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 1 – 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).

1 Introdução

A Internet é uma rede que trabalha com *comutação de pacotes* ao invés de *comutação de circuito*. Na *comutação de circuito* é feita uma reserva de um canal de comunicação entre a origem e o destino. Por exemplo, imagine a rede apresentada na Figura 1. Ela é composta por diversos nós (elementos da rede, representados pelos círculos cinzas e pelos computadores), os quais possuem diversos canais de comunicação com outros nós (representados pelas linhas). Para que o computador A converse com o computador B é necessário reservar um canal de comunicação passando por diversos nós (uma opção é 1, 2 e 4), como é exemplificado na figura pelos canais em vermelho.

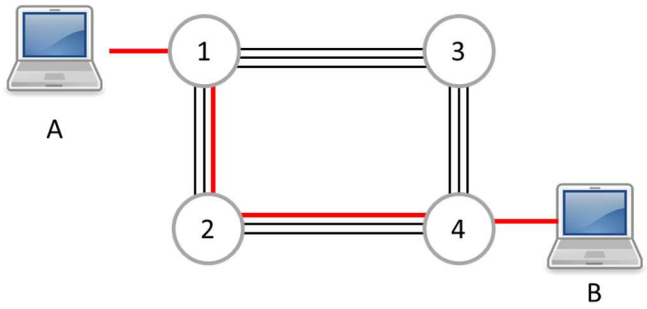


Figura 1: transmissão de mensagens em uma transmissão por comutação de circuitos.

A comutação de circuitos permite uma comunicação com taxa de transferência constante, já que um canal é fisicamente reservado. Mas ela tem algumas desvantagens: ela limita o número de usuários ativos (limitado pelo número de canais de comunicação, no exemplo 3 canais entre cada nó) e desperdiça infraestrutura caso os computadores tenham períodos de silêncio durante a comunicação (por exemplo, caso um computador espere uma ação do usuário para mandar uma nova mensagem).

Na *comutação de pacotes* as mensagens são organizadas em pacotes e cada nó direciona o pacote para um outro nó até que ele chegue ao destino. Ou seja, os pacotes são *repassados* de nó a nó até se chegar ao destino. Com isso, os canais de comunicação não são reservados e podem ser reutilizados por vários computadores. Por exemplo, na Figura 2 o computador A quer transmitir uma mensagem para o computador B. As mensagens são organizadas em pacotes (representados como retângulos vermelhos),

os quais são transmitidos pelos nós r1, r2 e r4 (representada pela linha tracejada vermelha). Caso o computador X queira transmitir uma mensagem para Y no mesmo instante, os pacotes (representados como retângulos verdes) poderiam passar pelos nós r2, r4 e r3 até chegar ao destino, usando alguns dos mesmos canais de comunicação usados por A e B.

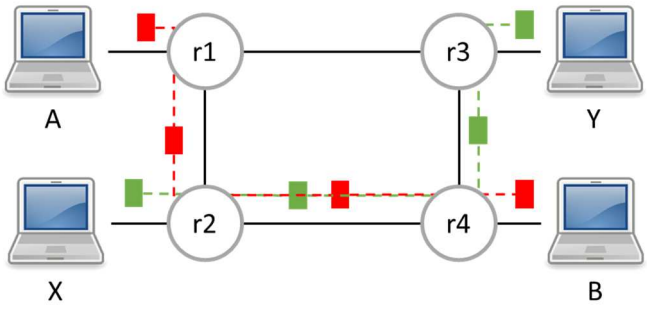


Figura 2: transmissão de pacotes entre computadores em uma transmissão por comutação de pacotes.

1.1 Funcionamento

Neste projeto simularemos a Internet de forma bem simplificada, trabalhando, portanto, com a comutação de pacotes. Quem quiser ver com detalhes como a Internet realmente funciona pode consultar o livro do Kurose e Ross[1]. Note que esse assunto também será tratado por *PTC3360 - Introdução a Redes e Comunicações*, que é uma disciplina do 3º ano de Engenharia Elétrica.

O foco neste primeiro EP é na troca de pacotes entre nós intermediários da rede – os círculos cinza da Figura 2. Na Internet esses nós são chamados de *roteadores*. Esses dispositivos simplesmente encaminham para um outro nó o pacote recebido, até que o pacote chegue ao seu destinatário. Para que eles consigam conversar, é necessário haver um padrão que define, entre outros detalhes, o formato do pacote e as ações que devem ser tomadas quando se recebe um pacote. Esse padrão é definido através de um *protocolo*. Por exemplo, em um protocolo pode-se definir que o pacote deve conter o endereço da origem, o endereço do destino e um dado. Na Internet, o principal protocolo nesta camada[2] é o protocolo IP (*Internet Protocol*) e nele o endereço dos nós é indicado pelo *endereço IP*. Neste simulador, por simplicidade, o endereço será um número inteiro.

