

AULA 9

ESTRUTURA DE DADOS

Deque

Deque

É uma estrutura de dados na qual os elementos podem ser **inseridos ou excluídos** de qualquer uma de suas **extremidades** (do início ou do fim).

Deque

É uma estrutura de dados na qual os elementos podem ser **inseridos ou excluídos** de qualquer uma de suas **extremidades** (do início ou do fim).

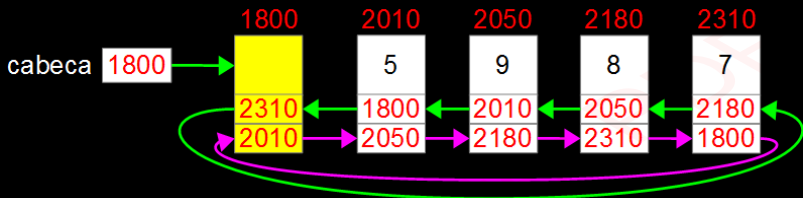
- Utilizaremos uma implementação **duplamente ligada** (ou duplamente encadeada), na qual cada elemento possui o endereço de seu antecessor e de seu sucessor.

Deque

É uma estrutura de dados na qual os elementos podem ser **inseridos ou excluídos** de qualquer uma de suas **extremidades** (do início ou do fim).

- Utilizaremos uma implementação **duplamente ligada** (ou duplamente encadeada), na qual cada elemento possui o endereço de seu antecessor e de seu sucessor.
- Utilizaremos um **nó cabeça** para facilitar o gerenciamento da estrutura.

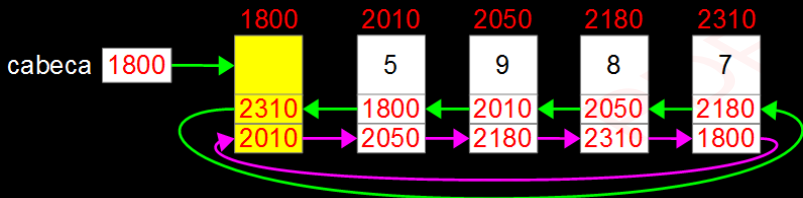
Deque



Temos um ponteiro para o **nó cabeça**

Cada elemento indica **seu antecessor e seu sucessor** (o último tem o nó cabeça como sucessor e o nó cabeça tem o último como antecessor).

Deque

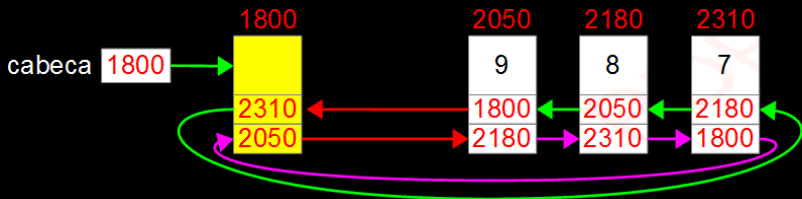


Temos um ponteiro para o **nó cabeça**

Cada elemento indica **seu antecessor e seu sucessor** (o último tem o nó cabeça como sucessor e o nó cabeça tem o último como antecessor).

Como excluimos um elemento do início?

Deque

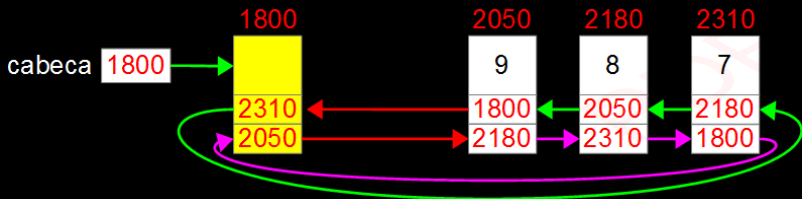


Temos um ponteiro para o **nó cabeça**

Cada elemento indica **seu antecessor e seu sucessor** (o último tem o nó cabeça como sucessor e o nó cabeça tem o último como antecessor).

Como excluimos um elemento do início?

Deque



Temos um ponteiro para o **nó cabeça**

Cada elemento indica **seu antecessor e seu sucessor** (o último tem o nó cabeça como sucessor e o nó cabeça tem o último como antecessor).

Como excluimos um elemento do início?

Como inserimos o elemento 1 no fim?

