Documentação TP3

Aluno: Henrique Daniel de Sousa

Matrícula: 2021031912

Introdução

Essa documentação tem por objetivo apresentar um programa que desenvolve um simulador de um servidor de e-mails, no qual pode-se fazer operações de inserção, busca e remoção de e-mails. A inserção simula o recebimento de uma mensagem; a busca, a pesquisa sobre alguma mensagem; a remoção, a exclusão de uma mensagem.

Implementação

O código está organizado entre arquivos ".h", que implementam os TADs "Mensage", "BST", "Node" e "Hash", e arquivos ".cpp" que implementam as funções de cada TAD e as executam. A execução das funcionalidades criadas é feita no arquivo "server.cpp".

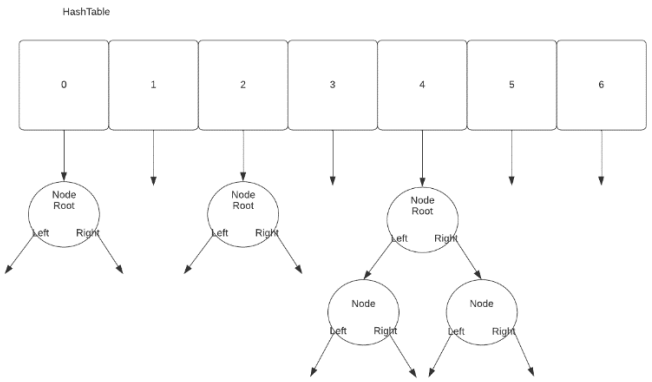
TAD Mensage: Uma mensagem é definida por uma string *"content"* e um inteiro *"key"*, que guarda a chave da mensagem inserida.

TAD Node: Um Node é definido por uma variável do tipo Mensage e dois ponteiros do tipo Node ,*"left"* e *"right"* .

TAD BST: Esse TAD define uma Árvore Binária de Busca (Binary Search Tree). Cada BST tem um Node que define sua raiz. Desse modo, é possível inserir, remover e procurar outros nodes a partir da raiz.

TAD Hash: O TAD Hash define uma tabela Hash na qual, cada posição referencia uma árvore binária de busca (BST). Cada posição corresponde a um diferente usuário do servidor de e-mails.

Dessa forma, a estrutura de dados implementada segue o seguinte modelo:



O programa recebe como parâmetros:

1. O nome de um arquivo de entrada, que tem o número de usuários, utilizado para criar a tabela Hash e as inserções, pesquisas e remoções utilizadas pelo email.

2. Um arquivo de saída, no qual são impressos os resultados das ações dos usuários.

3. Um arquivo para salvar o registro de acesso e o tempo de execução do programa.

A implementação desse problema foi realizada usando a linguagem C++ e compilada com o G++. Para a documentação desse problema, a máquina utilizada tem o Windows10(WSL2) como sistema operacional, um processador Intel(R) Core(TM) i5-9300H CPU @2.40GHz 2.40 GHz e 8GB de memória RAM.

Análise de Complexidade

Inserção de palavras:

Função *'BST::insert()'*: Essa função insere uma palavra numa árvore binária. Portanto, sua complexidade de tempo é O(1), no melhor caso, O(log n) no caso médio e O(n), no pior caso. O pior caso ocorre quando uma BST se transforma numa lista encadeada, ou seja, o valor das chaves passadas só aumenta ou só diminui.

Ordenação das palavras:

Função *'BST::search ()'*: Essa função realiza a busca de uma chave específica em uma árvore

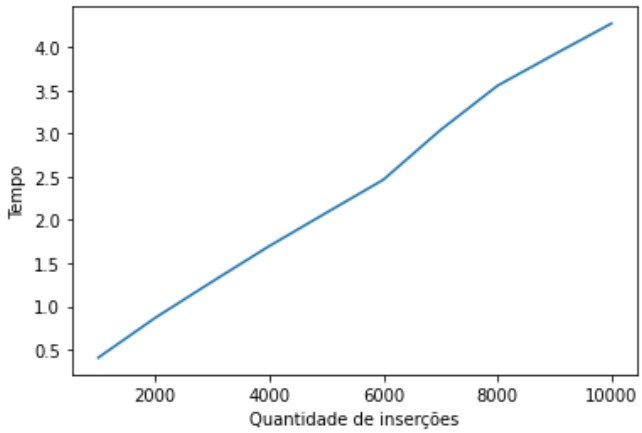
binária. Sua complexidade será a mesma da função de inserção: O(1), no melhor caso, O(log n) no caso médio e O(n), no pior caso.

