UFABC

Criando seu próprio tipo de dado

Prof. Fabrício Olivetti de França

Fração

Vamos criar uma biblioteca de funções para trabalhar com frações.

Uma fração é composta por numerador e denominador.

UFABC

Fração

```
int adiciona_fracao( int x_num, int x_den, int y_num, int
y_den )
{
    int z_num = x_num*y_den + y_num*x_den;
    int z_den = x_den*y_den;
    return ???;
}
```

UFABC

Fração

```
void adiciona_fracao( int x_num, int x_den, int y_num, int
y_den, int *z_num, int *z_den )
{
    *z_num = x_num*y_den + y_num*x_den;
    *z_den = x_den*y_den;
}
```

UFABC

DESESTRUTURADO!!!

Deve existir uma forma melhor de criar um novo tipo de dado.

UFABC

Struct

Em C, structs permitem definir tipos de dados como uma composição dos tipos vistos até agora.

```
struct nome {
    tipo1 var1;
    tipo2 var2;
};
```

UFABC

Fração

```
struct fracao {
   int num;
   int den;
};
```

UFABC

Fração

```
struct fracao adiciona_fracao( struct fracao x, struct fracao
y )
{
    struct fracao z;
    z.num = x.num*y_den + y.num*x_den;
    z.den = x.den*y.den;
    return z;
}
```

UFABC

Mas tenho que digitar muito!

O tipo “struct fracao” pode não ser muito prático para digitar diversas vezes, e não ajuda na organização do código.

UFABC

typedef

```
typedef struct fracao {
    int num;
    int den;
} fracao;

fracao x;
x.num = 1;
X.den = 2;
```

UFABC

typedef

A instrução **typedef** pode ser utilizada para definir quaisquer tipos:

```
typedef unsigned int uint;
uint x;
```

UFABC

Exercício 01
Números Complexos

Vamos criar uma biblioteca de funções para trabalhar com números complexos.

Um número complexo é composto por dois números reais, um denominado parte real e outro parte imaginária.

UFABC

Exemplo 1

Vamos converter um número decimal em binário:

$22 \rightarrow 10110$

UFABC

Convertendo para binário

Para isso temos que realizar sucessivas divisões por 2 e o resto será guardado como o dígito mais a direita:

22 / 2 = 11, resto = 0

11 / 2 = 5, resto = 1

5 / 2 = 2, resto = 1

2 / 2 = 1, resto = 0

1 / 2 = 0, resto = 1

UFABC

Convertendo para binário

Para facilitar vamos criar uma estrutura de dados composta de uma array de tamanho N e de um apontador para o último elemento dela.

Vamos chamá-la de pilha!

UFABC

Convertendo para binário

```
#define N 100

typedef struct pilha {
    int dados[N];
    int topo;
} pilha;
```

UFABC

Convertendo para binário

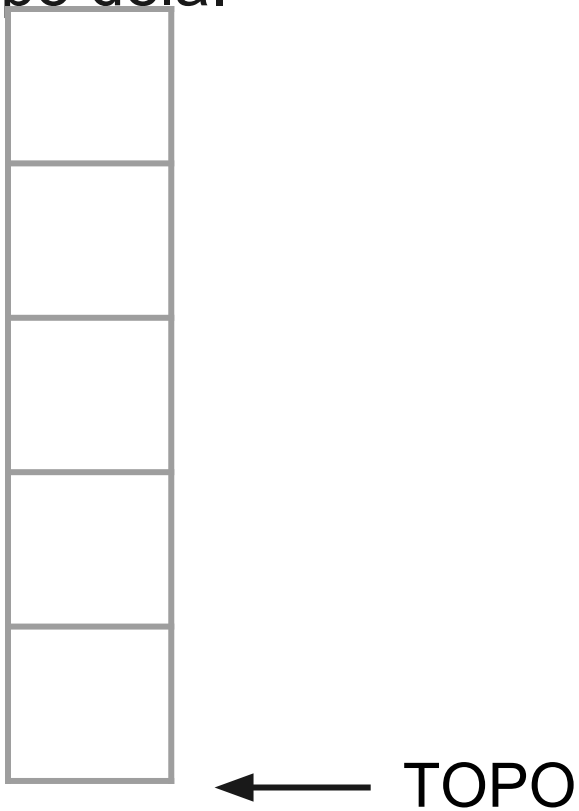
A ideia da estrutura pilha é que todo novo elemento inserido será colocado no final da array, e sua posição é definida pelo topo.

Além disso, apenas o elemento do topo pode ser recuperado, e ao recuperá-lo, apagamos ele da array (decrementando o valor de topo).

UFABC

Pilha

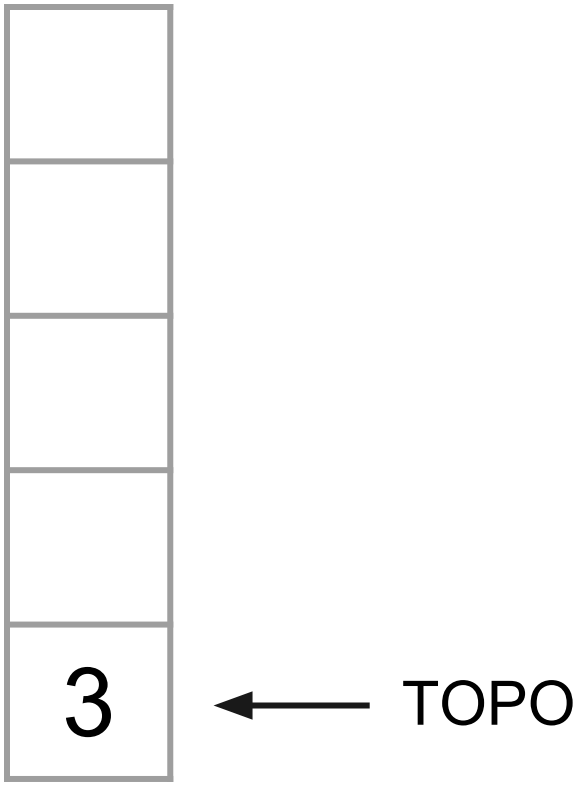
Pilha é uma estrutura de dados em que você pode inserir e remover apenas no topo dela.



