore binária equivalente àquela entrada na tabela hash. Ou seja, como visto anteriormente, espera-se que essa seja a fase mais custosa computacionalmente, e a seguir veremos qual o seu desempenho do ponto de vista da memória. Durante a *fase 1* temos a pesquisa dos emails dentro da tabela hash equivalente a cada destinatário. Nesse momento, há a pesquisa dentro da tabela hash de qual será a árvore de interesse e, dentro dessa árvore, qual é o email buscado. Por fim, temos a *fase 2*, que é a final. Essa parte do programa é responsável pela remoção dos emails indicados da árvore binária equivalente ao destinatário indicado. Nesse momento há a pesquisa, mas também a reescrita da árvore, uma vez que após a remoção, ela deve ser reestruturada.





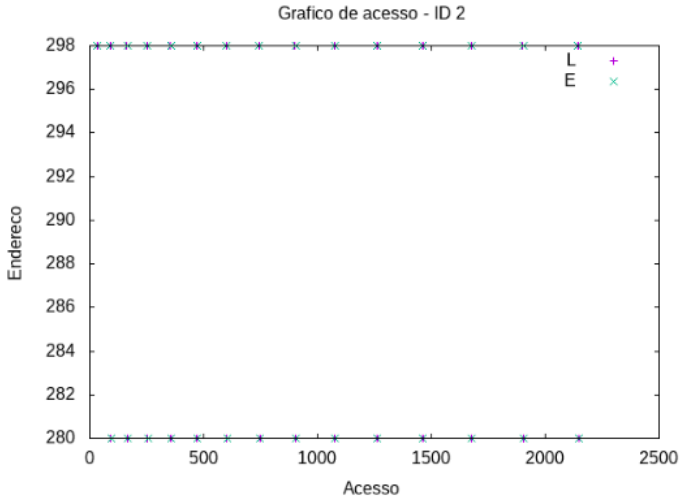


Figura 9. Gráficos de acesso à memória

Vamos começar essa análise na ordem sequencial das fases, que também corresponde à ordem lógica e implementação do programa. Percebe-se no gráfico de **ID 0** como a localidade de referência da memória está bem aplicada. No caso, as buscas e inserções são praticamente simultâneas, uma vez que, para a inserção é necessária a busca dentro da árvore binária. A boa localidade de referência fica visível pela proximidade dos acessos dentro da memória. Ou seja, os acessos estão em endereços próximos e acontecem de forma ordenada, o que facilita a busca e diminui o custo computacionalmente. temos cerca de 2100 acessos, todos muito próximos em termos de memória, ou seja, todos em um mesmo endereço. É interessante o formato do gŕafico, à medida que o número de acessos aumenta, o intervalo de endereços buscados diminui, o que é explicado pelo fato da árvore não estar balanceada devidamente. Contudo, ainda assim, do ponto de vista computacional essa fase é extremamente interessante, pois o acesso é *extremamente eficiente*. Logo, em termos de localidade de referência, pode-se dizer que a *fase 0* está muito bem.

Dando sequência, percebe-se que a fase 1 é bem distinta da anterior. No caso, essa é equivalente a busca dentro da árvore binária, ou seja, não existe escrita na memória. Essa é a primeira diferença, contudo, a mais evidente diz respeito à diferença na localidade de referência de ambas. Enquanto a primeira possui uma boa localidade de referência, a segunda não é tão interessante, uma vez que os acessos estão consideravelmente mais dispersos ao longo da memória. A explicação para esse fato vem da forma como a função é chamada dentro do programa. Na verdade, a construção da árvore binária implica uma ordem já estipulada para os nós, ou seja, os da esquerda devem ser menor do que o principal e os da direita maiores. Sendo assim, a pesquisa navega dentro das subárvores que, pela condição de existência da árvore binária possuem uma dispersão maior dentro da memória. Portanto, ainda que a localidade de referência dessa fase não seja a mais interessante, é o melhor possível para uma árvore binária de pesquisa, e ainda assim, não é tão negativa, ainda oferece uma localidade de referência consideravelmente boa.

Por fim, a *fase 2* é completamente diferente das anteriores. O menor número de chamadas implica também em uma menor ocupação dentro do gráfico, entretanto, ainda assim percebe-se uma maior dispersão entre os pontos. No caso, o mesmo dito anteriormente para a fase 1 é válido. Entretanto, a grande questão é que os nós estão ainda mais dispersos em uma remoção devido a organização de uma árvore binária de pesquisa. Dessa forma, o

gráfico de acesso reflete exatamente esse comportamento, apresentando os pontos bastante dispersos ao longo da memória.

