Algoritmos e estruturas de Dados

Filas e Pilhas

Prof. Maiquel de Brito

Filas (*Queues*)

Problemática / motivação:

Uma aplicação que gerencia tarefas que devem ser executadas na ordem em que são criadas.

As tarefas são armazenadas em uma lista encadeada. Novas tarefas são colocadas no final da lista.

Seja *n* a quantidade de tarefas armazenadas na lista.

O custo de seleção de uma tarefa para execução é *O(1)*

porque basta retirar o primeiro elemento da lista

O custo de inserção de uma tarefa é *O(n)*

porque é necessário percorrer a lista inteira para encontrar o último elemento

O custo de inserção pode ser um problema no caso de muitas inserções e poucas remoções e no caso de muitas tarefas armazenadas na lista

para *m* inserções, tem-se custo *O(m.n)*

Filas (*Queues*)

São estruturas de dados que possuem comportamento similar à uma fila de caixa do supermercado:

- A primeira pessoa que chegou na fila é atendida primeiro
- Os demais clientes entram na fila e aguardam atendimento

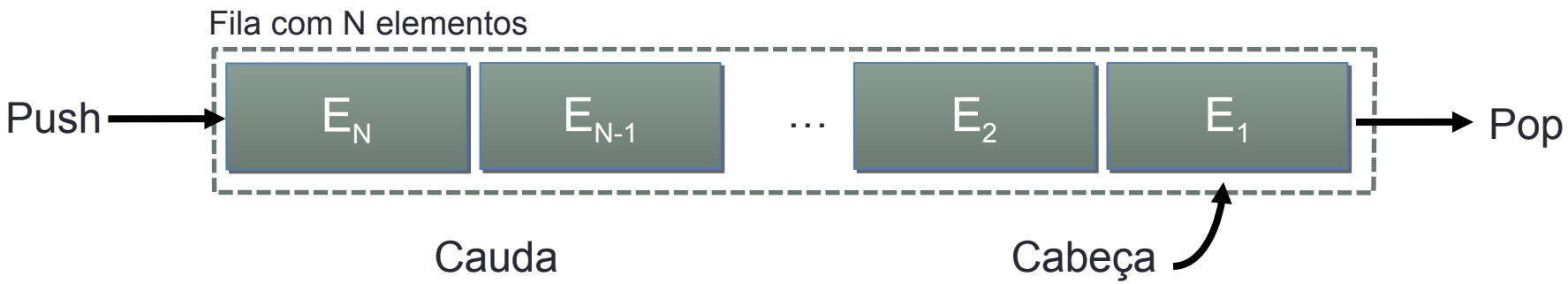
Uma fila pode ser considerada como uma **restrição de lista** (regras especiais para acesso dos elementos):

- Todo elemento que entra na lista, entra no fim
- Todo elemento que sai da lista, sai do início

Essa disciplina de acesso é conhecida como FIFO (First-In-First-Out)

Filas (*Queues*)

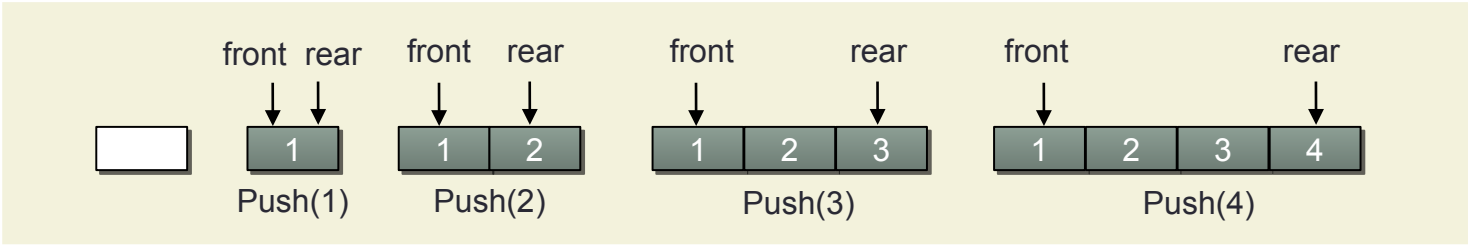
- Uma fila tem início (cabeça) e fim (cauda)
- A inserção de elementos se faz após o ultimo elemento (cauda).
- O elemento eliminado é sempre o que se encontrava na fila há mais tempo (cabeça)

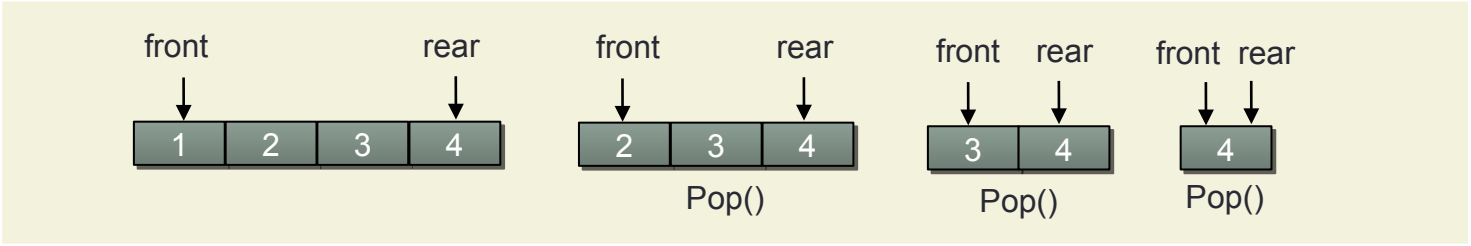


Filas (*Queues*)

Operações comuns:

- criar uma fila vazia
- adicionar (*push*)/remover(*pop*) elementos
- verificar qual o elemento da cabeça/cauda da fila (mais antigo/recente)





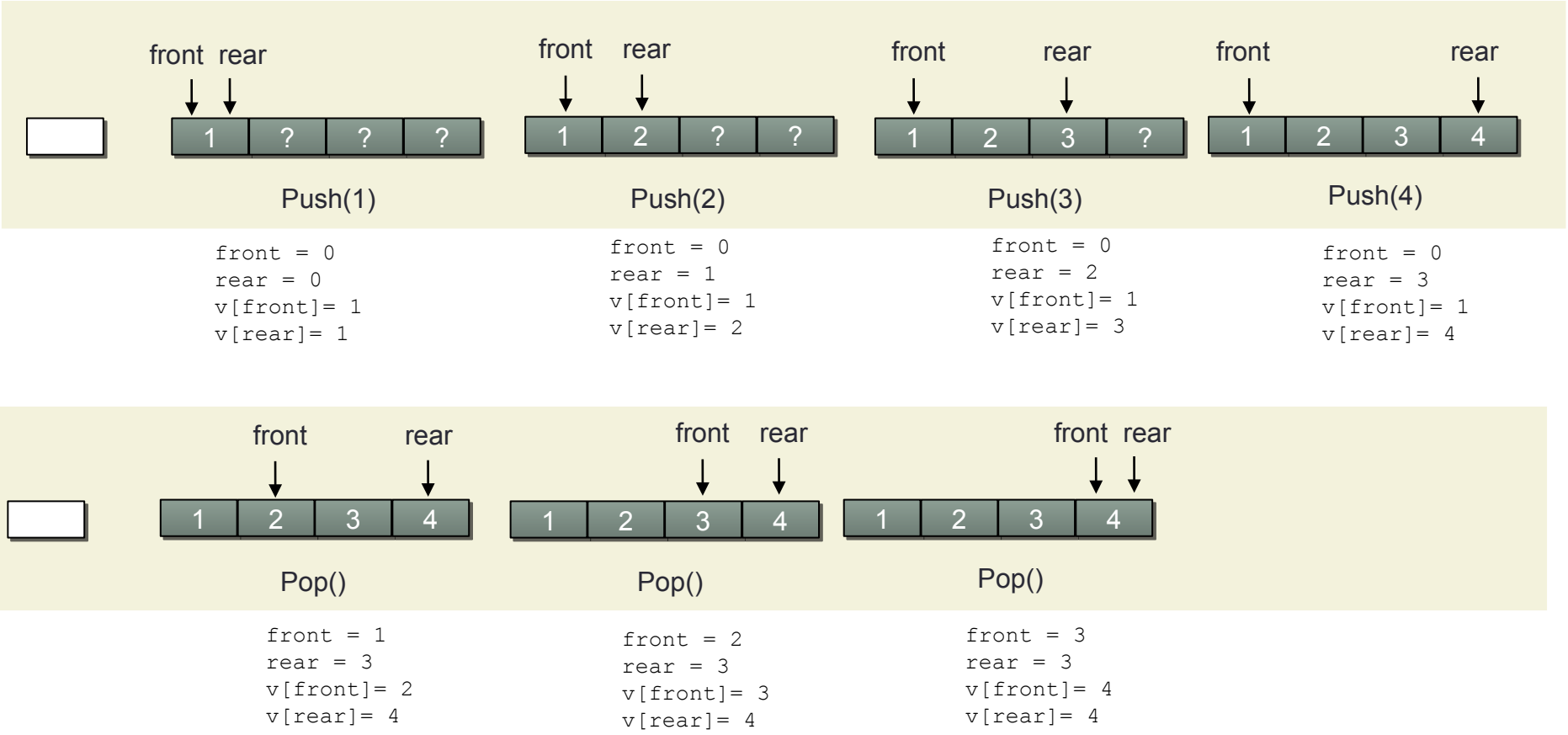
Implementação de Filas (*Queues*)

A implementação da fila pode ser feita com:

- Um **vetor**
  - Capacidade do vetor é pré-definida. É necessário saber o tamanho da fila e o índice dos elementos que estão na cauda e na cabeça da fila.
  - **Push**: Se a fila não estiver cheia, insere novo elemento na cauda e incrementa o índice respectivo. O índice volta a zero se se exceder o tamanho do vector (fila circular).
  - **Pop**: Se a fila não estiver vazia, remove o elemento da cabeça da fila e incrementa o índice respectivo. O índice volta a zero se exceder o tamanho do vector (fila circular).
- Uma **lista ligada**
  - **Push**: Insere novo elemento no fim da lista.
  - **Pop**: Remove o elemento do início da lista.

Fila (Implementação baseada em Vetor)

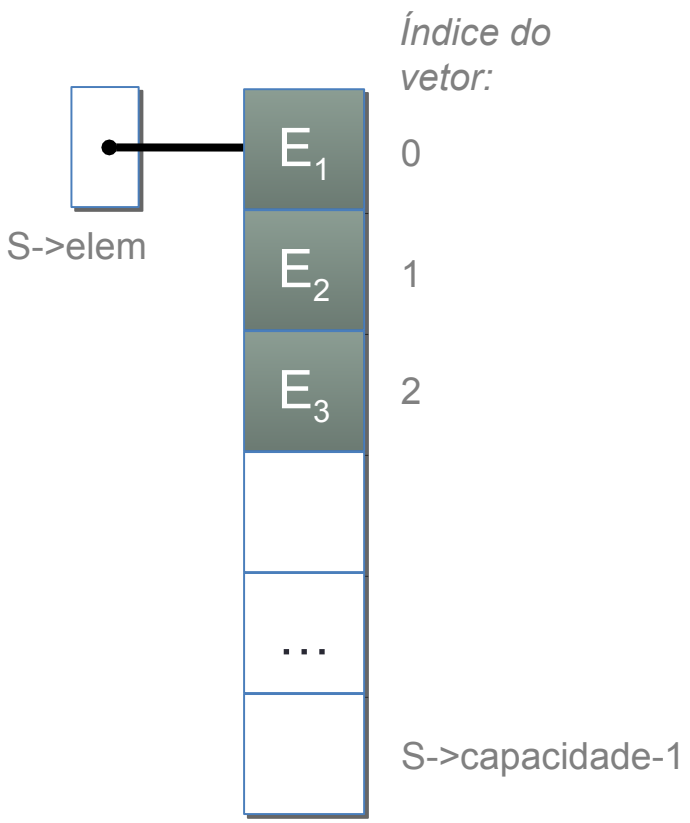
- O tamanho do vetor não se altera durante a execução do programa
- Somente são atualizados os índices da cabeça e da cauda



Fila (Implementação baseada em Vetor)

```c
struct filaItem
{
    int capacidade; /* capacidade da fila */
    int cabeca; /* índice da cabeça da fila */
    int cauda; /* índice da cauda da fila */
    int tam; /* tamanho da fila */
    int *elem; /* vetor com os elementos */
};
typedef struct filaItem* Fila;
```

```c
/* cria uma nova fila */
Fila criaFila( int maxSize );

/* insere um novo elemento na cauda*/
void Push( int X, Fila F ) ;

/* remove o elemento da frente */
void Pop(Fila F );

/* obtém o valor do elemento da frente */
int Front(Fila F );
```



Fila (Implementação baseada em Vetor)

```
Fila criaFila( int maxSize )
{
    Fila F1;
    if( maxSize < MIN_QUEUE_SIZE )
        printf( "Fila size is too small\n" );

    F1 = malloc( sizeof( struct filaItem ) );
    if( F1 == NULL ) {
        printf( "Out of space!\n" ); exit(EXIT_FAILURE);
    }
    F1->elem = malloc( sizeof( int ) * maxSize );
    if( F1->elem == NULL ) {
        printf( "Out of space!\n" ); exit(EXIT_FAILURE);
    }

    F1->capacidade = maxSize;
    F1->tam = 0;
    F1->cabeca = 0;
    F1->cauda = 0;
    return F1;
}
```

Fila (Implementação baseada em Vetor)

```
void Push( int X, Fila F )
{
    if( F->tam == F->capacidade ) printf( "Full queue\n" );
    else {
        F->tam++;
        F->elem[ F->cauda ] = X;
        if ( ++F->cauda == F->capacidade )
            F->cauda = 0;
    }
}

void Pop( Fila F )
{
    if( F->tam == 0 ) printf( "Empty queue\n" );
    else
    {
        F->tam--;
        if ( ++F->cabeca == F->capacidade )
            F->cabeca = 0;
    }
}
```

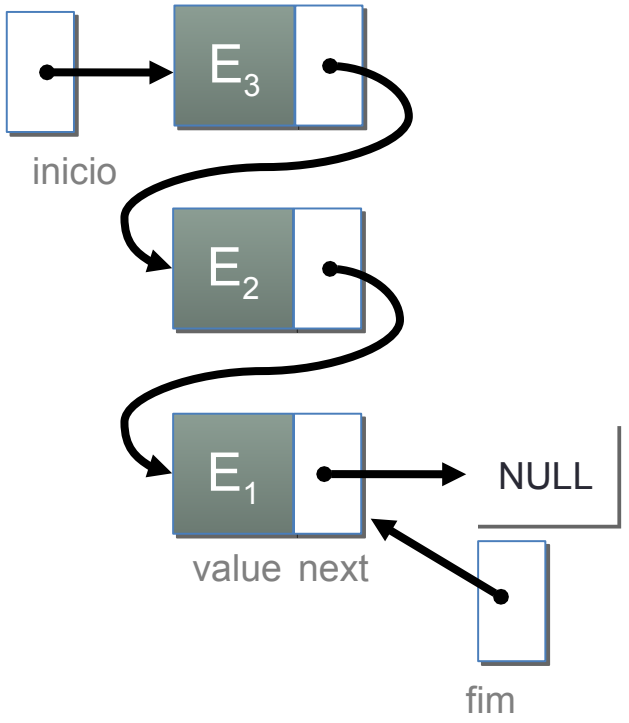
Fila (Implementação baseada em Vetor)

```
int Front( Fila F )
{
    if( F->tam != 0 )
        return F->elem[ F->cabeca ];
    printf( "Empty queue\n" ); return 0;
}

int main( )
{
    Fila F; int i;

    F = criaFila(15);
    for( i = 0; i < 10; i++ )
        Push( i, F );

    while( F->tam != 0 )
    {
        printf( "Cabeca: %d\tTamanho: %d\n", Front( F ), F-
>tam );
        Pop( F );
    }

    free( F->elem ); free( F);
}
```

Fila (Implementação baseada em Lista Ligada)

```
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* apontador para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;
```

```
/* cria uma nova fila */
Fila *criaFila();

/* insere um novo elemento na fila */
void Push( int X, Fila *F );

/* remove o elemento da frente */
void Pop( Fila *F );

/* obtém o valor do elemento da frente */
int Peek( Fila *F );
```



Fila (Implementação baseada em Lista Ligada)

```
Fila *criaFila( void ) {
    Fila *F = malloc( sizeof( Fila ) );
    if( F == NULL ) {
        printf(”Pop: Sem espaco na memoria!!!\n"); exit(EXIT_FAILURE);
    }
    F->inicio = NULL; F->fim = NULL;
    return F;
}

void Push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem ) );
    if( no == NULL ) {
        printf( "Push: sem espaco na memoria!!!\n" ); exit(EXIT_FAILURE);
    }
    no->valor = X;
    no->next = NULL;

    if (F->inicio == NULL){ //se a fila estiver vazia
        F->inicio = no; F->fim = no;
    } else {
        F->fim->next = no; F->fim = no;
    }
}
```

Fila (Implementação baseada em Lista Ligada)

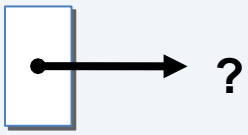
```
void Pop( Fila *F )
{
    if( F->inicio == NULL )
        printf( "Fila vazia\n" );
    else
    {
        Filaitem *aux = F->inicio;
        F->inicio = F->inicio->next;
        free(aux);
    }
}

int Peek( Fila *F )
{
    if( F->inicio != NULL )
        return F->inicio->valor;
    printf( "Fila vazia\n" );
    return -1;
}
```

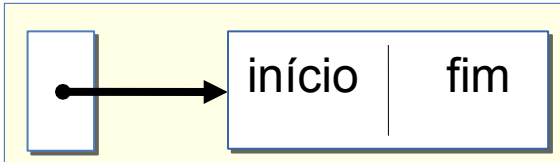
Fila (Implementação baseada em Lista Ligada)

```
int main( )
{
    Fila *F = criaFila();
    int i;

    for( i = 0; i < 10; i++ )
        Push( i, F );

    while( F->inicio != NULL )
    {
        printf( "Cabeca: %d\n", Peek( F ) );
        Pop( F );
    }

    free( F );
}
```

Fila – exemplo : criação de uma fila (slide 1/3)

```
int main(){
    Fila *fila = criaFila();
    ...
    ...
    ...
}
```






```
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


Fila *criaFila( void ) {
    Fila *F = malloc( sizeof( Fila ) );


}
```

A *struct* que armazena um item da fila é igual à que armazena itens de listas encadeadas.

