

Algoritmos e Estruturas de Dados 1 (AED1)

Filas em vetor circular e em lista ligada, interfaces, listas de adjacência

Uma fila (no inglês queue) é uma lista dinâmica, em que

- o primeiro a entrar é o primeiro a sair,
 - política First-In-First-Out (FIFO).
- Por isso, sempre removemos do início e inserimos no final da sequência.

Implementação de fila em vetor circular

Uma fila q é armazenada em um vetor de tamanho n

- alocado estática ou dinamicamente.

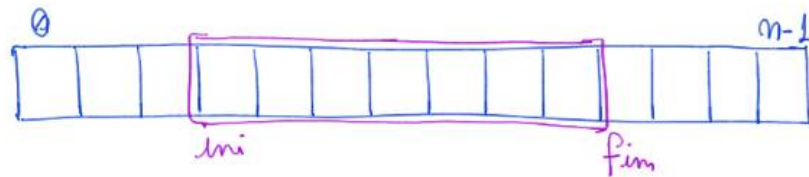
Um inteiro fim indica o final da fila,

- que é 1 a mais que a posição do último elemento e
- é a posição do próximo elemento a ser inserido.

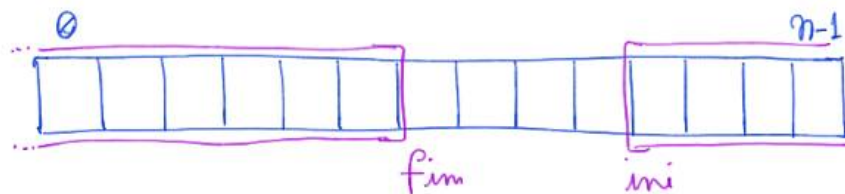
Um inteiro ini indica o início da fila, que é a posição do primeiro elemento e

- é a posição do próximo elemento a ser removido.

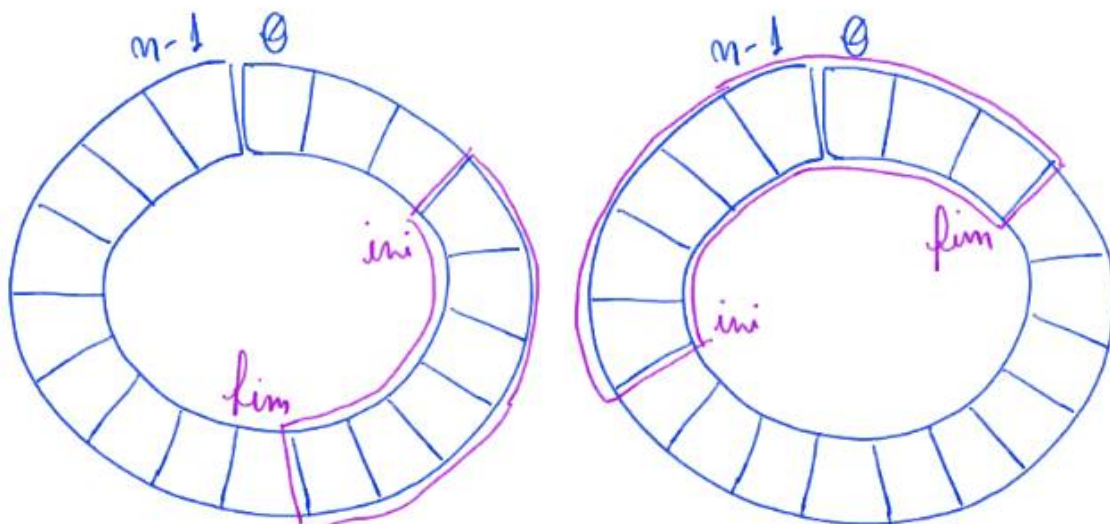
Na implementação circular, a fila está no subvetor $v[ini .. fim - 1]$



- ou na concatenação do subvetor $v[ini .. n - 1]$ com $v[0 .. fim - 1]$



Perspectiva circular das situações anteriores:



Para inserir um elemento x fazemos

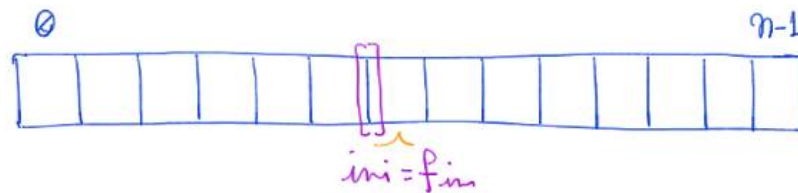
- $q[\text{fim}++] = x;$
- Implementação da circularidade
 - $\text{if } (\text{fim} == n) \text{ fim} = 0;$
- Circularidade com aritmética modular
 - $\text{fim} = \text{fim} \% n;$

Para remover um elemento e armazená-lo em x fazemos

- $x = q[\text{ini}++];$
- Implementação da circularidade
 - $\text{if } (\text{ini} == n) \text{ ini} = 0;$
- Circularidade com aritmética modular
 - $\text{ini} = \text{ini} \% n;$

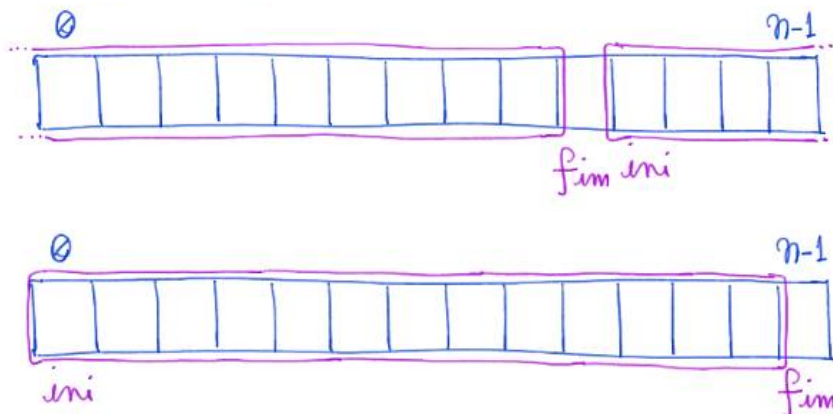
Fila vazia

- $\text{ini} == \text{fim};$



Fila cheia

- $\text{fim} + 1 == \text{ini} \parallel (\text{fim} + 1 == n \ \&\& \ \text{ini} == 0)$
- Alternativa com aritmética modular
 - $(\text{fim} + 1) \% n == \text{ini};$



- Quiz1: Note que, a posição fim sempre está desocupada. Por que?
 - Como diferenciar fila vazia de fila cheia?

Tamanho

- $\text{if } (\text{fim} \geq \text{ini}) \text{ tam} = \text{fim} - \text{ini};$
- $\text{if } (\text{fim} < \text{ini}) \text{ tam} = (n - \text{ini}) + (\text{fim} - 0);$

Note que as operações de manipulação da fila

- levam tempo constante, i.e., $O(1)$.

Biblioteca para implementação circular de fila em vetor

Segue o código da interface fila.h:

- observe que a definição do tipo “type” torna a fila genérica.

```
typedef struct fila Fila;

// typedef char type;
typedef int type;

Fila *criaFila();
void insereFila(Fila *q, type x);
type removeFila(Fila *q);
int filaVazia(Fila *q);
int filaCheia(Fila *q);
void imprimeFila(Fila *q);
int tamFila(Fila *q);
Fila *liberaFila(Fila *q);
```

