Listas, Filas e Pilhas

Rômulo César Silva

Unioeste

Adaptação: Renato Bobsin Machado

Abril de 2022

unioeste
Universidade Estadual do Oeste do Paraná

Sumário

Lista

Uma **Lista** é uma coleção de itens de dados do mesmo tipo.

Exemplos:

- lista de compras do supermercado
- lista de alunos da turma
- lista de clientes da loja
- lista de peças do estoque

Lista

Características

- Os itens estão dispostos sequencialmente $x_1, x_2, x_3, ..., x_n$, tal que x_1 é o primeiro item e x_n é o último item da lista
- Principais Operações:
  - inserção de um item na lista
  - remoção de um item da lista
  - busca de um item na lista

Lista

Aplicações na Computação

- **Sistemas Operacionais**: gerenciamento de fragmentação da memória principal
- **Compiladores**: alternativa para implementação da tabela de símbolos
- **Computação Gráfica**: listas de pontos que formam um polígono
- **IA**: computação simbólica usada nas linguagens LISP e Prolog
- **... e muitas outras**

TAD Lista

Principais operações

- `inserir(Lista l, Item x)`: insere o item x na lista l
- `remover(Lista l, Item x)`: remove o item x da lista l
- `busca(Lista l, Item x)`: busca o item x na lista l

TAD Lista

Outras operações

- `criar`: cria uma lista vazia
- `ordenar(Lista l)`: colocar os itens em ordem crescente ou decrescente
- `inserir(Lista l, Item x, Posicao p)`: inserir o Item x na posição p da lista l
- `remover(Lista l, Posicao p)`: remove o item que se encontra na posição p da lista l
- `concatenar(Lista l1, Lista l2 )`: concatena as listas l1 e l2

TAD Lista

Implementação

A estrutura de dados **lista** pode ser implementada de muitas maneiras. A escolha de qual maneira implementar depende da aplicação.

TAD Lista

Implementação

Dependendo das restrições que se impõe ao modo de funcionamento das operações de inserção e remoção, a lista recebe um nome específico:

- **Fila**: a inserção é sempre no final (cauda) da lista e a remoção é sempre do início (cabeça) da lista.
- **Pilha**: tanto a inserção quanto a remoção é feita sempre do início (cabeça) da lista.

TAD Lista

Implementação

As 2 maneiras mais comuns de se implementar lista são:

- lista estática: usando vetores e alocação estática sequencial
- lista dinâmica: usando ponteiros e alocação dinâmica não sequencial

Implementação de Lista Estática

- uso de um vetor alocado estaticamente para armazenar sequencialmente os elementos
- a lista pode ser percorrida em ambas direções
- a inclusão de um novo elemento pode ser feita após o último elemento, sem necessidade de percorrer a lista
- a inclusão de um novo elemento no meio da lista requer o deslocamento de todos os elementos localizados após o ponto de inserção
- a retirada de um elemento requer também o deslocamento dos itens subsequentes para preencher o espaço deixado vazio pelo item removido

Implementação de Lista Estática

1ª **implementação:**

```
#define TAM_MAX 1000 // número máximo de itens na lista

typedef int TipoItem;

// estrutura para lista estática
typedef struct {
    TipoItem item[TAM_MAX];
    int pos_livre;
} Lista;
```

Implementação de Lista Estática

1ª implementação:

- `TipoItem` representa o tipo dos elementos da lista. Nesta implementação, é um `int`
- a variável `pos_livre` marca a última posição livre do vetor
- a lista está vazia se `pos_livre` é zero
- a lista está cheia se `pos_livre` é `TAM_MAX`
- a inserção é feita sempre na última posição livre
- a retirada de um elemento `x` que esteja na posição `i` ocasiona o deslocamento dos elementos das posições `i+1`, `i+2`, ..., para a posição imediatamente anterior.

Implementação de Lista Estática

1ª implementação - protótipos das funções (.h):

```
//Cria uma lista vazia
Lista* cria_lista_vazia();

//Testa se uma lista é vazia
int vazia(Lista * l);

//Insere um item na lista
void insere(Lista* l, TipoItem x);

//Retira o item da lista
void retira(Lista* l, TipoItem x);

//Imprime os itens da lista
void imprime(Lista* l);
```

Implementação de Lista Estática

1ª implementação - (.c):

```
//Cria uma lista vazia
//Pré-condição: nenhuma
//Pós-condição: lista criada
Lista* cria_lista_vazia(){
    Lista * nova_lista = (Lista*) malloc(sizeof(Lista));
    nova_lista->pos_livre = 0;
    return nova_lista;
}
//Testa se uma lista é vazia
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: retorna 1 se a lista é vazia ou
//                      0 caso contrário
int vazia(Lista * l){
   return (l->pos_livre == 0);
}
```

Implementação de Lista Estática

1ª implementação - (.c):

```
//Insere um item na lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: item é inserido na lista
void insere(Lista* l, TipoItem x){
   if(l->pos_livre >= TAM_MAX)
     printf("Lista cheia");
   else
      l->item[l->pos_livre++] = x;
}
```

Implementação de Lista Estática

1ª implementação - (.c):

```
//Retira o item da lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: item é removido da lista
void retira(Lista* l, TipoItem x){
   if(vazia(l))
     printf("Lista vazia: sem elementos para retirar");
   else {
     int pos = 0;
     while(pos < l->pos_livre && x != l->item[pos])
       pos++;
     if(pos == l->pos_livre)
        printf("Elemento não encontrado");
     else { int i;
        for(i = pos; i < l->pos_livre; i++)
           l->item[i] = l->item[i+1];
        }
        l->pos_livre--;
    }
}
```

Implementação de Lista Estática

2ª **implementação:**

```
#define TAM_MAX 1000 // número máximo de itens na lista

typedef int TipoItem;

// estrutura para lista estática
typedef struct {
    TipoItem item[TAM_MAX];
    int primeiro, ultimo;
    int pos; // usado para implementar iterador
} Lista;
```

Implementação de Lista Estática

2ª implementação:

- `TipoItem` representa o tipo dos elementos da lista. Nesta implementação, é um `int`
- a variável `primeiro` marca a posição do primeiro elemento
- a variável `ultimo` marca a primeira posição livre após o último elemento
- a variável `pos` é usada para propósitos de navegação na lista.
- a retirada de um elemento `x` que esteja na posição `i` ocasiona o deslocamento dos elementos das posições `i+1`, `i+2`, ..., para a posição imediatamente anterior.

Implementação de Lista Estática

2ª implementação:

```
Lista* cria_lista_vazia(){
    Lista * nova_lista = (Lista*) malloc(sizeof(Lista));
    nova_lista->primeiro = 0;
    nova_lista->ultimo = 0;
    nova_lista->pos = -1;
    return nova_lista;
}

int vazia(Lista * l){
   return (l->primeiro == l->ultimo);
}
```

Implementação de Lista Estática

2ª implementação:

```
//Retorna o primeiro item da lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: retorna o primeiro item
TipoItem primeiro(Lista* l) {
   l->pos = -1;
   return proximo(l);
}
//Retorna o proximo item da lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: retorna próximo item
TipoItem proximo(Lista* l) {
 l->pos++;
 if(l->pos >= l->ultimo)
    return -1; // acabou itens
 else
    return l->item[l->pos];
}
```

Implementação de Lista Estática

2ª implementação:

```
//Imprime os itens da lista
//Pré-condição: ponteiro não nulo para estrutura lista
//Pós-condição: nenhuma
void imprime(Lista* l){
  if(vazia(l))
    printf("Lista vazia");
  else {
    int i = 0;
    printf("[");
    for(i = l->primeiro; i < l->ultimo; i++)
      printf("%d ", l->item[i]);
    printf("]\n");
    }
}
```

Implementação de Lista Estática

2ª **implementação:**

```
//imprime usando as funções iteradoras
// para percorrer a lista
void imprime2(Lista* l){
  if(vazia(l))
    printf("Lista vazia");
  else {
    TipoItem x = primeiro(l);
    printf("[");
    do
      printf("%d ", x);
    while((x = proximo(l))!= -1);
    printf("]\n");
    }
}
```

Implementação de Lista Dinâmica

- uso de um ponteiro para encadear um elemento com o seguinte
- a alocação de memória é feita à medida que se necessita inserir um novo elemento. Assim, os elementos não ocupam posições contíguas na memória
- a inclusão/retirada de um novo elemento pode ser feita sem necessidade de deslocamento dos demais elementos