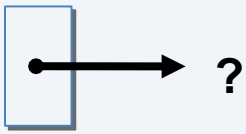
Uma *struct* que armazena uma fila consiste de dois ponteiros: um para o início e outro para o fim da fila.

Alocação de uma porção de memória para armazenar uma *struct* do tipo `Fila`.

O endereço da porção de memória alocada é atribuído ao ponteiro `F`.

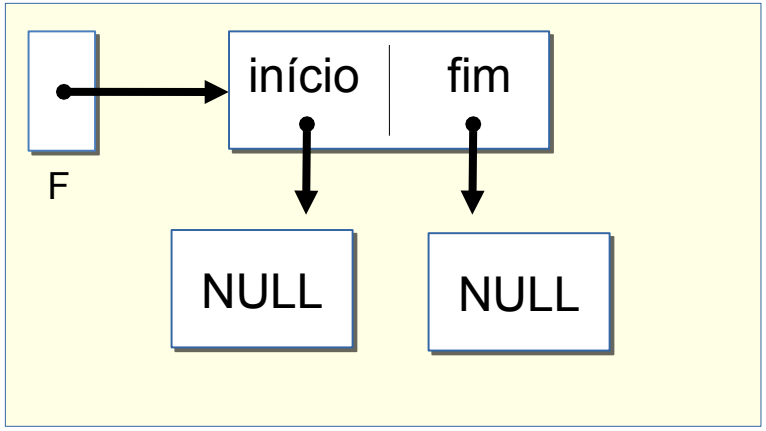
Fila – exemplo : criação de uma fila (slide 2/3)

```
int main(){
    Fila *fila = criaFila();
    ...
    ...
    ...
}
```



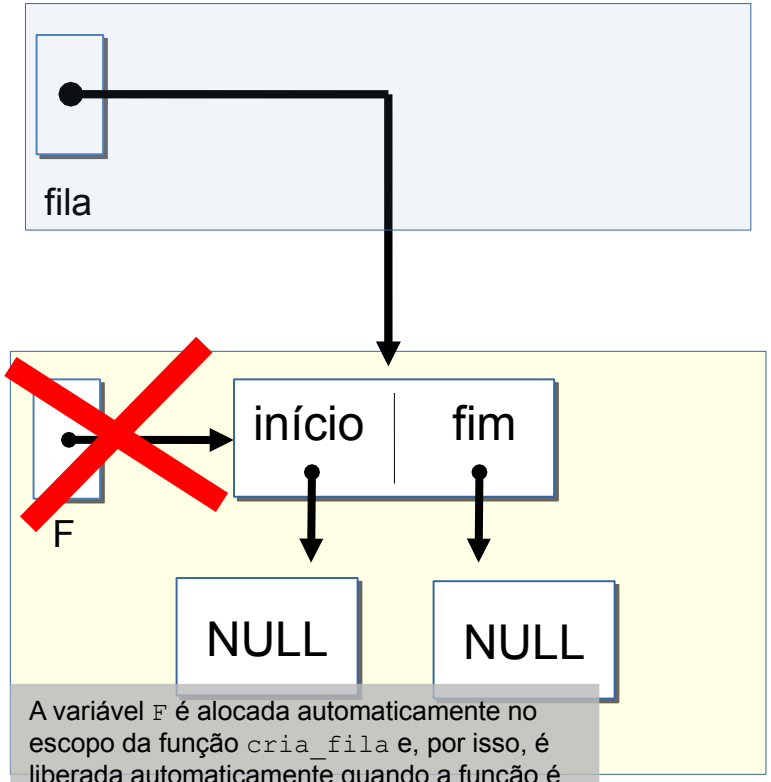

```
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


Fila *criaFila( void ) {
    Fila *F = malloc( sizeof( Fila ) );
    F->inicio = NULL; F->fim = NULL;

}
```

Em uma fila vazia, os ponteiros de início e fim apontam para NULL.



Fila – exemplo : criação de uma fila (slide 3/3)

```
int main(){
    Fila *fila = criaFila();
    ...
    ...
    ...
}
```



A variável `F` é alocada automaticamente no escopo da função `cria_fila` e, por isso, é liberada automaticamente quando a função é finalizada.

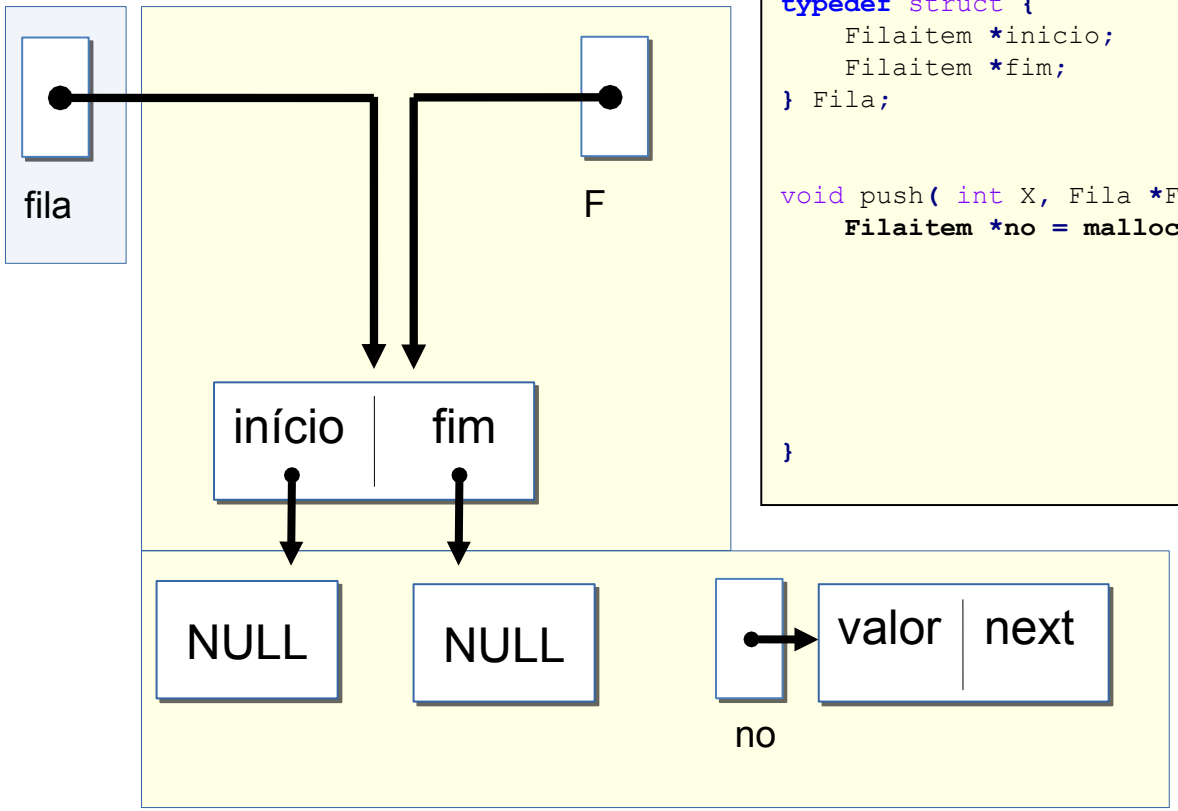
```
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


Fila *criaFila( void ) {
    Fila *F = malloc( sizeof( Fila ) );
    F->inicio = NULL; F->fim = NULL;
    return F;
}
```

A função retorna um ponteiro para porção de memória alocada.

Este ponteiro é atribuído à variável fila da função `main`.

Fila – exemplo : push (slide 1)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    ...
    ...
}
```




```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem ) );




}
```

Alocação de uma área de memória para armazenar um item da fila.

O endereço da área alocada é atribuído ao ponteiro `no`.

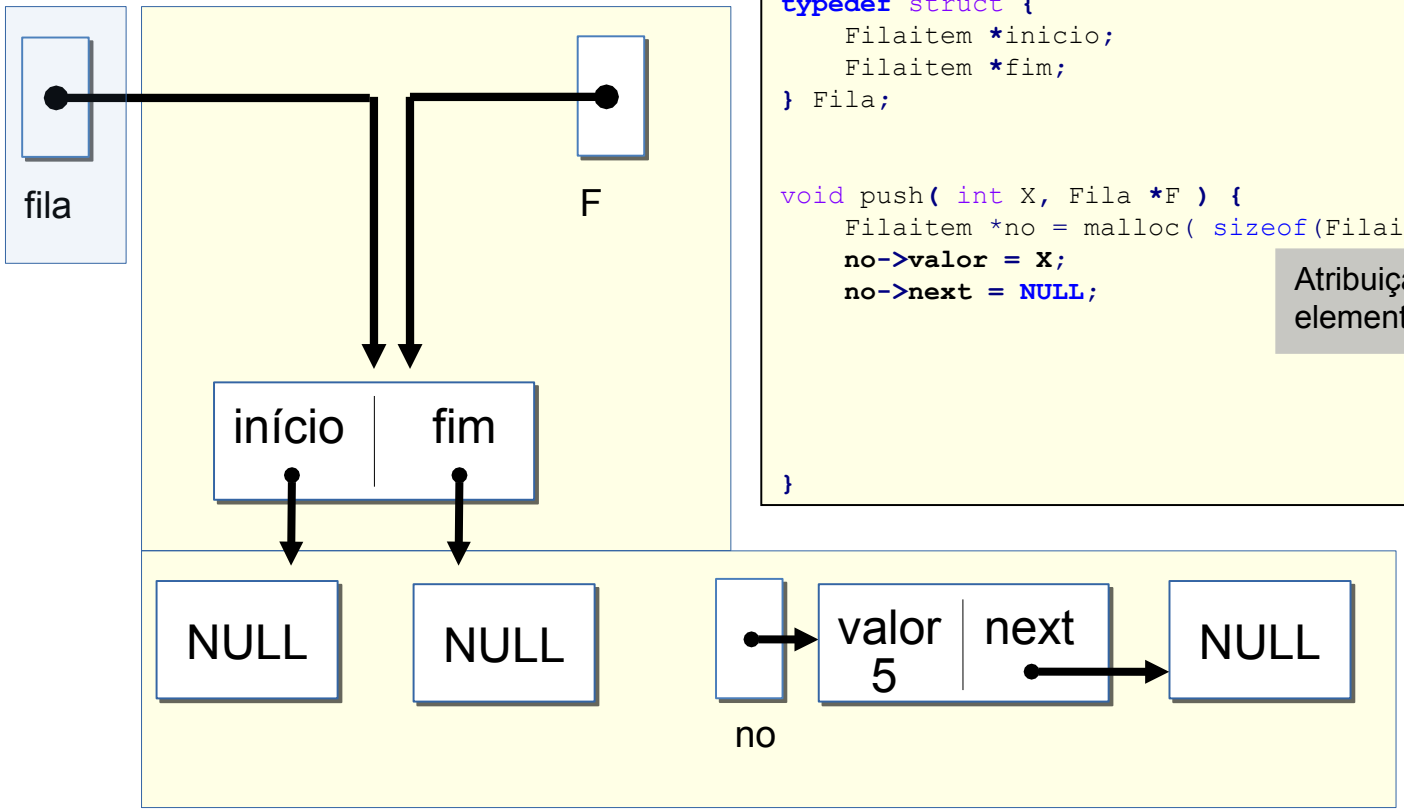
O parâmetro `F` (no escopo da função `Push`) aponta para o mesmo endereço para onde aponta o ponteiro `fila` (no escopo da função `main`).

Fila – exemplo : push (slide 2)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    ...
    ...
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem ) );
    no->valor = X;
    no->next = NULL;



}
```

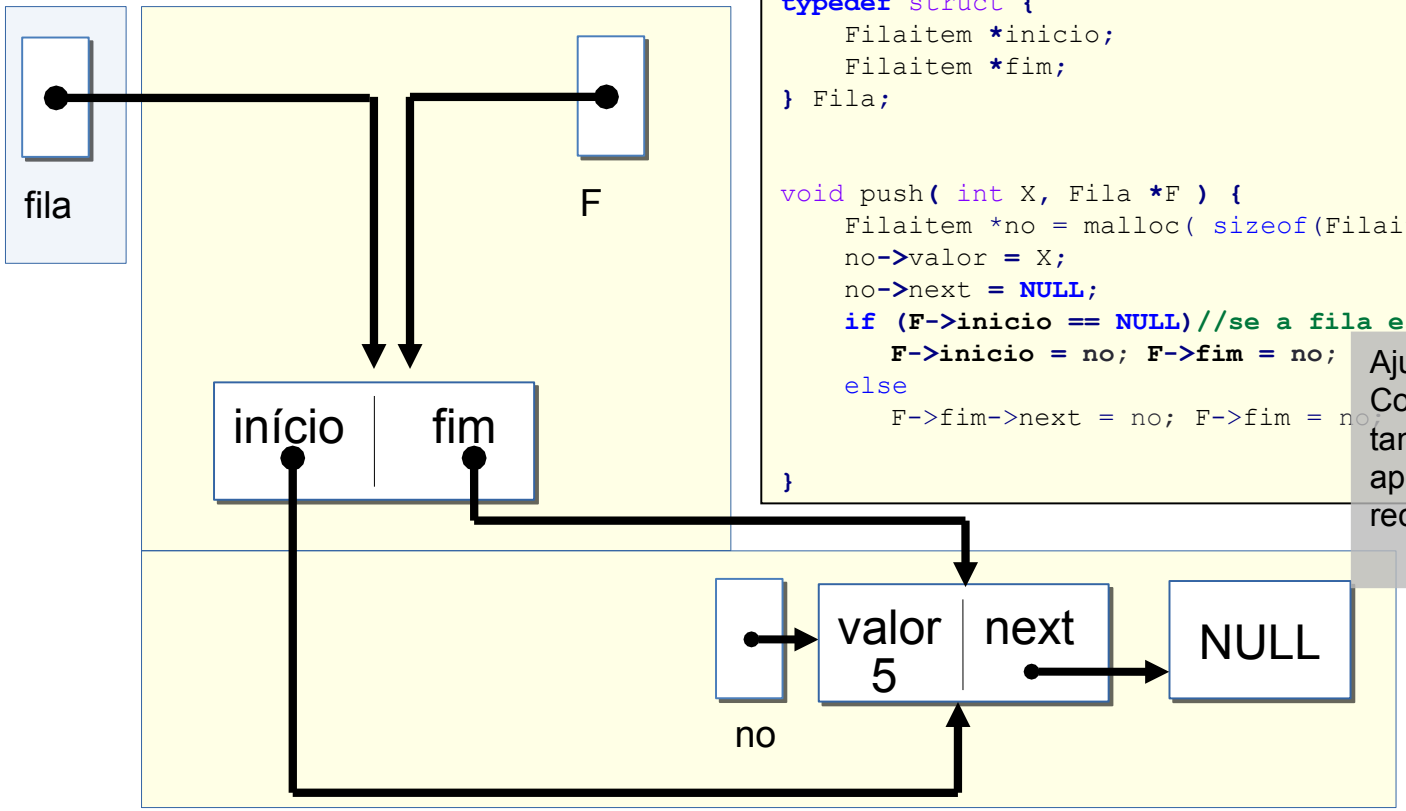
Atribuição de valores ao elemento da fila.



