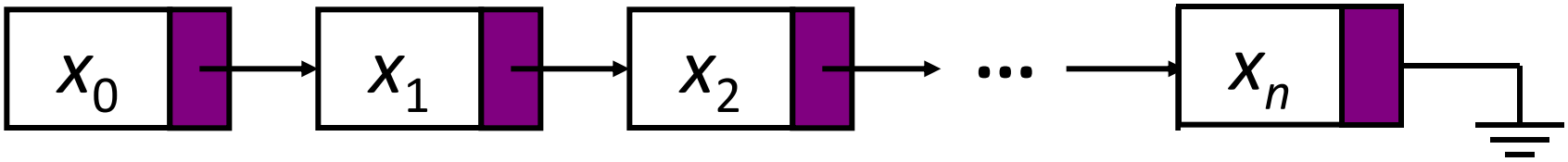
LISTAS ENCADEADAS OU NÃO- SEQÜENCIAIS

Listas Lineares

- Como visto anteriormente, as operações básicas para o “nosso” TAD Lista Linear são:
  - FLVazia
  - Vazia
  - Retira
  - Insere
  - Imprime
- A implementação através de *arrays* possui a grande desvantagem de ter que prever a quantidade máxima de elementos da lista antes de sua utilização efetiva

Implementação de Listas com Ponteiros

- Os elementos (nós ou células) da lista são registros com um dos componentes destinado a guardar o endereço do seu sucessor



- Desta forma, cada item da lista é encadeado com o seguinte através de uma variável do tipo Ponteiro
- Os nós da lista estão dispostos de maneira aleatória na memória (posições não obrigatoriamente contíguas) e são interligados por ponteiros: por isso é também chamada de lista não-seqüencial

A Lista Encadeada

- Assim, cada nó (ou célula) deve conter um item da lista e um "campo extra" para um apontador para o nó seguinte
- A alocação dinâmica de memória é a técnica utilizada: as posições de memória são alocadas (desalocadas) quando são necessárias (desnecessárias)
- Como visto, C faz a gerência de memória através das declarações *malloc* e *free*
- Existem outras implementações: simplesmente ou duplamente encadeadas e listas encadeadas circulares; com ou sem *nós sentinela*

Vantagens e Desvantagens

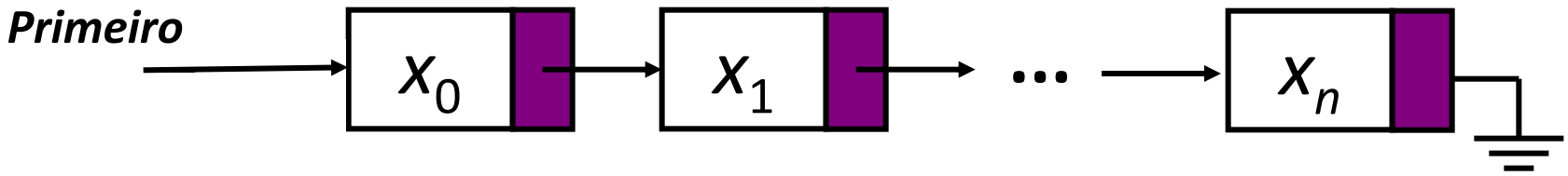
- Desvantagens
  1. Acesso indireto aos elementos
  2. Tempo variável para acessar os elementos (depende da posição do elemento)
  3. Gasto de memória maior pela necessidade de um novo campo para o ponteiro
- Vantagens
  1. A inserção e remoção de elementos podem ser feitas sem deslocar os itens seguintes da lista
  2. Não há necessidade de previsão do número de elementos da lista; o espaço necessário é alocado em tempo de execução
  3. Facilita o gerenciamento de várias listas (fusão, divisão,...)

