# BCC202 – Estruturas de Dados I (2023-02)

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

**AULA PRÁTICA 09**

---

- **Data de entrega: Até 8 de dezembro às 23:55.**

**- Procedimento para a entrega:**.

1. Submissão: via ***Moodle***.
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 ***Moodle***.
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!***

## UPA superlotada

Dado o surgimento de uma doença extremamente infecciosa, a Unidades de Pronto Atendimento (UPA) de uma cidade não está conseguindo organizar os pacientes em uma ordem específica. Para contornar esse problema, você foi contratado para desenvolver um algoritmo para ordenar os pacientes. Como você é muito sagaz e esperto e sabe que o pior caso não vai acontecer, você optou por implementar o *Quick Sort* iterativo.

Os pacientes presentes na UPA possuem um nome, idade e o estado de saúde em que ele se encontra (leve=1, médio=2 ou grave=3). Eles serão ordenados baseado nas suas informações.

A única coisa que você deve se atentar é que estados de saúde graves (maior prioridade) devem vir antes dos médios que por sua vez devem vir antes dos casos leves. A idade deve ser usada no desempate (maior prioridade é a maior idade). Se for igual, use o nome (ordem alfabética - letra A tem mais prioridade do que o Z).

### Considerações

O código-fonte deve ser modularizado corretamente conforme os arquivos de protótipo fornecidos. Você deve criar um TAD `UPA` que tem um vetor do TAD `Paciente` e a quantidade de pacientes. O TAD `Paciente` tem um nome (*char* com até 50 caracteres), idade (*int)* e o estado de saúde (*int*, 1 - leve, 2 - médio e 3 - grave). O TAD `Paciente` **precisa** ser alocado e desalocado dinamicamente, o de TAD `UPA` pode ser estático.

As funções do TAD `Pilha` já estão implementados.

- 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 primeira linha da entrada é um número inteiro *n* com o número de pacientes e depois com *p* pacientes com o nome, idade e estado de saúde, separados por um espaço em branco.

Você deve usar o método de ordenação *Quick Sort* iterativo para ordenar os pacientes da UPA. A saída do seu programa é a impressão do vetor de pacientes. A impressão do vetor é da primeira a última posição.

| Entrada | Saída |
|---|---|
| 5<br>Alice 28 1<br>Miguel 62 1<br>Sophia 26 1<br>Arthur 17 2<br>Helena 78 1 | Arthur 17 2<br>Helena 78 1<br>Miguel 62 1<br>Alice 28 1<br>Sophia 26 1 |

| Entrada | Saída |
|---|---|
| 5<br>Alice 28 1<br>Miguel 62 1<br>Sophia 17 1<br>Arthur 17 2<br>Helena 78 1 | Arthur 17 2<br>Sophia 17 2<br>Helena 78 1<br>Miguel 62 1<br>Alice 28 1 |

## Diretivas de Compilação

```
$ gcc -c sort.c -Wall
$ gcc -c pratica.c -Wall
$ gcc -c pilha.c -Wall
$ gcc pilha.o sort.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`.