Fila – exemplo : push (slide 3)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    ...
    ...
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem ) );
    no->valor = X;
    no->next = NULL;
    if (F->inicio == NULL)//se a fila estiver vazia
       F->inicio = no; F->fim = no;
    else
       F->fim->next = no; F->fim = no;

}
```

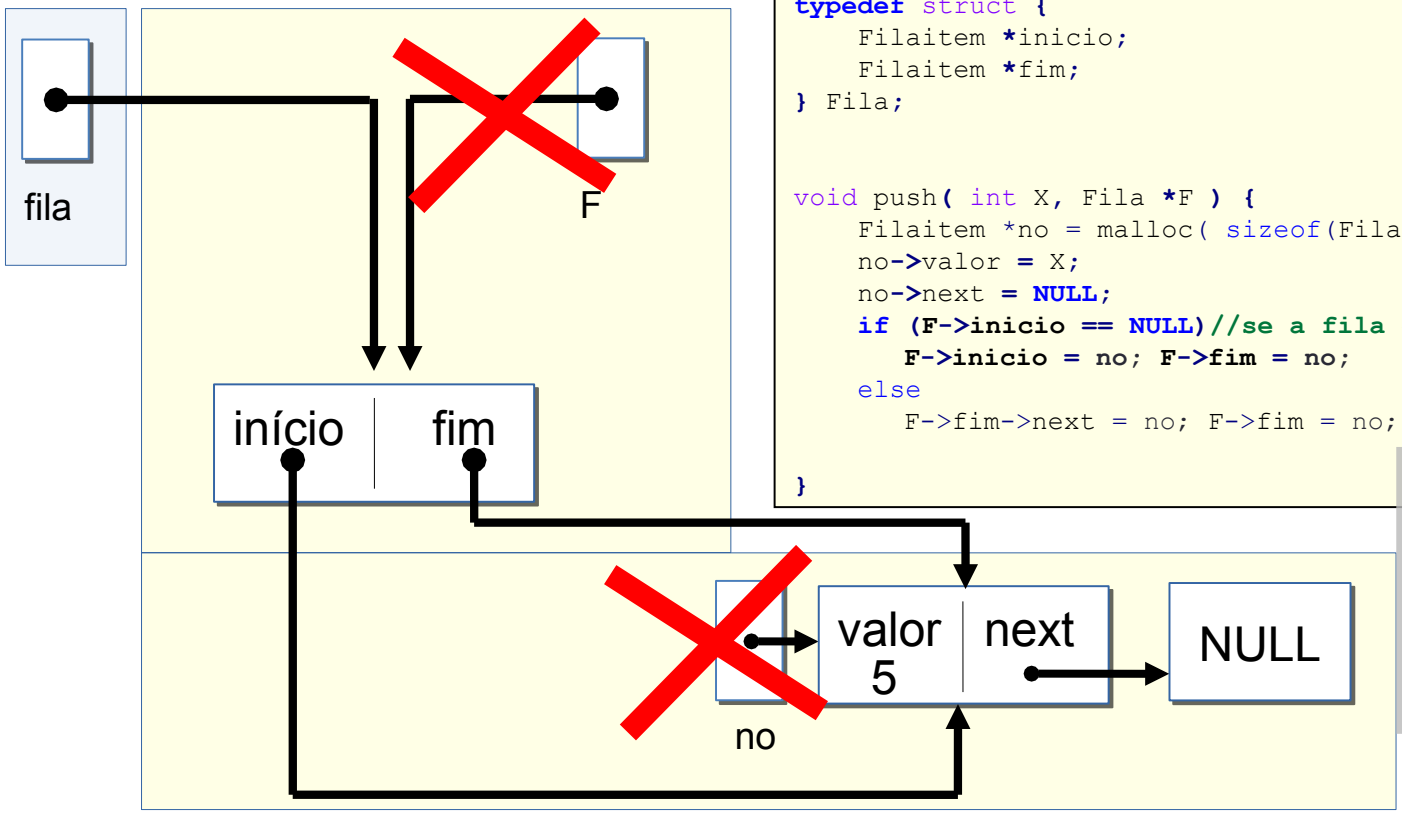


Ajuste de ponteiros:
Como a fila estava vazia, tanto o início quanto o fim apontam para o elemento recém criado.

Fila – exemplo : push (slide 4)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    ...
    ...
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem ) );
    no->valor = X;
    no->next = NULL;
    if (F->inicio == NULL)//se a fila estiver vazia
       F->inicio = no; F->fim = no;
    else
       F->fim->next = no; F->fim = no;

}
```

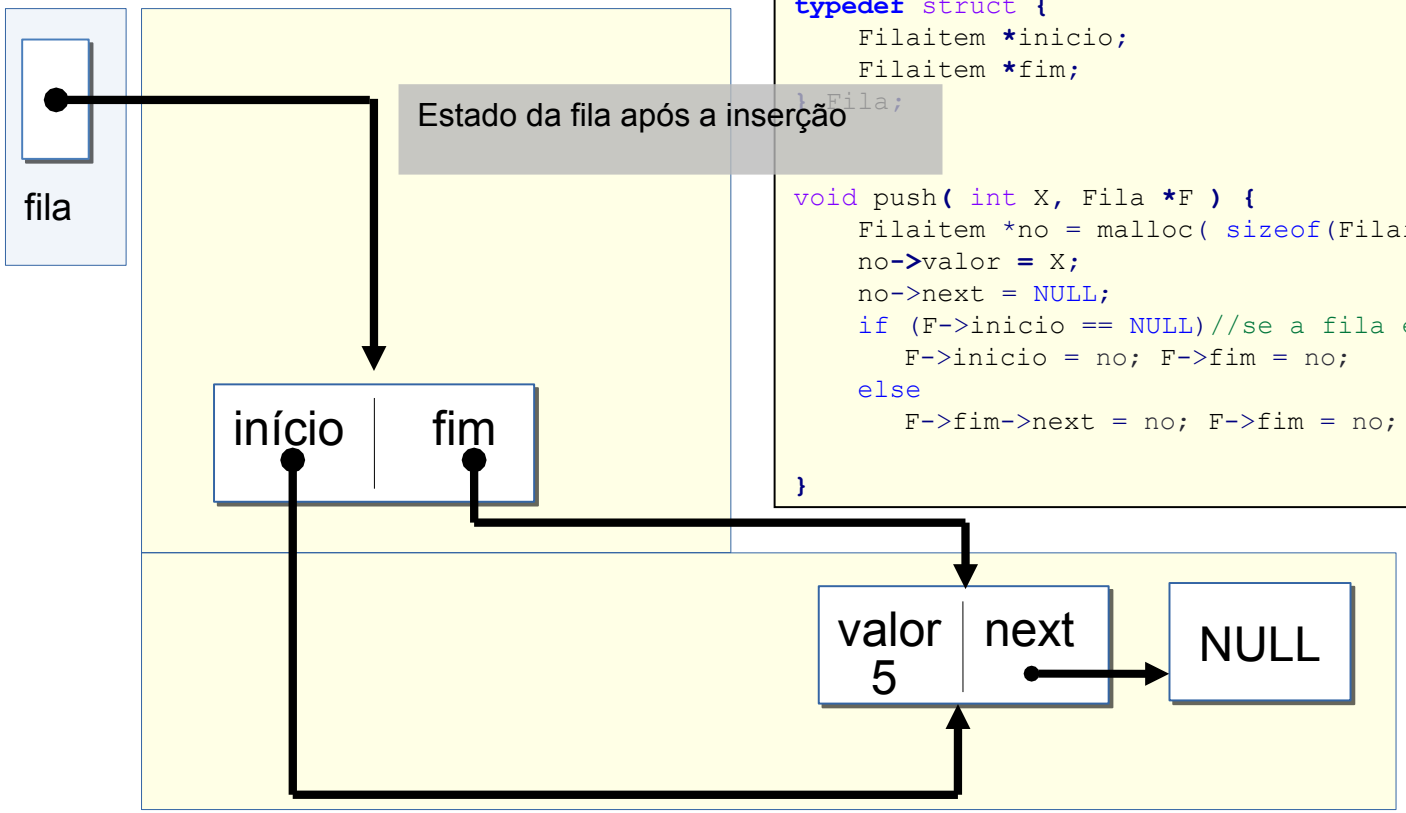


Ao final da execução, as variáveis `F` e `no`, que foram alocadas automaticamente no escopo da função `Push`, são liberadas automaticamente.

Fila – exemplo : push (slide 5)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    ...
    ...
}
```

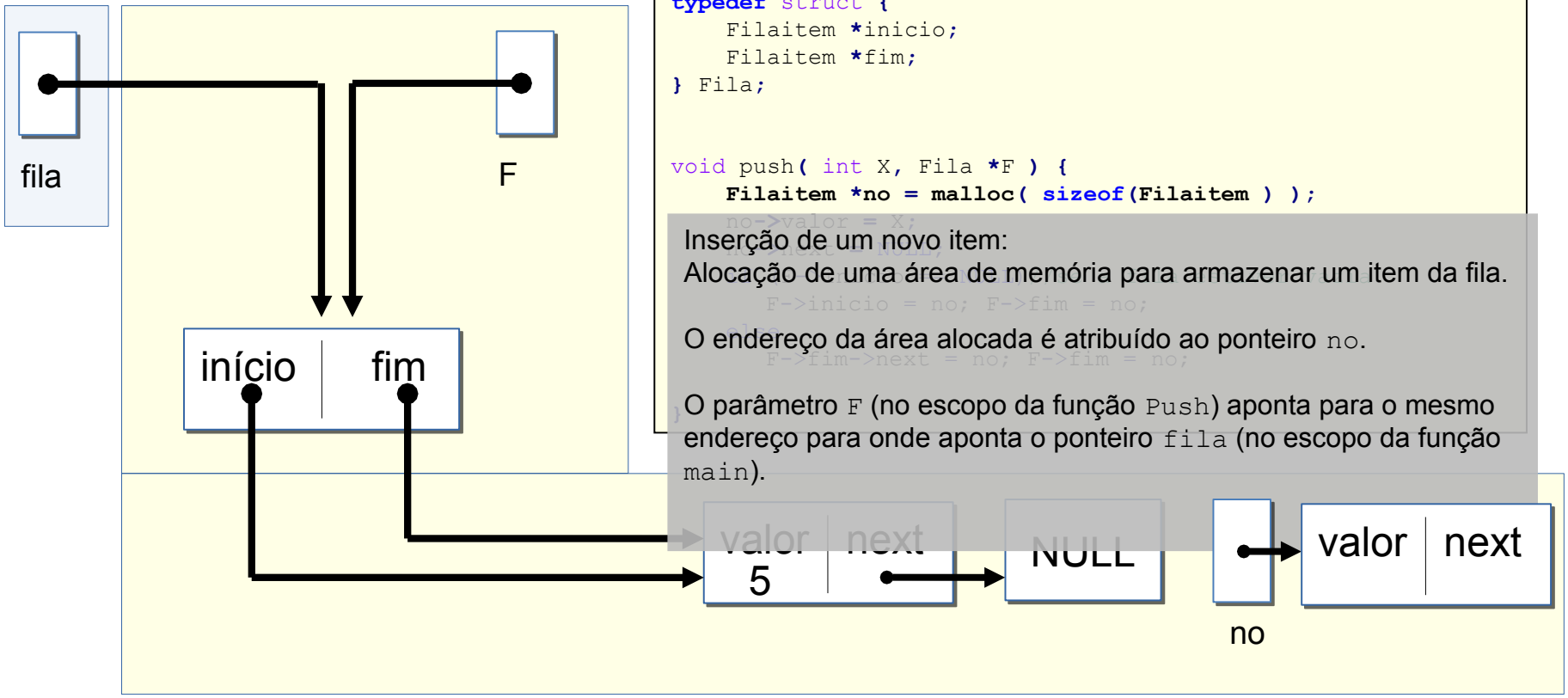
```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;

void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem ) );
    no->valor = X;
    no->next = NULL;
    if (F->inicio == NULL)//se a fila estiver vazia
       F->inicio = no; F->fim = no;
    else
       F->fim->next = no; F->fim = no;

}
```



Fila – exemplo : push (slide 6)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    push(9,fila);
    ...
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem ) );
```



Inserção de um novo item:
Alocação de uma área de memória para armazenar um item da fila.

O endereço da área alocada é atribuído ao ponteiro `no`.

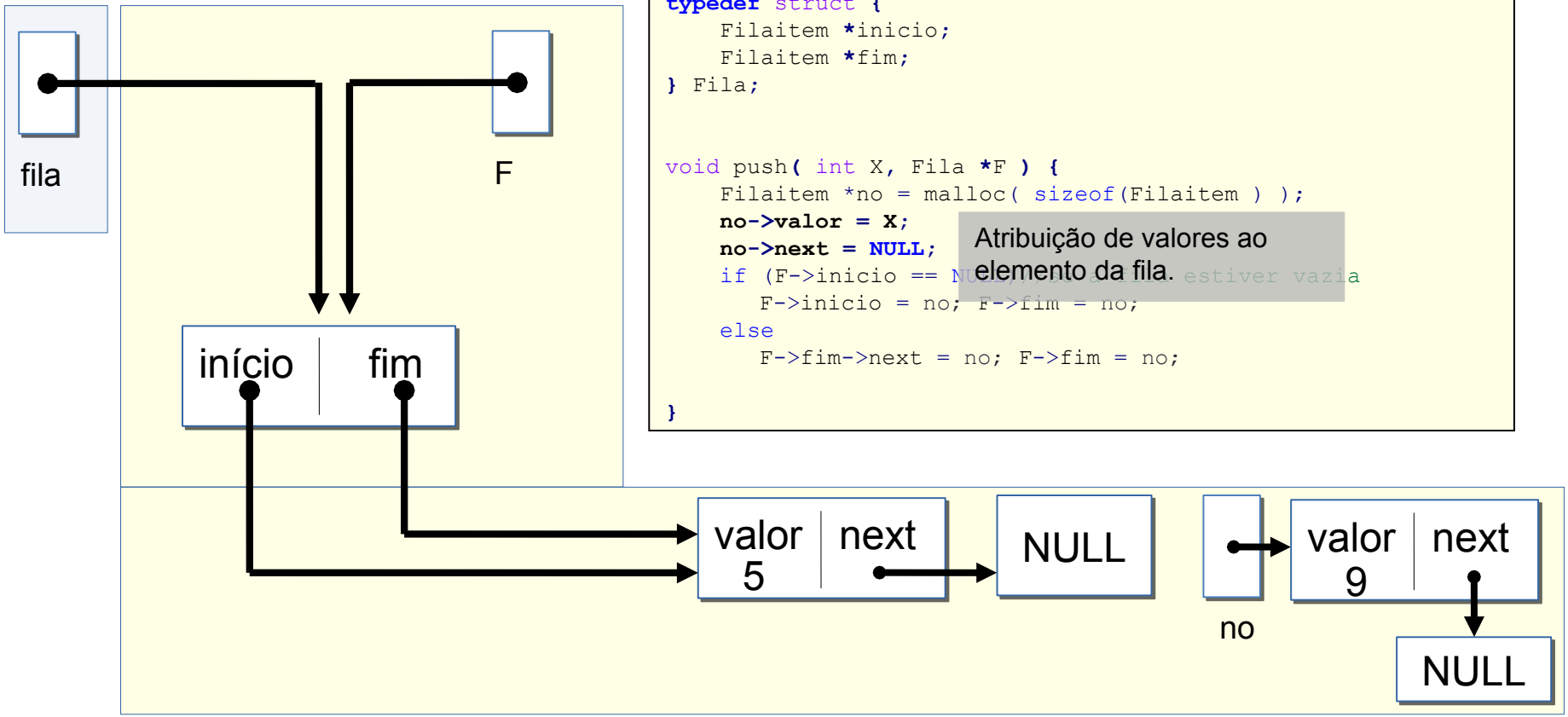
O parâmetro `F` (no escopo da função `Push`) aponta para o mesmo endereço para onde aponta o ponteiro `fila` (no escopo da função `main`).

Fila – exemplo : push (slide 7)

```
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    push(9,fila);
    ...
}
```

```
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem ) );
    no->valor = X;
    no->next = NULL;
    if (F->inicio == NULL) /* Fila estiver vazia
       F->inicio = no; F->fim = no;
    else
       F->fim->next = no; F->fim = no;

}
```

Atribuição de valores ao elemento da fila.



Fila – exemplo : push (slide 8)

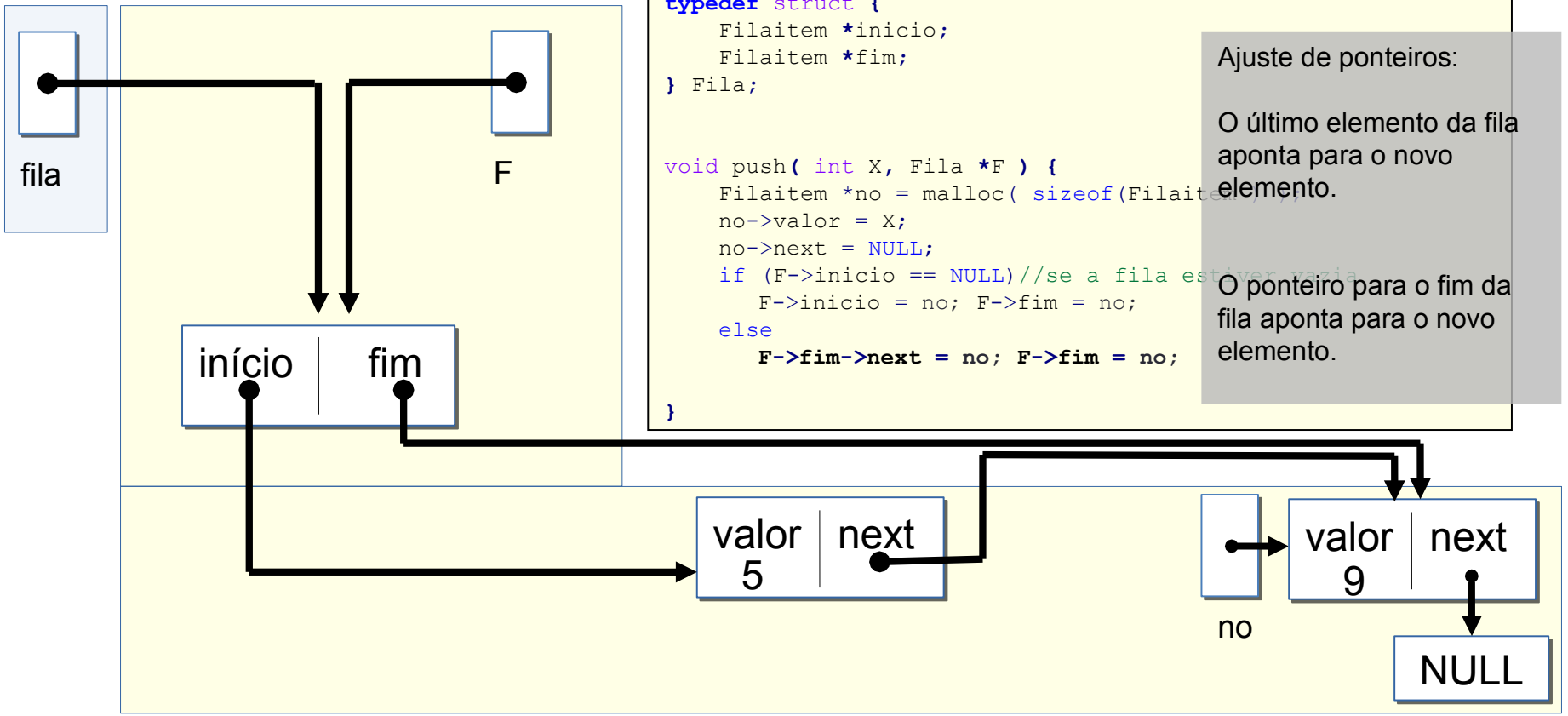
```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    push(9,fila);
    ...
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem) );
    no->valor = X;
    no->next = NULL;
    if (F->inicio == NULL)//se a fila estiver vazia
       F->inicio = no; F->fim = no;
    else
       F->fim->next = no; F->fim = no;

}
```

Ajuste de ponteiros:

O último elemento da fila aponta para o novo elemento.

