

# **AULA 09**

# **ESTRUTURA DE DADOS**

---

**Pilha - implementação dinâmica**

**Norton T. Roman & Luciano A. Digiampietri**

# Pilha

Pilha é uma estrutura linear na qual:

- As **inserções** ocorrem no **topo** da pilha;
- As **exclusões** ocorrem no **topo** da pilha.
- Utiliza a mesma lógica de uma **pilha de papéis**.

# Pilha - implementação dinâmica

# Pilha - implementação dinâmica

- Alocaremos e desalocaremos a memória para os elementos **sob demanda**;

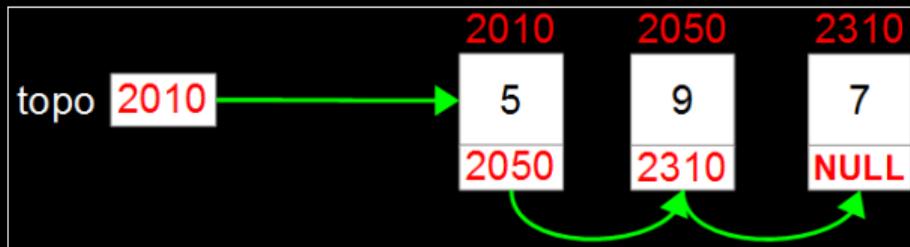
# Pilha - implementação dinâmica

- Alocaremos e desalocaremos a memória para os elementos **sob demanda**;
- Vantagem: não precisamos **gastar memória** que não estamos usando;

# Pilha - implementação dinâmica

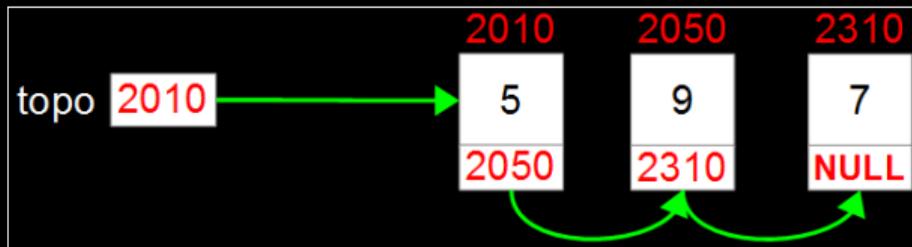
- Alocaremos e desalocaremos a memória para os elementos **sob demanda**;
- Vantagem: não precisamos **gastar memória** que não estamos usando;
- Cada elemento indicará quem é seu **sucessor** (quem está “abaixo” dele na pilha);
- Controlaremos o endereço do elemento que está no **topo** da pilha.