Lista Simplesmente Encadeada

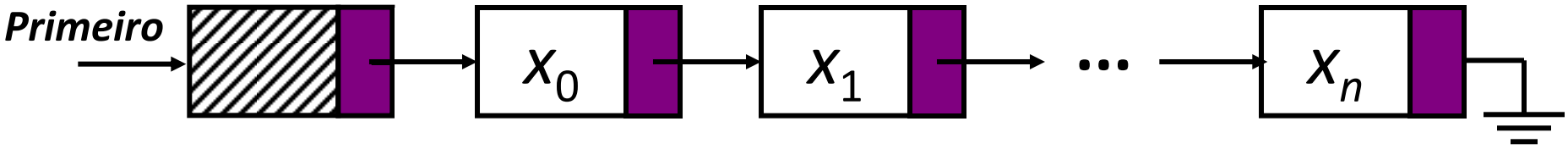
- Deve existir sempre uma indicação do primeiro nó: a partir dele a lista é percorrida (numa lista seqüencial, como seria?)
- Então, seria necessário um cuidado especial no tratamento do primeiro nó da lista:
  - Testes em algoritmos de inserção e remoção para verificar se este ponteiro é ou não igual a NULL
- Uma pequena variação na estrutura, com a adição de um nó chamado nó-cabeça, evita alguns testes com o 1o. nó e melhora o desempenho das operações na lista
- O nó-cabeça não contém informações relacionadas aos dados da lista e nunca é removido

Representação

Lista encadeada sem nó-cabeça



Lista encadeada com nó-cabeça



OBS: A implementação a seguir é de uma lista encadeada com nó-cabeça (ver mais informações nas referências indicadas)

Estrutura da Lista Encadeada

```
typedef int TipoChave;

typedef struct {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;

typedef struct Celula_str *Apontador;
typedef struct Celula_str {
  TipoItem Item;
  Apontador Prox;
} Celula;
```

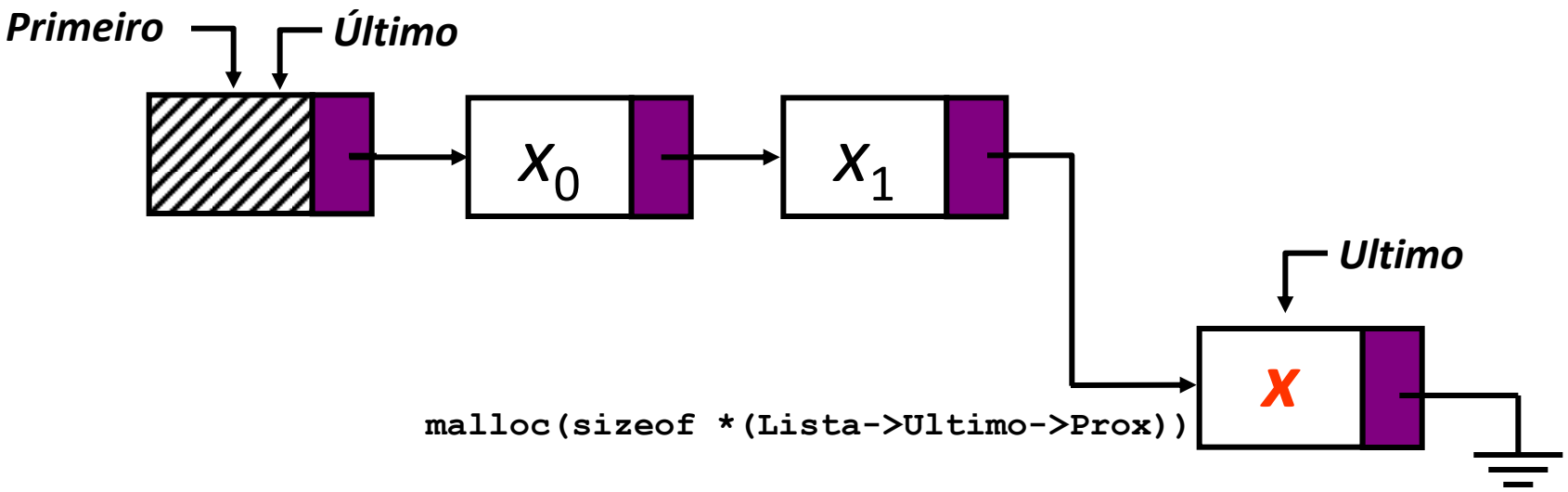
```
typedef struct {
  Apontador Primeiro, Ultimo;
} TipoLista;
```