Os pacotes recebidos por um roteador são colocados em uma fila[3] para que eles sejam processados. Isso é necessário pois o roteador pode receber vários pacotes ao mesmo tempo e, além disso, o processamento de um pacote não é imediato. Dessa forma, o roteador tira um pacote da fila, o processa, e o repassa. Esse repasse deve ser feito a um dos seus nós adjacentes, apesar de o roteador também receber pacotes cujos destinatários são nós mais distantes. A indicação de qual dos nós adjacentes o roteador deve repassar um pacote está em uma *tabela de repasse*[4], interna ao roteador. Essa tabela basicamente mapeia endereços a

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

[2] A arquitetura de uma rede é tipicamente organizada em várias camadas. Neste primeiro EP simularemos apenas uma dessas camadas, que é chamada de *camada de rede*.

[3] Fila é um conceito visto em *PCS3110 – Algoritmos e Estruturas de Dados para a Engenharia Elétrica*. Ela é um conjunto dinâmico que segue a política de que o primeiro elemento a entrar no conjunto é o primeiro a sair.

[4] Essa tabela é tipicamente uma *tabela hash*, assunto explicado em *PCS3110 – Algoritmos e Estruturas de Dados para a Engenharia Elétrica*.

nós adjacentes, existindo um dos nós adjacentes que é definido como padrão - caso o roteador receba um pacote com endereço de destino que ele não sabe para quem repassar. Por exemplo, o roteador r2 da Figura 2 pode ter em sua tabela de repasse que o roteador r1 é o padrão e que pacotes para o computador B devem ser repassados para o roteador r4.

Em uma rede de pacotes real existem diversos atrasos que fazem com que a entrega de um pacote não seja imediata. Por exemplo, há o atraso de processamento (o tempo que o roteador demora para processar o pacote), o atraso de fila (o tempo que o pacote fica na fila) e o atraso de propagação (o tempo para propagar o dado pelo meio físico, como uma fibra ótica ou um cabo de par trançado de cobre).

1.2 Objetivo

O objetivo deste projeto é fazer um simulador de uma rede simplificada de computadores. Este projeto será desenvolvido incrementalmente e **em dupla** nos dois Exercícios Programas de PCS3111.

Neste primeiro EP trabalharemos apenas com roteadores. Os roteadores possuirão uma fila, para armazenar os datagramas recebidos (como são chamados os pacotes nessa camada), e uma tabela de repasse. Todos os roteadores ficarão em uma rede. Para que se simule a passagem do tempo, há um agendador que é responsável por garantir que os eventos na rede aconteçam no instante correto, simulando também o atraso de propagação.

A solução deve empregar adequadamente conceitos de Orientação a Objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor – o que representa o conteúdo até, *inclusive*, a Aula 5. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

2. Projeto

Deve-se implementar em C++ as classes **Agendador**, **Datagrama**, **Evento**, **Fila**, **Rede**, **Roteador** e **TabelaDeRepasse**, além de criar uma `main` que permita o funcionamento do programa como desejado.

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. **Note que você poderá definir atributos e método privados, 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 outros `#defines` além dos definidos neste documento. Use os valores de `#define` deste documento.

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

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).

A sugestão é que se comece a implementar o EP a partir da *Aula 4*. Todas as classes possuem construtores e destrutores (conceito da *Aula 5* – são os métodos com o mesmo nome da classe), mas é possível começar a implementar os métodos ainda sem esse conceito.

2.1 Classe Datagrama

Um **Datagrama** é o pacote que é transmitido entre roteadores. Além do dado a ser transmitido, que neste EP será apenas uma string, o **Datagrama** também possui o endereço de origem (qual nó o enviou) e o de destino (qual nó deve recebê-lo).

A classe **Datagrama** deve possuir apenas os seguintes métodos **públicos**:

```
Datagrama(int origem, int destino, string dado);
~Datagrama();
int getOrigem();
int getDestino();
string getDado();

void imprimir();
```

Os métodos `getOrigem`, `getDestino` e `getDado` devem retornar, respectivamente, os valores do endereço de origem, do endereço de destino e do dado informados no construtor.

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

2.2 Classe Fila

Uma **Fila**[5] é um conjunto dinâmico que segue a política de que o primeiro elemento a entrar no conjunto é o primeiro a sair, assim como as filas que temos no mundo real (filas de cinema, do bandejão, etc.). Essa classe deve implementar uma fila de **Datagramas**, a qual será usada por um **Roteador**.

