

AED1 - Aula 16

Fila implementada em vetor, interfaces, cálculo de distâncias

Filas

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

- ou seja, uma sequência em que elementos podem ser removidos e inseridos,
- mas que possui regras bem específicas de funcionamento.

Em particular, as seguintes regras devem ser obedecidas:

- uma remoção sempre remove o elemento do início da sequência,
- uma inserção sempre insere o elemento no fim da sequência.

Costumamos resumir o comportamento de uma fila na frase

- o primeiro a entrar é o primeiro a sair.
- Por isso, filas também são conhecidas por FIFO,
 - acrônimo do inglês First-In-First-Out.

Implementação de fila usando vetor

Uma fila q é armazenada em um vetor de tamanho n

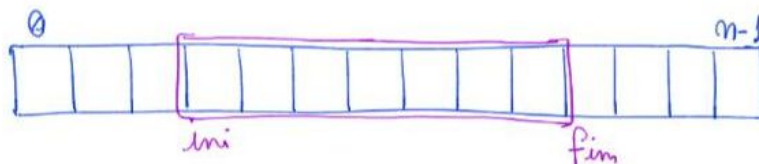
- alocado estática ou dinamicamente.

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.

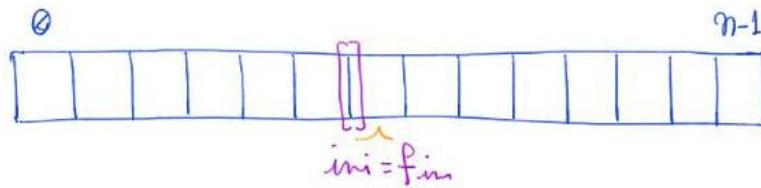
Um inteiro ini indica o início da fila,

- que é a posição do primeiro elemento e
- é a posição do próximo elemento a ser removido.

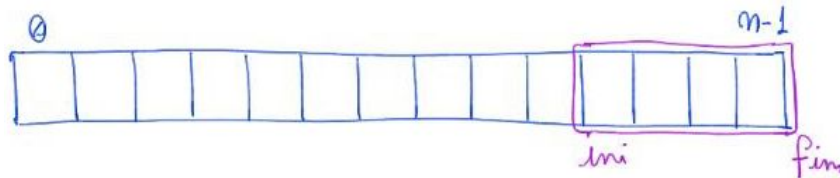


Note que $(fim - ini)$ corresponde ao número de elementos presentes na fila,

- e que $0 \leq ini \leq fim \leq n$.
- Assim,
 - se $fim - ini = 0$, ou seja, $fim = ini$, a fila está vazia



- se $\text{fim} = n$ a fila está cheia.



Para inserir um elemento x fazemos

- $q[\text{fim}++] = x;$
- que corresponde a
 - $q[\text{fim}] = x; \text{fim} = \text{fim} + 1;$
- Note que, esta operação não é segura se a fila estiver cheia,
 - i.e., se $\text{fim} = n$.

Para remover um elemento e armazená-lo em x fazemos

- $x = q[\text{ini}++];$
- que corresponde a
 - $x = q[\text{ini}]; \text{ini} = \text{ini} + 1;$
- Note que, esta operação não é segura se a fila estiver vazia,
 - i.e., se $\text{ini} = \text{fim}$.

Note que, as operações de manipulação da fila

- levam tempo constante em relação ao tamanho da mesma, i.e., $O(1)$.

Se o número de elementos crescer muito, a fila pode ficar cheia.

- Neste caso, uma alternativa é redimensionar a fila, por exemplo,
 - alocando um vetor com o dobro do tamanho do anterior
 - e copiando todos os elementos do vetor anterior para esse novo,
 - preservando a ordem dos elementos.
- No entanto, observe que esta implementação também apresenta limitação
 - quanto ao número máximo de operações de inserção,
 - ainda que o número de elementos na fila não aumente.
 - Por que?
 - Como evitar isso sem redimensionar o vetor?