Agora, para uma análise mais ampla, visando a quantificação dos acessos, faremos uma análise das distâncias dessa mesma operação:

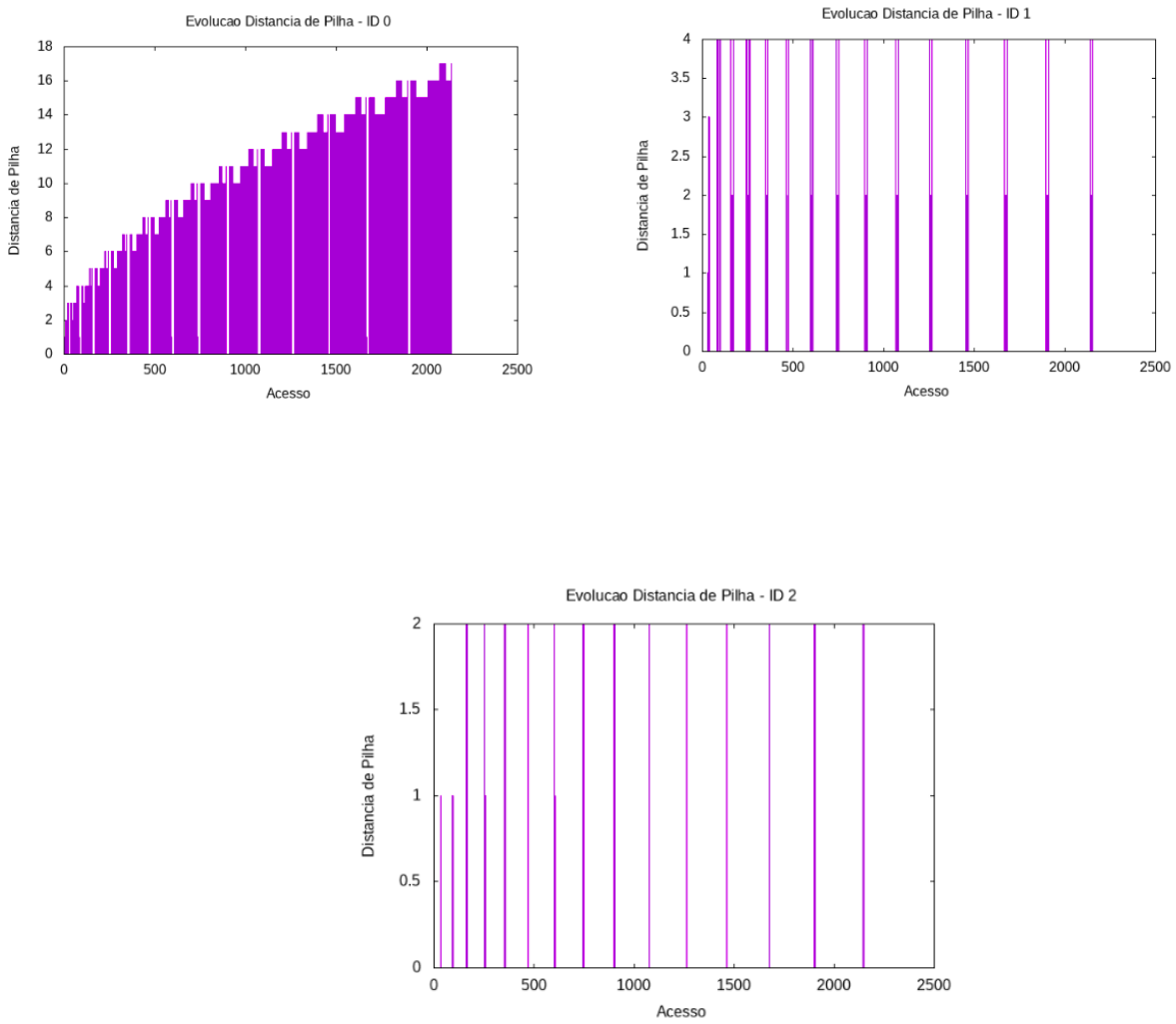


Figura 10. Gráficos de distância de pilha

Os três gráficos de distância de pilha são bastante distintos e evidenciam o que foi visto anteriormente: a diferença na execução de cada função. Em respeito a fase 0, ou seja, a inserção, a distância de pilha possui acessos muito concentrados em que a distância de pilha aumenta à medida que o número de acessos também aumenta. A maior distância de pilha é aproximadamente 17, e isso significa que o maior caminho percorrido dentro da árvore binária para a inserção é de 17, o que é consideravelmente expressivo. Nas fases seguintes, ou seja, 1 e 2, a distância de pilha é de uma ordem muito menor, entretanto, os acessos estão mais dispersos em questão de quantidade de acessos, uma vez que essas funções são chamadas menos vezes.