A classe **Fila** deve possuir apenas os seguintes métodos **públicos**:

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

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

void imprimir();
```

Implemente a **Fila** da forma que você achar o mais apropriado (pode ser uma *fila circular* usando vetor ou uma *lista ligada*[6]). O construtor deve receber o tamanho máximo da **Fila**, o qual deve representar o número

[5] Seguiremos a mesma nomenclatura usada por *PCS3110 – Algoritmos e Estruturas de Dados para a Engenharia Elétrica*. Mais detalhes do funcionamento e implementação de uma fila podem ser vistos nas videoaulas disponíveis em http://eaulas.usp.br/portal/video.action?idItem=17756, http://eaulas.usp.br/portal/video.action?idItem=17758 e http://eaulas.usp.br/portal/video.action?idItem=17760.

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

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

O método enqueue 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 retornar `false`. Caso o **Datagrama** seja colocado na **Fila**, este método deve retornar `true`. 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, retorne `NULL`.

O método `isEmpty`[7] 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.

2.3 Classe TabelaDeRepasse

Uma **TabelaDeRepasse** mapeia endereços a **Roteadores**, gerenciando para qual **Roteador** deve ser repassado o **Datagrama** que possui um determinado endereço de destino. Além disso, ela deve possuir um **Roteador** padrão, para o qual serão repassados os **Datagramas** cujos endereços não estão explicitamente na **TabelaDeRepasse**. Para simular o atraso de propagação, a tabela também armazena o número de instantes que demora para o **Datagrama** chegar ao **Roteador** do endereço mapeado[8].

Por simplicidade, recomenda-se que essa classe seja implementada usando três vetores: um contendo os endereços de destino, o outro contendo os **Roteadores** adjacentes e outro contendo os atrasos. Quando se fizer um mapeamento de um endereço a um **Roteador** deve-se colocar na próxima posição disponível do vetor de endereços o endereço informado, na mesma posição do vetor de roteadores o **Roteador** associado e na mesma posição do vetor de atrasos o atraso. No início ambos os vetores estarão vazios. Por exemplo, ao mapear o endereço 4 ao **Roteador** r1 com atraso 2, na posição 0 do vetor de endereço ficará o valor 4, na posição 0 do vetor de roteadores ficará a referência a r1 e na posição 0 do vetor de atrasos ficará o valor 2. Se em seguida for mapeado o endereço 7 ao **Roteador** r4 com atraso 1, na posição 1 do vetor de endereço ficará o valor 7, na posição 1 do vetor de roteadores ficará a referência à r4 e na posição 1 do vetor de atraso ficará o valor 1.

Com isso, a classe **TabelaDeRepasse** deve possuir apenas os seguintes métodos **públicos**:

[7] 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++.

[8] Isso é uma simplificação: em um simulador mais fiel, isso deveria estar no canal físico, que é uma camada não representada neste EP.

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

bool mapear(int endereco, Roteador* adjacente, int atraso);
Roteador** getAdjacentes();
int getQuantidadeDeAdjacentes();

void setPadrao(Roteador* padrao, int atraso);

Roteador* getProximoSalto(int endereco, int& atraso);

void imprimir();
```

O construtor recebe como parâmetro o tamanho da tabela, ou seja, o número máximo endereços de destinos, roteadores adjacentes e atrasos. No construtor, defina o roteador padrão como `NULL` e seu atraso 0. O destrutor deve destruir os vetores alocados, mas não deve destruir os **Roteadores** adicionados ao vetor.

O método `mapear` deve associar o endereço passado como parâmetro ao **Roteador** adjacente e ao atraso informados no método. Caso o endereço já esteja na tabela, não faça nada e retorne `false`. Também retorne `false` caso não seja possível fazer o mapeamento pois a tabela já contém o tamanho máximo de elementos (informado no construtor). Esse método deve retornar `true` se foi possível fazer o mapeamento.

Os **Roteadores** mapeados à tabela devem ser obtidos pelo método `getAdjacentes`, que retorna um vetor de **Roteadores** (note que é possível que um **Roteador** apareça várias vezes nesse vetor caso ele seja mapeado a vários endereços). A quantidade de elementos nesse vetor deve ser obtida pelo método `getQuantidadeDeAdjacentes`. Por exemplo, se o vetor tiver os Roteadores {r1, r2, r1, r3, r2} o método `getQuantidadeDeAdjacentes` deve retornar 5.

