

SIN5013

Análise de Algoritmos e Estruturas de Dados

Estruturas Lineares - Listas, Pilhas e Filas

Prof. Luciano Antonio Digiampietri

Lista linear

Estrutura de dados na qual **cada elemento é precedido por um elemento e sucedido por outro** (exceto o primeiro que não tem predecessor e o último que não tem sucessor).

Os elementos estão em uma dada **ordem** (por exemplo, a ordem de inclusão ou ordenados por uma chave).

Estruturas discutidas nesta aula (modelagem e funções principais)

1. Lista sequencial ordenada
2. Lista ligada (implementação dinâmica)
3. Lista ligada (circular e com nó-cabeça)
4. Pilha (implementação dinâmica)
5. Fila (implementação estática)
6. Fila (implementação dinâmica)

Funções típicas de gerenciamento

Inicializar a estrutura

Retornar a quantidade de elementos válidos

Exibir os elementos da estrutura

Buscar por um elemento na estrutura

Inserir elementos na estrutura

Excluir elementos da estrutura

Reinicializar a estrutura

Discutiremos as seguintes funções

Inicializar a estrutura

Retornar a quantidade de elementos válidos

Exibir os elementos da estrutura

Buscar por um elemento na estrutura

Inserir elementos na estrutura

Excluir elementos da estrutura

Reinicializar a estrutura

Lista linear sequencial

É uma **lista linear** na qual a **ordem lógica** dos elementos (a ordem “vista” pelo usuário) **é a mesma ordem física** (em memória principal) dos elementos. Isto é, elementos vizinhos na lista estarão em posições vizinhas de memória.

1. Lista sequencial ordenada

É uma **lista linear sequencial** na qual os elementos são **mantidos ordeados** segundo um critério de ordenação (valor de uma chave, por exemplo). As funções de inserção e exclusão devem garantir a ordenação.

Lista linear sequencial

Modelagem:

Modelaremos usando um **arranjo** de registros;

Registros conterão as informações de interesse do usuário;

Nosso arranjo terá um **tamanho fixo** e controlaremos o número de elementos com uma **variável adicional**.

Modelagem

```
#include <stdio.h>
#define MAX 50
#define ERRO -1
#define true 1
#define false 0

typedef int bool;
typedef int TIPOCHAVE;
```

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

```
typedef struct {
    REGISTRO A[MAX];
    int nroElem;
} LISTA;
```

Busca por elementos

O usuário diz qual elemento é buscado e a função retorna a posição desse elemento:

- Assumiremos nas primeiras implementações que as chaves dos elementos **não** estão em ordem crescente;
- Se o elemento não existir a função retorna -1;

Busca por elementos (busca sequência simples)

```
int buscaSequencial(LISTA* l, TIPOCHAVE ch) {  
    int i = 0;  
    while (i < l->nroElem) {  
        if(ch == l->A[i].chave) return i;  
        else i++;  
    }  
    return -1;  
}
```

Busca por elementos

Ideia: Ao invés de fazer duas comparações por iteração, seria possível fazer só uma?

- Precisamos sempre **comparar a chave do elemento atual** com a chave do elemento buscado;
- Mas como garantir que não iremos **passar do último elemento?**
- Garantindo que **a chave buscada será encontrada!**

Busca por elementos

Criação de um elemento **sentinela**:

- **Elemento extra** (um registro) adicionado à lista para auxiliar alguma operação;
- Será **inserido no final da lista** (após o último elemento válido) durante as buscas;
- Conterá a **chave do elemento buscado**.

Busca por elementos (sentinela)

```
int buscaSentinela(LISTA* l, TIPOCHAVE ch) {  
    int i = 0;  
    l->A[l->nroElem].chave = ch;  
    while(l->A[i].chave != ch) i++;  
    if (i == l->nroElem) return -1;  
    else return i;  
}
```

Busca por elementos (sentinela)

Há apenas um **probleminha**:

- Se a lista já estiver **cheia**, não haverá espaço para criar o sentinela;
- **O que fazer?**
- Criamos a lista com uma posição extra (**um registro a mais**) para garantir que haverá espaço para o sentinela.
- Essa posição extra **nunca terá um registro válido**.

Modelagem

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

```
#define MAX 50
```

```
#define ERRO -1
```

```
#define true 1
```

```
#define false 0
```

```
typedef int bool;
```

```
typedef int TIPOCHAVE;
```

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

```
typedef struct {  
    REGISTRO A[MAX+1];  
    int nroElem;  
} LISTA;
```

Busca por elementos (sentinela)

```
int buscaSequencial(LISTA* l,
                    TIPOCHAVE ch) {
    int i = 0;
    while (i < l->nroElem) {
        if(ch == l->A[i].chave) return i;
        else i++;
    }
    return -1;
}
```

```
int buscaSentinela(LISTA* l,
                   TIPOCHAVE ch) {
    int i = 0;
    l->A[l->nroElem].chave = ch;
    while(l->A[i].chave != ch) i++;
    if (i == l->nroElem) return -1;
    else return i;
}
```

Busca por elementos

Mas a busca binária não é mais eficiente?

- Sim, porém ela necessita que as chaves dos elementos estejam ordenadas;
- Para isso, precisaremos mudar nossa função de inserção de elementos.
- A função de inserção seguirá a lógica do *insertion sort*.

