UNIVERSIDADE FEDERAL DE MINAS GERAIS

INSTITUTO DE CIÊNCIAS EXATAS - ICEX

ALGORITMOS E ESTRUTURAS DE DADOS II

Ingrid Rosselis Sant’Ana da Cunha

TP2

Belo Horizonte

2016

★ Introdução:

Este trabalho tem como objetivo implementar uma Tabela Hash; um método de pesquisa que dispersa elementos distribuindo-os pela estrutura - um vetor, por exemplo - com a ajuda de chaves associadas. A Hash utiliza a chave de busca do elemento e, com uma função de transformação, define onde ele será inserido na tabela. Assim, quando o valor de um item é procurado na estrutura, a mesma aplica a função de transformação novamente e vai direto para posição do índice obtido, tornando a pesquisa mais rápida e simples.

Um grande problema da Hash é o fato de que são grandes as chances de ocorrerem colisões entre elementos conforme aumenta-se o tamanho da entrada, ou seja, a função de transformação retorna um índice que já possui um elemento associado. Para solucionar este imprevisto, existem duas possibilidades: encadeamento e endereçamento aberto. A primeira consiste em adicionar uma outra estrutura de dados da escolha do programador à cada espaço da tabela, onde serão colocadas todas as chaves destinadas àquele índice; já a segunda, depois de definir o índice do espaço onde será inserido o elemento, procura o espaço disponível mais próximo do índice ideal e insere o elemento lá.

Para este trabalho foi definido o uso de árvores binárias para tratar as colisões por encadeamento. Sendo assim, dois tipos de árvores binárias, com e sem balanceamento, foram utilizadas no trabalho e comparações foram feitas para definir a diferença de desempenho entre as duas.

★ Implementação:

A implementação do trabalho foi dividida em headers e arquivos .c: para cada .h, um .c.

Geral: Os arquivos .h e .c com nome “geral” possuem, respectivamente, o protótipo da função *leArquivo* e sua implementação. Essa função declara tudo que é necessário para leitura do arquivo e inserção dos elementos na tabela e chama as funções apropriadas para criar a Hash, criar o Elemento, inserir o elemento na tabela e imprimir e apagar a Hash. Para ler o arquivo é utilizado um while com condição de parada igual ao caractere de fim do arquivo. Dentro desse laço é usado um *ungetc* para não perder o caractere utilizado pelo *fgetc* da condição, seguido por um *fscanf* para ler o número de matrícula, um *fgets* para ler o nome do aluno, o *criaAluno* e a *insereNaHash*. A função recebe como parâmetros os nomes dos arquivos de entrada e saída, o tamanho da tabela desejado e o tipo de árvore que será utilizada na execução. Ela retorna 0 se concluir sem falhas e 1 se tiver algum problema.

Aluno: Os arquivos .h e .c com nome “aluno” possuem a TAD Aluno e os protótipos, no .h, e no .c as implementações relativas à funções *criaAluno*, *apagaAluno* e *imprimeAluno*:

criaAluno: Essa função declara, aloca memória e inicializa um ponteiro da estrutura Aluno com o nome e a matrícula que foram recebidos como parâmetro. Retorna o ponteiro criado.

apagaAluno: Desaloca o ponteiro do nome relacionado a uma estrutura Aluno recebida como parâmetro e, em seguida, desaloca o ponteiro da própria estrutura. Não possui retorno.

imprimeAluno: Limpa o buffer do arquivo de saída e grava os dados do Aluno com um *fprintf*, no formato “(matricula) nome”. Recebe o ponteiro do Aluno e o ponteiro do arquivo como parâmetros e não possui retorno.

ABP: (árvore binária de pesquisa sem balanceamento) Todos os .h e .c com nome “abp” possuem a TAD Arvore, os protótipos e implementações relativas à funções *criaArvore*, *pesquisa*, *insereElemento*, *removeDaArvore*, *achaMenor*, *imprimeArvore* e *apagaArvore*. Como as implementações feitas no trabalho são baseadas nos slides de aula, aqui será feito uma breve explicação sobre elas:

criaArvore: Declara e inicializa um ponteiro da estrutura Arvore; os ponteiros para as sub-árvores são setados como NULL e o registro é preenchido com o Elemento r recebido como parâmetro. Retorna o ponteiro criado.