# Ideia



**Temos um campo para indicar o endereço do elemento que está no topo**

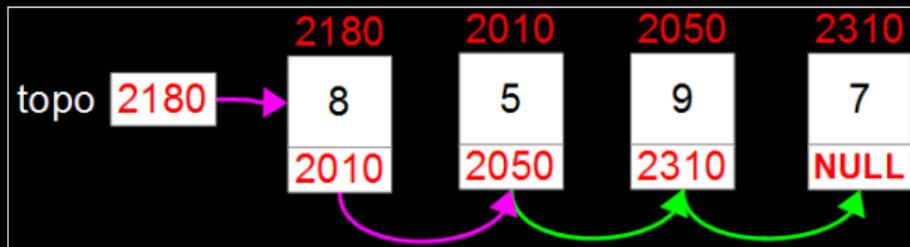
# Ideia



**Temos um campo para indicar o endereço do elemento que está no topo**

**Como inserimos o elemento 8?**

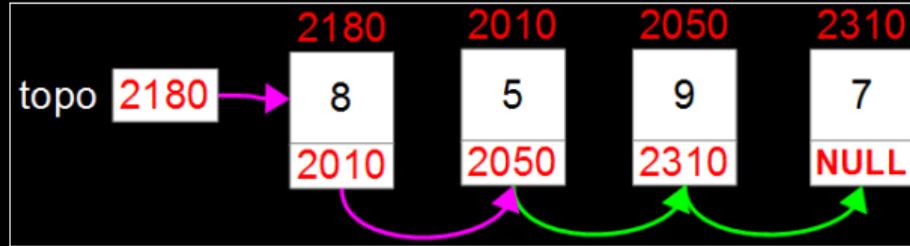
# Ideia



**Temos um campo para indicar o endereço do elemento que está no topo**

**Como inserimos o elemento 8?**

# Ideia

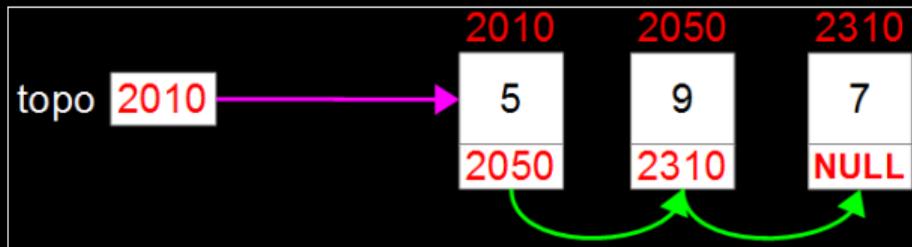


**Temos um campo para indicar o endereço do elemento que está no topo**

**Como inserimos o elemento 8?**

**Como excluimos um elemento?**

# Ideia



**Temos um campo para indicar o endereço do elemento que está no topo**

**Como inserimos o elemento 8?**

**Como excluimos um elemento?**

# Modelagem

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

```
typedef int TIPOCHAVE;
```

```
typedef struct {
    TIPOCHAVE chave;
    // outros campos...
} REGISTRO;
```

```
typedef struct aux {
    REGISTRO reg;
    struct aux* prox;
} ELEMENTO;
```

```
typedef ELEMENTO* PONT;
```

```
typedef struct {
    PONT topo;
} PILHA;
```

# Modelagem

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

typedef int TIPOCHAVE;

typedef struct {
    TIPOCHAVE chave;
    // outros campos...
} REGISTRO;
```

```
typedef struct aux {
    REGISTRO reg;
    struct aux* prox;
} ELEMENTO;

typedef ELEMENTO* PONT;

typedef struct {
    PONT topo;
} PILHA;
```

# Modelagem

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

```
typedef int TIPOCHAVE;
```

```
typedef struct {
    TIPOCHAVE chave;
    // outros campos...
} REGISTRO;
```

```
typedef struct aux {
    REGISTRO reg;
    struct aux* prox;
} ELEMENTO;
```

```
typedef ELEMENTO* PONT;
```

```
typedef struct {
    PONT topo;
} PILHA;
```

# Modelagem

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

typedef int TIPOCHAVE;

typedef struct {
    TIPOCHAVE chave;
    // outros campos...
} REGISTRO;
```

```
typedef struct aux {
    REGISTRO reg;
    struct aux* prox;
} ELEMENTO;
```

```
typedef ELEMENTO* PONT;
```

```
typedef struct {
    PONT topo;
} PILHA;
```

# Modelagem

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

typedef int TIPOCHAVE;

typedef struct {
    TIPOCHAVE chave;
    // outros campos...
} REGISTRO;
```

```
typedef struct aux {
    REGISTRO reg;
    struct aux* prox;
} ELEMENTO;

typedef ELEMENTO* PONT;

typedef struct {
    PONT topo;
} PILHA;
```

# Funções de gerenciamento

Implementaremos funções para:

Inicializar a estrutura

Retornar a quantidade de elementos válidos

Exibir os elementos da estrutura

Verificar se a **pilha está vazia**

Inserir elementos na estrutura (***push***)

Excluir elementos da estrutura (***pop***)

Reinicializar a estrutura

# Inicialização

Para inicializar uma pilha já criada pelo usuário, precisamos apenas acertar o valor do campo *topo*.

# Inicialização

Para inicializar uma pilha já criada pelo usuário, precisamos apenas acertar o valor do campo **topo**.

Já que o topo conterá o endereço do elemento que está no topo da pilha e a **pilha está vazia**, iniciaremos esse campo com valor **NULL**.

# Inicialização