Inserção de elementos - ordenada

```
bool inserirElemListaOrd(LISTA* l, REGISTRO reg) {  
    if(l->nroElem >= MAX) return false;  
    int pos = l->nroElem;  
    while(pos > 0 && l->A[pos-1].chave > reg.chave) {  
        l->A[pos] = l->A[pos-1];  
        pos--;  
    }  
    l->A[pos] = reg;  
    l->nroElem++;  
    return true;  
}
```

2010	A	5	9	44	?	?
	nroElem	3				

Inserção de elementos - ordenada

```
bool inserirElemListaOrd(LISTA* l, REGISTRO reg) {  
    if(l->nroElem >= MAX) return false;  
    int pos = l->nroElem;  
    while(pos > 0 && l->A[pos-1].chave > reg.chave) {  
        l->A[pos] = l->A[pos-1];  
        pos--;  
    }  
    l->A[pos] = reg;  
    l->nroElem++;  
    return true;  
}
```

2010	A	5	9	44	?	?
	nroElem	3				
l	2010	pos 3				
reg	33					

Inserção de elementos - ordenada

```
bool inserirElemListaOrd(LISTA* l, REGISTRO reg) {  
    if(l->nroElem >= MAX) return false;  
    int pos = l->nroElem;  
    while(pos > 0 && l->A[pos-1].chave > reg.chave) {  
        l->A[pos] = l->A[pos-1];  
        pos--;  
    }  
    l->A[pos] = reg;  
    l->nroElem++;  
    return true;  
}
```

2010	A						?	
	5	9	44	44				
nroElem		3						
l	2010	pos		2				
reg	33							

Inserção de elementos - ordenada

```
bool inserirElemListaOrd(LISTA* l, REGISTRO reg) {  
    if(l->nroElem >= MAX) return false;  
    int pos = l->nroElem;  
    while(pos > 0 && l->A[pos-1].chave > reg.chave) {  
        l->A[pos] = l->A[pos-1];  
        pos--;  
    }  
    l->A[pos] = reg;  
    l->nroElem++;  
    return true;  
}
```

2010	A					
	5	9	33	44	?	
nroElem		4				
l	2010	pos		2		
reg	33					

Busca binária

```
int buscaBinaria(LISTA* l, TIPOCHAVE ch) {
    int esq, dir, meio;
    esq = 0;
    dir = l->nroElem-1;
    while(esq <= dir) {
        meio = ((esq + dir) / 2);
        if(l->A[meio].chave == ch) return meio;
        else {
            if(l->A[meio].chave < ch) esq = meio + 1;
            else dir = meio - 1;
        }
    }
    return -1;
}
```

Elementos ordenados pelas chaves

Com a ordenação dos elementos pela chave:

- A busca ficou mais eficiente (busca binária);
- Não precisamos do sentinela;
- O que acontece com a exclusão?

Exclusão de elementos

```
bool excluirElemLista(LISTA* l, TIPOCHAVE ch) {  
    int pos, j;  
    pos = buscaBinaria(l,ch);  
    if(pos == -1) return false;  
    for(j=pos; j < l->nroElem-1; j++) l->A[j] = l->A[j+1];  
    l->nroElem--;  
    return true;  
}
```

Lista linear sequencial

Realizamos a **inserção ordenada pela chave**:

- A **busca** é feita de maneira **eficiente** (busca binária);
- Porém **a inserção e a exclusão** são **custosas**, pois potencialmente precisam deslocar vários elementos.

2. Lista ligada

Lista ligada

Para evitar o deslocamento de elementos durante a inserção e a exclusão utilizaremos uma **lista ligada**:

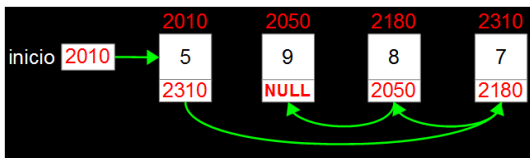
- É uma **estrutura linear** (cada elemento possui no máximo um predecessor e um sucessor);
- A **ordem lógica** dos elementos (a ordem “vista” pelo usuário) **não é a mesma ordem física** (em memória principal) dos elementos.
- Cada elemento precisa **indicar** quem é **o seu sucessor**.

Lista ligada

Há duas implementações bases principais: *estática* (usando um arranjo) e *dinâmica* (com alocamento dinâmico [sob demanda] da memória)].

- Estudaremos a *implementação dinâmica*, isto é, alocaremos e desalocaremos a memória para os elementos *sob demanda*.
- Vantagens: não precisamos *gastar memória* que não estamos usando e não precisamos gerenciar uma lista de *elementos disponíveis*.

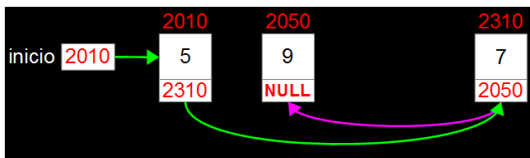
Lista ligada (ideia geral)



Temos um ponteiro para o primeiro elemento

Cada elemento indica seu sucessor

Lista ligada (ideia geral)

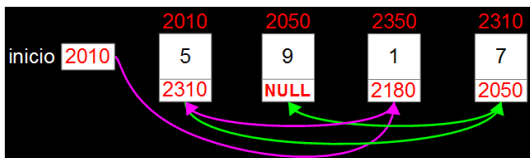


Temos um ponteiro para o primeiro elemento

Cada elemento indica seu sucessor

Como excluimos o elemento 8?

Lista ligada (ideia geral)



Temos um ponteiro para o primeiro elemento

Cada elemento indica seu sucessor

Como excluimos o elemento 8?