O método `setPadrao` deve definir o **Roteador** padrão para essa tabela e o seu respectivo atraso. O **Roteador** padrão deve ser retornado como destino para endereços que não estejam mapeados. Note que o **Roteador** padrão **não** deve ser retornado pelo método `getAdjacentes`, a menos que ele tenha sido mapeado a um endereço.

O método `getProximoSalto` é o método que retorna para qual **Roteador** deve ser repassado o **Datagrama** cujo destino foi passado como parâmetro. Ele também retorna o atraso, através do parâmetro passado por referência `atraso` (veja a Aula 2 sobre isso). Para isso, ele deve considerar os endereços mapeados pelo método `mapear` e o **Roteador** padrão (definido por `setPadrao`). Se o endereço estiver mapeado, o método deve retornar o **Roteador** e o atraso relacionado ao endereço. Caso o endereço não esteja mapeado, o método deve retornar o **Roteador** padrão e o seu respectivo atraso (note que se o padrão não estiver definido, o retorno deve ser NULL e 0 de atraso). Por exemplo, considere que um **Roteador** teve endereço 4 mapeado ao **Roteador** r4 com atraso 2, e tem o **Roteador** r2 com atraso 3 como padrão. A chamada `getDestino(4)` deve retornar o **Roteador** r4 e o valor 2 de atraso. Para qualquer outra chamada de `getDestino` deve-se retornar o **Roteador** r2 e atraso 3 - por exemplo, se for feito `getDestino(2)` ou `getDestino(8)`.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado. E veja na Seção 4 como lidar com o problema de dependência circular entre essa classe e a classe **Roteador**.

2.4 Classe Evento

O **Evento** representa o evento de recebimento de um **Datagrama** por um **Roteador**, simulando assim o atraso de propagação. Essa classe deve possuir apenas os seguintes métodos **públicos**:

```
Evento(int instante, Roteador* destino, Datagrama* d);
~Evento();

int getInstante();
Roteador* getDestino();
Datagrama* getDatagrama();

void imprimir();
```

O construtor deve receber o instante em que o evento deve ser processado, o **Roteador** destino e o **Datagrama**. Esses valores são retornados pelos método `getInstante`, `getDestino` e `getDatagrama`, respectivamente.

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

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado. Veja na Seção 4 como lidar com o problema de dependência circular entre essa classe e a classe **Roteador**.

2.5 Classe Roteador

O **Roteador** é o elemento central deste EP e que fará o repasse de **Datagramas**. Como esta rede não possui outros nós além de **Roteadores**, os **Datagramas** terão como origem e destino os **Roteadores**.

A classe **Roteador** deve possuir apenas os seguintes métodos **públicos** e este **define**:

```
#define TAMANHO 10

Roteador(int endereco);
~Roteador();

bool mapear(int endereco, Roteador* adjacente, int atraso);
void setPadrao(Roteador* padrao, int atraso);

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

void imprimir();
```

O construtor deve receber o endereço do **Roteador**, o qual é retornado pelo método `getEndereco`. Na criação de um **Roteador** você deve criar a **TabelaDeRepasse** e a **Fila** com `TAMANHO` de tamanho. No destrutor deve-se destruir a **TabelaDeRepasse** e a **Fila** que foram criadas.

O método `mapear` deve mapear na **TabelaDeRepasse** deste **Roteador** o endereço ao **Roteador** e o atraso informados. Ele deve ter o mesmo retorno do método `mapear` da **TabelaDeRepasse**: `true` se o mapeamento foi feito ou `false` caso o endereço já tenha um mapeamento ou a tabela esteja cheia. O método `setPadrao` define o **Roteador** padrão da **TabelaDeRepasse** deste **Roteador**, assim como o seu atraso associado.

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

```
\tFila em <endereço> estourou
```

Onde `<endereço>` é o endereço do **Roteador**. Note o '\t' (tab) para indentar o texto.

O processamento do **Datagrama** só será feito na chamada do método `processar`. O parâmetro `instante` indica o momento em que o processamento é feito (necessário para gerar os **Eventos**). 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 **Roteador**, deve-se destruir o **Datagrama** e retornar `NULL`.
2. Caso o destino não seja o endereço deste **Roteador**, deve-se consultar a **TabelaDeRepasse** para descobrir para qual **Roteador** o **Datagrama** deve ser repassado e o atraso a ser considerado.
    a. Caso a **TabelaDeRepasse** retorne NULL, deve-se destruir o **Datagrama** e retornar NULL.
    b. Caso a **TabelaDeRepasse** retorne um **Roteador** e um atraso, deve-se criar um **Evento** cujo instante é o instante passado como parâmetro de `processar` somado ao atraso, e cujo destino é o **Roteador** retornado pela **TabelaDeRepasse**. Esse **Evento** deve ser então retornado pelo método.