```
void inicializarPilha(PILHA* p) {  
    p->topo = NULL;  
}
```

# Inicialização

```
void inicializarPilha(PILHA* p) {  
    p->topo = NULL;  
}
```



**Retornar número de elementos**

# Retornar número de elementos

Já que não temos um campo com o número de elementos na pilha, precisaremos **percorrer todos os elementos** para contar quantos são.

# Retornar número de elementos

```
int tamanho(PILHA* p) {
```

```
}
```

# Retornar número de elementos

```
int tamanho(PILHA* p) {  
    PONT end = p->topo;  
    int tam = 0;  
  
}
```

# Retornar número de elementos

```
int tamanho(PILHA* p) {  
    PONT end = p->topo;  
    int tam = 0;  
    while (end != NULL) {  
        tam++;  
        end = end->prox;  
    }  
}
```

# Retornar número de elementos

```
int tamanho(PILHA* p) {  
    PONT end = p->topo;  
    int tam = 0;  
    while (end != NULL) {  
        tam++;  
        end = end->prox;  
    }  
    return tam;  
}
```

# Retornar número de elementos

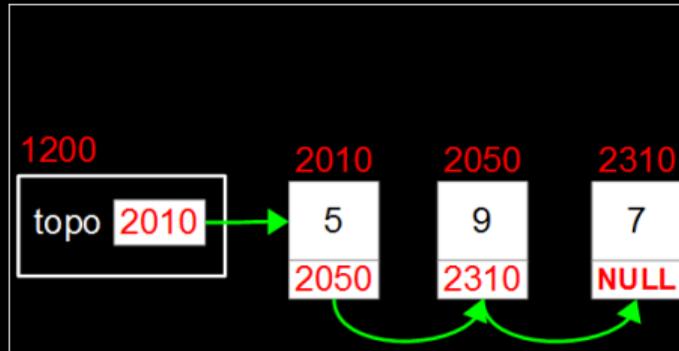
```
int tamanho(PILHA* p) {  
    PONT end = p->topo;  
    int tam = 0;  
    while (end != NULL) {  
        tam++;  
        end = end->prox;  
    }  
    return tam;  
}
```

```
int tamanho2(PILHA p) {  
    PONT end = p.topo;  
    int tam = 0;  
    while (end != NULL) {  
        tam++;  
        end = end->prox;  
    }  
    return tam;  
}
```

# Retornar número de elementos

```
int tamanho(PILHA* p) {  
    PONT end = p->topo;  
    int tam = 0;  
    while (end != NULL) {  
        tam++;  
        end = end->prox;  
    }  
    return tam;  
}
```

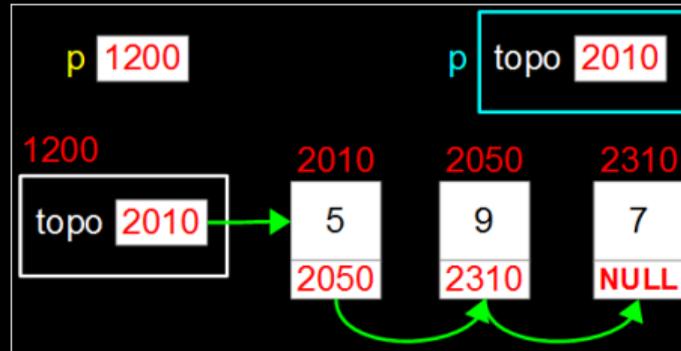
```
int tamanho2(PILHA p) {  
    PONT end = p.topo;  
    int tam = 0;  
    while (end != NULL) {  
        tam++;  
        end = end->prox;  
    }  
    return tam;  
}
```



# Retornar número de elementos