O ponteiro para o fim da fila aponta para o novo elemento.



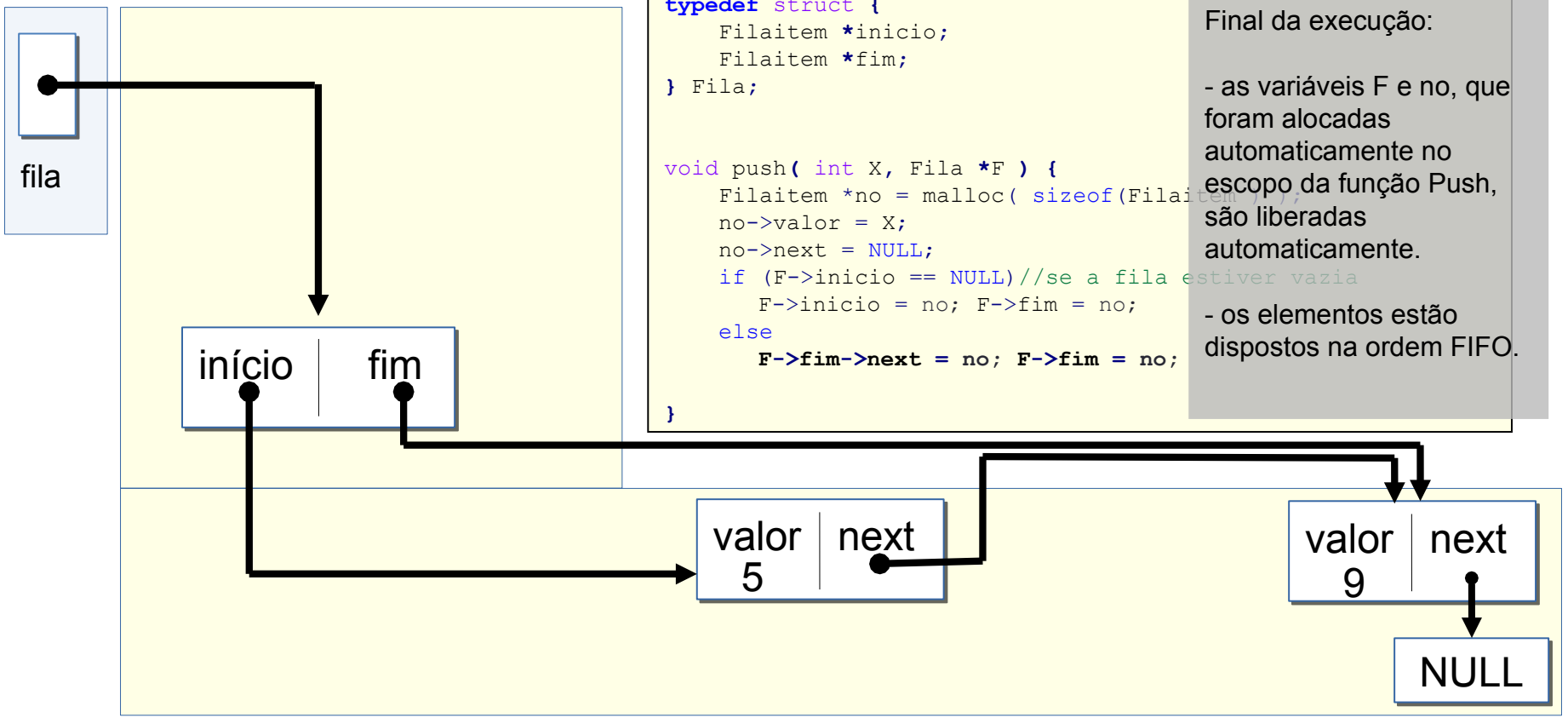
Fila – exemplo : push (slide 9)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    push(9,fila);
    ...
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem) );
    no->valor = X;
    no->next = NULL;
    if (F->inicio == NULL)//se a fila estiver vazia
       F->inicio = no; F->fim = no;
    else
       F->fim->next = no; F->fim = no;

}
```

Final da execução:

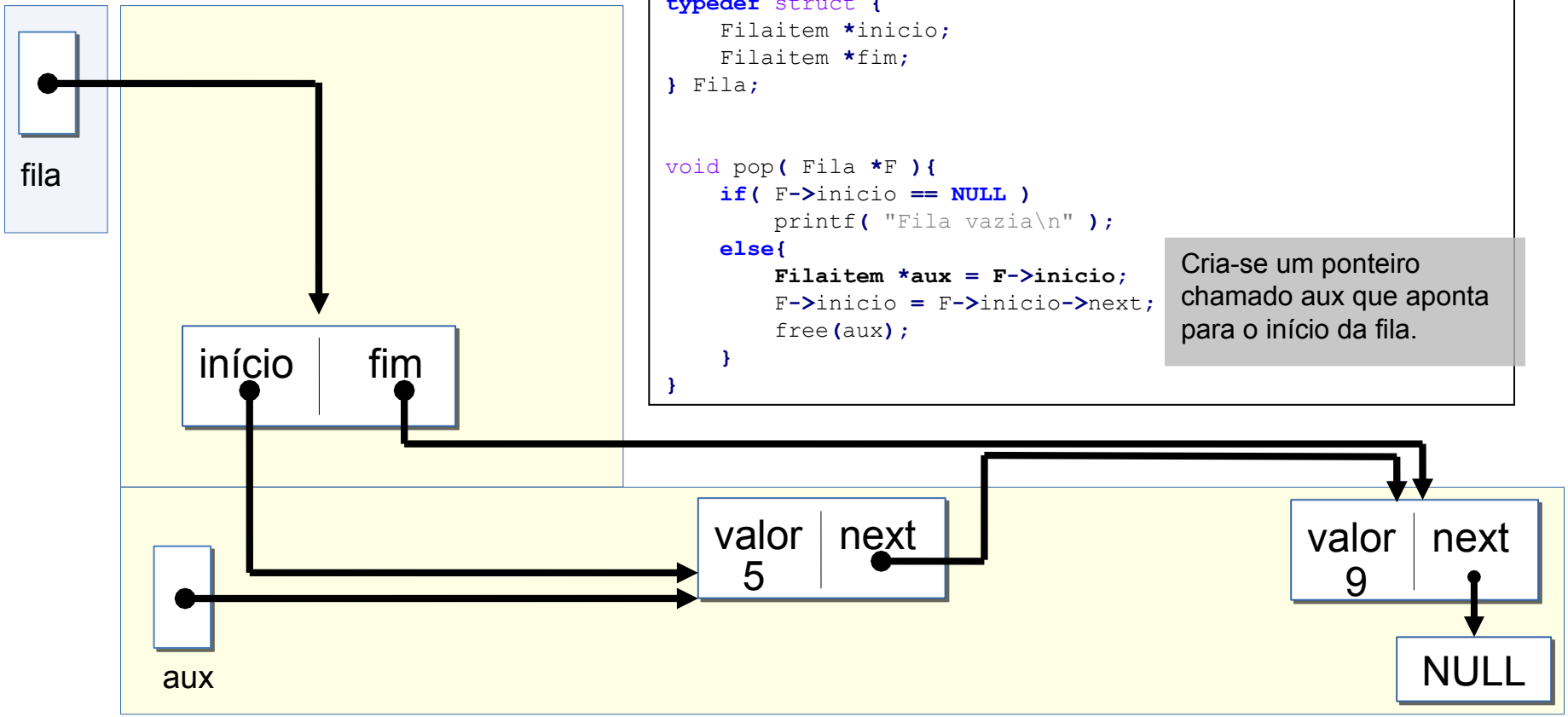
- as variáveis F e no, que foram alocadas automaticamente no escopo da função Push, são liberadas automaticamente.
- os elementos estão dispostos na ordem FIFO.



Fila – exemplo : pop (slide 1)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    push(9,fila);
    pop(fila);
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void pop( Fila *F ){
    if( F->inicio == NULL )
        printf( "Fila vazia\n" );
    else{
        Filaitem *aux = F->inicio;
        F->inicio = F->inicio->next;
        free(aux);
    }
}
```

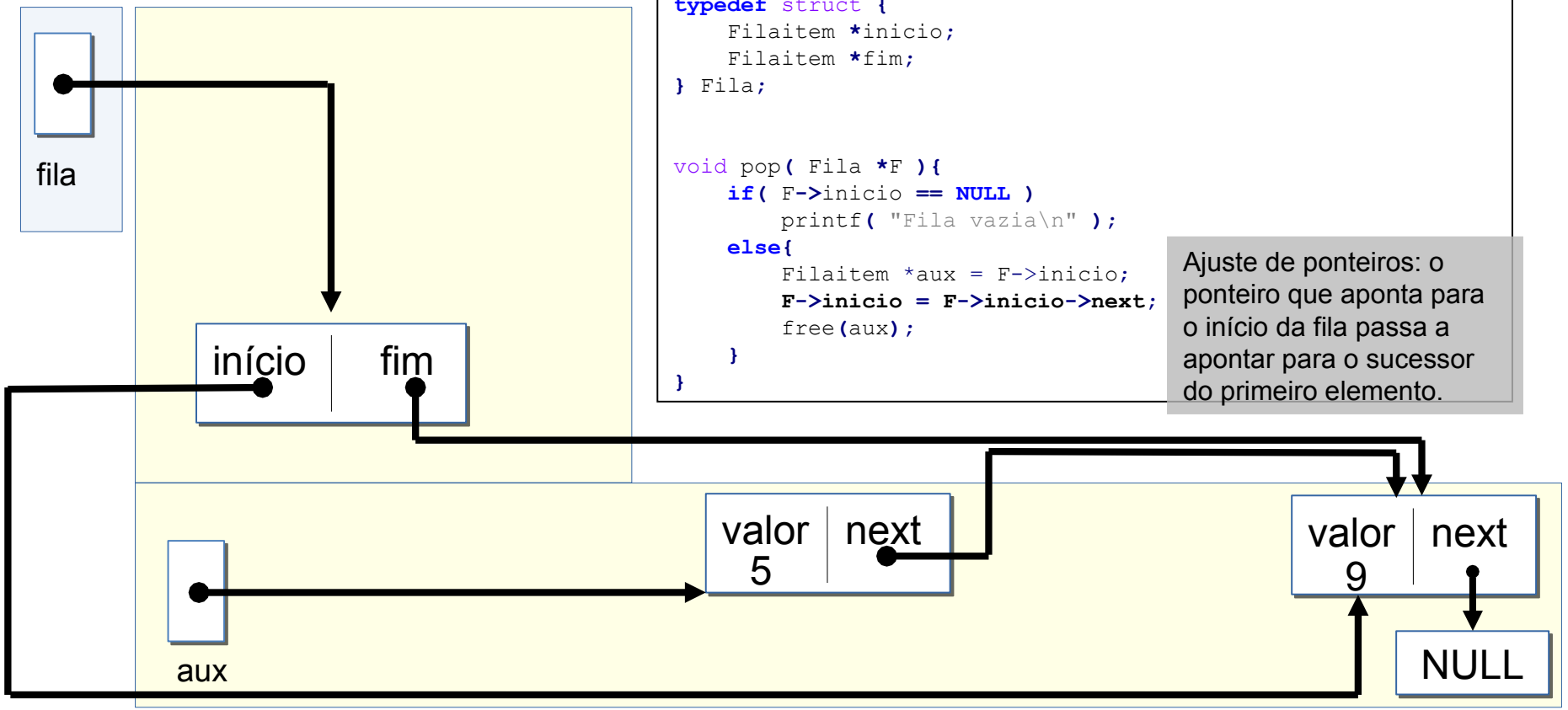


Cria-se um ponteiro chamado aux que aponta para o início da fila.

Fila – exemplo : pop (slide 2)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    push(9,fila);
    pop(fila);
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void pop( Fila *F ){
    if( F->inicio == NULL )
        printf( "Fila vazia\n" );
    else{
        Filaitem *aux = F->inicio;
        F->inicio = F->inicio->next;
        free(aux);
    }
}
```



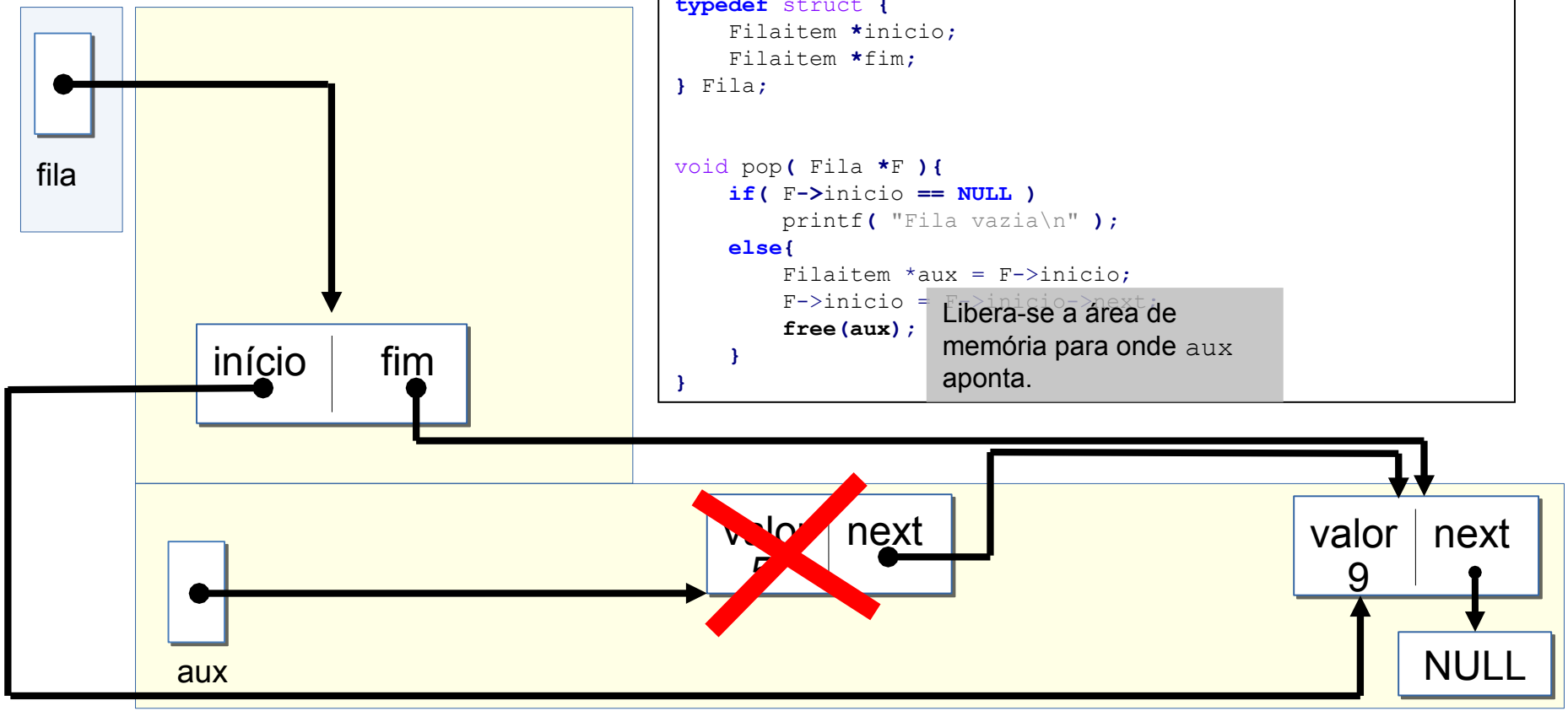
Ajuste de ponteiros: o ponteiro que aponta para o início da fila passa a apontar para o sucessor do primeiro elemento.

Fila – exemplo : pop (slide 3)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    push(9,fila);
    pop(fila);
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void pop( Fila *F ){
    if( F->inicio == NULL )
        printf( "Fila vazia\n" );
    else{
        Filaitem *aux = F->inicio;
        F->inicio = F->inicio->next;
        free(aux);
    }
}
```

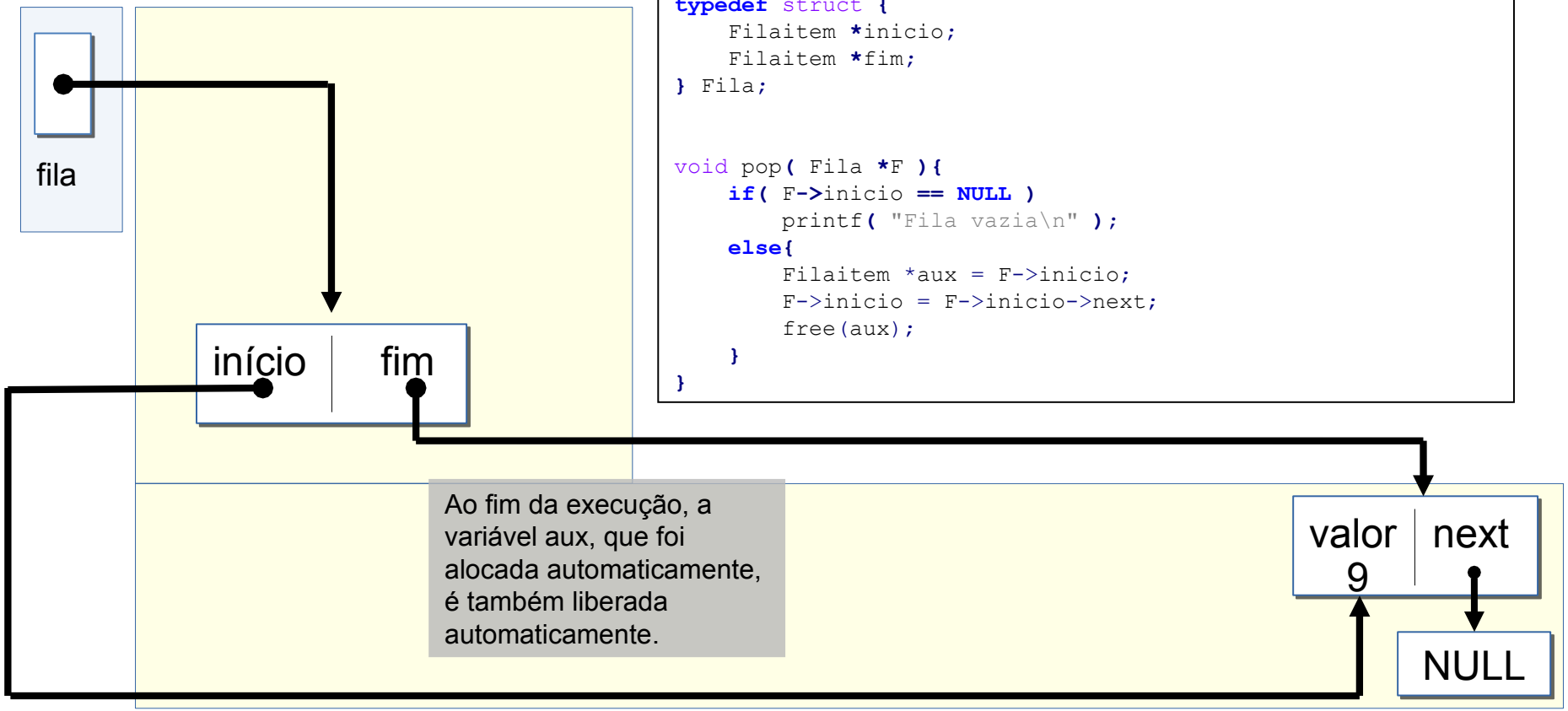
Liberta-se a área de memória para onde `aux` aponta.