Deque



Temos um ponteiro para o **nó cabeça**

Cada elemento indica **seu antecessor e seu sucessor** (o último tem o nó cabeça como sucessor e o nó cabeça tem o último como antecessor).

Como excluimos um elemento do início?

Como inserimos o elemento 1 no fim?

Modelagem

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

```
typedef int TIPOCHAVE;
```

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

```
typedef struct auxElem {
    REGISTRO reg;
    struct auxElem* ant;
    struct auxElem* prox;
} ELEMENTO;
```

```
typedef ELEMENTO* PONT;
```

```
typedef struct {
    PONT cabeca;
} DEQUE;
```

Modelagem

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

```
typedef int TIPOCHAVE;
```

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

```
typedef struct auxElem {
    REGISTRO reg;
    struct auxElem* ant;
    struct auxElem* prox;
} ELEMENTO;
```

```
typedef ELEMENTO* PONT;
```

```
typedef struct {
    PONT cabeca;
} DEQUE;
```

Modelagem

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

```
typedef int TIPOCHAVE;
```

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

```
typedef struct auxElem {
    REGISTRO reg;
    struct auxElem* ant;
    struct auxElem* prox;
} ELEMENTO;
```

```
typedef ELEMENTO* PONT;
```

```
typedef struct {
    PONT cabeca;
} DEQUE;
```

Funções de gerenciamento

Implementaremos funções para:

- Inicializar a estrutura

- Retornar a quantidade de elementos válidos

- Exibir os elementos da estrutura

- Inserir elementos na estrutura (duas funções)

- Excluir elementos da estrutura (duas funções)

- Reinicializar a estrutura

Inicialização

Para inicializarmos nosso deque, nós **precisamos**:

Inicialização

Para inicializarmos nosso deque, nós **precisamos**:

- **Criar o nó cabeça;**

Inicialização

Para inicializarmos nosso deque, nós **precisamos**:

- **Criar o nó cabeça**;
- A variável *cabeça* precisa apontar para ele;

Inicialização

Para inicializarmos nosso deque, nós **precisamos**:

- **Criar o nó cabeça**;
- A variável *cabeça* precisa apontar para ele;
- E o nó cabeça apontará para ele mesmo como **anterior e próximo**.

Inicialização

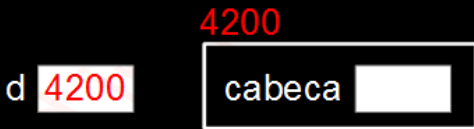
```
void inicializarDeque(DEQUE* d) {  
    d->cabeca = (PONT) malloc(sizeof(ELEMENTO));  
    d->cabeca->prox = d->cabeca;  
    d->cabeca->ant = d->cabeca;  
}
```

4200



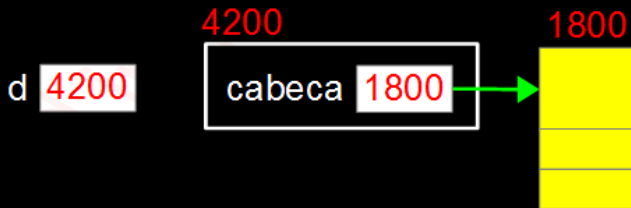
Inicialização

```
void inicializarDeque(DEQUE* d) {  
    d->cabeca = (PONT) malloc(sizeof(ELEMENTO));  
    d->cabeca->prox = d->cabeca;  
    d->cabeca->ant = d->cabeca;  
}
```



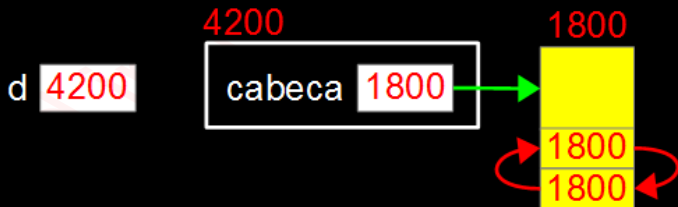
Inicialização

```
void inicializarDeque(DEQUE* d) {  
    d->cabeca = (PONT) malloc(sizeof(ELEMENTO));  
    d->cabeca->prox = d->cabeca;  
    d->cabeca->ant = d->cabeca;  
}
```



Inicialização

```
void inicializarDeque(DEQUE* d) {  
    d->cabeca = (PONT) malloc(sizeof(ELEMENTO));  
    d->cabeca->prox = d->cabeca;  
    d->cabeca->ant = d->cabeca;  
}
```



Retornar número de elementos

Precisaremos **percorrer todos os elementos** para contar quantos são.

