

## INF 213 - Roteiro da Aula Prática

O objetivo desta aula é praticar o uso e implementação de pilhas, filas e filas de prioridade.

Arquivos fonte e diagramas utilizados nesta aula:

[https://drive.google.com/file/d/1qXGYa12ODQF5eMSxbS\\_MxzIN5lrMG24F/view?usp=sharing](https://drive.google.com/file/d/1qXGYa12ODQF5eMSxbS_MxzIN5lrMG24F/view?usp=sharing)

Considere o programa “driver.cpp”. A não ser que seja dito o contrário em uma determinada etapa, implemente cada função “etapaN()” adicionando a implementação da respectiva etapa (você pode usar includes na área de cada etapa para incluir classes ou dividir seu código em vários arquivos).

(para testar a sua implementação das etapas 1 e 2, crie um arquivo Mediana.cpp vazio e submeta junto ao seu trabalho -- se não fizer isso seu código não compilará. A implementação do arquivo Mediana.cpp será realizada na etapa 3)

### Etapa 1

Usando a classe MyList2, implemente uma fila de prioridades onde a inserção seja  $O(1)$ ,  $\text{top()}/\text{pop()}$  sejam  $O(n)$ . Faça sua implementação na classe MyPriorityQueue2 (a função etapa1() do driver já está implementada -- ela realizará testes em sua classe). O arquivo MyPriorityQueue2.h contém a interface de sua classe (não altere nada na interface, nem na parte privada -- apenas implemente os métodos). Sua implementação será **bem simples!**

Teste seu programa usando os arquivos de teste providos como exemplo e meça o tempo de execução para as diferentes entradas (utilizando o utilitário time do Linux, ou algo similar). Anote em uma tabela (em um arquivo README.txt) os tempos para os diferentes tamanhos de entrada e relate o que observou sobre o comportamento das duas classes. Nas entradas eventualmente faremos chamadas a  $\text{top()}$  antes do  $\text{pop()}$ , ou seja, teremos bem menos  $\text{top()}$  do que  $\text{pop()}$  (não considere o número de  $\text{top()}$  ao analisar os tempos).

Obs: ao medir o tempo, redirecione a saída do seu programa para um arquivo (ex: ./a.out < entrada.txt > saidaTemp.txt).

### Etapa 2

Recentemente apareceu um jacaré na UFV e ele tem perturbado muito as capivaras. Assim, um grupo formado por N desses simpáticos roedores decidiu lutar contra o invasor da lagoa.

O jacaré inicialmente tem J pontos de vida e cada capivara i tem  $P_i$ . Quando uma capivara com X pontos ataca o jacaré, a pontuação do jacaré é reduzida em X unidades e a da capivara é reduzida para  $X/2$  (arredondado para baixo, quando não par). O jacaré será derrotado quando sua pontuação chegar a 0 ou menos.

Crie uma função (chamada `etapa2()`) que determina o número mínimo de ataques que as capivaras deverão realizar para derrotar o jacaré. Se isso não for possível, imprima uma linha contendo -1.

A entrada começa com N, seguido de J. A seguir, há as N pontuações de cada capivara.  $0 \leq N \leq 1$  milhão,  $0 \leq J \leq 1$  bilhão,  $0 \leq P_i \leq 1$  bilhão.

Exemplo de entrada (a entrada real do problema terá o número “2” no início, indicando que se trata da segunda etapa da aula):

Entrada	Saída
4 20 10 5 7 1	3
4 20 1 2 3 4	-1

No primeiro exemplo, a capivara com 5 pontos poderia atacar o jacaré primeiro. Assim, J viraria  $20 - 5 = 15$  e as pontuações das capivaras virariam: 10 2 7 1. A seguir, a capivara com 10 pontos poderia atacar (a pontuação do Jacaré se tornaria 5 e as das capivaras: 5 2 7 1). Finalmente, a capivara com 5 pontos (ou a com 7) poderia realizar um último ataque, zerando os pontos do jacaré.

### Etapa 3

Este é um problema com estilo similar ao de entrevistas de emprego em grandes empresas de tecnologia como Google e Facebook.

Crie uma estrutura de dados (usando uma classe) chamada Mediana (salve o cabeçalho com o nome `Mediana.h` e a implementação com o nome `Mediana.cpp`), que deverá possuir as seguintes funções públicas:

- `insere(x)` : insere o inteiro x na estrutura de dados -- a complexidade máxima deverá ser  $O(\log n)$
- `getMediana()` : retorna a mediana dos números já inseridos na classe -- a complexidade máxima deverá ser  $O(1)$ .

