Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DCC004 - ALGORITMOS E ESTRUTURAS DE DADOS II - TURMA TW

Trabalho Prático 2 – Processamento de Texto

Guilherme Saulo Alves
Professoras - Raquel Minardi e Raquel Prates

1. Introdução

O proposito deste trabalho é construir um processador de texto, com as palavras de um texto obtido de um arquivo de entrada, com busca e impressão em ordem alfabética, seguido das suas frequências no texto. Para isso, devemos utilizar o TAD da Árvore binárias de pesquisa. Esse mecanismo é a base de várias aplicações de recuperação de textos na web.

2. Implementação

Uma das primeiras implementações foi criar um tipo para representar um nó de uma arvore binaria, contendo sua chave e os demais atributos e ponteiros necessários. Cada função e estrutura serão explicadas detalhadamente ao longo da documentação.

2.1. Considerações Iniciais

O ambiente de desenvolvimento do código foi no CodeBlocks IDE 10.05 no Sistema Operacional Ubuntu Linux 12.04. Foi usado o compilador GNU GCC Compiler via linha de comando. Não foram feitos testes do programa em ambiente Windows. O comando para compilar é:

```
cd Documentos
gcc -c main.c arvore.c
gcc main.o arvore.o -o tp2
./tp2 < test.in
```

O código está dividido em três arquivos principais: *main.c* com o programa principal, *arvore.c* e *arvore .h.* com a implementação do TAD da arvore binaria de pesquisa.

2.2. Estruturas e Funções

As estruturas e funções que formam o TAD serão apresentadas a seguir:

2.2.1 Tipo *char TipoChave[Max_Alfabeto];*

É um tipo formado por um vetor de caracteres chamada TipoChave, que representa uma palavra lida do texto e inserida em um nó da arvore binaria.

2.2.2 Estrutura *TipoItem;*

A estrutura é formada por uma variável do TipoChave chamada Chave que armazena a palavra a ser inserida na célula e por uma variável int Num para contar o numero de ocorrências de uma certa palavra no texto.

2.2.3 Estrutura *No *Apontador*

A estrutura é formada por um apontador para um nó da arvore binaria que será criada posteriormente.

2.2.4 Estrutura *No*

A estrutura é formada por uma variável do TipoItem Item e por apontadores para o nó da esquerda e para direita de um nó pai.

2.2.5 Tipo *Apontador TipoArvore*

O tipo de dado possui um Apontador para o nó raiz da arvore binaria de pesquisa.

2.2.6 int Vazia(TipoArvore arvore);

Verifica se a arvore está vazia, retornando 0, ou se contém algum elemento, retornando 1. A arvore está vazia quando o ponteiros do primeiro nó aponta para NULL.

2.2.7 void Insere(TipoItem *x, Apontador *no);

Esta função deve inserir uma nova palavra na arvore binaria mantendo a ordem alfabética. Ao inserir uma palavra que já existe na árvore, a operação deve apenas incrementar a ocorrência desta palavra. Ao inserir com sucesso, a função imprime `insere <palavra>.` Caso a palavra já exista, a ocorrência da chave é incrementada e imprenso `incrementa <palavra>.`

2.2.8 int Busca(TipoChave c, Apontador *no);

A busca será feita pela chave passada por parâmetro, retornando 1 caso o elemento seja encontrado e 0 caso contrário. Também deverá ser impresso no arquivo de saída `true` caso a palavra seja encontrada e `false` caso contrário. Na busca, a função verifica se o no contém a palavra-chave, caso não tenha, verifica se a palavra-chave é menor que a chave do no corrente, se sim, a pesquisa continua pelo filho esquerdo, caso contrário, pelo filho direito e assim sucessivamente para cada no visitado, até que a

chave seja encontrada ou uma folha diferente da palavra-chave seja alcançada. O carácter especial # indica a busca da palavra posterior ao carácter. A função imprime a sequência de palavras dos nós visitados.