UFABC

Pilha

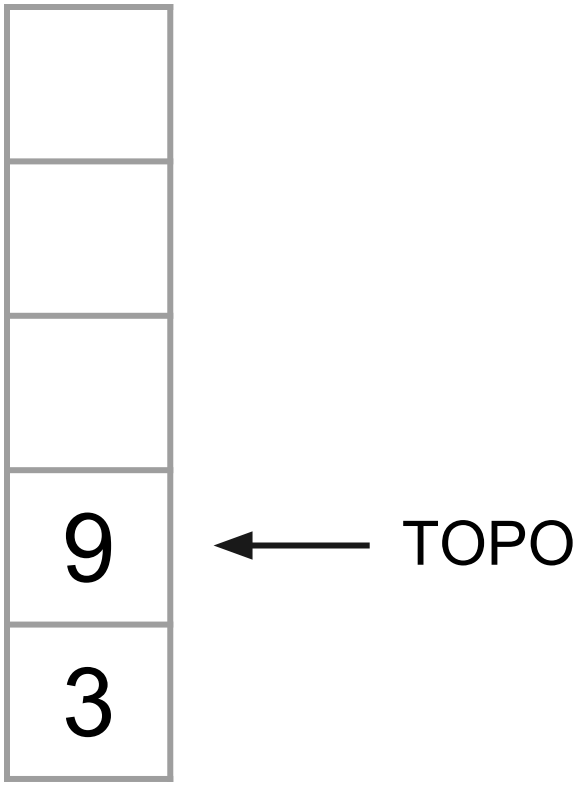
Inserir x = 3



UFABC

Pilha

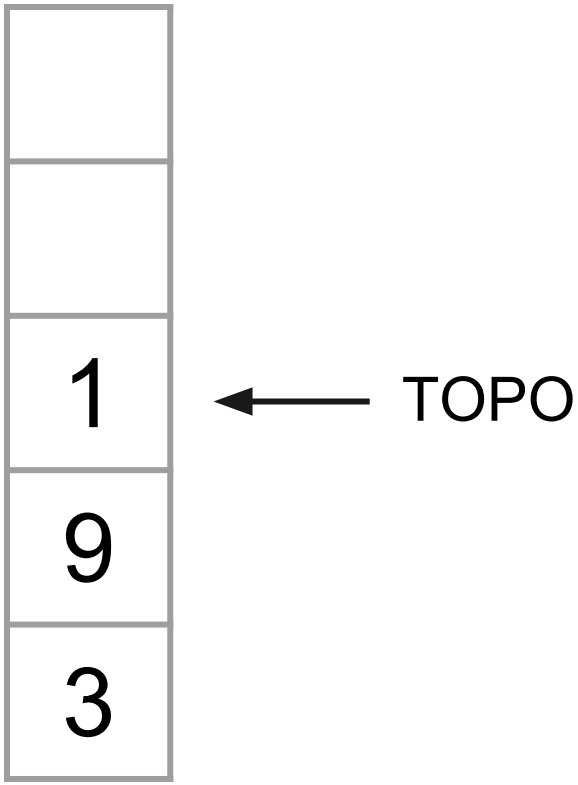
Inserir x = 9



UFABC

Pilha

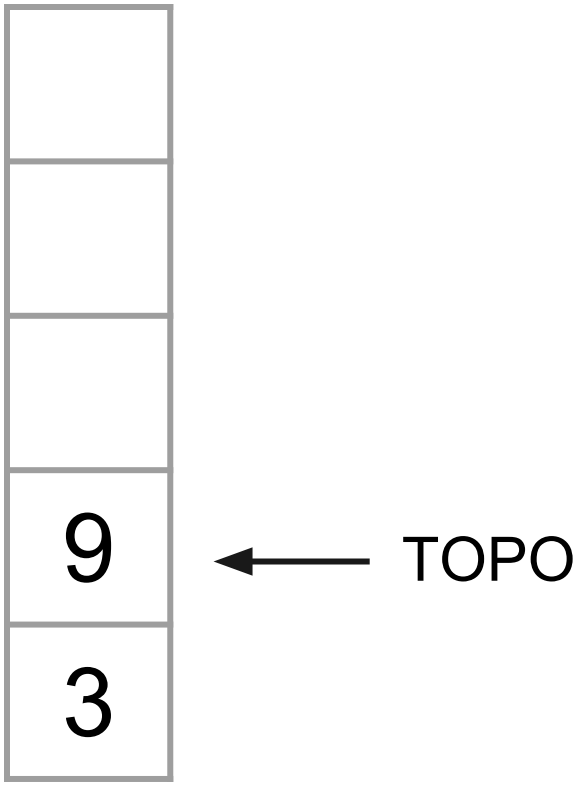
Inserir x = 1



UFABC

Pilha

Remover, x = 1



UFABC

Exemplo 1

```
int insere_pilha ( pilha * p, int x )
{
    If (p->topo + 1 < N) {
        ++(p->topo);
        p->dados[p->topo] = x;
        return 1;
    }
    return -1;
}
```

UFABC

Ponteiros de struct

Quando temos um ponteiro de struct, ao invés de usar “.” para acessar um elemento, temos que fazer:

```
struct * f;

(*f).elemento;

ou

f->elemento;
```

UFABC

Exemplo 1

```
int remove_pilha ( pilha * p )
{
    --(p->topo);
    return p->dados[p->topo + 1];
}
```

UFABC

Exemplo 1

```
pilha bin;
int x = 22;
bin.topo = -1;

while (x) {
    If ( insere_pilha( &p, x%2 )  != -1 ) {
        printf(“Espaço insuficiente na pilha!\n”);
        return -1;
    }
    x /= 2;
}
```

UFABC

Exemplo 1

```
while (bin.topo > -1) {
    printf(“%d”, remove_pilha(&p));
}

printf(“\n”);
```

UFABC

Exemplo 2 (fila)

O gerente de um banco quer tentar otimizar o número de caixas para atendimento.

Para isso ele quer encontrar o menor número de caixas **n** de tal forma que o tempo de atendimento de cada cliente não supere 15 minutos.

UFABC

Exemplo 2 (fila)

Ele sabe que a cada minuto tem:

50% de chances de entrar um novo cliente na fila
25% de chances de entrar dois novos clientes
25% de chances de um caixa ficar livre

UFABC

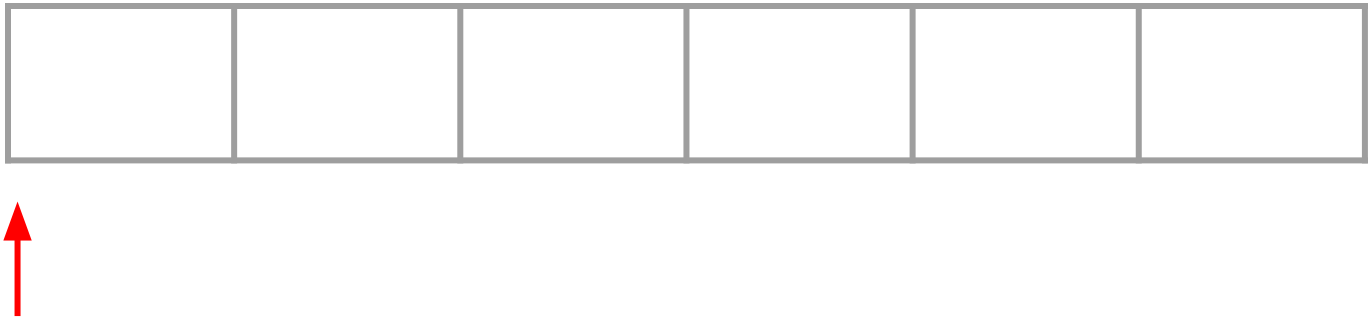
Exemplo 2 (fila)

```
fila f;
int minuto, t;

cria_fila(&f);

for (minuto=0; minuto<MAX_MINUTOS; minuto++) {
    pessoas_chegando(&f, minuto);
    t = pessoas_saindo(&f, n_caixas);
    if (minuto - t > 15) {
        return 1;
    }
}
return 0;
```

UFABC

Fila

Fila, diferente da pilha, você insere no final e remove do começo.