Como inserimos o elemento 1?

Modelagem

```
#include <stdio.h>
#include <malloc.h>
#define true 1
#define false 0

typedef int bool;
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 inicio;
} LISTA;
```

Funções de gerenciamento

Implementaremos funções para:

- Inicializar a estrutura

- Retornar a quantidade de elementos válidos

- Exibir os elementos da estrutura

- Buscar por um elemento na estrutura**

- Inserir elementos na estrutura**

- Excluir elementos da estrutura**

- Reinicializar a estrutura

Buscar por elemento

A função de busca deverá:

- Receber uma chave do usuário

- Retornar o endereço em que este elemento se encontra (caso seja encontrado)

- Retornar *NULL* caso não haja um registro com essa chave na lista

Busca sequencial

```
PONT buscaSequencial(LISTA* l, TIPOCHAVE ch) {
    PONT pos = l->inicio;
    while (pos != NULL) {
        if (pos->reg.chave == ch) return pos;
        pos = pos->prox;
    }
    return NULL;
}

// lista ordenada pelos valores das chaves dos registros
PONT buscaSeqOrd(LISTA* l, TIPOCHAVE ch) {
    PONT pos = l->inicio;
    while (pos != NULL && pos->reg.chave < ch) pos = pos->prox;
    if (pos != NULL && pos->reg.chave == ch) return pos;
    return NULL;
}
```

Inserção de um elemento

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

Realizaremos a inserção **ordenada pelo valor da chave** do registro passado e não permitiremos a inserção de **elementos repetidos**; Na inserção precisamos identificar **entre quais elementos** o novo elemento será inserido;

Alocaremos memória para o novo elemento.

Precisamos saber quem será o **predecessor do elemento**.

Inserção ordenada

Desenvolveremos uma **função auxiliar** para procurar por uma dada chave e nos informar:

- O **endereço desse elemento** se ele existir;
 - O **endereço** de quem seria o **predecessor** desse elemento (independentemente do elemento existir ou não na lista).
- Como a função irá nos passar **dois endereços** diferentes?

Busca - auxiliar

```
PONT buscaSequencialExc(LISTA* l, TIPOCHAVE ch, PONT* ant){
    *ant = NULL;
    PONT atual = l->inicio;
    while ((atual != NULL) && (atual->reg.chave<ch)) {
        *ant = atual;
        atual = atual->prox;
    }
    if ((atual != NULL) && (atual->reg.chave == ch)) return atual;
    return NULL;
}
```

Inserção ordenada

```
bool inserirElemListaOrd(LISTA* l, REGISTRO reg) {
    TIPOCHAVE ch = reg.chave;
    PONT ant, i;
    i = buscaSequencialExc(l, ch, &ant);
    if (i != NULL) return false;
    i = (PONT) malloc(sizeof(ELEMENTO));
    i->reg = reg;
    if (ant == NULL) {
        i->prox = l->inicio;
        l->inicio = i;
    } else {
        i->prox = ant->prox;
        ant->prox = i;
    }
    return true;
}
```

Exclusão de um elemento

- O usuário passa a chave do elemento que ele quer excluir
- Se houver um elemento com esta chave na lista, **exclui este elemento** da lista, **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 excluirElemLista(LISTA* l, TIPOCHAVE ch) {  
    PONT ant, i;  
    i = buscaSequencialExc(l, ch, &ant);  
    if (i == NULL) return false;  
    if (ant == NULL) l->inicio = i->prox;  
    else ant->prox = i->prox;  
    free(i);  
    return true;  
}
```

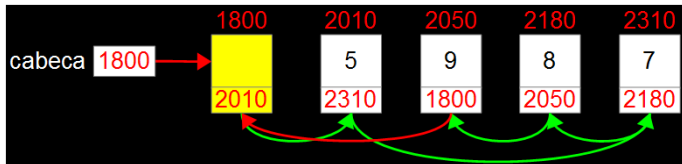
Lista ligada

Aprendemos como modelar e gerenciar **listas ligadas** utilizando o que chamamos de **implementação dinâmica**.

- Vantagens: não precisamos **gastar memória** que não estamos usando e não precisamos gerenciar uma lista de **elementos disponíveis**.
- **Agora** adicionaremos duas características a esta estrutura: ela será **circular** (o último elemento apontará para o primeiro) e possuirá um **nó cabeça** (um elemento inicial que sempre encabeçará a lista).

3. Lista ligada circular com nó cabeça

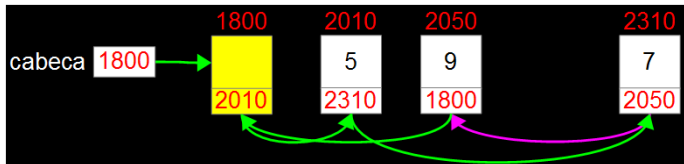
Lista ligada circular com nó cabeça



Temos um ponteiro para o **nó cabeça**

Cada elemento indica seu sucessor e o **último aponta para o cabeça**

Lista ligada circular com nó cabeça

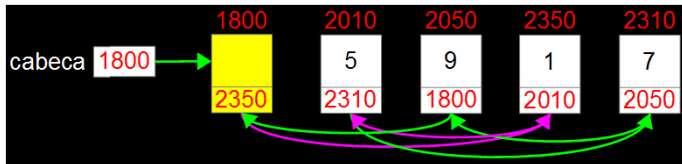


Temos um ponteiro para o **nó cabeça**

Cada elemento indica seu sucessor e o **último aponta para o cabeça**

Como excluimos o elemento 8?

Lista ligada circular com nó cabeça



Temos um ponteiro para o **nó cabeça**

Cada elemento indica seu sucessor e o **último aponta para o cabeça**

Como excluimos o elemento 8?

Como inserimos o elemento 1?

Modelagem

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

typedef int bool;
typedef int TIPOCHAVE;

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

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

typedef ELEMENTO* PONT;

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

Funções de gerenciamento

Implementaremos funções para:

- Inicializar a estrutura**

- Retornar a quantidade de elementos válidos

- Exibir os elementos da estrutura

- Buscar por um elemento na estrutura**

- Inserir elementos na estrutura**

- Excluir elementos da estrutura**

- Reinicializar a estrutura

Inicialização

Para inicializarmos nossa lista ligada circular e com nó cabeça, precisamos:

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

Inicialização