Lista Dinâmica

Variações de Implementação

- Lista Dinâmica Simples
- Lista Duplamente Encadeada
- Lista Duplamente Encadeada Circular
- Lista Encadeada com Cabeça e Cauda

Lista Encadeada Simples

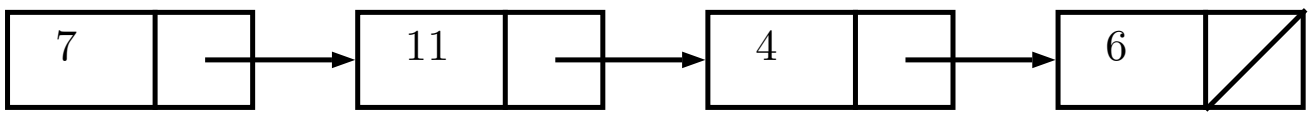
Estrutura:

```
typedef int TipoItem; // tipo dos elementos na lista

typedef struct no { //estrutura para lista encadeada
    TipoItem info;
    struct no * prox; // ponteiro para próximo elemento
} Lista;
```

Lista Encadeada Simples

Inserção na cabeça

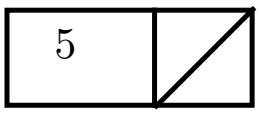
Inserir o elemento 5 na lista abaixo:

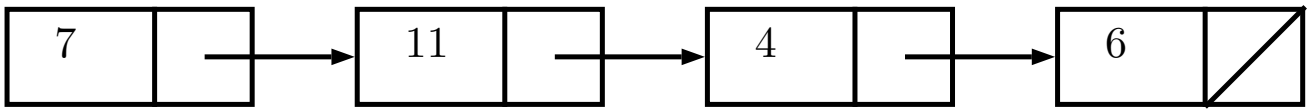


Lista Encadeada Simples

Inserção na cabeça

1^o passo: criar o nó para conter o elemento.

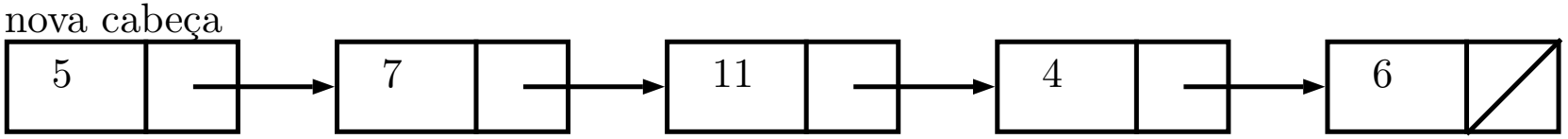




Lista Encadeada Simples

Inserção na cabeça

2º passo: apontar o próximo do nó criado para a cabeça e torná-lo a nova cabeça.



Lista Encadeada Simples

Inserção na cabeça

```
Lista* insere(Lista* l, TipoItem info) {
  Lista* aux = (Lista*) malloc(sizeof(Lista));
  aux->info = info;
  aux->prox = l; //encadeia com a cabeça da lista
  return aux;// retorna a nova cabeça
}
```

Lista Encadeada Simples

Remoção de um elemento

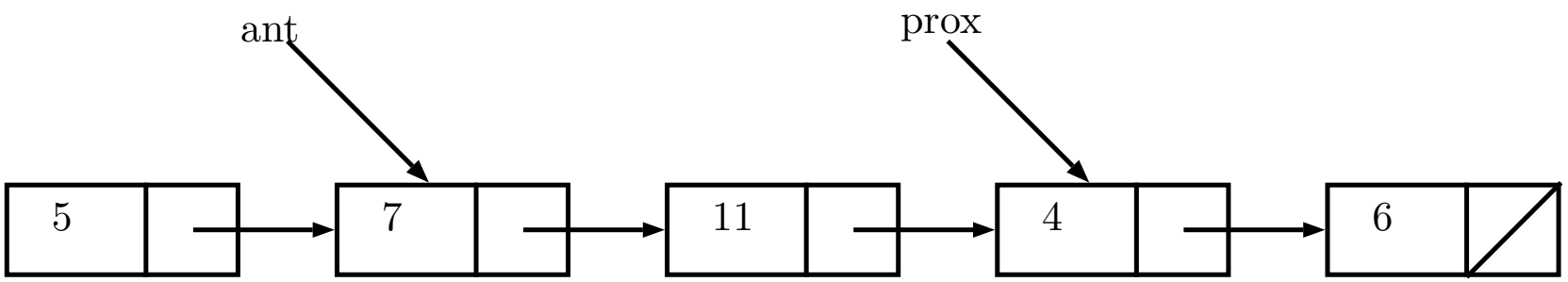
Há 2 casos:

- remoção no meio da lista
- remoção da cabeça

Lista Encadeada Simples

Remoção no meio

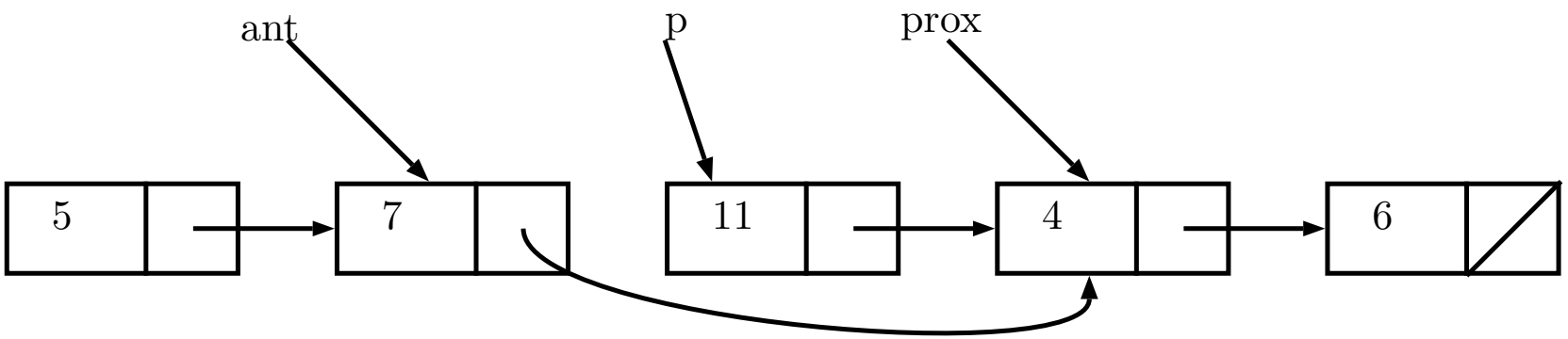
Remover o elemento 11 na lista abaixo:



Lista Encadeada Simples

Remoção no meio:
1^o passo: encadeiar o anterior com o próximo do elemento a ser removido

```
ant->prox = p->prox;
```

