ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
Departamento de Engenharia de Computação e Sistemas Digitais

PCS 3111 - LABORATÓRIO DE PROGRAMAÇÃO ORIENTADA A OBJETOS PARA A ENGENHARIA ELÉTRICA

EXERCÍCIO PROGRAMA 2 – 2º SEMESTRE DE 2021

Resumo

Os EPs de PCS3111 têm como objetivo exercitar os conceitos de Orientação a Objetos aprendidos em aula ao implementar um simulador de troca de mensagens entre computadores usando uma rede similar à Internet (mas bem simplificada). O software desenvolvido no EP1 será melhorado neste EP2 para permitir que ele tenha outras funcionalidades.

1 Introdução

O EP1 tratou da troca de mensagens entre nós intermediários da rede, os *roteadores*. No EP2 simularemos a rede com mais um tipo de nó: os *hospedeiros* (ou *host*, em inglês). Exemplos de hospedeiros (também chamados de *sistemas finais*) são computadores, celulares, smart TVs e autofalantes inteligentes. Assim como no EP1, quem quiser ver com detalhes como a Internet realmente funciona pode consultar o livro do Kurose e Ross[1]. Esse assunto também será tratado por *PTC3360 - Introdução a Redes e Comunicações*, que é uma disciplina do 3º ano de Engenharia Elétrica.

Da mesma forma que os roteadores, os hospedeiros possuem endereços e recebem pacotes. Mas, além disso, os hospedeiros podem executar vários *processos* ao mesmo tempo. Por exemplo, um hospedeiro como um computador pode executar um navegador (cada aba é um processo), o *Code::Blocks*, um leitor de PDF e o *Spotify*. Quando o hospedeiro está em rede (como a Internet), esses processos podem trocar *pacotes* com processos em outros hospedeiros. Por exemplo, o programa do *Spotify* troca pacotes com o servidor do *Spotify* para tocar uma música. Este EP tratará da troca de pacotes entre processos – e não simplesmente a troca de pacotes entre nós[2], que foi o foco do EP1. Por simplicidade, teremos apenas um tipo de processo: o chat. Um chat consegue trocar mensagens com um outro chat (enviar e receber), executando em um outro hospedeiro.

A troca de pacotes entre processos tem objetivos diferentes da troca de pacotes entre nós. Por exemplo, pode-se desejar que os pacotes cheguem na ordem que foram enviados e que nenhum pacote seja perdido. Outra função é identificar o processo de destino, ou seja, com qual dos vários processos do destinatário se deseja conversar. A forma de fazer isso é através de um número de *porta*. Cada processo fica esperando os pacotes que são recebidos pelo hospedeiro em uma porta específica. Por exemplo, o

[1] KUROSE, J. F.; ROSS, K. W. *Computer Networking: A top-down approach*. Pearson, 7.ed., 2017.

[2] Como comentado no enunciado do EP1, a arquitetura de uma rede é tipicamente organizada em várias camadas. No EP1 simulamos apenas a *camada de rede*, que tem como principais funções o endereçamento dos nós e roteamento dos pacotes. Neste EP2 simularemos também a *camada de transporte*, que atua na comunicação fim-a-fim realizando a troca de pacotes entre processos.

programa do *Spotify* (rodando no dispositivo do usuário) tipicamente espera pacotes da porta 4070. Na arquitetura da Internet, os números de porta são tratados pelos protocolos TCP ou UDP. O pacote nessa camada também possui um nome diferente: é chamado de *segmento*.

Para que os segmentos enviados por um processo cheguem ao processo destinatário, é necessário que os nós conversem entre si. A forma como uma rede faz isso é similar à forma como enviamos cartas pelos correios. O nome do destinatário e o conteúdo da carta não são relevantes para os correios; eles só precisam olhar o endereço no envelope para fazer com que a carta chegue ao local de destino. Nesse local podem existir vários moradores, então tipicamente alguém do local repassa a carta à pessoa que é o destinatário – e que será quem verá o conteúdo. Em uma rede, as pessoas seriam os *processos*, os armazéns dos correios (onde as cartas ficam armazenadas para serem entregues) seriam os *roteadores e* as residências das pessoas seriam os *hospedeiros*; o *segmento* seria o conteúdo da carta junto com o nome do destinatário (o nome da pessoa é a *porta*), e o *datagrama* seria o envelope. Portanto, o segmento é o conteúdo do datagrama e os nós repassam os pacotes olhando apenas para o datagrama.

Neste EP também trataremos da qualidade de serviço (chamado de QoS, de *quality of service*) em um roteador. Em roteadores reais, é possível reservar uma banda para algumas aplicações específicas – por exemplo, para assistir vídeos em *streaming*. Para tratar disso de forma simplificada, no EP usaremos apenas uma fila de prioridade para privilegiar pacotes para um determinado destino.

1.1 Objetivo

O objetivo deste projeto é fazer um simulador de uma rede simplificada, evoluindo o programa desenvolvido no EP1. Neste EP será possível que a rede possua também hospedeiros, os quais podem possuir chats, e roteadores que priorizam destinos. Para tornar o programa mais flexível, também será possível carregar uma rede descrita em um arquivo.

A solução deve empregar adequadamente conceitos de orientação a objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor, herança, classe abstrata, membros com escopo de classe, programação defensiva, persistência em arquivo e os containers da STL. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

Para desenvolver o EP deve-se manter a mesma dupla do EP1. Será possível apenas **desfazer a dupla**, mas não formar uma nova.

2. Projeto

Deve-se implementar em C++ as classes **Agendador**, **Chat**, **Datagrama**, **Evento**, **Fila**, **FilaComPrioridade**, **Hospedeiro**, **No**, **PersistenciaDeRede**, **Rede**, **Roteador, RoteadorComQoS, Segmento** e **TabelaDeRepasse**, além de criar uma `main` que permita o funcionamento do programa como desejado.

Cada uma das classes deve ter um arquivo de definição (".h") e um arquivo de implementação (".cpp"). Os arquivos devem ter exatamente o nome da classe. Por exemplo, deve-se ter os arquivos "Datagrama.cpp" e "Datagrama.h". Note que você deve criar os arquivos necessários. Não se esqueça de configurar o Code::Blocks para o uso do C++11 (veja a apresentação da Aula 03 para mais detalhes).

Atenção:

1. O nome das classes e a assinatura dos métodos **devem seguir exatamente** o especificado neste documento. As classes não devem possuir outros membros (atributos ou métodos) **públicos** além dos especificados**, a menos dos métodos definidos na superclasse e que precisarem ser redefinidos**. Excepcionalmente a classe **FilaComPrioridade** *pode* possuir o método `enqueue(Datagrama* d, int prioridade)`, conforme especificado na Seção 2.4. Note que você poderá definir atributos e método privados e protegidos caso necessário.
2. Não é permitida a criação de outras classes além dessas. A exceção é a classe **Elemento** que pode ser criada caso se deseje criar a fila usando uma lista ligada.
3. Não faça `#define` para constantes. Você pode (e deve) fazer `#ifndef/#define` para permitir a inclusão adequada de arquivos.

O não atendimento a esses pontos pode resultar em **erro de compilação** na correção automática e, portanto, nota `0` na correção automática.

Em relação às exceções (assunto da Aula 9), todas as especificadas são da biblioteca padrão (não se esqueça de fazer `#include <stdexcept>`). O texto usado como motivo da exceção não é especificado no enunciado e não será avaliado. Jogue as exceções criando um objeto usando `new`. Por exemplo, para jogar um `invalid_argument` faça algo como:

```
throw new invalid_argument("Mensagem de erro");
```

Caso a exceção seja jogada de outra forma, pode haver erros na correção e, consequentemente, desconto na nota.