Por fim, apenas citamos os histogramas de distância de pilha que confirmam o que foi dito anteriormente:

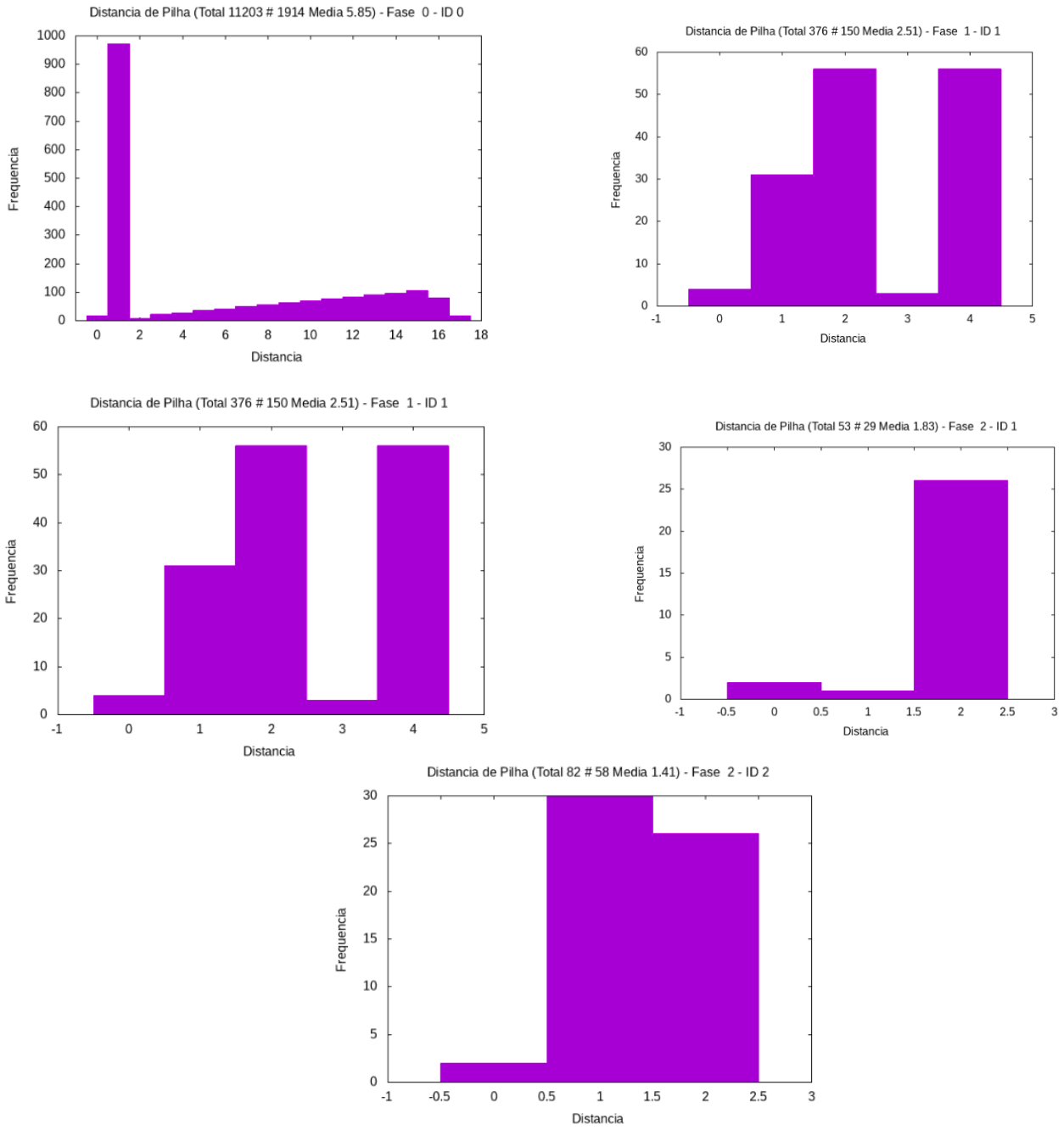


Figura 11. Histograma de distância de pilha

7. Conclusão:

O programa implementado envolveu o desenvolvimento de um algoritmo capaz de simular uma estrutura de um servidor de emails utilizando uma tabela hash e uma árvore binária de pesquisa para tratamento de colisões. Este trabalho definitivamente instigou o desenvolvimento de diversas técnicas que envolvem a programação. Esse desenvolvimento extrapolou apenas o desenvolvimento em si, mas também atingiu níveis como abstração, percepção da realidade, pesquisa, solução de problemas e o planejamento de software.

Sem dúvida, o criar um programa com um certo nível de complexidade como esse, partindo do zero, exige um certo planejamento, ao passo que seja possível uma modularização de código mais eficiente, permitindo atingir melhores resultados mais a frente, tanto em relação ao aproveitamento das Classes e Objetos, quanto aos resultados finais em si. Nesse sentido, a abstração também é desenvolvida, uma vez que é importante modularizar o código, à medida que, a implementação de alguns métodos e procedimentos não fiquem evidentes a um possível usuário final, mas apenas interesse o seu uso. Entretanto, em um sentido mais amplo, a abstração é fundamental para conseguir capturar as nuances de um problema e transformá-las em Classes, TADs, métodos e atributos dentro do código.

Além disso, ainda na implementação, esse trabalho proporcionou o desenvolvimento do uso de Estruturas de Dados de forma mais concreta, ou seja, exigindo a implementação dessas estruturas.

Por fim, houveram ganhos '*extras'* ao longo do desenvolvimento. Em um projeto amplo é esperado o surgimento de *bugs* e erros sucessivos. A fim de contornar esses problemas, é importante a pesquisa e a depuração. Logo, esse programa ainda proporcionou o desenvolvimento da capacidade de encontrar os problemas e solucioná-los, o que é fenomenal, dado que essa é uma competência fundamental dentro da Ciência da Computação.

