Trabalho Prático 3

Gabriela Tavares Barreto

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

gabrielabarreto@dcc.ufmg.br

1 Introdução

A prosposta deste trabalho prático era de implementar um dicionário de duas formas, uma em Árvore AVL, e a outra em Hash. Os elementos do dicionário são strings, verbetes seguidos de seus respectivos significados. Era também tarefa dessa trabalho analisar o desempenho de cada umas das estruturas usadas, assim como avaliar a complexidade assintótica das funções de cada ED (inserção, remoção de dados, ordenação, entre outros).

Para cumprir essa proposta foi empregada, além das estruturas de Hash e de Árvore, uma lista encadeada, usada para armazenar os significados de cada verbete. Mais detalhes sobre a implementação serão dados a seguir.

2 Método

O programa foi desenvolvido nas linguagens C e C++, e compilado pelo G++ da GNU Compiler Collection, usando o WSL. O computador utilizado tem 12 GB de RAM.

3 Implementação

3.1 Estrutura de dados para os significados

- **Classe CelulaSignificado**

Essa classe armazena uma string **significado**, e um ponteiro **prox**, que aponta para outra CélulaSignificado. Ela inicializa prox como NULL, e é uma classe amiga de ListaSignificado.

- **Classe ListaSignificado**

Essa classe é uma estrutura do tipo lista encadedada, e é usada para armazenar os possíveis vários significados de um verbete. Essa estrutura foi escolhida pois permite alocar memória para guardar os significados de acordo com a demanda da entrada. Abaixo, estão descritos os métodos dessa classe:

- **ListaSignificado():** construtor da classe, aloca memória para o atributo primeiro e aponta o atributo último para a cabeça da lista. Inicializa tamanho como zero.
- **void limpar():** é auxiliar ao destrutor e percorre a lista, a partir de **primeiro→prox**, e desaloca a memória usada por cada célula.

- **~ListaSignificado():** destrutor da classe, chama a função limpar e desaloca a memória de primeiro.
- **void inserir_começo(std::string significado):** cria uma nova CelulaSignificado que tem como conteúdo a string passado como parâmetro, e a insere no começo da lista.
- **void inserir(std::string significado):** se a lista estiver vazia chama função inserir_começo para alocação da célula, senão, adiciona o siginifcado ao final da lista.
- **void escreve(std::ofstream *file):** escreve no arquivo de saída file o significado de cada célula da lista.
- **int get_tamanho():** retorna o tamanho da lista.

3.2 Estrutura de dados para o verbete

◦ **struct Verbete**

Essa struct contém uma string **verbete** e uma ListaSignificado **lista**.

3.3 Estrutura auxiliar

◦ **struct Entrada**

Armazena duas strings, **verbete**,**siginificado** eum inteiro **altura**. Essa estrutura é usada para armazenar dados lidos do arquivo de entrada, e também como parâmetro nas funções dos dicionários implementados.

3.4 Dicionário em Árvore AVL

◦ **Classe Node**

Essa classe representa os nós da árvore AVL. Ela armazena uma struct do tipo verbete, denominada **v**, e três ponteiros **esq, dir e pai**, sendo que eles apontam, respectivamente, para o nó a esquerda, o nó a direita e o nó pai do nó. Os ponteiros são inicialidazos como NULL. Sua única função é **void escreve(std::ofstream *file)**, que escreve o verbete do nó no arquivo file, e chama função escreve para a lista de significados do verbete.

◦ **DicionárioÁrvore**

Essa classe é uma estrutura do tipo árvore AVL. Ela tem como atributos um node denomidado **raiz**, que é inicializado como NULL. Sua funções estão descritas a seguir:

- **void limpar(Node *node):** percorre a árvore desalocando a memória de cada nó a partir do node passado como parâmetro.
- **~DicionarioArvore():** destrutor da classe, chama a função limpar a partir da raiz.
- **int max(int a1, int a2):** retorna o maior dos inteiros passados como parâmetro.
- **int altura(Node *&node):** retorna a altura do nó.
- **int get_fb(Node *&node:** retorna o fator de balanceamento do nó, ou seja, a diferença de altura entre nó direito e esquerdo.
- **void rotacao_direita(Node *&node):** executa o rotação para a direita.
- **void rotacao_esquerda(Node *&node):** executa uma rotação para a esquerda.

