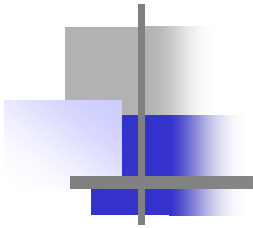


INF1007

Listas Encadeadas (*Linked Lists*)

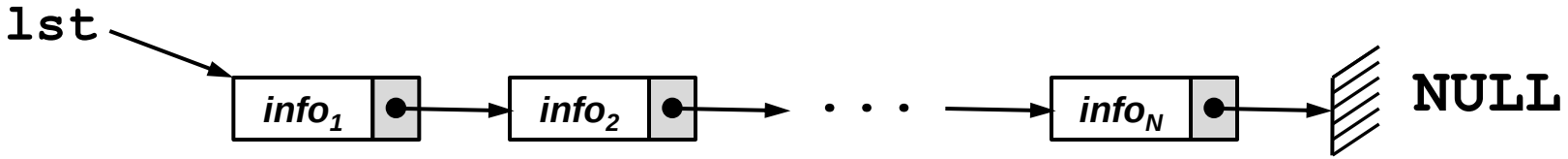
Dept. de Informática, PUC-Rio

Vetores *vs* Estruturas Dinâmicas

- **Vetores (*arrays*):**
  - **Ocupa um espaço contíguo de memória**
  - **Permite acesso randômico**
  - **Requer pré-dimensionamento de espaço de memória**
- **Estruturas Dinâmicas**
  - **Crescem (ou decrescem) à medida que elementos são inseridos (ou removidos)**
  - **Listas Encadeadas (*linked lists*)**

Listas Encadeadas

- **Lista Encadeada é uma sequência de elementos, onde cada elemento tem uma informação armazenada (*info*) e um ponteiro para o próximo elemento da sequência. O último aponta para NULL**



- **A maneira mais simples de representar uma lista é pelo ponteiro para o primeiro elemento da sequência**
  - **usualmente chamado de "*head*", "cabeça", ou simplesmente "lst"**
- **Como definir uma estrutura para ser um elemento de uma lista?**
- **… uma estrutura *recursiva* (c/ponteiro para a própria estrutura):**

```
struct elemento
{
   int info;
   struct elemento * prox;
};
typedef struct elemento Elemento;
```

ou "node"

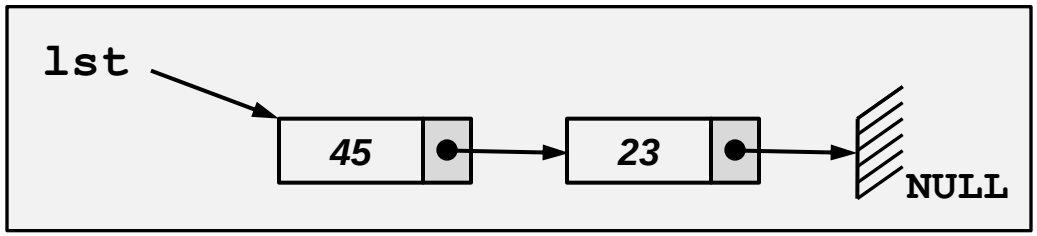
ou "* next"

```
struct elemento
{
   float info;
   struct elemento * prox;
};
typedef struct elemento Elemento;
```

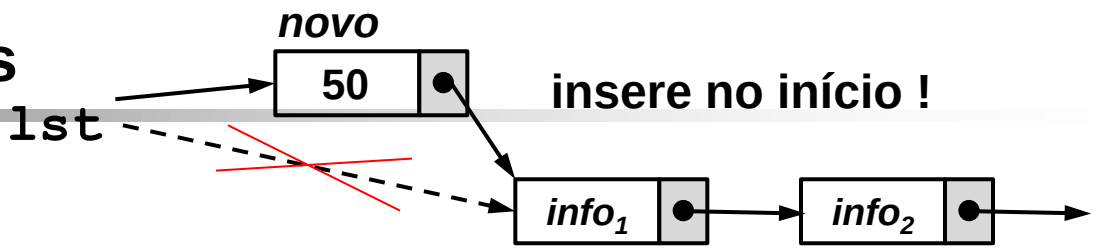
Agora, não vá de imediato para o próximo slide!