A seguir temos a implementação da biblioteca usando vetor circular.

```
#include <stdio.h>
#include <stdlib.h>

#include "fila.h"

#define TAM_MAX 100

struct fila {
    type *vetor;
    int ini;
    int fim;
};

Fila *criaFila() {
    Fila *q;
    q = (Fila *)malloc(sizeof(Fila));
    q->vetor = (type *)malloc(TAM_MAX * sizeof(type));
    // Quiz2: Por que os seguintes valores? Faz diferença?
    q->ini = TAM_MAX / 2;
    q->fim = TAM_MAX / 2;
    return q;
}
```

```

void insereFila(Fila *q, type x) {
    q->vetor[q->fim] = x;
    // (q->fim)++;
    // if (q->fim == N)
    //     q->fim = 0;
    q->fim = (q->fim + 1) % TAM_MAX;
}

type removeFila(Fila *q) {
    type x;
    x = q->vetor[q->ini];
    // (q->ini)++;
    // if (q->ini == N)
    //     q->ini = 0;
    q->ini = (q->ini + 1) % TAM_MAX;
    return x;
}

int filaVazia(Fila *q) {
    return q->fim == q->ini;
}

int filaCheia(Fila *q) {
    // return (q->fim + 1 == q->ini || (q->fim + 1 == TAM_MAX &&
    q->ini == 0));
    return (q->fim + 1) % TAM_MAX == q->ini;
}

void imprimeFila(Fila *q) {
    int i; // note que os prints dependem do tipo
    if (q->ini <= q->fim)
        for (i = q->ini; i < q->fim; i++)
            printf("%c ", q->vetor[i]);
    else // q->fim < q->ini
        {
            for (i = q->ini; i < TAM_MAX; i++)
                printf("%c ", q->vetor[i]);
            for (i = 0; i < q->fim; i++)
                printf("%c ", q->vetor[i]);
        }
    printf("\n");
}

```

```

int tamFila(Fila *q) {
    if (q->ini <= q->fim)
        return q->fim - q->ini;
    return (TAM_MAX - q->ini) + (q->fim - 0);
}

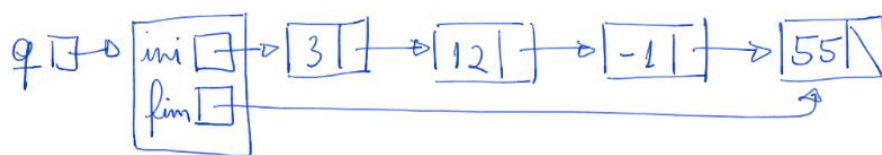
Fila *liberaFila(Fila *q) {
    free(q->vetor);
    free(q);
    return NULL;
}

```

Implementação de fila em lista ligada

Antes de começar a implementação,

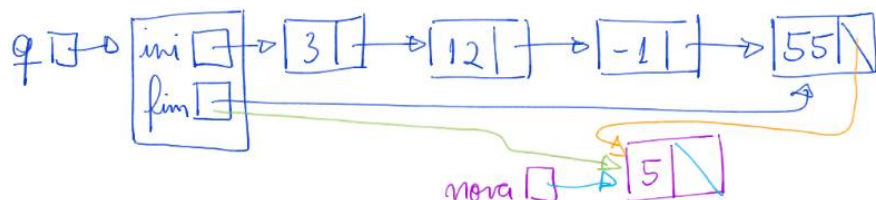
- uma importante decisão de projeto deve ser tomada.
- Teremos de manter um apontador para o início da lista
 - e outro para seu último elemento,
- Isso porque, na fila as operações de inserção e remoção
 - mexem em pontas opostas da estrutura.
- Quiz3: Daí vem a pergunta:
 - Dado que nós inserimos no final e removemos do início da fila,
 - em que ponta da lista devemos inserir
 - e em que ponta devemos remover?
 - Convém adotar o início da lista como o início da fila
 - e o final da lista como o final da fila.



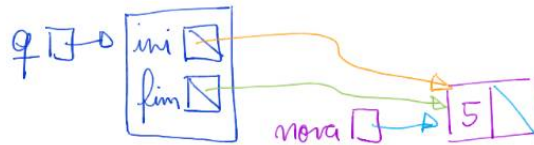
- Caso contrário, a remoção ficaria muito custosa. Por que?

Exemplo de inserção do 5:

- Se a lista não está vazia, precisamos atualizar
 - o apontador do último elemento e o apontador fim.