pesquisa: Executa a busca dentro da árvore de forma recursiva, caminhando para a esquerda ou direita caso a chave recebida seja menor ou maior que a do nodo atual. Recebe o ponteiro da Arvore e a chave desejada e retorna o ponteiro do Elemento a qual a chave pertence ou NULL, caso não encontre o elemento.

insereElemento: Insere um elemento na árvore, procurando o seu ponto de inserção como se fosse uma busca, porém, ao encontrar uma folha, declara um novo nodo, chama a função *criaArvore* para ele e liga o nodo à árvore. Recebe o ponteiro da Arvore e o ponteiro do Elemento e não possui retorno.

removeDaArvore: Apesar de não possuir uma utilidade na implementação direta do trabalho, a função serve de teste posterior. Ela remove um elemento da árvore, buscando-o pela chave na estrutura e, ao encontrar, executando um dos três casos de teste: se é um nodo folha - desaloca nodo e retorna NULL ao pai; se possui um filho na esquerda ou direita - cria um nodo auxiliar para o filho, desaloca o nodo desejado e retorna ao pai do nodo removido o ponteiro auxiliar; ou se possui os dois filhos, onde substitui o nodo a ser removido pelo menor elemento em sua sub-árvore direita e depois exclui o substituto. Ela recebe o ponteiro da Arvore e a chave do Elemento a ser retirado e retorna o ponteiro da Arvore após a alteração.

achaMenor: Função auxiliar da *removeDaArvore*, encontra o menor elemento da árvore e o retorna, dado o ponteiro da Arvore.

imprimeArvore: Grava a árvore no arquivo de saída, utilizando caminhamento central, ou seja, faz a chamada recursiva para a parte esquerda da árvore até chegar em uma folha, chama a função *imprimeAluno* para o registro (reg) do nodo e depois faz a chamada recursiva para a parte direita da árvore. Dessa forma, os elementos são impressos em ordem crescente. Recebe o ponteiro da Arvore e o ponteiro do arquivo e não possui retorno.

apagaArvore: Desaloca os dados da árvore recebida por parâmetro usando caminhamento pós-fixado à direita, ou seja, faz a chamada recursiva para a esquerda e para a direita primeiro, para depois desalocar o ponteiro com um *apagaAluno*, evitando que um nodo interno seja desalocado antes das folhas. Recebe o ponteiro da Arvore e não possui retorno.

SBB: (árvore binária de pesquisa com balanceamento) Os arquivos .h e .c com nome "sbb" possuem a TAD ArvoreSBB, os protótipos e implementações relativas à funções *criaArvoreSBB*, *pesquisaSBB*, *ee*, *dd*, *ed*, *de*, *iInsere*, *iInsereAqui*, *insereElementoSBB*, *inicializa*, *imprimeArvoreSBB* e *apagaArvoreSBB*. Vale lembrar que as implementações das duas árvores são bastante parecidas e que as implementações feitas no trabalho são baseadas nos slides de aula:

criaArvoreSBB: Declara e inicializa um ponteiro da estrutura ArvoreSBB; os ponteiros para as sub-árvores são setados como NULL, suas posições (esqtipo e dirtipo do pai) são setadas como verticais e o registro do nodo é preenchido com o elemento r recebido como parâmetro. Retorna o ponteiro criado.

pesquisaSBB: Essa função é quase igual à da ABP pelo fato de que ser ou não balanceada não interfere na busca. A distinção entre as duas é feita pelo ponteiro passado como parâmetro, visto que são duas estruturas diferentes. Recebe o ponteiro da ArvoreSBB e a chave procurada e retorna ponteiro do Elemento ou NULL, se a pesquisa for mal sucedida.

ee, dd, ed e de: As 4 funções são praticamente iguais, possuindo a mesma entrada e a mesma saída. A diferença entre elas é que a *ee* faz uma rotação à direita, a *dd* faz uma rotação à esquerda, a *ed* faz uma rotação à esquerda com o filho esquerdo, seguido por uma rotação à direita, e a *de* faz uma rotação à direita com o filho direito e depois uma rotação à esquerda. Todas têm o propósito de manter o balanceamento da SBB funcionando. Elas recebem o ponteiro de ponteiro da ArvoreSBB, porque nessas rotações os ponteiro são alterados, e não possuem retorno.

iInsere: Função auxiliar da *insereElementoSBB*. Ela basicamente procura o local de inserção do elemento e comanda quando as rotações são necessárias ou quando o processo de inserção chegou ao fim e a árvore está de acordo com suas propriedades. Recebe o ponteiro do Elemento, o ponteiro duplo para a ArvoreSBB, a inclinação do nodo como ponteiro e a flag de fim também como ponteiro. Não possui retorno.

