Documentação Trabalho Prático 2 de AED's 2

Prof: William Robson Schwartz

Aluno: Marcus Vinícius de Oliveira 2014144804

1. Introdução

Tabela Hash ou tabela de dispersão, é uma estrutura de dados que associa chaves de pesquisa a valores. É composto de um vetor onde cada uma das posições armazena zero, uma ou mais chaves. Entretanto, diferente do vetor comum, a Tabela Hash utiliza uma transformação aritmética sobre a chave de seus elementos para descobrir em qual índice a chave deverá ser armazenada.

Sua eficiência e simplicidade fazem com que ela seja comumente utilizada como método de pesquisa. Seu funcionamento é constituído de duas principais etapas:

i. Computar o valor da transformação aritmética(função Hashing)
ii. Tratar as colisões da tabela, uma vez que diferentes chaves podem estar na mesma posição.

Nosso trabalho consiste, principalmente, de implementar uma Tabela Hash e tratar suas colisões de duas formas diferentes: Utilizando uma árvore binaria e utilizando uma SBB.

2. Implementação

2.1 Leitura dos dados e o módulo "reader.c"/"reader.h"

Os arquivos de entrada são compostos pelo seguinte formato: Número de matrícula (linha 1) e nome do aluno (linha2). O padrão se repete até o final do arquivo, conforme a imagem:

```
009300
Alessandro Gomes Sifuentes
360565
Andre de Lima Guss
360573
Carlos Alberto de Oliveira
360580
Diego Freitas Siqueira Souza
326080
Ewerton Ozorio da Luz
360583
Filipe da Costa Matos Sousa Faria
360589
Guilherme Caires de Miranda
345016
...
```

Consideramos as matriculas como inteiros e os nomes como strings. A leitura dos dados é feita pelo módulo "reader.c"/"reader.h".

*Aluno*** **leituraArquivo**(char* arg) - Essa função recebe uma string contendo o endereço do arquivo a ser aberto e retorna um vetor de ponteiros para Aluno (conferir TAD aluno). Dentro dela é aberto o arquivo de entrada, criado dinamicamente um vetor de ponteiros para Aluno e enquanto é lido o arquivo é também criado elementos Aluno que são 'inseridos' nesse vetor.

Note que a criação dos elementos Aluno são dependentes da função criarAluno (conferir módulo aluno). Além disso, a função leituraArquivo é dependente da função numeroLinhas desse mesmo módulo.

Note que é necessário o uso de dois fgetc(file) para retirar o caractere '\n' entre o fscanf referente a linha 1 e a linha 2.

Considerando atribuições, a função executa alguns comandos O(1) e loop de 0 a n, onde n é o número de alunos no arquivos (número de linhas/2). Desta forma a função é delimitada por O(n).

int **numeroLinhas**(char* arg) - A função recebe uma string contendo o endereço do arquivo a ser aberto e retorna um inteiro com o número de linhas que o arquivo possui. A função é necessária pois a logica utilizada para a leitura do arquivo necessita dessa informação.

A função é bem simples e consiste em percorrer o arquivo e conferir se o caractere atual é igual a '\n'. Caso seja um contador é incrementado.

A função faz algumas atribuições O(1) e possui um laço que lê todo arquivo. Dessa forma, considerando atribuições, ela se comportar como O(n), onde n é o numero de caracteres do arquivo. [O(1) + O(1) + O(n) = O(n)].

void **apagaListaAlunos**(char* arg, *Aluno*** listaAlunos) - Recebe uma string contendo o endereço do arquivo de entrada e um vetor de ponteiros para alunos. Sua função é desalocar este vetor. É importante salientar que que as estruturas Aluno são desalocadas dentro do TAD das árvores durante o desalocamento da estrutura Hash.

2.2 TAD aluno e o módulo "aluno.c"/"aluno.h"

Conforme especificação, este é o TAD Aluno:

```
typedef int Chave
typedef struct Aluno{
        char* nome;
        Chave matricula;
}Aluno;
```

*Aluno** **criaAluno**(char* n, *Chave* mat) - Essa função recebe uma string com o nome completo do aluno e um inteiro contendo sua matrícula. Ela aloca uma estrutura aluno e seus elementos e atribui seus valores conforme parâmetro. Durante a alocação da string nome é utilizado a função strlen, da biblioteca string.h.

