Estrutura de Dados

Listas

Prof. Guilherme de Castro Pena
guilherme.pena@ufsj.edu.br
Sala: DCOMP 3.11

Departamento de Ciência da Computação
Universidade Federal de São João del-Rei

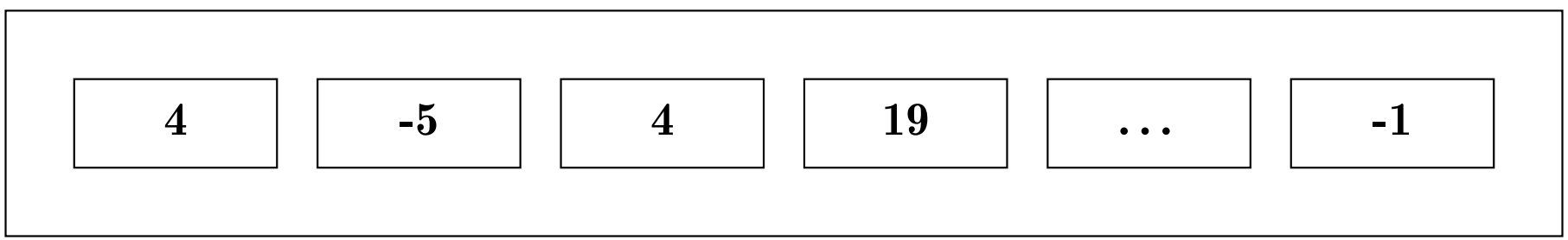
Material adaptado dos Profs. Alexandre, Elverton e Rafael

Agenda

Introdução às Listas

Visão geral:

- Em ciência da computação, uma lista é uma estrutura de dados linear utilizada para armazenar e organizar dados em um computador.
- Uma estrutura do tipo lista é uma sequência de elementos do mesmo tipo.
- Além disso, uma lista pode possuir elementos repetidos, assim como ser ordenada ou não, dependendo da aplicação.
- Como nas listas que conhecemos, a estrutura do tipo lista pode possuir N $(N \geq 0)$ elementos ou itens. Se $N = 0$, dizemos que a lista está vazia.



Introdução às Listas

Tipos de Listas

- Existem várias representações que dependem de **como os elementos são inseridos ou removidos da lista**, **o tipo de alocação usada** e **o tipo de acesso aos elementos**:

Quanto à inserção/remoção de elementos da lista:

- **Lista convencional:** pode ter elementos inseridos ou removidos de qualquer lugar dela;
- **Fila:** estrutura do tipo *FIFO (First In First Out - Primeiro a entrar é o primeiro a sair)*, os elementos só podem ser inseridos no final, e acessados ou removidos do início da lista.
- **Pilha:** estrutura do tipo *LIFO (Last In First Out) - O último a entrar é o primeiro a sair*, os elementos só podem ser inseridos, acessados ou removidos do final da lista.

Introdução às Listas

Tipos de Listas

- Existem várias representações que dependem de **como os elementos são inseridos ou removidos da lista**, **o tipo de alocação usada** e **o tipo de acesso aos elementos**:

Quanto à alocação de memória:

- **Alocação estática:** o espaço de memória é alocada no momento da compilação do programa. É necessário definir do número máximo de elementos que a lista irá possuir;
- **Alocação dinâmica:** o espaço de memória é alocado em tempo de execução. A lista cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos.

Introdução às Listas

Tipos de Listas

- Existem várias representações que dependem de **como os elementos são inseridos ou removidos da lista**, **o tipo de alocação usada** e **o tipo de acesso aos elementos**:

Quanto ao tipo de acesso aos elementos da lista:

- **Acesso sequencial:** os elementos são armazenados de forma consecutiva na memória (como em um *array* ou *vetor*). A posição de um elemento pode ser facilmente obtida a partir do início da lista;
- **Acesso encadeado:** cada elemento pode estar em uma área distinta da memória, não necessariamente consecutivas. É necessário que cada elemento da lista armazene, além da sua informação, o endereço de memória onde se encontra o próximo elemento. Para acessar um elemento, é preciso percorrer todos os seus antecessores na lista.

Introdução às Listas

Operações básicas de uma lista

- Independentemente do tipo de alocação e acesso usado na implementação de uma lista, as seguintes operações básicas são possíveis:
  - criação da lista;
  - inserção de um elemento na lista;
  - remoção de um elemento da lista;
  - busca por um elemento da lista;
  - destruição da lista;
  - além de informações com tamanho, se está cheia ou vazia.

Introdução às Listas

A operação de inserção na lista

- É o ato de guardar elementos dentro da lista, quando possível.
- Existem 3 tipos de inserção: *inserção no início*, *no final* ou *no meio* (isto é, entre dois elementos) da lista.
- A operação de inserção no meio da lista é comumente usada quando se deseja inserir um elemento de forma ordenada na lista.

- Dependendo da implementação, a operação de inserção envolve o **teste de estouro da lista**, ou seja, precisamos verificar se é possível inserir um novo elemento na lista (a lista ainda não está cheia).

Introdução às Listas

A operação de inserção na lista

- É o ato de guardar elementos dentro da lista, quando possível.
- Existem 3 tipos de inserção: *inserção no início*, *no final* ou *no meio* (isto é, entre dois elementos) da lista.
- A operação de inserção no meio da lista é comumente usada quando se deseja inserir um elemento de forma ordenada na lista.

- Dependendo da implementação, a operação de inserção envolve o **teste de estouro da lista**, ou seja, precisamos verificar se é possível inserir um novo elemento na lista (a lista ainda não está cheia).

Introdução às Listas

A operação de remoção na lista

- É o ato de excluir elementos de dentro da lista, quando possível.
- Existem 3 tipos de remoção: *remoção do início*, *do final* ou *do meio* (isto é, entre dois elementos) da lista.
- A operação de remoção do meio da lista é comumente usada quando se deseja remover um elemento específico da lista.

- No geral, a operação de remoção envolve o **teste de lista vazia**, ou seja, precisamos verificar se existem elementos dentro da lista antes de tentar removê-los.

Introdução às Listas

A operação de remoção na lista

- É o ato de excluir elementos de dentro da lista, quando possível.
- Existem 3 tipos de remoção: *remoção do início*, *do final* ou *do meio* (isto é, entre dois elementos) da lista.
- A operação de remoção do meio da lista é comumente usada quando se deseja remover um elemento específico da lista.

- No geral, a operação de remoção envolve o **teste de lista vazia**, ou seja, precisamos verificar se existem elementos dentro da lista antes de tentar removê-los.

Agenda

Lista sequencial estática

Definição

- Uma ***lista sequencial estática*** ou ***lista linear estática*** é uma lista definida utilizando alocação estática e acesso sequencial dos elementos.
- Trata-se do tipo mais simples de lista possível. Essa lista é definida utilizando um *array*, de modo que o sucessor de um elemento ocupa a posição física seguinte do mesmo.
- Além do array, essa lista utiliza um campo adicional (qtd/tam) que serve para indicar o quanto do array já está ocupado pelos elementos (dados) inseridos na lista.

Lista sequencial estática

Conforme apresentado antes:

TAD: Lista

Modelo Matemático:

- Uma lista L é uma sequência de zero ou mais itens $l_1, \ldots, l_n$, todos de um mesmo tipo, para os quais existe uma ordem relativa tal que, para $i = 1, \ldots, n-1$, l_i precede l_{i+1} e para $i = 2, \ldots, n$, l_i sucede l_{i-1}. Se $n = 0$, dizemos que a lista está vazia; caso contrário, dizemos que: n é o tamanho da lista, l_1 é o primeiro item da lista, l_n é o último e l_i é o i-ésimo item.