Tente (sem olhar o próximo slide) escrever as seguintes funções:

- **lst_insere**, que insere um novo elemento de lista no início da lista *lst* e retorna uma nova lista (i.e. uma nova cabeça);
- **lst_imprime**, que imprime todos os elementos de uma lista *lst*;
- **main**, que monta a seguinte lista:



Lista de inteiros

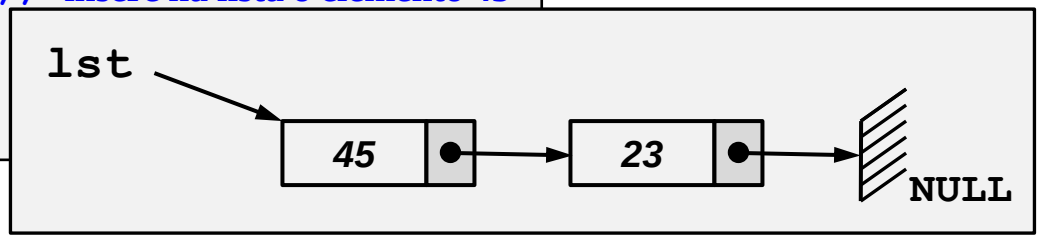


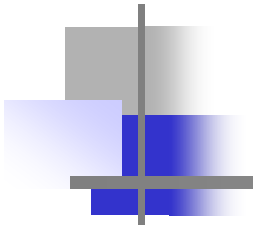

```
Elemento * lst_insere(Elemento * lst, int a)
{
   Elemento * novo = (Elemento *)malloc(sizeof(Elemento));
   if (novo != NULL)
   {
      novo->info = a;
      novo->prox = lst;
   }
   return novo;
}
```

Fazer "*if (novo == NULL) return NULL*" também funcionaria. Porém, sempre que possível (e se isto não comprometer a legibilidade do código), evitamos múltiplos *returns*. Um único ponto de saída facilita a análise da função.

```
void lst_imprime(Elemento * lst)
{
   Elemento * p;
   for (p = lst; p != NULL; p = p->prox)
      printf("info = %d\n", p->info);
}
```

```
int main (void)
{
   Elemento * lst; // declara uma lista não inicializada
   lst = NULL; // cria e inicializa lista como vazia
   lst = lst_insere(lst, 23); // insere na lista o elemento 23
   lst = lst_insere(lst, 45); // insere na lista o elemento 45
   lst_imprime(lst);
   return 0;
}
```





Agora, sem olhar o próximo slide, tente escrever a função **lst_busca** que busca o elemento contendo um valor dado e retorna o seu endereço.

Esta busca é sequential (linear).

Busca Sequencial

- **Retorna o ponteiro do elemento da lista que contém a informação procurada ou retorna NULL se não a encontrar**

```
Elemento * busca (Elemento * lst, int v)
{
   Elemento * p;
   for (p=lst; p!=NULL; p = p->prox)
      if (p->info == v)
         return p;
   return NULL;          /* não achou o elemento */
}
```

Aqui fica mais legível e fácil ter múltiplos *returns*.

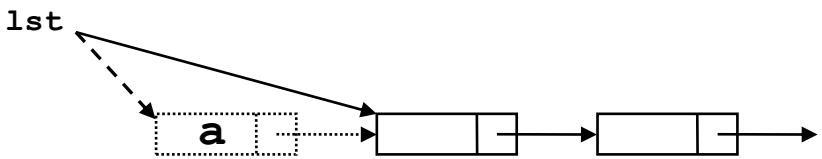
Agora, sem olhar o próximo slide, tente escrever a função

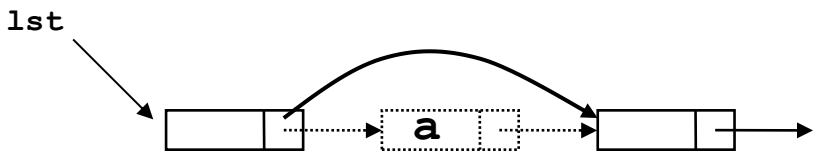
`Elemento * lst_retira(Elemento * lst, int a)`

que retira o primeiro elemento contendo o valor a.

