## 7. Filas

**7.1 - Introdução**

Uma fila é uma estrutura de dados dinâmica que admite [remoção](https://www.ime.usp.br/~pf/algoritmos/aulas/array.html#remocao) de elementos e [inserção](https://www.ime.usp.br/~pf/algoritmos/aulas/array.html#insercao) de novos objetos.  Mais especificamente, uma *fila* = *queue,* é uma estrutura sujeita à seguinte regra de operação:  sempre que houver uma remoção, o elemento removido é o que está na estrutura há *mais* tempo.



No dia a dia, estamos acostumados com as filas em diversos lugares: nos bancos, nos mercados, nos hospitais, nos cinemas entre outros. As filas são importantes pois elas determinam a ordem de atendimento das pessoas.

As pessoas são atendidas conforme a posição delas na fila. O próximo a ser atendido é o primeiro da fila. Quando o primeiro da fila é chamado para ser atendido a fila "anda", ou seja, o segundo passa a ser o primeiro, o terceiro passa a ser o segundo e assim por diante até a última pessoa.

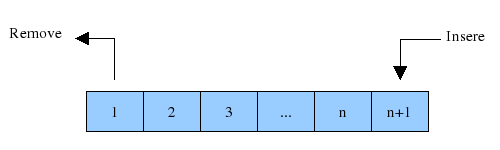
Normalmente, para entrar em uma fila, uma pessoa deve se colocar na última posição, ou seja, no fim da fila. Desta forma, quem chega primeiro tem prioridade.

Neste capítulo, estamos interessados em desenvolver estrutura de dados com os comportamentos das filas. Assim como Listas e Pilhas, as Filas são estruturas de dados que armazenam os elementos de maneira sequencial.

Assim como as Pilhas, as Filas têm operações mais **restritas** do que as operações das Listas. Nas Filas, os elementos são adicionados na última posição e removidos da primeira posição. Nas Listas, os elementos são adicionados e removidos de qualquer posição.

Então, podemos implementar uma Fila simplesmente colocando as restrições adequadas nas operações de adicionar e remover elementos de uma Lista. Isso é bem parecido ao que fizemos com as Pilhas.

Em outras palavras, o primeiro objeto inserido na fila é também o primeiro a ser removido. Essa política é conhecida pela sigla 0 ou seja, adiciona-se itens no fim e remove-se do início.



São exemplos de uso de fila em um sistema:

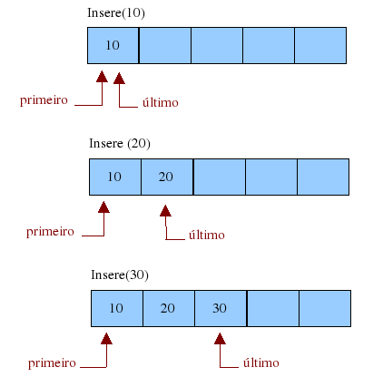
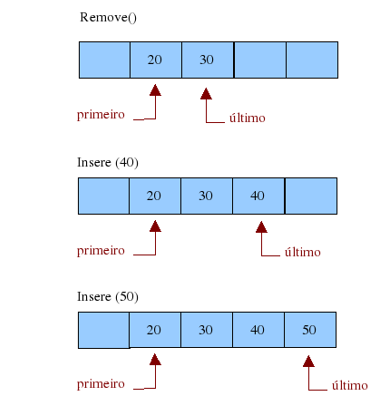
* Controle de documentos para impressão;
* Troca de mensagens entre computadores numa rede;
* Distâncias entre diferentes cidades (teoria dos grafos).

A implementação de filas pode ser realizada através de vetor (alocação do espaço de memória para os elementos é contígua) ou através de listas encadeadas (próxima aula).

**Operações com Fila:**

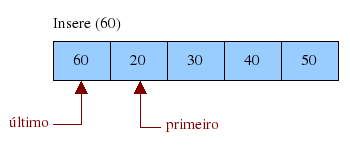
Todas as operações em uma fila podem ser imaginadas como as que ocorre numa fila de pessoas num banco, exceto que os elementos não se movem na fila, conforme o primeiro elemento é retirado. Isto seria muito custoso para o computador. O que se faz na realidade é indicar quem é o primeiro.