Operações:

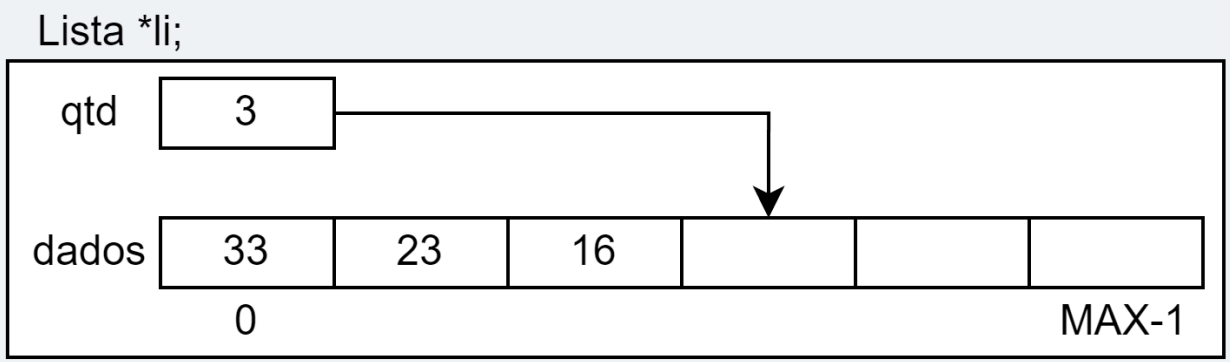
- CriaLista();
- Primeiro(L);
- Ultimo(L);
- N_esimo(L, n);
- Posicao(L,p);
- Procura(L,x);
- Recupera(L,p);
- Insere(L,x,p);
- Insere(L,x);
- Remove(L,p);
- Proximo(L,p);
- Anterior(L,p);
- Tamanho(L);
- Imprime(L);
- EstaVazia(L);

Lista sequencial estática

TAD: Lista Sequencial Estática

- Vetor estático (dados) com as seguintes operações:

- criaLista();
- destroiLista(L);
- InsereIni(L,x);
- InsereFim(L,x);
- RemoveIni(L);
- RemoveFim(L);
- Procura(L,x);
- Tamanho(L);
- Imprime(L);
- EstaVazia(L);
- EstaCheia(L);



Lista sequencial estática

Vantagens e desvantagens ao usar *array* (a se considerar na aplicação):

- Vantagens:
  - acesso rápido e direto aos elementos (índice do array);
  - tempo constante para acessar um elemento;
  - facilidade em modificar as suas informações.
- Desvantagens:
  - definição prévia do tamanho do *array* e, consequentemente, da lista;
  - dificuldade para inserir e remover um elemento entre outros dois.
- Quando usar? (situações):
  - listas pequenas;
  - inserção e remoção apenas no final da lista;
  - tamanho máximo da lista bem definido;
  - a busca é a operação mais frequente.

Lista sequencial estática

Vantagens e desvantagens ao usar *array* (a se considerar na aplicação):

- Vantagens:
  - acesso rápido e direto aos elementos (índice do array);
  - tempo constante para acessar um elemento;
  - facilidade em modificar as suas informações.
- Desvantagens:
  - definição prévia do tamanho do *array* e, consequentemente, da lista;
  - dificuldade para inserir e remover um elemento entre outros dois.
- Quando usar? (situações):
  - listas pequenas;
  - inserção e remoção apenas no final da lista;
  - tamanho máximo da lista bem definido;
  - a busca é a operação mais frequente.

Lista sequencial estática

Vantagens e desvantagens ao usar *array* (a se considerar na aplicação):

- Vantagens:
  - acesso rápido e direto aos elementos (índice do array);
  - tempo constante para acessar um elemento;
  - facilidade em modificar as suas informações.
- Desvantagens:
  - definição prévia do tamanho do *array* e, consequentemente, da lista;
  - dificuldade para inserir e remover um elemento entre outros dois.
- Quando usar? (situações):
  - listas pequenas;
  - inserção e remoção apenas no final da lista;
  - tamanho máximo da lista bem definido;
  - a busca é a operação mais frequente.

Lista sequencial estática

Vantagens e desvantagens ao usar *array* (a se considerar na aplicação):

- Vantagens:
  - acesso rápido e direto aos elementos (índice do array);
  - tempo constante para acessar um elemento;
  - facilidade em modificar as suas informações.
- Desvantagens:
  - definição prévia do tamanho do *array* e, consequentemente, da lista;
  - dificuldade para inserir e remover um elemento entre outros dois.
- Quando usar? (situações):
  - listas pequenas;
  - inserção e remoção apenas no final da lista;
  - tamanho máximo da lista bem definido;
  - a busca é a operação mais frequente.

Lista sequencial estática

Definindo o tipo Lista (sequencial estática)

- Vamos definir nossa lista, contendo um *array* de tamanho *MAX* e uma variável *qtd*;
- Além disso, o conjunto de operações que também serão implementadas no próprio arquivo *.h*;
- Vamos sempre manipular o TAD Lista utilizando um ponteiro, logo a criação da mesma, se dará por um *malloc*.

Atenção:

- ao limite MAX: *100*
- ao tipo dos dados: *int*

Pois podem ser alterados para outras implementações.

```
#ifndef LISTA_H
#define LISTA_H
//Bibliotecas..
#define MAX 100
typedef struct{
    int dados[MAX];
    int qtd;
}Lista;

#endif
```

Lista sequencial estática

Definindo o tipo Lista (sequencial estática)

- Vamos definir nossa lista, contendo um *array* de tamanho *MAX* e uma variável *qtd*;
- Além disso, o conjunto de operações que também serão implementadas no próprio arquivo *.h*;
- Vamos sempre manipular o TAD Lista utilizando um ponteiro, logo a criação da mesma, se dará por um *malloc*.

Atenção:

- ao limite MAX: *100*
- ao tipo dos dados: *int*

Pois podem ser alterados para outras implementações.

```
#ifndef LISTA_H
#define LISTA_H
//Bibliotecas..
#define MAX 100
typedef struct{
    int dados[MAX];
    int qtd;
}Lista;

#endif
```

Lista sequencial estática

Definindo o tipo Lista (sequencial estática)

- Vamos definir nossa lista, contendo um *array* de tamanho MAX e uma variável qtd;
- Além disso, o conjunto de operações que também serão implementadas no próprio arquivo $.h$;
- Vamos sempre manipular o TAD Lista utilizando um ponteiro, logo a criação da mesma, se dará por um *malloc*.

Operações do TAD:

- criaLista();
- destroiLista(L);
- InsereIni(L,x);
- InsereFim(L,x);
- RemoveIni(L);
- RemoveFim(L);
- Procura(L,x);
- Tamanho(L);
- Imprime(L);
- EstaVazia(L);
- EstaCheia(L);

Lista sequencial estática

Definindo o tipo Lista (sequencial estática)

- Vamos definir nossa lista, contendo um *array* de tamanho MAX e uma variável *qtd*;
- Além disso, o conjunto de operações que também serão implementadas no próprio arquivo *.h*;
- Vamos sempre manipular o TAD Lista utilizando um ponteiro, logo a criação da mesma, se dará por um *malloc*.

Operações do TAD:

- criaLista();
- destroiLista(L);
- InsereIni(L,x);
- InsereFim(L,x);
- RemoveIni(L);
- RemoveFim(L);
- Procura(L,x);
- Tamanho(L);
- Imprime(L);
- EstaVazia(L);
- EstaCheia(L);

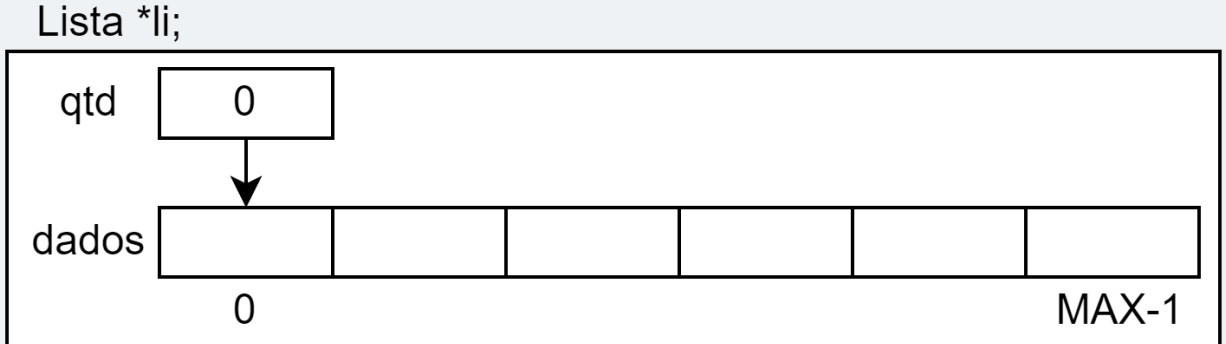
Lista sequencial estática

Criando a lista:

- Como vamos manipular por ponteiro, devemos retornar para a *Main.c* o endereço na operação que cria a lista por *malloc*.
- A quantidade de dados inicia zerada.

```c
#include <stdio.h>
#include "Lista.h"
int main(){
    Lista *L;
    L = criaLista();
    return 0;
}
```

```c
Lista* criaLista(){
    Lista *li;
    li = (Lista*) malloc (sizeof(Lista));
    if(li != NULL)
        li->qtd = 0;
    return li;
}
```