```
void inicializarLista(LISTA* l) {  
    l->cabeca = (PONT) malloc(sizeof(ELEMENTO));  
    l->cabeca->prox = l->cabeca;  
}
```

4200

cabeca

?

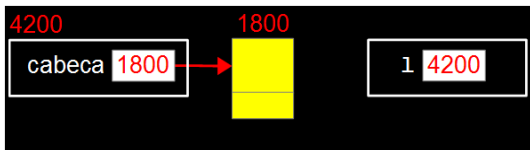
Inicialização

```
void inicializarLista(LISTA* l) {  
    l->cabeca = (PONT) malloc(sizeof(ELEMENTO));  
    l->cabeca->prox = l->cabeca;  
}
```



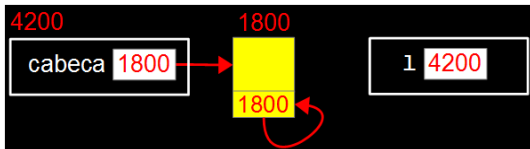
Inicialização

```
void inicializarLista(LISTA* l) {  
    l->cabeca = (PONT) malloc(sizeof(ELEMENTO));  
    l->cabeca->prox = l->cabeca;  
}
```



Inicialização

```
void inicializarLista(LISTA* l) {  
    l->cabeca = (PONT) malloc(sizeof(ELEMENTO));  
    l->cabeca->prox = l->cabeca;  
}
```



Buscar por elemento

A função de busca deverá:

- Receber uma chave do usuário

- Retornar o endereço em que este elemento se encontra (caso seja encontrado)

- Retornar *NULL* caso não haja um registro com essa chave na lista

Podemos usar o nó cabeça como sentinela

Busca sequencial

```
PONT buscaSentinela(LISTA* l, TIPOCHAVE ch) {  
    PONT pos = l->cabeca->prox;  
    l->cabeca->reg.chave = ch;  
    while (pos->reg.chave != ch) pos = pos->prox;  
    if (pos != l->cabeca) return pos;  
    return NULL;  
}
```

Busca sequencial - lista ordenada

```
// lista não precisa estar ordenada pelos valores das chaves
PONT buscaSentinela(LISTA* l, TIPOCHAVE ch) {
    PONT pos = l->cabeca->prox;
    l->cabeca->reg.chave = ch;
    while (pos->reg.chave != ch) pos = pos->prox;
    if (pos != l->cabeca) return pos;
    return NULL;
}
```

```
// lista ordenada pelos valores das chaves dos registros
PONT buscaSentinelaOrd(LISTA* l, TIPOCHAVE ch) {
    PONT pos = l->cabeca->prox;
    l->cabeca->reg.chave = ch;
    while (pos->reg.chave < ch) pos = pos->prox;
    if (pos != l->cabeca && pos->reg.chave==ch) return pos;
    return NULL;
}
```

Inserção de um elemento

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

Realizaremos a inserção **ordenada pelo valor da chave** do registro passado e não permitiremos a inserção de **elementos repetidos**; Na inserção precisamos identificar **entre quais elementos** o novo elemento será inserido;

Alocaremos memória para o novo elemento.

Precisamos saber quem será o **predecessor do elemento**.

Busca - auxiliar

```
PONT buscaSeqExc(LISTA* l, TIPOCHAVE ch, PONT* ant) {  
    *ant = l->cabeca;  
    PONT atual = l->cabeca->prox;  
    l->cabeca->reg.chave = ch;  
    while (atual->reg.chave<ch) {  
        *ant = atual;  
        atual = atual->prox;  
    }  
    if (atual != l->cabeca && atual->reg.chave == ch) return atual;  
    return NULL;  
}
```

Inserção ordenada - sem duplicação

```
bool inserirElemListaOrd(LISTA* l, REGISTRO reg) {  
    PONT ant, i;  
    i = buscaSeqExc(l, reg.chave, &ant);  
    if (i != NULL) return false;  
    i = (PONT) malloc(sizeof(ELEMENTO));  
    i->reg = reg;  
    i->prox = ant->prox;  
    ant->prox = i;  
    return true;  
}
```

Exclusão de um elemento

- O usuário passa a chave do elemento que ele quer excluir
- Se houver um elemento com esta chave na lista, **exclui este elemento** da lista, **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 excluirElemLista(LISTA* l, TIPOCHAVE ch) {  
    PONT ant, i;  
    i = buscaSeqExc(l, ch, &ant);  
    if (i == NULL) return false;  
    ant->prox = i->prox;  
    free(i);  
    return true;  
}
```