Biblioteca para fila implementada em vetor

Segue o código da interface fila.h:

```
typedef struct fila Fila;

Fila *criaFila();
void insereFila(Fila *q, char x);
char 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.

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

#include "fila.h"

#define TAM_MAX 100

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

Fila *criaFila()
{
    Fila *q;
    q = (Fila *)malloc(sizeof(Fila));
    q->vetor = (char *)malloc(TAM_MAX * sizeof(char));
    q->ini = 0;
    q->fim = 0;
    return q;
}
```

```

}

void insereFila(Fila *q, char x)
{
    q->vetor[q->fim] = x;
    (q->fim)++;
}

char removeFila(Fila *q)
{
    char x;
    x = q->vetor[q->ini];
    (q->ini)++;
    return x;
}

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

int filaCheia(Fila *q)
{
    return q->fim == TAM_MAX;
}

void imprimeFila(Fila *q)
{
    for (int i = q->ini; i < q->fim; i++)
        printf("%c ", q->vetor[i]);
    printf("\n");
}

int tamFila(Fila *q)
{

```

```

    return q->fim - q->ini;
}

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

```

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

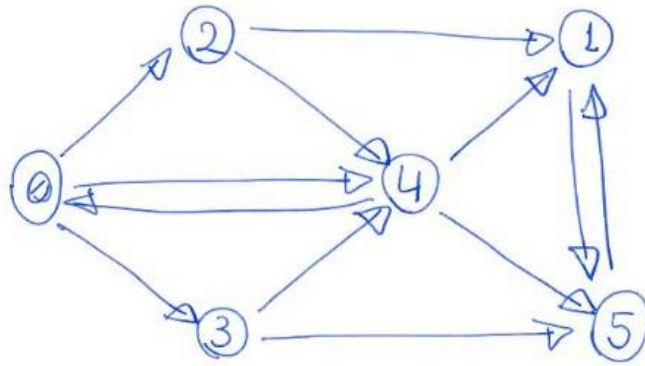
```
“gcc fila.c usaFila.c -o usaFila” ou
```

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

Aplicação de fila para cálculo de distâncias

Considere n cidades,

- numeradas de 0 a $n - 1$
- e interligadas por estradas de mão única.

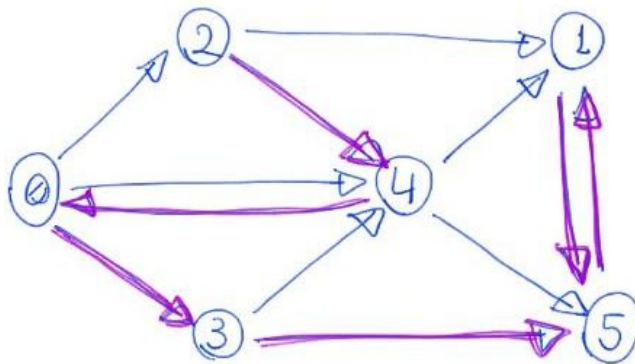


Um caminho que liga duas cidades i e j é uma sequência de cidades, tal que

- a primeira cidade é i , a última cidade é j ,
- e se cidades h e k aparecem uma seguida da outra no caminho,
 - então existe uma estrada indo de h para k .

O comprimento de um caminho

- é o número de estradas neste caminho, i.e.,
 - o número de saltos entre cidades adjacentes.
- Note que, o comprimento de um caminho
 - que contém n cidades (contando repetições)
 - é $n - 1$.

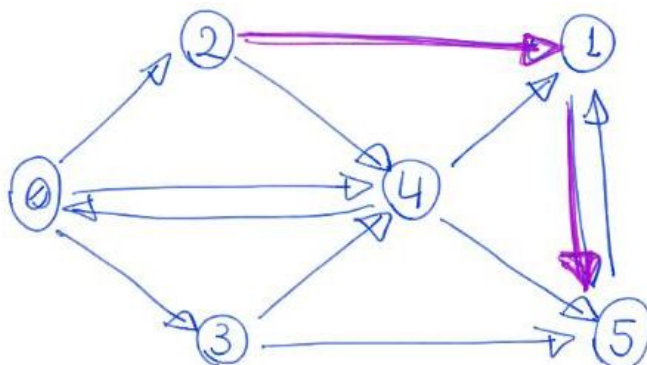


2-4-0-3-5-1-5

Comprimento = 6

A distância de uma cidade i a uma cidade j é

- o comprimento do menor caminho de i até j .