2.2.9 void Remove(TipoChave c, Apontador *no);

Esta função deve remover uma chave da arvore binaria. Caso o nó que contém a palavra-chave a ser retirado possui no máximo um filho, então a operação é simples. Caso o nó contenha dois filhos, nó a ser retirado deve ser substituído pelo registro mais à esquerda na sua sub árvore direita. A remoção imprime no arquivo de saída se a operação teve sucesso ou não, sendo `remove true <palavra-chave>` para sucesso e `remove false <palavra-chave>` caso contrário, seguindo das palavras-chave. O símbolo @ vindo do arquivo de entrada indica que a remoção da chave seguinte deve ocorrer.

2.2.10 void Antecessor (Apontador q, Apontador *r);

Essa função é ativada quando o nó que contém a chave a ser retirada possui 2 descendentes. A chave a ser retirada é substituído pelo registro mais à esquerda na subárvore direita.

2.2.11 void Imprime(Apontador *no);

A função lista todas as palavras da árvore em ordem alfabética e suas ocorrências no texto. O símbolo & vindo do arquivo de entrada indica que a impressão deve ocorrer. O caminhamento central foi usado para imprimir as palavras do texto em ordem alfabética.

2.2.12 void ExtrairPalavra(*char chave);

A função retira caracteres especiais e letras maiúsculas da chave conforme enunciado. O procedimento ler caracteres por caracteres e verifica se o mesmo é letra maiúscula ou caracter de pontuação. Caso a letra seja maiúscula (`isupper`), ela é transformada em minúscula com a função `tolower` da biblioteca `ctype.c`. Caso seja caracter de pontuação(`ispunct`), pulamos esse caracter. No final, a chave tornara uma sequencia de caracteres representados somente por letras e números.

2.3 Programa Principal

Contém a chamada da função *main*. Faz uso de todas as estruturas e módulos citados acima, trabalha com o tratamento da entrada e saída, bem como a interpretação dos comandos e verificação da sua consistência.

O programa principal lê da entrada padrão, os arquivos de entrada para vários testes. A saída é direcionada para o dispositivo de saída padrão (stdout).

O primeiro passo depois da declaração das variáveis é ler o carácter que corresponde ao teste realizado e comparamos se os nomes são iguais aos definidos do switch. Tais correspondem à @(Remover) #(Busca) &(Imprimir). Caso não seja nenhuma dessas opções, inserimos a palavra.

3. Análise de Complexidade

A análise de complexidade em tempo de execução das funções implementadas será feita em função da variável n que representa o número de vezes que a função é chamada (notação O).

Função Vazia: Essa função é bem simples e é executada 1 vez. Portanto, sua complexidade é constante dada por **O(1).**
Função Busca: A operação de busca em uma árvore binária é igual ao número de nós existentes no caminho desde a raiz da árvore até o nó procurado. Melhor caso: O(1), Pior caso: O(n) e Caso médio: **O(log n).**
Função Inserir: Na inserção o nó é sempre inserido em uma folha, gastando a altura da árvore **O(log n).**
Função Remover: Quando o nó a ser removido está em uma folha no nível mais baixo, gasta-se a altura da árvore, complexidade **O(log n).**
Função Imprime: A função utiliza o caminhamento central para imprimir os nós das arvores em ordem alfabética. A altura de uma árvore binária de busca pode ser **O(n).**
Função ExtraiPalavra: As comparações da função serão executadas n vezes, em que n é o numero de caracteres da palavra lido do arquivo. **O(n)**.
Programa principal: O programa principal executa alguns comandos O(1) antes de aparecer o loop principal para lê cada palavra. É executado um teste a cada laço. A ordem de complexidade dentro do loop é o teste de maior ordem existente O(n). Portanto, como o loop é executado n vezes, a ordem de complexidade do programa principal como um todo é **O(n^2).**