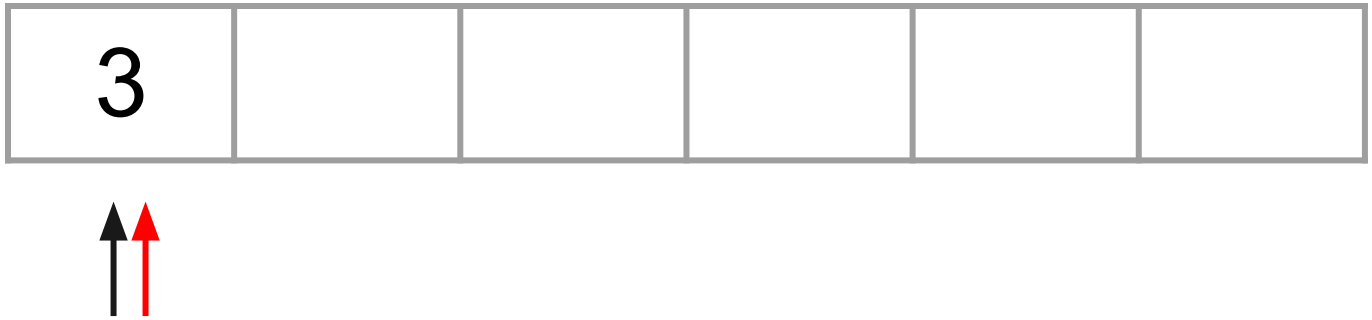


início
fim

UFABC

Fila

Insere x = 3

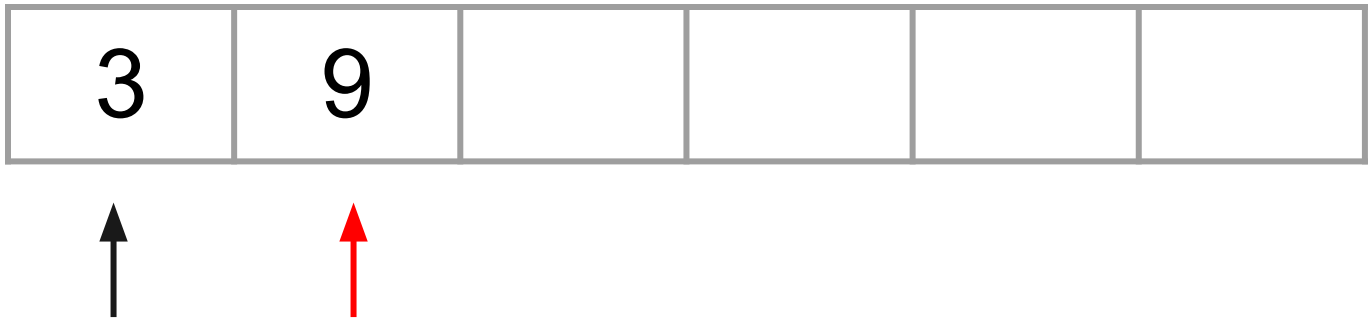


início
fim

UFABC

Fila

Insere x = 9

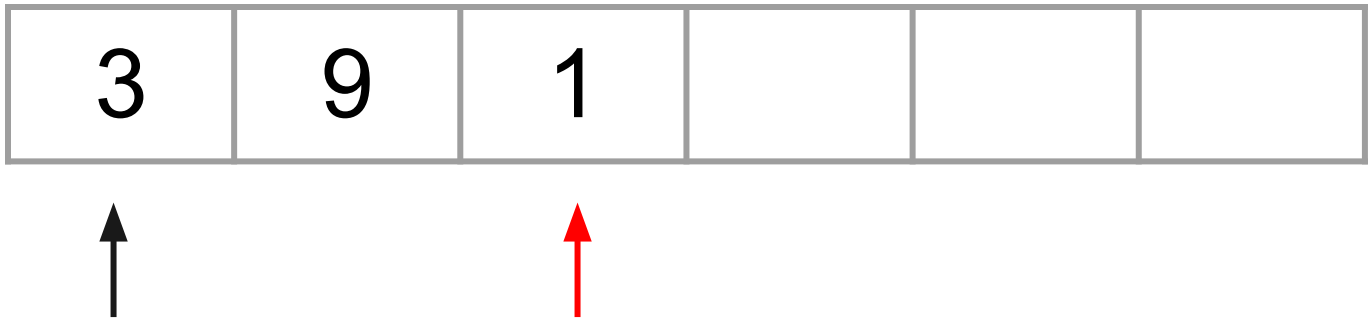


início
fim

UFABC

Fila

Insere x = 1

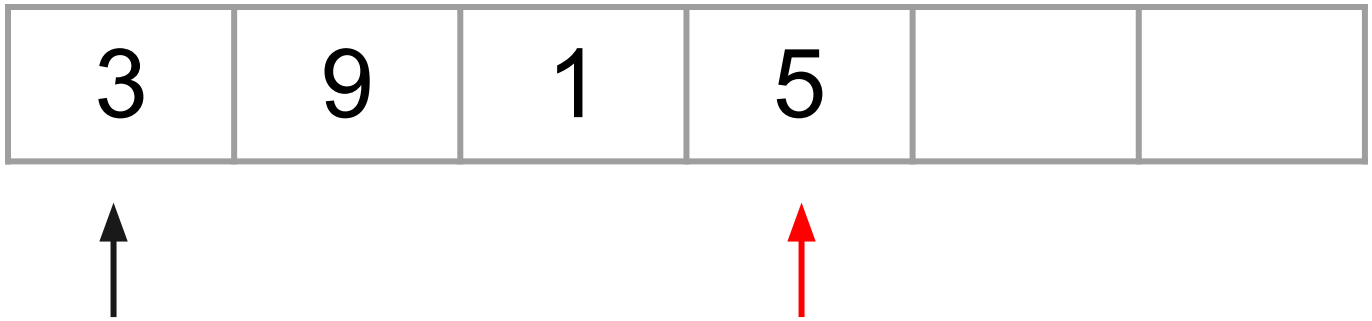


início
fim

UFABC

Fila

Insere x = 5

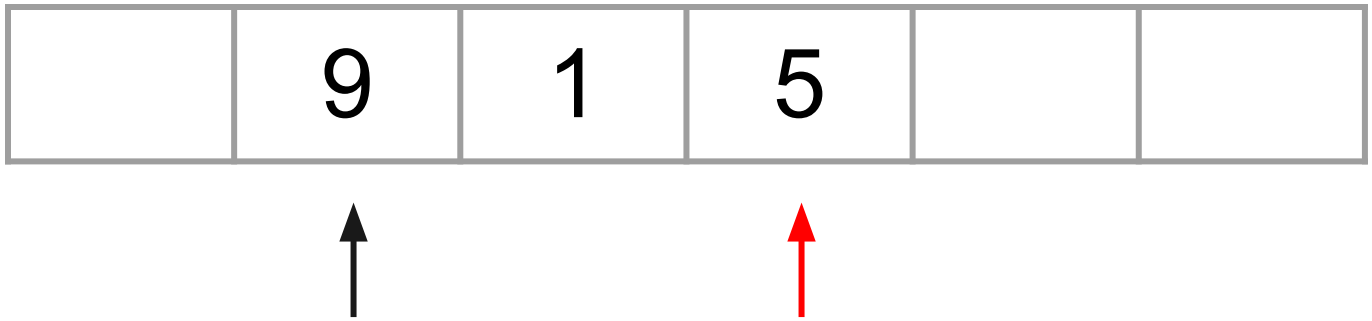


início
fim

UFABC

Fila

Remove, x = 3

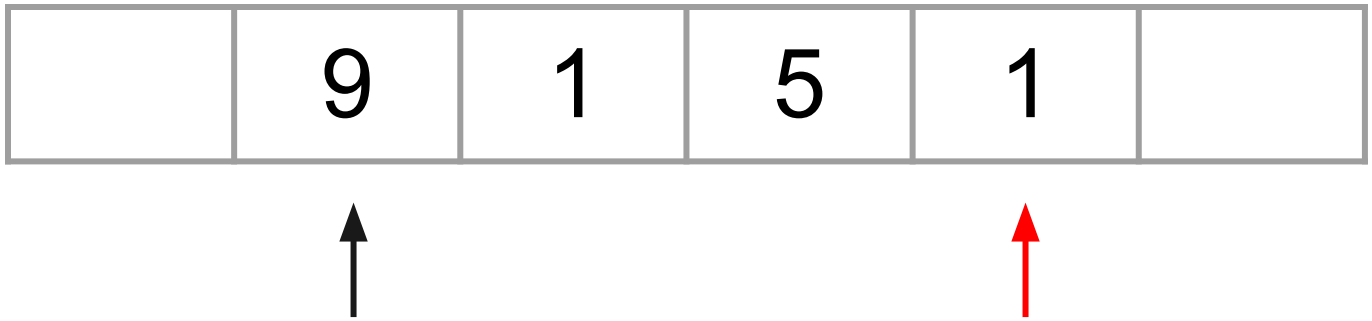


início
fim

UFABC

Fila

Insere x = 1

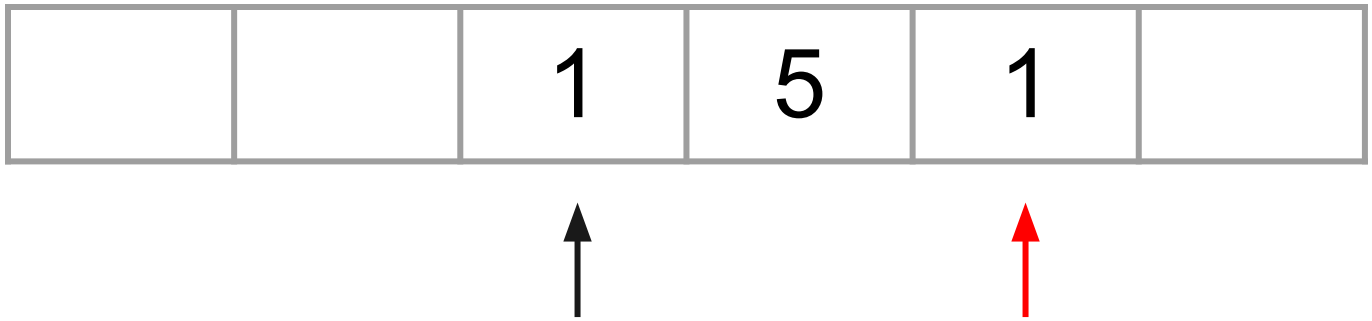


início
fim

UFABC

Fila

Remove, x = 9

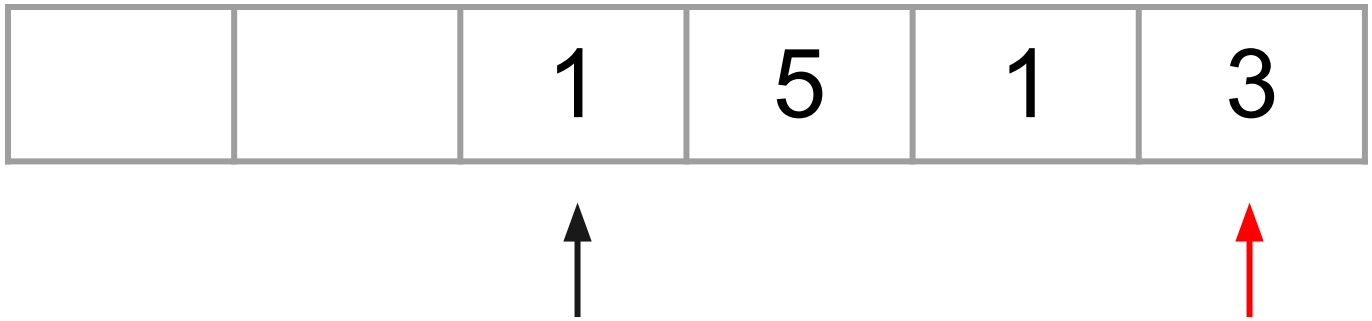


← início
← fim

UFABC

Fila

Insere x = 3