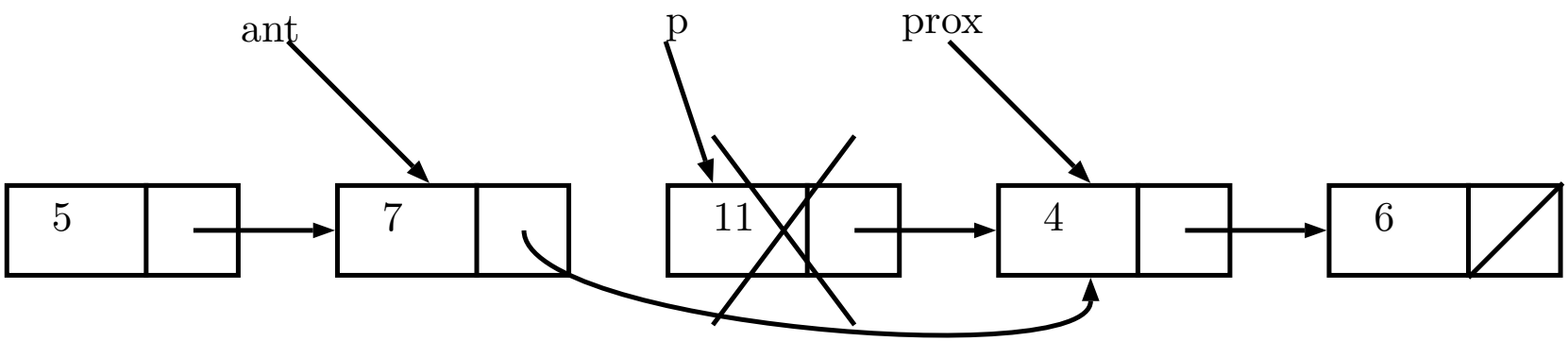


Lista Encadeada Simples

Remoção no meio

2º passo: liberar o nó do elemento removido

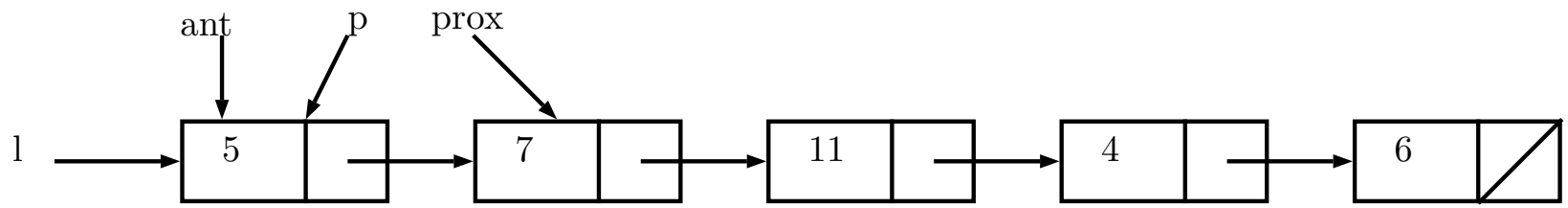
```
free(p);
```



Lista Encadeada Simples

Remoção na cabeça

Remover o elemento 5 da lista abaixo:



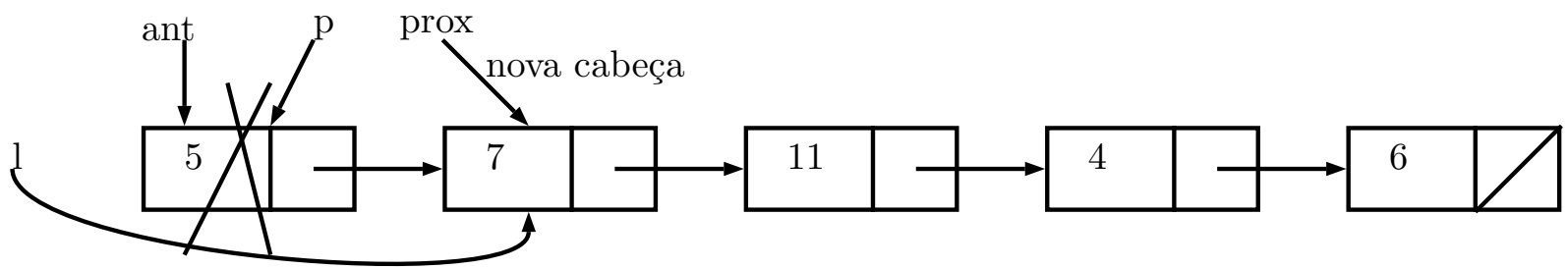
Lista Encadeada Simples

Remoção na cabeça

1º passo: a cabeça aponta para o próximo elemento

2º passo: o nó da cabeça antiga é liberado

```
l = l->prox;
free(p);
```

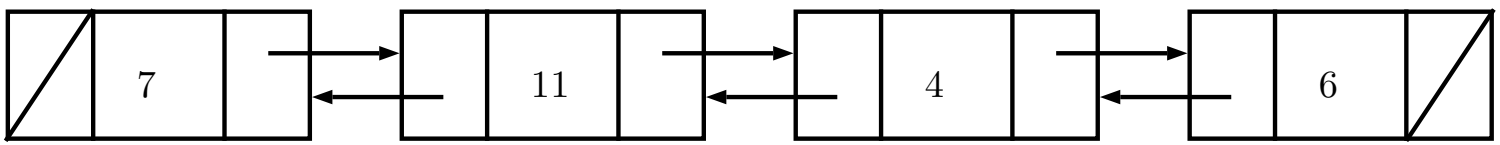


Lista Encadeada Simples

Remoção: juntando os 2 casos no código

```
Lista * retira(Lista* l, TipoItem info) {
  Lista * ant = l; // elemento anterior
  Lista * p = l; // usado para encontrar o elemento
                 // a ser retirado
  while(p!= NULL && p->info != info) { // localiza o elemento
     ant = p;
     p = p->prox;
  }
  if(p != NULL) { // elemento encontrado
    if(p == l) // remoção na cabeça
       l = l->prox; // atualiza a cabeça
    else // remoção no meio
       ant->prox = p->prox;
    free(p); // libera o nó do elemento removido
  }
  else printf("Elemento não encontrado");
  return l;
}
```

Lista Duplamente Encadeada

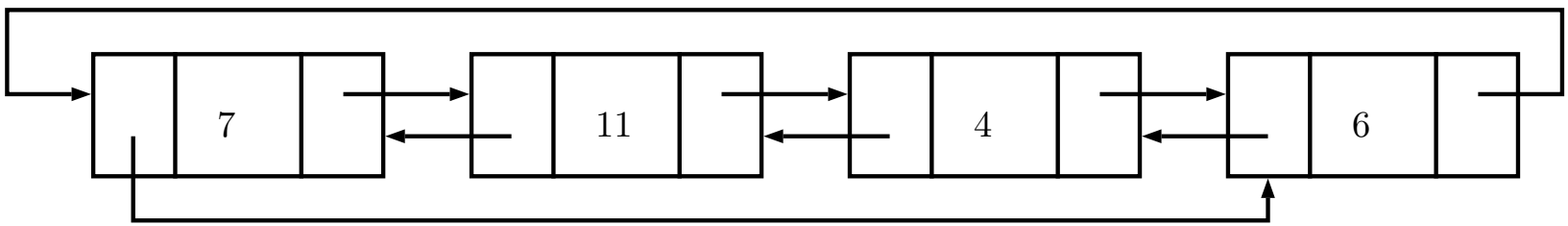


Lista Duplamente Encadeada

Estrutura:

```
typedef int TipoItem; // tipo dos elementos na lista

typedef struct no { //estrutura para lista duplamente encadeada
    TipoItem info;
    struct no * ant; // ponteiro para o elemento anterior
    struct no * prox; // ponteiro para próximo elemento
} Lista;
```

Lista Duplamente Encadeada Circular



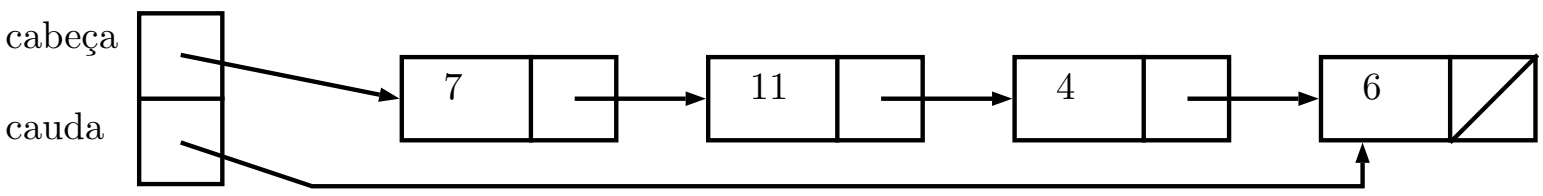
Lista Duplamente Encadeada Circular

Estrutura:

- é a mesma da Lista Encadeada Circular
- as operações de inserção e remoção devem ser implementadas de modo a garantir a circularidade (o próximo do último é o primeiro, e o anterior do primeiro é o último)

```
typedef int TipoItem; // tipo dos elementos na lista

typedef struct no {  //estrutura para
                     //lista duplamente encadeada circular
    TipoItem info;
    struct no * ant; // ponteiro para o elemento anterior
    struct no * prox; // ponteiro para próximo elemento
} Lista;
```

Lista Encadeada com Cabeça e Cauda



Lista Encadeada com Cabeça e Cauda

Estrutura:

```
typedef int TipoItem;

struct no {  // estrutura de nó para lista encadeada
    TipoItem info;
    struct no * prox;
};

typedef struct { // estrutura para lista encadeada com cabeça e cauda
   struct no* cabeca;
   struct no* cauda;
} Lista;
```

Lista Dinâmica

É importante o aluno dominar a implementação das diferentes variantes!