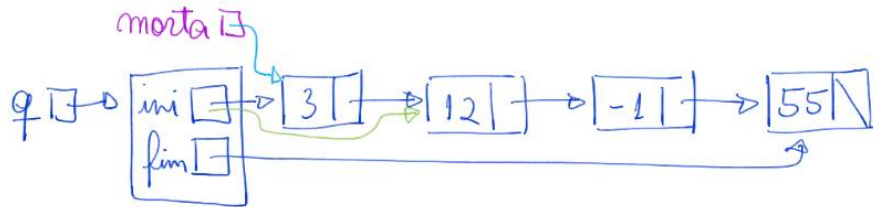


- Se a lista está vazia, precisamos atualizar
 - o apontador ini e o apontador fim.

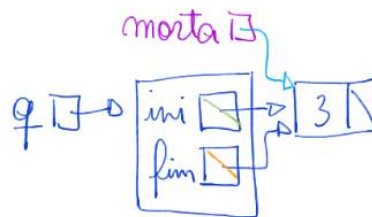


Exemplo de remoção:

- Se a lista tem vários elementos, só precisamos atualizar o apontador ini.



- Se a lista tem apenas um elemento, precisamos atualizar
 - o apontador ini e o apontador fim.



Fila vazia

- apontador ini == NULL ou apontador fim == NULL

Fila cheia

- só ocorre se a memória do programa acabar.

Tamanho

- Necessário percorrer a lista contando,
 - ou manter uma variável tam auxiliar
 - que é atualizada nas inserções e remoções.

Note que as operações de manipulação da fila

- levam tempo constante, i.e., $O(1)$,
 - com a possível exceção do cálculo do tamanho.

Biblioteca para fila implementada com lista ligada

A seguir temos a implementação da biblioteca usando lista ligada.

```
#include <stdio.h>
#include <stdlib.h>

#include "fila.h"
```

```

typedef struct celula {
    type conteudo;
    struct celula *prox;
} Celula;

struct fila {
    Celula *ini;
    Celula *fim;
    int tam;
};

Fila *criaFila() {
    Fila *q;
    q = (Fila *)malloc(sizeof(Fila));
    q->ini = NULL;
    q->fim = NULL;
    q->tam = 0;
    return q;
}

void insereFila(Fila *q, type x) {
    Celula *nova;
    nova = (Celula *)malloc(sizeof(Celula));
    nova->conteudo = x;
    nova->prox = NULL; // inserção no final da lista
    if (q->fim == NULL) { // fila vazia
        q->ini = nova;
        q->fim = nova;
    }
    else {
        q->fim->prox = nova;
        q->fim = nova;
    }
    (q->tam)++;
}

```

```

type removeFila(Fila *q) {
    type x;
    Celula *morta;
    morta = q->ini;
    x = morta->conteudo;
    q->ini = morta->prox; // remove do início da lista
    if (q->ini == NULL) // fila ficou vazia
        q->fim = NULL;
    free(morta);
    (q->tam)--;
    return x;
}

int filaVazia(Fila *q) {
    return q->fim == NULL;
    // return q->ini == NULL;
    // return tam == 0;
}

int filaCheia(Fila *q) {
    Celula *p;
    p = malloc(sizeof(Celula)); // versão segura
    if (p == NULL)
        return 1;
    free(p);
    return 0;
}

void imprimeFila(Fila *q) {
    Celula *p;
    p = q->ini;
    while (p != NULL) {
        printf("%c ", p->conteudo);
        p = p->prox;
    }
    printf("\n");
}

```



```

int tamFila(Fila *q) {
    // Celula *p;
    // int tam = 0;
    // p = q->ini;
    // while (p != NULL)
    // {
    //     tam++;
    //     p = p->prox;
    // }
    // return tam;
    return q->tam;
}

Fila *liberaFila(Fila *q) {
    Celula *p, *morta;
    p = q->ini;
    while (p != NULL) {
        morta = p;
        p = p->prox;
        free(morta);
    }
    free(q);
    return NULL;
}

```

Quiz4: Compare as implementações de fila em vetor e em lista ligada, segundo:

- eficiência de tempo das operações,
- uso de memória,
- limitações de tamanho.