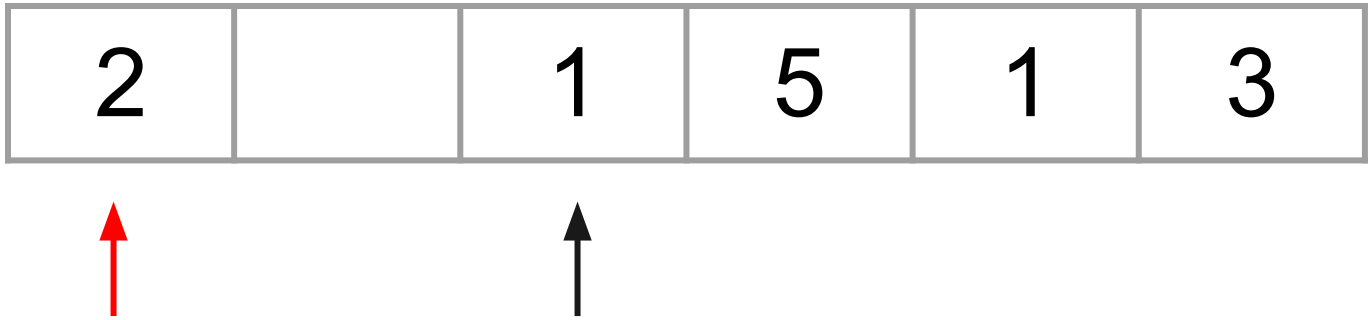


início
fim

UFABC

Fila

Insere x = 2



← início
← fim

UFABC

Fila

```
typedef struct fila {
    int pessoas[N];
    int frente;
    int fim;
    int elems;
} fila;
```

UFABC

Fila

```
void cria_fila (fila * f)
{
    f->frente = f->fim = -1;
    f->elems = 0;
}
```

UFABC

Fila

```
int insere_fila ( fila * f, int t )
{
    if (f->elems + 1 > N) return 0;

    if (f->elems == 0) f->frente = f->fim = 0;
    else               f->fim = (f->fim + 1) % N;

    f->pessoas[f->fim] = t;
    ++(f->elems);

    return 1;
}
```

UFABC

Fila

```
int remove_fila( fila * f )
{
    int t;

    if (f->elems == 0) return -1;