Considerando atribuições, sua complexidade é de O(1), pois possui apenas seis atribuições diretas e nenhum laço.

void **apagaAluno**(*Aluno** a) - Recebe Aluno e o desaloca. Sua complexidade é O(1), pois apenas desaloca dois elementos.

void **imprimeAluno**(*Aluno** a) - Recebe Aluno e o imprime no stdout no seguinte formato:

```
(9300) Alessandro Gomes Sifuentes
```

Sua complexidade é O(1).

2.3 TAD Arvore e o módulo "binarytree.c"/"binarytree.h"

Conforme especificação, este é o TAD Arvore:

```
typedef Aluno Elemento;
typedef struct Arvore {
        struct Arvore* esq;
        struct Arvore* dir;
Elemento* reg;
}Arvore;
```

*Arvore** **criaArvore**(*Elemento** r) - recebe como entrada um Elemento r, aloca/cria uma Arvore e coloca o Elemento r como raiz dessa Arvore. São feitas quatro atribuições, caracterizando a função como O(1).

*Elemento** **pesquisa**(*Arvore* *t, Chave x) - Recebe uma Arvore t e uma Chave x. A função pesquisa na Arvore t a Chave x. Caso a chave x exista na Arvore, é retornado o endereço da Arvore t* contendo a Chave x.

Considerando comparações, a complexidade é dada pela altura da árvore. No pior caso da pesquisa é igual O(n), isso acontece quando a arvore recebe um vetor ordenado. O melhor caso a busca é de O(log n).

void **insereElemento**(*Arvore** t, *Elemento* *n) - Essa função recebe uma Arvore t e um elemento n. Sua função é a inserção do Elemento n na Arvore t.

A insereElementos precisa localizar o local para fazer a inserção. Isso faz com que sua complexidade seja dependente da altura h da Arvore. A complexidade de inserção é O(h).

*Arvore** **removeDaArvore**(*Arvore** t, *Chave* c) -Recebe uma Arvore t e uma Chave c que será removida de t, caso esta exista. Quatro casos são possíveis:

i. A chave não existe em t.
ii. O nodo contendo c não tem filhos.
iii. O nodo contendo c tem um filho.
iv. O nodo contendo c tem dois filhos.

Para o caso i não há nada o que fazer.

No caso ii o nodo pai do nodo removido passa a apontar para NULL.

No caso iii o pai do nó removido aponta para o filho do nó removido.

No caso iv o valor do nodo a ser removido é substituído pelo maior valor da subárvore esq. A função é chamada recursivamente e os ajustes necessários serão feitos (estes ajustes serão um dos casos anteriores).

A remoção precisa primeiro localizar o nó que contém a chave a ser removida. Dessa forma a complexidade de remoção é O(h), onde h é a altura da árvore. Lembrando que no pior caso, a altura de uma árvore binária de busca pode ser O(n).

*Arvore** **achaMenor**(*Arvore** t) - Essa função recebe uma Arvore t e retorna o endereço da subArvore que contem a menor Chave de t. Seu funcionamento é trivial e consiste em retornar o último nodo à esquerda de t. Sua complexidade é log(h) para uma Arvore.

void **imprimeArvore**(*Arvore* *t) - Recebe Arvore t e a imprime no stdout. Considerando comparação, sua complexidade é O(n), onde n é o número de elementos contidos em t.

void **apagaArvore**(*Arvore* *t) - Recebe Arvore t e a desaloca. Percorre toda a Arvore t e a desaloca. Sua complexidade é O(n), onde n é o número de elementos contidos em t.