2.1 Classe Segmento

Um **Segmento** é o pacote transmitido entre processos. Ele possui a porta de origem, a porta de destino e o dado que será transmitido. Com isso a classe **Segmento** deve possuir apenas os seguintes métodos **públicos**:

```
Segmento(int portaDeOrigem, int portaDeDestino, string dado);
virtual ~Segmento();

virtual int getPortaDeOrigem();
virtual int getPortaDeDestino();
virtual string getDado();

virtual void imprimir();
```

Os métodos `getPortaDeOrigem`, `getPortaDeDestino` e `getDado` devem retornar, respectivamente, os valores da porta de origem, da porta de destino e do dado informados no construtor.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado.

2.2 Classe Datagrama

Um **Datagrama** é o pacote que é transmitido entre roteadores. A diferença do **Datagrama** no EP2 é que o conteúdo não é uma **string**: ele é um **Segmento**. Um impacto disso é no destrutor e no método

`getDado` que foi renomeado para `getSegmento`. Com isso a classe **Datagrama** deve possuir apenas os seguintes métodos **públicos**:

```
Datagrama(int origem, int destino, Segmento* dado);
virtual ~Datagrama();

virtual int getOrigem();
virtual int getDestino();
virtual Segmento* getSegmento();

virtual void imprimir();
```

Os métodos `getOrigem`, `getDestino` e `getSegmento` devem retornar, respectivamente, os valores do endereço de origem, do endereço de destino e do **Segmento** informados no construtor. No destrutor do **Datagrama** destrua o **Segmento** recebido.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado.

2.3 Classe Fila

A **Fila** implementa uma fila de **Datagramas**, a qual será usada por um **No**. Essa classe teve poucas alterações no EP2, apenas para permitir o uso de exceções (note que a assinatura de enqueue mudou por causa disso). Com isso, a classe **Fila** deve possuir apenas os seguintes métodos **públicos**:

```
Fila(int tamanho);
virtual ~Fila();

virtual void enqueue(Datagrama* d);
virtual Datagrama* dequeue();
virtual bool isEmpty();
virtual int getSize();

virtual void imprimir();
```

Assim como no EP1, a **Fila** pode ser implementada da forma que você achar mais apropriada (pode ser uma *fila circular* usando vetor ou uma *lista ligada*[3]).

O construtor deve receber o tamanho máximo da **Fila**, o qual deve representar o número máximo de elementos que a **Fila** deve efetivamente possuir. Ou seja, se o tamanho for 4, no máximo 4 **Datagramas** poderão ser colocados na fila em um determinado momento. Ao tentar fazer o enqueue do 5º **Datagrama** deve ocorrer um *overflow*. Mas note que se forem colocados 4 **Datagramas**, em seguida retirados os 4 **Datagramas**, deve ser possível colocar mais 4 **Datagramas**. Caso o tamanho informado seja menor ou igual a 0, o construtor deve jogar uma exceção do tipo `invalid_argument` da biblioteca padrão.

No destrutor destrua os objetos alocados dinamicamente e os **Datagramas**.

[3] Caso você deseje implementar usando uma *lista ligada*, crie uma classe chamada **Elemento** para evitar problemas na correção automática.

O método enqueue não tem mais retorno. Ele deve inserir o **Datagrama** passado como parâmetro na última posição da **Fila**. Caso a **Fila** não tenha espaço disponível (*overflow*), esse método não deve inserir o **Datagrama** e deve jogar uma exceção do tipo `overflow_error`, da biblioteca padrão. O método dequeue deve remover o primeiro **Datagrama** da **Fila** e o retornar. Em caso de *underflow*, ou seja, a tentativa de remover um elemento em uma **Fila** vazia, o método deve jogar a exceção `underflow_error`, da biblioteca padrão.

O método `isEmpty`[4] informa se a **Fila** está vazia (retornando `true`) ou não (retornando `false`). O método `getSize` deve informar o número de **Datagramas** que estão na **Fila** (note que não é o tamanho alocado).

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado.

2.4 Classe FilaComPrioridade

A **FilaComPrioridade** é um subtipo de **Fila** que implementa uma fila de prioridades de **Datagramas** simplificada. Ela será usada por roteadores que priorizam pacotes para um determinado destino (**RoteadorComQoS**). O único método adicional à classe mãe **Fila** é um método enqueue sobrecarregado que recebe também um booleano, informando a prioridade. Com isso ela deve ter os seguintes métodos públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
FilaComPrioridade(int tamanho);
virtual ~FilaComPrioridade();

virtual void enqueue(Datagrama* d, bool prioritario);
```

Da mesma forma que na classe mãe, o construtor deve receber o tamanho máximo da **FilaComPrioridade** e jogar uma exceção do tipo `invalid_argument` da biblioteca padrão caso o tamanho seja menor ou igual a 0. O destrutor também deve destruir os objetos alocados dinamicamente e os **Datagramas**.

Essa classe possui um método enqueue sobrecarregado que recebe, além do **Datagrama**, um booleano informando se ele é prioritário. Um **Datagrama** prioritário deve ser retornado antes dos **Datagramas** não prioritários. Mas note que ainda assim deve ser seguida a política de fila (o primeiro a entrar deve ser o primeiro a sair *para aquela prioridade*). Por exemplo, considere uma **FilaComPrioridade** em que foram enfileirados os **Datagramas** *d1* (sem prioridade), *d2* (sem prioridade), *p1* (com prioridade) e *p2* (com prioridade), nesta ordem. A chamada do método dequeue deve retornar os **Datagramas** na seguinte ordem: *p1*, *p2*, *d1*, *d2*. Assim como o método enqueue da classe mãe, esse método deve jogar uma exceção do tipo `overflow_error` caso a **FilaComPrioridade** não tenha espaço disponível (*overflow*).

No caso do método enqueue herdado (que só recebe um **Datagrama**), ele deve enfileirar o **Datagrama** considerando que ele *não tem prioridade*. Um detalhe de C++: caso você **não** redefina esse método, você deve colocar o comando `using Fila::enqueue;` (sem retorno e sem parâmetros) dentro do

[4] Não usaremos o nome `Queue-Empty` de PCS3110 pois ele é redundante em uma solução Orientada a Objetos – o método é da classe **Fila** (*queue* em inglês) e o nome não precisa repetir essa informação. Além disso, o '-' não é um caractere válido para nomes em C++.

rótulo `public` (ou seja, junto com os métodos da classe). O motivo disso é que, por uma decisão de projeto, o C++ não permite a sobrecarga entre escopos diferentes[5].

Caso você deseje, você *pode* criar (não é obrigatório) um outro método sobrecarregado que recebe a prioridade como um número inteiro (os valores de menor prioridade ficariam no começo da fila, o que pode ser útil para o **Agendador**). Ou seja, se desejar você pode criar o método:

```
void enqueue(Datagrama* d, int prioridade);
```

Esse método não será testado.

2.5 Classe TabelaDeRepasse

Uma **TabelaDeRepasse** mapeia endereços a **Nos**, gerenciando para qual **No** deve ser repassado o **Datagrama** que possui um determinado endereço de destino. O funcionamento dessa classe é praticamente o mesmo do especificado no EP1. A principal diferença é que os métodos trabalham com **Nos** em vez de **Roteadores**, uma vez que a rede é composta por **Roteadores** e **Hospedeiros**. Além disso, deve-se jogar exceções no construtor e nos casos de erro de mapeamento. Com isso, a classe **TabelaDeRepasse** deve possuir apenas os seguintes métodos **públicos**:

```
TabelaDeRepasse(int tamanho);
virtual ~TabelaDeRepasse();

virtual void mapear(int endereco, No* adjacente, int atraso);
virtual No** getAdjacentes();
virtual int getQuantidadeDeAdjacentes();

virtual void setPadrao(No* padrao, int atraso);

virtual No* getProximoSalto(int endereco, int& atraso);

virtual void imprimir();
```

O construtor recebe como parâmetro o tamanho da tabela, ou seja, o número máximo de endereços de destino, nós adjacentes e atrasos. Caso o tamanho informado seja menor ou igual a 0, o construtor deve jogar uma exceção do tipo `invalid_argument` da biblioteca padrão. No construtor, defina o nó padrão como `NULL` e seu atraso 0. O destrutor deve destruir os vetores alocados, mas não deve destruir os **Nos** adicionados ao vetor.

O método `mapear` deve associar o endereço passado como parâmetro ao **No** adjacente e ao atraso informados no método. Caso o endereço já esteja na tabela, não faça nada e jogue uma exceção do tipo `invalid_argument`, da biblioteca padrão. Jogue a exceção `overflow_error`, da biblioteca padrão, caso não seja possível fazer o mapeamento pois a tabela já contém o tamanho máximo de elementos (informado no construtor). Note que o método agora é `void` por causa das exceções.

Veja no enunciado do EP1 a explicação detalhada do funcionamento dos métodos `getAdjacentes`, `setPadrao` e `getProximoSalto` (lembrando-se de que agora eles trabalham com **Nos**).

[5] Veja explicações mais detalhadas em https://isocpp.org/wiki/faq/strange-inheritance#overload-derived ou https://www.programmerinterview.com/c-cplusplus/c-name-hiding/.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado. Por fim, note que agora não há mais a dependência circular, já que essa classe não depende mais de **Roteador**.

2.6 Classe Evento

O **Evento** representa o evento de recebimento de um **Datagrama** por um **No**, simulando assim o atraso de propagação. A única diferença para o EP1 é que ela agora usa um **No**, ao invés de **Roteador**. Dessa forma, essa classe deve possuir apenas os seguintes métodos **públicos**:

```
Evento(int instante, No* destino, Datagrama* d);
virtual ~Evento();

virtual int getInstante();
virtual No* getDestino();
virtual Datagrama* getDatagrama();

virtual void imprimir();
```

O construtor deve receber o instante em que o evento deve ser processado, o **No** destino e o **Datagrama**. Esses valores são retornados pelos método `getInstante`, `getDestino` e `getDatagrama`, respectivamente.

No destrutor não destrua o **No** e tampouco o **Datagrama**.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado. Note que a dependência circular ainda existe, mas com a classe **No** em vez de **Roteador**.

2.7 Classe No

O **No** representa elementos da rede que possuem um endereço e recebem **Datagramas**, gerenciando, portanto, uma fila de **Datagramas**. Um **No** pode ser um **Roteador** ou um **Hospedeiro**. Um **Roteador** é um **No** intermediário da rede que repassa os **Datagramas**, enquanto que um **Hospedeiro** é um **No** que executa **Chats**.

A classe **No** deve ser uma classe **abstrata**. Escolha o(s) método(s) mais adequado(s) para ser(em) **abstrato(s)**. Em relação ao EP1, esta classe absorveu alguns dos métodos do **Roteador**. A seguir são apresentados os métodos **públicos** dessa classe e a constante:

```
No(int endereco);
No(int endereco, Fila* fila);
virtual ~No();

virtual int getEndereco();

virtual Evento* processar(int instante);
virtual void receber(Datagrama* d);

virtual void imprimir();
static const int TAMANHO_FILA = 5;
```

O **No** possui dois construtores: um que recebe apenas o endereço do **No** e outro que recebe também a **Fila** que será usada (isso é necessário para o **RoteadorComQoS**). No construtor que recebe apenas o endereço, você deve criar uma **Fila** com `TAMANHO_FILA` de tamanho (mantenha essa constante com valor

5). No destrutor deve-se destruir a **Fila** associada ao **No** (ou seja, destrua, independe de qual construtor foi chamado).

O endereço informado no construtor deve ser retornado pelo método `getEndereco`.

O funcionamento do método `processar` depende do tipo de **No** e, por isso, ele será explicado nas classes **Roteador** e **Hospedeiro**.

O método `receber` deve adicionar o **Datagrama** recebido como parâmetro na **Fila** do **No**. Caso a fila estoure, não adicione o **Datagrama**, destrua-o e imprima a mensagem (pulando uma linha no final):

```
\tFila em <endereço> estourou
```

Onde `<endereço>` é o endereço do **No**. Note o '\t' (tab) para indentar o texto.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado. Note que para acompanhar o que está acontecendo no **No** e em seus subtipos devem ser feitas algumas impressões em tela (usando o `cout`).

2.8 Classe Roteador

O **Roteador** é um subtipo de **No** que faz o repasse de **Datagramas**. Comparando com o EP1, a responsabilidade de gerenciamento de **Datagramas** ficou na classe **No**. Além disso, o mapeamento agora é feito com **Nos** e se usa uma constante para definir o tamanho da **TabelaDeRepasse**. Com isso ela deve ter os seguintes membros públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
Roteador(int endereco);
virtual ~Roteador();

virtual void mapear(int endereco, No* adjacente, int atraso);
virtual void setPadrao(No* padrao, int atraso);

static const int TAMANHO_TABELA = 10;
```

O construtor deve receber o endereço do **Roteador**. Na criação de um **Roteador** você deve criar a **TabelaDeRepasse** com **TAMANHO_TABELA** de tamanho (mantenha a constante com valor 10). No destrutor deve-se destruir a **TabelaDeRepasse** criada.

Para permitir o funcionamento da classe **RoteadorComQoS**, crie um outro construtor, *protegido*[6]:

```
Roteador(int endereco, Fila* fila);
```

Sendo protegido, ele só poderá ser chamado pelas subclasses (no caso, **RoteadorComQoS**). Esse construtor deve usar o construtor de **No** que recebe o endereço e a fila.

Os métodos `mapear`, `setPadrao` e `processar` devem ter o mesmo comportamento especificado no EP1. Em relação às impressões do método `processar`, considere a nova impressão do **Datagrama** apresentada na Seção 2.10 para o caso do sem próximo; para o recebido pelo roteador, imprima o texto

[6] Um detalhe de *design*: o construtor de **No** não precisa ser protegido, já que somente as classes filhas podem usá-lo (como a classe é abstrata, ela não pode ser instanciada diretamente). Por outro lado, como **Roteador** é uma classe concreta, esse construtor é protegido para que somente as classes filhas possam usá-lo.

– e não o **Segmento**. No caso do método `mapear`, a única diferença é que ele não tem retorno (deixe a exceção ser propagada).

Note que essa classe não possui mais dependência circular, dado que **Evento** e **TabelaDeRepasse** não dependem mais de **Roteador** (elas dependem de **No**).