Fila – exemplo : pop (slide 4)

```c
int main(){
    Fila *fila = criaFila();
    push(5,fila);
    push(9,fila);
    pop(fila);
}
```

```c
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* ponteiro para para o elemento seguinte */
    struct filaItem *next;
} Filaitem;

typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
} Fila;


void pop( Fila *F ){
    if( F->inicio == NULL )
        printf( "Fila vazia\n" );
    else{
        Filaitem *aux = F->inicio;
        F->inicio = F->inicio->next;
        free(aux);
    }
}
```



Fila - Complexidade da Implementação baseada em Lista Ligada

```
void push( int X, Fila *F ) {
    Filaitem *no = malloc( sizeof(Filaitem ) );
    if( no == NULL ) {
        printf( "Push: sem espaco na memoria!!!\n" );
        exit(EXIT_FAILURE);
    }
    no->valor = X;
    no->next = NULL;

    if (F->inicio == NULL){ //se a fila estiver vazia
            F->inicio = no; F->fim = no;
      } else {
            F->fim->next = no; F->fim = no;
      }
}
```

```
void pop( Fila *F )
{
    if( F->inicio == NULL )
        printf( "Fila vazia\n" );
    else
    {
        Filaitem *aux = F->inicio;
        F->inicio = F->inicio->next;
        free(aux);
    }
}
```

Os elementos a serem inseridos/retirados são acessados diretamente (isto é, não é necessário percorrer a fila).

Assim, as operações *push* e *pop* têm complexidade *O(1).*

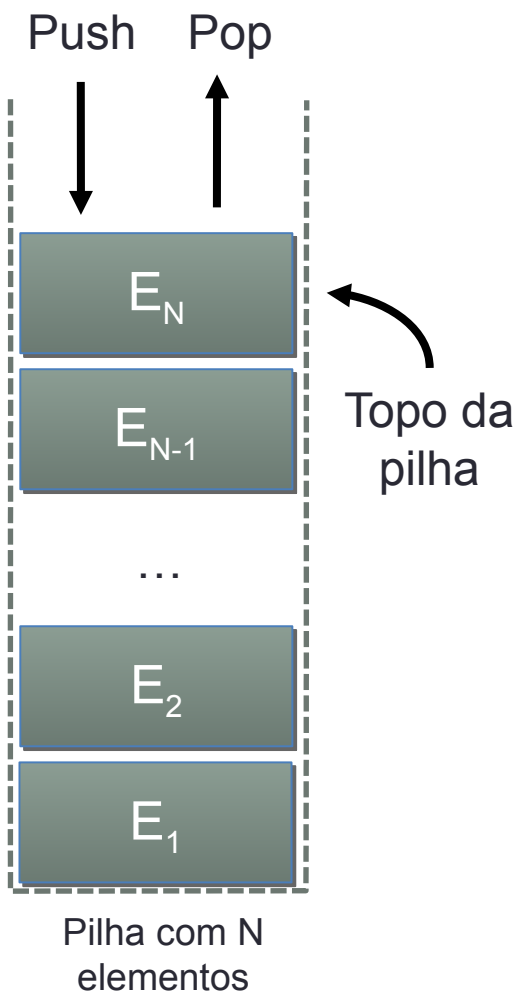
Ou seja, estas operações sempre terão o mesmo custo computacional, independente da quantidade de elementos armazenados na fila.

Pilha (*Stack*)

- Uma pilha também pode ser considerada como uma **lista com restrição de acesso**.
- A inserção e remoção de elementos se faz pela mesma extremidade (topo da pilha)
- O elemento eliminado é sempre o mais recente (cabeça)
- Os elementos da pilha são retirados na ordem inversa à que foram introduzidos: LIFO (Last-In-First-Out)

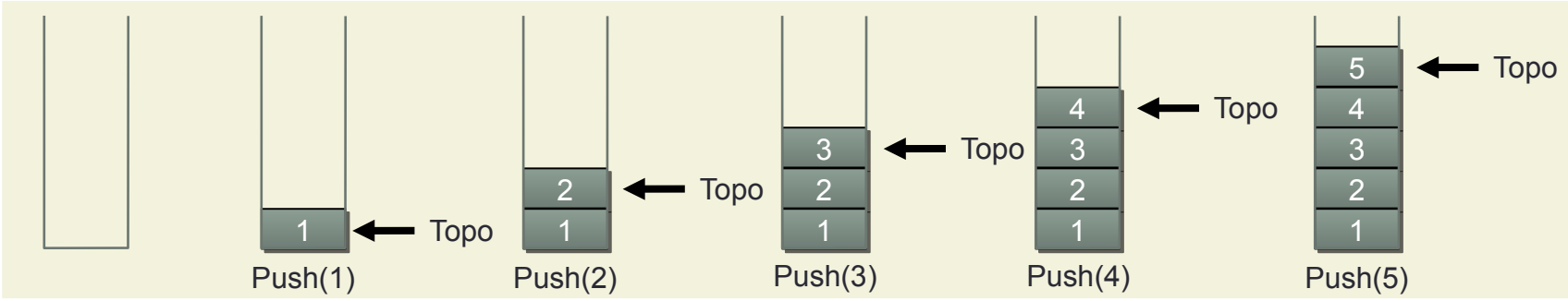
Exemplos :

- Pilha de pratos em um restaurante
- Vagões de um trem
- Retirada de mercadorias em um caminhão de entregas



Pilha / Stack

- **Operações comuns:**
  - criar uma pilha vazia
  - adicionar (push)/remover(pop) um elemento à pilha
  - verificar qual é o topo da pilha (último elemento adicionado)



Implementação da Pilha / *Stack*

A implementação da pilha pode ser feita com:

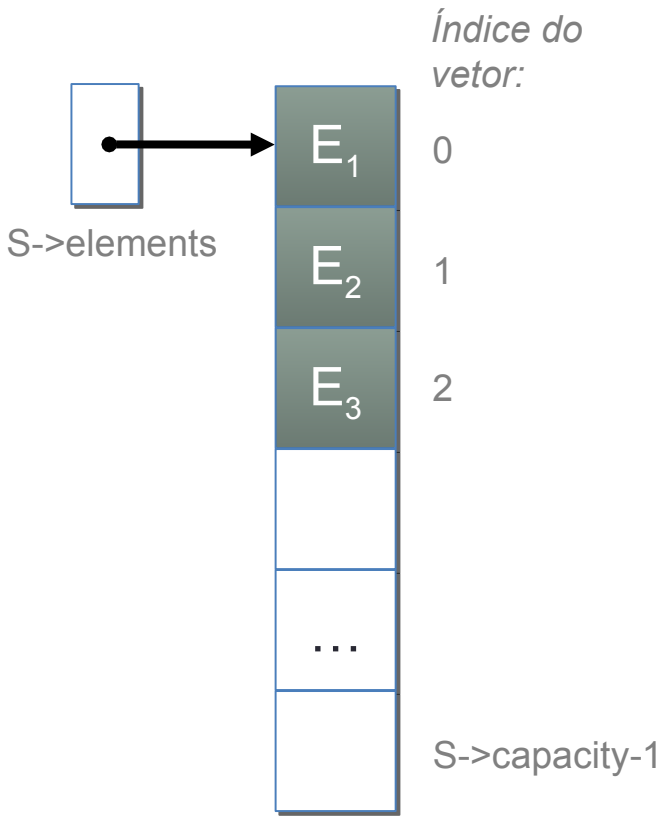
- Um **vetor**
  - Capacidade do vetor é pré-definida e é necessário saber o índice do elemento que está no topo da pilha.
  - **Push**: Se o vetor não estiver cheio, insere novo elemento na primeira posição vazia e incrementa o índice do topo.
  - **Pop**: Se o vetor não estiver vazio, remove o elemento que cujo índice corresponde ao topo da pilha e decrementa o índice.

- Uma **lista ligada**
  - **Push**: Insere novo elemento no início da lista.
  - **Pop**: Remove o elemento do início da lista.

Pilha (Implementação baseada em Vetor)

```c
#define PILHA_VAZIA -1
#define MIN_TAM_PILHA 5
struct pilhaItem
{
    int capacidade; /* capacidade da pilha*/

    int topo; /* índice do topo da pilha*/
    int *elem; /* vetor elementos */

};
typedef struct pilhaItem* Pilha;
```

```c
/* cria uma nova pilha (stack) */
Pilha criaPilha( int maxSize );

/* insere um novo elemento no topo */
void Push( int X, Pilha S ) ;

/* remove o elemento do topo */
void Pop( Pilha S );

/* obtém o valor do elemento do topo */
int Top( Pilha S );
```



Pilha (Implementação baseada em Vetor)

```
Pilha criaPilha( int maxSize )
{
    Pilha S;
    if( maxSize < MIN_TAM_PILHA ) {
        printf("Pilha size is too small\n" ); exit(EXIT_FAILURE);
    }

    S = malloc( sizeof(struct pilhaItem) );
    if( S == NULL ) {
        printf( "Out of space!\n" ); exit(EXIT_FAILURE);
    }
    S->elem = malloc( sizeof(int) * maxSize );
    if( S->elem == NULL ) {
        printf( "Out of space!\n" );
        exit(EXIT_FAILURE);
    }
    S->capacidade = maxSize;
    S->topo = PILHA_VAZIA;
    return S;
}
```

Pilha (Implementação baseada em Vetor)

```
void Push( int X, Pilha S )
{
    if( S->topo == S->capacidade - 1 ) {
        printf( “Pilha Cheia\n" ); exit(EXIT_FAILURE);
    }
    else S->elem[ ++S->topo ] = X;
}

void Pop( Pilha S )
{
    if( S->topo == PILHA_VAZIA )
        printf( “Pilha Vazia\n" );
    else S->topo--;
}

int Top( Pilha S )
{
    if( S->topo != PILHA_VAZIA )
        return S->elem[ S->topo ];
    printf( “Pilha cheia" ); exit(EXIT_FAILURE);
    return 0;
}
```

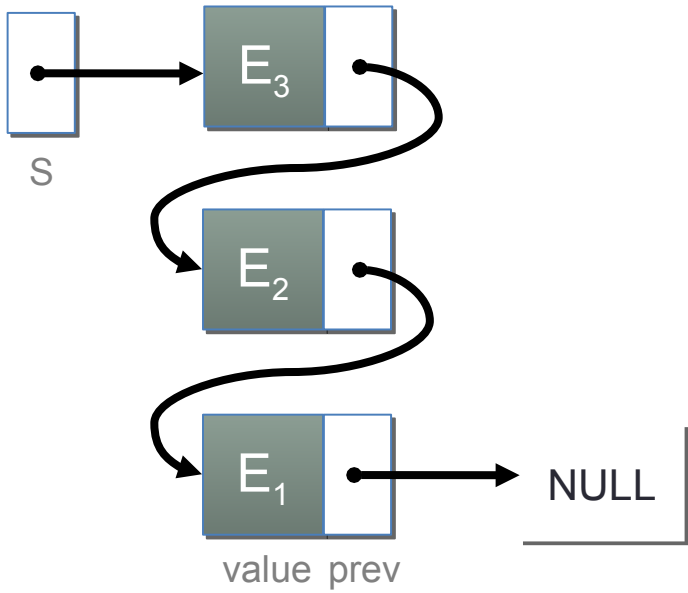
Pilha (Implementação baseada em Vetor)

```
int main( )
{
    Pilha S;
    int i;

    S = criaPilha( 15 );
    for( i = 0; i < 10; i++ )
        Push( i, S );

    while( S->topo != PILHA_VAZIA )
    {
        printf("Topo: %d\n", Top( S ) );
        Pop( S );
    }

    if( S != NULL ) {
        free( S->elem ); free( S );
    }
}
```

Pilha (Implementação baseada em Lista Ligada)

```
typedef struct pilhaItem
{
   int valor; /* valor do elemento da pilha */
   struct pilhaItem *prev; /* apontador para o elemento anterior */
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;
```

```
/* cria uma nova pilha (stack) */
Pilha *criaPilha( );

/* insere um novo elemento no topo */
void Push( int X, Pilha *pilha ) ;

/* remove o elemento do topo */
void Pop( Pilha *pilha );

/* obtém o valor do elemento do topo */
int Peek( Pilha *pilha );
```



Pilha (Implementação baseada em Lista)

```
Pilha *criaPilha( )
{
     Pilha *pilha = malloc( sizeof( Pilha ) );
     if( pilha == NULL ) {
          printf("Sem espaço na memória!!!\n"); exit(EXIT_FAILURE);
     }
     pilha->inicio = NULL;
     return pilha;
}


void Push( int X, Pilha *pilha )
{
     Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
     if( tmp == NULL ) {
          printf( "Push: Sem espaço na memória!!!\n" );
          exit(EXIT_FAILURE);
     }
     else {
          tmp->valor = X;
          tmp->prev = P->inicio;
          pilha->inicio = tmp;
     }
}
```

Pilha (Implementação baseada em Lista)

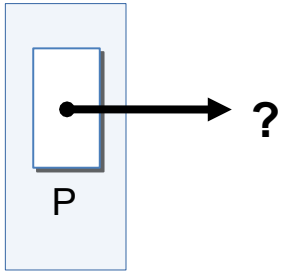
```
void Pop( Pilha *pilha )
{
    Pilhaitem *firstElem;
    if( pilha->inicio == NULL )
        printf( "Pilha vazia\n" );
    else
    {
        firstElem = pilha->inicio;
        pilha->inicio = pilha->inicio->prev;
        free( firstElem );
    }
}

int Peek( Pilha *pilha )
{
    if( pilha->inicio != NULL )
        return pilha->inicio->valor;
    printf( "Pilha vazia\n" );
    return -1;
}
```

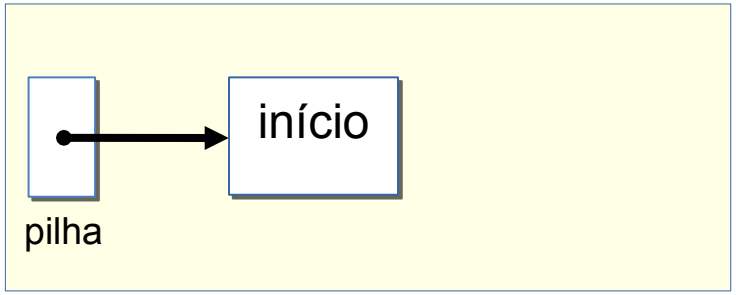
Pilha (Implementação baseada em Lista)

```
int main( )
{
     Pilha *P = criaPilha( );
     int i;

          for( i = 0; i < 10; i++ )
          Push( i, P );

     while( P->inicio != NULL )
     {
          printf( "%d\n", Peek( P ) );
          Pop( P );
     }
     free( P );
}
```

Pilha – exemplo : criação de uma pilha (slide 1/3)

```
int main(){
    Pilha *P = criaPilha();
    ...
    ...
    ...
}
```






```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


Pilha *criaPilha() {
    Pilha *pilha = malloc( sizeof( Pilha ) );


}
```

A *struct* que armazena uma pilha consiste em um dado e um ponteiro para o elemento anterior.

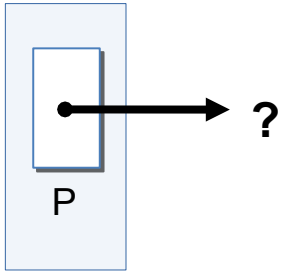
Uma *struct* que armazena uma pilha consiste um ponteiro para o topo da pilha.

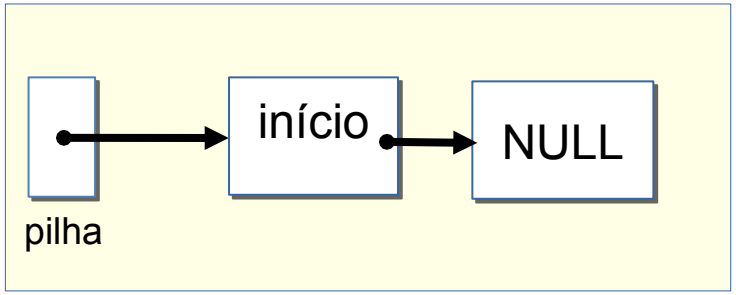
Alocação de uma porção de memória para armazenar uma *struct* do tipo `Pilha`.

O endereço da porção de memória alocada é atribuído ao ponteiro `pilha`.