Por exemplo, suponha que o **Roteador** com endereço 2 recebeu (pelo método `receber`) um **Datagrama** `{origem=1, destino=6, dado="Oi"}` e depois um outro **Datagrama** `{origem=4, destino=2, dado="Alo"}`. Considere que a primeira chamada do método `processar` ocorre no instante 10. Ela deve retirar o **Datagrama** `{origem=1, destino=6, dado="Oi"}` da **Fila**. Suponha que a **TabelaDeRepasse** do **Roteador** indique que o próximo salto para o endereço 6 é o **Roteador** r5 com atraso 1. Deve-se então criar um **Evento** com os valores `{instante=11, destino=r5, datagrama={origem=1, destino=6, dado="Oi"}}` e retorná-lo. Na próxima chamada do método `processar`, no instante 11, deve-se retirar da fila o **Datagrama** `{origem=4, destino=2, dado="Alo"}`. Como o endereço de destino é o do próprio **Roteador**, deve-se destruir o **Datagrama** e retornar `NULL`.

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

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

  ```
  Roteador <e>
  ```

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

  ```
  Roteador 1
  ```

  Além disso, deve ser impresso o resultado do processamento do **Datagrama** pelo **Roteador** da seguinte forma (pule uma linha após a impressão):

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

    ```
    \tRepassado para <r> (instante <i>): <datagrama>
    ```

Onde <r> é o endereço do **Roteador** para o qual o **Datagrama** foi repassado, <i> é o instante agendado para o **Evento** ser recebido pelo **Roteador** destino e <datagrama> são as informações do **Datagrama**. Note que \t é um tab. Por exemplo:

```
\tRepassado para 2 (instante 3): Origem: 1, Destino: 5, Algo
```

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

```
\tRecebido: <d>
```

Onde <d> é o dado do **Datagrama** como, por exemplo:

```
\tRecebido: Exemplo
```

- Caso o **Datagrama** não tenha um **Roteador** para o próximo salto (ou seja, `getProximoSalto` retornar `NULL`):

```
\tSem proximo: <datagrama>
```

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

```
\tSem proximo: Origem: 1, Destino: 5, Algo
```

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

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

Onde:

- <origem>: é o endereço de origem no **Datagrama**;
- <destino>: é o endereço de destino no **Datagrama**;
- <d>: é o dado.

Por exemplo, a impressão do **Datagrama** {origem=1, destino=4, dado="Ex"} seria:

```
Origem: 1, Destino: 4, Ex
```

Não faça outras impressões nesse método, pois isso pode afetar a correção. Assim como nas outras classes, nesta classe também é definido um método `imprimir`, o qual não tem seu funcionamento especificado. Implemente-o como desejado. E veja na Seção 4 como lidar com o problema de dependência circular entre essa classe e as classes **TabelaDeRepasse** e **Evento**.

2.6 Classe Rede

A **Rede** é a classe responsável por ter a lista de **Roteadores.** Ela deve possuir apenas os seguintes métodos **públicos**:

```
Rede(int tamanho);
~Rede();

bool adicionar(Roteador* r);
Roteador* getRoteador(int endereco);

Roteador** getRoteadores();
int getQuantidade();

void imprimir();
```

O construtor da **Rede** deve receber a quantidade máxima de **Roteadores** que podem ser adicionados. No destrutor destrua todos os **Roteadores** e os objetos alocados dinamicamente.
O método `adicionar` deve adicionar o **Roteador** à **Rede**. Caso não haja espaço disponível (a **Rede** já alcançou a quantidade máxima) ou já haja um **Roteador** com o endereço, esse método deve retornar `false` (e não adicionar o **Roteador**). Retorne `true` caso contrário.

O método `getRoteador` deve retornar o **Roteador** (dentre os adicionados) que possui o endereço informado. Caso não haja um **Roteador** com esse endereço, este método deve retornar `NULL`.

O método `getRoteadores` deve retornar um vetor com todos os **Roteadores** adicionados (retorne-os na ordem em que eles foram adicionados, ou seja, o primeiro adicionado deve ficar na posição 0 e o segundo na posição 1 etc.). A quantidade de elementos desse vetor deve ser retornada pelo método `getQuantidade`.

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

2.7 Classe Agendador

O **Agendador** cuida da "passagem do tempo" no nosso simulador. A chamada ao método `processar` dela primeiro processa todos os **Eventos** agendados para aquele instante e, em seguida, chama o método `processar` de todos os **Roteadores** da **Rede**, simulando que cada **Roteador** consegue processar um **Datagrama** por instante de tempo. Essa classe também permite agendar um **Evento**, o que será usado pelo `main`.