Por isso:

- implemente primeiro a variante mais simples (Lista Dinâmica Simples) para somente depois implementar as mais complexas.
- ao projetar os algoritmos de inserção e retirada de elementos, procure identificar os *casos especiais* que existem em cada variante.
- teste cada caso separadamente

Lista Encadeada em Arquivo

- uso de arquivo no lugar da memória principal para armazenar os nós da lista
- se mantêm na memória principal apenas os registros que se está manipulando no momento, devendo ser escritos em seguida no arquivo
- ao invés de usar funções de alocação de memória (`malloc`), escreve-se um registro no final arquivo ou em uma posição não ocupada.
- o encadeamento, ao invés de usar ponteiros, é feito usando as posições dos registros no arquivo

Lista Encadeada em Arquivo

- pode-se manter uma lista de nós livres que aumenta ou diminui conforme são retirados ou inseridos novos elementos
- é útil ter um cabeçalho no início do arquivo em que se guardam os endereços das cabeças da lista de nós ativos e da lista de nós livres, além da próxima posição após o final do arquivo.
- o mais adequado é usar arquivo-binário e não arquivo-texto para armazenar a lista para dificultar alterações indesejadas diretamente sobre o arquivo.

Lista Encadeada em Arquivo

funções para manipular arquivos em C: **abertura**

```
FILE* fopen(char* nomeArq, char* modo)
```

- `nomeArq` deve ser um nome válido no sistema operacional
- `modo` indica quais serão as operações válidas. `w+b` para leitura e gravação em modo binário
- A função retorna um ponteiro para arquivo (*handle*)
- no caso de erro, a função retorna um ponteiro nulo (`NULL`)

Lista Encadeada em Arquivo

funções para manipular arquivos em C: **posicionamento**

```
int fseek(FILE* fp, long nro_bytes,int origem)
```

- a função move a posição corrente `nro_bytes` a partir de um ponto especificado (`origem`), que pode ser:
  - `SEEK_SET`: a partir do início do arquivo
  - `SEEK_CUR`: a partir da posição corrente
  - `SEEK_END`: a partir do fim do arquivo

Lista Encadeada em Arquivo

funções para manipular arquivos em C: **leitura**

```
unsigned fread(void* buffer, int nro_bytes, int
count, FILE* fp)
```

- `buffer`: região de memória onde os dados serão armazenados
- `nro_bytes`: número de bytes lidos por unidade
- `count`: quantidade de unidades a serem lidas
- a função retorna o número de unidades efetivamente lidas

Lista Encadeada em Arquivo

funções para manipular arquivos em C: **escrita**

```
unsigned fwrite(void* buffer, int nro_bytes, int
count, FILE* fp)
```

- parâmetros similares aos de `fread`, porém para escrita

Lista Encadeada em Arquivo

Estrutura:

```
typedef int TipoItem;

// cabecalho do arquivo
typedef struct {
   int pos_cabeca; //posição do início da lista
   int pos_topo; // 1a posição não usada no fim do arquivo
   int pos_livre; // posição do início da lista de nós livres
} cabecalho;

// estrutura de no para lista encadeada
typedef struct {
    TipoItem info;
    int prox;
} no;
```

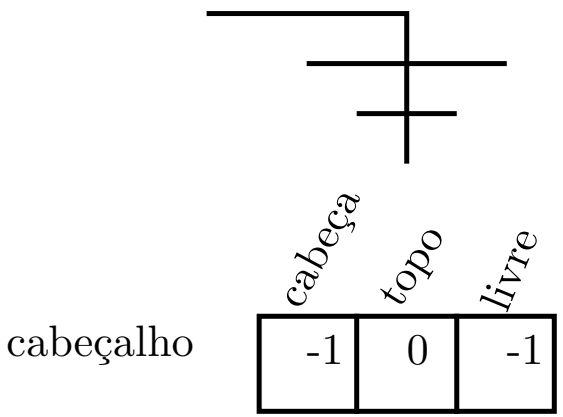
Lista Encadeada em Arquivo

Função para criar uma lista vazia em um arquivo:

```
//Cria uma lista nova em arquivo
//Pré-condição: arquivo aberto para leitura/escrita
//Pós-condição: arquivo é inicializado com uma lista vazia

void cria_lista_vazia(FILE* arq){
     cabecalho * cab = (cabecalho*) malloc(sizeof(cabecalho));
     cab->pos_cabeca = -1;
     cab->pos_topo = 0;
     cab->pos_livre = -1;
     escreve_cabecalho(arq,cab);
     free(cab);
}
```

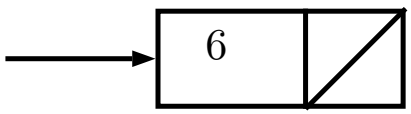
Lista Encadeada em Arquivo

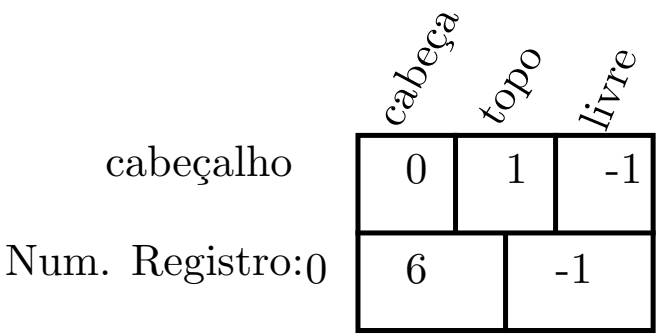
Inserir 6, 13, 11, 37 e 22 nesta ordem em uma lista inicialmente vazia:



Lista Encadeada em Arquivo

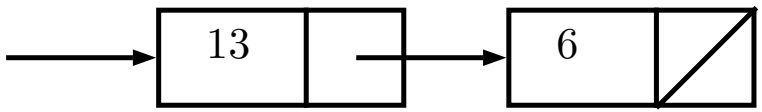
Inserção de 6:





Lista Encadeada em Arquivo

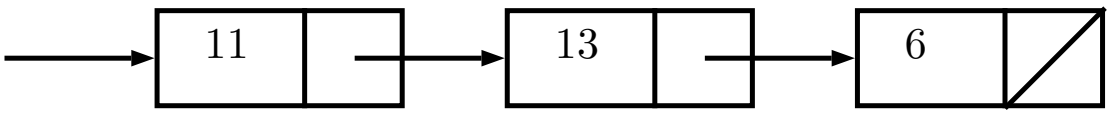
Inserção de 13:



	cabeça	topo	livre
cabeçalho	1	2	-1
Num. Registro: 0	6	-1	
1	13	0	

Lista Encadeada em Arquivo

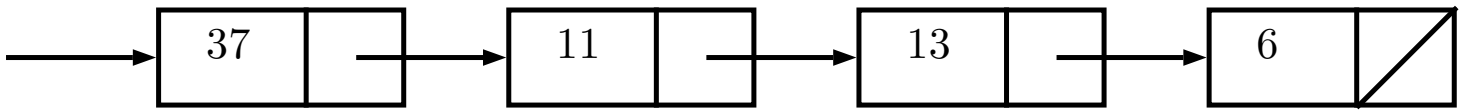
Inserção de 11:



	cabeça	topo	livre
cabeçalho	2	3	-1
Num. Registro: 0	6	-1	
1	13	0	
2	11	1	

Lista Encadeada em Arquivo

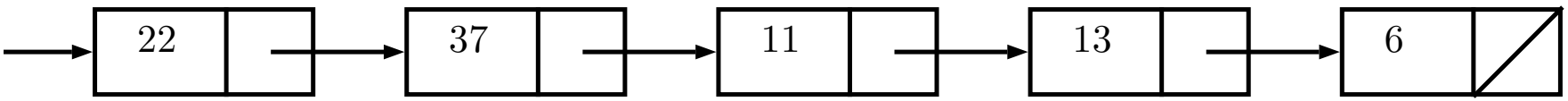
Inserção de 37:



	cabeça	topo	livre
cabeçalho	3	4	-1
Num. Registro:			
0	6	-1	
1	13	0	
2	11	1	
3	37	2	