1. criação da fila (informar a capacidade no caso de implementação sequencial - vetor);
2. enfileirar (enqueue) - o elemento é o parâmetro nesta operação;
3. desenfileirar (dequeue);
4. mostrar a fila (todos os elementos);
5. verificar se a fila está vazia (isEmpty);
6. verificar se a fila está cheia (isFull - implementação sequencial - vetor).

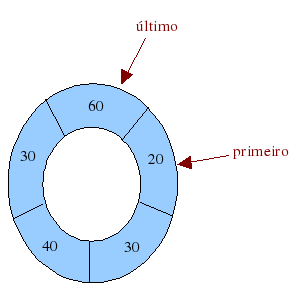
**Supondo uma fila com capacidade para 5 elementos (5 nós).**

Na realidade a remoção de um elemento da fila é realizada apenas alterando-se a informação da posição do último.

Para evitar problemas de não ser capaz de inserir mais elementos na fila, mesmo quando ela não está cheia, as referências primeiro e último circundam até o inicio do vetor, resultando numa fila circular.



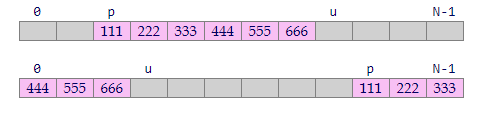
Desta forma a fila simula uma representação circular:



**Implementação circular**

Na implementação que adotamos acima, a fila anda para a direita dentro do vetor que a abriga.  Isso pode tornar difícil prever o valor que o [parâmetro N](https://www.ime.usp.br/~pf/algoritmos/aulas/fila.html#N) deve ter para evitar o transbordamento da fila.  Uma implementação circular pode ajudar a adiar o transbordamento, como mostraremos a seguir.

Suponha que os elementos da fila estão dispostos no vetor  fila[0..N-1]  de uma das seguintes maneiras:   fila[p..u-1]   ou   fila[p..N-1] fila[0..u-1] .



Teremos sempre  0 ≤ p < N  e  0 ≤ u < N,  mas não podemos supor que p ≤ u .  A fila está  *vazia*  se   u == p  e  *cheia*  se   u+1 == p   ou   u+1 == N e p == 0   (ou seja, se (u+1) % N == p).

A posição anterior a p ficará sempre desocupada para que possamos distinguir uma fila cheia de uma vazia.  Com essas convenções, a remoção de um elemento da fila pode ser escrita assim:

int tiradafila (void) {

int x = fila[p++];

if (p == N) p = 0;

return x;

}

(desde que a fila não esteja vazia).  A inserção de um objeto y na fila pode ser escrita assim:

void colocanafila (int y) {

fila[u++] = y;

if (u == N) u = 0;

}

(desde que a fila não esteja cheia).

**Implementação em um vetor**

Suponha que nossa fila mora em um vetor fila[0..N-1].  (A natureza dos elementos do vetor é irrelevante: eles podem ser inteiros, bytes, ponteiros, etc.)  Digamos que a parte do vetor ocupada pela fila é

fila[p..u-1] .

O primeiro elemento da fila está na posição p e o último na posição u-1.  A fila está *vazia* se  p == u  e *cheia* se  u == N.   A figura mostra uma fila que contém os números 111, 222, … , 666:



Para *tirar*, ou *remover* (= [*delete*](https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/deletar.html) = *de-queue*), um elemento da fila basta fazer

x = fila[p++];

Isso [equivale](https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/value-of-expr.html) ao par de instruções  x = fila[p]; p += 1;,  nesta ordem.  É claro que você só deve fazer isso se tiver certeza de que a fila não está vazia.   Para *colocar*, ou *inserir* (= *insert* = *enqueue*), um objeto y na fila basta fazer

fila[u++] = y;

Isso equivale ao par de instruções  fila[u] = y; u += 1;,  nesta ordem.  Note como esse código funciona corretamente mesmo quando a fila está vazia. É claro que você só deve inserir um objeto na fila se ela não estiver cheia; caso contrário, a fila *transborda* (ou seja, ocorre um *overflow*).