iInsereAqui: Função auxiliar da *iInsere*. Ela cria o novo nodo e o preenche, como faz a *criaArvoreSBB*, porém o insere na horizontal e seta o fim como falso, avisando à *iInsere* de que o processo ainda não acabou e que o balanceamento da árvore deve ser conferido para ter certeza de

que não há violação das propriedades. Ela possui os mesmos parâmetros que a *iInsere* e também não possui retorno.

insereElementoSBB: Essa função cria as flags de inclinação e fim e chama a função *iInsere* para começar a inserir o elemento na árvore. Recebe um ponteiro duplo para a ArvoreSBB e o ponteiro do Elemento que se deseja inserir. Não possui retorno.

inicializa: Função que inicializa a raiz da árvore com NULL. Recebe o ponteiro duplo para a ArvoreSBB e não possui retorno.

imprimeArvoreSBB: Igual à *imprimeArvore*, exceto pelo de estrutura passado. Também imprime a árvore por caminhamento central e recebe o ponteiro da ArvoreSBB e o ponteiro do arquivo. Não possui retorno.

apagaArvoreSBB: Desaloca o dados da ArvoreSBB por caminhamento pós-fixado à direita, evitando um erro de segmentação, assim como a *apagaArvore*. Recebe o ponteiro da ArvoreSBB e não possui retorno.

Hash: Os arquivos .h e .c com nome "hash" possuem a TAD Hash, os protótipos e implementações relativas à funções *funcaoHash*, *criaHash*, *apagaHash*, *insereNaHash*, *obtemDaHash*, *imprime* e *obtemNumElem*:

funcaoHash: Recebe o ponteiro da Hash e uma chave, aplicando a função de transformação nesta. Ela basicamente retorna o módulo (resto da divisão) entre a chave recebida e o tamanho da Hash, valor que será utilizado como índice para inserir ou buscar dentro da tabela.

criaHash: A função declara e aloca espaço para a tabela Hash, além de zerar o número de elementos, setar o tamanho da tabela e o tipo de árvore, recebidos como parâmetro, na estrutura e alocar e inicializar as árvores a serem utilizadas no trabalho. Retorna, por fim, o ponteiro criado.

apagaHash: Desaloca a Hash recebida como parâmetro junto com todas as árvores utilizadas. Basicamente, verifica o tipo de árvore utilizada e vai chamando a função apaga respectiva para o tamanho da tabela, por fim, desaloca o ponteiro da Hash. Não possui retorno.

insereNaHash: Insere um Elemento na Hash, ambos recebidos como parâmetro. Aplica a *funcaoHash* e verifica se a árvore é nula: se sim, cria a árvore; se não, insere nela. Por fim, incrementa o número de elementos. Pode não inserir o elemento, se ele já existir, e não possui retorno.

obtemDaHash: Busca uma determinada chave na tabela Hash. A função aplica a *funcaoHash*, e faz uma busca interna, com *pesquisa* ou *pesquisaSBB*, na árvore do índice. Recebe o ponteiro da Hash e a chave e retorna o ponteiro do Elemento encontrado, ou NULL.

imprime: Função que imprime a tabela Hash. Cria o arquivo de saída, verifica qual tipo de árvore foi utilizada na execução e, para cada uma delas, chama a função de imprimir da própria árvore. Por fim, fecha o arquivo preenchido. Recebe o ponteiro da tabela e o nome do arquivo de saída e não possui retorno.

obtemNumElem: A função retorna o número de elementos inseridos em uma tabela Hash, recebida como parâmetro, até o momento da chamada, sendo o número de elementos referido como n.

Uma análise da implementação: Considerando a busca como fator determinante da complexidade do algoritmo, existem 3 possibilidades do número de comparações dentro das árvores ABP e SBB:

No melhor caso, as duas árvores fazem apenas uma comparação, pois o elemento da chave buscada é o primeiro (ou o único) da árvore (complexidade: O(1)).

No caso médio, as duas árvores começam a divergir em termos do número de comparações: a ABP faz, em média, (n - 1)/2 comparações (complexidade: O(n)), enquanto a SBB faz aproximadamente log(n) comparações (complexidade: O(log n)).