Compilando biblioteca

Para implementar e compilar um programa que usa nossa biblioteca,

- primeiro incluímos uma chamada para ela no início do programa,

```
#include "fila.h"
```

- então compilamos a biblioteca em um programa objeto
“gcc -c fila.c” ou
“gcc -Wall -O2 -pedantic -Wno-unused-result -c fila.c”
- e, finalmente, compilamos o programa principal usando esse programa objeto
“gcc fila.o usaFila.c -o usaFila” ou
“gcc -Wall -O2 -pedantic -Wno-unused-result fila.o usaFila.c -o usaFila”

Também podemos compilar o programa principal em um programa objeto

“gcc -c usaFila.c” ou

“gcc -Wall -O2 -pedantic -Wno-unused-result -c usaFila.c”

- e então compilar os dois programas objetos no executável

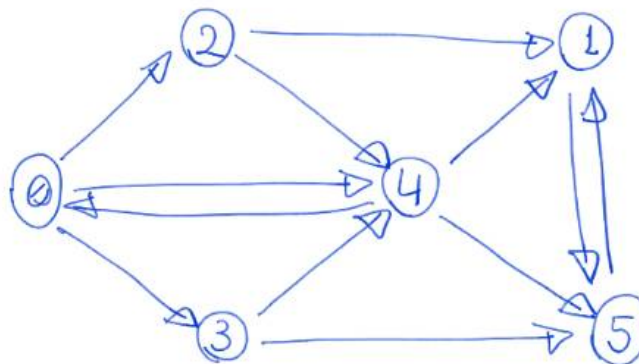
“gcc fila.o usaFila.o -o usaFila”

Ou, no extremo oposto, compilar tudo diretamente, sem usar programas objeto

“gcc fila.c usaFila.c -o usaFila” ou

“gcc -Wall -O2 -pedantic -Wno-unused-result fila.c usaFila.c -o usaFila”

Representação de redes



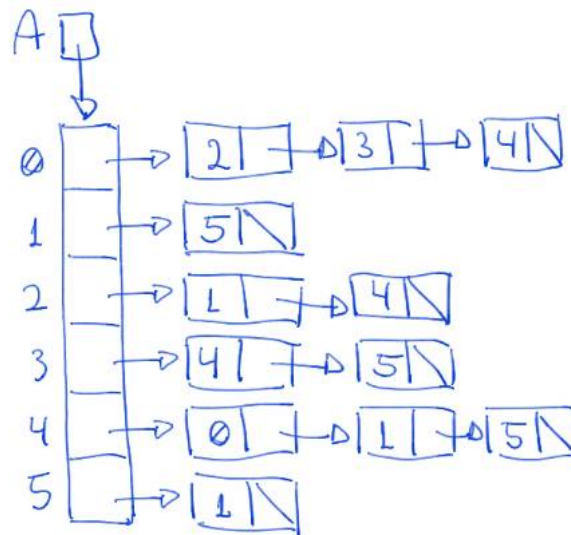
Representação da rede em uma matriz:

A

	0	1	2	3	4	5
0	0	0	1	1	1	0
1	0	0	0	0	0	1
2	0	1	0	0	1	0
3	0	0	0	0	1	1
4	1	1	0	0	0	1
5	0	1	0	0	0	0