Função *'BST::substituteNode()'*: Essa função retorna o nó necessário para fazer uma remoção na árvore binária. Desse modo, sua complexidade de tempo será O(log n).

Função *'BST::remove()'*: Essa função realiza a remoção de um node com uma chave especifica da árvore. Desse modo, para achar a chave a ser removida, a complexidade é O(log n) e para achar o node substituto, a complexidade é O(log n). Desse modo, a complexidade de tempo total é O(log n) + O(log n) = O(log n).

Função *'BST::preOrder()'*: Essa função caminha pela lista pela pré-ordem. Logo, ela tem O(n) como complexidade de tempo e O(1) como complexidade de espaço. Essa função é utilizada para gerar os gráficos de acesso de memória, na função ***'Hash::acessaTabela()'***, que, basicamente, executa a função ***'BST::preOrder()'***, para todas posições da tabela Hash.

Dimensão	Tempo
1000	0.405706400
2000	0.867040600
3000	1.286581800
4000	1.697585000
5000	2.082752500
6000	2.465430600
7000	3.039746000
8000	3.552787500
9000	3.916882500
10000	4.273368200



"Gráfico gerado usando o tempo de execução do programa e o número de palavras, por meio das funções da biblioteca 'matplotlib', em Python"

Estratégias de robustez

O programa usa a biblioteca "msgassert.h" para implementar a robustez e corretude das entradas recebidas.

Análise experimental

Desempenho computacional

Para a análise de desempenho computacional foram utilizadas entradas com m=13, com 1000 inserções até 10000 inserções, com 500 pesquisas até 5000 pesquisas e com 200 remoções até 2000 remoções. As mensagens eram formadas por palavras curtas, de até 10 caracteres, geradas aleatoriamente.

Observa-se então, de acordo com o tempo de execução e com as dimensões, o seguinte gráfico:

Desse modo, é possível perceber que, pelo formato da curva gerada, o programa tende para um custo linear de complexidade. Isso pode acontecer pela árvore não estar sendo balanceada, como seria o caso de uma árvore AVL. Nesse caso, seria necessário implementar rotações para balancear a árvore.

Observa-se, então, que o programa tende a uma complexidade de O(n), mesmo utilizando uma BST como estrutura, o que mostra que o desempenho do programa depende diretamente da entrada e que essa estrutura ainda pode ser otimizada.

A análise gerada pelo gprof foi a seguinte:

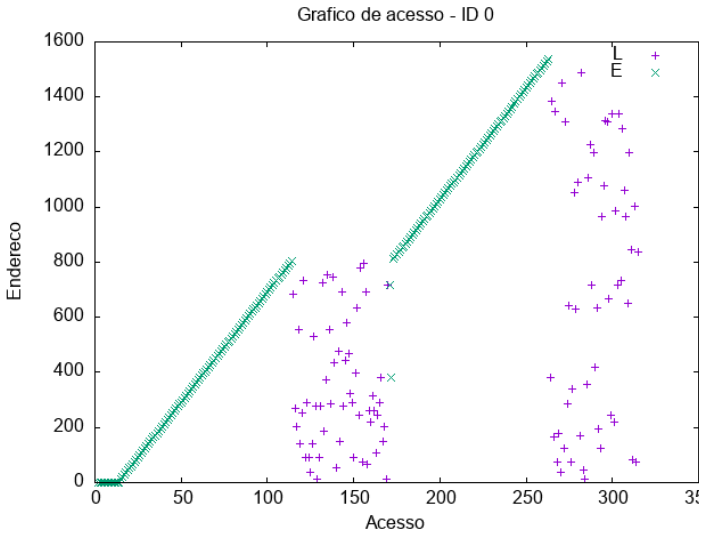
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.37      0.01     0.01    20000     0.00     0.00  BST::insert(Node*&, Mensage)
 33.37      0.02     0.01        1    10.01    10.01  _GLOBAL__sub_I__ZN4HashC2Ei
 33.37      0.03     0.01                             main
  0.00      0.03     0.00   310998     0.00     0.00  Mensage::~Mensage()
  0.00      0.03     0.00   271739     0.00     0.00  Mensage::Mensage(Mensage const&)
  0.00      0.03     0.00    88000     0.00     0.00  int __gnu_cxx::__stoa<long, int, char, int>(long (*)
  0.00      0.03     0.00    88000     0.00     0.00  std::__cxx11::stoi(std::__cxx11::basic_string<char,
  0.00      0.03     0.00    88000     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long (*)(cha
  0.00      0.03     0.00    88000     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long (*)(cha
  0.00      0.03     0.00    88000     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long (*)(cha
  0.00      0.03     0.00    52003     0.00     0.00  bool std::operator==<char, std::char_traits<char>, s
  0.00      0.03     0.00    39250     0.00     0.00  Mensage::Mensage(std::__cxx11::basic_string<char, st
  0.00      0.03     0.00    34001     0.00     0.00  std::operator|(std::_Ios_Openmode, std::_Ios_Openmod
  0.00      0.03     0.00    19262     0.00     0.00  Mensage::operator=(Mensage const&)
  0.00      0.03     0.00    19256     0.00     0.00  Mensage::Mensage()
  0.00      0.03     0.00    19250     0.00     0.00  Node::Node(Mensage)
  0.00      0.03     0.00    10000     0.00     0.00  BST::search[abi:cxx11](Node*&, int)
  0.00      0.03     0.00     4000     0.00     0.00  BST::remove(Node*&, int, std::basic_ofstream<char, s
  0.00      0.03     0.00       13     0.00     0.00  escreveMemLog(long, long, int)
  0.00      0.03     0.00       13     0.00     0.00  BST::preOrder(Node*&)
  0.00      0.03     0.00        3     0.00     0.00  swapNodes(Node*, Node*)
  0.00      0.03     0.00        3     0.00     0.00  BST::substituteNode(Node*&)
  0.00      0.03     0.00        3     0.00     0.00  Node::Node()
  0.00      0.03     0.00        3     0.00     0.00  Node::~Node()
  0.00      0.03     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z9swapNodesP4NodeS0_
  0.00      0.03     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN7MensageC2ENSt7__cxx1112basic_stri
  0.00      0.03     0.00        1     0.00     0.00  _GLOBAL__sub_I_regmem
  0.00      0.03     0.00        1     0.00     0.00  parse_args(int, char**, char*, char*)
  0.00      0.03     0.00        1     0.00     0.00  clkDifMemLog(timespec, timespec, timespec*)
  0.00      0.03     0.00        1     0.00     0.00  iniciaMemLog(char*)
  0.00      0.03     0.00        1     0.00     0.00  desativaMemLog()
  0.00      0.03     0.00        1     0.00     0.00  finalizaMemLog()