- **balanceia(Node *&node):** identifica se há algum desbalancemento, e se sim, qual caso é. Primeiro ela verifica o fator de balanceamento do nó passado como parâmetro. Se identificar um desbalancemento para a esquerda, verifica qual o FB do nó esquerdo. Se ele também for negativo, executa uma rotação para a direita. Caso contrário executa uma rotação para esquerda e outra para direita. Já se um desbalancemento para a direita for identificado, verifica qual o FB do nó direito. Se ele também for positivo, executa uma rotação para a esquerda. Caso contrário executa uma rotação para direita e depois outra para a querda.
- **void insere_recursivo(Node *&node, Entrada p):** percorre a árvore, comparando o verbete de entrada com nós. Se um nó com o mesmo verbete não for encontrado, a função aloca memória para um novo nó. Caso contrário, ela verificase se o siginficado da entrada é vazio, caso não seja, adiciona um novo siginficado a lista do verbete do nó.
- **void insere(Entrada p)**: chama insere_recursivo para a raiz,passando p como parâmetro.
- **void in_ordem(std::ofstream *file, Node *&node):** faz um caminhamento em ordem na árvore, a partir do nó passado, ou seja, percorre a árvore em ordem lexicográfica. Para cada nó chama a função de escrita passando file como parâmetro, escrevevndo assim o verbete e os seu(s) significado(s) no arquivo de saída.
- **void escreve(std::ofstream *file):** chama a função in_ordem para a raiz, passando file como parâmetro.
- **void antecessor(Node *&q, Node* r):** obtém o nó mais a direita do nó esquerdo de q, ou seja, obtém o antecessor de q. Tendo identificado o antecessor, percorre a árvore de baixo para cima, atualizando as alturas dos e rebalenceando a árvore caso necessário. Atualiza os atributos do antecessor para que ele possa ocupar o lugar do nó que, além disso, a função faz uma reatribuição do atributo pai de todos nós envolvidos. Por fim o antecesso ocupar o lugar de q, e q tem sua memória desalocada.
- **std::string remove_nodes(Node *&node):** percorre a árvore a partir do nó de parâmetro por meio de chamadas recursivas, afim de identicar e deletar nós da árvore que tenham verbetes com significados. Ao encontrar um nó a ser deletado, verifica se ele tem ou não filhos. Se ele tiver os dois filhos, chama a função antecessor. Após deletar um nó, chama a função altura e balanceia, para rebalancear a árvore caso necessário. Ao deletar um nó, retorna seu verbete. Caso contrário, retorna "0", para representar uma string vazia.
- **void atualiza():** chama a função remove_nodes a partir da raiz, até que ela retorne "0", indicando que todos nós com significado da árvore foram apagados.

3.5 Dicionário em Hash

◦ **struct Info**

Armazena uma string **verbete** e um inteiro **pos**. É uma estrutura auxiliar a classe DicionárioHash, e é usada para a ordenação indireta do hash. Sua utilização ficará mais clara na descrição da ordenação de DicionárioHash, logo a seguir.

◦ **Classe DicionarioHash**

Essa classe é uma estrtura da dados do tipo Hash de endereçamento aberto, e armazena dois ponteiros e um inteiro **m**. O ponteiro **tabela** aponta para um endereço de Verbete, e **vazio**, que aponta para um endereço de inteiro. Para evitar o mal do agrupamento, mas ao mesmo tempo não consumir muita memória, optei em criar tabelas com tamanho três vezes maior que da entrada.

As funções da classe são as seguintes:

- **DicionarioHash(int n):** recebe o número máximo de elementos da tabela, e aloca a memória de tabela e vazio com $3 * n$ posições. Atribui a todas posições de vazio o valor 1 e atribui o valor $3 * n$ à m.
- $\sim$ **DicionarioHash(int n):** desaloca a memória de tabela e vazio.
- **int f_hash(std::string p):** soma o valor ascii de cada caracter do verbete, e obtém o resto da divisão dessa soma pelo tamanho da m da tabela e o retorna.
- **void insere(Entrada p):** aplica a função f_hash sobre p, obtendo sua posição na tabela. Se a posição já estiver ocupada, percorre sequencialmente as posições seguintes, até encontrar alguma com o status vazio. Atribui a posição o verbete e significado passados como parâmetro e torna o status da posição de vazio como falso.
- **void quick_sort(Info *A, int n):** recebe um vetor de Infos **A** e seu tamanho **n**, e chama a função ordena.
- **void ordena(int esq, int dir, Info *A):** recebe uma posição à esquerda **esq** e a posição à direita **dir** do vetor A, e chama o método de partição. A partir da atualização dos índices de esquerda e direita, faz duas chamadas recursivas, dividindo o problema de ordenar o vetor em dois problemas menores.
- **void particao(int esq, int dir, int *i, int *j, info *A):** resumidamente, essa função seleciona o elemento do meio do vetor como um **pivô x**, e executa comparações dos verbetes dos elementos do vetor com o verbete de x, fazendo trocas nas posições dos elementos. Ao final do algoritmo de partição, o vetor A[esq, ..., dir] está particionado de forma que:
  - Os elementos em A[esq], até A[j] tem verbetes lexicograficamente menores ou iguais ao de x;
  - Os elementos em A[i] até A[dir] tem em verbetes lexicograficamente maiores ou iguais ao de x.
- **void escreve(std::ofstream *file):** cria uma vetor auxiliar com elementos do tipo Info, com tamanho de 1/3 do tamanho da tabela do Hash. Depois, percorre a tabela hash, ao encontrar um elemento não vazio, salva seu verbete e posição na tabela auxiliar. O processo termina quando as n posições do vetor auxiliar forem preenchidas, ou quando toda tabela hash tiver sido percorrida. Após isso, o vetor_aux é ordenado de 0 até count(número de posições ocupadas no vetor), usando o quick_sort. Por fim, o vetor é percorrido, e o atributo pos de cada um de seus itens é usado para obter a posição a ser acessada na tabela_hash. A cada acesso na tabela_hash o verbete e seus significados são escritos em file.
- **void remove_verbetes():** dá status vazio para verbetes com tamanho de lista maior ou igual a um, ou seja, verbetes com signifcado.

3.6 Programa principal

O programa principal irá utilizar, a depender dos parâmetros passados pelo terminal, alguma das implementações citadas na seção anterior para construir o dicionário. Ao fim da execução do programa, um arquivo de saída é gerado. Para isso, foram criadas as funções a seguir:

- **void uso():** imprime o menu de usuário, explicando como passar corretamente os parâmetros junto ao executável.

- **void parse_args(int argc, char ** argv):** essa função usa como método auxiliar a função getopt da biblioteca getopt.h. Ela faz a leitura dos parâmetros passados pelo usuário, e guarda qual estrutura de dados a usar (hash ou arv), o nome do arquivo de entrada e o de saída. Ao identificar algum parâmetro inválido, ou a ausência de parâmetros, chama a função uso para imprimir o menu e encerra o programa.
- **int conta_linhas(char *nome_entrada):** abre o arquivo de entrada e o percorre, contando o número de linhas que ele tem, e retorna esse valor.
- **void separa_linha(std::string linha, std::string vet[3]):** recebe linha inteira lida do arquivo, e a separa em substrings, alocando essa substrings no vetor de strings.
- **int main(int argc, char ** argv):** o programa principal roda a função parse_args. Ele chama a função conta_linhas para obter o número de máximo de verbetes que podem ser inseridos no dicionário, e imprime uma mensagem no terminal informando essa valor. Depois, ele faz a abertura do arquivo de entrada para sua leitura, usando um objeto do tipo ifstream, e cria o arquivo de saída para escrita usando um objeto do tipo ofstream. Um objeto DcionárioHash ou DicionárioÁrvore é decarado. Após isso, um loop de for é rodado, de forma que todo arquivo é percorrido e todos verbetes e respectivos significados são armazenados na estrutura esolhida. Por fim os arquivos de entrada e saída são fechados, e o programa é encerrado.

4 Análise de Complexidade

4.1 Classe ListaSignificado

Agora, será apresentada a análise de complexidade de tempo e espaço para os métodos apresentados na seção 3.1 valor n aqui significa o tamanho da lista encadeada.

- A função construtora e a função get_tamanho da classe tem complexidade $O(1)$, pois realizam sempre o mesmo número de operações ao serem chamadas.
- **void limpar():** essa função tem complexidade $O(n)$, pois precisa percorrer toda a lista para deletá-la.
- **~ListaSiginificado:** tem complexidade $O(n)$, pois chama a função limpar e executa além disso somente a ação de deletar a cabeça da lista.
- **void inserir_começo(std::string significado):** tem complexidade assintótica $O(1)$, sempre insere a lista no começo, e executa para isso uma quantidade constante de passos.
- **void inserir(std::string):** complexidade $O(1)$, pois executa um número constante de passos, ou chama inserir_começo ou insere o significado na última posição da lista.
- **void escreve(std::ofstream * file):** complexidade $\theta(n)$ pois precisa percorrer toda a lista para imprimí-la.