Lista Encadeada em Arquivo

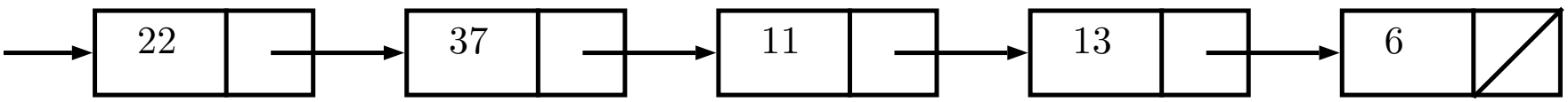
Inserção de 22:



		cabeça	topo	livre
cabeçalho		4	5	-1
Num. Registro:	0	6	-1	
	1	13	0	
	2	11	1	
	3	37	2	
	4	22	3	

Lista Encadeada em Arquivo

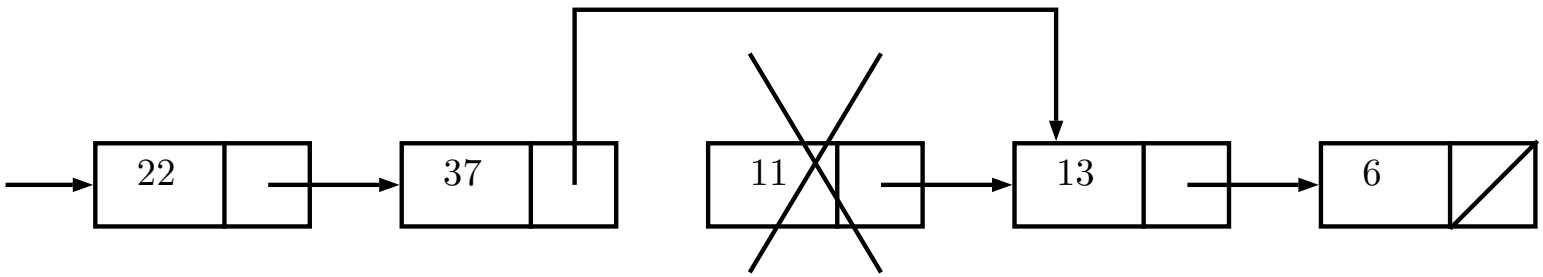
Remover 11, 22 e 6 nesta ordem da lista abaixo:



		cabeça	topo	livre
cabeçalho		4	5	-1
Num. Registro:	0	6	-1	
	1	13	0	
	2	11	1	
	3	37	2	
	4	22	3	

Lista Encadeada em Arquivo

Removendo 11:

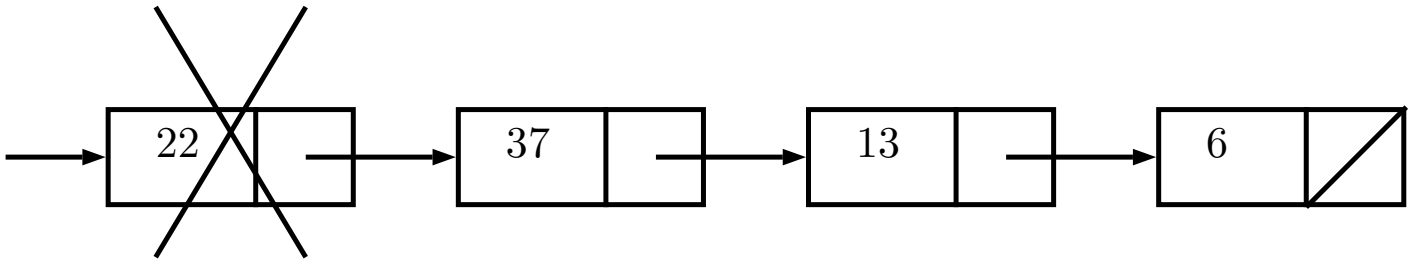


	cabeça	topo	livre
cabeçalho	4	5	2

Num. Registro:		
0	6	-1
1	13	0
2	11	-1
3	37	1
4	22	3

Lista Encadeada em Arquivo

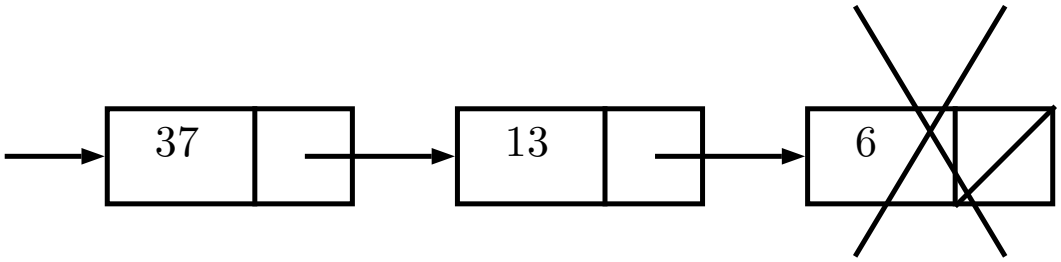
Removendo 22:



cabeçalho	3	5	4
Num. Registro:			
0	6	-1	
1	13	0	
2	11	-1	
3	37	1	
4	22	2	

Lista Encadeada em Arquivo

Removendo 6:

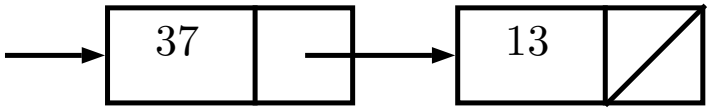


	cabeça	topo	livre
cabeçalho	3	5	0

Num. Registro:		
0	6	4
1	13	-1
2	11	-1
3	37	1
4	22	2

Lista Encadeada em Arquivo

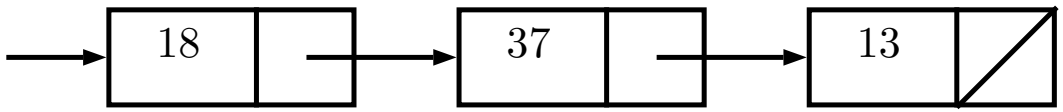
Inserir 18 na lista abaixo:



	cabeça	topo	livre
cabeçalho	3	5	0
Num. Registro: 0	6	4	
1	13	-1	
2	11	-1	
3	37	1	
4	22	2	

Lista Encadeada em Arquivo

Inserindo 18:



	cabeça	topo	livre
cabeçalho	0	5	4

Num. Registro:		
0	18	3
1	13	-1
2	11	-1
3	37	1
4	22	2

Lista Encadeada em Arquivo

Funções para ler e escrever o cabeçalho do arquivo:

```
//Lê o cabeçalho do arquivo contendo as informações da lista
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
//Pós-condição: retorna o ponteiro para o cabeçalho lido

cabecalho* le_cabecalho(FILE * arq) {
   cabecalho * cab = (cabecalho*) malloc(sizeof(cabecalho));
   fseek(arq,0,SEEK_SET); // posiciona no início do arquivo
   fread(cab,sizeof(cabecalho),1,arq);
   return cab;
}

//Escreve no arquivo o cabeçalho contendo as informações da lista
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
//Pós-condição: cabeçalho escrito no arquivo

void escreve_cabecalho(FILE* arq, cabecalho* cab){
   fseek(arq,0,SEEK_SET); //posiciona no início do arquivo
   fwrite(cab,sizeof(cabecalho),1,arq);
}
```

Lista Encadeada em Arquivo

Função para ler um nó da lista no arquivo:

```
//lê um nó em uma determinada posição do arquivo
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
//                pos deve ser uma posição válida da lista
//Pós-condição: ponteiro para nó lido é retornado

no* le_no(FILE* arq, int pos) {
    no* x = malloc(sizeof(no));
    fseek(arq,sizeof(cabecalho)+ pos*sizeof(no),SEEK_SET);
    fread(x,sizeof(no),1,arq);
    return x;
}
```

Lista Encadeada em Arquivo

Função para escrever um nó da lista no arquivo:

```
//Escreve um nó em uma determinada posição do arquivo
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
//              pos deve ser uma posição válida do arquivo
//Pós-condição: nó escrito no arquivo

void escreve_no(FILE* arq, no* x, int pos){
    fseek(arq,sizeof(cabecalho)+ pos*sizeof(no),SEEK_SET);
    fwrite(x,sizeof(no),1,arq);
}
```

