# Algoritmos e Estrutura de Dados I - Listas de Exercícios

Jorge Augusto Salgado Salhani

Agosto, 2022

## 1 Lista 3

### 1.1 O que é e para que serve uma pilha?

Podemos definir uma pilha em dois níveis. No nível lógico, consiste em um tipo abstrato de dado (TAD) capaz de armazenar elementos (ou itens) que apresentam prioridade de acesso quanto maior for o tempo desde que sua inserção no conjunto tenha ocorrido. Também posto na forma *FIFO: First in, First OUT*.

Em nível de operações e estrutura, temos as seguintes relações (utilizamos aqui uma lista linear dinâmica e sequencial para facilitar a descrição, mas vale lembrar que caso encadeada e/ou circular, a definição seria alterada.)

**Estrutura Protocolo**

- *FIFO: First in, First Out*

**Definição**

- ITEM itens[MAX]
- int tamanho
- int inicio
- int fim

**Operações**

- **fila_criar()**

  - pré-condição: -
  - pós-condição: fila criada, não nula

- **fila_apagar()**
  - pré-condição: fila existe
  - pós-condição: memória da fila liberada. Endereço nulo

- **fila_inserir()**
  - pré-condição: fila existe e fila não cheia
  - pós-condição: fila com novo elemento na posição final. Seu tamanho é incrementado

- **fila_remover()**
  - pré-condição: fila existe e fila não vazia
  - pós-condição: fila sem o elemento na posição inicial. Elemento que o precede, agora é o início. Seu tamanho é decrementado

- **fila_frente()**
  - pré-condição: fila existe e fila não vazia
  - pós-condição: fila inalterada

- **fila_imprimir()**
  - pré-condição: fila existe e fila não vazia
  - pós-condição: fila inalterada

## 1.2 Em quais situações uma fila pode ser utilizada?

Filas são utilizadas quando processos mais antigos devem receber prioridade. Por exemplo, podemos citar sistemas de priorização de execução de tarefas, onde tarefas são requisitadas e, por não ser possível executá-las simultaneamente, elas compõe uma fila de espera, onde a primeira requisição deve ser executada antes da última requisição.

Nesta classe de processamento de múltiplas requisições podemos citar sistemas de entrada *i/o* de digitação e inserção em arquivos de texto e também requisições para servidores em nuvem.

**1.3 Desenvolva uma função (com parâmetros) para testar se uma fila F1 tem mais elementos do que uma fila F2 (não se esqueça de mexer nas filas apenas através de seus operadores primitivos). Use fila encadeada e dinâmica.**

```
// ====== Arquivo fila.h ======
#ifndef FILA_H
    #define FILA_H
    #include <stdbool.h>

    typedef struct no_ NO;
    struct no_ {
        ITEM* item;
        NO* no_anterior;
    };

    typedef struct fila_ FILA;

    // Operacoes padrao omitidas.
    // Destaque para as que serao utilizadas
    bool fila_comparar_tamanho(FILA* fila1, FILA* fila2);
#endif

// ====== Arquivo fila.c ======
#include <stdio.h>
#include <stdlib.h>
#include "fila.h"

struct fila_{
    NO* no_inicio;
    NO* no_fim;
};

// Supondo variavel 'tamanho' imbutida na TAD fila
bool fila_comparar_tamanho(FILA* fila1, FILA* fila2) {
    if (fila == NULL || fila == NULL) return false;
    return (fila1->tamanho > fila2->tamanho) ? true : false;
}

// Caso nao exista variavel 'tamanho'. Tambem supomos que
// todo no eh criado com no->no_anterior = NULL
bool fila_comparar_tamanho(FILA* fila1, FILA* fila2) {
```

```
    if (fila1 == NULL || fila2 == NULL) return false;
    NO* no1 = fila1->no_inicio;
    NO* no2 = fila2->no_inicio;
    while (no1->no_anterior != NULL && no2->no_anterior != NULL) {
        no1 = no1->no_anterior->no_anterior;
        no2 = no2->no_anterior->no_anterior;
    }
    if (no1 != NULL) return true; // elementos de fila2 exauridos. Ainda existem
        elementos em fila1. Logo fila1 > fila2
    return false;
}
```

### 1.4 Usando conceitos de TAD, implemente uma fila encadeada e dinâmica

Código

### 1.5 Escreva uma função para inverter os elementos de um deque alocado estaticamente.

Código