2.4 TAD ArvoreSBB e o módulo "sbb.c"/"sbb.h"

Conforme especificação, este é o TAD ArvoreSBB:

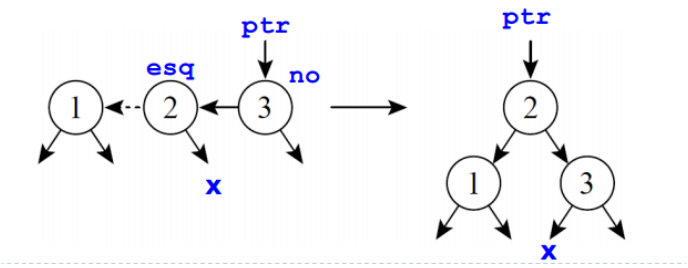
```
typedef Aluno Elemento;
typedef struct ArvoreSBB {
        Elemento* reg;
        struct ArvoreSBB* esq;
        struct ArvoreSBB* dir;
        int esqtipo;
        int dirtipo;
}ArvoreSBB;
```

*ArvoreSBB** **criaArvoreSBB**(*Elemento** r) - Recebe como entrada um Elemento r, aloca/cria uma ArvoreSBB e coloca o Elemento r como raiz dessa Arvore. É retornado o endereço da ArvoreSBB. São feitas seis atribuições, caracterizando a função como O(1).

*Elemento** **pesquisaSBB**(*ArvoreSBB* *t, *Chave* x) - Recebe uma ArvoreSBB t e uma Chave x. A função pesquisa na Arvore t a Chave x. Caso a Chave x exista na Arvore, é retornado o endereço da Arvore t* contendo a Chave x.

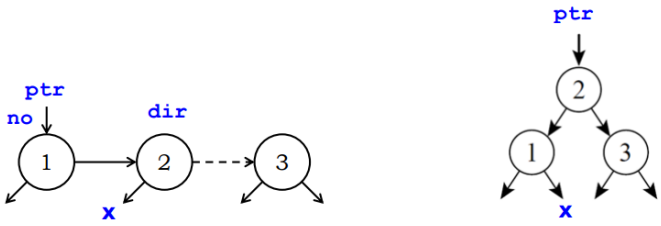
Considerando comparações, a complexidade é dada pela altura da árvore. O melhor caso a busca é de O(1), o caso médio e o pior caso são O(log(n)). Lembrando que a altura da arvore sbb considera a altura horizontal k e a vertical h, onde h<=k<=2h.

void **ee**(*ArvoreSBB*** ptr) - Recebe uma ArvoreSBB ptr e faz mudanças conforme a seguinte imagem:



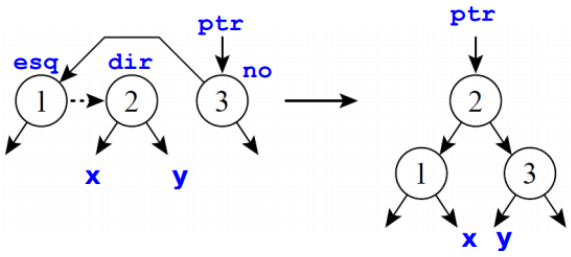
Essa mudança faz prevalecer as propriedades da ArvoreSBB e é chamada quando existem dois apontadores horizontais esquerdos consecutivos. É retornado o endereço da mesma ArvoreSBB, porém com as mudanças feitas.

void **dd**(*ArvoreSBB*** ptr) - Recebe uma ArvoreSBB ptr e faz mudanças conforme a seguinte imagem:



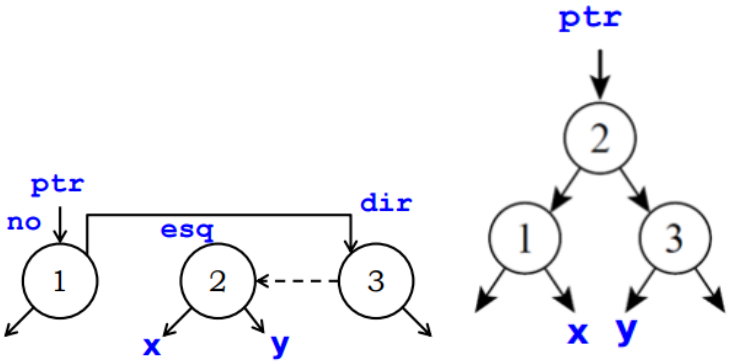
Essa mudança faz prevalecer as propriedades da ArvoreSBB e é chamada quando existem dois apontadores horizontais direitos consecutivos. É retornado o endereço da mesma ArvoreSBB, porém com as mudanças feitas.