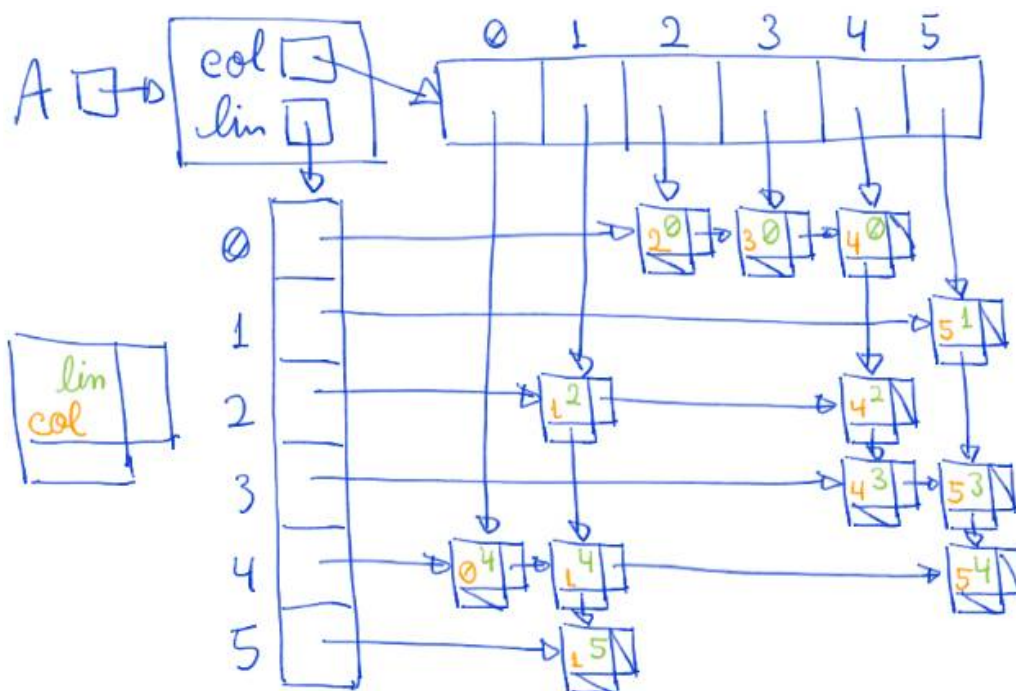
- Vantagens
 - Acessar um elemento $A[i][j]$ qualquer leva tempo constante.
 - Economia de espaço quando a rede é densa,
 - pois é possível operar sobre uma matriz de bits.
- Desvantagens
 - Ocupa espaço proporcional a n^2 , ainda que a rede seja esparsa,
 - resultando na maioria dos elementos da matriz iguais a zero.
 - Visitar todos os nós para os quais um nó i tem conexão,
 - leva tempo proporcional a n , ainda que i tenha poucos vizinhos.
 - O mesmo vale para visitar todos os nós que tem conexão para i .

Representação da rede usando listas de adjacências:



- Vantagens
 - Economia de memória quando a rede é esparsa,
 - pois ocupa espaço proporcional a $n + m$,
 - sendo n o número de nós da rede
 - e m o número de conexões entre nós.
 - Visitar todos os nós para os quais um nó i tem conexão,
 - leva tempo proporcional ao número de vizinhos de i .
- Desvantagens
 - Verificar se um nó i tem conexão para um nó j
 - leva tempo linear no número de vizinhos do nó i .
 - Quando a rede é densa, a ordem de grandeza
 - tanto da memória quanto do tempo serão quadráticos.
 - A memória ocupada por conexão é maior que na matriz.
 - Verificar quais nós tem conexão para um nó j
 - exige percorrer todas as listas.
 - Para contornar essa limitação, podemos usar listas ortogonais.

Representação da rede em listas ortogonais:



Aplicação de fila para cálculo de distâncias

Código que opera com listas de adjacências:

```
#include <stdio.h>
#include <stdlib.h>

#include "fila.h"

typedef struct celula {
    int indice;
    struct celula *prox;
} Celula;

int main(int argc, char *argv[]) {
    Celula **Rede, *p;
    int i, j, aux, n, *dist;

    printf("Digite o numero de cidades.\n");
    scanf("%d", &n);
    Rede = (Celula **)malloc(n * sizeof(Celula *));
    for (i = 0; i < n; i++)
        Rede[i] = NULL;

    // lendo a matriz e convertendo para listas de adjacências
    printf("Digite a matriz da rede.\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
            scanf("%d", &aux);
            if (aux != 0) {
                p = (Celula *)malloc(sizeof(Celula));
                p->indice = j;
                p->prox = Rede[i];
                Rede[i] = p;
            }
        }
}
```

```

// imprimindo a rede como lista de adjacências
printf("Imprimindo a rede lida como listas de adjacencias:\n");
for (i = 0; i < n; i++) {
    printf("%d: ", i);
    p = Rede[i];
    while (p != NULL) {
        printf("%d ", p->indice);
        p = p->prox;
    }
    printf("\n");
}

// dist = distancias(Rede, n, 0);
dist = distancias(Rede, n, n / 2);

// imprimindo distâncias calculadas
printf("cidades:  ");
for (i = 0; i < n; i++)
    printf("%d ", i);
printf("\n");
printf("distancias: ");
for (i = 0; i < n; i++)
    printf("%d ", dist[i]);
printf("\n");

// Liberando Rede e suas Listas
for (i = 0; i < n; i++) {
    while (Rede[i] != NULL) {
        p = Rede[i];
        Rede[i] = Rede[i]->prox;
        free(p);
    }
}

free(Rede);
free(dist);
return 0;
}

```

Implementação da função distancias operando sobre listas de adjacências:

*// A função recebe um inteiro origem, uma lista de adjacências Rede
// e o número de cidades da Rede n, com $0 \leq \text{origem} < n$. Ela devolve
// um vetor com a distância de origem até cada elemento entre 0 e
n-1.*

```
int *distancias(Celula **Rede, int n, int origem) {
    int i, corr; // auxiliar que guarda a cidade corrente
    int *dist;
    Fila *fila;
    Celula *p;

    dist = malloc(n * sizeof(int));
    /* inicializa a fila */
    fila = criaFila();
    /* inicializa todos como não encontrados, exceto pela origem */
    for (i = 0; i < n; i++)
        dist[i] = -1;
    dist[origem] = 0;
    /* colocando origem na fila */
    insereFila(fila, origem);
    /* enquanto a fila dos ativos (encontrados mas não visitados)
não estiver vazia */
    while (!filaVazia(fila)) {
        /* remova o mais antigo da fila */
        corr = removeFila(fila);
        /* para cada vizinho deste que ainda não foi encontrado */
        p = Rede[corr];
        while (p != NULL) {
            i = p->indice;
            if (dist[i] == -1) {
                /* calcule a distancia do vizinho */
                dist[i] = dist[corr] + 1;
                insereFila(fila, i); // e o coloque na fila
            }
            p = p->prox;
        }
    }
    fila = liberaFila(fila);
    return dist;
}
```

Eficiência de tempo:

- $O(n + m)$,
 - sendo n o número de nós na rede
 - e m o número de conexões entre nós.
- Isso porque, em cada iteração do laço externo do algoritmo
 - temos um nó “corrente” corr retirado da fila.
- Note que cada nó entra na fila no máximo uma vez,
 - também sendo retirado no máximo uma vez.
 - Portanto, o número de iterações do laço externo $\leq n$.
- Em cada iteração do laço interno do algoritmo,
 - é considerada uma conexão de corr com algum vizinho.
- Note que, cada conexão de corr é considerada apenas uma vez
 - e o nó corr nunca mais será “corrente”.
 - Portanto, cada conexão é considerada no máximo uma vez
 - ao longo de todas as iterações do algoritmo.
 - Assim, o número total de iterações do laço interno $\leq m$.

Eficiência de espaço:

- Fila auxiliar ocupa espaço adicional $O(n)$.
 - Matriz de entrada ocupa espaço $O(n + m)$.

Quiz5:

- Considere uma rede com 100000 cidades
 - e uma média de 10 estradas saindo de cada cidade.
- Compare a eficiência de tempo
 - do algoritmo que representa a Rede como uma matriz
 - com a eficiência do algoritmo que usa listas de adjacência.
- Faça a mesma comparação
 - em relação à eficiência de espaço dos algoritmos.