4.2 Classe Node

Sua única função é a escreve(), que escreve o verbete no arquivo de saída e chama a função escreve para a lista encadeada de singificados. Assim, sua complexidade é a mesma de lista.escreve, sendo esta $\theta(n)$.

4.3 DicionárioArvore

Aqui n representa a quantidade de nós na árvore.

- **void limpar(Node *node):** complexidade $O(n)$, pois percorre toda árvore para eliminar seus elementos a partir do ponto dado.
- **~DicionarioArvore():** complexidade $\theta(n)$, pois chama a função limpar para percorre toda árvore e deletar todos nós.
- **int max(int a1, int a2), int altura(Node *&node) e int get_fb(Node *&node):** todas tem complexidade $O(1)$, pois sempre executa o mesmo número de passos dado uma entrada.
- **void rotacao_direita e void rotacao_esquerda):** complexidade $O(1)$, pois sempre executam a mesma quantidade de passos.
- **balanceia(Node *&node):** $O(1)$, pois no pior caso executa duas rotações (2x$O(1)$=$O(1)$).
- **void insere_recursivo(Node *&node, Entrada p):** complexidade $O(\log(n))$, pois para achar o local do nó e atualizar as alturas os custos são ambos $O(\log(n))$.
- **void insere(Entrada p)**: chama insere_recursivo, logo sua complexidade é também $O(\log(n))$
- **void escreve(std::ofstream *file):** como ela percorre todos os nós para escrevê-los no aqruivo de saída, sua complexidade é $\theta(n)$
- **void antecessor(Node *&q, Node* r):** sua complexidade de tempo é $O(\log(n))$, pois para encontrar o antecessor o custo é $O(\log(n))$ e para atualizar as alturas o custo é $O(\log(n))$ também, e $O(\log(n)) + O(\log(n)) = O(\log(n))$.
- **std::string remove_nodes(Node *&node):** custo $O(\log(n))$, pois a chama a função antecessor, que é $O(\log(n))$, e além dela realiza operações também de custo $O(\log(n))$, como encontrar o nó a ser deletado e atualizar as alturas da árvore, ou de custo $O(1)$(balanceamentos).
- **void atualiza():** $O(\log(n))$, pois chama a função remove_node a partir da raiz.

4.4 DicionarioHash

Aqui N representa o número de verbetes no dicionário.

- **DicionarioHash(int n):** temm complexidade de tempo $\theta(n)$ e complexidade de espaço $\theta(n)$ também, pois inicializa duas tabelas com tamanho 3*n passado como parâmetro.
- $\sim$ **DicionarioHash(int n):** complexidade $\theta(n)$, pois precisa desalocar completamente a tabela de tamanho n, ou seja, esse processo tem tempo depende do tamanho de n.
- **int f_hash(std::string p):** complexidade $\theta(n)$, sendo aqui n o comprimento da string, pois precisa percorre completamente a string passada como parâmetro.
- **void insere(Entrada p):** no melhor caso consegue inserir o verbete na posição gerada pela f_hash e tem complexidade $O(1)$. No pior caso, tem que percorrer sequencialmente os n-1 endereços após o seu até encontrar uma posição vazia, tendo ai=í complexidade $O(n)$.
- **void quick_sort(Info *A, int n):** o método de ordenação QuickSort tem a seguinte complexidade de tempo:
  - **Pior caso:** $O(n^2)$. Ocorre quando o pivô é escolhido repetidamente da pior maneira possível: como um dos extremos do vetor, estando ele já ordenado.

  - **Caso médio:** $O(n \log(n))$, pois de acordo com Sedgewick e Flajolet, o número médio de comparações $C(n)$ do algortimo é: $1,386n \log(n)$–$0,846n$.
  - **Melhor caso:** $O(n \log(n))$. No melhor caso, o vetor sempre é dividido em dois subvetores de tamanho igual.

