AED1 - Aula 19
Filas com implementação circular em vetor e em listas encadeadas, interfaces, listas de adjacência e ortogonais

Filas

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.

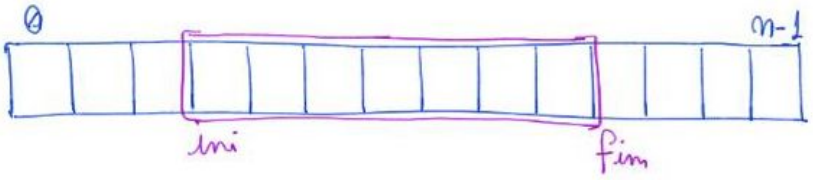
Implementação circular de fila em vetor

Uma fila q é armazenada em um vetor de tamanho n
- alocado estática ou dinamicamente.

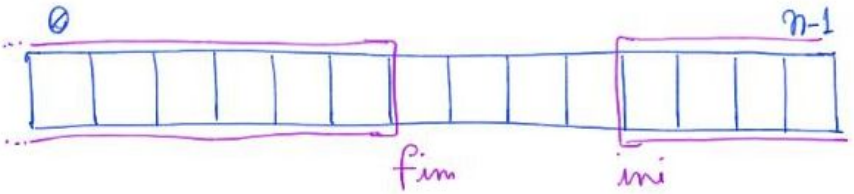
Um inteiro fim indica o final da fila,
- que é 1 a mais que a posição do último elemento e
- é a posição do próximo elemento a ser inserido.

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.

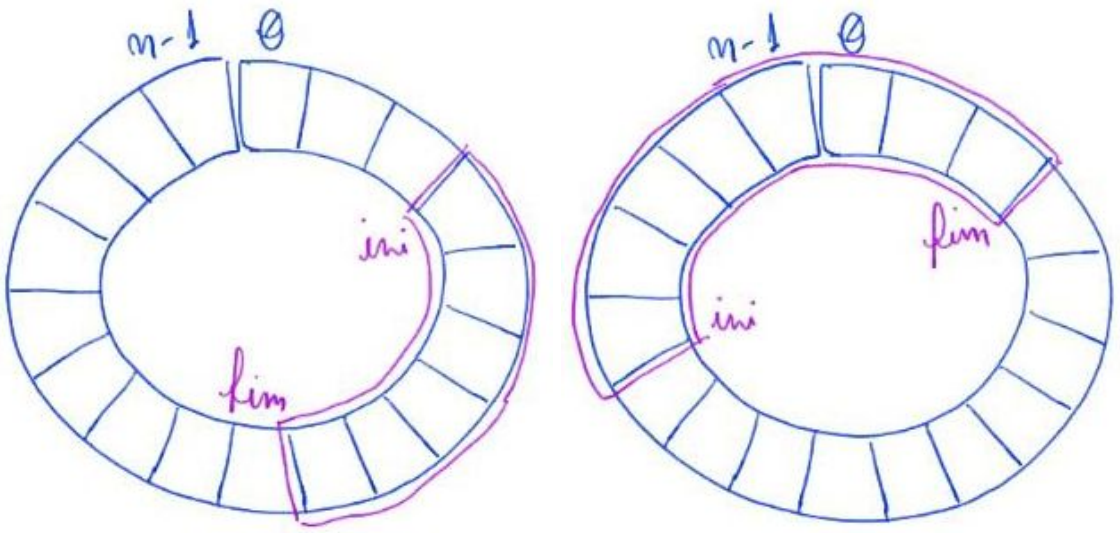
Na implementação circular,
- a fila está no subvetor v[ini .. fim - 1]



- 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

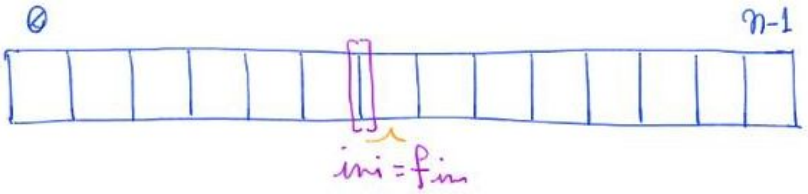
- q[fim++] = x;
- Implementação da circularidade
    - if (fim == n) fim = 0;
- Circularidade com aritmética modular
    - fim = fim % n;