Sendo assim, a implementação do TP3 foi fundamental para o meu crescimento, tanto profissional quanto pedagógico. Lidar com novas ferramentas ofereceram uma oportunidade para aumentar meu conhecimento, e esse aprendizado me instigou a correr atrás de novos instrumentos, o que aumentou o meu repertório. Portanto, a implementação do TP3 ofereceu grande margem de aprendizado, em múltiplas competências, como citado anteriormente.

8. Bibliografia:

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++:˜ Capítulo 3: Estruturas de Dados Básicas. Editora Cengage.

9. Instruções para compilação e execução:

A compilação do programa é "automática", via Makefile, implementado para esse programa. O comando "*make all"* deve ser realizado via terminal na raiz do diretório TP3. Esse comando irá compilar o programa, gerando o executável e todos os arquivos **.out* necessários para o funcionamento. Caso seja necessário "*reiniciar"* a compilação, o comando "*make clean"* apaga todos os **.o* e **.out*, permitindo uma recompilação.

Após compilar, o programa implementado terá o seu '*executável'* disponível na pasta bin. Esse executável terá nome *tp.exe* e **nessa pasta deve estar contido o arquivo de entrada** com **NOME** *indicado pela flag -i.*

ATENÇÃO: *Caso o arquivo de entrada.txt não esteja presente na pasta 'bin', a execução do programa será interrompida, e uma mensagem de erro irá indicar o problema.*

A execução do programa é bastante simples, e a saída **deve ser determinada pelo usuário,** por meio da flag *-p,* e também ficará disponível na pasta bin.

Seguem as flags esperadas via linha de comando:

-o: Determina o nome do arquivo de saída do programa. *Exemplo de uso:*

./executável -o [OUTPUT.o] (...)

-i: Determina o nome do arquivo de entrada. *Exemplo de uso:*

./executável -i [INPUT.i] (...)

ATENÇÃO: TODAS *as flags citadas anteriormente são* ***OBRIGATÓRIAS*** *para o funcionamento do programa. Caso alguma dessas flags não estejam presentes, o programa* ***não irá executar.***

ATENÇÃO: *A ordem em que cada flag aparece* ***NÃO É RELEVANTE PARA O FUNCIONAMENTO DO PROGRAMA.***