2.9 Classe RoteadorComQoS

O **RoteadorComQoS** é um subtipo de **Roteador** que prioriza pacotes para alguns destinos específicos (por simplicidade o destino é especificado apenas pelo endereço). Essa classe possui apenas dois métodos adicionais, um para priorizar um endereço e outro para ver a lista de endereços priorizados. A seguir são apresentados os métodos públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
RoteadorComQoS(int endereco);
virtual ~RoteadorComQoS();

virtual void priorizar(int destino);
virtual vector<int>* getDestinosPriorizados();
```

O construtor deve receber apenas um endereço e deve chamar o construtor protegido de **Roteador** (para isso, crie e passe uma **FilaComPrioridade** de tamanho `TAMANHO_FILA`). No destrutor destrua todos os atributos alocados dinamicamente por esta classe (note que a classe ancestral **No** é responsável por destruir a **Fila**).

O método `priorizar` deve adicionar o endereço como um destino priorizado. Não se preocupe com endereços repetidos. O método `getDestinosPriorizados` deve retornar um **vector** de inteiros (da biblioteca padrão – assunto da Aula 11) com os endereços informados pelo método `priorizar`.

No método `receber` (herdado), os **Datagramas** cujos destinos são priorizados devem ser adicionados à **FilaComPrioridade** com prioridade (`true` no parâmetro prioritário) – naturalmente, os que não estão na lista não devem ser priorizados. Não se esqueça que o método `receber` deve imprimir em tela uma mensagem caso a fila estoure (a mensagem é especificada na Seção 2.7).

2.10 Classe Hospedeiro

O **Hospedeiro** representa um dispositivo computacional, como um computador, um celular, uma Smart TV, etc. Essa classe é um subtipo de **No** que possui **Chats** que conversam com outros **Chats** através da rede. Ele deve ter os seguintes métodos públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
Hospedeiro(int endereco, Roteador* gateway, int atraso);
virtual ~Hospedeiro ();

virtual void adicionarChat(int porta, int enderecoDestino, int portaDestino);
virtual vector<Chat*>* getChats();
virtual Chat* getChat(int porta);
```

O construtor deve receber o endereço do **Hospedeiro**, o seu *gateway padrão* (também chamado de *roteador de borda padrão*, ou simplesmente de *gateway*) e o atraso de propagação para enviar

Datagramas ao *gateway*. O *gateway* é um **Roteador** que permite que as mensagens geradas pelos **Chats** do **Hospedeiro** sejam repassadas para a rede.

No destrutor deve-se destruir todos os objetos **Chat** criados, assim como o `vector` (assunto da Aula 11) que armazena os **Chats** (caso ele seja alocado dinamicamente).

O método `adicionarChat` deve criar um **Chat** na porta especificada e se comunicando com o **Chat** no endereço e porta destino. Caso já exista um **Chat** neste hospedeiro na porta informada, o método deve jogar uma exceção do tipo `logic_error`, da biblioteca padrão. O **Chat** criado deve ser adicionado ao `vector` de chats do **Hospedeiro** seguindo a ordem de adição (ou seja, o último **Chat** adicionado deve ficar na última posição do `vector`). Esse `vector` deve ser retornado pelo método `getChats`.

O método `getChat` deve retornar o **Chat** escutando na porta informada ou `NULL` caso não exista um **Chat** nessa porta.

O processamento do **Datagrama** por um **Hospedeiro** é diferente do feito por um **Roteador**. Caso a **Fila** esteja vazia, o método `processar` não deve fazer nada e retornar `NULL`. Caso contrário, esse método deve retirar 1 (e apenas 1) **Datagrama** da **Fila** e fazer o seguinte:

1. Caso o destino do **Datagrama** seja o endereço deste **Hospedeiro**, deve-se procurar o **Chat** que está escutando a porta de destino indicada no **Segmento** do **Datagrama**:
    a. Caso haja um **Chat** na porta de destino, o método `receber` do **Chat** deve ser chamado passando o **Datagrama**.
    b. Caso não haja um **Chat**, deve-se apenas imprimir uma mensagem (especificada adiante) e destruir o **Datagrama**.

    O método deve retornar NULL.
2. Caso o destino do **Datagrama** seja um outro endereço, deve-se criar um **Evento** cujo instante é o instante passado como parâmetro de `processar` somado ao atraso do *gateway*, e cujo destino é o *gateway*. Esse **Evento** deve ser então retornado pelo método.

Por exemplo, suponha que o **Hospedeiro** com endereço 20, *gateway* r3 e atraso 2, tem um **Chat** na porta 80. Também suponha que ele recebeu (pelo método `receber`) um **Datagrama** `{origem=1, portaOrigem=50, destino=20, portaDestino=80, dado="Alo"}` e depois um outro **Datagrama** `{origem=20, portaOrigem=80, destino=2, portaDestino=5, dado="Oi"}`. Considere que a primeira chamada do método `processar` ocorre no instante 5. Ela deve retirar o **Datagrama** `{origem=1, portaOrigem=50, destino=20, portaDestino=80, dado="Alo"}` da **Fila**. Como o destino deste **Datagrama** é o endereço do **Hospedeiro** e como existe um **Chat** na porta 80, deve-se repassar o **Datagrama** para o **Chat** e retornar NULL. Na próxima chamada do método `processar`, no instante 6, deve-se retirar da fila o **Datagrama** `{origem=20, portaOrigem=80, destino=2, portaDestino=5, dado="Oi"}`. Como o endereço de destino é um outro **No**, deve-se retornar um novo **Evento** com instante 6 + 2 = 8, destino o **Roteador** r3 e esse **Datagrama**.

Para acompanhar o que está acontecendo no **Hospedeiro** devem ser feitas algumas impressões em tela (usando o `cout`) durante o método `processar`:

- Não deve ser apresentada a informação do processamento caso o **Hospedeiro** não tenha **Datagramas** em sua **Fila**. Caso ele possua **Datagramas**, deve ser impresso, pulando uma linha no final:

```
Hospedeiro <e>
```

Onde `<e>` é o endereço do **Hospedeiro** como, por exemplo:

```
Hospedeiro 20
```

Além disso, deve ser impresso o resultado do processamento do **Datagrama** pelo **No** da seguinte forma (pule uma linha após cada impressão):

- Caso o **Datagrama** retirado da **Fila** tenha um **Chat** como destinatário:

  ```
  \tMensagem recebida
  <texto completo do Chat>
  ```

  Onde `<texto completo do Chat>` é o retorno do método `getTextoCompleto` do **Chat** **<u>após</u>** a chamada do método `receber` (ou seja, considerando o **Datagrama** recebido). O conteúdo retornado pelo método `getTextoCompleto` é especificado na Seção 2.11.

- Caso o **Datagrama** não tenha um **Chat** como destinatário:

  ```
  \tSem chat: <datagrama>
  ```

  Onde `<datagrama>` são as informações do **Datagrama**. Por exemplo:

  ```
  \tSem chat: Origem: 1:20, Destino: 20:80, Algo
  ```

- Caso o **Datagrama** retirado da **Fila** seja repassado ao *gateway*:

  ```
  \tMensagem enviada
  ```

- Em relação a `<datagrama>`, ele deve possuir o seguinte formato:

  ```
  Origem: <origem>:<pOrigem>, Destino: <destino>:<pDestino>, <d>
  ```

  Onde:

  - `<origem>`: é o endereço de origem no **Datagrama**;
  - `<pOrigem>`: é a porta de origem no **Segmento** do **Datagrama**;
  - `<destino>`: é o endereço de destino no **Datagrama**;
  - `<pDestino>`: é ao porta de destino no **Segmento** do **Datagrama**;
  - `<d>`: é o dado no **Segmento** do **Datagrama**.

  Por exemplo, a impressão do **Datagrama** `{origem=1, portaOrigem=20, destino=20, portaDestino=80, dado="Algo"}` seria:

  ```
  Origem: 1:20, Destino: 20:80, Algo
  ```

Não faça outras impressões nesse método, pois isso pode afetar a correção.

Por fim, note que essa classe possui uma dependência circular com a classe **Chat**.

2.11 Classe Chat

O **Chat** representa um processo executando em um **Hospedeiro** que permite a troca de mensagens com um outro **Chat**, localizado (potencialmente) em um outro **Hospedeiro** da **Rede**. **Chats** enviam **Datagramas** para o **Hospedeiro** e recebem **Datagramas** do **Hospedeiro**, armazenando os textos enviados e recebidos. Essa classe deve possuir apenas os seguintes métodos **públicos**:

```
Chat(Hospedeiro* h, int porta, int enderecoDestino, int portaDestino);
virtual ~Chat();

virtual int getPorta();
virtual void enviar(string texto);
virtual void receber(Datagrama* d);
virtual string getTextoCompleto();

virtual void imprimir();
```

O construtor deve receber o **Hospedeiro** que envia e recebe **Datagramas** do **Chat**, a porta que o **Chat** escuta e o endereço e porta do **Chat** de destino. O texto completo do **Chat** deve ser iniciado com "" (`string` vazia). Caso o **Hospedeiro** seja NULL, o método deve jogar a exceção `invalid_argument`, da biblioteca padrão.