```
int tamanho(PILHA* p) {  
    PONT end = p->topo;  
    int tam = 0;  
    while (end != NULL) {  
        tam++;  
        end = end->prox;  
    }  
    return tam;  
}
```

```
int tamanho2(PILHA p) {  
    PONT end = p.topo;  
    int tam = 0;  
    while (end != NULL) {  
        tam++;  
        end = end->prox;  
    }  
    return tam;  
}
```



**Verificar se a pilha está vazia**

# Verificar se a pilha está vazia

Por que não usar a **função tamanho** para verificar se a pilha está vazia?

# Verificar se a pilha está vazia

Por que não usar a **função tamanho** para verificar se a pilha está vazia?

É bem mais simples verificar se *topo* está armazenando o endereço **NULL**.

# Verificar se a pilha está vazia

```
bool estaVazia(PILHA* p) {  
    if (p->topo == NULL) return true;  
    return false;  
}
```

# Exibição/Impressão

Para exibir os elementos da estrutura precisaremos percorrer os **elementos** (iniciando pelo elemento do topo da pilha) e, por exemplo, **imprimir suas chaves**.

# Exibição/Impressão

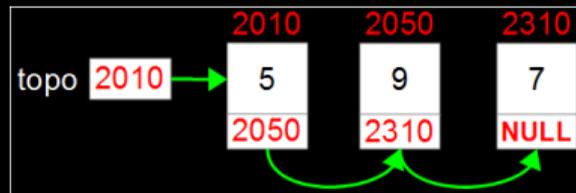
```
void exibirPilha(PILHA* p) {
    PONT end = p->topo;
    printf("Pilha: \n ");
    while (end != NULL) {
        printf("%i ", end->reg.chave);
        end = end->prox;
    }
    printf("\n\n");
}
```

# Exibição/Impressão

```
void exibirPilha(PILHA* p) {
    PONT end = p->topo;
    printf("Pilha: \n ");
    while (end != NULL) {
        printf("%i ", end->reg.chave);
        end = end->prox;
    }
    printf("\n\n");
}
```

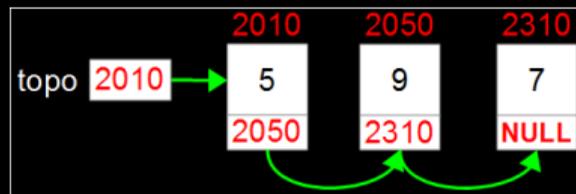
# Exibição/Impressão

```
void exibirPilha(PILHA* p) {  
    PONT end = p->topo;  
    printf("Pilha: \" \");  
    while (end != NULL) {  
        printf("%i ", end->reg.chave);  
        end = end->prox;  
    }  
    printf("\n\n");  
}
```



# Exibição/Impressão