void **ed**(*ArvoreSBB*** ptr) - Recebe uma ArvoreSBB ptr e faz mudanças conforme a seguinte imagem:



Essa mudança faz prevalecer as propriedades da ArvoreSBB e é chamada quando existem dois apontadores horizontais um esquerdo e outro direito consecutivos. É retornado o endereço da mesma ArvoreSBB, porém com as mudanças feitas.

void **de**(*ArvoreSBB*** ptr) - Recebe uma ArvoreSBB ptr e faz mudanças conforme a seguinte imagem:



Essa mudança faz prevalecer as propriedades da ArvoreSBB e é chamada quando existem dois apontadores horizontais um direito e outro esquerdo consecutivos. É retornado o endereço da mesma ArvoreSBB, porém com as mudanças feitas.

void **iInsere**(*Elemento** r, *ArvoreSBB*** ptr, int *incli, int *fim) - Recebe ArvoreSBB prt, ponteiro para incli e outro para fim. Está função encontra o local onde deve ser inserido o elemento r em prt. Ao encontrar e ele insere e rebalanceia a ArvoreSBB.

void **iInsereAqui**(Elemento* r, *ArvoreSBB*** ptr, int *incli, int *fim) - Recebe Elemento r , ArvoreSBB prt e dois inteiros , um que representa a inclinação e outro que diz ser o fim da ArvoreSBB . O Elemento r é a inserido em prt. Complexidade O(1).

void **insereElementoSBB**(*ArvoreSBB*** t, *Elemento* *n) - Recebe ArvoreSBB t e um Elemento n que deve ser inserido em verticalmente. Está função necessita da iInsere.Complexidade O(1).

void **inicializa**(*ArvoreSBB*** raiz) - Inicializa ArvoreSBB com a raiz da árvore apontando para NULL. Complexidade O(1).

void **imprimeArvoreSBB**(*ArvoreSBB** t) - Recebe ArvoreSBB t. Percorre toda a ArvoreSBB e a imprimi In-Order utilizando a função imprimiAluno do TAD aluno.

void **apagaArvoreSBB**(*ArvoreSBB* *t) - Recebe ArvoreSBB t. A função percorre toda a arvore e a desaloca por completo. O custo é O(n), onde n é o número de elementos contidos em t.

2.5 TAD Hash e o módulo "hash.c"/"hash.h"

Conforme especificação, este é o TAD Hash:

```
typedef struct Hash{
        Arvore** hash;
        ArvoreSBB** hashSBB;
        int tam;
        int nElem;
}Hash;
```

int **funcaoHash**(*Elemento* *ele, *Hash* *h) - Essa função recebe Elemento ele e uma tabela Hash h. Ela faz a transformação aritmética da chave da seguinte maneira:

Hashing = Chave MOD tamanhoHash

Sua complexidade é O(1).

*Hash** **criaHash**(int t, int boolSBB) - Recebe inteiro t relativo a tamanho que o Hash deve ser criado e um inteiro boolSBB relativo a qual árvore deve ser alocada, Arvore ou ArvoreSBB. A função cria/aloca a tabela Hash e a árvore que deve ser criada.

Considerenado atriuição, a complexidade é O(n), onde n é o tamanho do Hash.

void **apagaHash**(*Hash** h, int boolSBB) - Recebe um Hash h e um inteiro boolSBB relativo a qual árvore foi usada. Percorre toda a Hash h e apaga/desaloca toda estrutura.

Sua complexidade é O(n), onde n é o número de elementos contidos na Hash.

void **insereNaHash**(*Hash** h, Elemento* x, int boolSBB) - Recebe uma tabela Hash h, um Elemento x e um inteiro boolSSB (Que indica qual árvore está sendo utilizada). Esta função determina em qual posição o Elemento x deve ser indexado na tabela h e o insere de acordo com a árvore que está sendo utilizada.