Retornar número de elementos

```
int tamanho(DEQUE* d) {
```

```
}
```

Retornar número de elementos

```
int tamanho(DEQUE* d) {
    PONT end = d->cabeca->prox;

```


Retornar número de elementos

```
int tamanho(DEQUE* d) {  
    PONT end = d->cabeca->prox;  
    int tam = 0;  
  
}
```

Retornar número de elementos

```
int tamanho(DEQUE* d) {  
    PONT end = d->cabeca->prox;  
    int tam = 0;  
    while (end != d->cabeca) {  
        tam++;  
        end = end->prox;  
    }  
  
}
```

Retornar número de elementos

```
int tamanho(DEQUE* d) {  
    PONT end = d->cabeca->prox;  
    int tam = 0;  
    while (end != d->cabeca) {  
        tam++;  
        end = end->prox;  
    }  
    return tam;  
}
```

Retornar número de elementos

```
int tamanho(DEQUE* d) {  
    PONT end = d->cabeca->prox;  
    int tam = 0;  
    while (end != d->cabeca) {  
        tam++;  
        end = end->prox;  
    }  
    return tam;  
}
```

```
int tamanho2(DEQUE* d) {  
    PONT end = d->cabeca->ant;  
    int tam = 0;  
    while (end != d->cabeca) {  
        tam++;  
        end = end->ant;  
    }  
    return tam;  
}
```

Exibição/Impressão

Para exibir os elementos da estrutura precisaremos percorrer os **elementos** válidos e, por exemplo, **imprimir suas chaves**.

Precisamos lembrar que o nó cabeça não é um dos elementos **válidos** do nosso deque.

Podemos percorrer do início para o fim ou do fim para o início.

Exibição/Impressão

```
void exibirDequeFim(DEQUE* d) {
```

```
}
```

Exibição/Impressão

```
void exibirDequeFim(DEQUE* d) {
    PONT end = d->cabeca->ant;
```

}

Exibição/Impressão

```
void exibirDequeFim(DEQUE* d) {  
    PONT end = d->cabeca->ant;  
    printf("Deque partindo do fim: \" \");  
  
}
```


Exibição/Impressão

```
void exibirDequeFim(DEQUE* d) {  
    PONT end = d->cabeca->ant;  
    printf("Deque partindo do fim: \" \");  
    while (end != d->cabeca) {  
        printf("%i ", end->reg.chave);  
        end = end->ant;  
    }  
  
}
```

Exibição/Impressão

```
void exibirDequeFim(DEQUE* d) {  
    PONT end = d->cabeca->ant;  
    printf("Deque partindo do fim: \" \");  
    while (end != d->cabeca) {  
        printf("%i ", end->reg.chave);  
        end = end->ant;  
    }  
    printf("\n");  
}
```

Inserção de um elemento

O usuário escolhe a função que insere no início ou no fim e passa como parâmetro o registro a ser inserido

Inserção de um elemento

O usuário escolhe a função que insere no início ou no fim e passa como parâmetro o registro a ser inserido

A função **aloca memória** para o novo elemento;

Inserção de um elemento

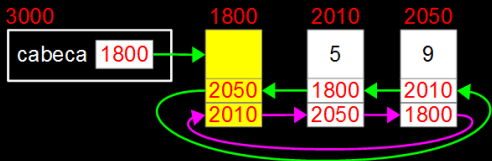
O usuário escolhe a função que insere no início ou no fim e passa como parâmetro o registro a ser inserido

A função **aloca memória** para o novo elemento;

Acerta quatro endereços/ponteiros: os dois do elemento novo e o campo *ant* do elemento que será o posterior do novo e *pos* do elemento que será o anterior do novo.