Pilha – exemplo : criação de uma pilha (slide 2/3)

```
int main(){
    Pilha *P = criaPilha();
    ...
    ...
    ...
}
```






```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


Pilha *criaPilha() {
    Pilha *pilha = malloc( sizeof( Pilha ) );
    pilha->inicio = NULL;

}
```

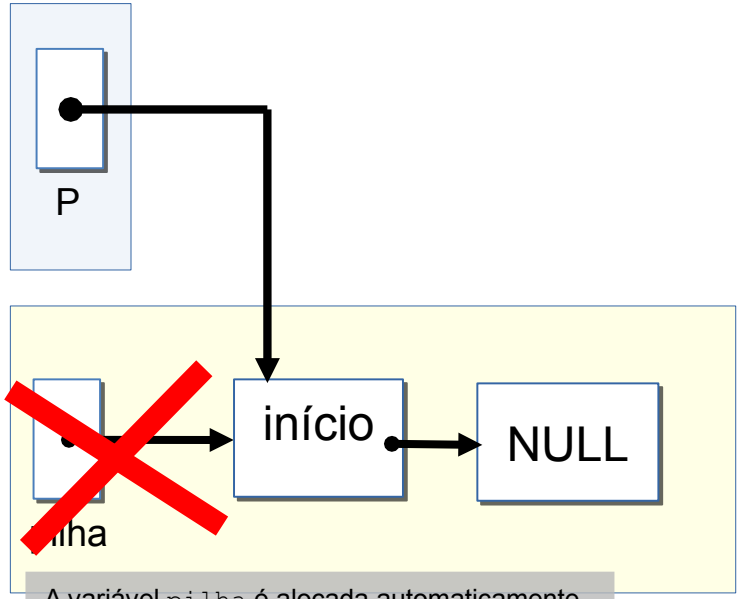
A *struct* que armazena uma pilha consiste em um dado e um ponteiro para o elemento anterior.

Uma *struct* que armazena uma pilha consiste um ponteiro para o topo da pilha.

O topo de uma pilha vazia aponta para NULL.

Pilha – exemplo : criação de uma pilha (slide 3/3)

```
int main(){
    Pilha *P = criaPilha();
    ...
    ...
    ...
}
```



A variável `pilha` é alocada automaticamente no escopo da função `criaPilha` e, por isso, é liberada automaticamente quando a função é finalizada.

```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;

typedef struct {
    Pilhaitem *inicio;
} Pilha;

Pilha *criaPilha() {
    Pilha *pilha = malloc( sizeof( Pilha ) );
    pilha->inicio = NULL;
    return pilha;
}
```

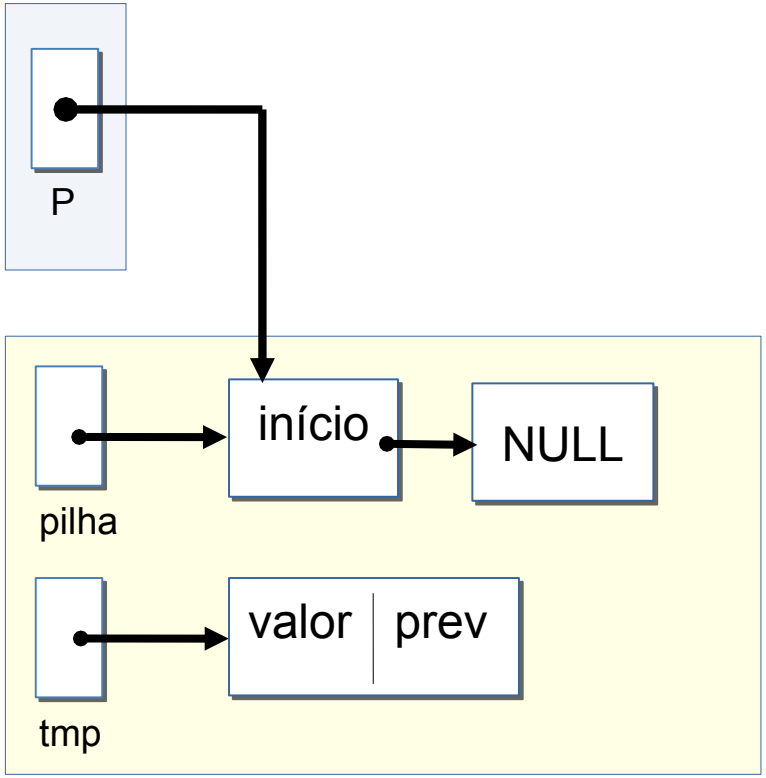
A *struct* que armazena uma pilha consiste em um dado e um ponteiro para o elemento anterior.

Uma *struct* que armazena uma pilha consiste um ponteiro para o topo da pilha.

O topo de uma pilha vazia aponta para NULL.

Pilha – exemplo : push (slide 1/9)

```
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    ...
    ...
}
```




```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void push( int X, Pilha *pilha ) {
      Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );



}
```

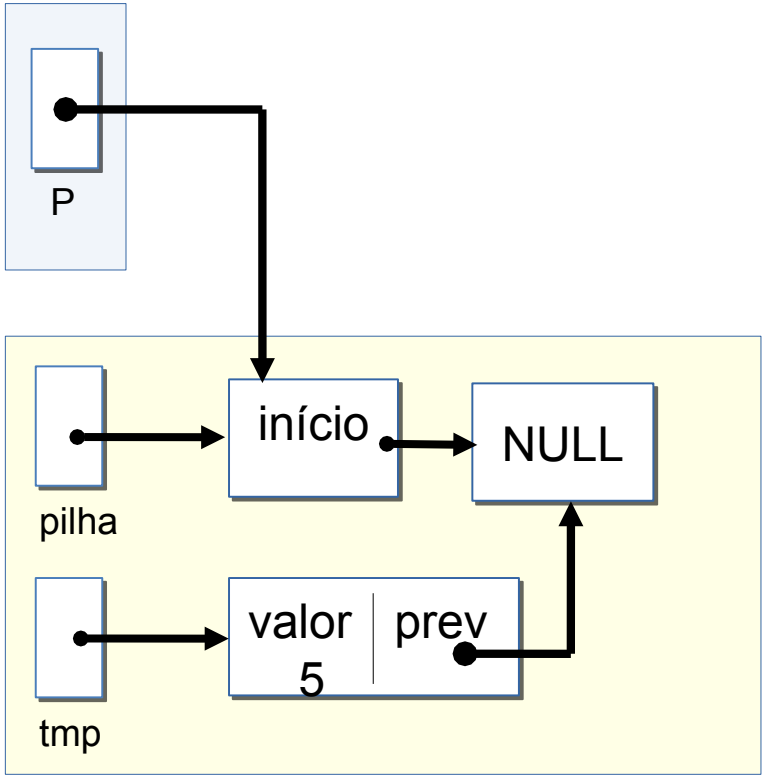
Alocação de uma porção de memória para armazenar uma *struct* do tipo `Pilha`.

O endereço da porção de memória alocada é atribuído ao ponteiro `tmp`.

O ponteiro `pilha`, que pertence ao escopo da função `push`, aponta para a mesma área de memória para onde aponta o ponteiro `P`, que pertence ao escopo da função `main`.

Pilha – exemplo : push (slide 2/9)

```
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    ...
    ...
}
```




```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void push( int X, Pilha *pilha ) {
      Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
      tmp->valor = X;
      tmp->prev = pilha->inicio;

}
```

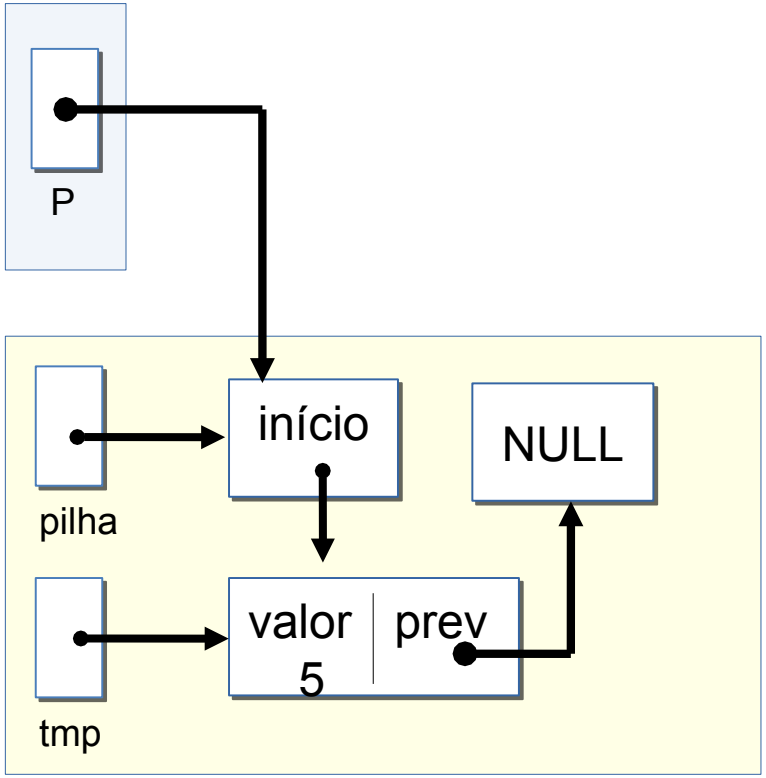
Atribuição de valores.

O elemento recém criado aponta para a mesma área de memória que ocupa o topo da pilha (neste caso, NULL).

O elemento recém criado ainda não faz parte da pilha.

Pilha – exemplo : push (slide 3/9)

```
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    ...
    ...
}
```

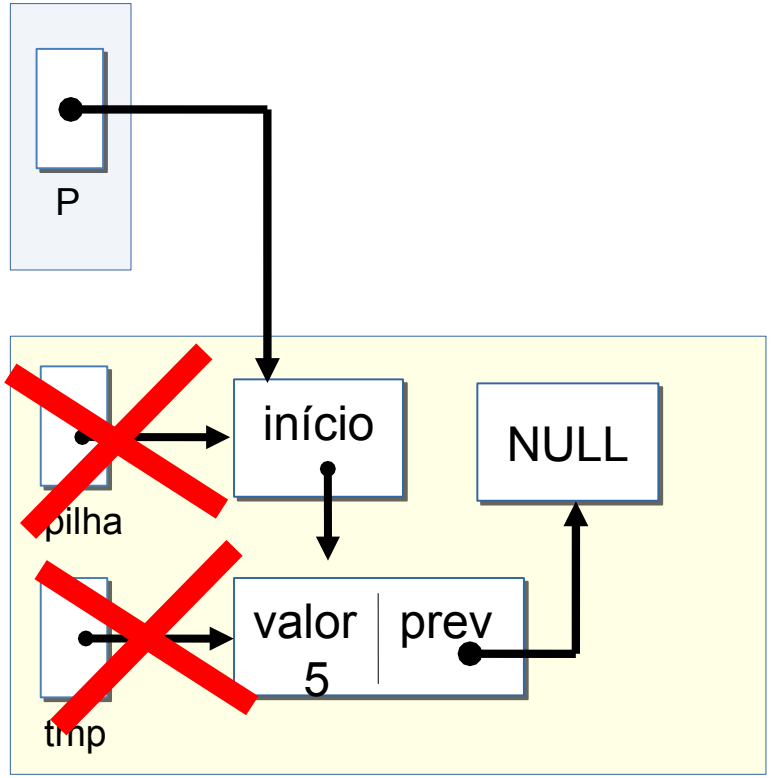



```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void push( int X, Pilha *pilha ) {
      Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
      tmp->valor = X;
      tmp->prev = pilha->inicio;
      pilha->inicio = tmp;
}
```

O elemento recém criado passa a ocupar o topo da pilha.

Pilha – exemplo : push (slide 4/9)

```
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    ...
    ...
}
```



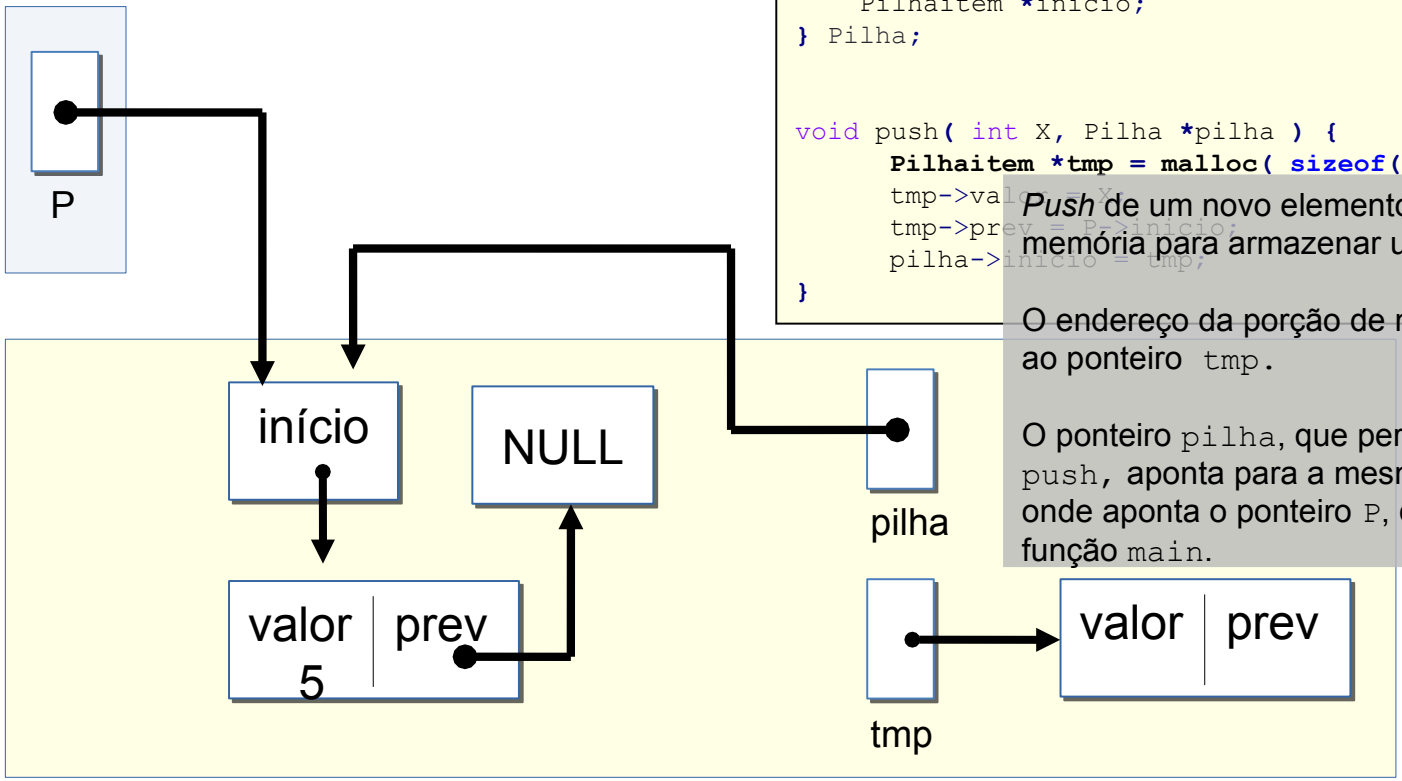

```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void push( int X, Pilha *pilha ) {
      Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
      tmp->valor = X;
      tmp->prev = P->inicio;
      pilha->inicio = tmp;
}
```

Ao final da execução, as variáveis `pilha` e `tmp`, que foram alocadas automaticamente no escopo da função `Push`, são liberadas automaticamente.

Pilha – exemplo : push (slide 5/9)

```c
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    ...
}
```

```c
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void push( int X, Pilha *pilha ) {
      Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
      tmp->valor = X;
      tmp->prev = P->inicio;
      pilha->inicio = tmp;
}
```



Push de um novo elemento: Alocação de uma porção de memória para armazenar uma *struct* do tipo `Pilha`.

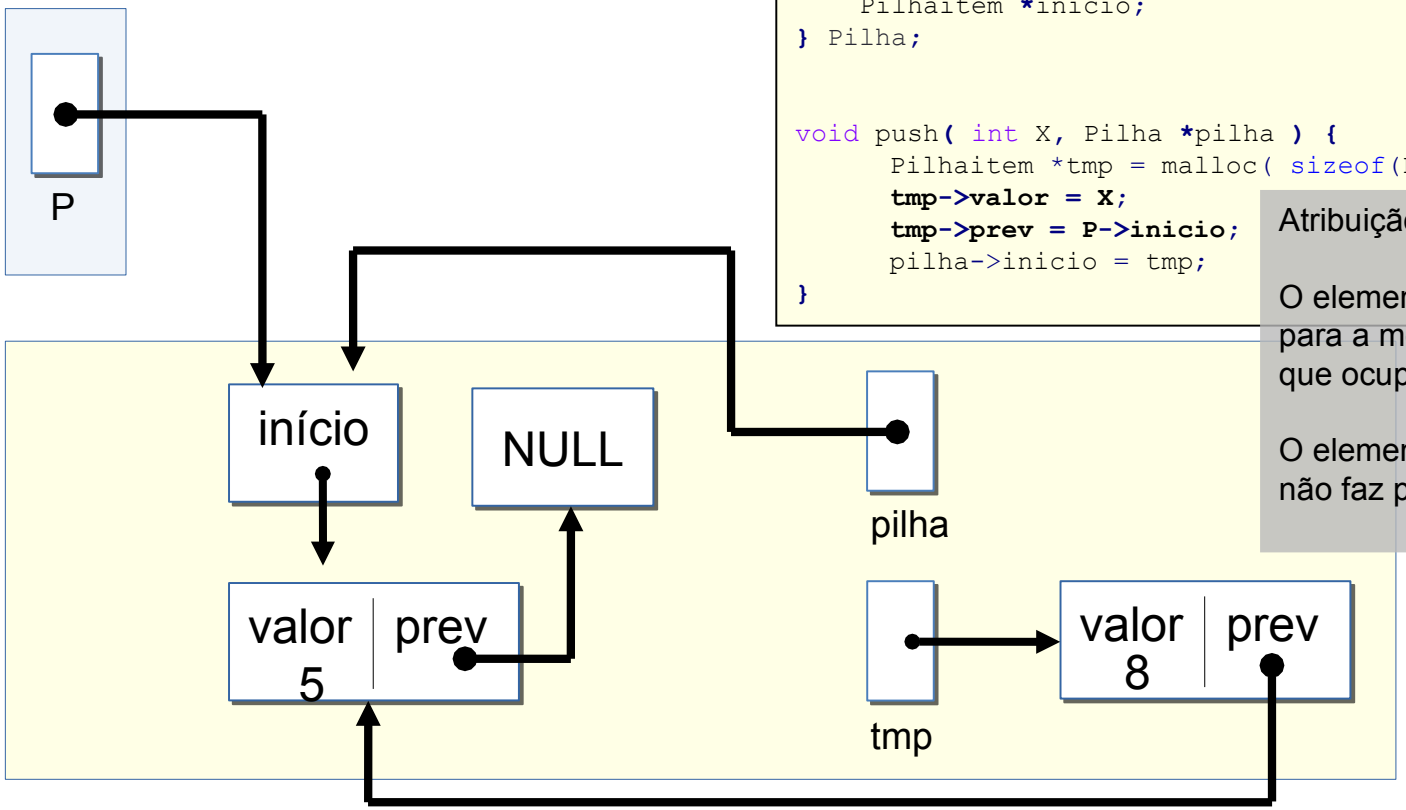
O endereço da porção de memória alocada é atribuído ao ponteiro `tmp`.