Observe que, por simplicidade, não utilizaremos templates (vamos supor que apenas inteiros serão armazenados na classe).

Note que caso o número de elementos na estrutura seja par, a mediana será a média de dois valores. Calcule a média usando divisão inteira (i.e., a mediana da lista [3,6] será 4, não 4.5).

Utilize apenas as estruturas de dados vistas em sala de aula (você não pode utilizar a STL, mas pode reutilizar o código das estruturas de dados mostradas em sala)

Dica: utilize duas filas de prioridade na sua classe! Em que elas poderiam ajudar?

Utilizando sua classe, implemente a função “etapa3()” de modo que ela le um número N. Então, N números deverão ser lidos da entrada padrão. Para cada número lido, imprima a mediana de todos os números lidos até o momento e um espaço em branco. Imprima uma nova linha (vazia) após o resultado.

Entrada: 5 1 2 3 4 5	Saida esperada: 1 1 2 2 3
----------------------------	------------------------------

Explicação: a mediana da lista [1] é 1, a mediana (parte inteira) de [1,2] é 1, a mediana de [1,2,3] é 2, a mediana de [1,2,3,4] é 2, a mediana de [1,2,3,4,5] é 3.

Os casos de teste poderão ter até 1 milhão de números.

Observe que uma implementação ineficiente (MedianaLenta) e trivial foi provida como exemplo. Qual a ordem de complexidade da etapa3() utilizando essa implementação? E usando sua implementação mais eficiente?

Utilizando o utilitário time() do Linux, meça o tempo da versão provida para as entradas input\_n.txt (n=1,2,3,4,5,6) e compare com o tempo da sua versão. Como os tempos crescem a medida em que n cresce?

A implementação que você entregar pelo submitty deverá usar a sua classe mediana, não a classe MedianaLenta.

Exemplo de aplicação para este problema: uma empresa de cartão de crédito pode verificar se o valor gasto em um dia está muito maior do que a mediana dos gastos do cliente até o momento (isso poderia indicar que o cartão foi roubado).

#### **Etapas 4**

Uma importante aplicação de filas e pilhas é realizar operações do tipo “busca” (isso será estudado em mais detalhes no final da disciplina de Estrutura de Dados, em Projeto e Análise de Algoritmos e em Teoria e Modelos de Grafos). Mais especificamente, pilhas podem ser facilmente utilizadas para realizar buscas “em profundidade”, enquanto filas podem ser utilizadas para implementar buscas “em largura” (não se preocupe com a definição formal desses dois tipos de busca agora).

Um exemplo de problema que pode ser resolvido com busca é determinar se é possível atingir uma saída de um labirinto (representado por uma matriz retangular onde cada elemento representa uma “celula” do labirinto) a partir de um ponto inicial.

Um algoritmo para resolver esse problema consiste no seguinte:

Crie uma pilha ou fila S e adicione a S as coordenadas do ponto inicial

Enquanto S não estiver vazia, remova o primeiro elemento de S e se ele ainda não tiver sido visitado marque-o como visitado e adicione seus vizinhos (que possam ser visitados, ou seja, que não representarem um muro) a S. Se o elemento já tiver sido visitado ignore-o.

Pode-se mostrar que será possível sair do labirinto se, e somente se, a saída dele for eventualmente visitada. Observe que ao marcar um elemento como visitado evitamos visitá-lo duas vezes (o que poderia causar, por exemplo, um loop infinito -- por que?).

Teste os programas `resolveLabirinto.cpp` e `resolveFila.cpp` (utilizando os labirintos de exemplo). O primeiro utiliza uma pilha para determinar se é possível achar a saída de um labirinto a partir da entrada enquanto o segundo utiliza uma fila. Durante o processo as células do labirinto que forem sendo visitadas são marcadas e exibidas em tela.

Observe que a letra X representa uma parede do labirinto, \$ representa a saída, @ representa o início e “.” representa uma área vazia. Observe também que supomos que cada célula é vizinha de, no máximo, 4 outras células (as vizinhas que estiverem na mesma linha ou na mesma coluna).

Qual a diferença que você observou entre a busca em largura e em profundidade?

Anotar seus comentários e observações em um arquivo `README.txt` e submeter junto com a prática (nesta etapa não há implementações a serem submetidas).

### **Submissão da aula prática:**

A solução (exceto a implementação da etapa 4, que não precisa ser entregue) deve ser submetida até as 18 horas do dia 11/06 utilizando o sistema `submittity` (`submittity.dpi.ufv.br`). Atualmente a submissão só pode ser realizada dentro da rede da UFV.