4. Pilha

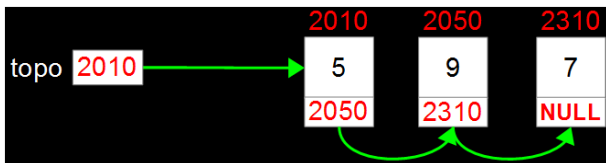
Pilha é uma estrutura linear na qual:

- 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

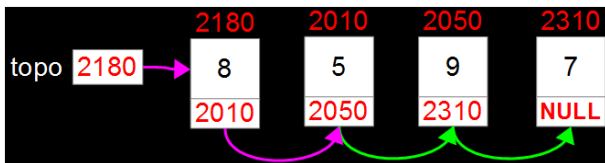
- Alocaremos e desalocaremos a memória para os elementos **sob demanda**;
- Vantagem: não precisamos **gastar memória** que não estamos usando;
- Cada elemento indicará quem é seu **sucessor** (quem está “abaixo” dele na pilha);
- Controlaremos o endereço do elemento que está no **topo** da pilha.

Ideia



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

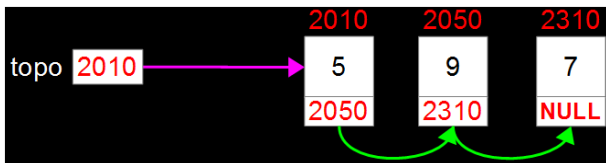
Ideia



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?

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;
```

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

Inserção de um elemento (*push*)

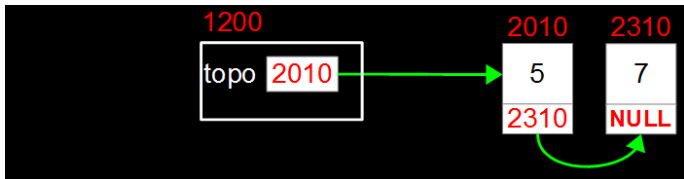
O usuário passa como parâmetro um registro a ser inserido na pilha

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..

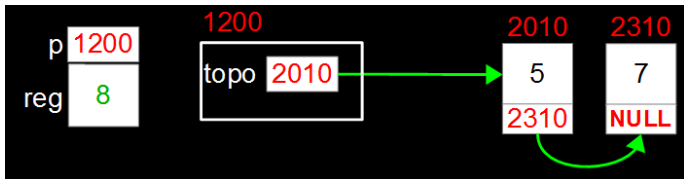
Inserção de um elemento (*push*)

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



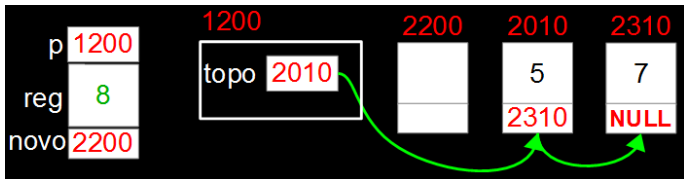
Inserção de um elemento (*push*)

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



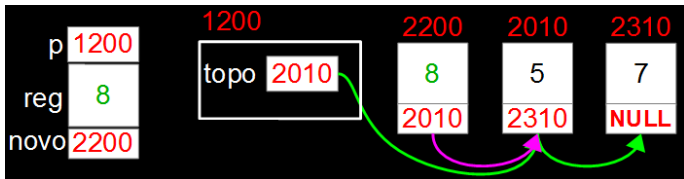
Inserção de um elemento (*push*)

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



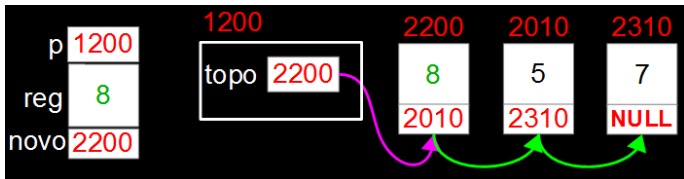
Inserção de um elemento (*push*)

```
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**:
 - 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) {  
    if ( p->topo == NULL) return false;  
    *reg = p->topo->reg;  
    PONT apagar = p->topo;  
    p->topo = p->topo->prox;  
    free(apagar);  
    return true;  
}
```

Fila

É uma estrutura linear na qual:

- As **inserções** ocorrem no **final** da fila;
- As **exclusões** ocorrem no **início** da fila.
- Utiliza a mesma lógica de uma **fila de pessoas**.

5. Fila (implementação estática)

Fila - implementação estática

Utilizaremos um **arranjo** de elementos de tamanho predefinido;
Controlaremos a posição do elemento que está no **início** da fila.

Controlaremos o **número de elementos** da fila.

Fila (ideia geral)

Temos um **arranjo** de elementos, um campo indicando a **posição do primeiro** e um indicando o **número de elementos**

O sucessor de cada elemento está na **próxima posição do arranjo** (exceto o sucessor do último que estará na posição 0)

A	?	?	5	7	2
inicio	2				
nroElem	3				

Fila (ideia geral)

Temos um **arranjo** de elementos, um campo indicando a **posição do primeiro** e um indicando o **número de elementos**

O sucessor de cada elemento está na **próxima posição do arranjo** (exceto o sucessor do último que estará na posição 0)

Como inserimos o **elemento 8**?

A	8	?	5	7	2
inicio	2				
nroElem	4				

Fila (ideia geral)

Temos um **arranjo** de elementos, um campo indicando a **posição do primeiro** e um indicando o **número de elementos**

O sucessor de cada elemento está na **próxima posição do arranjo** (exceto o sucessor do último que estará na posição 0)

Como inserimos o **elemento 8**?

Como **excluimos um elemento**?

A	8	?	5	7	2
inicio	3				
nroElem	3				

Modelagem

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