Esta é uma função difícil de escrever!

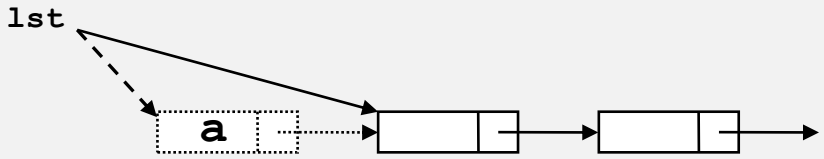
Dica: use um ponteiro auxiliar ***ant***, que sempre guarda o endereço do elemento imediatamente anterior ao elemento corrente. E considere que há duas situações distintas (o valor a ser retirado está na cabeça da lista ou o valor está em um elemento interno). Lembre também que antes de liberar a memória, você tem que fazer a ligação entre os elementos

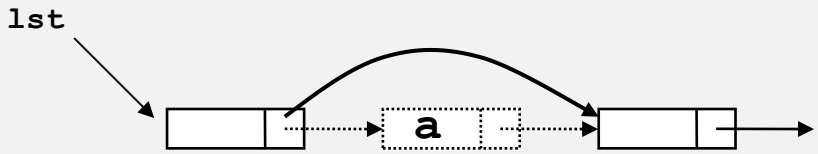




Remover um Elemento

```
Elemento * lst_retira(Elemento * lst, int a)
{
   Elemento * ant = NULL; /* ponteiro para elemento anterior */
   Elemento * p = lst;    /* ponteiro para percorrer a lista */
   /* procura elemento na lista, guardando anterior */
   while (p != NULL && p->info != a)
   {
      ant = p;
      p = p->prox;
   }
   if (p == NULL)
      return lst;  /* não achou: retorna lista original */
   /* retira elemento */
   if (ant == NULL)
   { /* retira elemento do inicio */
      lst = p->prox; }
   else { /* retira elemento do meio da lista */
      ant->prox = p->prox; }
   free(p);
   return lst;
}
```





Agora, escreva as seguintes funções:

- **lst_libera**, que destrói a lista, liberando todos os elementos alocados (o cuidado aqui é guardar a referência para o próximo elemento usando um ponteiro auxiliar);
- **lst_igual**, que testa se duas listas são iguais (sugestão: crie dois ponteiros, um para percorrer cada lista)
- **lst_insere_ordenado** que, dada uma lista ordenada, insere um novo valor dado no local certo (sugestão: se inspire no algoritmo de remover)

Libera a Lista e Testa Igualdade

- **Destrói a lista, liberando todos os elementos alocados**

```
void lst_libera(Elemento * lst)
{
   Elemento * p = lst, * t;
   while (p != NULL) {
      t = p->prox;     /* guarda referência p/ próx. elemento   */
      free(p);        /* libera a memória apontada por p          */
      p = t;          /* faz p apontar para o próximo     */
   }
}
```

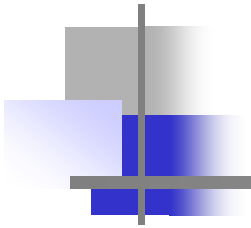
- **Igualdade requer dois ponteiros**

```
int lst_igual(Elemento * lst1, Elemento * lst2)
{
   Elemento * p1;  /* ponteiro para percorrer lst1 */
   Elemento * p2;  /* ponteiro para percorrer lst2 */
   for (p1=lst1, p2=lst2; p1 != NULL && p2 != NULL;
         p1 = p1->prox, p2 = p2->prox)
      if (p1->info != p2->info)
         return 0;
   return p1 == p2;
}
```

Esta comparação considera o caso das listas de tamanho diferente. Se os dois chegaram ao NULL (ou já eram ambos NULL) significa que são do mesmo tamanho. Se um é NULL e o outro não, então têm tamanhos diferentes!
Poderia também ser: `return p1==NULL && p2==NULL;`