Distância de 2
até 5 = 2

- Se não existir caminho, a distância é infinita.

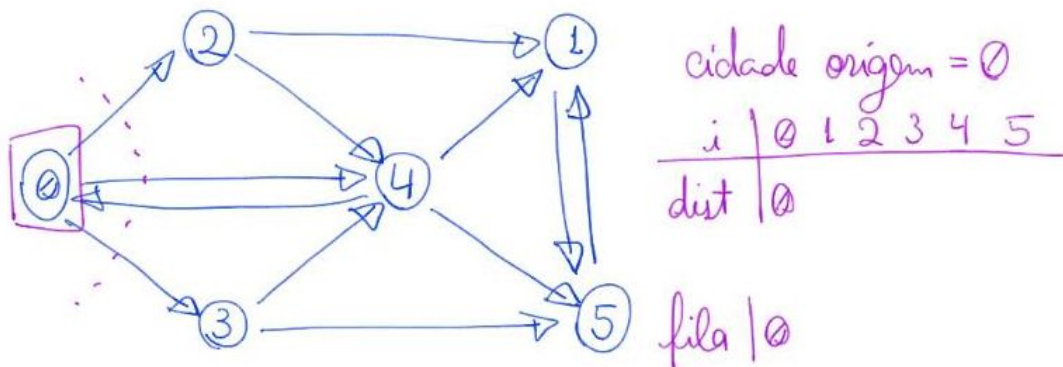
- Note que, a definição de distância já inclui ideia de minimalidade.
- Por isso, expressões como
 - “distância mínima” ou “menor distância” são pleonasmos
 - e valem dois pontos - <https://www.youtube.com/watch?v=vy43cO9cXks>

Queremos resolver o problema de

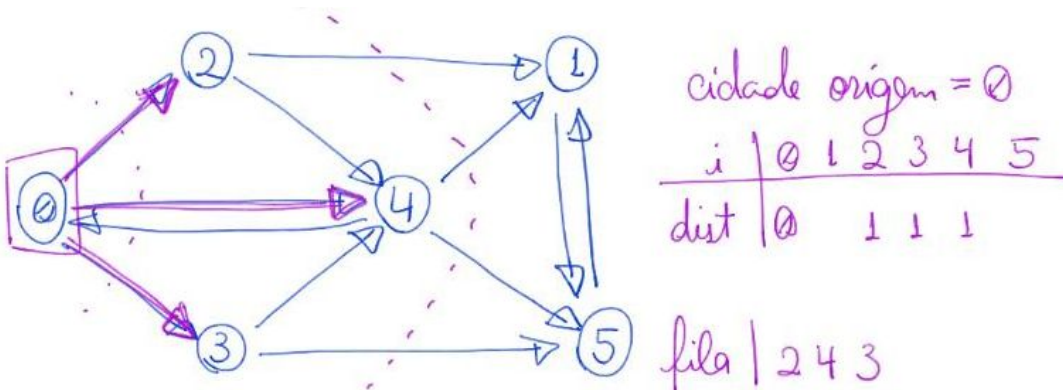
- calcular a distância de uma cidade de origem
 - até todas as demais cidades da nossa rede.
- Observe que este é problema do cálculo de distâncias não ponderado,
 - pois não existem pesos/custos associados às estradas.
- Embora essa restrição possa causar estranheza,
 - esse problema surge naturalmente
 - quando calculamos distâncias em certos tipos de redes,
 - como nas redes sociais.