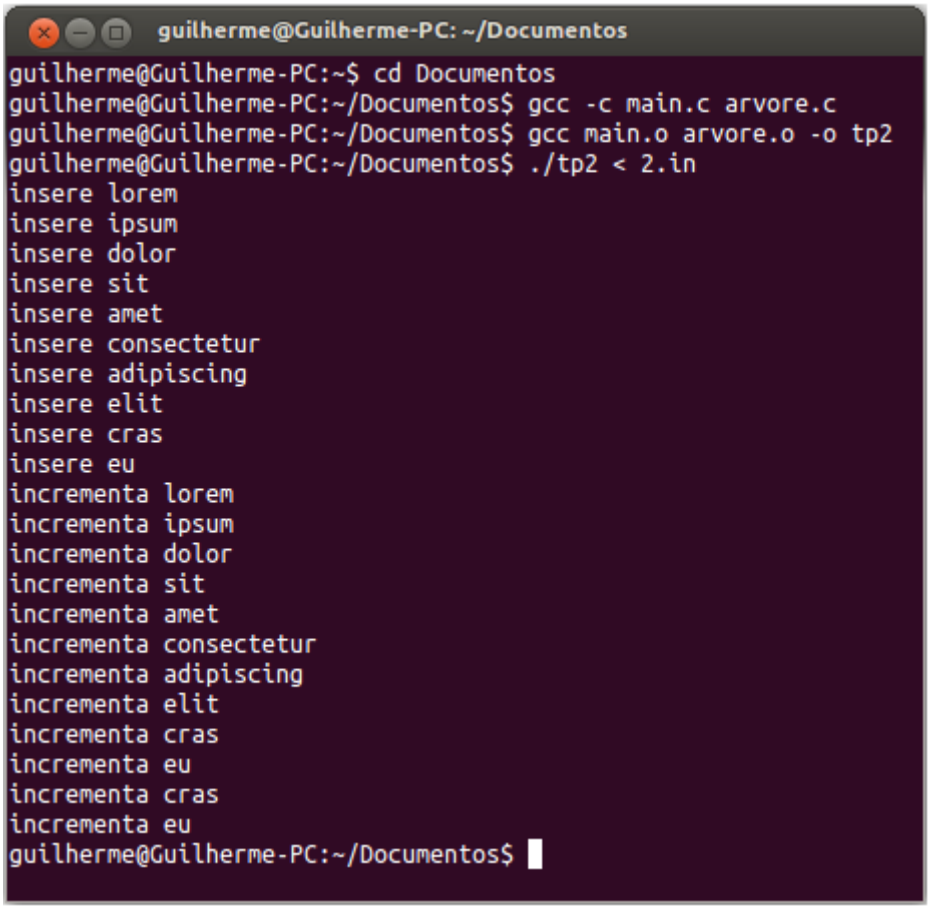
4. Testes

Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. Os testes foram realizados em um Atom CPU D525, com 2 Gb de memória. A figura abaixo mostra a saída de alguns testes disponibilizados no moodle quando executado na linha de comando.

