Estruturas de Dados

Aula 7 — Filas e Pilhas

Prof. Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia — Campus Feira de Santana

2026

Agenda

Filas

Pilhas

Comparação: Fila vs. Pilha

Exercícios

Filas

First In, First Out — FIFO

Fila — Conceito

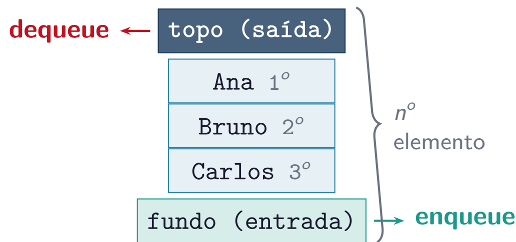
O que é uma Fila?

Estrutura de dados que obedece à ordem `FIFO` (*First In, First Out*): o primeiro a entrar é o primeiro a sair. Implementada com **lista duplamente encadeada** para garantir $O(1)$ em ambas as operações.

Operações fundamentais

Enfileirar (*enqueue*) — insere no **fundo**
Desenfileirar (*dequeue*) — remove do **topo**

Analogia: fila de banco



Usos práticos

- Filas de processos em **SO's**
- Filas de pacotes em **Redes**
- Impressoras, buffers, chamadas de API
- **BFS** — busca em largura em grafos

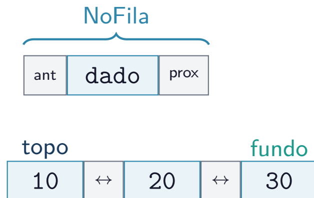
Fila — Struct em C

NoFila.c

```c
typedef struct NoFila{
  int  conteudo;
  struct NoFila *prox;//para o
      fundo
  struct NoFila *ant;//para o topo
}NoFila;
typedef struct {
  NoFila *topo;//saída
  NoFila *fundo;//entrada
  int n;
}Fila;
void inicFila(Fila *F) {
  F->topo  = NULL;
  F->fundo = NULL;
  F->n = 0;
}
```

Representação do nó

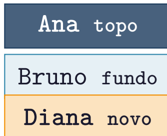
Cada nó guarda **dois ponteiros**: `prox` (em direção ao fundo) e `ant` (em direção ao topo).

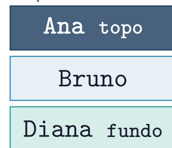


enfileirar.c — $O(1)$

```c
//Insere no FUNDO da fila
void enfileirar(Fila *F,int val){
  NoFila *novo =(NoFila*)malloc(
      sizeof(NoFila));
  novo->conteudo = val;
  novo->prox = NULL;
  novo->ant  = NULL;
  if(F->fundo==NULL){//vazia
    F->topo  = novo;
    F->fundo = novo;
  }else {
    novo->prox    = F->fundo;
    F->fundo->ant = novo;
    F->fundo      = novo;
  }
  F->n++;
}
```

Enfileirando “Diana” em {Ana, Bruno}





desenfileirar.c — $O(1)$

```c
//Remove do TOPO da fila
int desenfileirar(Fila *F) {
  if (F->topo == NULL){
    printf("Fila vazia!\n");
    return -1;
  }
  NoFila *alvo = F->topo;
  int val = alvo->conteudo;
  F->topo = alvo->ant;
  if (F->topo!=NULL)
    F->topo->prox = NULL;
  else F->fundo=NULL;//vazia
  free(alvo);
  F->n--;
  return val;
}
```

Desenfileirando {Ana, Bruno, Carlos}

antes:

Ana topo (alvo)
Bruno
Carlos fundo

free(alvo)
Ana removida

depois:

Bruno topo
Carlos fundo

▪ Retorno

Retorna o **valor** removido, ou `-1` se a fila estiver vazia.

Fila — Buscar e Imprimir

buscarFila.c

```c
NoFila* buscarFila(Fila *F,int val){
  NoFila *p = F->topo;
  while (p != NULL) {
    if (p->conteudo == val)
        return p; //achou
    p = p->ant; //topo->fundo
  }
  return NULL;
}
```

imprimirFila.c

```c
void imprimirFila(Fila *F){
  if (F->topo == NULL) {
    printf("[vazia]\n");
    return;}
  NoFila *p = F->topo;
  printf("topo->[");
  while (p != NULL){
    printf("%d",p->conteudo);
    if (p->ant != NULL)
        printf(",");
    p = p->ant;}
  printf("]<-fundo\n");}
```

Complexidade

Busca: $O(N)$ — percorre do topo ao fundo. Não há atalho sem estrutura auxiliar.