Para remover um elemento e armazená-lo em x fazemos

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

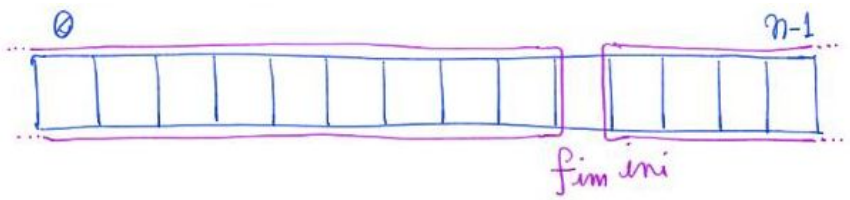
Fila vazia

- ini == fim;



Fila cheia

- fim + 1 == ini || (fim + 1 == n && ini == 0)
- Alternativa com aritmética modular
    - (fim + 1) % n == ini;



- Note que, a posição fim sempre está desocupada.
    - Isso porque precisamos diferenciar fila vazia de fila cheia.

Tamanho

- if (fim >= ini) tam = fim - ini;
- if (fim < ini) tam = (n - ini) + (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 circular usando vetor.

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

#include "fila.h"

#define N 100

struct fila
{
   type *v;
   int ini;
```

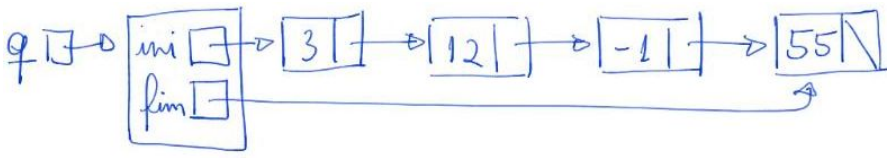
```
   int fim;
};

Fila *criaFila()
{
   int size = N;
   Fila *q;
   q = (Fila *)malloc(sizeof(Fila));
   q->v = (type *)malloc(size * sizeof(type));
   q->ini = N / 2;
   q->fim = N / 2;
   return q;
}

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

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

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

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

void imprimeFila(Fila *q)
{
   int i;
```

```
    if (q->ini <= q->fim)
        for (i = q->ini; i < q->fim; i++)
            printf("%c ", q->v[i]);
    else // q->fim < q->ini
    {
        for (i = q->ini; i < N; i++)
            printf("%c ", q->v[i]);
        for (i = 0; i < q->fim; i++)
            printf("%c ", q->v[i]);
    }
    printf("\n");
}

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

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

Implementação de fila em lista encadeada

Antes de começar a implementação,

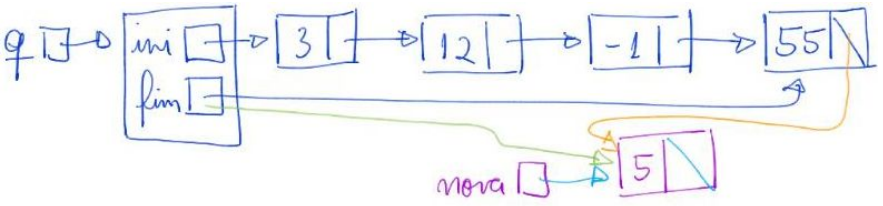
- uma importante decisão de projeto deve ser tomada.
- Teremos de manter um apontador para o início da lista
    - e outro para seu último elemento.



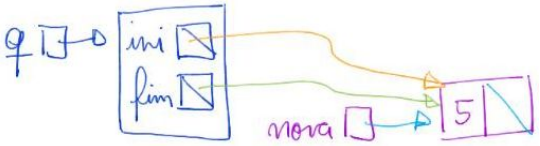
- Como na fila nós inserimos no final e removemo do início
    - convém adotar o início da lista como o início da fila
        - e o final da lista como o final da fila.
    - Caso contrário, a remoção ficaria muito custosa.
        - Por que?

Exemplo de inserção do 5:

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

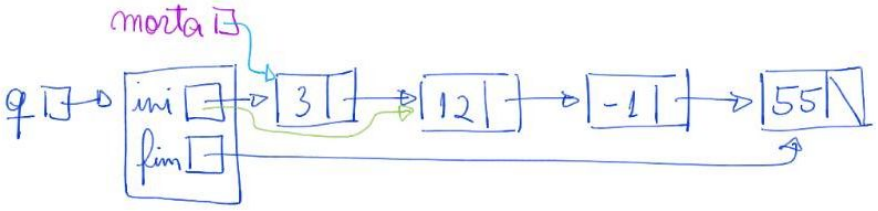


- Se a lista está vazia, precisamos atualizar
    - o apontador ini e o apontador fim.