Elemento * **obtemDaHash**(*Hash** h, *Chave* c) - RecebeHash h e Chave c a ser buscada. Retorna endereço do elemento contendo a Chave, caso ela exista

void **imprime**(*Hash** h, int boolSBB) - Recebe uma tabela Hash h e um inteiro boolSSB (Que indica qual árvore está sendo utilizada). Percorre toda a Hash h e seus elementos árvores e os imprime no stdout.

A complexidade da função é O(1).

int **obtemNumElem**(Hash* h) - Recebe um Hash h e retorna o número de elementos inseridos na Hash. A propria estrutura já guarda essa informação. Sua complexidade é O(1).

2.6 Função main

A argumentação passada para a função main é apresentada da seguinte forma:

```
./exec entrada.txt saida.txt 50 0
    ou
./exec entrada.txt saida.txt 50 1
```

Onde entrada.txt representa o arquivo de input; saida.txt representa o arquivo de output; 50 é o tamanho do Hash a ser criado; 0 indica que a árvore a ser utilizada é uma binária sem balanceamento; 1 indica que a árvore a ser utilizada é uma SBB.

Acontece a declaração das variáveis. O argumento argv[4] é convertido em int e salvo na variável boolSBB que é passada como parâmetro para criação da Hash. De maneira similar, argv[3] é convertido em int e salvo na variável tamHash que é passada como parâmetro para criação da Hash.

É criado um vetor de Aluno listaAlunos e a tabela Hash h.

É inserido cada item do vetor listaAlunos na tabela h através de um laço.

Após inserção é fechado o stdout e aberto o arquivo (argv[2]) em seu lugar.

Então é impressa a tabela h e liberada toda a mémoria alocada.

3. Resultado

A implementação do trabalho transcorreu sem maiores problemas entretanto os dados de tempo obtidos estão inconclusivos.

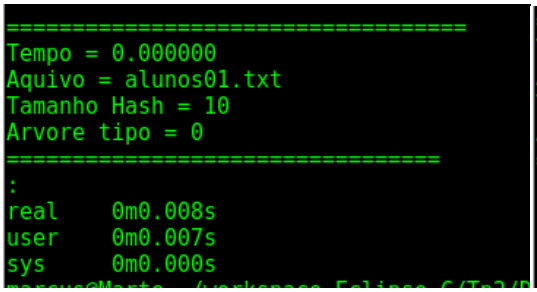
Foram utilizados duas maneiras de se medir o tempo.

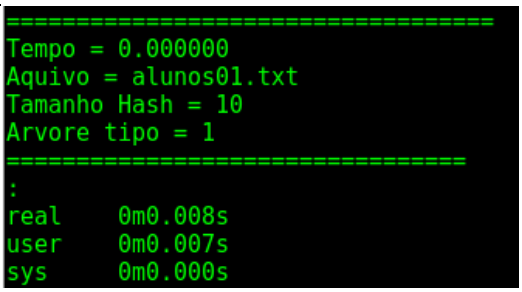
A primeira foi importando o a biblioteca time.h e utilizando o código:

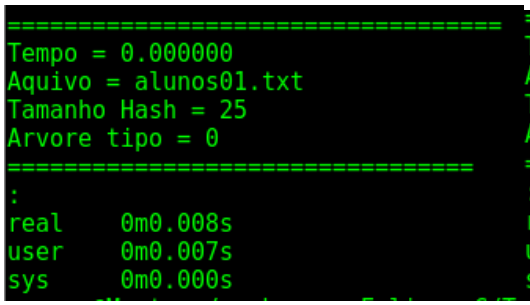
```
long double tempo;

clock_t t_ini, t_fim;

t_ini = clock();


//Codigo a ser examinado


t_fim = clock();

tempo = (t_fim - t_ini)/CLOCKS_PER_SEC;

Printf ("%Lf", tempo);
```

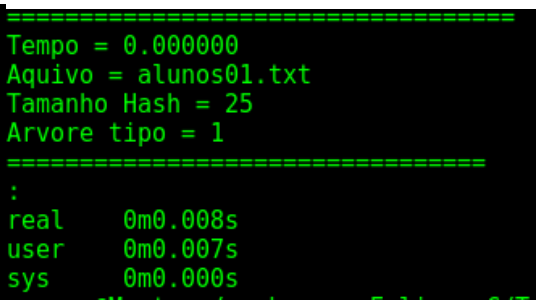
A segunda foi utilizando o comando time do bash antes da execução do programa.