Saída

```
topo->[10,20,30]<-fundo
```

Fila — Programa Principal

main_fila.c

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    Fila F;
    inicFila(&F);
    enfileirar(&F, 10);
    enfileirar(&F, 20);
    enfileirar(&F, 30);
    printf("Após enfileirar:\n");
    imprimirFila(&F);
    int v = desenfileirar(&F);
    printf("Removeu: %d\n", v);
    printf("Fila restante:\n");
    imprimirFila(&F);
    while (F.n > 0) //libera fila
        desenfileirar(&F);
    return 0;}
```

Saída esperada

```
Após enfileirar:
topo->[10,20,30]<-fundo
Removeu: 10
Fila restante:
topo->[20,30]<-fundo
```

Complexidades

Operação	Complexidade
Enfileirar	$O(1)$
Desenfileirar	$O(1)$
Buscar	$O(N)$
Imprimir	$O(N)$

Pilhas

Last In, First Out — LIFO

Pilha — Conceito

O que é uma Pilha?

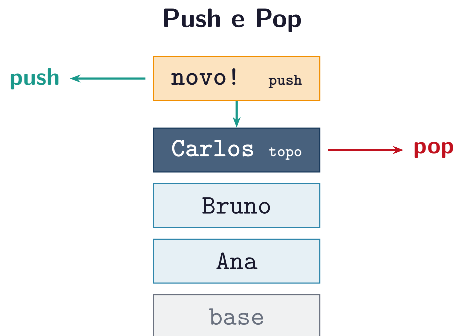
Estrutura de dados **LIFO** (*Last In, First Out*): o último a entrar é o primeiro a sair. Todas as operações ocorrem no **topo** — mais simples do que a fila.

Analogias

- **Pilha de pratos** — retira-se sempre do topo
- **Ctrl+Z** — desfaz a operação mais recente
- **Pilha de chamadas** — funções retornam em ordem inversa
- **Histórico do navegador** — botão Voltar

Operações fundamentais

Empilhar (*push*) — insere no **topo**
Desempilhar (*pop*) — remove do **topo**
Espiar (*peek*) — consulta sem remover



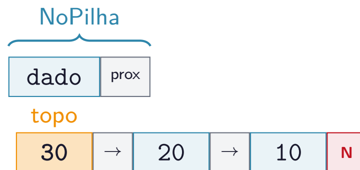
Pilha — Struct em C

NoPilha.c

```c
//Pilha usa lista SIMPLESMENTE
//encadeada: só precisa do topo
typedef struct NoPilha {
  int conteudo;
  struct NoPilha *prox; //abaixo
}NoPilha;
typedef struct {
  NoPilha *topo;
  int n;
}Pilha;
void inicPilha(Pilha *P) {
  P->topo = NULL;
  P->n = 0;
}
```

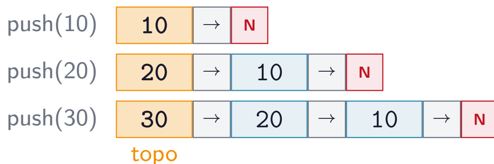
Mais simples que a Fila

Sem ponteiro `ant` e sem `fundo`: a pilha precisa apenas do **topo**. Menos memória por nó.



empilhar.c — $O(1)$

```c
/* Insere no TOPO */
void empilhar(Pilha *P, int val) {
  NoPilha *novo =
    (NoPilha*)malloc(sizeof(
        NoPilha));
  novo->conteudo = val;
  novo->prox = P->topo;
//empilha sobre o topo atual
  P->topo = novo;
  P->n++;
}
```



Invariante

Após `empilhar`, `P->topo` sempre aponta para o mais recente. Complexidade: $O(1)$.

Pilha — Desempilhar (pop) e Espiar (peek)

desempilhar.c — pop

```c
int desempilhar(Pilha *P) {
    if (P->topo == NULL) {
        printf("Pilha vazia!\n");
        return -1;
    }
    NoPilha *alvo = P->topo;
    int val = alvo->conteudo;
    P->topo = alvo->prox;
    free(alvo);
    P->n--;
    return val;
}
```

peek.c — consulta sem remover

```c
int peek(Pilha *P) {
    if (P->topo == NULL)
        return -1;
    return P->topo->conteudo;
}
```

Diferença

`pop` remove e libera a memória. `peek` apenas lê o valor do topo, sem alterar a estrutura. Útil para verificar sem destruir.

Stack overflow

Em recursão profunda, a pilha de chamadas do SO pode estourar. Pilha explícita evita esse problema.

Pilha — Imprimir e Programa Principal

imprimirPilha.c