- **void escreve(std::ofstream *file):** tem complexidade de tempo $O(n)$, pois precisa percorrer a tabela de tamanho 3*n para obter seus elementos. Sua complexidade de espaço é também $O(n)$, pois precisa de memória extra para alocar um vetor de tamanho n, onde guarda as structs Info.
- **void remove_verbetes():** complexidade de tempo $O(n)$, pois precisa percorrer a tabela completamente e verificar todos seus elementos para poder setá-los como vazio ou não.

5 Estratégias de Robustez

O programa feito depende de entradas do usuário para saber que comandos seguir. Logo, foi necessário pensar em que erros ele poderia cometer, e em como lidar com eles. Por exemplo, o usuário pode informar parâmetros inválidos, ou esquecer de passar algum parâmetro.

Como estratégia de programação defensiva, foram estabelecidos parâmetros de verificação, que retornam mensagens de erro quando percebido algum erro, e que encerram a execução do programa. Assim, é garantida a execução do programa apenas quando parâmetros válidos são informados.

Essa verificação é feita usando objetos do tipo **erroAssert**, definido no header MsgAssert.h, criado pelo professor Wagner Meira. Esse objeto verifica uma condição, e caso ela seja falsa, ele imprime uma mensagem explicando o erro na tela e encerra o programa. Ele foi usado para verificar as seguintes condições:

- se uma opção de estrutura de dados foi fornecida,
- se um arquivo de entrada foi informado,
- se um nome para o arquivo de saída foi fornecido,
- se a tabela do hash estiver completamente cheia, e houver tentativa de inserir algum elemento

6 Análise experimental

6.1 Localidade de referência

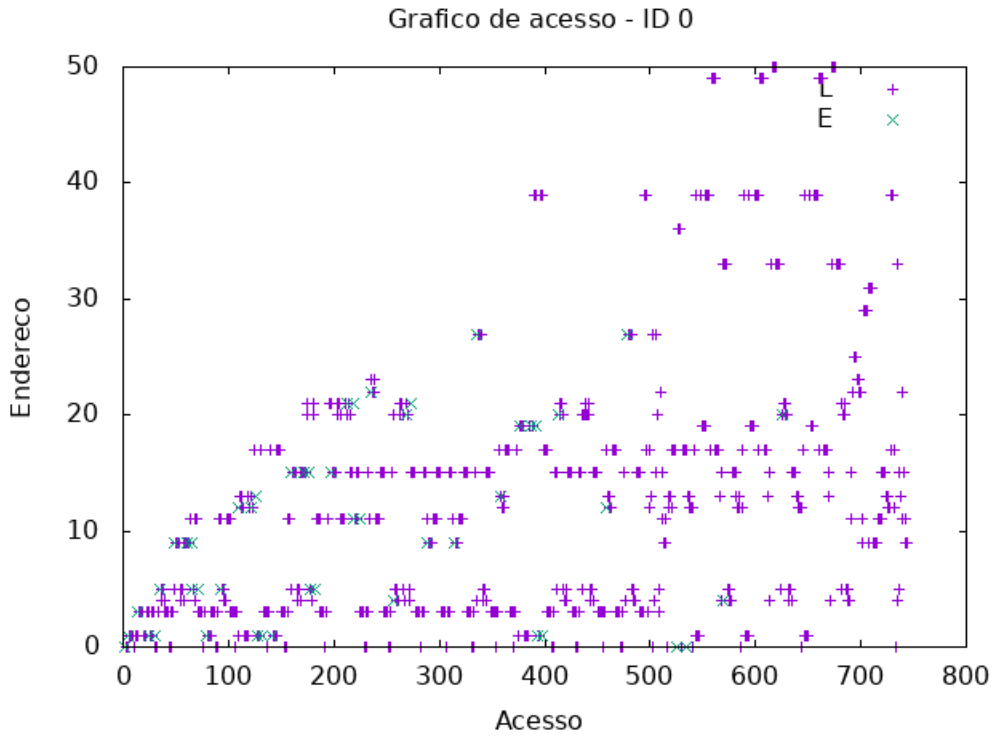
Para a análise experimental e de localidade de referência foram usadas as funções de registro de acesso de leituras e escritas na memória da biblioteca Memlog.

As duas implementações foram testadas para o mesmo arquivo de entrada, que continho 22 verbetes diferentes. Não foi possível registar todos acessos dos métodos porque alguma leitura eram feitas sobre variáveis declaradas na stack, o que tornava o range de acesso de memória muito grande, impossibilitando o uso do AnalisaMem.