A porta que o **Chat** *escuta* (ou seja, a porta dele, informada no construtor) é retornada pelo método `getPorta`.

O método `enviar` deve chamar o método `receber` do **Hospedeiro** passando um **Datagrama** criado com o endereço do **Hospedeiro** e porta do **Chat** como origem, e o endereço e porta destino (informados no construtor) como destino. O dado do **Segmento** deve ser a `string` recebida como parâmetro pelo método `enviar`. Este método deve também concatenar ao texto completo do **Chat** o seguinte texto (pulando uma linha no final usando o `\n`):

```
\t\tEnviado: <texto>
```

Onde `<texto>` é a `string` recebida como parâmetro pelo método `enviar`.

O método `receber` deve concatenar ao texto completo do **Chat** o seguinte texto (pulando uma linha no final usando o `\n`):

```
\t\tRecebido: <texto>
```

Onde `<texto>` é a `string` que está no dado do Segmento do **Datagrama** recebido como parâmetro pelo método `receber`. Por simplicidade não é necessário verificar se o destino do **Datagrama** está correto. Após usar o **Datagrama**, destrua-o.

O método `getTextoCompleto` deve retornar a `string` com o texto completo enviado e recebido pelo **Chat**. Inicialmente esse texto deve ser vazio (""). Por exemplo, suponha que o **Chat** recebeu uma mensagem "Ola". Ao chamar o `getTextoCompleto`, deve ser retornada a `string`:

```
\t\tRecebido: Ola\n
```

Se em seguida o **Chat** enviar o texto "Oi", o método `getTextoCompleto` deve retornar:

```
\t\tRecebido: Ola\n\t\tEnviado: Oi\n
```

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado.

2.12 Classe Rede

A **Rede** é a classe responsável por ter a lista de **Roteadores.** Ela sofreu algumas alterações no EP2. A principal delas é que a **Rede** agora é composta por **Nos** – e não simplesmente por **Roteadores**. Além disso, os **Nos** devem ser armazenados em um `list` (da biblioteca padrão, assunto da Aula 11). Também

foi adicionado um método que retorna um `list` com os **Hospedeiros**. Dessa forma, esta classe deve possuir apenas os seguintes métodos **públicos**:

```
Rede();
virtual ~Rede();

virtual void adicionar(No* n);
virtual No* getNo(int endereco);
virtual list<No*>* getNos();
virtual list<Hospedeiro*>* getHospedeiros();

virtual void imprimir();
```

O construtor da **Rede** não recebe parâmetros. No destrutor *destrua* todos os nós adicionados à **Rede**, assim como a estrutura auxiliar criada para armazená-los.

O método `adicionar` deve adicionar o **No** à **Rede**. Caso já exista um **No** na **Rede** com o mesmo endereço, jogue um `logic_error` (da biblioteca padrão) – e não adicione o **No**.

O método `getNo` deve retornar o **No** (dentre os adicionados) que possui o endereço informado. Caso não haja um **No** com esse endereço, este método deve retornar `NULL`.

O método `getNos` deve retornar um `list` (apresentado na Aula 11) com todos os **Nos** (**Roteadores** e **Hospedeiros**) adicionados à **Rede** (retorne-os na ordem em que eles foram adicionados). Caso a **Rede** não possua **Nos**, o `list` retornado deve estar vazio. O método `getHospedeiros` deve retornar um `list` que contém apenas os **Hospedeiros** existentes na **Rede**. Para fazer isso, a recomendação é criar um novo **list** cada vez que o método for chamado. Caso a **Rede** não possua **Hospedeiros**, o `list` retornado deve ser vazio.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado.

2.13 Classe Agendador

O **Agendador** cuida da "passagem do tempo" no nosso simulador. Ela deve possuir o mesmo comportamento especificado no EP1. A única diferença é que o método `agendar` agora recebe um **No** em vez de um **Roteador**. Com isso, essa classe possui os seguintes métodos públicos:

```
Agendador(int instanteInicial, Rede* rede, int tamanho);
virtual ~Agendador();

virtual bool agendar(int instante, No* n, Datagrama* d);
virtual void processar();
virtual int getInstante();
```

Veja a especificação desses métodos no enunciado do EP1 (evitou-se fazer grandes alterações nessa classe). Apesar de ela não ter mudança no comportamento, o código deverá ser corrigido para usar a nova interface (membros públicos) da **Rede**.

3 Persistência

O software permitirá carregar **Redes** salvas em arquivos. Para isso deve ser implementada a classe **PersistenciaDeRede**. A seguir é apresentado o formato do arquivo, um exemplo de arquivo e a especificação da classe.

3.1 Formato do arquivo

A persistência da **Rede** deve seguir o formato de arquivo apresentado a seguir. Entre "<" e ">" são especificados os valores esperados. Por simplicidade, será utilizado um caractere de nova linha ('\n') ou espaço (' ') como delimitador (pode ser qualquer um deles).

```
<quantidade de roteadores>
<roteador 1>
<roteador 2>
...
<quantidade de hospedeiros>
<hospedeiro 1>
<hospedeiro 2>
...
<tabela de repasse do roteador 1>
<tabela de repasse do roteador 2>
...
```

Note que **há uma linha em branco no final do arquivo**.

O formato do roteador deve seguir o seguinte padrão:

- Se for um **Roteador**:
  `r <endereço>`
- Se for um **RoteadorComQoS**:
  `q <endereço> <quantidade de destinos priorizados> <destino1> <destino2> ...`

O formato do **Hospedeiro** deve seguir o seguinte padrão:

```
<endereço> <gateway> <atraso> <quantidade de chats> <chat1> <chat2> ...
```

O formato do **Chat** deve ser o seguinte:

```
<porta> <endereço de destino> <porta de destino>
```

Por fim, a tabela de repasse deve ter o seguinte formato:

```
<roteador> <roteador padrão> <atraso padrão> <quantidade de mapeamentos>
<mapeamento 1>
<mapeamento 2>
...
```

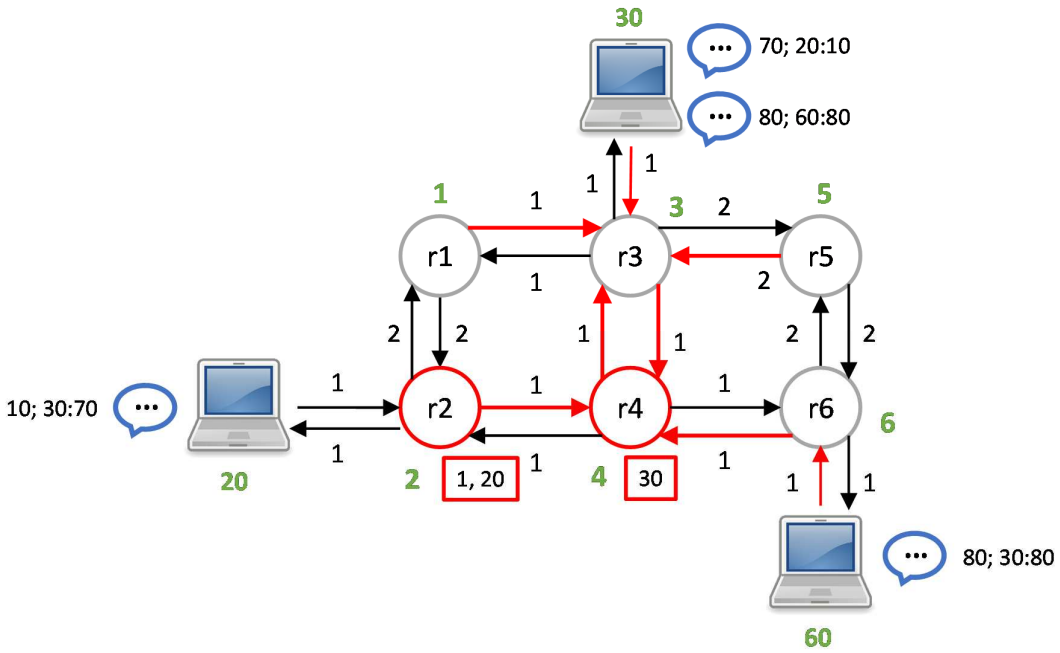
Note que, por simplicidade, todo **Roteador** possui um **Roteador** padrão. O **Roteador** padrão não deve ser representado no mapeamento (todos os outros **Roteadores** e **Hospedeiros** devem ser considerados).