- Entrada: 2.in

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras eu. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras eu. Cras eu.
```

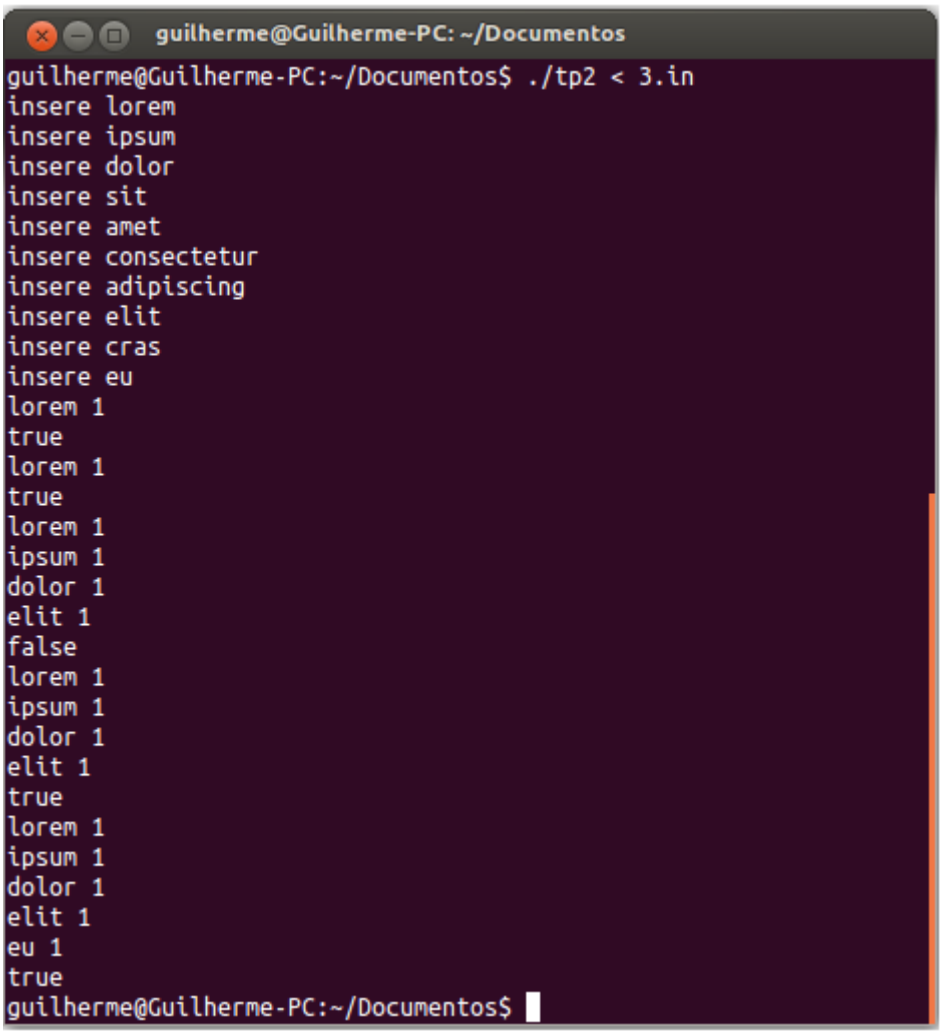
- Saida: 2.out



- Entrada: 3.in

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras eu. #Lorem #lorem #dolorem #elit #eu
```