Insere Ordenado

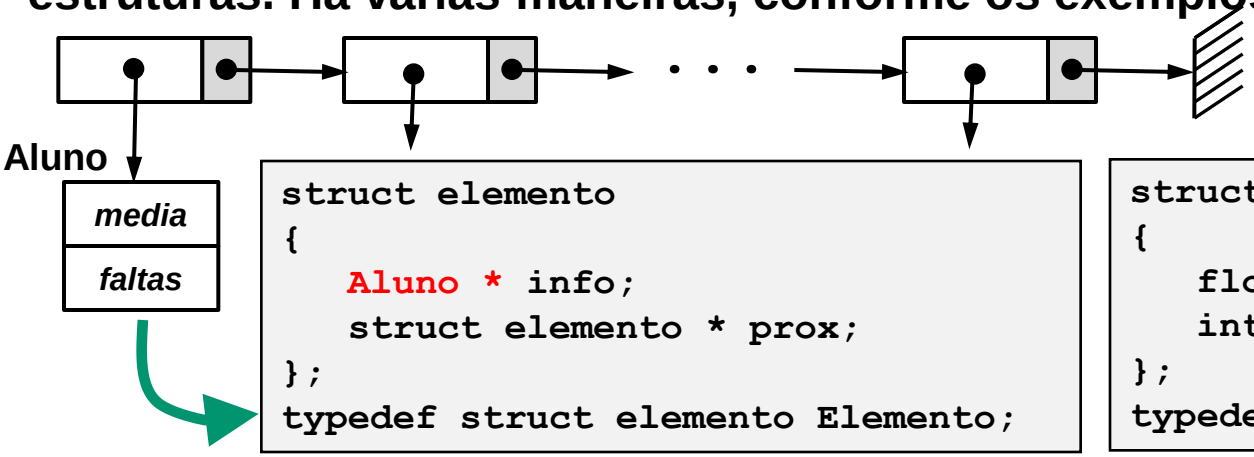
```
Elemento * lst_insere_ordenado(Elemento * lst, int a)
{
   Elemento * novo;
   Elemento * ant = NULL;   /* ponteiro para elemento anterior */
   Elemento * p = lst;      /* ponteiro para percorrer a lista */
   /* procura posição de inserção */
   while (p != NULL && p->info < a)
   {  ant = p; p = p->prox; }
   /* cria novo elemento */
   novo = (Elemento *) malloc(sizeof(Elemento));
   novo->info = a;
   /* encadeia elemento */
   if (ant == NULL)
   { /* insere elemento no início    */
      novo->prox = lst;
      lst = novo; }
   else { /* insere elemento no meio da lista */
      novo->prox = ant->prox;
      ant->prox = novo; }
   return lst;
}
```



LISTAS DE TIPOS ESTRUTURADOS

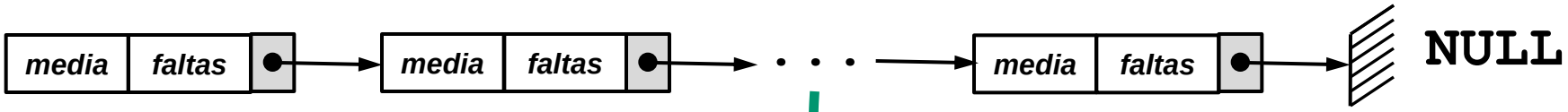
Listas de Tipos Estruturados

- **A informação associada a cada elemento pode ser mais complexa, do tipo estruturas. Há várias maneiras, conforme os exemplos a seguir:**

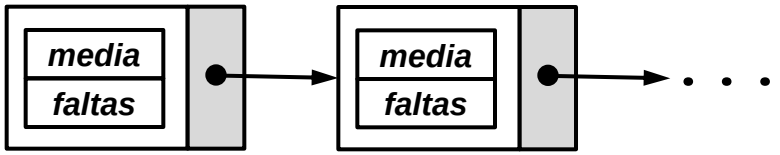



```
struct elemento
{
   Aluno * info;
   struct elemento * prox;
};
typedef struct elemento Elemento;
```