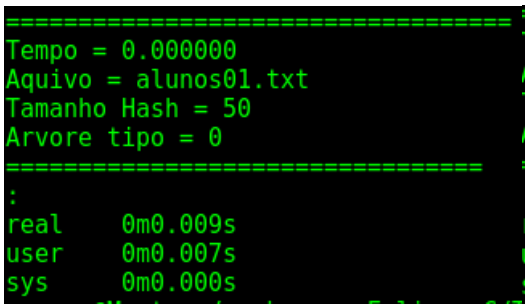
Os resultados foram os seguintes para o arquivo aluno01.txt:

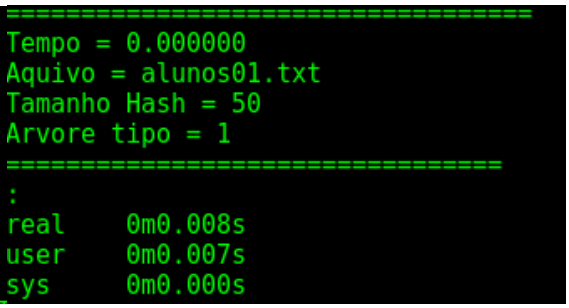












```
==================================
Tempo = 0.000000
Aquivo = alunos01.txt
Tamanho Hash = 100
Arvore tipo = 0
==================================
:
real    0m0.008s
user    0m0.007s
sys     0m0.000s
```

```
==================================
Tempo = 0.000000
Aquivo = alunos01.txt
Tamanho Hash = 100
Arvore tipo = 1
==================================
:
real    0m0.008s
user    0m0.007s
sys     0m0.000s
```

Para o arquivo aluno02.txt:

```
==================================
Tempo = 0.000000
Aquivo = alunos02.txt
Tamanho Hash = 25
Arvore tipo = 0
==================================
:
real    0m0.008s
user    0m0.007s
sys     0m0.000s
```

```
==================================
Tempo = 0.000000
Aquivo = alunos02.txt
Tamanho Hash = 25
Arvore tipo = 1
==================================
:
real    0m0.008s
user    0m0.007s
sys     0m0.000s
```

```
==================================
Tempo = 0.000000
Aquivo = alunos02.txt
Tamanho Hash = 50
Arvore tipo = 0
==================================
:
real    0m0.008s
user    0m0.007s
sys     0m0.000s
```

```
==================================
Tempo = 0.000000
Aquivo = alunos02.txt
Tamanho Hash = 50
Arvore tipo = 1
==================================
:
real    0m0.008s
user    0m0.007s
sys     0m0.000s
```

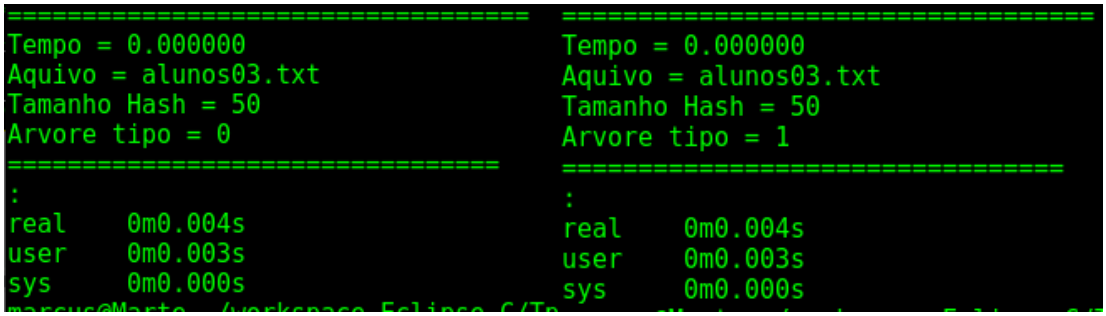
```
==================================
Tempo = 0.000000
Aquivo = alunos02.txt
Tamanho Hash = 100
Arvore tipo = 0
==================================
:
real    0m0.008s
user    0m0.007s
sys     0m0.000s
```

```
==================================
Tempo = 0.000000
Aquivo = alunos02.txt
Tamanho Hash = 100
Arvore tipo = 1
==================================
:
real    0m0.008s
user    0m0.007s
sys     0m0.000s
```