Lista Encadeada em Arquivo

Função para inserir um nó na cabeça da lista no arquivo:

```
void insere(FILE* arq, TipoItem info){
    cabecalho* cab = le_cabecalho(arq);
    no x;
    x.info = info;
    x.prox = cab->pos_cabeca;
    if(cab->pos_livre == -1) { // não há nós livres, então usar o topo
          escreve_no(arq,&x,cab->pos_topo);
          cab->pos_cabeca = cab->pos_topo;
          cab->pos_topo++;
    }
     else { // usar nó da lista de livres
          no* aux = le_no(arq,cab->pos_livre);
          escreve_no(arq,&x,cab->pos_livre);
          cab->pos_cabeca = cab->pos_livre;
          cab->pos_livre = aux->prox;
          free(aux);
    }
    escreve_cabecalho(arq,cab); free(cab);
}
```

Lista Encadeada em Arquivo

Função para retirar um nó na cabeça da lista no arquivo:

```
//Retira um nó da lista
//Pré-condição: arquivo deve estar aberto e ser um arquivo de lista
//Pós-condição: nó retirado da lista caso pertença a ela

void retira(FILE* arq, TipoItem x){
    cabecalho* cab = le_cabecalho(arq);
    int pos_aux = cab->pos_cabeca;
    int pos_ant = cab->pos_cabeca;
    no* aux = NULL;
    while(pos_aux != -1 && // procura o elemento a ser retirado
          ((aux = le_no(arq,pos_aux))!= NULL) &&
          aux->info != x){
      pos_ant = pos_aux;
      pos_aux = aux->prox;
      free(aux);
      aux = NULL;
    }
```

Lista Encadeada em Arquivo

Função para retirar um nó na cabeça da lista no arquivo (cont.):

```
    if(pos_aux != -1) { //encontrou o elemento
      if(pos_ant == pos_aux){ // remoção na cabeça
        cab->pos_cabeca = aux->prox;
      }
      else { // remoção no meio
        no * ant = le_no(arq,pos_ant);
        ant->prox = aux->prox;
        escreve_no(arq,ant,pos_ant);
        free(ant);
      }
      aux->prox = cab->pos_livre; // torna o nó removido um nó livre
      cab->pos_livre = pos_aux;
      escreve_no(arq,aux,pos_aux);
      escreve_cabecalho(arq,cab);
      free(aux);
    }
    free(cab);
}
```

Filas

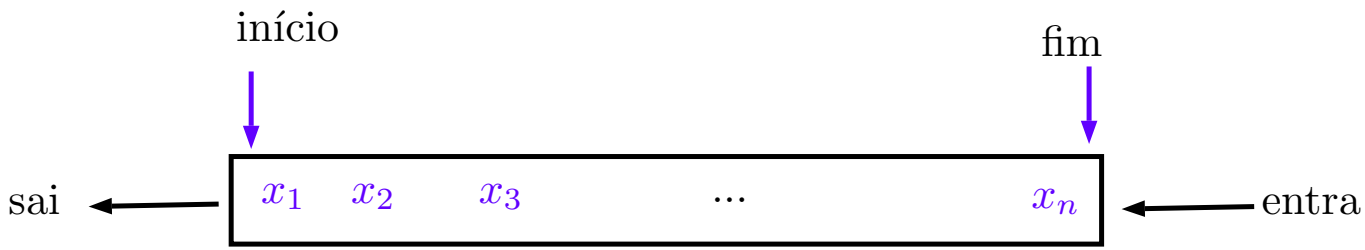
Fila

Uma **fila** é uma lista linear em que as inserções são sempre feitas no final (cauda) e as remoções no início (cabeça).

- O primeiro a entrar é o primeiro a sair (**F***irst*-**I***n* **F***irst*-**O***ut*)
- Exemplo típico: fila de caixa bancário
- Exemplos de aplicações na Computação:
  - **Sistemas Operacionais**: implementação de escalonadores de processos (*scheduler*)
  - **Servidores de Banco de Dados**: atendimento de requisições de consultas

Fila

Representação Esquemática:



Filas

Fila

- Termo em inglês para fila: *queue*. Operações mais comuns:
  - `Enqueue`: enfileirar um elemento
  - `Dequeue`: desenfileirar um elemento

Implementação de Fila usando vetor

```
#define TAM_MAX 1000 // número máximo de itens na fila

typedef int TipoItem;

// estrutura para fila usando vetor
typedef struct {
    TipoItem vet[TAM_MAX];
    int inicio, fim; // índices do início e fim da fila
} Fila;
```

- elementos são dispostos sequencialmente no vetor
- `inicio` e `fim` correspondem à posição do 1^o elemento da fila e à primeira posição livre após o último elemento da fila, respectivamente.

Implementação de Fila usando vetor

```
Fila* cria_fila_vazia() {
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f->inicio = 0;
    f->fim = 0;
    return f;
}

int vazia(Fila* f) {
  return (f->inicio == f->fim);
}

int cheia(Fila* f) {
  return (f->fim >= TAM_MAX);
}
```

- uma fila está vazia quando `inicio` é igual ao `fim`
- uma fila está cheia se a posição de `fim` já passou da última posição válida do vetor (`0..TAM_MAX -1`)

Implementação de Fila usando vetor

```
// Enfileira um elemento
void enqueue(Fila* f, TipoItem x) {
   if(!cheia(f)){
      f->vet[f->fim] = x;
      f->fim++;
   else printf("Fila cheia!");
   }
}
// Desenfileira um elemento
TipoItem* dequeue(Fila * f) {
   if(!vazia(f)) {
     TipoItem*  aux = (TipoItem*) malloc(sizeof(TipoItem));
     *aux = f->vet[f->inicio++];
     return aux;
   }
   else  {
     printf("Fila vazia!");
     return NULL;
   }
}
```

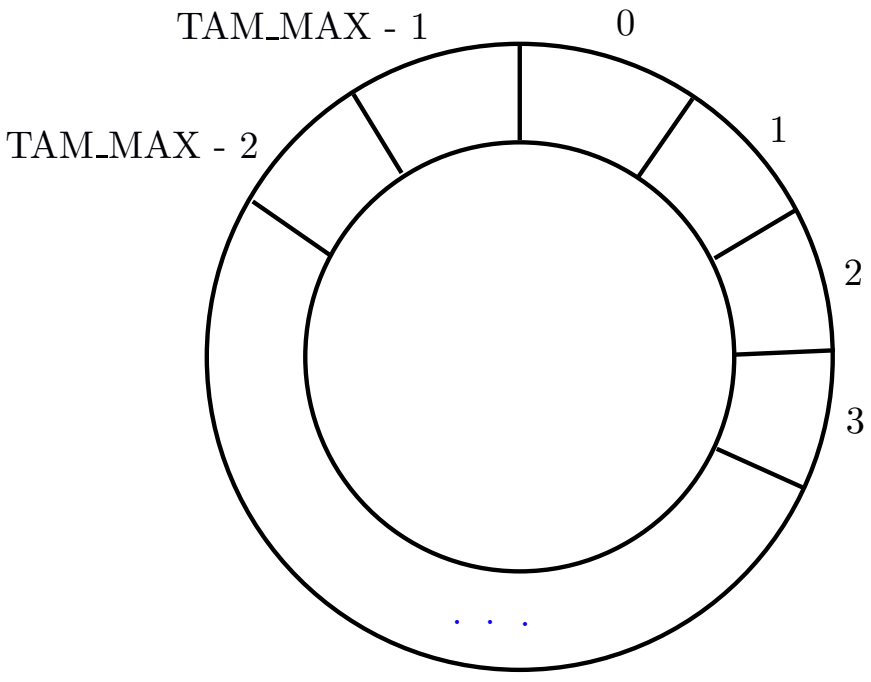
Implementação de Fila usando vetor

Problemas com essa implementação:

- como os índices `fim` e `inicio` deslocam-se sempre para a direita, a fila pode receber no máximo `TAM_MAX` elementos, e uma única vez.
- quando `inicio` e `fim` valerem `TAM_MAX` tanto a função `vazia` quanto a função `cheia` retornará verdadeiro, sendo incoerente.