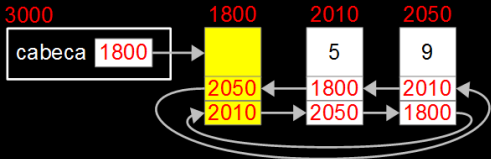
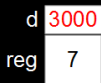
Inserção em deque

```
bool inserirDequeFim(DEQUE* d, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = d->cabeca;  
    novo->ant = d->cabeca->ant;  
    d->cabeca->ant = novo;  
    novo->ant->prox = novo;  
    return true;  
}
```



Inserção em deque

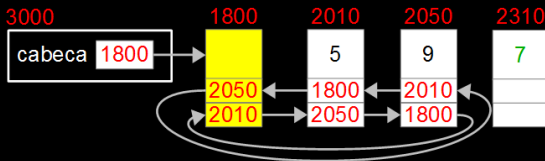
```
bool inserirDequeFim(DEQUE* d, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = d->cabeca;  
    novo->ant = d->cabeca->ant;  
    d->cabeca->ant = novo;  
    novo->ant->prox = novo;  
    return true;  
}
```



Inserção em deque

```
bool inserirDequeFim(DEQUE* d, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = d->cabeca;  
    novo->ant = d->cabeca->ant;  
    d->cabeca->ant = novo;  
    novo->ant->prox = novo;  
    return true;  
}
```

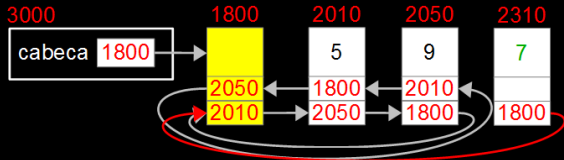
d	3000
reg	7
novo	2310



Inserção em deque

```
bool inserirDequeFim(DEQUE* d, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = d->cabeca;  
    novo->ant = d->cabeca->ant;  
    d->cabeca->ant = novo;  
    novo->ant->prox = novo;  
    return true;  
}
```

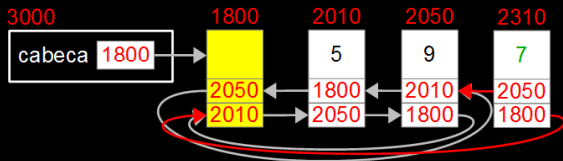
d	3000
reg	7
novo	2310



Inserção em deque

```
bool inserirDequeFim(DEQUE* d, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = d->cabeca;  
    novo->ant = d->cabeca->ant;  
    d->cabeca->ant = novo;  
    novo->ant->prox = novo;  
    return true;  
}
```

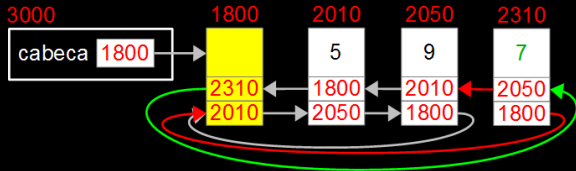
d	3000
reg	7
novo	2310



Inserção em deque

```
bool inserirDequeFim(DEQUE* d, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = d->cabeca;  
    novo->ant = d->cabeca->ant;  
    d->cabeca->ant = novo;  
    novo->ant->prox = novo;  
    return true;  
}
```

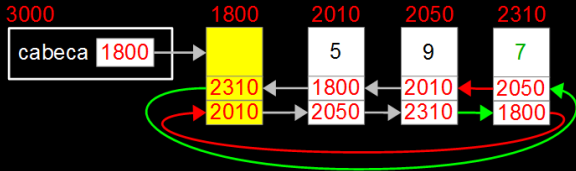
d	3000
reg	7
novo	2310



Inserção em deque

```
bool inserirDequeFim(DEQUE* d, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = d->cabeca;  
    novo->ant = d->cabeca->ant;  
    d->cabeca->ant = novo;  
    novo->ant->prox = novo;  
    return true;  
}
```

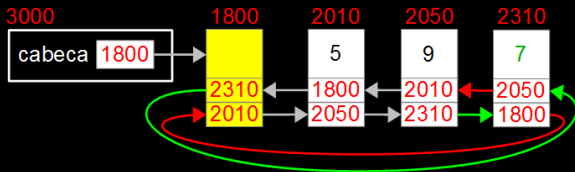
d	3000
reg	7
novo	2310



Inserção em deque

```
bool inserirDequeFim(DEQUE* d, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = d->cabeca;  
    novo->ant = d->cabeca->ant;  
    d->cabeca->ant = novo;  
    novo->ant->prox = novo;  
    return true;  
}
```

d	3000
reg	7
novo	2310



Exclusão de um elemento

O usuário passa a chave do elemento que ele quer excluir

Se houver um elemento com esta chave no deque, **exclui este elemento** do deque, **acerta os ponteiros** envolvidos e retorna *true*.
Caso contrário, retorna *false*

Exclusão de um elemento

O usuário passa a chave do elemento que ele quer excluir

Se houver um elemento com esta chave no deque, **exclui este elemento** do deque, **acerta os ponteiros** envolvidos e retorna *true*.