Por fim, a partir dos relatórios gerados das execuções, usando o Analisamem e o GnuPlot, foram produzidos os gráficos abaixo.

1.DicionarioArvore

O gráfico acima registra os acessos feitos na memória da árvore AVL em cinco momentos diferentes.

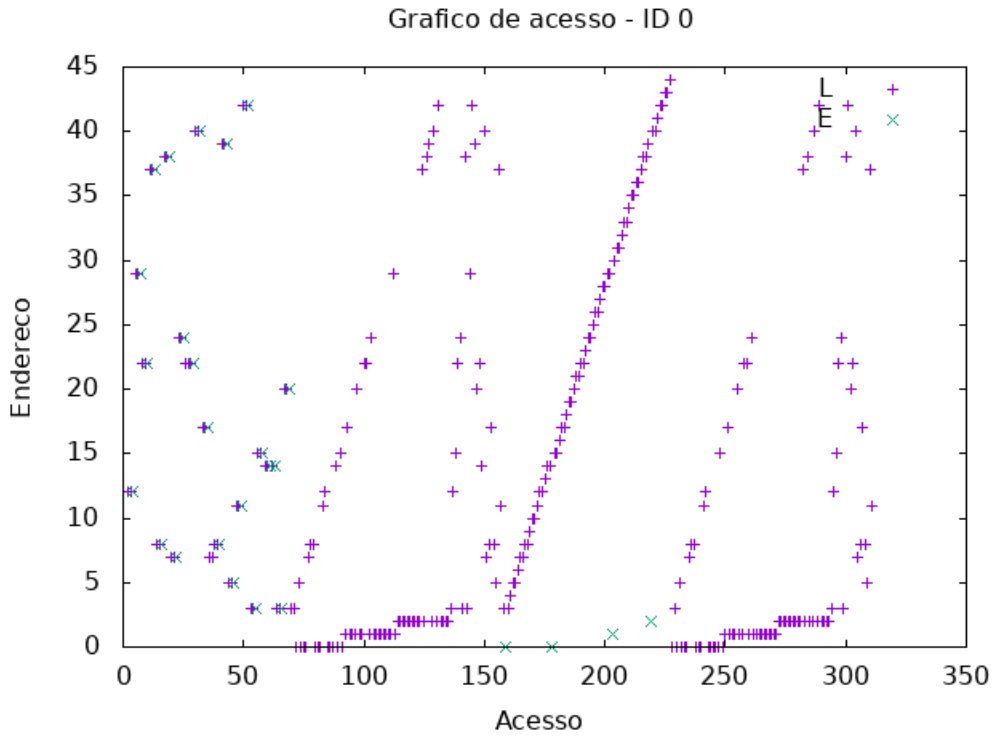


- **Acesso 1 a 494:** ocorrem na inserção dos verbetes na árvore. Vemos no ponto (0,0) a primeira escrita na árvore, indicando a inserção da raiz. Como esperado, é nesse treho do gráfico que vemos mais escritas. Além de indicarem a inserção de elementos na árvore, as escritas nesse trecho do gráfico representam também os rebalanceamentos feitos para manter o FB da árvore na faixa correta. Vale apontar que quanto mais na ponta da árvore um nó é inserido, são feitas várias leituras dos nós acima, para garantir seu rebalancemanto e atualizar suas alturas.
- **Acesso de 495 a 515:** mostra as leituras feitas para escrita dos verbetes do arquivo de saída Podemos observer uma linha vertical de vários acessos de leitura logo acima do valor 500 do eixo x, indicando o caminhamento em ordem feito na árvore. Devido aos rebalanceamentos decorrentes da inserção de verbetes, os nós não necessarimanente estão próximos de seus pais e filhos na árvore. Isso explica o porque termos acessos quebrados e não totalmente sequenciais. Como o arquivo só possuia 22 verbetes, essa fase termina rapidamente.
- **Acesso 516 a 727:** ocorridos no método atualiza, que remove os nós sem significado da árvore. Na árvore haviam apenas dois nós com significados, porém nesse trecho há mais de dois registros de escrita. Isso indica que foi necessário exexcutar um rebalanceamento da árvore ao remover um desses nós. Vemos que para uma árvore de 22 nós, muitas leituras são necessárias para apagar dois nós e garantir seu balanceamento.
- **Acesso 728 a 744:** representa novamente acessos do método escreve, agora para a árvore menor.