    t = f->pessoas[f->frente];

    --(f->elems);

    if (f->elems == 0) f->frente = f->fim = -1;
    else               f->frente = (f->frente + 1) % N;

    return t;
}
```

UFABC

Exemplo 3 (lista ligada)

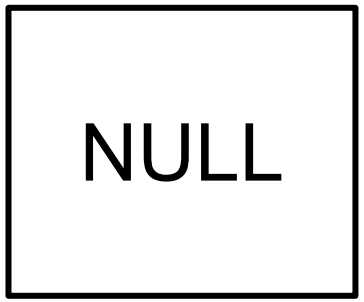
Uma outra estrutura de dados interessante é a lista ligada, ela permite gerar um array, pilha, lista de tamanho dinâmico, ou seja, que cresce e encolhe conforme a necessidade.

UFABC

Exemplo 3 (lista ligada)

Inicializa lista ligada:

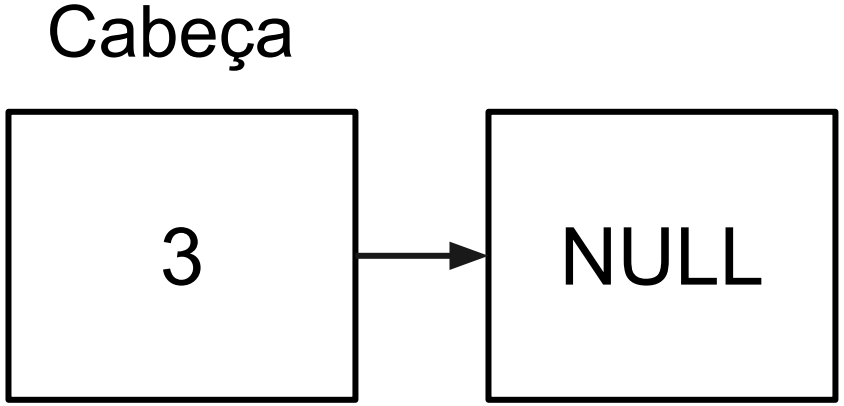
Cabeça



UFABC

Exemplo 3 (lista ligada)

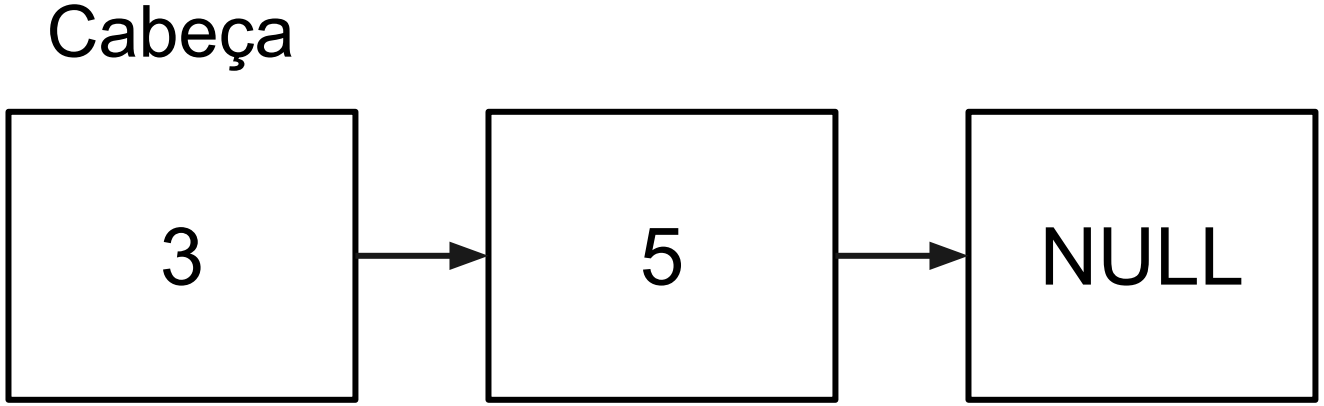
Insere x = 3



UFABC

Exemplo 3 (lista ligada)

