# BCC202 – Estruturas de Dados I (2022-02)

Departamento de Computação - Universidade Federal de Ouro Preto - MG
Professor: **Pedro Silva** (`www.decom.ufop.br/`)

**AULA PRÁTICA 10**

---

- **Data de entrega: Até 19 de março às 23:59:59.**

**- Procedimento para a entrega:**.

1. Submissão: via ***run.codes***.
2. Os nomes dos arquivos e das funções devem ser especificados considerando boas práticas de programação.
3. Funções auxiliares, complementares aquelas definidas, podem ser especificadas e implementadas, se necessário.
4. A solução deve ser devidamente modularizada e separar a especificação da implementação em arquivos *.h* e *.c* sempre que cabível.
5. Os arquivos a serem entregues, incluindo aquele que contém *main()*, devem ser compactados (*.zip*), sendo o arquivo resultante submetido via ***run.codes***.
6. Caracteres como acento, cedilha e afins não devem ser utilizados para especificar nomes de arquivos ou comentários no código.
7. Siga atentamente quanto ao formato da entrada e saída de seu programa, exemplificados no enunciado.
8. Durante a correção, os programas serão submetidos a vários casos de testes, com características variadas.
9. A avaliação considerará o tempo de execução e o percentual de respostas corretas.
10. Eventualmente, serão realizadas entrevistas sobre os estudos dirigidos para complementar a avaliação.
11. Considere que os dados serão fornecidos pela entrada padrão. Não utilize abertura de arquivos pelo seu programa. Se necessário, utilize o redirecionamento de entrada.
12. Os códigos fonte serão submetidos a uma ferramenta de detecção de plágios em software.
13. Códigos cuja autoria não seja do aluno, com alto nível de similaridade em relação a outros trabalhos, ou que não puder ser explicado, acarretará na perda da nota.
14. Códigos ou funções prontas específicos de algoritmos para solução dos problemas elencados não são aceitos.
15. Não serão considerados algoritmos parcialmente implementados.

***- Bom trabalho!***

## As palavras de um texto

Tia Andréa é uma respeitada professora e tem vários alunos. Em sua última aula, ela prometeu que iria mostrar uma lista de todas as diferentes palavras que existem em um texto, em ordem alfabética, para os seus alunos. Como é um trabalho muito laborioso para uma pessoa fazer e você é um ótimo programador, você se ofereceu para criar um programa para fazer isso para ela. A sua ideia foi utilizar uma árvore binária de pesquisa. Mas você deseja evitar o pior caso das árvores binárias de pesquisa e, por isso, você decidiu utilizar a árvore AVL.

Você pode entender uma palavra como sendo uma sequência de letras maiúsculas ou minúsculas. Palavras com apenas uma letra também devem ser consideradas. Além disso, o seu programa deverá ser "CaSe InSeNsItIvE", isto é, palavras como "Apple", "apple" ou "APPLE" devem ser considerados como a mesma palavra.

Você deverá desconsiderar qualquer caractere que não seja uma letra ('a' a 'z' e 'A' a 'Z').

### Considerações

O código-fonte deve ser modularizado corretamente conforme os arquivos de protótipo fornecidos. Uma árvore binária deve ser criada, preenchida e percorrida para determinação da solução. **Nenhum algoritmo de ordenação deve ser utilizado**. Os procedimentos `criaNo`, `insere`, `fb`, `libera`, `altura` e `inicia`, bem como o TAD AVL são similares aos vistos na aula teórica, porém, precisam ser adaptados para lidarem com strings.

- Não altere o nome dos arquivos.
- O arquivo `.zip` deve conter na sua raiz somente os arquivos-fonte.
- Há vários casos de teste. Você terá acesso (entrada e saída) de casos específicos para realizar os seus testes.

## Especificação da Entrada e da saída

A entrada contém no máximo 10.000 linhas de texto, cada uma delas com no máximo 200 caracteres. O fim da entrada é determinado por **`EOF`**.

Seu programa deve imprimir a altura da árvore criada e uma lista das diferentes palavras que aparecem no texto (**sem repetição**), uma palavra por linha e a quantidade de vezes em que ela apareceu no texto. Todas as palavras devem ser impressas com letras **minúsculas**, em ordem alfabética. Haverá no máximo 5.000 palavras distintas.

Note que na saída não há os caracteres ‘:’, ‘"’, ‘.’, ‘(’, ‘*’, ‘$’, ‘#’, etc.

| **Entrada** | **Saída** |
|---|---|
| `Ex(*$a#.mpl.e:`<br>`Adventures in Disneyland`<br>`Two people were going to Disneyland when`<br>`they came to a fork in the road.`<br>`The sign read:  "Disneyland LEFT."`<br>`So they went home.` | `5`<br>`a 2`<br>`adventures 1`<br>`came 1`<br>`disneyland 3`<br>`e 1`<br>`ex 1`<br>`fork 1`<br>`going 1`<br>`home 1`<br>`in 2`<br>`left 1`<br>`mpl 1`<br>`people 1`<br>`read 1`<br>`road 1`<br>`sign 1`<br>`so 1`<br>`the 2`<br>`they 2`<br>`to 2`<br>`two 1`<br>`went 1`<br>`were 1`<br>`when 1` |

## Diretivas de Compilação

```
$ gcc -c arvore.c -Wall
$ gcc -c pratica.c -Wall
$ gcc arvore.o pratica.o -o exe
```

## Avaliação de *leaks* de memória

Uma forma de avaliar se não há *leaks* de memória é usando a ferramenta valgrind. Um exemplo de uso é:

```
gcc -g -o exe *.c -Wall; valgrind --leak-check=yes -s ./exe < casoteste.in
```

Espera-se uma saída com o fim semelhante a:

```
==38409== ERROR SUMMARY: 0 errors from 0 contexts (suppressed:  0 from 0)
```

Para instalar no Linux, basta usar: `sudo apt install valgrind`.