Para ajudar o leitor humano, podemos embalar as operações de remoção e inserção em duas pequenas funções. Se os objetos da fila forem números inteiros, podemos escrever

int tiradafila (void) {

return fila[p++];

}

void colocanafila (int y) {

fila[u++] = y;

}

Estamos supondo aqui que as variáveis fila, p, u e N são [globais](https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/global-variable.html), isto é, foram definidas fora do código das funções.  (Para completar o pacote, precisaríamos de mais três funções: uma que crie uma fila, uma que verifique se a fila está vazia e uma que verifique se a fila está cheia; veja exercício [abaixo](https://www.ime.usp.br/~pf/algoritmos/aulas/fila.html#filadeints-module-simple).)

**7.2 - Interface de uso**

Vamos implementar na sequência uma Fila de Aluno. Iremos aproveitar a classe Aluno feita no capítulo de armazenamento sequencial.

As operações que formam a interface de uso da Fila de alunos são:

1. Insere um Aluno (coloca um aluno no fim da Fila).
2. Remove um Aluno (retira o aluno que está no começo da Fila).
3. Informa se a Fila está vazia.

O esboço da classe Fila seria mais ou menos assim:

public class Fila {

public void insere(Aluno aluno) {

// implementação

}

public Aluno remove() {

// implementação

}

public boolean vazia() {

// implementação

}

}

Agora que já temos a interface de uso da Fila definida vamos escrever algum teste sobre como ela deveria se comportar.

public class Teste {

public static void main(String[] args) {

Fila fila = new Fila();

Aluno aluno = new Aluno();

fila.insere(aluno);

Aluno alunoRemovido = fila.remove();

if (fila.vazia()) {

System.out.println("A fila está vazia");

}

}

}

Como já foi dito aqui vamos implementar Fila utilizando algum tipo de Lista. Neste capítulo, vamos utilizar a classe LinkedList para armazenar os alunos da Fila.

public class Fila {

private List<Aluno> alunos = new LinkedList<Aluno>();

}

**7.3 - Operações em Fila**

Em seguida, implementaremos as operações da Fila de aluno.

**7.4 - Inserir um aluno**

Os alunos que entram na Fila devem sempre se colocar no fim da mesma. Vamos definir que o fim da Fila é o fim da Lista que estamos utilizando para implementar.

Então, entrar na Fila e adicionar no fim da Lista.

public class Fila {

private List<Aluno> alunos = new LinkedList<Aluno>();

public void insere(Aluno aluno) {

this.alunos.add(aluno);

}

}

**7.5 - Remover um aluno**

O próximo aluno a ser atendido é sempre o que está no início da Fila. No nosso caso, quem está no início da Fila é o aluno que está no início da Lista.

Então, basta remover o primeiro aluno.

public class Fila {

private List<Aluno> alunos = new LinkedList<Aluno>();

...

public Aluno remove() {

return this.alunos.remove(0);

}

}

É bom observar que se o método remove() for usado com a Fila vazia então uma exceção será lançada pois o método removeFirst() lança IndexOutOfBoundsException quando não existir elemento para remover.

**7.6 - Informar se a Fila está vazia**

Para implementar esta operação basta verificar se o tamanho da Lista é zero.

public class Fila {

private List<Fila> alunos = new LinkedList<Fila>();

...

public boolean vazia() {

return this.alunos.size() == 0;

}

}

**7.7 - Generalização**

Nossa Fila só funciona para guardar objetos da classe Aluno. Vamos generalizá-la para poder armazenar qualquer tipo de objeto. Isso será feito utilizando a classe Object da qual todas as classes derivam direta ou indiretamente. Criaremos uma LinkedList de Object em vez de uma LinkedList de Aluno.

public class Fila {

private List<Object> objetos = new LinkedList<Object>();

public void insere(Object objeto) {

this.objetos.add(objeto);

}

public Object remove() {

return this.objetos.remove(0);

}

public boolean vazia() {

return this.objetos.size() == 0;

}

}

Agora, podemos guardar qualquer tipo de objeto na Fila. Isso é uma grande vantagem pois a classe Fila poderá ser reaproveitada em diversas ocasiões. Mas, há uma desvantagem, quando removemos um elemento da Fila não podemos garantir qual é o tipo de objeto que virá.