A classe **Agendador** possui os seguintes métodos públicos:

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

bool agendar(int instante, Roteador* r, Datagrama* d);
void processar();
int getInstante();
```

O construtor deve receber o valor do instante em que a simulação começa, a **Rede** que será simulada e a quantidade de **Eventos** que o **Agendador** pode receber ao mesmo tempo (tamanho). No destrutor destrua todos os **Eventos** agendados (e não processados) e os objetos alocados dinamicamente. Não destrua a **Rede**.

O método agendar deve criar um **Evento** com as informações passadas (instante, **Roteador** destino e **Datagrama**) e adicioná-lo ao vetor de **Eventos** agendados. Caso o vetor de eventos agendados não tenha mais espaço (já tenha a quantidade *tamanho* de **Eventos**), o método deve retornar `false` e não agendar o **Evento**. Caso contrário o método deve retornar `true`.

O método `processar` deve fazer o seguinte:

1. Todos os **Eventos** agendados para o instante atual devem ser processados, encaminhando para o **Roteador** correspondente o **Datagrama**. Após encaminhar, destrua o **Evento**.
2. Chame o método `processar` de todos os **Roteadores** obtidos pelo método `getRoteadores` da **Rede**, começando pelo **Roteador** na posição 0 e terminando com o **Roteador** na posição

`getQuantidade()` – 1. Adicione os **Eventos** retornados pelos métodos `processar` dos **Roteadores** à lista de **Eventos** do **Agendador**.

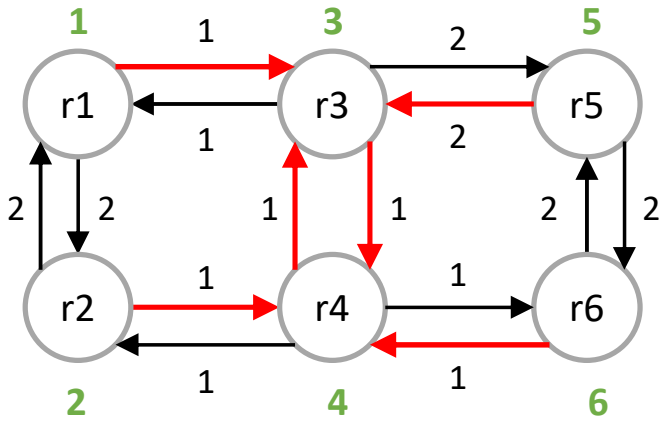
3. Incremente o instante (o tempo passou).

Por exemplo, considere que o **Agendador** no instante 1 possua os seguintes **Eventos**: `{instante: 1, destino: r3, datagrama: d1}`, `{instante: 4, destino: r1, datagrama: d2}`, `{instante: 1, destino: r2, datagrama: d3}` e `{instante: 2, destino: r4, datagrama: d4}` (por simplicidade não é apresentado o conteúdo do **Datagrama** neste exemplo). Ao chamar o método `processar`, o **Evento** `{instante: 1, destino: r3, datagrama: d1}` deve ser removido do vetor de **Eventos** agendados e deve-se chamar o método `receber` do **Roteador** r3 com o **Datagrama** d1. Da mesma forma, o **Evento** `{instante: 1, destino: r2, datagrama: d3}` também deve ser removido do vetor e o método receber do **Roteador** r2 deve ser chamado com d3. Em seguida todos os **Roteadores** devem ter o seu método `processar` chamados. Suponha que nessa chamada seja adicionado um novo **Evento** à lista do **Agendador**: `{instante: 3, destino: r1, datagrama: d5}`. Com isso, o **Agendador** possuirá os seguintes **Eventos**: `{instante: 4, destino: r1, datagrama: d2}`, `{instante: 2, destino: r4, datagrama: d4}` e `{instante: 3, destino: r1, datagrama: d5}`. Com todos os **Roteadores** processados, o instante é incrementado para 2.

O método `getInstante` deve retornar o instante de tempo do **Agendador**. Esse instante começa com o instante inicial informado no construtor e é incrementado em 1 a cada vez que o método `processar` é chamado.

3 Main

Coloque a `main` em um arquivo separado, chamado `main.cpp`. Nele você deverá criar a rede apresentada a seguir, composta por 6 roteadores (r1 a r6) e cujos endereços vão de 1 a 6 (apresentados em verde). Os mapeamentos são apresentados como arestas (setas) e o atraso é o peso. As setas em vermelho são os roteadores padrão. O instante inicial do **Agendador** deve ser 1 e a fila deve permitir no máximo 10 **Eventos**.

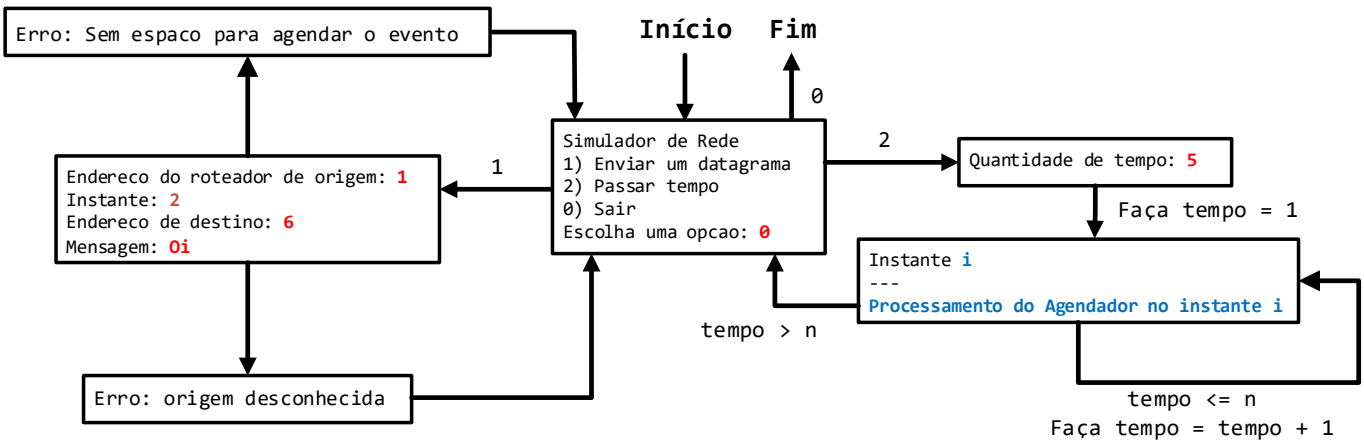