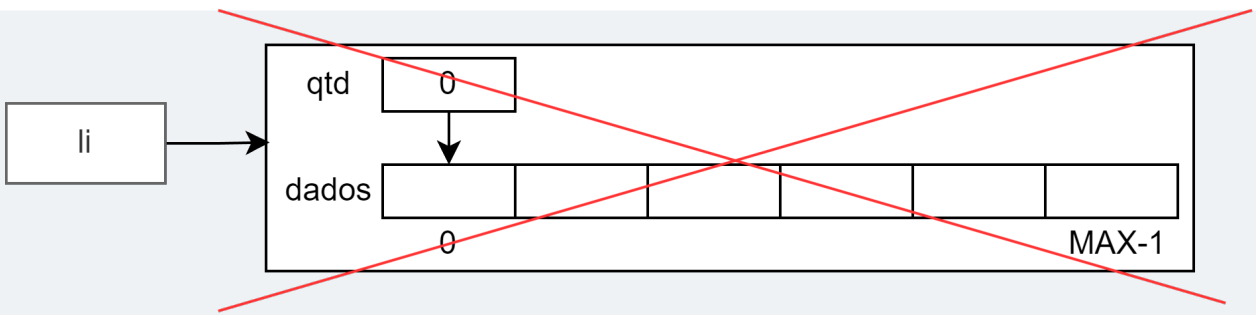


Lista sequencial estática

Destruindo a lista:

- Para destruir a lista, basta passar o endereço e usar o free() para liberar a memória alocada;

```
void destroiLista(Lista* li){
    if(li != NULL)
        free(li);
}
```



Lista sequencial estática

Informações básicas da lista

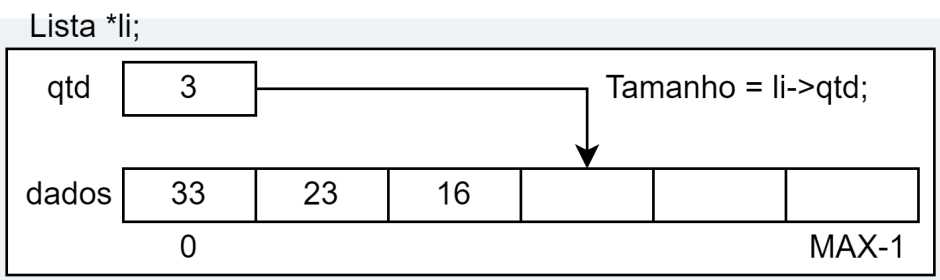
- As operações de inserção, remoção e busca são consideradas as principais operações de uma lista.
- Mas pode ser necessário ter algumas outras informações mais básicas sobre a lista, por exemplo, não podemos remover um elemento da lista se a mesma estiver vazia.
- Assim, vamos implementar funções que retornam as três principais informações sobre o status atual da lista: o seu tamanho, se ela está cheia e se ela está vazia.

Lista sequencial estática

Tamanho da lista

- A operação de retornar o tamanho pode ser necessária para o usuário tomar conhecimento da quantidade de elementos atuais da lista.

```
int tamanhoLista(Lista* li){
    if(li == NULL)
        return -1;
    return li->qtd;
}
```

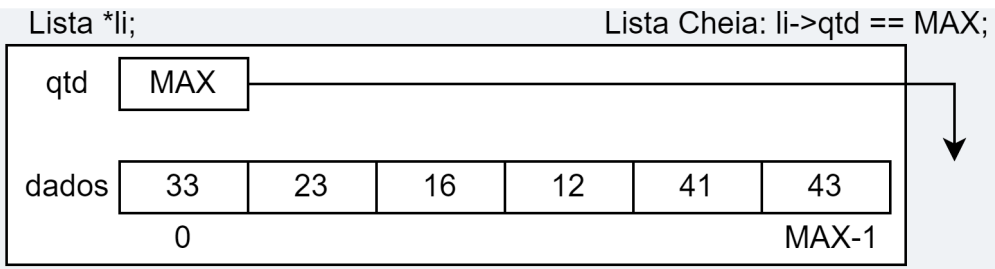


Lista sequencial estática

Lista cheia

- A operação de verificar se a lista está cheia leva em consideração o *qtd* e o *MAX*.
- Ela é importante de se analisar ao inserir um novo elemento por exemplo, para manter a estrutura consistente.

```
int listaCheia(Lista* li){
    if(li == NULL)
        return -1;
    return (li->qtd == MAX);
}
```

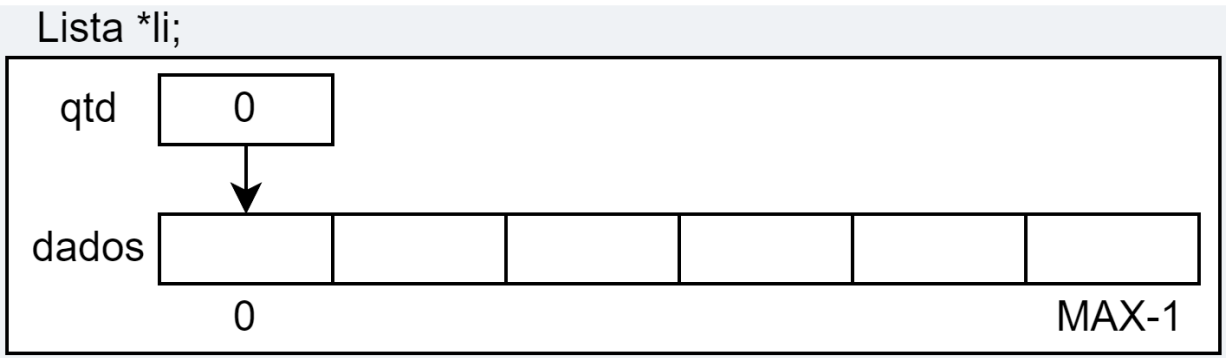


Lista sequencial estática

Lista vazia

- A operação de verificar se a lista está vazia só leva em consideração o *qtd*, caso ele seja zero.
- Ela é importante de se analisar ao tentar remover algum elemento.

```
int listaVazia(Lista* li){
    if(li == NULL)
        return -1;
    return (li->qtd == 0);
}
```



Lista sequencial estática

Operações de Inserção

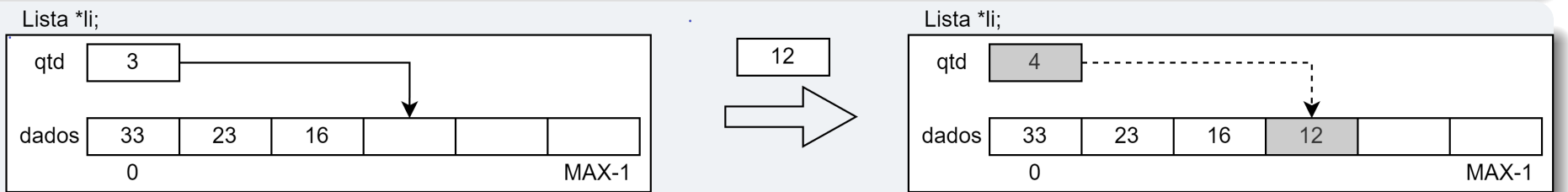
- A operação de inserção é a mais importante em uma lista.
- Existem várias formas diferentes de se fazer uma inserção na lista, no entanto, cada uma delas exige uma grande atenção aos detalhes da implementação.
- Inserção no fim, no início, no meio, ordenada, etc..
- De qualquer forma, para tais, vamos sempre considerar a lista cheia quando a quantidade de elementos (qtd) for igual ao tamanho do array (MAX).
- Para o nosso TAD, vamos demonstrar as inserções no fim e no início, as outras ficam como exercício.

Lista sequencial estática

Inserção no fim da lista

- A operação de **inserção no fim** exige a verificação da lista cheia, mas não necessita que se mude o lugar dos demais elementos da lista.

```
1  int insereFim(Lista *li, int elem){
2    if(li == NULL) return 0;
3    if(!listaCheia(li)){
4      li->dados[li->qtd] = elem;
5      li->qtd++;
6      printf("Elemento inserido com sucesso!\n"); return 1;
7    }else{
8      printf("Elemento nao inserido - Lista Cheia!\n"); return 0;
9    }
10 }
```



Lista sequencial estática

Inserção no fim da lista

- A operação de **inserção no fim** exige a verificação da lista cheia, mas não necessita que se mude o lugar dos demais elementos da lista.