Implementação: FLVazia e Vazia

```
void FLVazia(TipoLista *Lista)
{
  Lista->Primeiro = malloc(sizeof *(Lista->Primeiro));
  Lista->Ultimo = Lista->Primeiro;
  Lista->Primeiro->Prox = NULL;
}

int Vazia(TipoLista Lista)
{
  return (Lista.Primeiro == Lista.Ultimo);
}
```

Inserção em lista encadeada

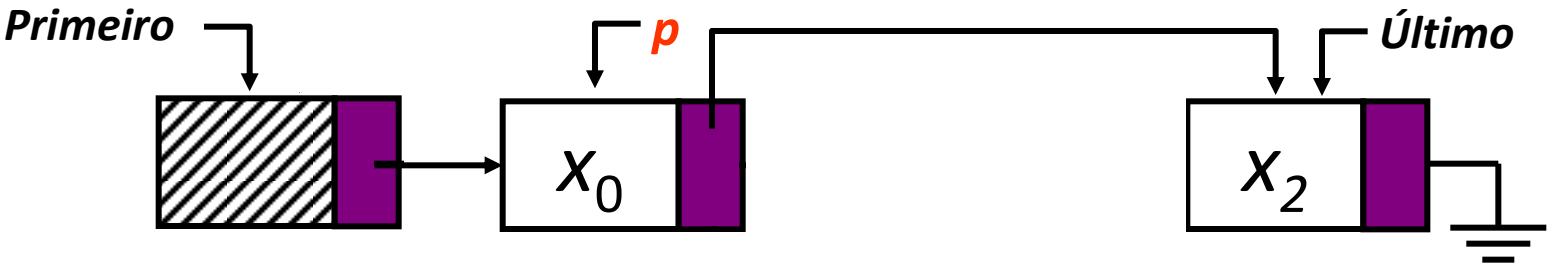



```
Lista->Ultimo->Prox=malloc(sizeof *(Lista->Ultimo->Prox));
Lista->Ultimo = Lista->Ultimo->Prox;
Lista->Ultimo->Item = x;
Lista->Ultimo->Prox = NULL;
```

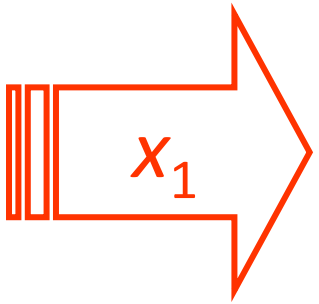
Implementação: Insere

```
void Insere(TipoItem x, TipoLista *Lista)
{
  Lista->Ultimo->Prox = malloc(sizeof *(Lista->Ultimo->Prox));
  Lista->Ultimo = Lista->Ultimo->Prox;
  Lista->Ultimo->Item = x;
  Lista->Ultimo->Prox = NULL;
}
```

Remoção em lista encadeada




```
aux = p->Prox;
*Item = aux->Item;
p->Prox = aux->Prox;
if(p->Prox == NULL)
   Lista->Último = p;
free(aux);
```



Implementação: Retira

```
void Retira(Apontador p, TipoLista *Lista, TipoItem *Item)
{
  /*  ---   Obs.: o item a ser retirado eh o seguinte ao apontado por  p --- */
  Apontador aux;
  if (Vazia(*Lista) || p == NULL || p->Prox == NULL)
  { printf(" Erro:  Lista vazia ou posicao nao existe\n");
    return;
  }
  aux = p->Prox;
  *Item = aux->Item;
  p->Prox = aux->Prox;
  if (p->Prox == NULL) Lista->Ultimo = p;
  free(aux);
}
```

Implementação: Imprime

```
void Imprime(TipoLista Lista)
{
  Apontador Aux;
  Aux = Lista.Primeiro->Prox;
  while (Aux != NULL)
    { printf("%d\n", Aux->Item.Chave);
      Aux = Aux->Prox;
    }
}
```