  0.00      0.03     0.00        1     0.00     0.00  defineFaseMemLog(int)
```

"Saída gerada pelo "gprof", aplicando uma entrada com 10000 inserções."

Registro de memória e localidade de referência

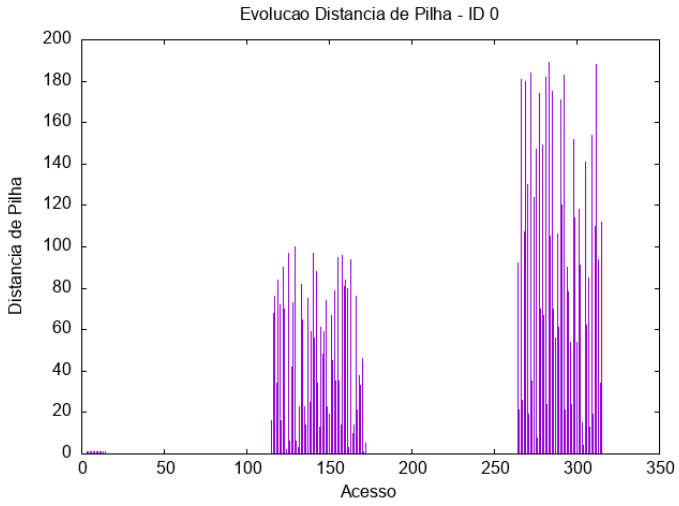
Para a análise do registro de memória foi utilizada a biblioteca *'memlog.h'* e programa *'Analisamem'*.

Assim o gráfico de acesso de memória gerado, a partir de uma entrada com m = 13, 100 inserções, 50 pesquisas e 20 remoções, gerada aleatoriamente



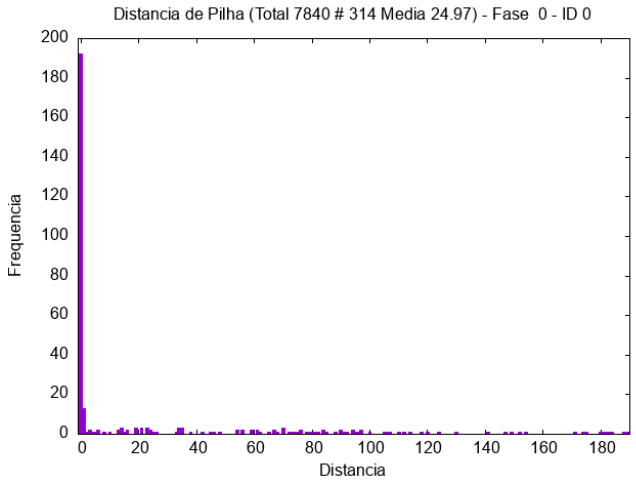
"Gráfico gerado usando o programa gnuplot

Esse gráfico representa a alocação de memória da tabela hash, a qual cada posição aponta para uma BST. É possível ver que, na inserção, as memórias acessadas têm localidade de referência próximas umas das outras, já que as posições alocadas apresentam proximidade.



"Gráfico gerado usando o programa "gnuplot"

Observa-se que a distância entre os endereços de cada posição acessada varia muito, ou seja, a distância de pilha tem valores muito variados. Isso pode ser visto no seguinte gráfico:



"Gráfico gerado usando o programa "gnuplot", a partir da análise de distância de pilha total."

Observa-se, também, que pela proximidade entre os endereços alocados, a frequência da distância 0 é a mais recorrente no gráfico.

Conclusão

Os principais temas abordados foram a abstração do servidor como uma tabela hash, usando o TAD Hash, do email como um TAD Mensage e do usuário como um TAD BST. Na primeira parte, foi implementada uma árvore binária de busca, que tinha as funções de inserção, busca e remoção. A segunda parte, por sua vez, constitui a implementação de uma tabela Hash, sendo que,

para cada posição dessa tabela, havia um ponteiro para uma BST, sendo que cada BST representava um usuário.

Observa-se que há várias formas de otimizar uma árvore binária. Um exemplo é transformar a BST em AVL, de modo que a árvore seja sempre pseudobalanceada diante das operações de inserção e remoção. Desse modo, o custo de O(n) pode ser evitado e transformado em O(log n).

Ao fim deste trabalho ficou evidente a importância da implementação de uma estrutura de dados coerente e que possa abstrair ao máximo uma aplicação realística dessa situação: um servidor de e-mails. Além disso, observa-se que as árvores são estruturas de armazenamento extremamente eficientes e, mesmo assim, podem sofrer muitas otimizações.

Bibliografia

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT.

Instruções para compilação e execução

1. Acesse o diretório chamado **‘TP’**.
2. Adicione o arquivo de entrada nesse diretório.
3. No Makefile, modifique o nome da entrada, em **‘ARQIN’** para o nome de seu arquivo de entrada, o nome da saída em **‘ARQOUT’**.
4. No terminal, execute o Makefile com o comando make. A saída será gerada no diretório **‘TP’**.