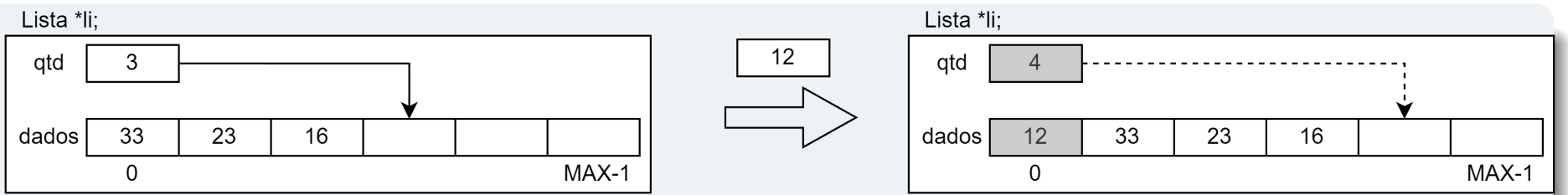
```
1 int insereFim(Lista *li, int elem){
2   if(li == NULL) return 0;
3   if(!listaCheia(li)){
4     li->dados[li->qtd] = elem;
5     li->qtd++;
6     printf("Elemento inserido com sucesso!\n"); return 1;
7   }else{
8     printf("Elemento nao inserido - Lista Cheia!\n"); return 0;
9   }
10 }
```

- Vale lembrar que o tipo do elemento é um detalhe importante a verificar, caso seja a inserção de um elemento *string*, ou algum tipo *struct* por exemplo.

Lista sequencial estática

Inserção no início da lista

- Na operação de **inserção no início**, além da verificação da lista cheia, ela necessita que se *mude o lugar dos demais elementos da lista.*
- Isso faz com que a operação seja mais trabalhosa e exige uma atenção maior à implementação;



Lista sequencial estática

Inserção no início da lista

- Na operação de **inserção no início**, além da verificação da lista cheia, ela necessita que se *mude o lugar dos demais elementos da lista.*
- Isso faz com que a operação seja mais trabalhosa e exige uma atenção maior à implementação;

```
1  int insereInicio(Lista *li, int elem){
2    if(li == NULL) return 0;
3    if(!listaCheia(li)){
4      int i;
5      for(i=li->qtd-1; i>=0; i--)
6          li->dados[i+1] = li->dados[i];
7      li->dados[0] = elem;
8      li->qtd++;
9      printf("Elemento inserido com sucesso!\n"); return 1;
10   }else{
11     printf("Elemento nao inserido - Lista Cheia!\n"); return 0;
12   }
13 }
```

Lista sequencial estática

Operações de Remoção

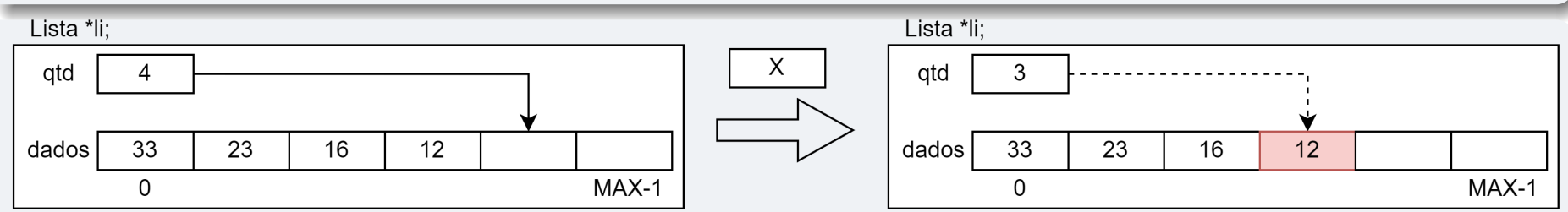
- A operação de remoção também é muito importante em uma lista.
- E similar à inserção, também existem várias formas diferentes de se fazer uma remoção, relembrando dos detalhes da implementação.
- Remoção do fim, do início, do meio, de um elemento específico, etc;
- Na remoção, devemos testar se a lista está vazia antes de seguir com a operação;
- Para o nosso TAD, vamos demonstrar as remoções do fim e do início, as outras ficam como exercício.

Lista sequencial estática

Remoção do fim da lista

- A operação de **remoção do fim** exige a verificação da lista vazia, como dito, mas não necessita que se mude o lugar dos demais elementos da lista.

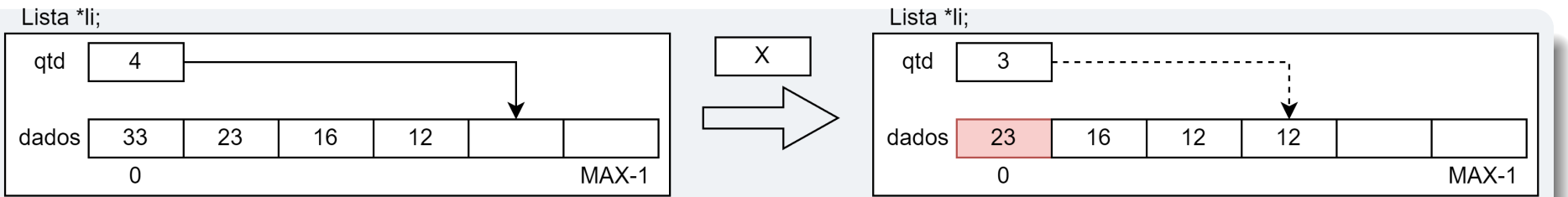
```
1 int removeFim(Lista *li){
2   if(li == NULL) return 0;
3   if(!listaVazia(li)){
4     li->qtd--;
5     printf("Elemento removido com sucesso!\n"); return 1;
6   }else{
7     printf("Nenhum elemento removido - Lista Vazia!\n"); return
          0;
8   }
9 }
```



Lista sequencial estática

Remoção do início da lista

- Na operação de **remoção do início**, além da verificação da lista vazia, ela necessita que se *mude o lugar dos demais elementos da lista.*
- Da mesma forma, se torna uma operação mais trabalhosa e exige uma atenção maior à implementação;



Lista sequencial estática

Remoção do início da lista

- Na operação de **remoção do início**, além da verificação da lista vazia, ela necessita que se *mude o lugar dos demais elementos da lista.*
- Da mesma forma, se torna uma operação mais trabalhosa e exige uma atenção maior à implementação;

```
1 int removeInicio(Lista *li){
2   if(li == NULL) return 0;
3   if(!listaVazia(li)){
4     int i;
5     for(i=0; i<li->qtd-1; i++)
6         li->dados[i] = li->dados[i+1];
7     li->qtd--;
8     printf("Elemento removido com sucesso!\n"); return 1;
9   }else{
10    printf("Nenhum elemento removido - Lista Vazia!\n"); return
          0;
11  }
12 }
```

Lista sequencial estática

Impressão dos elementos

- A operação para imprimir todos os elementos só precisa do ponteiro que contém o TAD da lista. Visto que as informações de quantidade e do dados já estão presentes.
- Um possível teste de robustez a ser feito seria a verificação de lista vazia antes de imprimir.

```
1  int imprimeLista(Lista* li){
2      if(li == NULL) return 0;
3      if(listaVazia(li)){
4          printf("Lista vazia!\n"); return 0;
5      }
6      printf("Elementos:\n");
7      int i;
8      for(i=0; i<li->qtd; i++){
9          printf("%d ", li->dados[i]);
10     }
11     printf("\n");
12     return 1;
13 }
```

Lista sequencial estática

Considerações adicionais

- Como visto, o TAD: Lista sequencial estática tem suas vantagens e desvantagens.
- O detalhe de implementação que o torna limitado é o uso do MAX como alocação estática.
- No entanto, pode haver um **TAD: Lista Sequencial Dinâmica** em que o MAX passa a ser uma característica do TAD, permitindo que se altere caso a lista fique cheia (usar um novo *malloc*).
- Veja a comparação das structs:

```c
#ifndef LISTA_H
#define LISTA_H
//Bibliotecas..
#define MAX 100
typedef struct{
    int dados[MAX];
    int qtd;
}Lista;
```

```c
#ifndef LISTADIN_H
#define LISTADIN_H
//Bibliotecas..
typedef struct{
    int *dados;
    int max;
    int qtd;
}ListaDin;
```