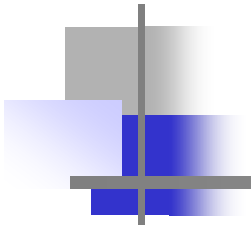
```
struct aluno
{
   float media;
   int faltas;
};
typedef struct aluno Aluno;
```




```
struct elemento
{
   float media;
   int faltas;
   struct elemento * prox;
};
```



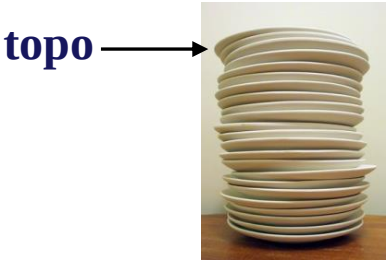

```
struct elemento
{
   Aluno info;
   struct elemento * prox;
};
```



DOIS USOS ESPECIAIS DE LISTAS: PILHA E FILA

PILHA e FILA

- **PILHA (*stack*): novo elemento é inserido no topo e acesso é apenas ao topo**
  - **... como numa pilha de pratos**



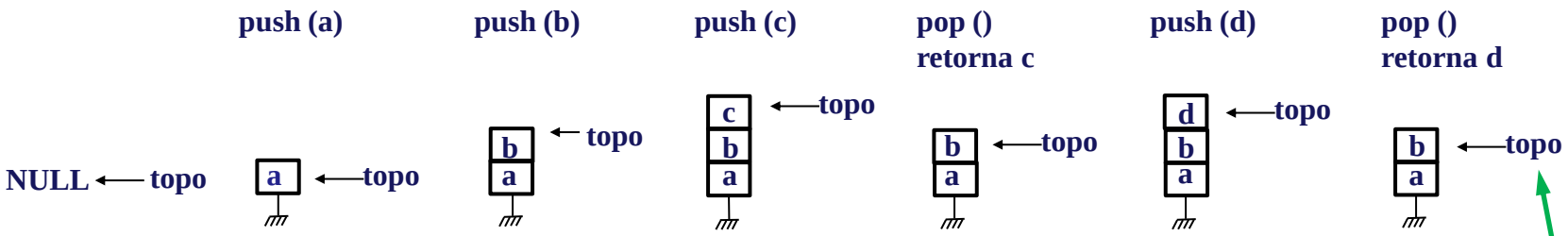
- **PILHA: único elemento que pode ser acessado e removido é o do topo**
- **PILHA: tem duas operações básicas**
  - **Empilhar (*push*) um novo elemento, inserindo-o no topo**
  - **Desempilhar (*pop*) um elemento, removendo-o do topo (i.e., leu, então removeu)**
- **Na PILHA, os elementos são retirados na ordem inversa à ordem em que foram colocados: o 1o. que sai é o último que entrou**

- **Na FILA (*Queue*) é o inverso: 1o. a chegar, primeiro a sair**
- **FILA tem duas operações básicas:**
  - **Inserir: colocar um elemento no fim da fila**
  - **Retirar: remover um elemento no início da fila**

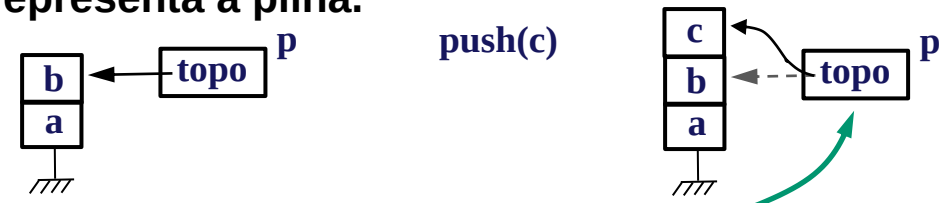
Tentar remover elemento de uma PILHA vazia ou de uma FILA vazia gera exceção (i.e. use *exit*()).