A tabela de repasse dos roteadores é apresentada a seguir, usando o formato *roteador/atraso*. Ou seja, o roteador r1 tem o roteador r3 como roteador padrão com atraso 1 e repassa ao roteador r2 o endereço 2 com atraso 2. Da mesma forma, o roteador r2 tem o roteador r4 como padrão com atraso 1 e repassa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

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

3.1 Interface

Além de criar a rede, o `main` deve possuir uma interface em console que permite enviar um **Datagrama** e 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.



- A opção 1 ("Enviar um datagrama") deve enviar um **Datagrama** usando o método `agendar` do **Agendador**.
- 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 **Roteadores**. Essas chamadas fazem impressões (veja na Seção 2.5 quais são as impressões esperadas). 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 tratamento disso. Assuma que o usuário *sempre* digitará um valor correto - a, menos, claro do endereço do **Roteador** de origem que deve ser um endereço de um nó que está na **Rede**.
- Por simplicidade considere que o dado do **Datagrama** 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.

3.2 Exemplo

Segue um exemplo de funcionamento do programa com a saída esperada e ressaltando em **vermelho** os dados digitados pelo usuário.

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

Endereco do roteador de origem: 1
Instante: 1
Endereco de destino: 5
Mensagem: Msg1

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

Endereco do roteador de origem: 3
Instante: 2
Endereco de destino: 5
Mensagem: Msg2

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

Endereco do roteador de origem: 2
Instante: 1
Endereco de destino: 3
Mensagem: Msg3

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

Quantidade de tempo: 3

Instante 1
---
Roteador 1
        Repassado para 3 (instante 2): Origem: 1, Destino: 5, Msg1
Roteador 2
        Repassado para 4 (instante 2): Origem: 2, Destino: 3, Msg3

Instante 2
---
Roteador 3
        Repassado para 5 (instante 4): Origem: 3, Destino: 5, Msg2
Roteador 4
        Repassado para 3 (instante 3): Origem: 2, Destino: 3, Msg3

Instante 3
---
```

```
Roteador 3
        Repassado para 5 (instante 5): Origem: 1, Destino: 5, Msg1

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

Endereco do roteador de origem: 2
Instante: 4
Endereco de destino: 4
Mensagem: Msg4

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

Quantidade de tempo: 3

Instante 4
---
Roteador 2
        Repassado para 4 (instante 5): Origem: 2, Destino: 4, Msg4
Roteador 3
        Recebido: Msg3
Roteador 5
        Recebido: Msg2

Instante 5
---
Roteador 4
        Recebido: Msg4
Roteador 5
        Recebido: Msg1

Instante 6
---

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

4 Dependência circular

Um problema de compilação é a existência de *dependências circulares*. Caso uma classe **A** use a classe **B** e a classe **B** use a classe **A**, ocorre uma dependência circular. O problema disso é que para compilar a classe **A** é preciso *antes* compilar a classe **B**, mas para compilar a classe **B** é preciso *antes* compilar a classe **A**! Esse problema acontece no EP com as classes **Roteador** e **TabelaDeRepasse**, e **Roteador** e **Evento**.

Para resolver esse problema em C++ é preciso definir um protótipo da classe. A seguir é apresentado o exemplo com as classes **A** e **B**:

```h
#ifndef A_H
#define A_H

#include "B.h"

class B; // Protótipo

class A {
private:
  B* b; // Exemplo de uso de B em A
  ...
};

#endif // A_H
```

```h
#ifndef B_H
#define B_H

#include "A.h"

class A; // Protótipo

class B {
private:
  A* a; // Exemplo de uso de A em B
  ...
};

#endif // B_H
```

Note que é feito um `#include` da classe usada, como usual. Porém, após o `#include` é definido um protótipo da outra classe, assim como protótipos de função (note que ambas as classes `A` e `B` precisam ter protótipos). **Mas tome cuidado**: só coloque o protótipo de uma classe se for realmente necessário. Colocar um protótipo de uma classe quando não é necessário pode gerar erros de compilação difíceis de identificar! Neste EP a única situação que há esse problema é na relação das classes **Roteador** e **TabelaDeRepasse,** e **Roteador** e **Evento.**

Compiladores de outras linguagens resolvem esse problema de outras formas!

5 Entrega

O projeto deverá ser entregue até dia **22/10** em um Judge específico, disponível em <https://laboo.pcs.usp.br/ep/> (nos próximos dias vocês receberão um login e uma senha).

As duplas podem ser formadas por alunos de qualquer turma e elas devem ser informadas no e-Disciplinas até dia 05/10. Caso não seja informada a dupla, será considerado que o aluno está fazendo o EP sozinho. **Note que no EP2 deve-se manter a mesma dupla do EP1 (será apenas possível *desfazer* a dupla, mas não formar uma nova).**

> **Atenção:** não copie código de um outro grupo. Qualquer tipo de cópia será considerado plágio e **todos** os alunos dos 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**). O nome do arquivo não pode conter espaço ou ".". 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 à 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:

- Datagrama construtor getters e destrutor
- Evento construtor e destrutor
- Evento getters
- Fila construtor e destrutor
- Fila dequeue e underflow
- Fila enqueue dequeue e getSize
- Fila enqueue e depois dequeue até o tamanho
- Fila enqueue e dequeue mais que tamanho datagramas no total
- Fila enqueue e getSize
- Fila enqueue e isEmpty
- Fila enqueue e overflow
- Fila enqueue intercalado com dequeue até tamanho
- Fila overflow e mais que tamanho datagramas no total
- Fila underflow e mais que tamanho datagramas no total
- Rede adicionar e getRoteadores
- Rede adicionar e getRoteadores overflow
- Rede adicionar já possui roteador com endereço
- Rede construtor destrutor
- Rede getRoteador
- Rede getRoteador endereços inválidos
- TabelaDeRepasse construtor e destrutor
- TabelaDeRepasse getProximoSalto
- TabelaDeRepasse getProximoSalto e padrão
- TabelaDeRepasse getProximoSalto e padrão NULL
- TabelaDeRepasse getProximoSalto e substituição
- TabelaDeRepasse mapear até tamanho
- TabelaDeRepasse mapear com endereço igual
- TabelaDeRepasse mapear e getAdjacentes
- TabelaDeRepasse setPadrao e getAdjacentes

Parte 2:

- Roteador construtor getEndereco e destrutor
- Roteador receber estourou
- Roteador processar vazio
- Roteador receber e processar destino
- Roteador receber e processar sem próximo
- Roteador receber e processar repassando
- Roteador receber e processar variado1
- Roteador receber e processar variado2

Parte 3:

- Agendador: Construtor Destrutor e getInstante
- Agendador: processar sem evento
- Agendador: processar 1 evento no inicio
- Agendador: processar vários eventos mesmo instante
- Agendador: agendar mais eventos que tamanho e processar
- Agendador: processar com agendamento após processamento
- Agendador: processar eventos variados 1
- Agendador: processar eventos variados 2

6 Dicas

- 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).

- É muito trabalhoso testar o programa ao executar o `main` *com menus*, já que é necessário informar vários dados para agendar **Eventos** e processar o **Agendador**. 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() {
}
```

- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- 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.**