A solução para este problema é utilizar o recurso do Generics. A nossa classe Fila vai ser uma classe parametrizada. Assim, quando criarmos uma Fila poderemos definir com qual tipo de objetos ela deve trabalhar.

Algo deste tipo:

Fila de alunos fila = new Fila de alunos();

Traduzindo este código para Java, ficaria assim:

Fila<Aluno> fila = new Fila<Aluno>();

Agora, precisamos parametrizar a classe Fila.

public class Fila<T> {

private List<T> objetos = new LinkedList<T>();

public void insere(T t) {

this.objetos.add(t);

}

public T remove() {

return this.objetos.remove(0);

}

public boolean vazia() {

return this.objetos.size() == 0;

}

}

Vamos criar duas Filas, uma para Aluno e outra para String.

public class Teste {

public static void main(String[] args) {

Fila<Aluno> fila = new Fila<Aluno>();

Aluno aluno = new Aluno();

fila.insere(aluno);

Aluno alunoRemovido = fila.remove();

if (fila.vazia()) {

System.out.println("A fila está vazia");

}

Fila<String> filaDeString = new Fila<String>();

filaDeString.insere("Adelaide");

filaDeString.insere("Carolina");

String carolina = filaDeString.remove();

String adelaide = filaDeString.remove();

System.out.println(carolina);

System.out.println(adelaide);

}

}

Em tempo de compilação, é verificado o tipo de objetos que estão sendo adicionados na Fila. Portanto, se você tentar inserir um objeto do tipo Aluno em uma Fila de String um erro de compilação será gerado.

public class Teste {

public static void main(String[] args) {

Aluno aluno = new Aluno();

Fila<String> filaDeString = new Fila<String>();

// este código não compila

filaDeString.insere(aluno);

}

}

**7.8 - API do Java**

Na biblioteca do Java, existe uma interface que define a estrutura de dados Fila. Essa interface chama-se Queue, umas das classes que implementam Queue é a LinkedList. O funcionamento fica extremamente parecido com a implementação que fizemos neste capítulo.

public class Teste {

public static void main(String[] args) {

Queue fila = new LinkedList();

Aluno aluno = new Aluno();

fila.offer(aluno);

Aluno alunoRemovido = (Aluno)fila.poll();

if(fila.isEmpty()){

System.out.println("A fila está vazia");

}

}

}

Para evitar fazer casting de objetos, podemos utilizar o recurso de Generics aqui também.

public class Teste {

public static void main(String[] args) {

Queue<Aluno> fila = new LinkedList<Aluno>();

Aluno aluno = new Aluno();

fila.offer(aluno);

Aluno alunoRemovido = fila.poll();

if(fila.isEmpty()){

System.out.println("A fila está vazia");

}

}

}

**7.9 - Exercícios: Fila**

1. Implemente a classe Fila para alunos vista neste capítulo. Coloque a classe no pacote **br.com.caelum.ed.filas**

package br.com.caelum.ed.filas;

import java.util.LinkedList;

import java.util.List;

import br.com.caelum.ed.Aluno;

public class Fila {

private List<Aluno> alunos = new LinkedList<Aluno>();

public void insere(Aluno aluno) {

this.alunos.add(aluno);

}

public Aluno remove() {

return this.alunos.remove(0);

}

public boolean vazia() {

return this.alunos.size() == 0;

}

}

Faça alguns testes.

package br.com.caelum.ed.filas;

import br.com.caelum.ed.Aluno;

public class Teste {

public static void main(String[] args) {

Fila fila = new Fila();

Aluno aluno = new Aluno();

fila.insere(aluno);

Aluno alunoRemovido = fila.remove();

if (aluno != alunoRemovido) {

System.out.println("Erro: o aluno removido não é " +

" igual ao que foi inserido");

}

if (!fila.vazia()) {

System.out.println("Erro: A fila não está vazia");

}

}

}

Se não for impresso nenhuma mensagem de erro significa que a Fila está funcionando.