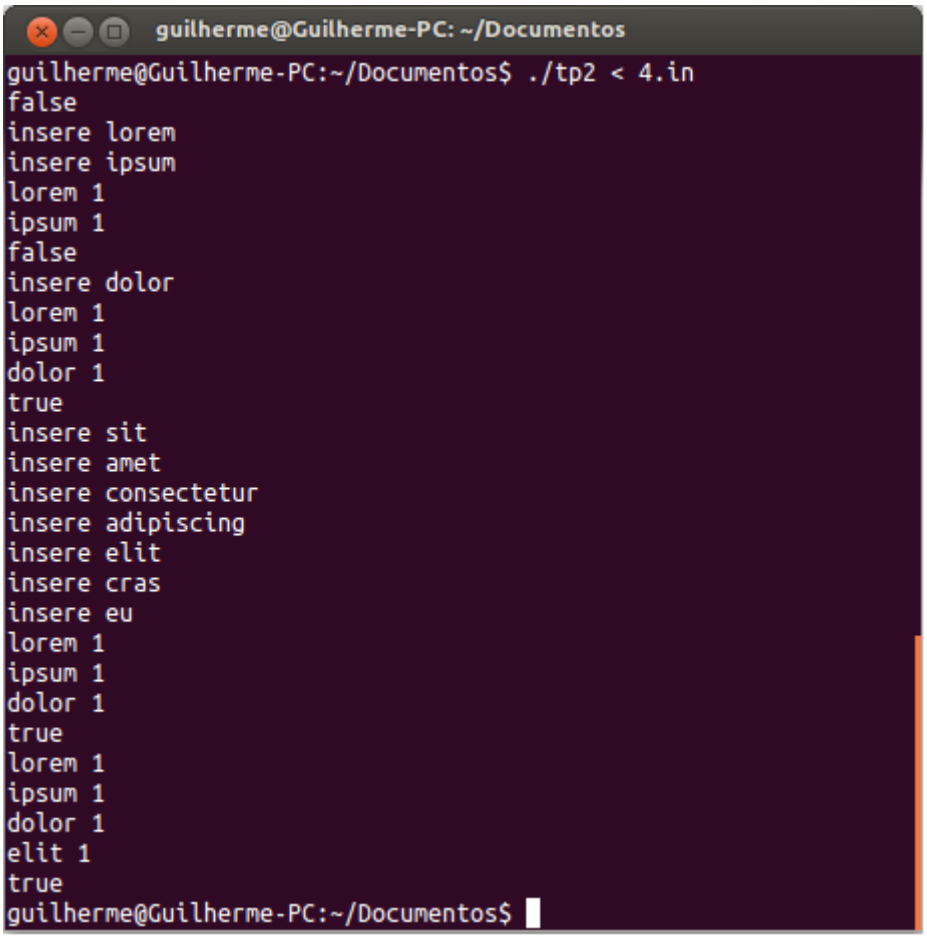
- Saida: 3.out



- Entrada: 4.in

```
#elit Lorem ipsum #dolor dolor #dolor sit amet,
consectetur adipiscing elit. Cras eu. #dolor #elit
```