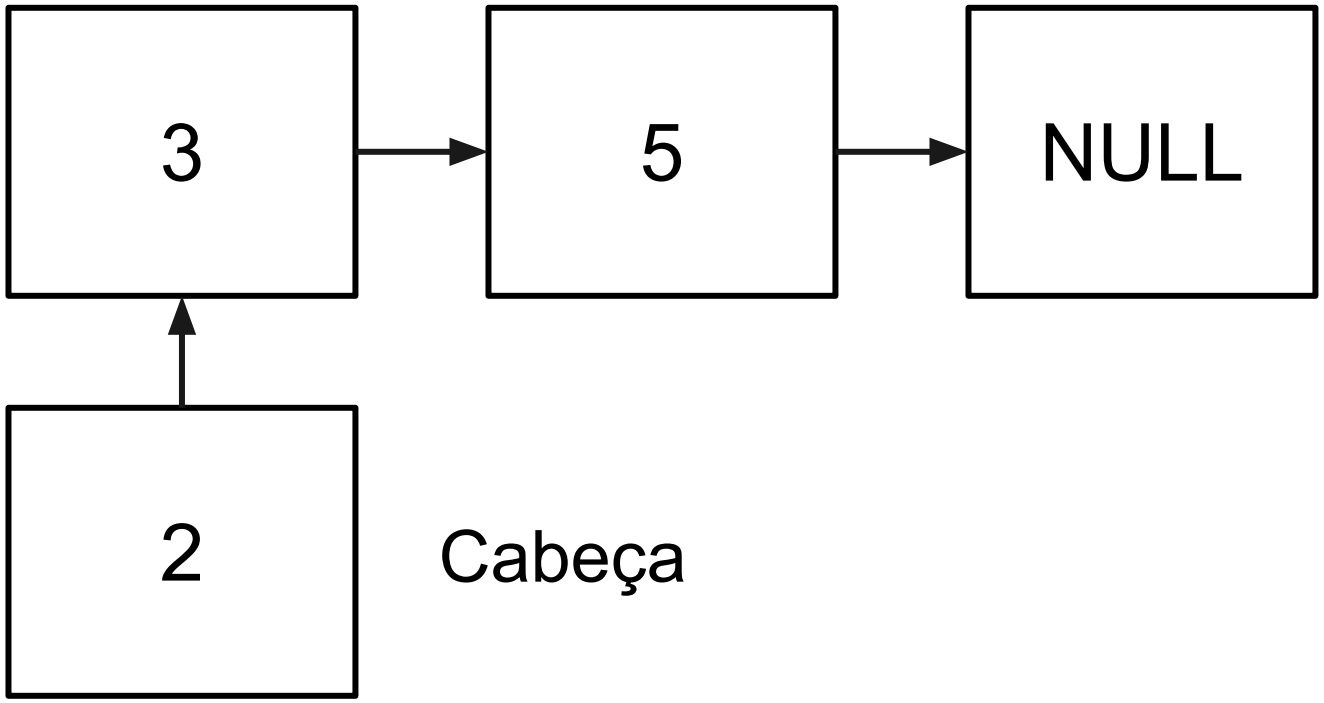
Insere x = 5 no final



UFABC

Exemplo 3 (lista ligada)

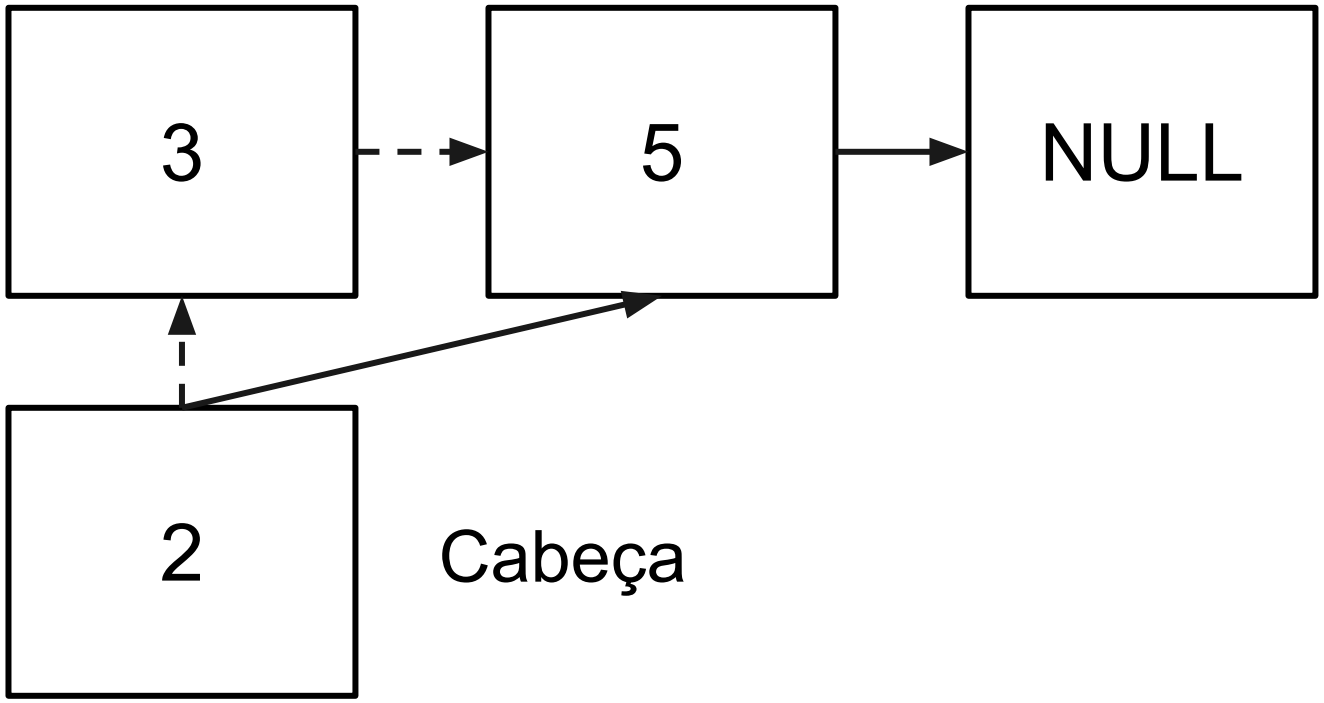
Insere x = 3 no começo



UFABC

Exemplo 3 (lista ligada)