1. Implemente a classe FilaGenerica para objetos (genérica) vista neste capítulo. Coloque a classe no pacote **br.com.caelum.ed.filas**

package br.com.caelum.ed.filas;

import java.util.LinkedList;

import java.util.List;

public class FilaGenerica {

private List<Object> objetos = new LinkedList<Object>();

public void insere(Object objeto) {

this.objetos.add(objeto);

}

public Object remove() {

return this.objetos.remove(0);

}

public boolean vazia() {

return this.objetos.size() == 0;

}

}

Faça alguns testes.

package br.com.caelum.ed.filas;

import br.com.caelum.ed.Aluno;

public class TesteFilaGenerica {

public static void main(String[] args) {

FilaGenerica filaDeAlunos = new FilaGenerica();

Aluno aluno = new Aluno();

filaDeAlunos.insere(aluno);

Aluno alunoRemovido = filaDeAlunos.remove();

if (aluno != alunoRemovido) {

System.out.println("Erro: o aluno removido não é igual " +

" ao que foi inserido");

}

if (!filaDeAlunos.vazia()) {

System.out.println("Erro: A fila não está vazia");

}

}

}

Perceba que a classe TesteFilaGenerica contém um erro de compilação. Quando você remove um elemento da FilaGenerica você recebe uma referência do tipo Object e não do tipo Aluno.

Altere a linha:

Aluno alunoRemovido = filaDeAlunos.remove();

Por:

Object alunoRemovido = filaDeAlunos.remove();

Isso faz o código compilar mas agora você não tem mais a garantia de tipo. Não sabe se a referência que você recebeu realmente está apontado para um objeto do tipo Aluno.

1. Implemente a classe FilaParametrizada utilizando o recurso do **Generics**. Coloque a classe no pacote **br.com.caelum.ed.filas**

package br.com.caelum.ed.filas;

import java.util.LinkedList;

import java.util.List;

public class FilaParametrizada<T> {

private List<T> objetos = new LinkedList<T>();

public void insere(T t) {

this.objetos.add(t);

}

public T remove() {

return this.objetos.remove(0);

}

public boolean vazia() {

return this.objetos.size() == 0;

}

}

Faça alguns testes:

package br.com.caelum.ed.filas;

import br.com.caelum.ed.Aluno;

public class TesteFilaGenerica {

public static void main(String[] args) {

FilaParametrizada<Aluno> filaDeAlunos =

new FilaParametrizada<Aluno>();

Aluno aluno = new Aluno();

filaDeAlunos.insere(aluno);

Aluno alunoRemovido = filaDeAlunos.remove();

if (aluno != alunoRemovido) {

System.out.println("Erro: o aluno removido não é igual " +

" ao que foi inserido");

}

if (!filaDeAlunos.vazia()) {

System.out.println("Erro: A fila não está vazia");

}

FilaParametrizada<String> filaDeString =

new FilaParametrizada<String>();

filaDeString.insere("Diana");

filaDeString.insere("Joaquim");

System.out.println(filaDeString.remove());

System.out.println(filaDeString.remove());

}

}

1. (**opcional**) É possível implementar a nossa Fila utilizando internamente uma ArrayList em vez de LinkedList? Teremos algum ganho ou perda no consumo de tempo de alguma das operações? Mostre a diferença através de um código que adiciona e remova muita gente da fila.

1) Existem partes de sistemas operacionais que cuidam da ordem em que os programas devem ser executados. Por exemplo, em um sistema de computação de tempo compartilhado (“time-shared”) existe a necessidade de manter um conjunto de processos em uma fila, esperando para serem executados.

Escreva um programa que seja capaz de ler uma série de solicitações para:

a. Incluir novos processos na fila de processo;

b. Retirar da fila o processo com o maior tempo de espera;

c. Imprimir o conteúdo da lista de processo em determinado momento.

Assumir que cada processo é representado por um registro composto por um número identificador do processo.

2) Uma empresa de lavagem de carros deseja controlar a fila de veículos que são atendidos. Os carros são atendidos na ordem em que chegam. Cada lavagem requer 15 minutos para ser completamente lavado, secado e limpo. Logo que o serviço termina o carro deixa a garagem. Implemente uma classe que controle essa fila de atendimento e que contenha métodos para:

a. Saber o número de carros esperando atendimento;

b. O tempo de espera ao inserir novos carros

c. O tempo de total que os carros permaneceram na fila (considerar todo o histórico de carros atendidos no dia).