PILHA como LISTA

Há duas maneiras de trabalhar com listas: diretamente com um ponteiro ou através de uma estrutura. Por exemplo, no caso de uma PILHA, podemos ter:



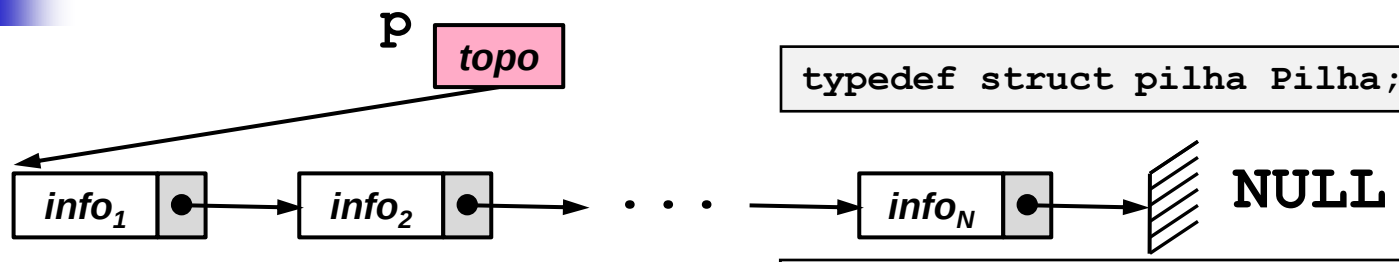
No caso acima, a pilha é representada pelo ponteiro *topo*.
Porém, é mais geral e fácil colocar o ponteiro para o topo da pilha dentro de uma estrutura chamada "pilha" (que representará a pilha). Na figura abaixo, a estrutura "p" representa a pilha.



Representar a pilha desta forma (com uma nova estrutura contendo apenas um ponteiro, ao invés de uma simples variável apontando para a lista) é necessária para definir serviços gerais de pilhas. Você verá isto com detalhes na disciplina que trata de tipos abstratos de dados.

OBS: tanto PILHA como FILA também podem ser representadas por *arrays* (usamos *arrays* quando o tamanho de memória é pequeno e fixo e quando queremos rapidez de acesso)

Exemplo de PILHA




```
typedef struct pilha Pilha;
```

```
struct elemento
{
   int info;
   struct elemento * prox;
};
typedef struct elemento Elemento;
```

```
struct pilha
{
   Elemento * topo;
};
```

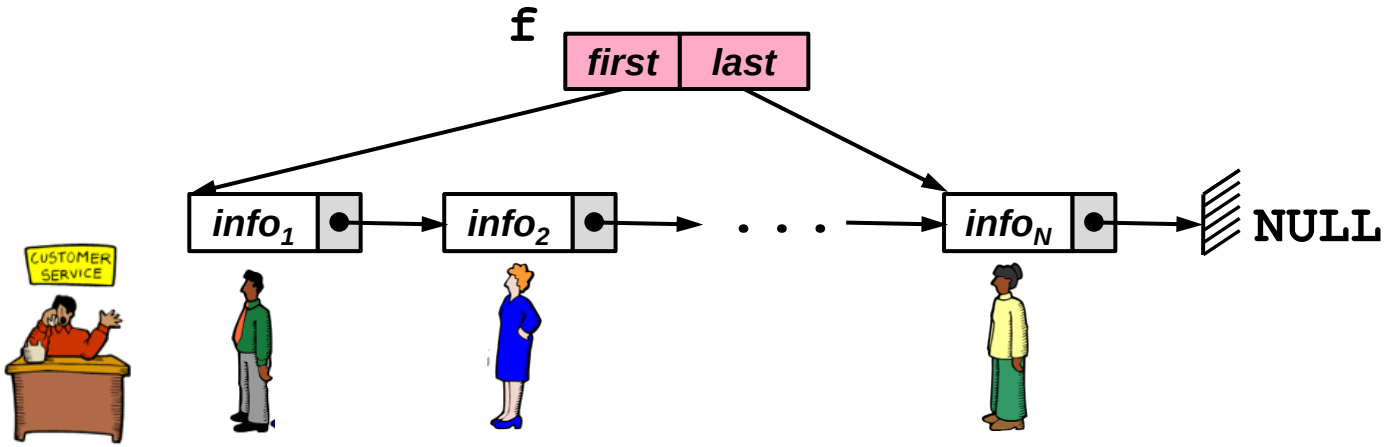
```
Pilha * pilha_cria(void)
{
   Pilha * p =
     (Pilha *) malloc(sizeof(Pilha));
   if (p != NULL)
       p->topo = NULL;
   return p;
}
```

```
int pilha_pop(Pilha * p)
{
   Elemento * t;
   int a;
   if (p->topo == NULL)
       exit(1);
   t = p->topo;
   a = t->info;
   p->topo = t->prox;
   free(t);
   return a;
}
```

```
void pilha_push(Pilha * p, int a)
{
   Elemento * t=(Elemento *) malloc(sizeof(Elemento));
   if (t==NULL) exit(1);
   t->info= a;
   t->prox= p->topo;
   p->topo= t;
}
```