O ponteiro `pilha`, que pertence ao escopo da função `push`, aponta para a mesma área de memória para onde aponta o ponteiro `P`, que pertence ao escopo da função `main`.

Pilha – exemplo : push (slide 6/9)

```c
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    ...
}
```

```c
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void push( int X, Pilha *pilha ) {
      Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
      tmp->valor = X;
      tmp->prev = P->inicio;
      pilha->inicio = tmp;
}
```



Atribuição de valores.

O elemento recém criado aponta para a mesma área de memória que ocupa o topo da pilha.

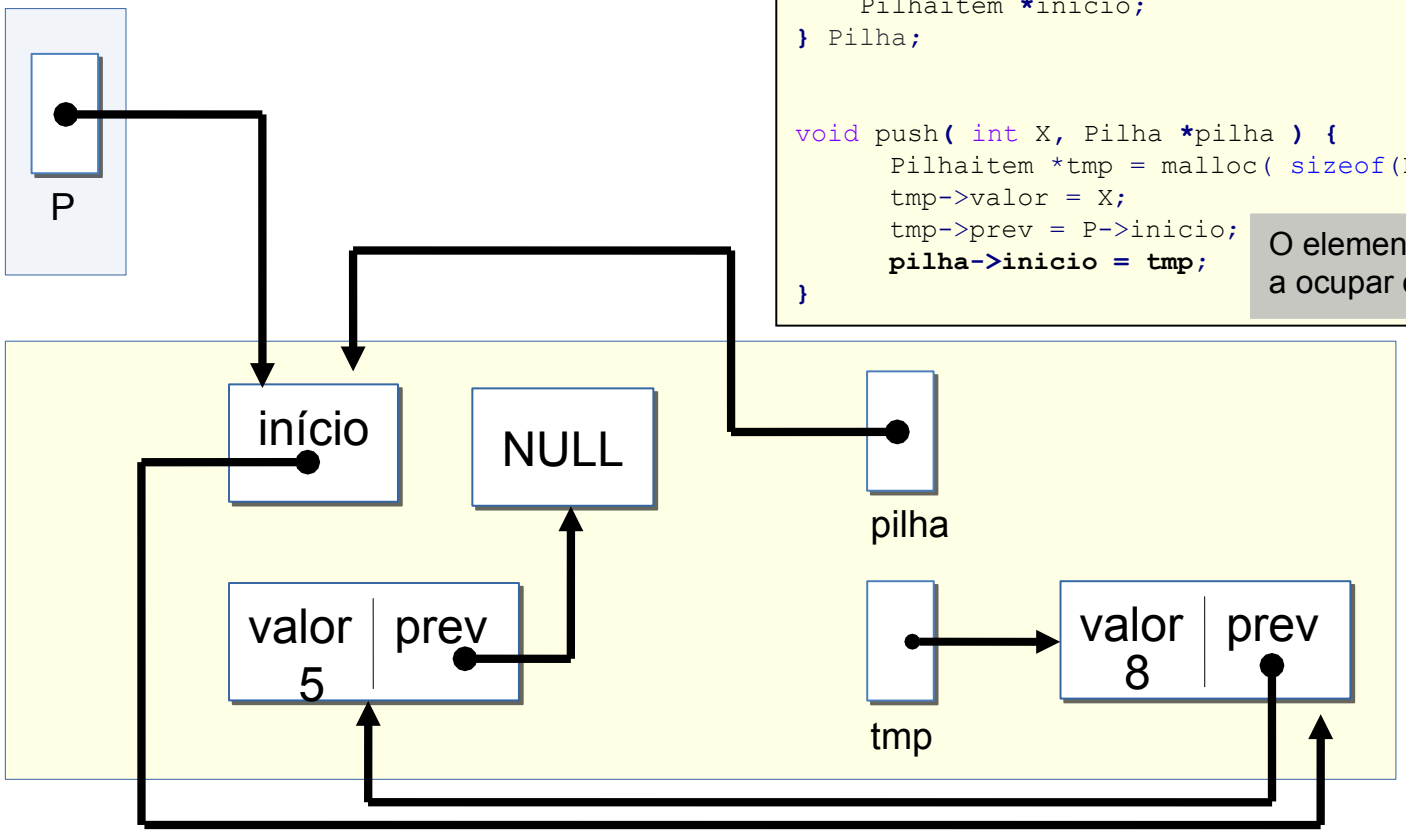
O elemento recém criado ainda não faz parte da pilha.

Pilha – exemplo : push (slide 7/9)

```
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    ...
}
```

```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void push( int X, Pilha *pilha ) {
      Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
      tmp->valor = X;
      tmp->prev = P->inicio;
      pilha->inicio = tmp;
}
```

O elemento recém criado passa a ocupar o topo da pilha.

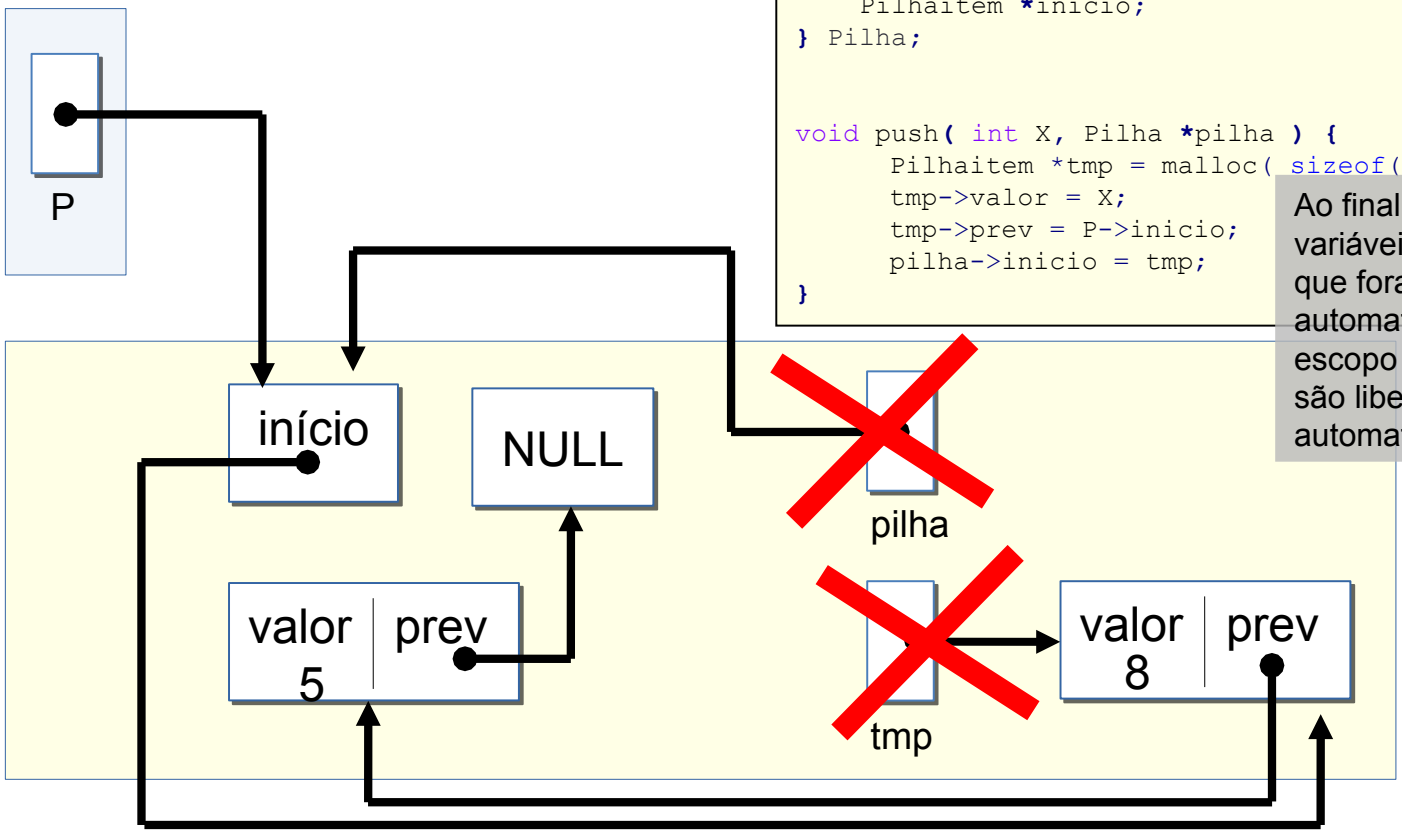


Pilha – exemplo : push (slide 8/9)

```
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    ...
}
```

```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void push( int X, Pilha *pilha ) {
      Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
      tmp->valor = X;
      tmp->prev = P->inicio;
      pilha->inicio = tmp;
}
```

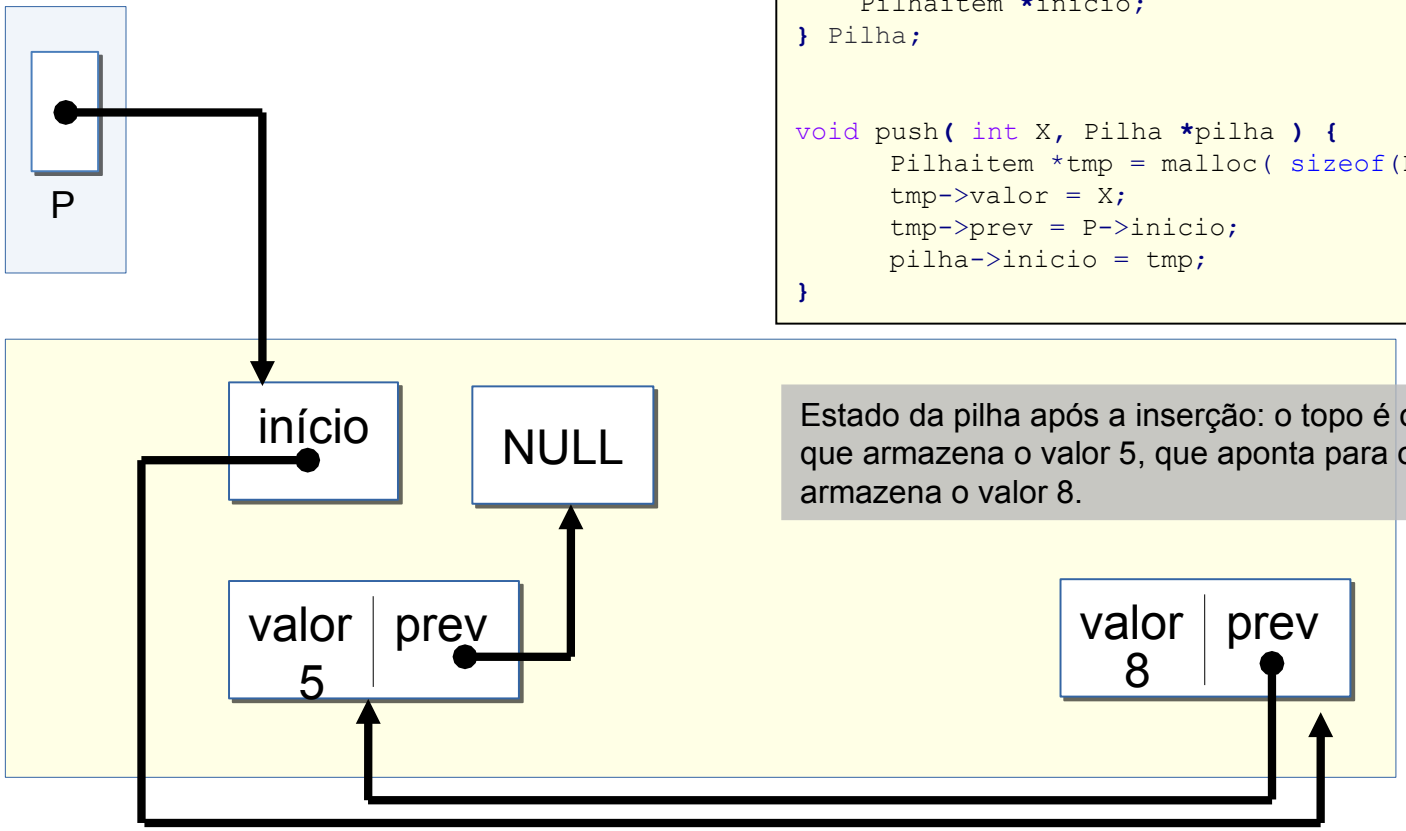
Ao final da execução, as variáveis `pilha` e `tmp`, que foram alocadas automaticamente no escopo da função `Push`, são liberadas automaticamente.



Pilha – exemplo : push (slide 9/9)

```c
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    ...
}
```

```c
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void push( int X, Pilha *pilha ) {
      Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
      tmp->valor = X;
      tmp->prev = P->inicio;
      pilha->inicio = tmp;
}
```

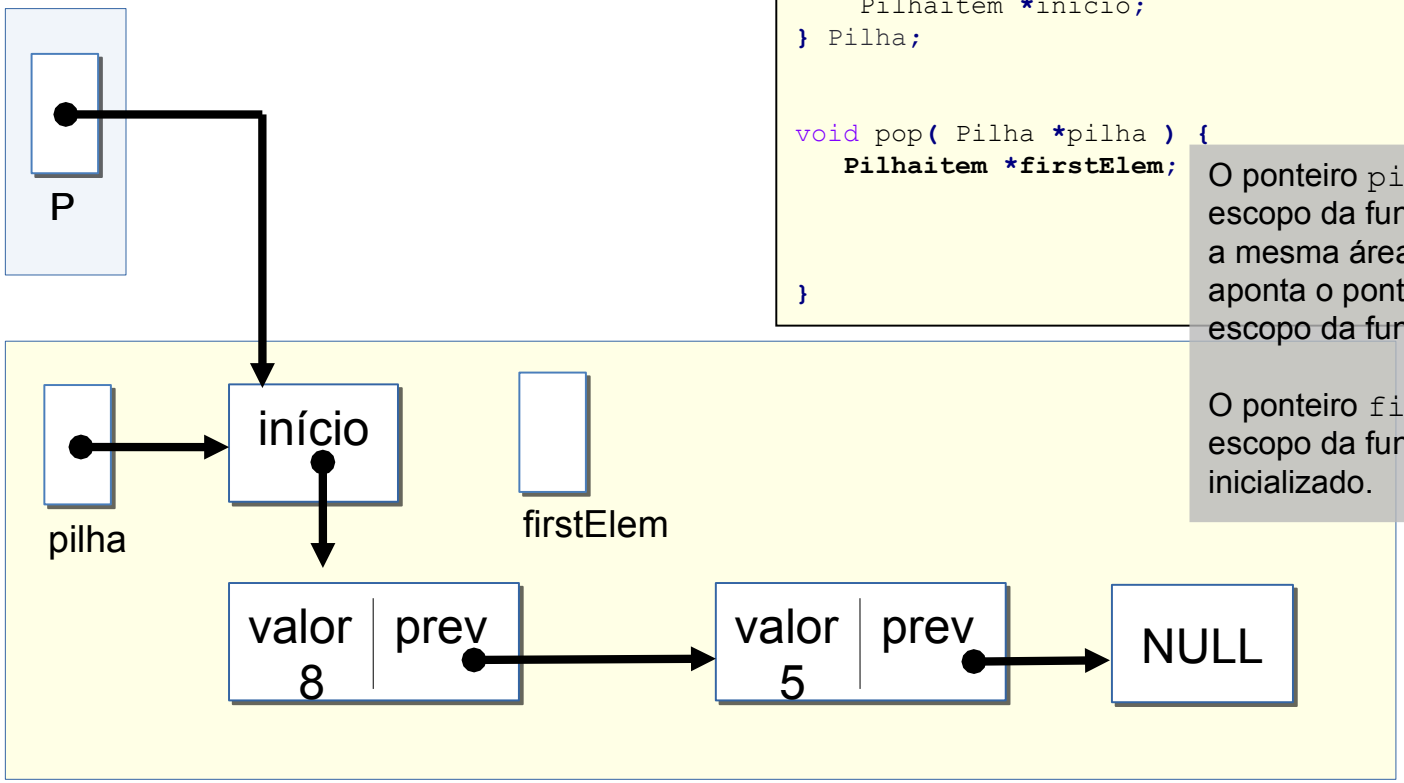


Estado da pilha após a inserção: o topo é ocupado pelo elemento que armazena o valor 5, que aponta para o antecessor, que armazena o valor 8.