**Aplicação: distâncias**

A ideia de fila aparece naturalmente no cálculo de [distâncias em um grafo](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/shortestpaths.html).  (Uma versão simplificada do problema é a [varredura por níveis](https://www.ime.usp.br/~pf/algoritmos/aulas/bint.html#level-traversal) de uma [árvore binária](https://www.ime.usp.br/~pf/algoritmos/aulas/bint.html).)  Imagine N cidades numeradas de 0 a N-1 e interligadas por estradas de mão única. As ligações entre as cidades são representadas por uma matriz A da seguinte maneira:

A[i][j]  vale  1  se existe estrada de i para j e vale 0 em caso contrário. Suponha que a matriz tem zeros na diagonal, embora isso seja irrelevante.  Segue um exemplo em que N vale 6:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | 4 | 5 | | 0 | 0 | 1 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 1 | 0 | 0 | 0 | | 2 | 0 | 0 | 0 | 0 | 1 | 0 | | 3 | 0 | 0 | 1 | 0 | 1 | 0 | | 4 | 1 | 0 | 0 | 0 | 0 | 0 | | 5 | 0 | 1 | 0 | 0 | 0 | 0 | |  | [distâncias em um grafo] |

A [distância](https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/distancia.html) de uma cidade c a uma cidade j é o menor número de estradas que precisamos percorrer para ir de c a j.  (A distância de c a j é, em geral, diferente da distância de j a c.)  Nosso [problema](https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/problem-solving.html):  dada uma cidade c,

determinar a distância de c a cada uma das demais cidades.

As distâncias podem ser armazenadas em um vetor dist: a distância de c a j é dist[j].  É preciso tomar uma decisão de projeto para cuidar do caso em que é impossível ir de c a j.  Poderíamos dizer que dist[j] é infinito nesse caso;  mas é mais limpo e prático dizer que dist[j] vale N, pois nenhuma distância real pode ser maior que N-1.  Se tomarmos c igual a 3 no exemplo acima, teremos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 |
| dist[i] | 2 | 3 | 1 | 0 | 1 | 6 |

Eis a ideia de um algoritmo que usa uma fila de cidades ativas para resolver nosso problema. Uma cidade i é ativa se dist[i] já foi calculada mas as estradas que começam em i ainda não foram todas exploradas. Em cada [iteração](https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/interativo.html), o algoritmo

tira da fila uma cidade i e coloca na fila todas as cidades vizinhas a i cujas distâncias ainda não foram calculadas.

Segue o código que implementa a ideia. (Veja antes um [rascunho em pseudocódigo](https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/distancias-pseudocode.html).)  Para simplificar, as variáveis fila, p, u e dist são [globais](https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/global-variable.html), ou seja, são definidas fora do código das funções.

[#define](https://www.ime.usp.br/~pf/algoritmos/apend/preprocessor.html#define-directive) N 100

int fila[N], int p, u;

int dist[N];

void criafila (void) {

p = u = 0;

}

int filavazia (void) {

return [p >= u](https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/value-of-boolean.html);

}

int tiradafila (void) {

return fila[p++];

}

void colocanafila (int y) {

fila[u++] = y;

}

// Esta função recebe uma matriz A

// que representa as interligações entre

// cidades 0..N-1 e preenche o vetor dist

// de modo que dist[i] seja a distância

// da cidade c à cidade i, para cada i.

void **distancias** (int A[][N], int c) {

for (int j = 0; j < N; ++j) dist[j] = N;

dist[c] = 0;

**criafila ()**;

**colocanafila (c)**;

while (! **filavazia ()**) {

int i = **tiradafila ()**;

for (int j = 0; j < N; ++j)

if (A[i][j] == 1 && dist[j] >= N) {

dist[j] = dist[i] + 1;

**colocanafila (j)**;

}

}

}

(Poderíamos operar a fila diretamente, sem invocar as funções de manipulação da fila. O resultado [seria mais curto e compacto](https://www.ime.usp.br/~pf/algoritmos/aulas/solucoes/fila0.html), mas um pouco menos legível.)