O mapeamento deve ser simplesmente:

```
<endereço do destino> <endereço do nó adjacente> <atraso>
```

Note que há uma linha vazia no final do arquivo.

Por exemplo, considere a **Rede** a seguir, composta por 6 roteadores (os círculos; em cinza são objetos **Roteador** normais e os em vermelho são objetos **RoteadorComQoS**) e 3 hospedeiros (os notebooks da figura). Os endereços dos nós são apresentados em verde; nos quadros em vermelho são apresentados os destinos priorizados pelos objetos **RoteadorComQoS** (por exemplo r2 prioriza os endereços 1 e 20). Os mapeamentos são apresentados como arestas (setas) e o atraso é o peso. As setas em vermelho são os roteadores padrão.



Nos hospedeiros, o **Chat** é apresentado pela "<porta>;<destino>:<porta do destino>". Por exemplo, o **Hospedeiro** com endereço 20 possui um **Chat** na porta 10 com o **Hospedeiro** de endereço 30, na porta 70 dele.

A tabela de repasse dos roteadores é apresentada a seguir, usando o formato *nó/atraso*. Ou seja, o roteador r1 tem o roteador r3 como roteador padrão com atraso 1 e repassa ao roteador r2 datagramas direcionados ao endereço 2 com atraso 2. Da mesma forma, o roteador r2 tem o roteador r4 como padrão com atraso 1 e repassa ao roteador 1 datagramas direcionados ao endereço 1 com atraso 2.

	Endereço									
Roteador	**Padrão**	**1**	**2**	**3**	**4**	**5**	**6**	**20**	**30**	**60**
r1	r3/1		r2/2							
r2	r4/1	r1/2						20/1		
r3	r4/1	r1/1				r5/2			30/1	
r4	r3/1		r2/1				r6/1	r2/1		r6/1
r5	r3/2						r6/2			
r6	r4/1					r5/2				60/1

A seguir é apresentado o arquivo `exemplo.txt` (fornecido) com esse exemplo:

```
6
r 1
q 2 2 1 20
r 3
q 4 1 30
r 5
r 6
3
20 2 1 1
10 30 70
30 3 1 2
70 20 10
80 60 80
60 6 1 1
80 30 80
1 3 1 1
2 2 2
2 4 1 2
1 1 2
20 20 1
3 4 1 3
1 1 1
5 5 2
30 30 1
4 3 1 4
2 2 1
6 6 1
20 2 1
60 6 1
5 3 2 1
6 6 2
6 4 1 2
5 5 2
60 60 1
```

3.2 Classe PersistenciaDeRede

A classe **PersistenciaDeRede** é a classe responsável por carregar a **Rede** de um arquivo texto. Os únicos métodos públicos que a classe deve possuir são:

```
PersistenciaDeRede(string arquivo);
virtual ~PersistenciaDeRede();

virtual Rede* carregar();
```

O construtor deve `receber` o arquivo a ser lido. Esse arquivo deve ser usado pelo método `carregar` para criar uma nova **Rede** com a configuração persistida e retorná-la. Caso o arquivo não exista ou caso haja algum problema de leitura (erro de formato ou outro problema), jogue uma exceção do tipo `logic_error`.

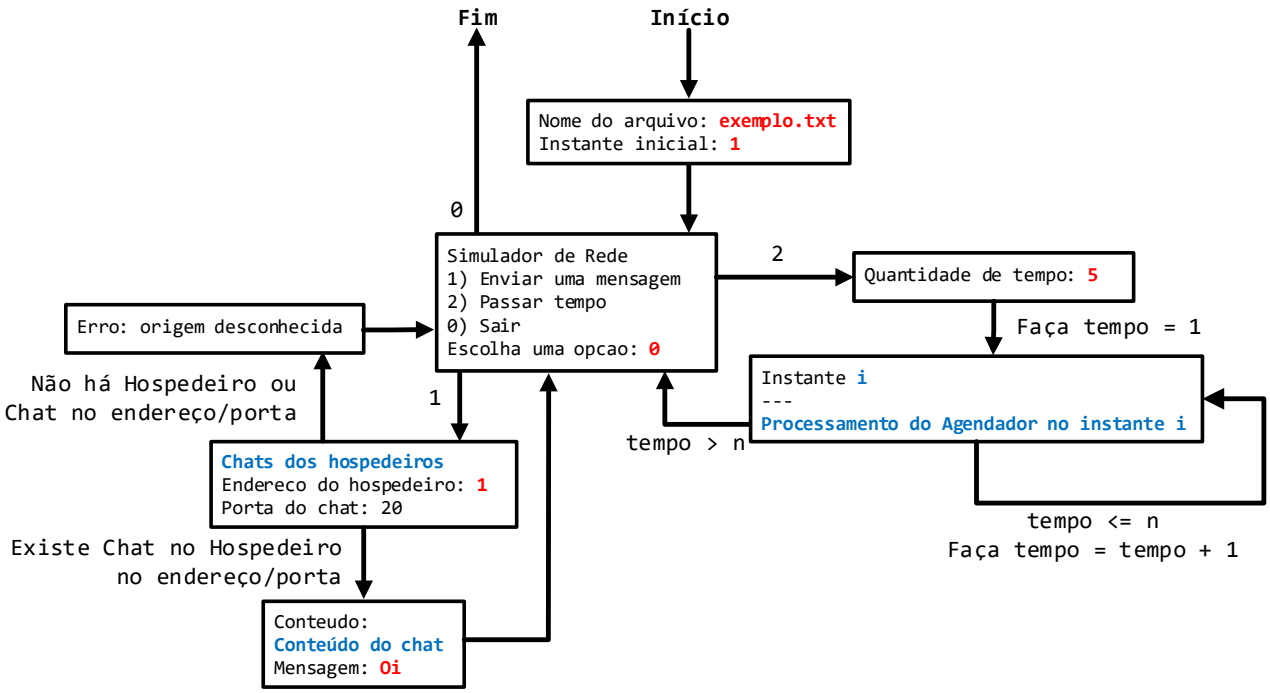
Ao carregar não é preciso verificar se o **Chat** possui um par.

4 Main

Coloque a `main` em um arquivo separado, chamado `main.cpp`.

4.1 Interface

O `main` deve possuir uma interface em console que permite carregar uma **Rede** de um arquivo, enviar mensagens de um **Chat** para outro, além de simular a passagem de tempo. Essa interface é apresentada esquematicamente no diagrama a seguir. Cada retângulo representa uma "tela", ou seja, um conjunto de texto impresso e solicitado ao usuário. As setas representam as transições de uma tela para outra – os textos na seta representam o valor que deve ser digitado para ir para a tela destino ou a condição necessária (quando não há um texto é porque a transição acontece incondicionalmente). Quando a transição apresenta "Faça", considere que é um comando a ser executado. Em **vermelho** são apresentados exemplos de dados inseridos pelo usuário; em **Azul** são as informações que dependem do contexto.