```
#define MAX 50
```

```
#define true 1
```

```
#define false 0
```

```
typedef int bool;
```

```
typedef int TIPOCHAVE;
```

```
typedef struct {  
    TIPOCHAVE chave;  
} REGISTRO;
```

```
typedef struct {  
    REGISTRO A[MAX];  
    int inicio;  
    int nroElem;  
} FILA;
```

Funções de gerenciamento

Implementaremos funções para:

- Inicializar a estrutura

- Retornar a quantidade de elementos válidos

- Exibir os elementos da estrutura

- Inserir elementos na estrutura (no fim)**

- Excluir elementos da estrutura (no início)**

- Reinicializar a estrutura

Inserção de um elemento

O usuário passa como parâmetro um registro a ser inserido no final da fila

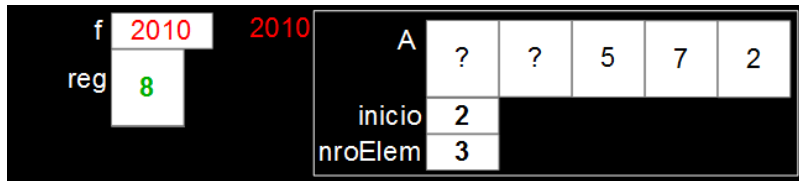
Se a fila não estiver **cheia**, **precisamos**:

Identificar a **posição** no arranjo na qual o registro será inserido e inseri-lo lá;

Alterar o valor campo ***nroElem***.

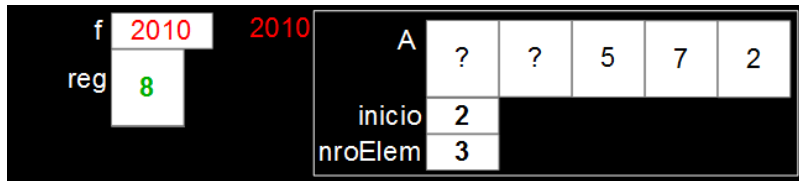
Inserção de um elemento

```
bool inserirElementoFila(FILA* f, REGISTRO reg) {  
    if (f->nroElem >= MAX) return false;  
    int posicao = (f->inicio + f->nroElem) % MAX;  
    f->A[posicao] = reg;  
    f->nroElem++;  
    return true;  
}
```



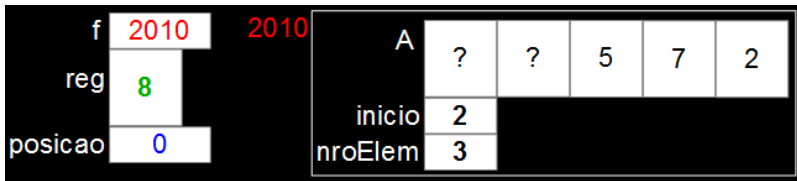
Inserção de um elemento

```
bool inserirElementoFila(FILA* f, REGISTRO reg) {  
    if (f->nroElem >= MAX) return false;  
    int posicao = (f->inicio + f->nroElem) % MAX;  
    f->A[posicao] = reg;  
    f->nroElem++;  
    return true;  
}
```



Inserção de um elemento

```
bool inserirElementoFila(FILA* f, REGISTRO reg) {  
    if (f->nroElem >= MAX) return false;  
    int posicao = (f->inicio + f->nroElem) % MAX;  
    f->A[posicao] = reg;  
    f->nroElem++;  
    return true;  
}
```



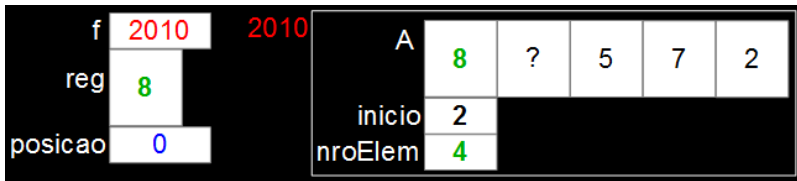
Inserção de um elemento

```
bool inserirElementoFila(FILA* f, REGISTRO reg) {  
    if (f->nroElem >= MAX) return false;  
    int posicao = (f->inicio + f->nroElem) % MAX;  
    f->A[posicao] = reg;  
    f->nroElem++;  
    return true;  
}
```

f	2010	2010	A	8	?	5	7	2
reg	8							
posicao	0		inicio	2				
			nroElem	3				

Inserção de um elemento

```
bool inserirElementoFila(FILA* f, REGISTRO reg) {  
    if (f->nroElem >= MAX) return false;  
    int posicao = (f->inicio + f->nroElem) % MAX;  
    f->A[posicao] = reg;  
    f->nroElem++;  
    return true;  
}
```



Inserção de um elemento

```
bool inserirElementoFila(FILA* f, REGISTRO reg) {  
    if (f->nroElem >= MAX) return false;  
    int posicao = (f->inicio + f->nroElem) % MAX;  
    f->A[posicao] = reg;  
    f->nroElem++;  
    return true;  
}
```

f	2010	2010	A	8	?	5	7	2
reg	8							
posicao	0		inicio	2				
			nroElem	4				

Exclusão de um elemento

O usuário solicita a exclusão do elemento do **início da fila**. Se a fila **não estiver vazia**:

- Iremos **copiar esse elemento para um local indicado pelo usuário**;
- Acertar o valor do campo ***nroElem***;
- Acertar o valor do campo ***inicio***.

Exclusão de um elemento