Exercícios - Lista Sequencial Estática

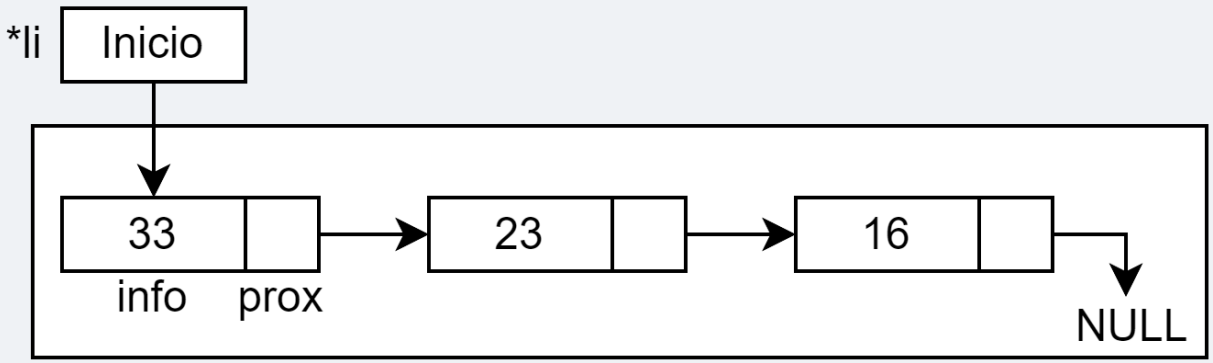
1. Reimplemente o TAD: Lista Sequencial Estática visto em aula.
2. Implemente a operação (**Procura(L, x)**) que busca e retorna o índice de um elemento X na lista, ou -1 caso contrário.
3. Crie uma nova operação de inserção, de forma que a lista se mantenha ordenada à cada nova inserção;
4. Crie uma nova operação de remoção, que remove a primeira ocorrência de um elemento específico na lista, caso exista;
5. Defina uma struct aluno (nome, matrícula e nota) e modifique o TAD: Lista Sequencial Estática para ser uma lista de alunos agora. Mantenha a operação que insere ordenado, ordenando os nomes em forma alfabética, e a remoção de um elemento específico pode usar a matrícula do aluno como chave.

Agenda

Lista Simplesmente Encadeada

Definição

- Uma **Lista Simplesmente Encadeada** ou *lista dinâmica encadeada* é definida por alocação dinâmica e acesso encadeado dos elementos.
- Os elementos são armazenados em blocos de memória denominados **nós**.
- Cada **nó** possui dois campos:
  - **info**: onde fica o dado/informação do elemento.
  - **prox**: endereço do próximo **nó** da lista.
- Além disso, a lista utiliza **um ponteiro externo (*li)** (declarado na Main.c), para indicar o primeiro elemento (***Inicio*** da lista).
- O campo **prox** do último nó armazena **NULL** para indicar o fim da lista.

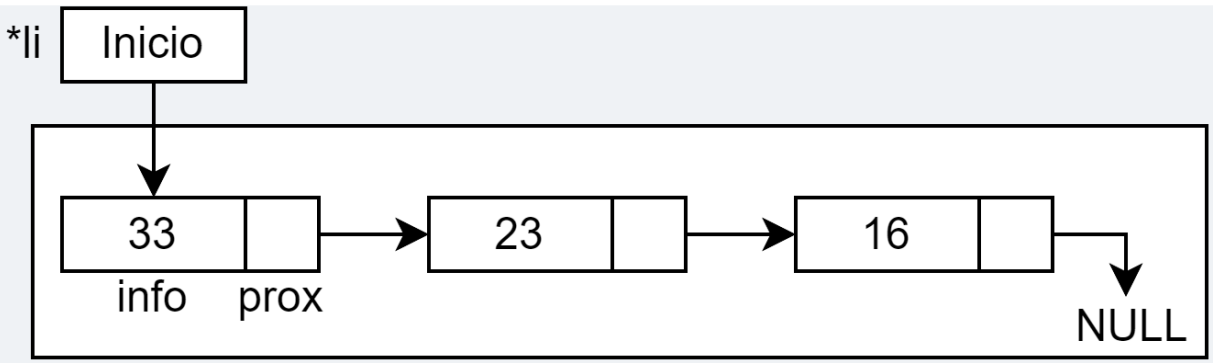


Lista Simplesmente Encadeada

TAD: Lista Simplesmente Encadeada (LSE)

- Na LSE não temos mais uma estrutura que define a lista, apenas a estrutura que define os seus elementos (**NÓs**). Quando inserimos ou removemos um elemento do início da lista, por exemplo, estamos mudando o endereço em que essa lista se inicia. Mas as operações se mantém:

- criaLista();
- destroiLista(L);
- InsereIni(L,x);
- InsereFim(L,x);
- RemoveIni(L);
- RemoveFim(L);
- Procura(L,x);
- Tamanho(L);
- Imprime(L);
- EstaVazia(L);



Lista Simplesmente Encadeada

Vantagens e desvantagens ao usar a LSE (a se considerar na aplicação):

- Vantagens:
  - melhor utilização dos recursos de memória;
  - não é preciso definir previamente o tamanho da lista;
  - não precisa movimentar os elementos nas operações de inserção e remoção.
- Desvantagens:
  - acesso indireto aos elementos;
  - necessidade de percorrer a lista para acessar determinado elemento.
- Quando usar? (situações):
  - não há necessidade de garantir um espaço mínimo para a execução da aplicação;
  - inserção e remoção em lista ordenada são as operações mais frequentes;
  - tamanho máximo da lista não é definido.

Lista Simplesmente Encadeada

Vantagens e desvantagens ao usar a LSE (a se considerar na aplicação):

- Vantagens:
  - melhor utilização dos recursos de memória;
  - não é preciso definir previamente o tamanho da lista;
  - não precisa movimentar os elementos nas operações de inserção e remoção.
- Desvantagens:
  - acesso indireto aos elementos;
  - necessidade de percorrer a lista para acessar determinado elemento.
- Quando usar? (situações):
  - não há necessidade de garantir um espaço mínimo para a execução da aplicação;
  - inserção e remoção em lista ordenada são as operações mais frequentes;
  - tamanho máximo da lista não é definido.

Lista Simplesmente Encadeada

Vantagens e desvantagens ao usar a LSE (a se considerar na aplicação):

- Vantagens:
  - melhor utilização dos recursos de memória;
  - não é preciso definir previamente o tamanho da lista;
  - não precisa movimentar os elementos nas operações de inserção e remoção.
- Desvantagens:
  - acesso indireto aos elementos;
  - necessidade de percorrer a lista para acessar determinado elemento.
- Quando usar? (situações):
  - não há necessidade de garantir um espaço mínimo para a execução da aplicação;
  - inserção e remoção em lista ordenada são as operações mais frequentes;
  - tamanho máximo da lista não é definido.

Lista Simplesmente Encadeada

Vantagens e desvantagens ao usar a LSE (a se considerar na aplicação):

- Vantagens:
  - melhor utilização dos recursos de memória;
  - não é preciso definir previamente o tamanho da lista;
  - não precisa movimentar os elementos nas operações de inserção e remoção.
- Desvantagens:
  - acesso indireto aos elementos;
  - necessidade de percorrer a lista para acessar determinado elemento.
- Quando usar? (situações):
  - não há necessidade de garantir um espaço mínimo para a execução da aplicação;
  - inserção e remoção em lista ordenada são as operações mais frequentes;
  - tamanho máximo da lista não é definido.

Lista Simplesmente Encadeada

Definindo o tipo Lista Simplesmente Encadeada (LSE)

- Vamos definir nossa lista, contendo um *struct* que representa um **nó** da lista (contendo os campos *info* e *prox*);
- Além disso, o conjunto de operações que também serão implementadas no próprio arquivo *.h*;
- Vamos sempre manipular o TAD: LSE utilizando ponteiro, logo a criação da mesma, se dará por um *malloc*.

Atenção:

- ao tipo do campo *info*: *int*
- ao novo ***typedef Lista***, que é um ***ponteiro para NO***.

```
#ifndef LSE_H
#define LSE_H
//Bibliotecas..
typedef struct NO{
    int info;
    struct NO* prox;
}NO;
typedef struct NO* Lista;

#endif
```

Lista Simplesmente Encadeada

Definindo o tipo Lista Simplesmente Encadeada (LSE)

- Vamos definir nossa lista, contendo um *struct* que representa um **nó** da lista (contendo os campos *info* e *prox*);
- Além disso, o conjunto de operações que também serão implementadas no próprio arquivo *.h*;
- Vamos sempre manipular o TAD: LSE utilizando ponteiro, logo a criação da mesma, se dará por um *malloc*.