Exemplo de solução:

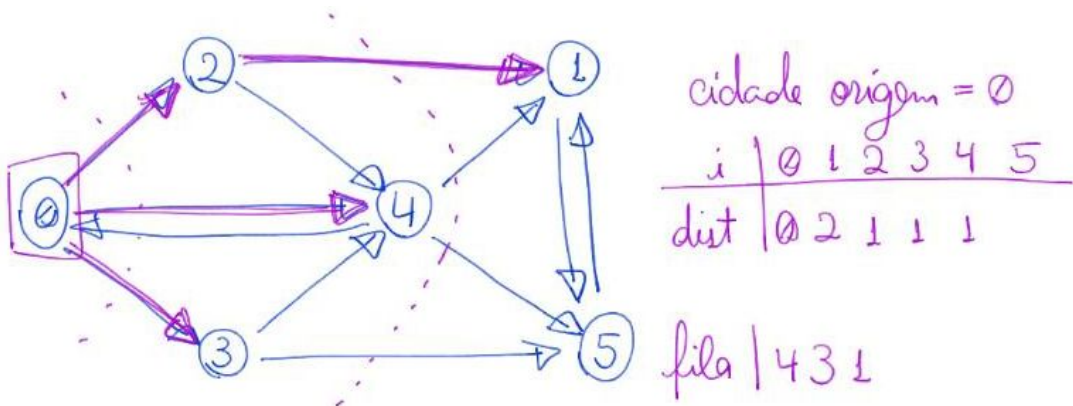
- No início apenas a cidade origem = 0 é alcançável e tem distância 0.



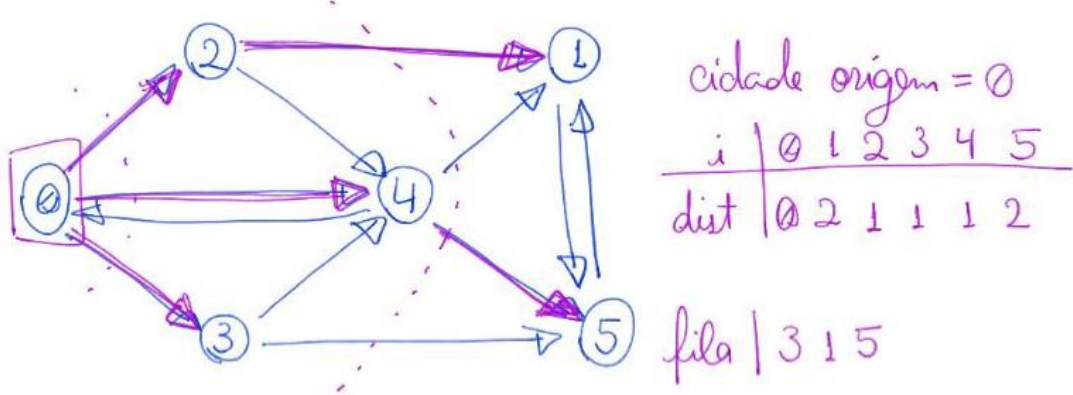
- Em cada iteração podemos encontrar novas cidades
 - e atualizar suas distâncias como
 - sendo 1 a mais que a distância de quem a encontrou.



- Observe a importância de armazenar as cidades descobertas numa fila
 - para preservar a ordem de descoberta
 - e assim calcular corretamente as distâncias.



- Por exemplo, se usássemos uma pilha, primeiro encontraríamos
 - o caminho que vai até 5 passando por 1,
 - que tem comprimento 3.



- Depois de alcançar todas as cidades podemos parar.