```
bool excluirElementoFila(FILA* f, REGISTRO* reg) {  
    if (f->nroElem==0) return false;  
    *reg = f->A[f->inicio];  
    f->inicio = (f->inicio+1) % MAX;  
    f->nroElem--;  
    return true;  
}
```

6. Fila (implementação dinâmica)

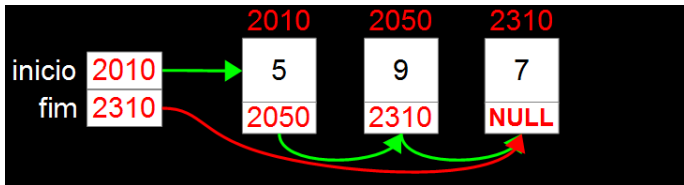
Fila - implementação dinâmica

- Alocaremos e desalocaremos a memória para os elementos **sob demanda**;
- Cada elemento indicará quem é seu **sucessor** (quem é o “próximo” na fila);
- Controlaremos os endereços dos elemento que estão no **início** e no **fim** da fila.

Fila (ideia geral)

Temos dois campos para indicar os endereços de quem está no **início** no **fim** da fila

Cada elemento aponta para seu **sucessor**

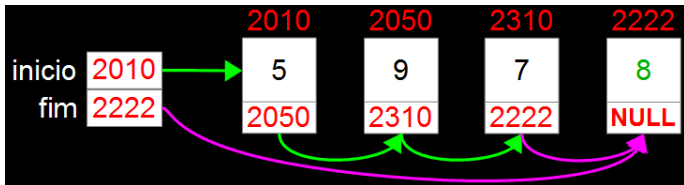


Fila (ideia geral)

Temos dois campos para indicar os endereços de quem está no **início** no **fim** da fila

Cada elemento aponta para seu **sucessor**

Como inserimos o **elemento 8**?



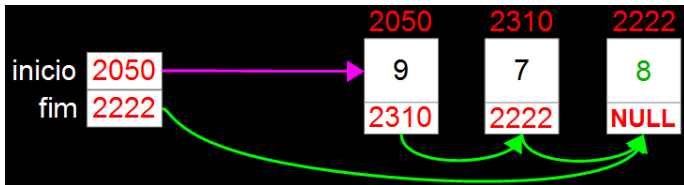
Fila (ideia geral)

Temos dois campos para indicar os endereços de quem está no **início** no **fim** da fila

Cada elemento aponta para seu **sucessor**

Como inserimos o **elemento 8**?

Como **excluimos um elemento**?



Modelagem

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

```
#define true 1
#define false 0
```

```
typedef int bool;
typedef int TIPOCHAVE;
```

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

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

```
typedef struct {
    PONT inicio;
    PONT fim;
} FILA;
```

Funções de gerenciamento

Implementaremos funções para:

- Inicializar a estrutura

- Retornar a quantidade de elementos válidos

- Exibir os elementos da estrutura

- Inserir elementos na estrutura (no fim)**

- Excluir elementos da estrutura (no início)**

- Reinicializar a estrutura

Inserção de um elemento

O usuário passa como parâmetro um registro a ser inserido no final da fila, **precisamos**:

- Alocar** a memória para este novo elemento;

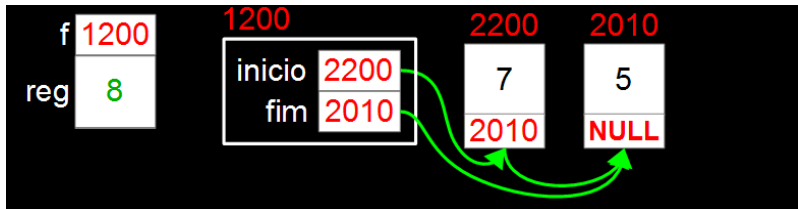
- Colocá-lo **após o último elemento** da fila;

- Alterar o valor do campo *fim*.

- Atenção**: a fila poderia estar **vazia**.

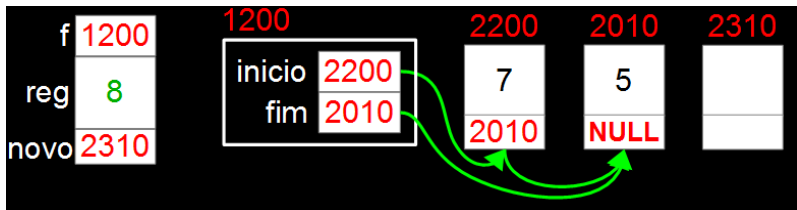
Inserção de um elemento

```
bool inserirNaFila(FILA* f,REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = NULL;  
    if (f->inicio==NULL) f->inicio = novo;  
    else f->fim->prox = novo;  
    f->fim = novo;  
    return true;  
}
```



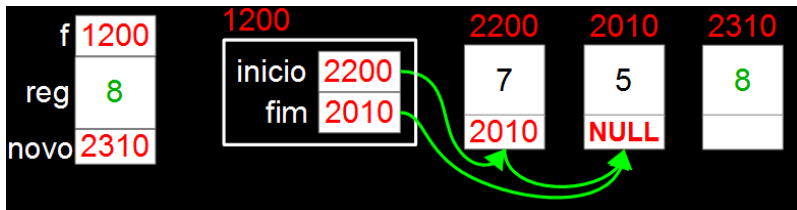
Inserção de um elemento