Atenção:

- ao tipo do campo *info*: *int*
- ao novo ***typedef Lista***, que é um ***ponteiro para NO***.

```
1 #ifndef LSE_H
2 #define LSE_H
3 //Bibliotecas..
4 typedef struct NO{
5     int info;
6     struct NO* prox;
7 }NO;
8 typedef struct NO* Lista;
9
10 #endif
```

Lista Simplesmente Encadeada

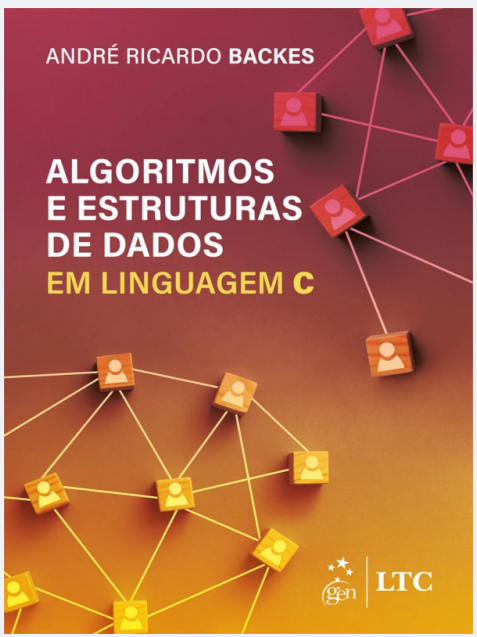
Considerações Importantes (Comparativo):

- Na **lista sequencial estática**, declara-se apenas um ponteiro para manipular a lista, porque a lista em si é uma única estrutura contendo o seu tamanho e um *array* de dados.
- Alterações realizadas (inserção ou remoção) na lista sequencial estática alteram apenas o conteúdo da estrutura, mas nunca o endereço onde ela se encontra na memória.

Lista Simplesmente Encadeada

Considerações Importantes (Comparativo):

- Nesse ponto sugiro fortemente a leitura do capítulo 5, seção **Trabalhando com ponteiro para ponteiro**, página 104, do livro do Backes.



Lista Simplesmente Encadeada

Considerações Importantes (Comparativo):

- Na **Lista Simplesmente Encadeada (LSE)** todos os elementos são ponteiros alocados dinamicamente e de forma independente.
- Não temos mais uma estrutura que define a lista, apenas a estrutura que define os seus elementos.
- Essa diferença de implementação faz com que a passagem de uma LSE para uma função tenha que ser feita utilizando um **ponteiro para ponteiro** em vez de um simples ponteiro.
- Isso ocorre, pois, ao passar um ponteiro para uma função (passagem por referência), *podemos alterar somente o conteúdo apontado por aquele ponteiro, nunca o endereço guardado dentro dele.*
- Desse modo, criamos um indicador que nunca muda sua posição na memória e que sempre aponta para o início da lista (independentemente de onde ele está).

Lista Simplesmente Encadeada

Definindo o tipo Lista Simplesmente Encadeada (LSE)

```
1 #ifndef LSE_H
2 #define LSE_H
3 //Bibliotecas..
4 typedef struct NO{
5     int info;
6     struct NO* prox;
7 }NO;
8 typedef struct NO* Lista;
9
10 #endif
```

- Por que um ponteiro no comando *typedef* para o tipo Lista?
- Porque pela linguagem C, temos que trabalhar com um *ponteiro para ponteiro* para fazer modificações no início da lista.
- É genial, pois assim, mantém-se a mesma notação utilizada pela *lista sequencial estática*:
  - Lista *li: declaração da **lista sequencial estática** (ponteiro);
  - Lista *li: declaração da **lista simplesmente encadeada** (ponteiro para ponteiro).

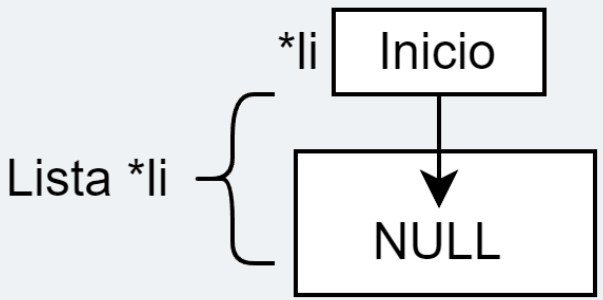
Lista Simplesmente Encadeada

Criando a lista:

- A implementação é *ponteiro para ponteiro*, mas fica suavizada pelo *typedef* *Lista*, dando a impressão de um ponteiro simples.
- **Detalhe importante:** o acesso ao conteúdo (onde ficará a informação do primeiro elemento) desse ponteiro inicial *li* é feito por ***li** (linha 5).

```c
#include <stdio.h>
#include "ListaSE.h"
int main(){
    Lista *L;
    L = criaLista();
    return 0;
}
```

```c
1 Lista* criaLista(){
2     Lista *li;
3     li = (Lista*) malloc (sizeof(Lista));
4     if(li != NULL)
5         *li = NULL;
6     return li;
7 }
```

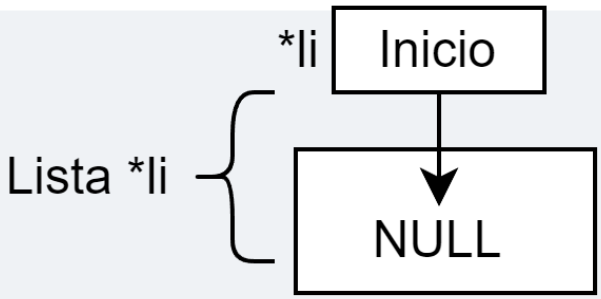


Lista Simplesmente Encadeada

Lista vazia

- Para a operação de lista vazia, basta comparar o conteúdo da região apontada por *li* e verificar se é **NULL**.
- Ela é importante de se analisar ao tentar inserir elemento no fim (primeira inserção) ou remover algum elemento.

```
int listaVazia(Lista* li){
  if(li == NULL) //Erro na alocacao inicial
    return 1;
  if(*li == NULL)
    return 1; // Lista Vazia - True
  return 0; //Lista nao vazia - False
}
```



Lista Simplesmente Encadeada

Operações de Inserção

- Existem várias formas diferentes de se fazer uma inserção na LSE, da mesma forma, cada uma exige grande atenção aos detalhes da implementação.
- Inserção no início, no fim, no meio, ordenada, etc..
- A LSE não fica cheia, só se estourar a memória do computador, na criação de novos **nós**.
- Para o nosso TAD, vamos demonstrar as inserções no início e no fim, as outras ficam como exercício.

Lista Simplesmente Encadeada

Operações complementares

- Operações de inserção e remoção trabalham com alocação e liberação de memória dos **nós**, assim usaremos as seguintes operações complementares:
- Alocar um novo nó:

```
NO* alocarNO(){
  return (NO*) malloc (sizeof(NO));
}
```

- Liberar a memória de um nó:

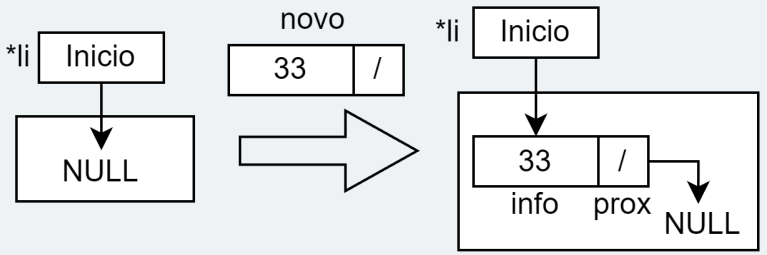
```
void liberarNO(NO* q){
  free(q);
}
```

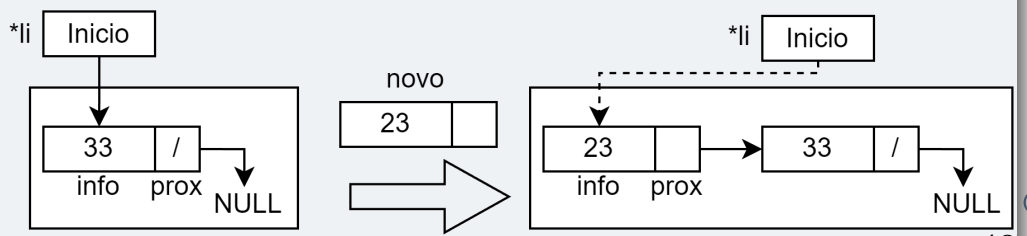
Lista Simplesmente Encadeada

Inserção no início da lista

- Na operação de **inserção no início**, não necessita que se *mude o lugar dos demais elementos da lista.*
- Temos alocar espaço para o novo elemento da lista e mudar os valores de alguns ponteiros. (***O início aponta para o novo - linha 7***).

```
1 int insereIni(Lista* li, int
      elem){
2   if(li == NULL) return 0;
3   NO* novo = alocarNO();
4   if(novo == NULL) return 0;
5   novo->info = elem;
6   novo->prox = *li;
7   *li = novo;
8   return 1;
9 }
```