No pior caso, a diferença entre o número de comparações aumenta ainda mais, apesar das complexidades se manterem em O(n) e O(log n): o pior caso da ABP acontece quando os elementos a serem inseridos estão em ordem crescente ou decrescente, nesse caso temos (n - 1) comparações. O pior caso da SBB, graças ao seu balanceamento, faz apenas log(n) comparações.

Essas comparações de eficiência entre as duas árvores vão ganhando mais força conforme o n é aumentado. Por exemplo, se o número de elementos inseridos em cada árvore for igual a 100 e a ordem de inserção for a mesma, a ABP faria 99 comparações no pior caso e aproximadamente 50 comparações no caso médio. Já a SBB faria mais ou menos 7 comparações para os casos pior e médio.

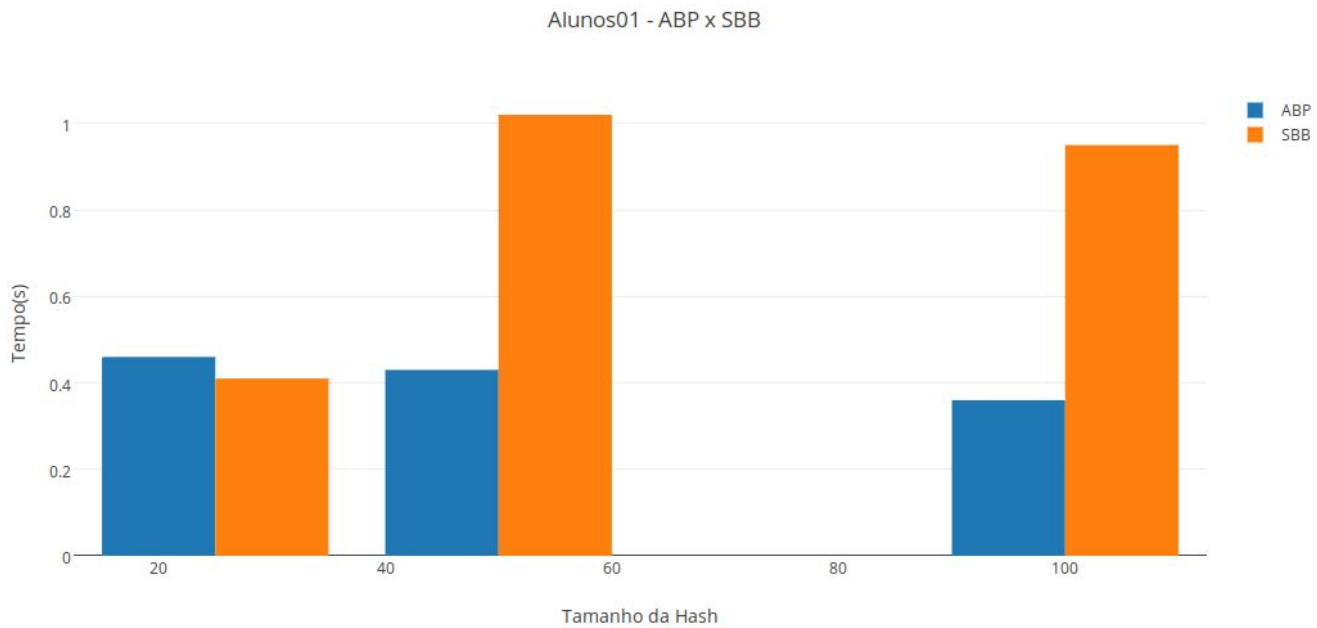
Apesar de que no geral a intenção da tabela Hash é sempre pesquisar a O(1), neste trabalho sua complexidade total é maior porque a função de transformação usada é muito simples e os tamanhos de tabela testados são pequenos. Quanto mais colisões acontecem, maior é o tempo de execução, sem contar que, para alguns casos, a função de transformação definiu muitos elementos para um só índice, aumentando o tempo de busca da árvore. Outro ponto negativo é que para árvores binárias a ordem de inserção importa, ou seja, alguns casos são prejudicados por causa dessa influência.