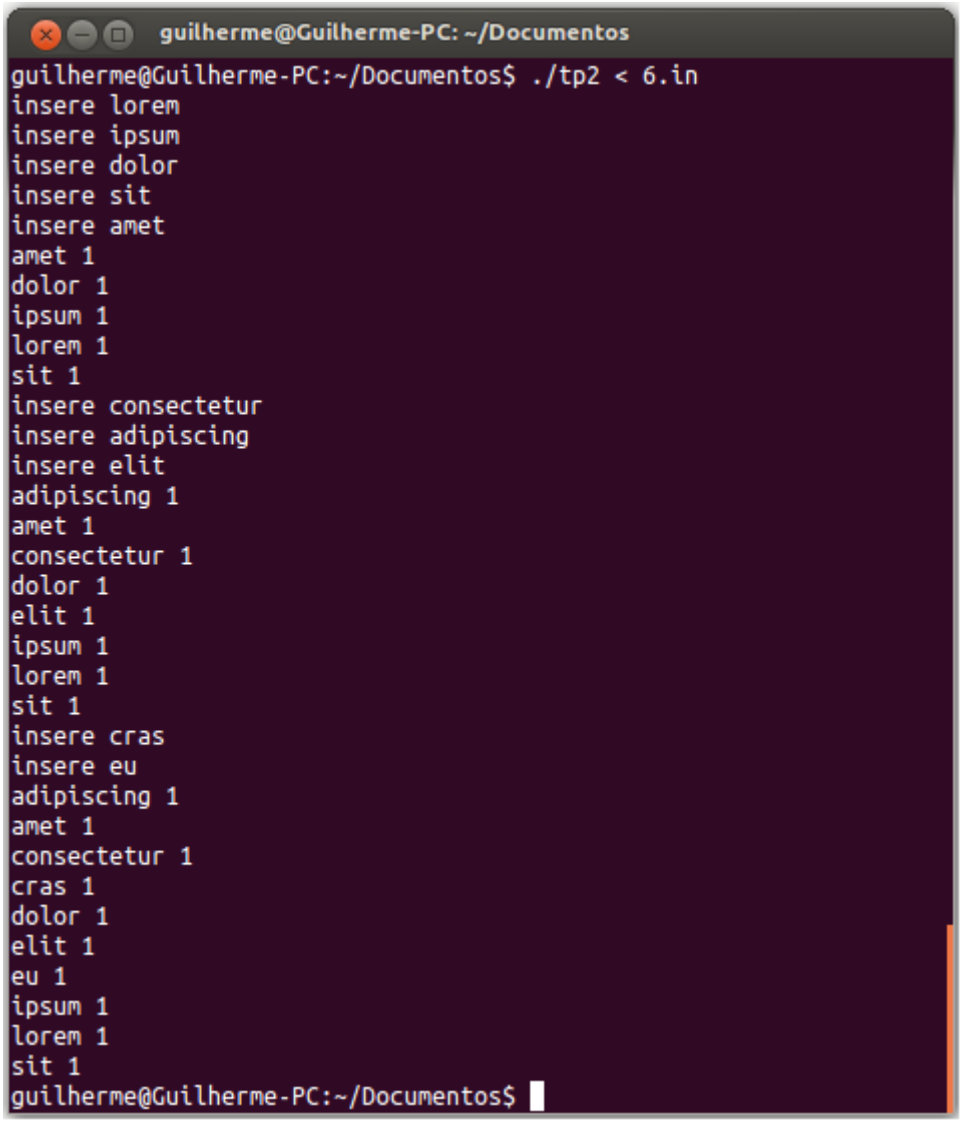
- Saída: 4.out



- Entrada: 6.in

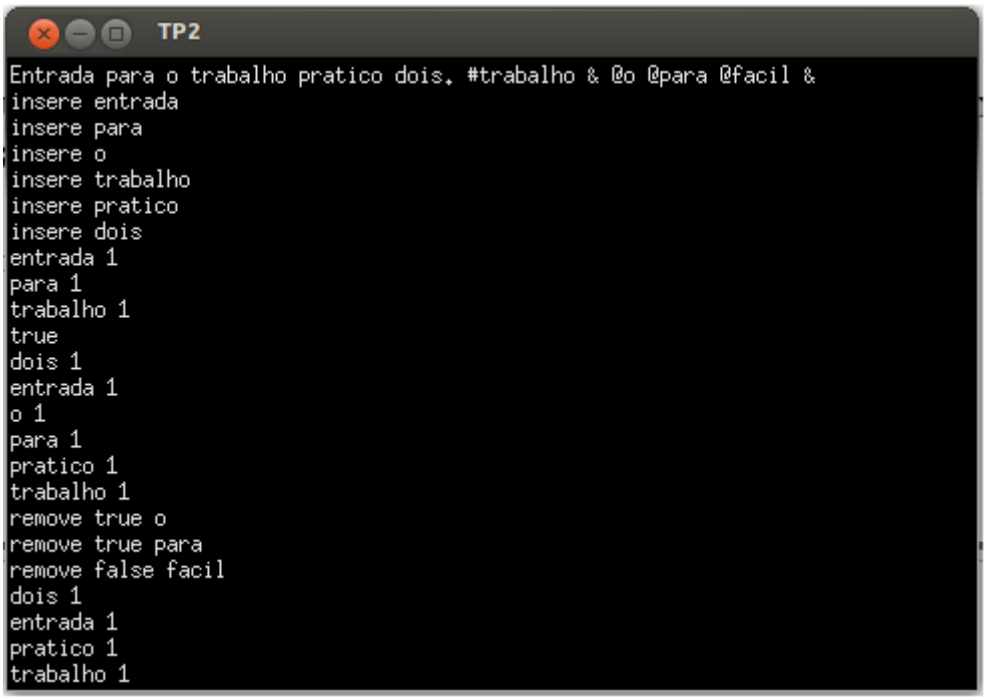
```
Lorem ipsum dolor sit amet &, consectetur adipiscing elit &. Cras eu. &
```