- Ao iniciar o programa, carregue uma rede a partir do arquivo especificado pelo usuário. Em seguida, crie um **Agendador** com o instante inicial informado pelo usuário. A fila do **Agendador** deve permitir no máximo 10 **Eventos**.
- A opção 1 ("Enviar uma mensagem") deve enviar um texto a partir de um **Chat**.
    - A impressão dos "Chats dos hospedeiros" deve ser no formato:

```
Hospedeiro: <endereço 1>
\tChat: <porta1>
\tChat: <porta2>
...
Hospedeiro: <endereço 2>
\tChat: <porta1>
\tChat: <porta2>
...
```

Onde \t é uma tabulação. Por exemplo:

```
Hospedeiro: 20
```

```
\tChat: 10
Hospedeiro: 30
\tChat: 70
\tChat: 80
```

  - A impressão do "Conteúdo do chat" deve imprimir o retorno do método `getTextoCompleto` do **Chat** (descrito na Seção 2.11).
- A opção 2 ("Passar tempo") deve chamar o método `processar` do **Agendador** na quantidade de vezes que for informada como tempo. O instante i apresentado deve ser o instante indicado pelo método `getInstante` do **Agendador**.
  - Na chamada do método `processar` do **Agendador** é chamado o método `processar` dos **Nós**. Essas chamadas fazem impressões (veja as Seções 2.8, 2.9 e 2.10). Isso é representado no diagrama por "*Processamento do Agendador no instante i*".
- No diagrama não são apresentados casos de erro (por exemplo, a digitação de uma opção de menu inválida ou de um texto em um valor que deveria ser um número). **Não é necessário fazer o tratamento disso**. Assuma que o usuário *sempre* digitará um valor correto. Em caso de exceções, capture-as e apresente a mensagem de erro e então encerre o programa.
- Por simplicidade considere que o texto da mensagem não possui espaços.

> **Atenção:** A interface com o usuário deve seguir exatamente a ordem definida (e exemplificada). Se a ordem não for seguida, haverá desconto de nota. Não adicione outros textos além dos apresentados no diagrama e especificados.

4.2 Exemplo

Segue um exemplo de funcionamento do programa com a saída esperada e ressaltando em **vermelho** os dados digitados pelo usuário. O arquivo `exemplo.txt` contém a rede de exemplo mostrada na seção 3.1.

```
Nome do arquivo: exemplo.txt
Instante inicial: 1

Simulador de Rede
1) Enviar uma mensagem
2) Passar tempo
0) Sair
Escolha uma opcao: 1

Hospedeiro: 20
        Chat: 10
Hospedeiro: 30
        Chat: 70
        Chat: 80
Hospedeiro: 60
        Chat: 80
Endereco do hospedeiro: 30
Porta do chat: 70
Conteudo:

Mensagem: Msg1

Simulador de Rede
1) Enviar uma mensagem
2) Passar tempo
0) Sair
Escolha uma opcao: 1
```

```
Hospedeiro: 20
        Chat: 10
Hospedeiro: 30
        Chat: 70
        Chat: 80
Hospedeiro: 60
        Chat: 80
Endereco do hospedeiro: 20
Porta do chat: 10
Conteudo:

Mensagem: Msg2

Simulador de Rede
1) Enviar uma mensagem
2) Passar tempo
0) Sair
Escolha uma opcao: 2

Quantidade de tempo: 1

Instante 1
---
Hospedeiro 20
        Mensagem enviada
Hospedeiro 30
        Mensagem enviada

Simulador de Rede
1) Enviar uma mensagem
2) Passar tempo
0) Sair
Escolha uma opcao: 1

Hospedeiro: 20
        Chat: 10
Hospedeiro: 30
        Chat: 70
        Chat: 80
Hospedeiro: 60
        Chat: 80
Endereco do hospedeiro: 20
Porta do chat: 10
Conteudo:
                Enviado: Msg2

Mensagem: Msg3

Simulador de Rede
1) Enviar uma mensagem
2) Passar tempo
0) Sair
Escolha uma opcao: 2

Quantidade de tempo: 6

Instante 2
---
Roteador 2
        Repassado para 4 (instante 3): Origem: 20:10, Destino: 30:70, Msg2
Roteador 3
        Repassado para 4 (instante 3): Origem: 30:70, Destino: 20:10, Msg1
Hospedeiro 20
        Mensagem enviada

Instante 3
---
Roteador 2
        Repassado para 4 (instante 4): Origem: 20:10, Destino: 30:70, Msg3
Roteador 4
        Repassado para 3 (instante 4): Origem: 20:10, Destino: 30:70, Msg2

Instante 4
```

```
---
Roteador 3
        Repassado para 30 (instante 5): Origem: 20:10, Destino: 30:70, Msg2
Roteador 4
        Repassado para 3 (instante 5): Origem: 20:10, Destino: 30:70, Msg3

Instante 5
---
Roteador 3
        Repassado para 30 (instante 6): Origem: 20:10, Destino: 30:70, Msg3
Roteador 4
        Repassado para 2 (instante 6): Origem: 30:70, Destino: 20:10, Msg1
Hospedeiro 30
        Mensagem recebida
                Enviado: Msg1
                Recebido: Msg2

Instante 6
---
Roteador 2
        Repassado para 20 (instante 7): Origem: 30:70, Destino: 20:10, Msg1
Hospedeiro 30
        Mensagem recebida
                Enviado: Msg1
                Recebido: Msg2
                Recebido: Msg3

Instante 7
---
Hospedeiro 20
        Mensagem recebida
                Enviado: Msg2
                Enviado: Msg3
                Recebido: Msg1

Simulador de Rede
1) Enviar uma mensagem
2) Passar tempo
0) Sair
Escolha uma opcao: 0
```

5 Entrega

O projeto deverá ser entregue até dia **07/12** no Judge em <https://laboo.pcs.usp.br/ep/>.

Atenção

- Deve ser mantida a mesma dupla do EP1. É possível apenas *desfazer* a dupla. Com isso, cada aluno deve fazer uma entrega diferente (e em separado). Caso você deseje fazer isso, envie um e-mail para levy.siqueira@usp.br até dia **29/11** informando os números USP dos alunos e mandando-o com cópia para a sua dupla.
- Não copie código de um outro grupo. Qualquer tipo de cópia será considerado plágio e os grupos envolvidos terão **nota 0 no EP**. Portanto, **não envie** o seu código para um colega de outro grupo!

Entregue todos os arquivos, inclusive o `main` (que deve **obrigatoriamente** ficar em um arquivo "main.cpp"), em um arquivo comprimido no formato ZIP (outros formatos, como RAR e 7Z, *podem* não ser reconhecidos e acarretar **nota 0**). Os códigos fonte não devem ser colocados em pastas. A submissão pode ser feita por qualquer um dos membros da dupla – recomenda-se que os dois submetam.

Atenção: faça a submissão do mesmo arquivo nos **3 problemas** (Parte 1, Parte 2 e Parte 3). Isso é necessário por uma limitação do Judge. Caso isso não seja feito, parte do seu EP não será corrigido – impactando a nota.

Siga a convenção de nomes para os arquivos ".h" e ".cpp". O não atendimento disso pode levar a erros de compilação (e, consequentemente, **nota zero**).

Ao submeter os arquivos no Judge, será feita **apenas** uma verificação básica de modo a evitar erros de compilação devidos a erros de digitação no nome das classes e dos métodos públicos. **Note que a nota dada não é a nota final**: nesse momento não são executados testes – o Judge apenas tenta chamar todos os métodos definidos neste documento para todas as classes.

Você pode submeter quantas vezes quiser, sem desconto na nota.