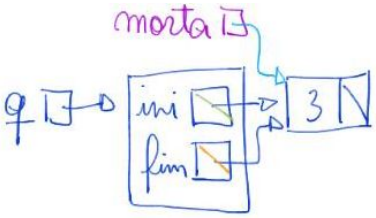


Exemplo de remoção:

- 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 encadeada

A seguir temos a implementação da biblioteca usando lista encadeada.

```c
#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;
   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;
    if (q->ini == NULL) // fila ficou vazia
        q->fim = NULL;
    free(morta);
    (q->tam)--;
    return x;
}

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

int filaCheia(Fila *q)
{
    Celula *p;
    p = malloc(sizeof(Celula));
    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;
}
```

Compare as implementações de fila

- em vetor e em lista encadeada, 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

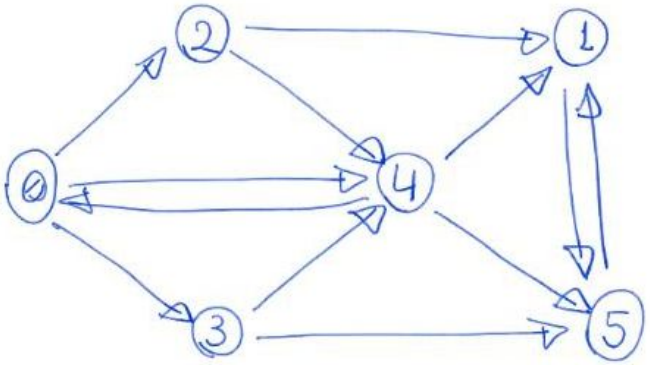
“gcc -c usaFila.c” ou

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

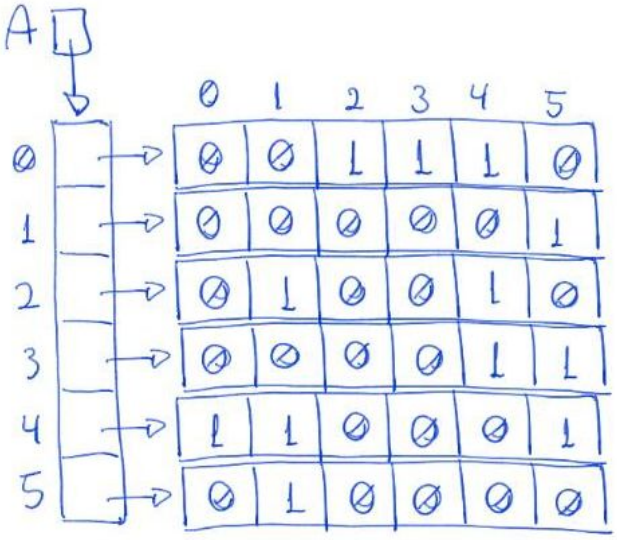
- e então compilar os dois programas objetos no executável

  “gcc fila.o usaFila.o -o usaFila”

Representação de redes

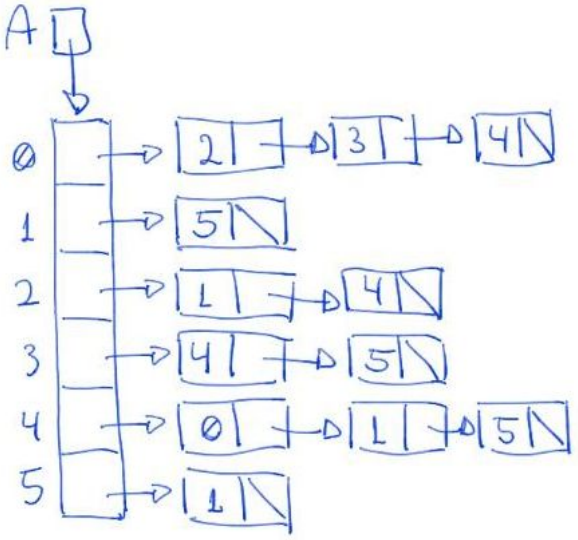


Representação da rede em uma matriz:



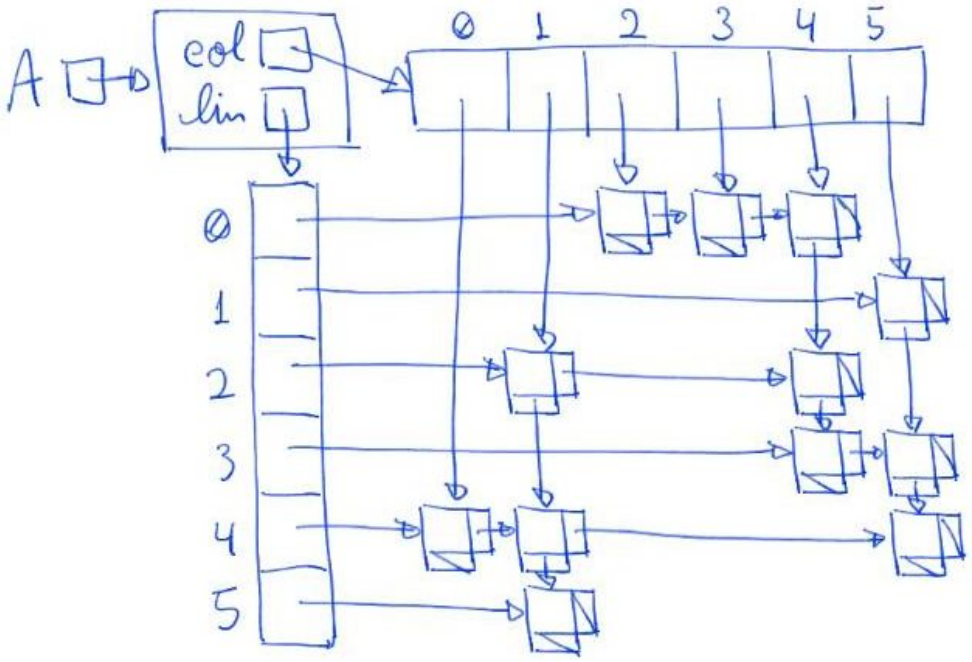
- Vantagens
  - Acessar um elemento A[i][j] qualquer leva tempo constante.
  - Economia de espaço quando a rede é densa,
    - pois é possível operar sobre uma matriz de bits.
- Desvantagens
  - Ocupa espaço proporcional a n^2, ainda que a rede seja esparsa,
    - resultando na maioria dos elementos 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 em uma lista 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:

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

#include "fila.h"

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

// A função recebe um inteiro x, uma matriz de inteiros A e a dimensão da matriz n, sendo 0 <= x < n.
// Ela devolve um vetor contendo a distância de x até cada elemento entre 0 e n-1.
int *distancias(Celula **A, int n, int x)
{
   int i, y;
   int *dist;
   Fila *q;
   Celula *p;

   dist = malloc(n * sizeof(int));
   /* inicializa a fila */
   q = criaFila();
   /* inicializa todos como não encontrados, exceto pelo x */
   for (i = 0; i < n; i++)
```

```
        dist[i] = -1;
    dist[x] = 0;
    /* colocando x na fila */
    insereFila(q, x);
    /* enquanto a fila dos ativos
    (encontrados mas não visitados)
    não estiver vazia */
    while (!filaVazia(q))
    {
        /* remova o mais antigo da fila */
        y = removeFila(q);
        /* para cada vizinho deste que ainda não foi encontrado */
        p = A[y];
        while (p != NULL)
        {
            i = p->indice;
            if (dist[i] == -1)
            {
                /* calcule a distancia do vizinho
                e o coloque na fila */
                dist[i] = dist[y] + 1;
                insereFila(q, i);
            }
            p = p->prox;
        }
    }
    q = liberaFila(q);
    return dist;
}

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

    printf("Digite o numero de cidades.\n");
    scanf("%d", &n);
    A = (Celula **)malloc(n * sizeof(Celula *));
    for (i = 0; i < n; i++)
        A[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 = A[i];
                A[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 = A[i];
        while (p != NULL)
        {
            printf("%d ", p->indice);
            p = p->prox;
        }
        printf("\n");
    }
    // dist = distancias(A, n, 0);
    dist = distancias(A, 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 A e suas listas
    for (i = 0; i < n; i++)
    {
        while (A[i] != NULL)
        {
            p = A[i];
            A[i] = A[i]->prox;
            free(p);
        }
    }
    free(A);
    free(dist);
    return 0;
}
```

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” y 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 <= n.
- Em cada iteração do laço interno do algoritmo,
    - é considerada uma conexão de y com algum vizinho.
- Note que, cada conexão de y é considerada apenas uma vez
    - e o nó y 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 <= m.

Eficiência de espaço:

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