Caso contrário, retorna *false*

Para esta função precisamos saber quem é o **predecessor** do elemento a ser excluído.

Exclusão de um elemento

```
bool excluirElemDequeInicio(DEQUE* d, REGISTRO* reg) {
```

}

Exclusão de um elemento

```
bool excluirElemDequeInicio(DEQUE* d, REGISTRO* reg) {  
    if (d->cabeca->prox == d->cabeca) return false;  
  
    }  
}
```

Exclusão de um elemento

```
bool excluirElemDequeInicio(DEQUE* d, REGISTRO* reg) {  
    if (d->cabeca->prox == d->cabeca) return false;  
    PONT apagar = d->cabeca->prox;  
  
}
```

Exclusão de um elemento

```
bool excluirElemDequeInicio(DEQUE* d, REGISTRO* reg) {  
    if (d->cabeca->prox == d->cabeca) return false;  
    PONT apagar = d->cabeca->prox;  
    *reg = apagar->reg;  
  
}
```

Exclusão de um elemento

```
bool excluirElemDequeInicio(DEQUE* d, REGISTRO* reg) {  
    if (d->cabeca->prox == d->cabeca) return false;  
    PONT apagar = d->cabeca->prox;  
    *reg = apagar->reg;  
    d->cabeca->prox = apagar->prox;  
  
}
```

Exclusão de um elemento

```
bool excluirElemDequeInicio(DEQUE* d, REGISTRO* reg) {  
    if (d->cabeca->prox == d->cabeca) return false;  
    PONT apagar = d->cabeca->prox;  
    *reg = apagar->reg;  
    d->cabeca->prox = apagar->prox;  
    apagar->prox->ant = d->cabeca;  
  
}
```

Exclusão de um elemento

```
bool excluirElemDequeInicio(DEQUE* d, REGISTRO* reg) {  
    if (d->cabeca->prox == d->cabeca) return false;  
    PONT apagar = d->cabeca->prox;  
    *reg = apagar->reg;  
    d->cabeca->prox = apagar->prox;  
    apagar->prox->ant = d->cabeca;  
    free(apagar);  
  
}
```

Exclusão de um elemento

```
bool excluirElemDequeInicio(DEQUE* d, REGISTRO* reg) {  
    if (d->cabeca->prox == d->cabeca) return false;  
    PONT apagar = d->cabeca->prox;  
    *reg = apagar->reg;  
    d->cabeca->prox = apagar->prox;  
    apagar->prox->ant = d->cabeca;  
    free(apagar);  
    return true;  
}
```

Reinicialização de deque

Reinicialização de deque

Para reinicializar a estrutura, precisamos **excluir todos os elementos válidos** e atualizar os campos *prox* e *prox* do nó cabeça.

Reinicialização do deque

```
void reinicializarDeque(DEQUE* d) {
```

```
}
```

Reinicialização do deque

```
void reinicializarDeque(DEQUE* d) {
    PONT end = d->cabeca->prox;
```

}

Reinicialização do deque

```
void reinicializarDeque(DEQUE* d) {  
    PONT end = d->cabeca->prox;  
    while (end != d->cabeca) {  
        PONT apagar = end;  
        end = end->prox;  
        free(apagar);  
    }  
  
}
```

Reinicialização do deque

```
void reinicializarDeque(DEQUE* d) {  
    PONT end = d->cabeca->prox;  
    while (end != d->cabeca) {  
        PONT apagar = end;  
        end = end->prox;  
        free(apagar);  
    }  
    d->cabeca->prox = d->cabeca;  
}
```

Reinicialização do deque

```
void reinicializarDeque(DEQUE* d) {  
    PONT end = d->cabeca->prox;  
    while (end != d->cabeca) {  
        PONT apagar = end;  
        end = end->prox;  
        free(apagar);  
    }  
    d->cabeca->prox = d->cabeca;  
    d->cabeca->ant = d->cabeca;  
}
```

AULA 9

ESTRUTURA DE DADOS

Deque