```
bool inserirNaFila(FILA* f,REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = NULL;  
    if (f->inicio==NULL) f->inicio = novo;  
    else f->fim->prox = novo;  
    f->fim = novo;  
    return true;  
}
```



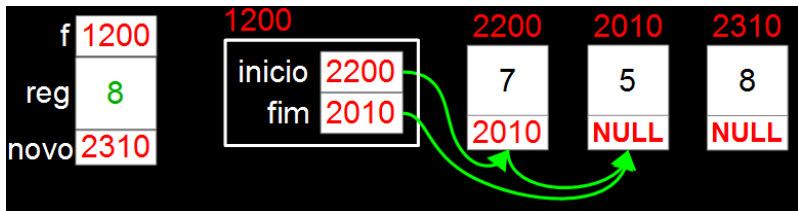
Inserção de um elemento

```
bool inserirNaFila(FILA* f, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = NULL;  
    if (f->inicio==NULL) f->inicio = novo;  
    else f->fim->prox = novo;  
    f->fim = novo;  
    return true;  
}
```



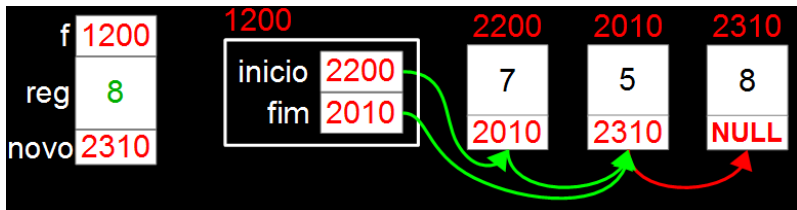
Inserção de um elemento

```
bool inserirNaFila(FILA* f,REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = NULL;  
    if (f->inicio==NULL) f->inicio = novo;  
    else f->fim->prox = novo;  
    f->fim = novo;  
    return true;  
}
```



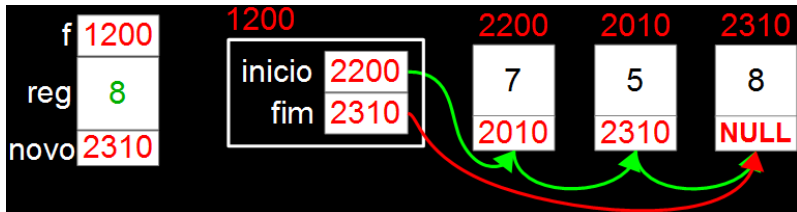
Inserção de um elemento

```
bool inserirNaFila(FILA* f,REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = NULL;  
    if (f->inicio==NULL) f->inicio = novo;  
    else f->fim->prox = novo;  
    f->fim = novo;  
    return true;  
}
```



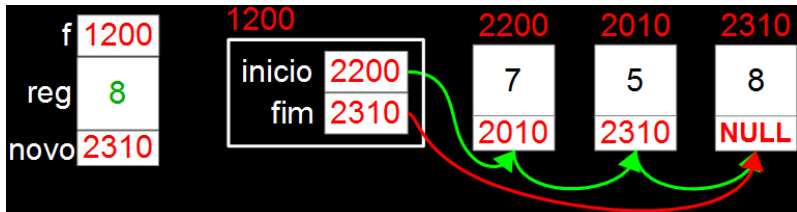
Inserção de um elemento

```
bool inserirNaFila(FILA* f,REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = NULL;  
    if (f->inicio==NULL) f->inicio = novo;  
    else f->fim->prox = novo;  
    f->fim = novo;  
    return true;  
}
```



Inserção de um elemento

```
bool inserirNaFila(FILA* f,REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = NULL;  
    if (f->inicio==NULL) f->inicio = novo;  
    else f->fim->prox = novo;  
    f->fim = novo;  
    return true;  
}
```



Inserção de um elemento

```
bool inserirNaFila(FILA* f, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = NULL;  
    if (f->inicio==NULL) f->inicio = novo;  
    else f->fim->prox = novo;  
    f->fim = novo;  
    return true;  
}
```

f	1200
reg	8

1200

inicio	NULL
fim	NULL

Inserção de um elemento

```
bool inserirNaFila(FILA* f, REGISTRO reg) {  
    PONT novo = (PONT) malloc(sizeof(ELEMENTO));  
    novo->reg = reg;  
    novo->prox = NULL;  
    if (f->inicio==NULL) f->inicio = novo;  
    else f->fim->prox = novo;  
    f->fim = novo;  
    return true;  
}
```



Exclusão de um elemento

O usuário solicita a exclusão do elemento do **início da fila**. Se a fila **não estiver vazia**:

- Iremos **copiar esse elemento para um local indicado pelo usuário**;
- Acertar o valor do campo **início**;
- Eventualmente acertar o valor do campo **fim**.

Exclusão de um elemento

```
bool excluirDaFila(FILA* f, REGISTRO* reg) {  
    if (f->inicio==NULL) return false;  
    *reg = f->inicio->reg;  
    PONT apagar = f->inicio;  
    f->inicio = f->inicio->prox;  
    free(apagar);  
    if (f->inicio == NULL) f->fim = NULL;  
    return true;  
}
```

SIN5013

Análise de Algoritmos e Estruturas de Dados

Estruturas Lineares - Listas, Pilhas e Filas

Prof. Luciano Antonio Digiampietri