Pilha – exemplo : pop (slide 1/5)

```
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    pop(P);
}
```

```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void pop( Pilha *pilha ) {
   Pilhaitem *firstElem;



}
```



O ponteiro `pilha`, que está no escopo da função `Pop`, aponta para a mesma área de memória para que aponta o ponteiro `P`, que está no escopo da função `main`.

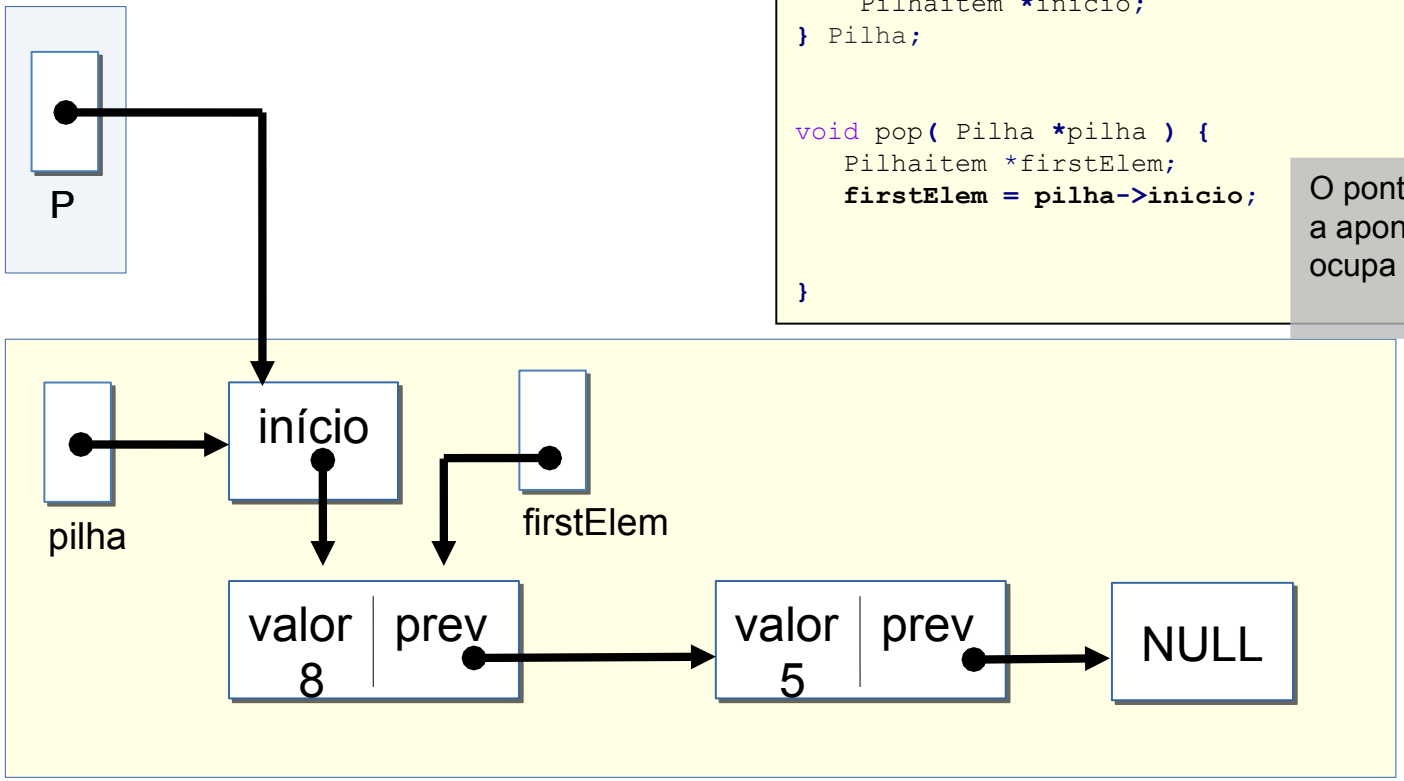
O ponteiro `firstElem`, que está no escopo da função `Pop`, não foi inicializado.

Pilha – exemplo : pop (slide 2/5)

```
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    pop(P);
}
```

```
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void pop( Pilha *pilha ) {
   Pilhaitem *firstElem;
   firstElem = pilha->inicio;


}
```

O ponteiro `firstElem` passa a apontar para o elemento que ocupa o topo da pilha.

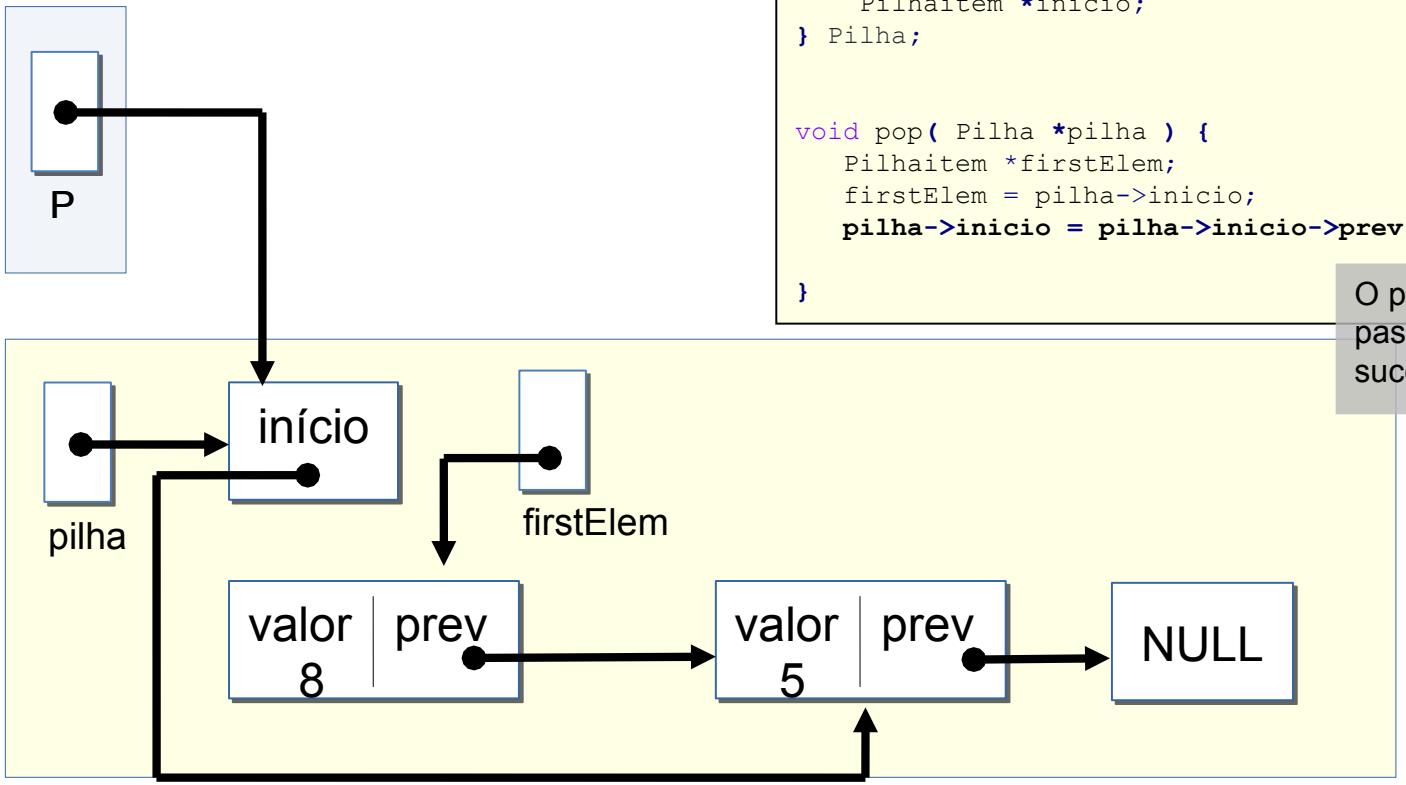


Pilha – exemplo : pop (slide 3/5)

```c
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    pop(P);
}
```

```c
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void pop( Pilha *pilha ) {
   Pilhaitem *firstElem;
   firstElem = pilha->inicio;
   pilha->inicio = pilha->inicio->prev;

}
```

O ponteiro para o topo da pilha passa a apontar para o sucessor do topo atual.

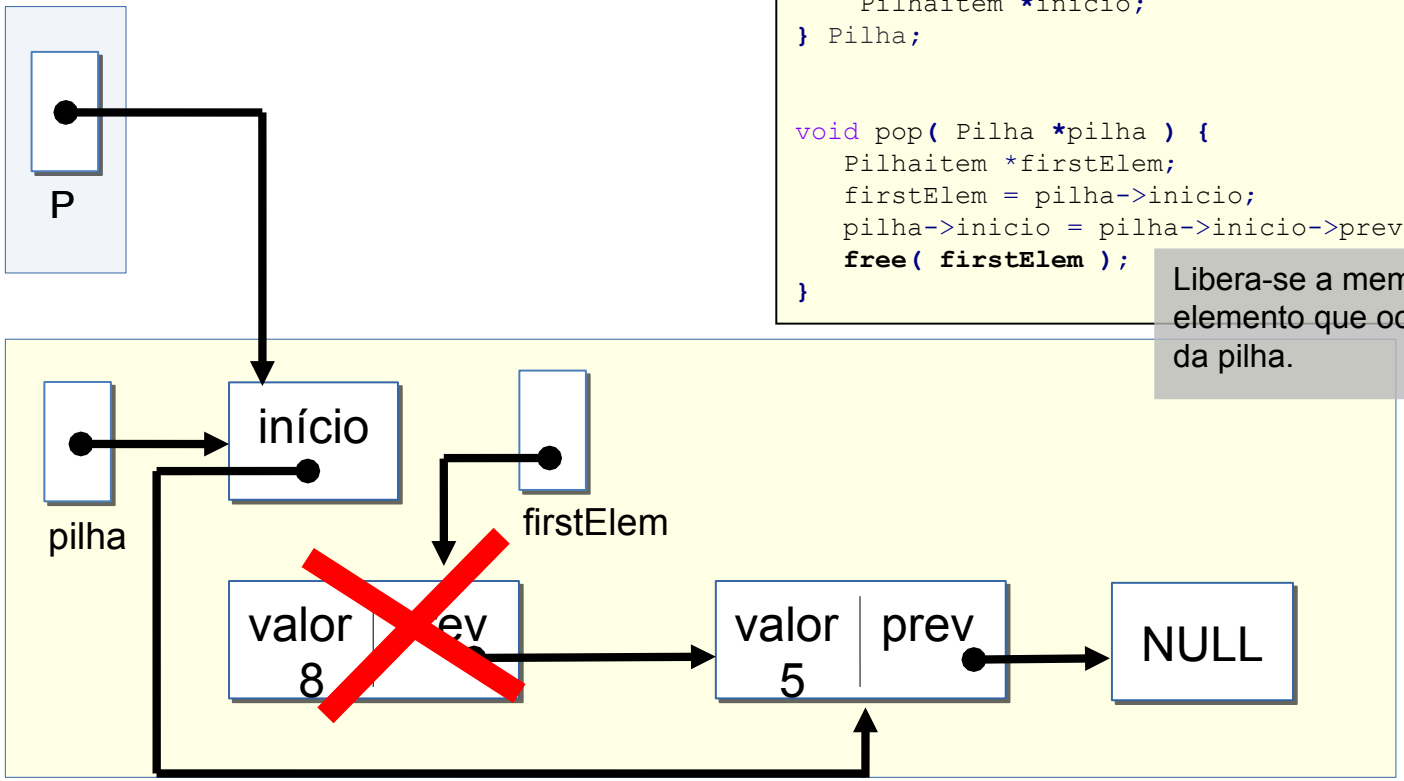


Pilha – exemplo : pop (slide 4/5)

```c
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    pop(P);
}
```

```c
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void pop( Pilha *pilha ) {
   Pilhaitem *firstElem;
   firstElem = pilha->inicio;
   pilha->inicio = pilha->inicio->prev;
   free( firstElem );
}
```

Libera-se a memória do elemento que ocupava o topo da pilha.

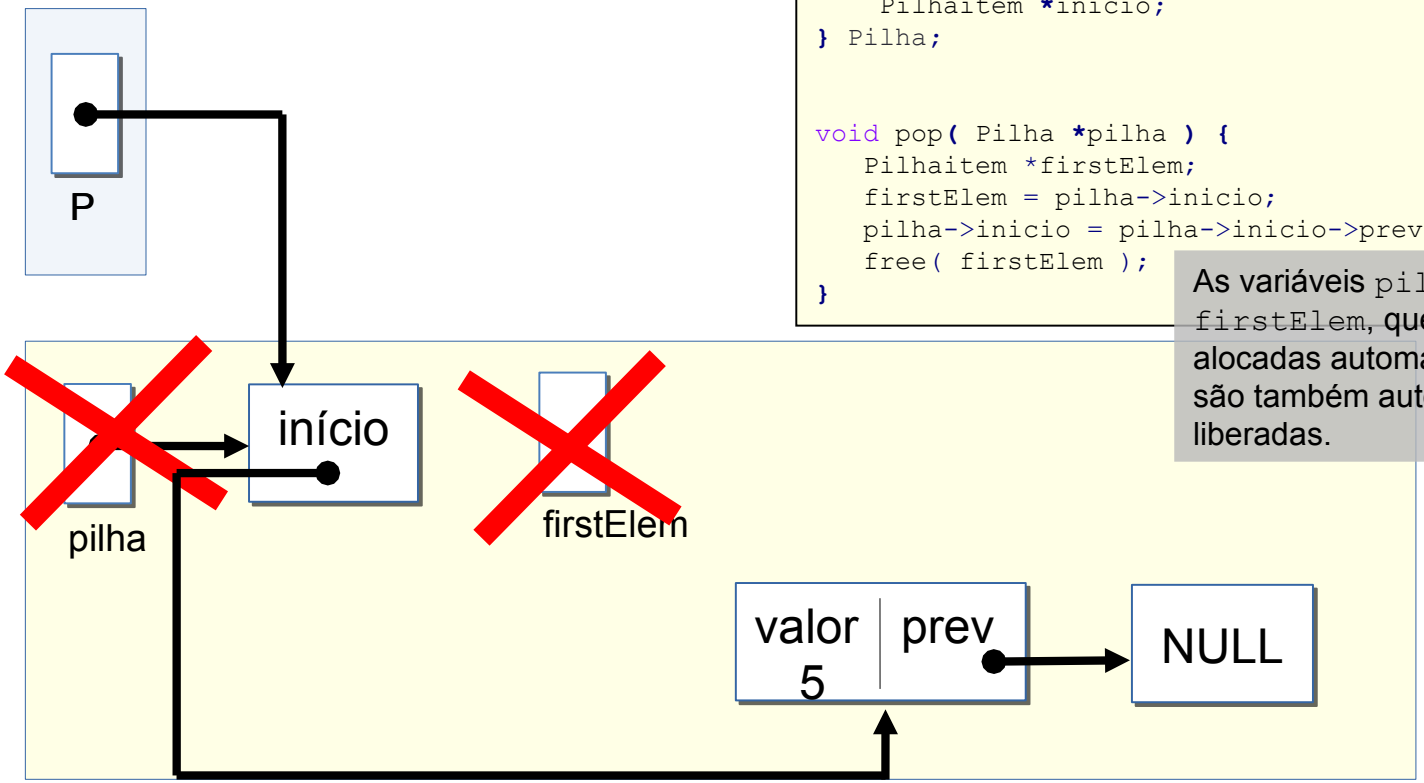


Pilha – exemplo : pop (slide 5/5)

```c
int main(){
    Pilha *P = criaPilha();
    push(5,P);
    push(8,P);
    pop(P);
}
```

```c
typedef struct pilhaItem {
  int valor; /* valor do elemento da pilha */
  struct pilhaItem *prev; /*ponteiro para o anterior*/
} Pilhaitem;


typedef struct {
    Pilhaitem *inicio;
} Pilha;


void pop( Pilha *pilha ) {
   Pilhaitem *firstElem;
   firstElem = pilha->inicio;
   pilha->inicio = pilha->inicio->prev;
   free( firstElem );
}
```

As variáveis `pilha` e `firstElem`, que foram alocadas automaticamente, são também automaticamente liberadas.



Pilha - Complexidade da Implementação baseada em Lista Ligada

```
void push( int X, Pilha *P )
{
   Pilhaitem *tmp = malloc( sizeof(Pilhaitem ) );
   if( tmp == NULL ) {
      printf( "Push: Sem espaço na memória!!!\n" );
      exit(EXIT_FAILURE);
   }
   else {
         tmp->valor = X;
         tmp->prev = P->inicio;
         P->inicio = tmp;
        }
   }
```

```
void pop( Pilha *P )
{
      Pilhaitem *firstElem;
      if( P->inicio == NULL )
            printf( "Pilha vazia\n" );
      else
      {
            firstElem = P->inicio;
            P->inicio = P->inicio->prev;
            free( firstElem );
      }
}
```

Os elementos a serem inseridos/retirados são acessados diretamente (isto é, não é necessário percorrer a fila).

Assim, as operações *push* e *pop* têm complexidade *O(1).*

Ou seja, estas operações sempre terão o mesmo custo computacional, independente da quantidade de elementos armazenados na pilha.

Filas e Pilhas – considerações finais

Filas e pilhas podem ser considerados casos especiais de listas

pois preservam sua características essenciais: (i) as estruturas preservam a noção de precedência entre os elementos e (ii) todo elemento é precedido e sucedido por, no máximo, um elemento.

A diferença entre filas, pilhas e listas está na disciplina de acesso:

cada estrutura impõe diferentes disciplinas para acessar os dados: listas permitem o acesso livre a qualquer elemento (isto é, a disciplina de acesso consiste na ausência de disciplina de acesso) enquanto filas e pilhas implementam, respectivamente, as disciplinas FIFO e LIFO.