Para o arquivo aluno03.txt:

```
==================================
Tempo = 0.000000
Aquivo = alunos03.txt
Tamanho Hash = 25
Arvore tipo = 0
==================================
:
real    0m0.005s
user    0m0.003s
sys     0m0.000s
```

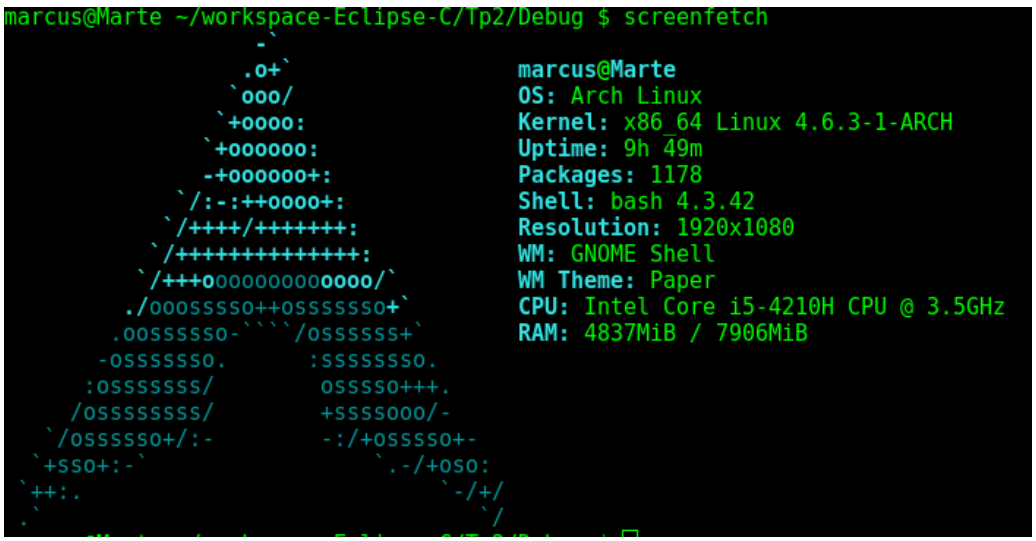
```
==================================
Tempo = 0.000000
Aquivo = alunos03.txt
Tamanho Hash = 25
Arvore tipo = 1
==================================
:
real    0m0.004s
user    0m0.003s
sys     0m0.000s
```



Comparando a função que utiliza árvore binária sem balanceamento e a a outra que usa uma SBB, não notamos diferença alguma. Uma vez que os resultados são idênticos, não podemos argumentar sobre eles.

A única alteração notada é a pequena queda de tempo entre os resultados do arquivo 01 e 02 com o 0. Está queda é esperada , uma vez que o arquivo é menor. Entretanto são poucos dados para se analisar.

Lendo em fóruns, uma das causas desta anomalia é a configuração do sistema e sensibilidade das funções utilizadas. Dentro da main utilizamos long double para armazenar o tempo, e no bash usamos time defaut. Abaixo um resumo do sistema utilizado:



4. Conclusão

Como esperado quanto maior o arquivo mais tempo o programa demora para executar. Apesar disso o programa se comporta plenamente com a leitura dos arquivos.

Não foi possível notar a diferença entre o tempo de execução da função que utiliza árvore binária sem balanceamento e a outra que utiliza uma SBB. Mais casos testes devem ser feitos para um melhor resultado.

5. Referências

http://www.ime.usp.br/~song/mac5710/slides/06bst.pdf

http://www.ime.usp.br/~song/mac5710/slides/09bt.pdf

http://www2.dcc.ufmg.br/livros/algoritmos/cap5/slides/c/completo1/cap5.pdf

https://www.vivaolinux.com.br/dica/Em-C-escrever-em-arquivo-facil