Representação da rede:

- temos uma matriz n x n de 0s e 1s.
- Se existe estrada de i para j então $A[i][j] = 1$
- Caso contrário $A[i][j] = 0$

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

Código:

```
// A função recebe uma matriz de inteiros Rede de dimensão n e um
// inteiro origem, sendo  $0 \leq \text{origem} < n$ . Ela devolve um vetor
// contendo a distância de origem até cada elemento entre 0 e n-1.
int *distancias(int **Rede, int n, int origem)
{
    int i, corr; // auxiliar que guarda a cidade corrente
    int *dist;
    int *fila;
    int ini, fim;

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

```

        fila[fim++] = i;
    }
}
free(fila);
return dist;
}

```

Intuitivamente, o algoritmo calcula corretamente as distâncias, pois

- ao retirar uma cidade da fila (que chamaremos de cidade corrente),
 - todos as cidades mais próximas da origem que ela já foram analisadas
 - e as distâncias dos vizinhos destas já foram atualizadas.
- Assim, se a cidade corrente tiver vizinhos ainda não alcançados,
 - ela é a cidade mais próxima da origem que os alcança
 - e, portanto, um caminho mínimo até eles passa por ela.

Eficiência de tempo:

- $O(n^2)$, sendo n o número de cidades.
- 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,
 - verificamos todos as n cidades,
 - para encontrar as vizinhas de corr.
- Assim, para cada uma das n iterações do laço externo
 - temos da ordem de n iterações do laço interno.

Eficiência de espaço:

- Fila auxiliar ocupa espaço adicional da ordem de n , i.e., $O(n)$.
- No entanto, matriz de entrada ocupa espaço $O(n^2)$.
 - Será que isso é necessário?
 - Considere o cenário citado das redes sociais,
 - nas quais temos bilhões de “cidades”,
 - mas cada “cidade” tem poucas “vizinhas”.

Invariante e corretude (opcional):

- Dizemos que uma cidade i é encontrada quando ela é colocada na fila.
 - Isso porque, i só é colocada na fila se o algoritmo
 - encontrou um caminho até i e atualizou $dist[i]$.
- Dizemos que uma cidade $corr$ é visitada depois da iteração

- em que ela foi removida da fila.
- Isso porque, nesta iteração são analisadas
 - todas as estradas que conectam corr a seus vizinhos.
- Note que, na fila estão apenas as cidades encontradas e ainda não visitadas.
- Os invariantes principais, que valem no início de cada iteração do while, são:
 1. Todas as cidades encontradas estão com a distância correta em $\text{dist}[\]$.
 2. Todas as cidades vizinhas de cidades visitadas já foram encontradas.
 3. Para algum inteiro não negativo k , temos que na fila estão
 - zero ou mais cidades à distância k da cidade origem,
 - seguidos de zero ou mais cidades à distância $k + 1$ da origem.
 4. Todas as cidades com distância $< k$ já foram visitadas
- Note que os invariantes valem trivialmente no início da primeira iteração
 - já que apenas a cidade origem foi encontrada
 - e nenhuma cidade foi visitada.
- O invariante se preserva de uma iteração para outra, pois
 - removemos da fila a cidade corr mais antiga
 - dentre as encontradas e ainda não visitadas.
 - Note que, pelo invariante 3, corr tem distância k da origem.
 - Então, verificamos cada vizinho i de corr ainda não encontrado
 - e atualizamos $\text{dist}[i] = \text{dist}[\text{corr}] + 1 = k + 1$.
 - Note que, existe um caminho da origem até i
 - passando por corr com comprimento $k + 1$.
 - Note também que, todo caminho mais curto até i
 - precisaria passar por alguma cidade
 - vizinha de i que tenha distância $< k$.
 - Pelo invariante 4, sabemos que estas cidades já foram visitadas.
 - Pelo invariante 2, temos que os vizinhos destas já foram encontrados.
 - Como i ainda não havia sido encontrado,
 - temos que ele não é vizinho de uma cidade com distância $< k$,
 - e que não existe caminho até i com comprimento $< k + 1$.
 - Portanto, $k + 1$ é a distância correta para i ,
 - por ser o comprimento de um caminho mínimo até i .
- Ao final das iterações, temos que $\text{dist}[\]$ possui
 - o valor correto da distância de todas as cidades alcançáveis.
 - Se $\text{dist}[i] = -1$ então não existe caminho da origem até i .