```
void exibirPilha(PILHA* p) {  
    PONT end = p->topo;  
    printf("Pilha: \" \");  
    while (end != NULL) {  
        printf("%i ", end->reg.chave);  
        end = end->prox;  
    }  
    printf("\n\n");  
}
```



Saída:

\$ Pilha: " 5 9 7 "

# Inserção de um elemento (*push*)

O usuário passa como parâmetro um registro a ser inserido na pilha

# Inserção de um elemento (*push*)

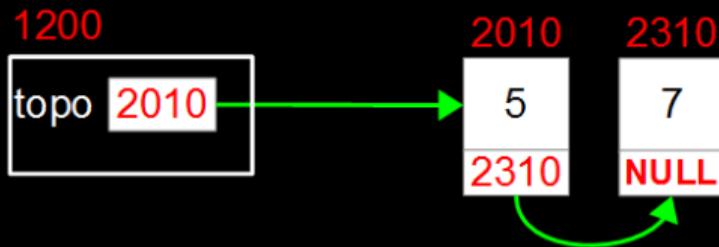
O usuário passa como parâmetro um registro a ser inserido na pilha

O elemento será inserido no topo da pilha, ou melhor, **“acima” do elemento que está no topo da pilha.**

O novo elemento irá **apontar** para o elemento que estava no topo da pilha..

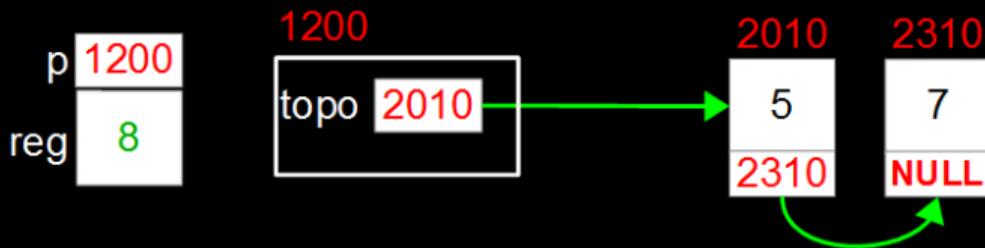
# Inserção de um elemento (*push*)

```
bool inserirElemPilha(PILHA* p, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = p->topo;  
    p->topo = novo;  
    return true;  
}
```



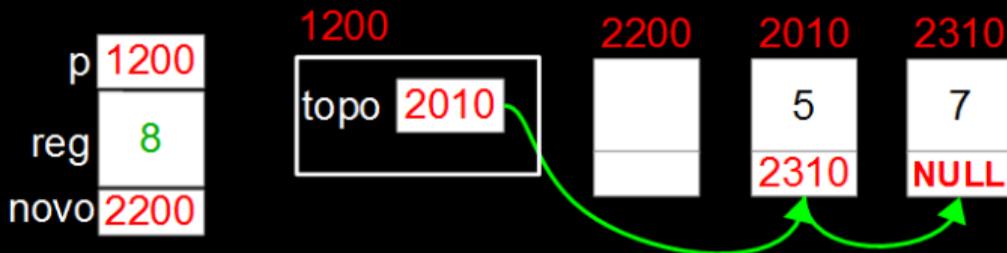
# Inserção de um elemento (*push*)

```
bool inserirElemPilha(PILHA* p, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = p->topo;  
    p->topo = novo;  
    return true;  
}
```



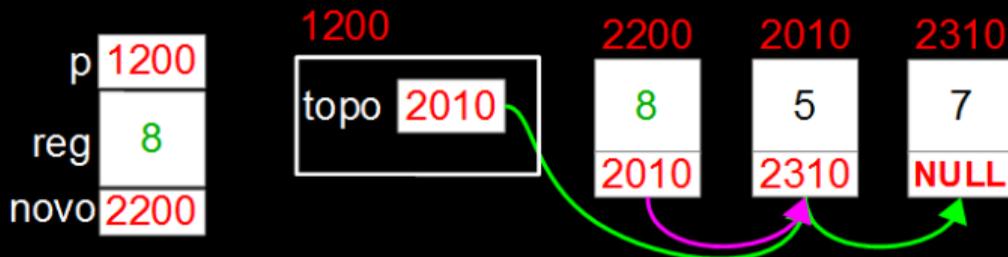
# Inserção de um elemento (*push*)

```
bool inserirElemPilha(PILHA* p, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = p->topo;  
    p->topo = novo;  
    return true;  
}
```



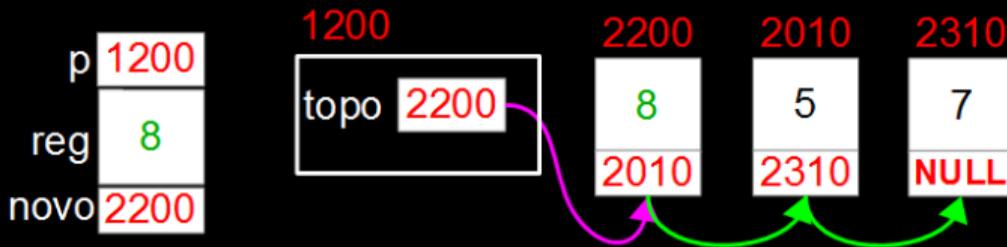
# Inserção de um elemento (*push*)