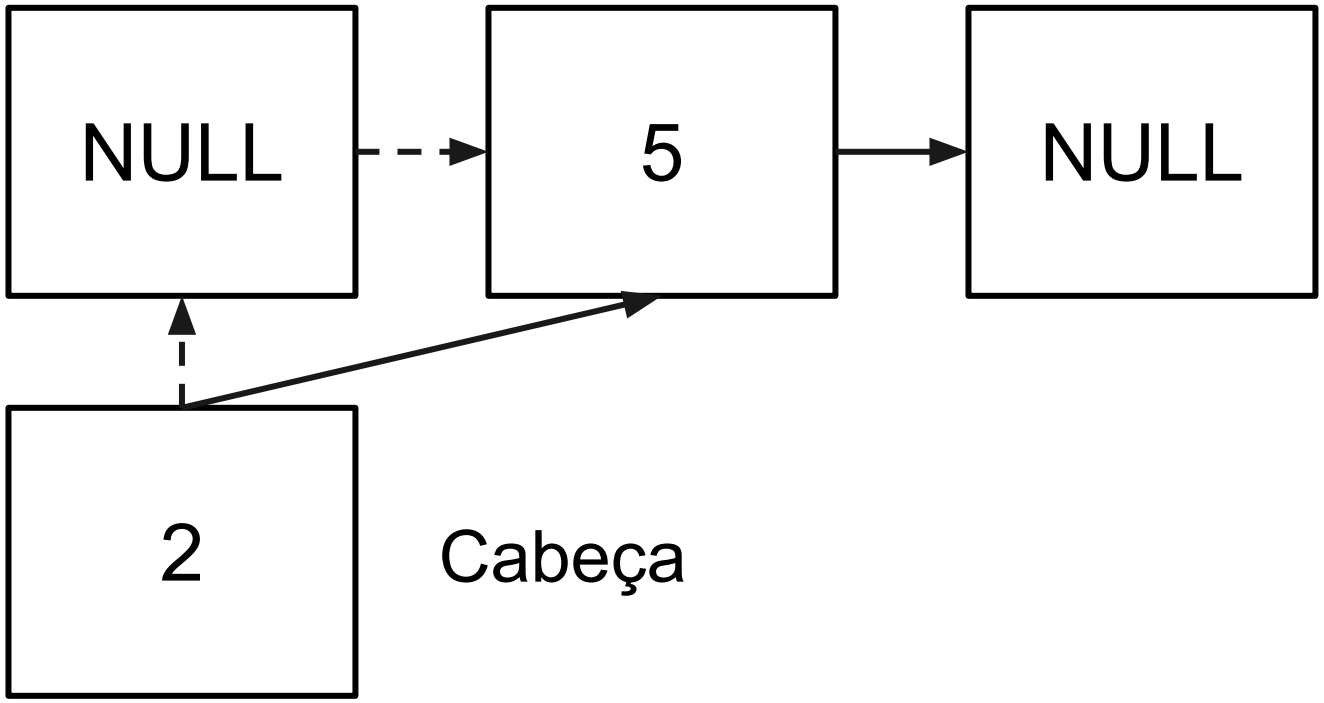
Remove x = 3



UFABC

Exemplo 3 (lista ligada)

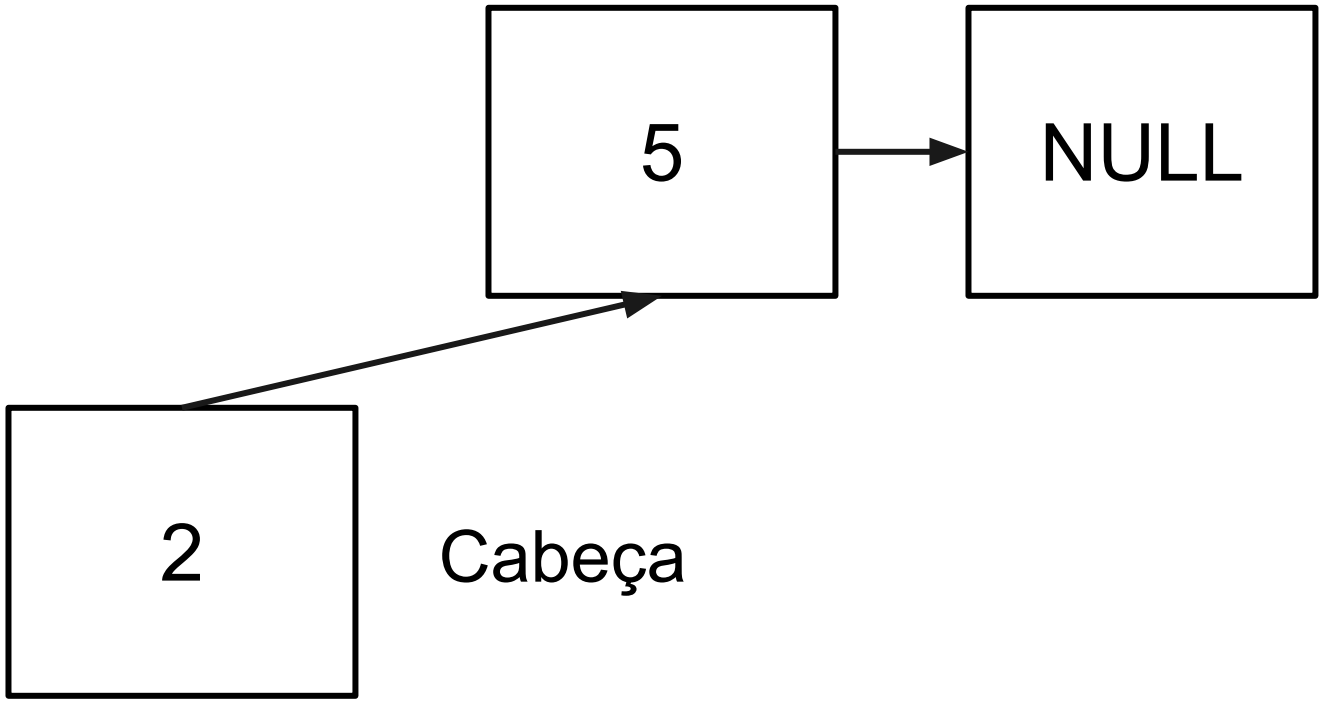
Remove x = 3



UFABC

Exemplo 3 (lista ligada)