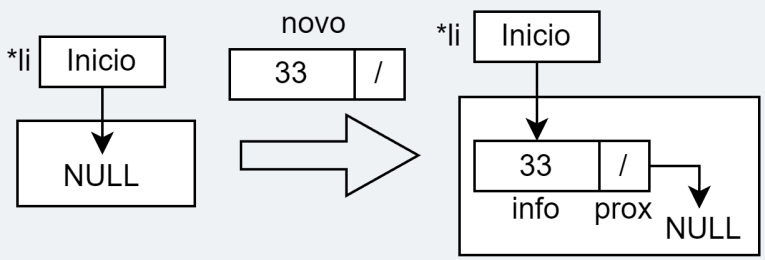


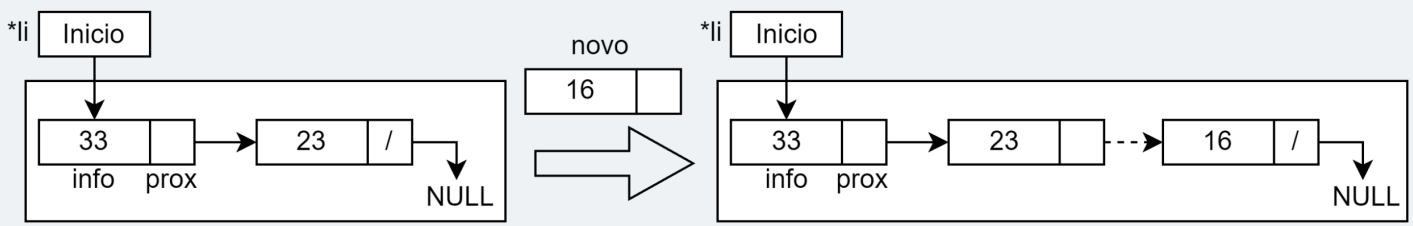


Lista Simplesmente Encadeada

Inserção no fim da lista

- A operação de **inserção no fim** é mais trabalhosa.
- Ela não necessita que se mude o lugar dos demais elementos da lista, no entanto, é preciso percorrer a lista toda para descobrir o último elemento, e assim fazer a inserção após ele.





Lista Simplesmente Encadeada

Inserção no fim da lista

- A ideia de percorrer, é um usar um **nó auxiliar**, que parte do *início* da lista, e que acha o último elemento pelo *prox*.

```
1  int insereFim(Lista* li, int elem){
2    if(li == NULL) return 0;
3    NO* novo = alocarNO();
4    if(novo == NULL) return 0;
5    novo->info = elem;
6    novo->prox = NULL;
7    if(listaVazia(li)){
8      *li = novo;
9    }else{
10     NO* aux = *li;
11     while(aux->prox != NULL)
12       aux = aux->prox;
13     aux->prox = novo;
14   }
15   return 1;
16 }
```

Lista Simplesmente Encadeada

Operações de Remoção

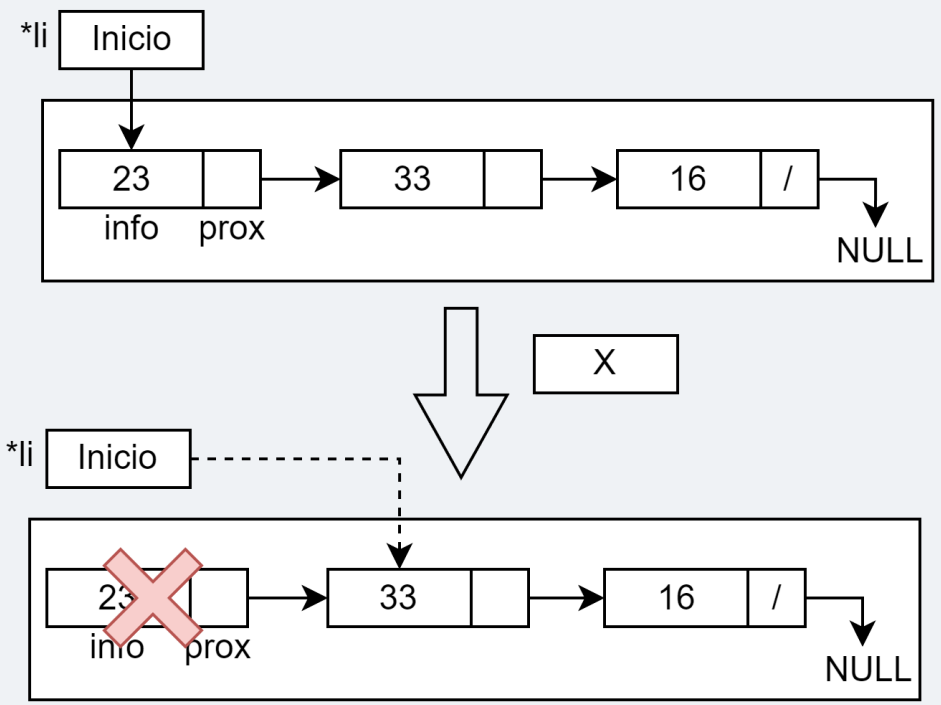
- Similar à inserção, também pode-se fazer uma remoção do fim, do início, do meio, de um elemento específico, relembrando dos detalhes da implementação.
- Na remoção, devemos testar se a lista está vazia antes de seguir com a operação;
- Para o nosso TAD, vamos demonstrar as remoções do início e do fim, as outras ficam como exercício.

Lista Simplesmente Encadeada

Remoção do início da lista

- A operação de **remoção do início** verifica a lista vazia, como dito, mas não necessita que se mude o lugar dos demais elementos da lista.
- Ela é mais simples e basta mudar alguns ponteiros a partir do início e liberar a memória do nó removido.

```
1 int removeIni(Lista* li){
2   if(li == NULL) return 0;
3   if(listaVazia(li)) return 0;
4   NO* aux = *li;
5   *li = aux->prox;
6   liberarNO(aux);
7   return 1;
8 }
```

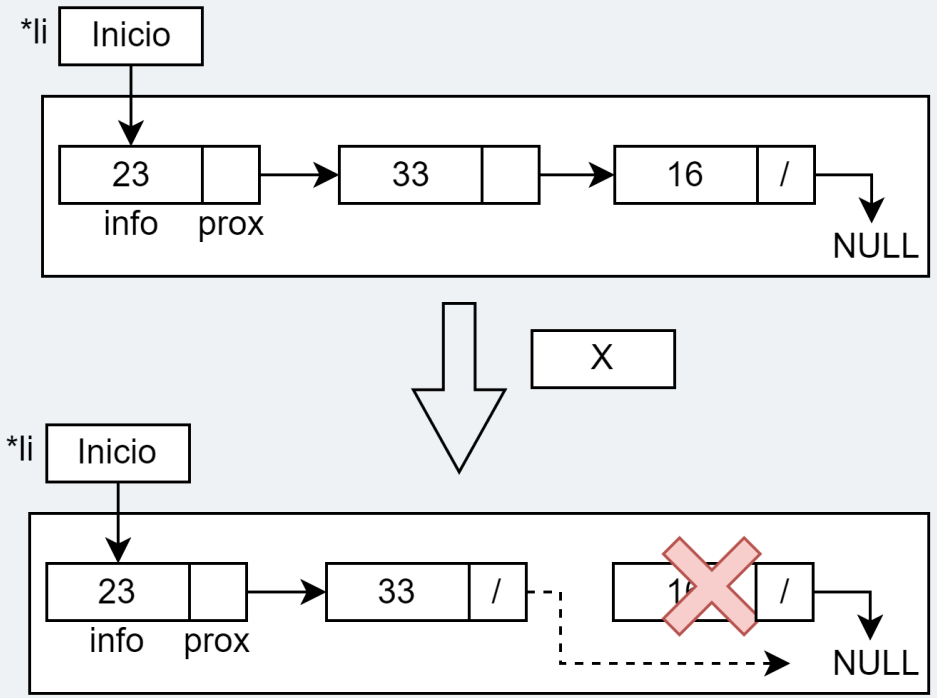


Lista Simplesmente Encadeada

Remoção do fim da lista

- Na **remoção do fim**, também verifica-se a lista vazia.
- Ela é mais trabalhosa pois deve percorrer para encontrar o último elemento e exige uma atenção maior à implementação;
- Além disso, deve guardar a informação do **nó anterior**.

```
1  int removeFim(Lista* li){
2    if(li == NULL) return 0;
3    if(listaVazia(li)) return 0;
4    NO *ant, *aux = *li;
5    while(aux->prox != NULL){
6      ant = aux;//Anterior
7      aux = aux->prox;
8    }
9    if(aux == *li)//primeiro?
10     *li = aux->prox;
11   else
12     ant->prox = aux->prox;
13   liberarNO(aux);
14   return 1;
15 }
```