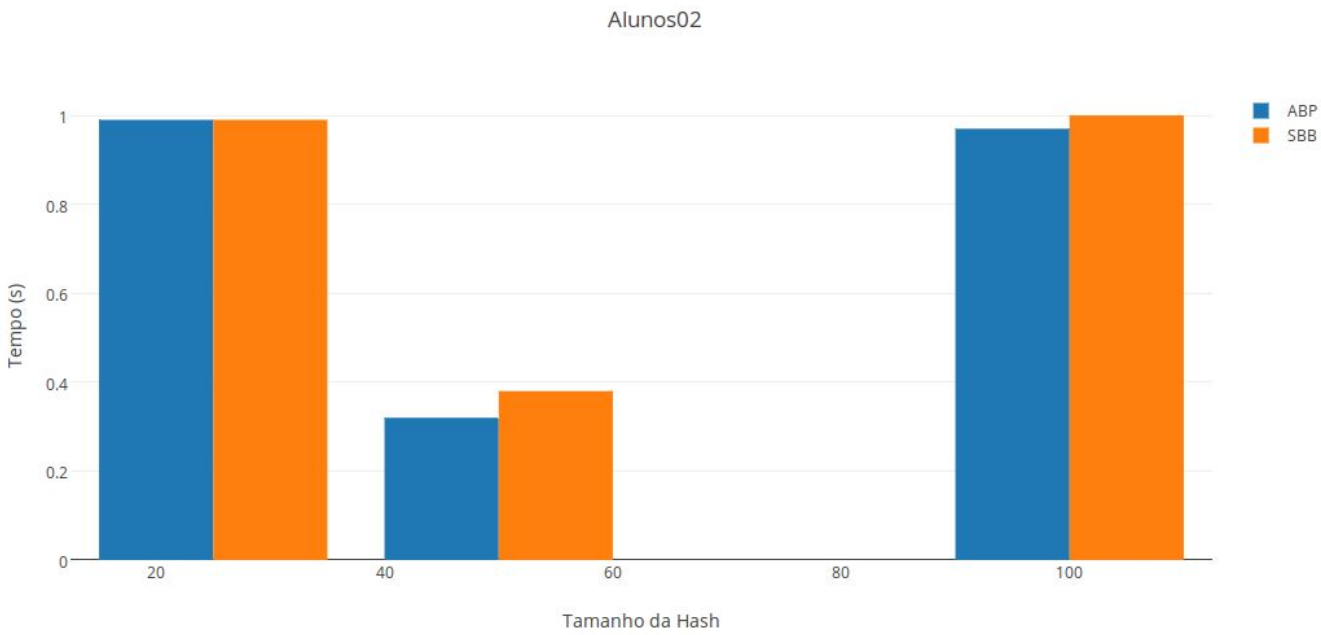
Em questão de espaço, a implementação tem complexidade O(n) para as duas árvores e é bastante estável, pois não há variáveis para influenciar o quanto de memória é alocada se não for o tipo de estrutura e o número de elementos. Comparando ABP e SBB, e considerando que a entrada é a mesma para ambas, fica claro que a segunda ocupa mais espaço por possuir mais dados em sua estrutura.

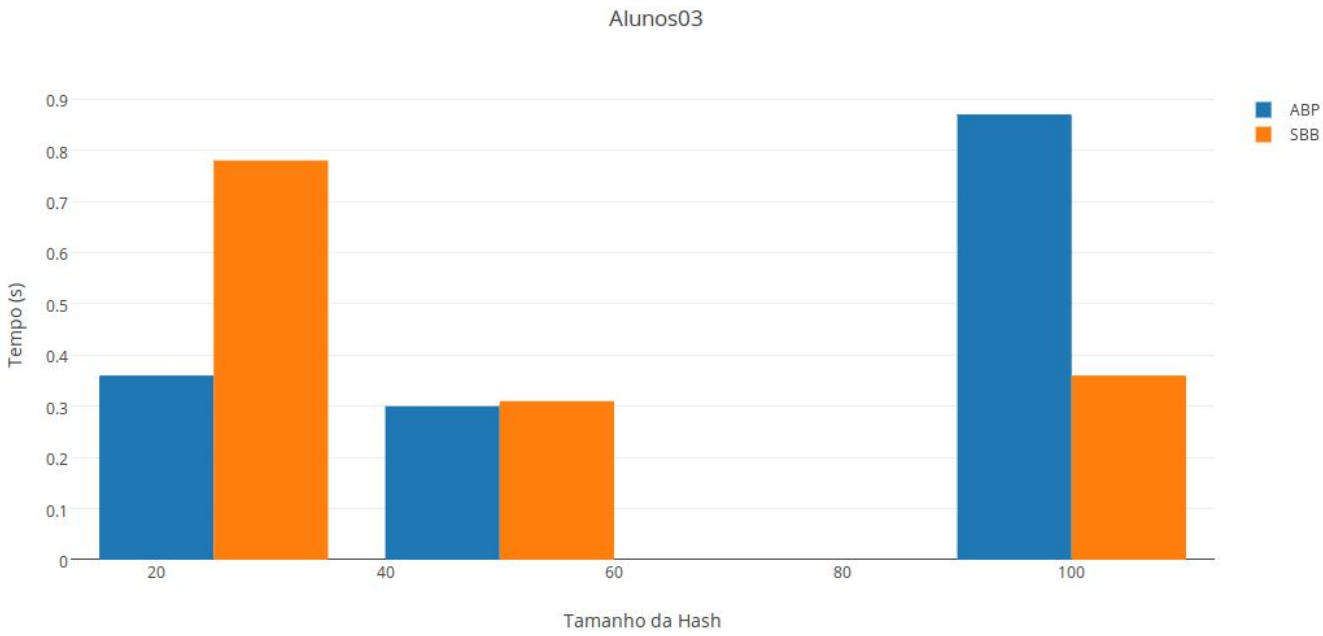
Portanto, tanto para tempo quanto para espaço, o algoritmo possui complexidade polinomial pois as funções nunca necessitam percorrer mais de uma vez as estruturas utilizadas. Seu custo, no entanto, poderia ser menor se a função de transformação da tabela Hash fosse melhor elaborada e se o tamanho da tabela fosse maior, diminuindo as colisões e trazendo a complexidade do algoritmo para O(1) total, sem a necessidade de percorrer estruturas auxiliares.