Ao fim do prazo serão executados os seguintes testes:

Parte 1:
- Chat nao tem classe base
- Chat construtor getPorta e destrutor
- Chat sem hospedeiro
- Chat textoCompleto so um enviar
- Chat textoCompleto so com enviar
- Chat textoCompleto so com receber
- Chat textoCompleto enviar e receber variado1
- Chat textoCompleto enviar e receber variado2
- Datagrama construtor getters e destrutor
- Evento construtor e destrutor
- Evento getters
- Fila construtor e destrutor
- Fila construtor tamanho invalido
- Fila enqueue e isEmpty
- Fila enqueue e getSize
- Fila enqueue e overflow
- Fila dequeue e underflow
- Fila enqueue intercalado com dequeue ate tamanho
- Fila enqueue e dequeue mais que tamanho datagramas no total
- Fila overflow underflow e mais que tamanho datagramas no total
- FilaComPrioridade e filha de Fila
- FilaComPrioridade construtor e destrutor
- FilaComPrioridade construtor tamanho invalido
- FilaComPrioridade enqueue e isEmpty
- FilaComPrioridade enqueue e getSize
- FilaComPrioridade enqueue e overflow
- FilaComPrioridade dequeue e underflow
- FilaComPrioridade enqueue com um parametro
- FilaComPrioridade prioridade misturada
- FilaComPrioridade prioridade misturada mais que tamanho datagramas no total
- FilaComPrioridade prioridade misturada e overflow underflow
- Rede construtor destrutor
- Rede adicionar e getNos
- Rede adicionar ja possui roteador com endereco
- Rede getNo
- Rede getNo enderecos invalidos
- Rede getHospedeiros com so nos hospedeiros
- Rede getHospedeiros misto 1

Parte 2:
- Hospedeiro construtor getEndereco e destrutor
- Hospedeiro adicionarChat e getChats so um chat
- Hospedeiro adicionarChat e getChats varios chats
- Hospedeiro adicionarChat porta ja ocupada
- Hospedeiro adicionarChat e getChat
- Hospedeiro adicionarChat e getChat nao encontrado
- Hospedeiro receber estourou
- Hospedeiro processar vazio
- Hospedeiro processar chat destino
- Hospedeiro processar chat destino com varias mensagens
- Hospedeiro processar sem chat destino
- Hospedeiro processar repassar gateway
- Hospedeiro processar variado1
- Hospedeiro processar variado2
- No eh abstrata
- No nao tem classe base
- No Roteadores e Hospedeiro sao filhas
- Roteador construtor getEndereco e destrutor
- Roteador receber estourou
- Roteador processar vazio
- Roteador receber e processar destino
- Roteador receber e processar sem proximo
- Roteador receber e processar repassando
- Roteador receber e processar variado1
- Roteador receber e processar variado2
- RoteadorComQoS construtor getEndereco e destrutor
- RoteadorComQoS priorizar e getDestinosPriorizados um destino
- RoteadorComQoS priorizar e getDestinosPriorizados varios destinos
- RoteadorComQoS receber estourou
- RoteadorComQoS processar vazio
- RoteadorComQoS receber e processar destino
- RoteadorComQoS receber e processar sem proximo
- RoteadorComQoS receber e processar repassando com prioridade
- RoteadorComQoS receber e processar variado

Parte 3:
- Agendador: Construtor Destrutor e getInstante
- Agendador: processar sem evento e 1 evento no inicio

- Rede getHospedeiros misto 2
- Rede getHospedeiros misto alterando os nos
- Segmento construtor getters e destrutor
- TabelaDeRepasse construtor e destrutor
- TabelaDeRepasse construtor tamanho invalido
- TabelaDeRepasse mapear e getAdjacentes
- TabelaDeRepasse mapear com endereco igual
- TabelaDeRepasse mapear com overflow
- TabelaDeRepasse setPadrao e getAdjacentes
- TabelaDeRepasse getProximoSalto
- TabelaDeRepasse getProximoSalto e substituicao
- TabelaDeRepasse getProximoSalto e padrao
- TabelaDeRepasse getProximoSalto e padrao NULL
- Agendador: processar varios eventos mesmo instante
- Agendador: processar eventos variados
- PersistenciaDeRede arquivo inexistente
- PersistenciaDeRede erro formato Roteador
- PersistenciaDeRede erro formato Hospedeiro ou Chat
- PersistenciaDeRede erro TabelaDeRepasse
- PersistenciaDeRede 2 Roteadores
- PersistenciaDeRede 1 Roteador 1 RoteadorComQoS e 1 Hospedeiro
- PersistenciaDeRede Enunciado
- PersistenciaDeRede Rede Variada1
- PersistenciaDeRede Rede Variada2

6 Dicas

- Implemente a solução aos poucos – não deixe para implementar tudo no final. As classes **Hospedeiro**, **Rede** e **RoteadorComQoS** usam conceitos da Aula 11 – você pode deixá-las para o final ou mesmo deixar os métodos que usam `list` e `vector` para depois.
- O Code::Blocks tem um commando bastante útil para implementar os arquivos cpp. Ao clicar com o botão direito, vá em "insert/refactor"->"All class methods without implementation..." para que ele crie um esqueleto de todos os métodos da classe. Mas cuidado: esse comando não funciona corretamente com templates! Ele não coloca o tipo entre "<" e ">".
- Caso o programa esteja travando, execute o programa no modo de depuração do Code::Blocks. O Code::Blocks mostrará a pilha de execução do programa no momento do travamento, o que é bastante útil para descobrir onde o erro acontece!
- Faça `#include` apenas das classes que são usadas naquele arquivo. Por exemplo, se o arquivo .h não usa a classe **X**, mas o .cpp usa essa classe, faça o `include` da classe **X** *apenas* no .cpp. Incluir classes desnecessariamente pode gerar erros de compilação (por causa de referências circulares).
    - Inclua todas as dependências necessárias. Não dependa de `#includes` feitos por outros arquivos incluídos.
- É muito trabalhoso testar o programa ao executar o `main` *com menus*, já que é necessário informar vários dados. Para testar o programa faça o `main` chamar uma função de teste que cria objetos com valores interessantes para testar, sem pedir entrada para o usuário. Não se esqueça de remover a função de teste ao entregar a versão final do EP.
- O método `imprimir` é útil para testes, mas não é obrigatório implementar um comportamento para ele. Por exemplo, se você não quiser implementar esse método para a classe **Rede** você pode fazer no .cpp simplesmente:

```
void Rede::imprimir() {
}
```

    - O mesmo pode ser feito em métodos (ou mesmo classes) que você não conseguiu implementar. Crie implementações vazias (ou que apenas retornam um valor específico) para não ter erro de compilação.
- Separe o `main` em várias funções para reaproveitar código. Planeje isso!

- Submeta no Judge o código com antecedência para descobrir problemas na sua implementação. É normal acontecerem *RuntimeErrors* e outros tipos de erros no Judge que não aparecem no Code::Blocks (especialmente nas versões antigas do Code::Blocks que usam um outro compilador). Veja a mensagem de erro do Judge para descobrir o problema. Caso você queira testar o projeto em um compilador similar ao do Code::Blocks, use o site https://repl.it/ (note que ele não tem depurador *ainda*).
    - Em geral *RuntimeErrors* acontecem porque você não inicializou um atributo que é usado. Por exemplo, caso você não crie um vetor ou não inicialize o atributo quantidade, para controlar o tamanho do vetor, ocorrerá um *RuntimeError*.
- Use o canal #duvidas-gerais para esclarecer dúvidas no enunciado ou problemas de submissão no Judge.
- **Evite submeter nos últimos minutos do prazo de entrega. É normal o Judge ficar sobrecarregado com várias submissões e demorar para compilar.**