Remove x = 3



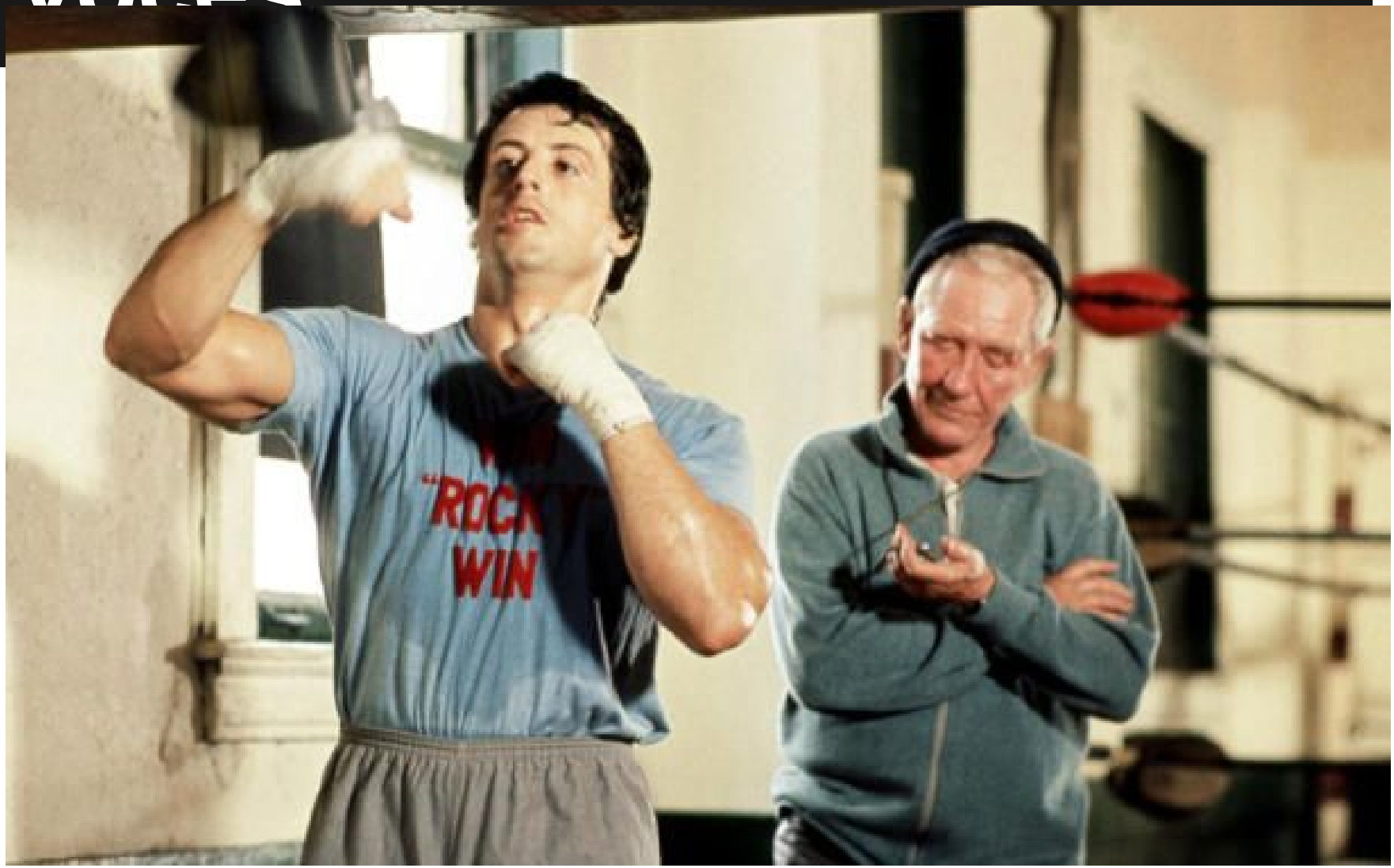
UFABC

Exemplo 3 (lista ligada)

```
typedef struct node {
    int minuto_chegada;
    struct node * prox;
} node;
```

UFABC

Era para isso que eu treinei vocês



Segmentation Fault

Esse tipo de estrutura que envolve ponteiros é responsável pelos mais complicados casos de **segmentation fault**.

Vamos ver como prevenir.

UFABC

Exemplo 3 (lista ligada)

```
node * cria_no (int t)
{
    node * n = (node *) malloc(sizeof(node));

    if (n != NULL) {
        n->minuto_chegada = t;
        n->prox = NULL;
    }

        return n;
}
```

UFABC

Exemplo 3 (lista ligada)

```
node * insere_fila ( node * f, int t )
{
    node * cursor = f;

    if (f == NULL)     return cria_no(t);

    while (cursor->prox != NULL) cursor = cursor->prox;

    cursor->prox = cria_no(t);

    return f;
}
```

UFABC

Exemplo 3 (lista ligada)

```
node * remove_fila( node * f )
{
    node * novo_f;

    if (f == NULL) return NULL;

    novo_f = f->prox;
    free(f);

    return novo_f;
}
```

UFABC

Lista Ligada

Professor! Você falou que quando é passado um ponteiro para uma função, o valor da variável passada era alterado, por que agora temos que retornar o ponteiro após alteração?

UFABC

Lista Ligada

Agora não estamos alterando **o conteúdo** para onde o ponteiro aponta, mas estamos alterando a **direção apontada**.

Nesse caso, o ponteiro funciona como uma variável “normal”.

UFABC

Boas práticas com structs

Não se esqueça de:

tipo função(tipo);

Fará uma cópia de tamanho sizeof(tipo) para a função e fará uma cópia de sizeof(tipo) para o retorno da função.

UFABC

Boas práticas com structs

Se o tamanho da estrutura em bytes for muito grande, considere passar e retornar apenas ponteiro:

tipo * função(tipo *);

UFABC

Alinhamento dos Dados

Na aula sobre ponteiros e alocação de memória vimos que o computador aloca memória sequencialmente, isso nos permite fazer:

*(array + n) para acessar o n-ésimo elemento da array.

UFABC

Alinhamento dos Dados

Supondo bytes alocados sequencialmente, para o processador acessar um inteiro (4 bytes) ele precisa fazer 4 leituras da memória na sequência.

Isso pode tornar o processo de trabalhar com valores de tamanho > 1 byte muito custoso.

UFABC

Alinhamento dos Dados

Os fabricantes de processadores resolveram esse problema criando **data banks** que são segmentos de memória paralela que são lidas como sequenciais!

UFABC

Alinhamento dos Dados

Banco de dados de 32 bits:

0	1	2	3
4	5	6	7
...			

UFABC

Alinhamento dos Dados

Esses bancos de dados permitem a leitura de **N** bytes sequenciais em uma única operação.

Em processadores de 32-bits temos 4 data banks, 64-bits temos 8 data banks.

UFABC

Alinhamento dos Dados

Imagine que temos uma estrutura definida como:

```
struct tipo {
    char var1;
    int var2;
    char var3;
}
```

UFABC

Alinhamento dos Dados

Com 32-bits teremos:

var1	var2	var2	var2
var2	var3		

Nessa situação, toda vez que tentarmos acessar o componente **var3** da estrutura, precisaríamos de duas operações.

UFABC

Alinhamento dos Dados

Para resolver esse problema podemos criar um **padding**:

```
struct tipo {
    char var1;
    char pad1[3];
    int var2;
    char var3;
    char pad2[3];
}
```

UFABC

Alinhamento dos Dados

Permitindo que a leitura e escrita de cada variável use apenas uma única operação.

var1	pad1	pad1	pad1
var2	var2	var2	var2
var3	pad2	pad2	pad2

UFABC

Alinhamento dos Dados

Os compiladores já fazem o padding automaticamente para você.

Mas, tem outra solução...

UFABC

Alinhamento dos Dados

Muitas vezes basta ordenar os elementos da sua estrutura para que ela tenha um alinhamento correto, evitando o padding:

```
struct tipo {
    int var2;
    char var1;
    char var3;
}
```

UFABC

Alinhamento dos Dados

Ocupando menos espaço na memória.

var2	var2	var2	var2
var1	var3	pad	pad

UFABC

EX01 - Números Complexos

Crie funções para realizar as seguintes operações com números complexos:

- Conjugado
- Adição
- Subtração
- Multiplicação
- Divisão
- Recíproco
- Raíz Quadrada
- Magnitude

UFABC

Ex02 - Data e Hora

Crie uma estrutura para guardar data e hora e crie a operação de adição (adicionar x dias e n minutos para uma data).

UFABC