★ Resultados:

Os testes com os 3 arquivos disponibilizados (alunos01, alunos02 e alunos03) foram enviados junto com o código e possuem nome no estilo: “saida-(número do arquivo de entrada)-(tamanho da tabela)-(tipo de árvore).txt“. O programa também apresenta na tela o tempo de execução e o número de elementos inseridos. Logo abaixo seguem gráficos em barras com comparações entre as implementações das árvores em relação aos tempos de execução de cada um dos 3 arquivos para as duas árvores em 3 tamanhos de tabela (25, 50 e 100):







Como pode ser observado em vários casos, o tempo de execução da SBB é maior que o da ABP, diferente do que foi analisado mais acima. Isso acontece porque a árvore SBB só é realmente mais vantajosa se o número de elementos é alto. Os casos de teste usados tem entre 63 e 90 elementos bem distribuídos, logo a SBB aplica rotações que são muitas vezes desnecessárias e custosas, visto que o número de elementos na árvore de cada índice não chega a ser grande. Por exemplo se 3 elementos são inseridos em ordem crescente em determinado índice da tabela Hash, o custo para buscar um deles é mais baixo que o custo de aplicar uma rotação à direita para encaixar as propriedades da SBB. Deve ser explicitado também que a precisão do tempo obtido nesses testes pode variar muito de execução para execução, mas servem para fazer a comparação entre as duas estruturas e ajudam a dar ênfase no ponto de vista aqui defendido.

É seguro afirmar, com base nos dados obtidos, que um dos pontos positivos de se usar a tabela Hash é verdadeiro: quanto maior for o tamanho da tabela, menos colisões o programa terá e mais rápido será a execução. Exceto alguns casos esporádicos, foi visto uma tendência do tempo de execução de diminuir conforme o tamanho da tabela aumenta, porque eram menos árvores utilizadas e menos elementos em cada uma delas. Com um tamanho igual a 100, houve vários índices com apenas 1 elemento associado, nesses casos o objetivo da tabela Hash, fazer pesquisa a O(1), foi obtido.

★ Conclusão:

Este trabalho teve como objetivo implementar uma tabela Hash por encadeamento de árvores. As maiores dificuldades foram a modularização, onde eu descobri que as TAD e as funções de cada estrutura deveriam ficar em arquivos distintos, algo que eu nunca havia trabalhado antes -, lidar com o tamanho do programa, várias vezes o tanto de funções chegou a ser confuso, implementar a inserção na árvore SBB, demorou um certo tempo até os ponteiros de ponteiros fazerem sentido e a inserção é realmente complicada, e, por último, foi obter os tempos de execução porque foi algo confuso e trabalhoso, além de requerer um cuidado maior para gravar os arquivos.

Fazer este trabalho me mostrou novas formas de pesquisa que podem ser bastante eficientes e deixou claro que toda a implementação tem seus pontos fortes e fracos, situações onde é mais vantajosa que outras.

★ Referências:

https://gist.github.com/tonious/1377667