Por fim, vale informar que o tempo de execução do programa foi de 0.0176 segundos.

1.DicionarioHash

O gráfico acima registra os acessos feitos na memória da árvore AVL em cinco momentos



diferentes.

- **Acesso 0 a 69:** mostra os acessos ocorridos na inserção de elementos na tabela do Hash. Vemos que eles ocorrem de forma não sequencial, o que é esperado, visto que a função f_hash gera posições não sequenciais.
- **Acesso 70 a 157:** acessos no método escreve. Temos uma linha de acessos que começa no endereço 3, acesso 70, e que vai até o endereço 45, acesso 110, indicando as leituras feitas na tabela vazio em busca de posições não vazias. Há outra linha de acessos sequenciais, que ocorrem em paralelo a linha vertical, e que representam as structs info salvas no vetor auxiliar. Como já trabalhamos bastante o QuickSort no último TP, optei em não mostrar seus acessos a memória, visto que o enfoque deste trabalho é outro. Depois que o quickSort ordena o vetor auxiliar, temos uma última linha de acessos de aparencia não sequencial, que começam a ocorrer a partir do acesso 140, que são os acessos na tabela para escrita dos verbetes no arquivo de saída.
- **Acesso 158 a 227:** nesse trecho do gráfico temos os acessos feitos na hora de remover os verbetes com significado. Os acessos sequenciais do endereço 3 a 45 são ad tabela estar sendo percorrida em busca desses elementos. Os registro de escrita são da atribuição do valor 1 a vazio, nas posições dos elementos deletados.
- **Acesso 228 a 311:** escrita no arquivo de saída do dicionário somente com os verbetes sem significado. Os padrões de acesso são análogos ao do acesso 70 a 157, porém como agora o dicionário é menor, menos acessos são feitos.

O tempo de execução foi de 0.008 segundos.

6.2 Comparação entre as estruturas

As estruturas hash e árvore tem funcionamentos bem diferentes entre si. Essa diferença ficou bem explícita com os gráficos usados na seção anterior: ambas execuções receberam a mesma entrada e tem o mesmo resultado, porém usam algoritmos bem diferentes. Cada um deles tem suas vantagens e desvantagens. A principal vantagem do Hash frente a árvore AVL, é o melhor caso de inserção, pesquisa, e remoção terem custo constante. Por outro lado, para que o custo seja constante, a tabela do hash deve ser um número razoavelmente de vezes maior que o tamanho da entrada, ou seja, para essa estrutura funcionar de forma ágil, há uma desvantagem de consumo maior de memória.

Por outro lado, a estrutura árvore tem inserção, remoção e pesquisa com complexidade $O(\log(n))$, mais caro que do hash. Em contraponto a isso, a árvore tem a vantagem de poder alocar memória somente quando um nó for ser inserido, ou seja, seu consumo de memória é bem mais baixo que do hash. Outra vantagem da árvore frente ao hash é a facilidade de obter os itens em ordem, pois quando são inseridos o custo de ordenação é amortizado. Enquanto isso é necessária memória extra para ordenar os verbetes sem mudar sua posição na tabela.

7 Conclusão

Concluo esse trabalho tendo tirado um grande aprendizado sobre árvores AVL e Hash. Poder implementar, testar e debugar cada uma dessas estruturas foi um processo que exigiu bastante de mim, principalmente ao criar o método de remoção dos elementos da árvore.

Gostei muito de poder aprofundar meus conhecimentos e treinar a construção de funções recursivas, que são muito usadas na classe da árvore.

Foi interessante comparar as estruturas para as mesmas entradas, nesse processo pude ver na prática as desvantagens e vantagens de cada implementação, reforçando o aprendizado teórico proveniente da sala de aula.

8 Bibliografia

- Material didático usado em sala(slides).
- http://wiki.icmc.usp.br

9 Instruções de Compilação e execução

Para compilar o programa, deve-se seguir os seguintes passos:

- Acessar o diretório TP;
- Utilizando um terminal, digitar make, de forma a gerar o arquivo executável **run.out**;
- Na pasta bin, chamar o executável ./run.out, junto aos parâmetros de execução.
- No caso de dúvida, deve-se rodar o executável sozinho, de forma a imprimir o menu de uso do programa na tela.