```c
void imprimirPilha(Pilha *P){
    if (P->topo == NULL){
        printf("[vazia]\n");
        return;
    }
    NoPilha *p = P->topo;
    printf("topo->[");
    while (p != NULL){
        printf("%d", p->
            conteudo);
        if (p->prox != NULL)
            printf(",");
        p = p->prox;
    }
    printf("]<-base\n");
}
```

main_pilha.c

```c
int main(void) {
  Pilha P;
  inicPilha(&P);
  empilhar(&P, 10);
  empilhar(&P, 20);
  empilhar(&P, 30);
  imprimirPilha(&P);
  //topo->[30,20,10]<-base
  printf("pop: %d\n",desempilhar(&P));
  printf("pop: %d\n",desempilhar(&P));
  printf("peek: %d\n",peek(&P));
  imprimirPilha(&P);
  while (P.n > 0) desempilhar(&P);
  return 0;
}
```

Pilha — Aplicação: Parênteses Balanceados

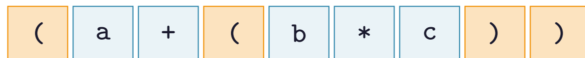
Algoritmo

1. Percorre a expressão char a char
2. Abre `( [ {` → **empilha**
3. Fecha `) ] }` → **desempilha** e verifica par
4. Ao final: pilha vazia ⇒ balanceado

Outros usos clássicos

Avaliação de expressões pós-fixas (RPN), DFS em grafos, histórico de navegação, desfazer/refazer em editores de texto.

Trace: `(a+(b*c))`



i=0: (**push**

i=3: ((**push**

i=7: (**pop (ok**

i=8: vazia **pop (ok**

BALANCEADO

Comparação

Fila vs. Pilha

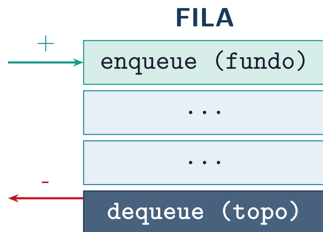
Fila vs. Pilha — Tabela Comparativa

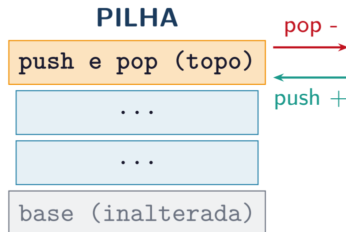
Característica	Fila (Queue)	Pilha (Stack)
Sigla	**FIFO**	**LIFO**
Inserção	No **fundo** — *enqueue* $O(1)$	No **topo** — *push* $O(1)$
Remoção	Do **topo** — *dequeue* $O(1)$	Do **topo** — *pop* $O(1)$
Struct base	Lista **duplamente** encadeada	Lista **simplesmente** encadeada
Ponteiros/nó	2 (`prox` + `ant`)	1 (`prox`)
Analogia	Fila de banco	Pilha de pratos / Ctrl+Z
Uso em TI	SO, redes, BFS	Recursão, expressões, DFS

Fila vs. Pilha — Diagrama Comparativo

▪ Mesma ideia, pontos de acesso diferentes

Ambas são listas encadeadas. A diferença está em **onde inserir e de onde remover**.





Exercícios

Pratique!

Exercícios

Exercício 1 — Fila

Simule o atendimento de uma fila de banco: enfileire 5 clientes, depois desenfileire um a um imprimindo a fila após cada operação.

Exercício 2 — Pilha

Use uma pilha para **inverter** uma string. Empilhe cada caractere; depois desempilhe tudo para reconstruir ao contrário.

Exercício 3 — Parênteses

Implemente a verificação de parênteses balanceados com pilha. Teste:

`(a+(b*c))` → balanceado

`((a+b)` → não balanceado

`[{()}]` → balanceado

Exercício 4 (desafio)

Implemente uma **fila usando duas pilhas**. Use apenas `push`/`pop` para simular `enqueue`/`dequeue`. Qual a complexidade amortizada de cada operação?

Fim da Aula 7

Dúvidas e próximos passos

Próxima Aula e Referências

Próxima aula

- **Árvores Binárias**: definição, percursos
- **BST**: inserção, busca, remoção
- Aplicações: dicionários, conjuntos ordenados

Referências

- CORMEN et al. *Introduction to Algorithms*, 4ª ed. Cap. 10.
- TENENBAUM et al. *Estruturas de Dados usando C*. Cap. 4.
- SEDGEWICK; WAYNE. *Algorithms*, 4ª ed. Cap. 1.3.

Resumo

- **Fila**: FIFO, dupla encadeada, $O(1)$ enqueue/dequeue
- **Pilha**: LIFO, simples encadeada, $O(1)$ push/pop
- Mesma base (lista), pontos de acesso distintos

Dúvidas?

carolsoko@ifba.edu.br

IFBA – Campus Feira de Santana