BCC202 – Estruturas de Dados I

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

Heap Sort

- **Submissão via *Moodle***.

- **Data e hora de entrega disponíveis no *Moodle*.**

- Procedimento para a entrega:.

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

- Bom trabalho!

Sistema de Resgate

Considere a situação onde você está trabalhando no desenvolvimento de um sistema para gerenciar resgates de emergência em uma grande cidade. As chamadas de emergência têm diferentes níveis de prioridade, dependendo da gravidade da situação. Um sistema eficiente é essencial para garantir que as situações mais urgentes sejam atendidas primeiro. As prioridades variam de **1** a **10**, onde **1** representa uma chamada de emergência de baixa prioridade (não urgente) e **10** uma chamada de extrema urgência (como um acidente grave ou ataque cardíaco).

Para isso, você deverá utilizar uma estrutura de dados de **fila de prioridade**, onde as chamadas com **maior prioridade sempre serão atendidas primeiro**. Se caso houver **empate** na prioridade, considere a **pessoa mais velha** com maior prioridade (**não há prioridade sobre pessoas da mesma idade**). A fila de prioridade deve ser implementada usando um *Max-Heap*, garantindo que a chamada com a maior prioridade esteja sempre no topo da fila.

Sua missão é implementar o TAD *Heap* que vai garantir a ordem de prioridade das chamadas de emergência. A sua escolha se deu pelo custo $O(logn)$ para inserir uma nova chamada de emergência.

Considerações

O código-fonte deve ser modularizado corretamente conforme os arquivos de protótipo fornecidos. Implementar o TAD `Heap` que representa o *Max-Heap* e deve ser implementado como um vetor. O heap deverá armazenar as chamadas de emergência em **ordem de prioridade**. Terão três chamadas: (i) a função `inserirChamada` adiciona uma nova chamada com sua prioridade na fila; (ii) a função `atenderChamada` remove a chamada de maior prioridade para ser atendida; e (iii) a função `consultarSeTemProximaChamada` apenas retorna se ainda há chamadas no heap, sem fazer nenhuma remoção da fila.

O TAD `Heap` é uma fila de prioridade do tipo TAD `Chamada`, o qual armazena informações básicas sobre a chamada de emergência, como: (i) nome do paciente (21 caracteres); (ii) prioridade de um 1 a 10; e um TAD `Data` com dia, mês e ano de nascimento.

- 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 será dada por várias linhas. Uma linha pode começar com o código **1** que significa que uma nova chamada será inserida. Ela também pode começar com o código **2** que significa um chamado foi resolvido (ou seja, removido do *heap*) e deve ser impresso na tela. Por fim, há o código **3**, o qual representa fim da execução e que todos os chamados devem ser removidos do *heap* e impressos em tela.

A saída corresponde a ordem de remoção do *heap*.

Entrada	Saída

`1 jose 5 1 5 1990`
`1 guilherme 6 9 3 1998`
`1 valentina 9 1 3 1945`
`2`
`1 jesus 10 25 1 0000`
`3`
	`09 | 01/03/1945 | valentina`
	`10 | 25/01/0000 | jesus`
	`06 | 09/03/1998 | guilherme`
	`05 | 01/05/1990 | jose`

Entrada	Saída

`1 jose 5 1 5 1990`
`1 guilherme 6 9 3 1998`
`1 valentina 9 1 3 1945`
`1 jesus 10 25 1 0000`
`3`
	`10 | 25/01/0000 | jesus`
	`09 | 01/03/1945 | valentina`
	`06 | 09/03/1998 | guilherme`
	`05 | 01/05/1990 | jose`

Diretivas de Compilação

```
$ gcc -c heap.c -Wall
$ gcc -c pratica.c -Wall
$ gcc heap.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`.