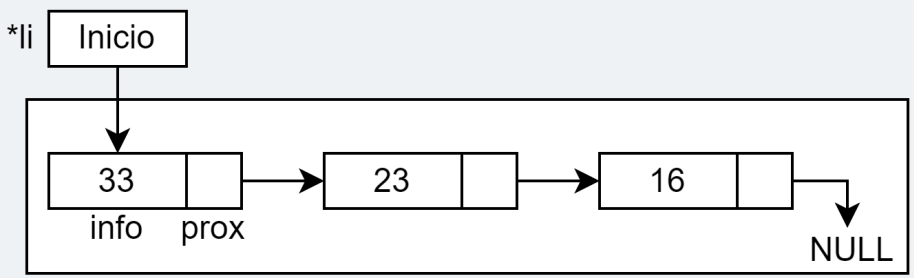


Lista Simplesmente Encadeada

Imprimindo a lista:

- Para imprimir cada elemento, é preciso percorrer toda a lista a partir do início até chegar ao fim, usando o ponteiro **aux**.
- Nesse caso, imprime-se a informação do campo ***info*** de cada **nó**;

```
1  void imprimeLista(Lista* li){
2    if(li == NULL) return;
3    if(listaVazia(li)){
4      printf("Lista vazia!\n");
5      return;
6    }
7    printf("Elementos:\n");
8    NO* aux = *li;
9    while(aux != NULL){
10     printf("%d ", aux->info);
11     aux = aux->prox;
12   }
13   printf("\n");
14 }
```

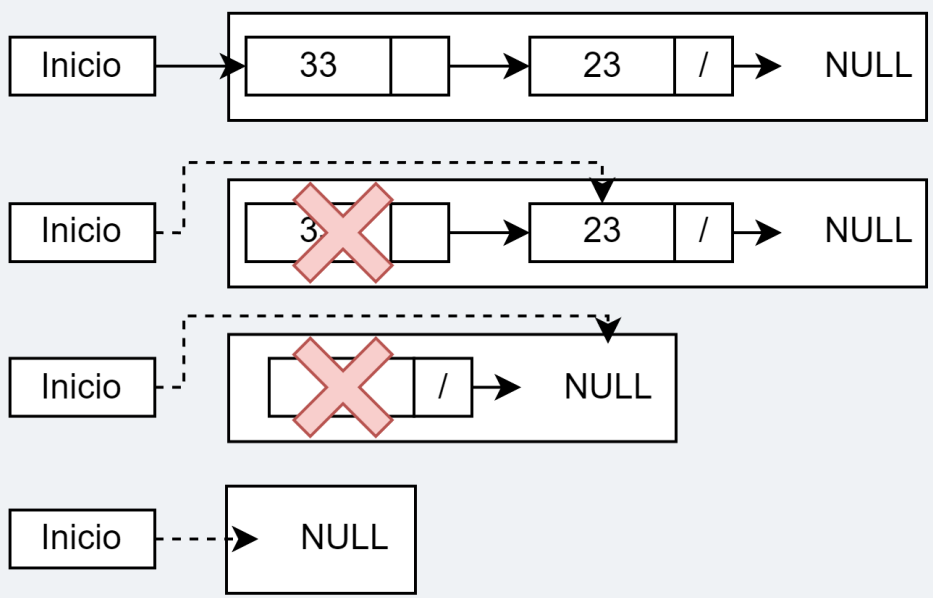


Lista Simplesmente Encadeada

Destruindo a lista:

- Para liberar uma lista que utilize alocação dinâmica e seja encadeada é preciso percorrer toda a lista, liberando a memória alocada para cada **nó** inserido.

```
1  void destroiLista(Lista* li){
2    if(li != NULL){
3      NO* aux;
4      while((*li) != NULL){
5        aux = *li;
6        *li = (*li)->prox;
7        liberarNO(aux);
8      }
9      free(li);
10   }
11 }
```



Lista Simplesmente Encadeada

Considerações adicionais

- A Lista Simplesmente Encadeada é uma estrutura muito poderosa por ser dinâmica e mais eficiente na inserção e remoção de elementos, levando em consideração que *não há movimentação de elementos na memória*. **Apenas reajuste de ponteiros!**
- Como visto, ela tem seus vantagens e desvantagens, mas os pontos mais importantes seriam:
  - A atenção aos detalhes da implementação (atribuições corretas quando trabalhar com *ponteiros*, *ponteiros para ponteiros*);
  - A atenção ao campo **info**, de um nó, quando trabalhar com um tipo struct por exemplo (Aluno).

```
1 typedef struct NO{
2     int info;
3     struct NO* prox;
4 }NO;
5 typedef struct NO* Lista;
```

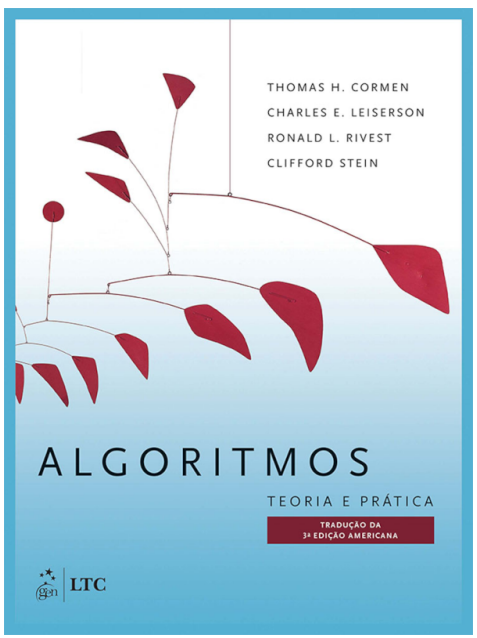
Exercícios - Lista Simplesmente Encadeada

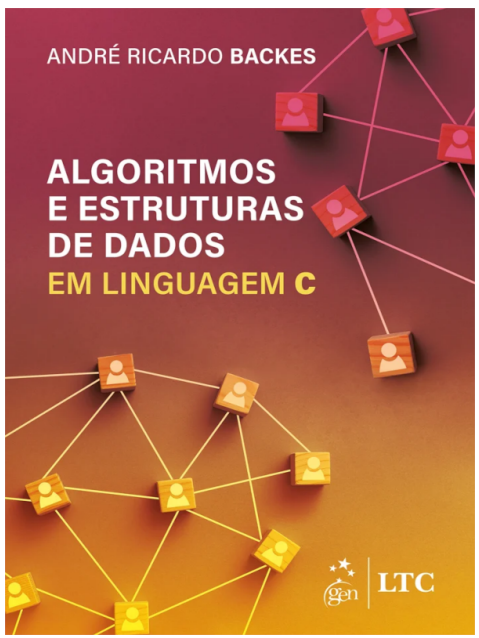
1. Reimplemente o TAD: Lista Simplesmente Encadeada visto em aula. Faça o desenho rastreio (no papel) da inserção no fim de 3 elementos a partir de uma lista vazia, depois impressão, e depois liberação da lista.
2. Ficou faltando as operações (**Tamanho(L)**) e (**Procura(L, x)**) para o TAD, implemente-as e teste.
3. Crie uma nova operação de inserção, de forma que a lista se mantenha ordenada à cada nova inserção;
4. Crie uma nova operação de remoção, que remove a primeira ocorrência de um elemento específico na lista, caso exista;
5. Defina uma struct aluno (nome, matrícula e nota) e crie um TAD: Lista Simplesmente Encadeada para ser uma lista de alunos agora. Mantenha a operação que insere ordenado, ordenando os nomes em forma alfabética, e a remoção de um elemento específico pode usar a matrícula do aluno como chave.

Bibliografias

Bibliografia Básica

1. N. ZIVIANI, Projeto de Algoritmos com Implementações em Pascal e C, Editora Thomson, 2004.
2. T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, C. STEIN, Algoritmos, Teoria e Prática, Campus, 2002.
3. BACKES, André Ricardo. Algoritmos e estruturas de dados em linguagem C. Rio de Janeiro, LTC, 2023, ISBN 9788521638315.





Bibliografias

Bibliografia Complementar

1. ZIVIANI, Nivio. Projeto de algoritmos: com implementações em Java e C++. São Paulo: Thomson, 2007.
2. D. E. KNUTH, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, 1997.
3. D. E. KNUTH, The Art of Computer Programming, Volume 3: Searching and Sorting, Addison-Wesley, 1997.
4. N. WIRTH, Algoritmos e Estruturas de Dados, LTC, 1989.
5. J. L. SZWARCFITER, L. MARKENZON, Estruturas de Dados e Seus Algoritmos, LTC, 2002.
6. VELOSO, C. Santos, O, Azeredo, A. Furtado, Estruturas de Dados, Campus, 1983.