FILA como LISTA

- **A FILA é representada por uma estrutura que contém um ponteiro para o primeiro elemento e um outro ponteiro para o último elemento.**




```
struct fila
{
   Elemento * first;
   Elemento * last;
};
typedef struct fila Fila;
```

Nessa **fila**, podemos inserir novos elementos no fim e ler/atender no início (como numa fila de banco, onde o caixa atende o primeiro da fila e diminui a fila). Outros nomes usuais para ***first*** e ***last*** são: ***front*** e ***rear***.

Você vai trabalhar com **Filas** na disciplina de Estruturas de Dados Avançadas.

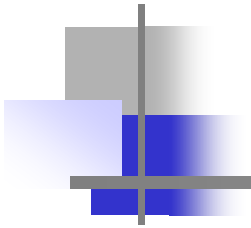
- **Escreva as funções *insere* e *retira***
- **Uma outra variante é incluir o número de elementos:**

Exemplo de FILA

```
Fila * fila_cria(void)
{
   Fila * f = (Fila *)malloc(sizeof(Fila));
   if (f != NULL)
      f->first = f->last = NULL;
   return f;
}
```

```
void fila_insere (Fila * f, int v)
{
   Elemento * e = (Elemento *)malloc(sizeof(Elemento));
   if (e == NULL) exit(1);
   n->info = v;  // armazena informacao
   n->prox = NULL;  // novo elemento eh o ultimo
   if (f->first != NULL) // fila nao vazia?
      f->last->prox = e;
   else
      f->first = e;
   f->last = e;   // fila aponta para novo elemento
}
```

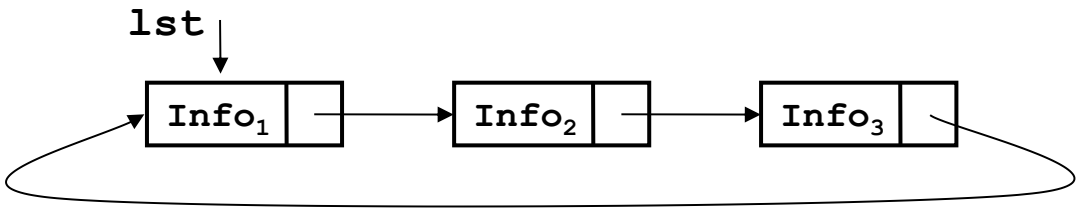
```
int fila_retira(Fila * f)
{
   Elemento * t;
   int v;
   if (f->first == NULL)
      exit(1);
   t = f->first;
   v = t->info;
   f->first = t->prox;
   if (f->first == NULL) // vazia?
      f->last = NULL;
   free(t);
   return v;
}
```