- Saída: 6.out



Também foram realizados testes no *CodeBlocks* durante a elaboração do trabalho para verificar os valores de entrada e saída de algumas variáveis. O exemplo de entrada usado foi o do enunciado do TP:

```
Entrada para o trabalho pratico dois. #trabalho & @o @para @facil &
```



5. Conclusão

Com elaboração desse trabalho foi possível relembrar as técnicas de programação aprendidas na disciplina de AEDS 2 e também fixar os novos conceito de arvores binarias de pesquisa.

A árvore de pesquisa binaria é uma estrutura de dados extremamente eficientes para se fazer buscas. Para inserção, uma árvore binária como a do trabalho é extremamente eficaz e mais simples do que outros métodos de pesquisas.

Além do que, na árvore, como as palavras eram inseridas de maneira lexicográfica, bastou fazer um caminhamento central para que fossem impressas as chaves em ordem alfabética.

5.1 Críticas ao TP2

Desejo relatar algumas inconsistências no que foi cobrado no enunciado do TP2 e no Prático. No enunciado do TP não fala os tipos de palavras que seriam cobradas nos testes do Prático. No Enunciado é relatado o seguinte:

> *"Uma palavra é considerada como uma sequência de letras e dígitos, começando com uma letra. Portanto, ignore sinais de pontuação. Você pode assumir que os textos não terão acentuação".*

> *"Palavras com letras em maiúsculas devem ser primeiramente transformadas para minúsculas antes da inserção na árvore de busca. Desta forma, a mesma palavra apresentada com letras minúsculas ou maiúsculas não serão diferenciada"s.*

> *"Uma palavra pode ocorrer múltiplas vezes na mesma linha de um documento, ou mesmo em múltiplas linhas de um mesmo documento".*

Fica evidente que o enunciado não esclarece que as palavras devem ser quebradas caso a mesma contenha caracteres especiais, como foi esclarecido no fórum de duvidas da turma:

> *"A impressão ocorrerá sempre que for encontrado o símbolo & em qualquer parte do texto. Com isso, pa&la&vra gera insere pa - imprime - insere la - imprime - insere vra e && gera duas impressões consecutivas".*

Meu programa já estava pronto a uma semana antes de liberarem o pratico e todas as saídas estavam compatíveis com testes disponibilizados do *Moodle*. Porém, na sexta-feira dia 15, 4 dias antes da entrega, o prático é liberado e não passei em nenhum teste por causa de uma informação que não estava explicita no enunciado do TP2. Com esse tempo até daria para fazer as modificações, mas não estudo só AEDS II na UFMG e tenho que estudar para as provas finais que ocorre na semana seguinte.

Perante os fatos mencionados, percebe-se uma inconsistência no que é cobrado no enunciado do TP2 e nos testes do pratico. Com um enunciado fraco e arquivos de teste infrutuosos fica inviável fazer o trabalho. O correto seria ter um enunciado e exemplos que condizem com os testes realizados no pratico. O objetivo principal do TP devia ser a implementação da árvore binária em si e não estudar manipulação de strings em nível avançado.

6. Referências

[1] NETO, Samuel Dias – Linguagem C – Intermediário – DCC UFMG - http://homepages.dcc.ufmg.br/~joaoreis/Site%20de%20tutoriais/c_int/ – Acessado em 08 de Novembro de 2013.

[2] Ninja Code – Arvore Binaria com a Linguagem C - http://www.ninjacode.com.br/post/2011/05/11/Arvore-Binaria-com-a-Linguagem-C.aspx - Acessado em 05 de Novembro de 2013.

[3] Nívio Ziviani, Projeto de Algoritmos com implementação em Pascal e C - http://www2.dcc.ufmg.br/livros/algoritmos/ Acessado em 05 de Novembro de 2013.