```
bool inserirElemPilha(PILHA* p, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = p->topo;  
    p->topo = novo;  
    return true;  
}
```



# Inserção de um elemento (*push*)

```
bool inserirElemPilha(PILHA* p, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = p->topo;  
    p->topo = novo;  
    return true;  
}
```



# Exclusão de um elemento (*pop*)

O usuário solicita a exclusão do elemento do **topo**  
da **pilha**:

# Exclusão de um elemento (*pop*)

O usuário solicita a exclusão do elemento do **topo da pilha**:

Se a pilha **não estiver vazia**, além de excluir esse elemento da pilha iremos **copiá-lo para um local indicado pelo usuário**.

# Exclusão de um elemento (*pop*)

```
bool excluirElemPilha(PILHA* p, REGISTRO* reg) {
```

```
}
```

# Exclusão de um elemento (*pop*)

```
bool excluirElemPilha(PILHA* p, REGISTRO* reg) {  
    if ( p->topo == NULL) return false;
```

```
}
```

# Exclusão de um elemento (*pop*)

```
bool excluirElemPilha(PILHA* p, REGISTRO* reg) {  
    if ( p->topo == NULL) return false;  
    *reg = p->topo->reg;  
  
}
```

# Exclusão de um elemento (*pop*)

```
bool excluirElemPilha(PILHA* p, REGISTRO* reg) {  
    if ( p->topo == NULL) return false;  
    *reg = p->topo->reg;  
    PONT apagar = p->topo;  
  
}
```

# Exclusão de um elemento (*pop*)

```
bool excluirElemPilha(PILHA* p, REGISTRO* reg) {  
    if ( p->topo == NULL) return false;  
    *reg = p->topo->reg;  
    PONT apagar = p->topo;  
    p->topo = p->topo->prox;  
  
}
```

# Exclusão de um elemento (*pop*)

```
bool excluirElemPilha(PILHA* p, REGISTRO* reg) {  
    if ( p->topo == NULL) return false;  
    *reg = p->topo->reg;  
    PONT apagar = p->topo;  
    p->topo = p->topo->prox;  
    free(apagar);  
  
}
```

# Exclusão de um elemento (*pop*)

```
bool excluirElemPilha(PILHA* p, REGISTRO* reg) {
    if ( p->topo == NULL) return false;
    *reg = p->topo->reg;
    PONT apagar = p->topo;
    p->topo = p->topo->prox;
    free(apagar);
    return true;
}
```

# Reinicialização da pilha

# Reinicialização da pilha

Para reinicializar a pilha, precisamos **excluir** todos os seus elementos e colocar **NULL** no campo *topo*

# Reinicialização da pilha

```
void reinicializarPilha(PILHA* p) {  
    PONT apagar;  
    PONT posicao = p->topo;  
  
}
```

# Reinicialização da pilha

```
void reinicializarPilha(PILHA* p) {  
    PONT apagar;  
    PONT posicao = p->topo;  
    while (posicao != NULL) {  
        apagar = posicao;  
        posicao = posicao->prox;  
        free(apagar);  
    }  
}
```

# Reinicialização da pilha

```
void reinicializarPilha(PILHA* p) {  
    PONT apagar;  
    PONT posicao = p->topo;  
    while (posicao != NULL) {  
        apagar = posicao;  
        posicao = posicao->prox;  
        free(apagar);  
    }  
    p->topo = NULL;  
}
```

# **AULA 09**

# **ESTRUTURA DE DADOS**

---

**Pilha - implementação dinâmica**

**Norton T. Roman & Luciano A. Digiampietri**