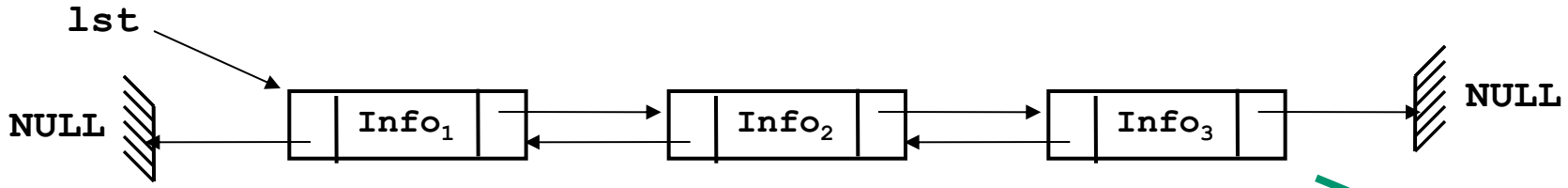
LISTAS CIRCULARES E DUPLAMENTE ENCADEADAS

Listas Circulares e Duplamente Encadeadas

- **Listas Circulares**



- **Listas Duplamente Encadeadas**

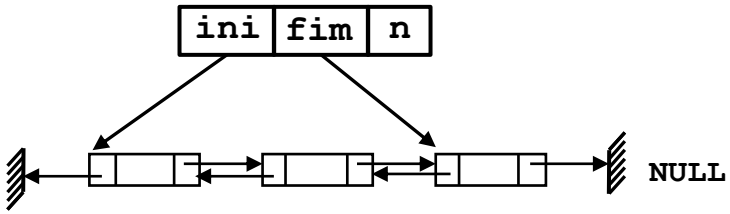



```
Elemento2 * lst2_insere(Elemento2 * lst, int a)
{
   Elemento2 * novo = (Elemento2 *) malloc(sizeof(Elemento2));
   novo->info = a;
   novo->prox = lst;
   novo->ant = NULL;
   /* verifica se lista não estava vazia */
   if (lst != NULL)
      lst->ant = novo;
   return novo;
}
```

```
struct elemento2
{
   int info;
   struct elemento2 * ant;
   struct elemento2 * prox;
};
typedef struct elemento2 Elemento2;
```

Exemplos de FILA como Lista

- **Fila duplamente encadeada**



- **Também pode ser circular simplesmente encadeada (Fig. a) ou circular duplamente encadeada (Fig. b). A estrutura [`ini` , `fim` , `n`] também pode ser usada (Fig. c)**

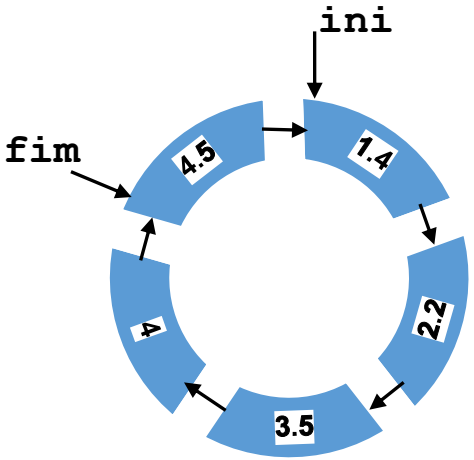


Fig. a

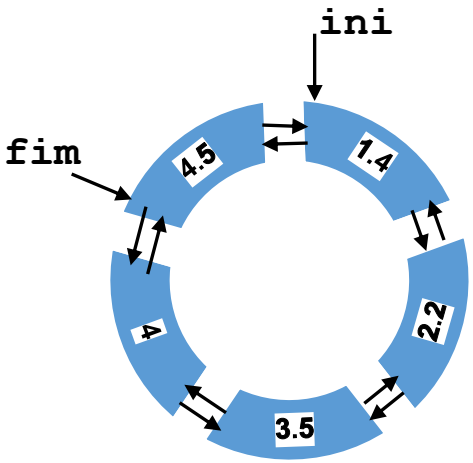


Fig. b

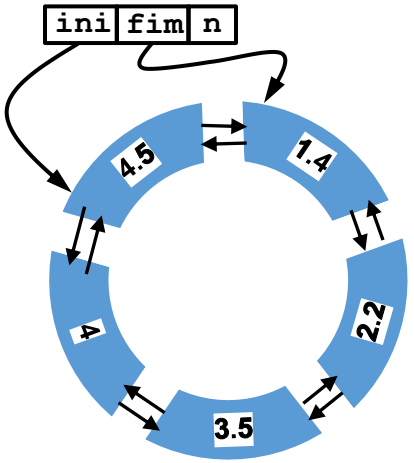


Fig. c