Solução: usar uma estrutura circular

Implementação de Fila Circular usando vetor



- a fila está vazia quando `inicio` é igual a `fim`
- a fila está cheia quando `fim + 1` módulo `TAM_MAX` é igual a `inicio`
- a posição `fim` deve ficar sempre desocupada para diferenciar a lista cheia da lista vazia

Implementação de Fila Circular usando vetor

```
// Testa se a fila está vazia
int vazia(Fila* f) {
   return (f->inicio == f->fim);
}

//Testa se a fila está cheia

int cheia(Fila* f) {
 //testa se a posição seguinte ao fim é igual a posição de início
 // usando aritmética modular
   return ((f->fim + 1)%TAM_MAX == f->inicio);
}
```

- a fila está cheia quando o próximo do `fim` no vetor circular é o indíce `inicio`

Implementação de Fila Circular usando vetor

```
// Enfileira um elemento
void enqueue(Fila* f, TipoItem x) {
   if(!cheia(f)){
      f->vet[f->fim] = x;
      f->fim++;
      if(f->fim == TAM_MAX) // implementa circularidade
        f->fim = 0;
   else printf("Fila cheia!");
   }
}
```

Implementação de Fila Circular usando vetor

```
// Desenfileira um elemento
TipoItem* dequeue(Fila * f) {
   if(!vazia(f)) {
     TipoItem* aux = (TipoItem*) malloc(sizeof(TipoItem));
     *aux = f->vet[f->inicio];
     f->inicio++;
     if(f->inicio == TAM_MAX) // implementa circularidade
       f->inicio = 0;
     return aux;
   }
   else {
         printf("Fila vazia!");
         return NULL;
   }
}
```

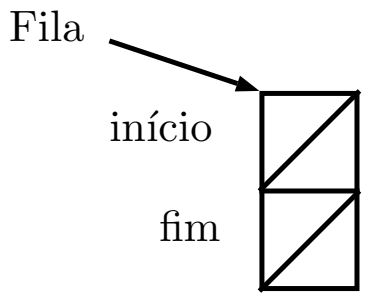
Implementação de Fila usando lista encadeada

```
typedef int TipoItem;

struct no {
     TipoItem info;
     struct no* prox;
};
typedef struct {
    struct no* inicio;
    struct no* fim;
} Fila;
```

- Uso de lista encadeada com cabeça e cauda para facilitar as inserções e remoções

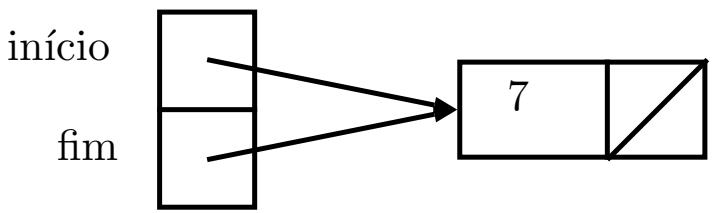
Fila

Inserir 7, 11, 4 e 6 em uma fila inicialmente vazia.
Fila vazia:



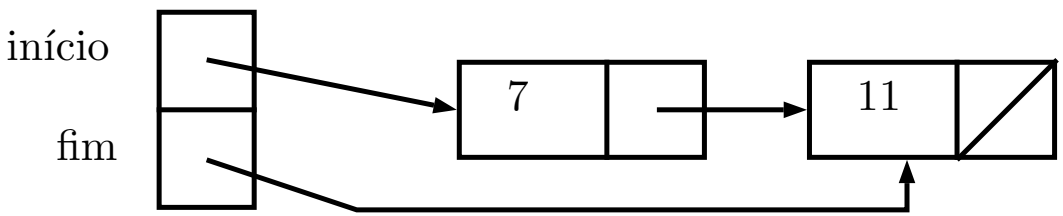
Fila

7 inserido:



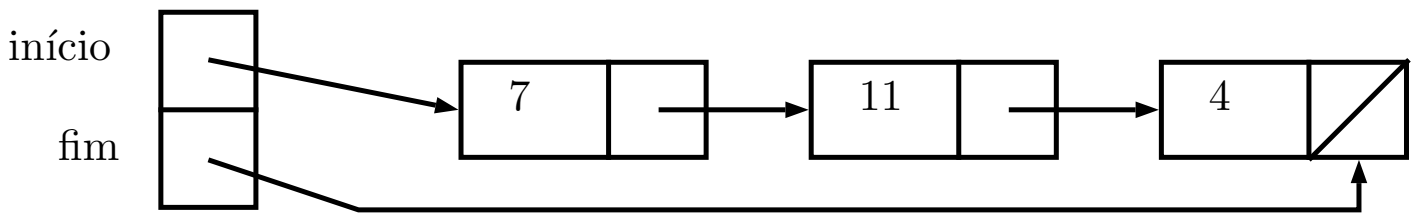
Fila

11 inserido:



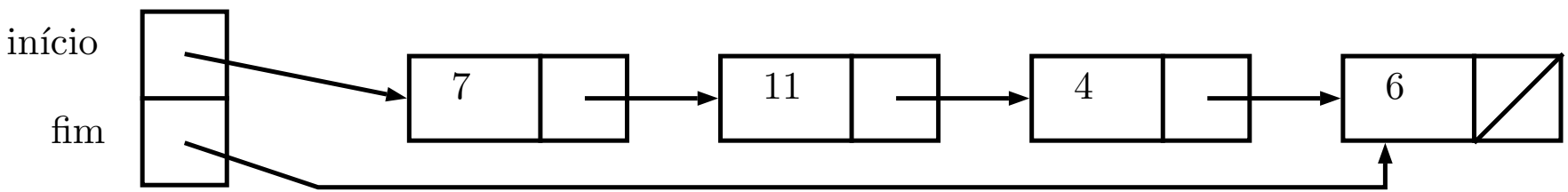
Fila

4 inserido:



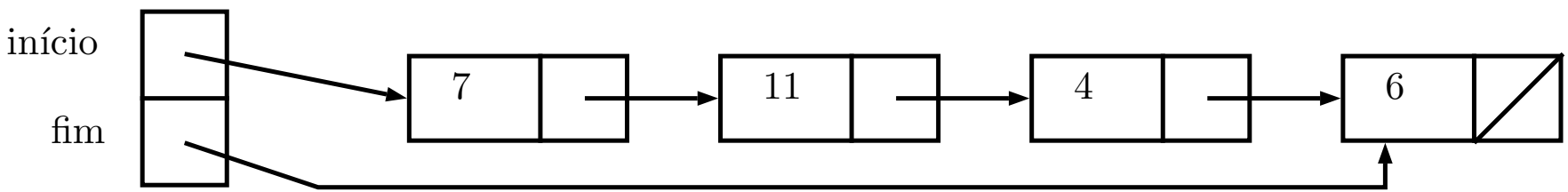
Fila

6 inserido:



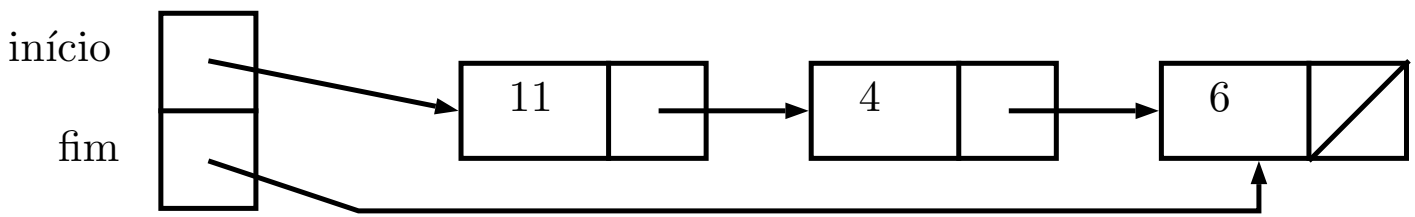
Fila

Efetuar 2 operações *dequeue* na fila abaixo:



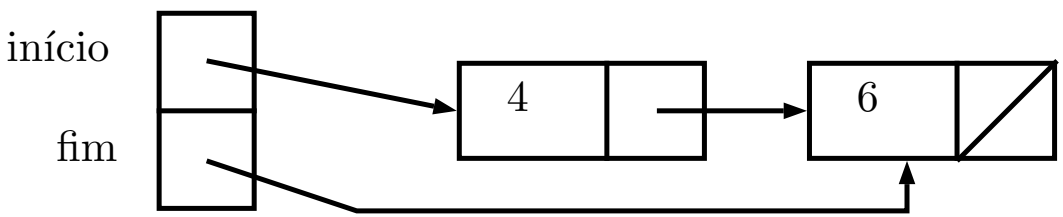
Fila

Resultado do 1º *dequeue*:



Fila

Resultado do 2^o *dequeue*:



Implementação de Fila usando lista encadeada

```
int vazia(Fila* f) {
  return (f->inicio == NULL);
}

Fila* cria_fila_vazia() {
  Fila* f = (Fila*) malloc(sizeof(Fila));
  f->inicio = NULL;
  f->fim = NULL;
  return f;
}
```

Implementação de Fila usando lista encadeada

```
\\Enfileira um elemento
void enqueue(Fila* f, TipoItem x) {
  struct no* aux = (struct no*) malloc(sizeof(struct no));
  aux->info = x;
  aux->prox = NULL;
  if(vazia(f)) {
    f->inicio = aux;
   }
   else {
     f->fim->prox = aux;
   }
  f->fim = aux;
}
```

Implementação de Fila usando lista encadeada

```
//Desenfileira um elemento
TipoItem* dequeue(Fila* f) {
  if(!vazia(f)) {
     TipoItem* x = (TipoItem*) malloc(sizeof(TipoItem));
     struct no* aux = f->inicio;
     *x = f->inicio->info;
     if(f->inicio == f->fim) // só tem 1 elemento
        f->fim = NULL;
     f->inicio = f->inicio->prox;
     free(aux);
     return x;
  }
  else return NULL;

}
```

Pilhas

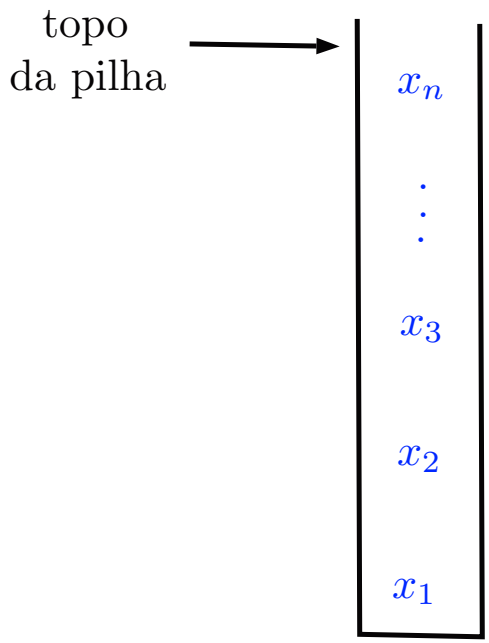
Pilha

Uma **pilha** é uma lista linear em que tanto as inserções quanto remoções são sempre feitas na cabeça (ou topo da pilha).

- O último a entrar é o primeiro a sair (**L***ast*-**I***n* **F***irst*-**O***ut*)
- Exemplos típicos: pilha de livros, pilha de pratos.
- Exemplos de aplicações na Computação:
  - **Editores de texto**: mecanismo desfazer/refazer
  - **Compiladores**: implementação de analisadores sintáticos de linguagens de programação
  - **Compiladores**: modelo de ambiente de execução adotado pelas linguagens de programação Pascal, C, C++, Java, entre outras, em que parâmetros e endereço de retorno de cada procedimento/função/subrotina/método invocado é armazenado em uma pilha para controle das chamadas.
  - **Algoritmos**: pode ser usada como estrutura auxiliar na implementação de algoritmos como forma de eliminação de recursão.

Pilha

Representação Esquemática:



Pilhas

Pilha

- Termo em inglês para pilha: *stack*. Operações mais comuns:
  - `push`: empilhar um elemento
  - `pop`: desempilhar um elemento

Implementação de Pilha usando vetor

```
#define TAM_MAX 1000

typedef int TipoItem ;

typedef struct {
    TipoItem vet[TAM_MAX];
    int topo;
} Pilha;
```

- topo marca o topo da pilha, que corresponde à primeira posição livre do vetor.

Implementação de Pilha usando vetor

```
int vazia(Pilha * p) {
 return (p->topo == 0);
}

int cheia(Pilha* p) {
 return (p->topo == TAM_MAX);
}

Pilha* criar_pilha_vazia(){
  Pilha* p = (Pilha*) malloc(sizeof(Pilha));
  p->topo = 0;
  return p;
}
```

- A pilha está vazia se o topo é igual a zero e está cheia se topo é igual a TAM_MAX

Implementação de Pilha usando vetor

```
//Empilha um elemento
void push(Pilha* p, TipoItem x) {
   if(!cheia(p)){
     p->vet[p->topo++] = x;
   }
   else printf("Pilha cheia!");
}

//Desempilha um elemento
TipoItem* pop(Pilha* p) {
  if(!vazia(p)){
    TipoItem* aux = (TipoItem*) malloc(sizeof(TipoItem));
     topo--;
    *aux = p->vet[p->topo];
    return aux;
  else {
    printf("Pilha vazia!");
    return NULL;
  }
```

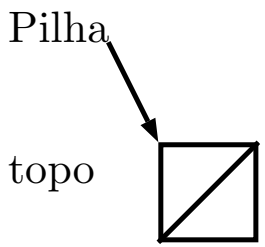
Implementação de Pilha usando lista encadeada

```
typedef int TipoItem;

// estrutura para lista encadeada
struct no {
   TipoItem info;
   struct no *prox;
   };

// estrutura de lista com cabeça
// para implementação de pilha
typedef struct {
  struct no* topo;
} Pilha;
```

- uso de lista encadeada com cabeça para marcação do topo da pilha.

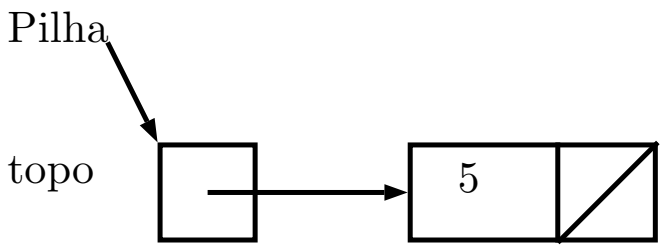
Pilha

Considere empilhar 5, 8 e 3 em uma pilha inicialmente vazia
Pilha vazia:



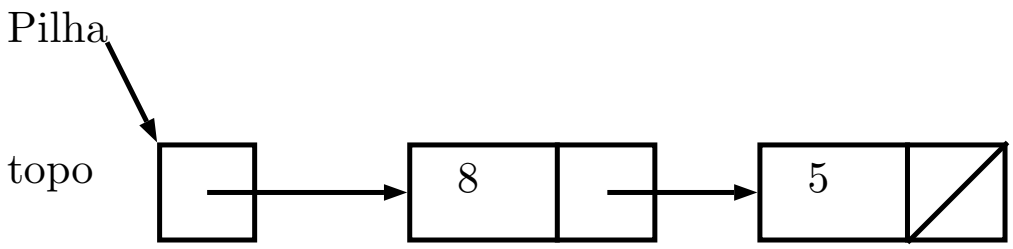
Pilha

5 empilhado:



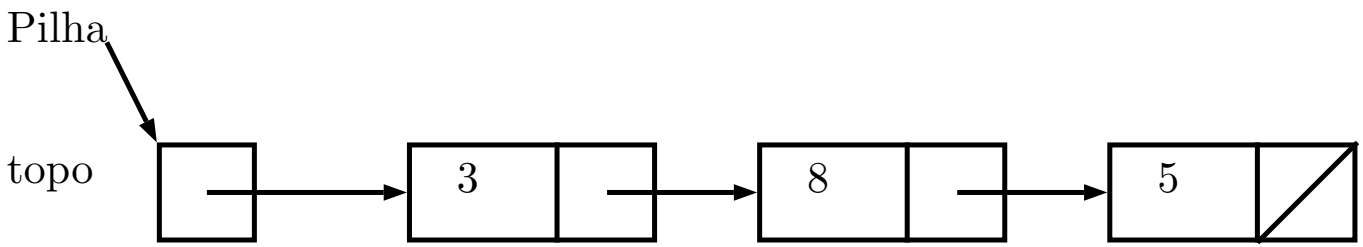
Pilha

8 empilhado:



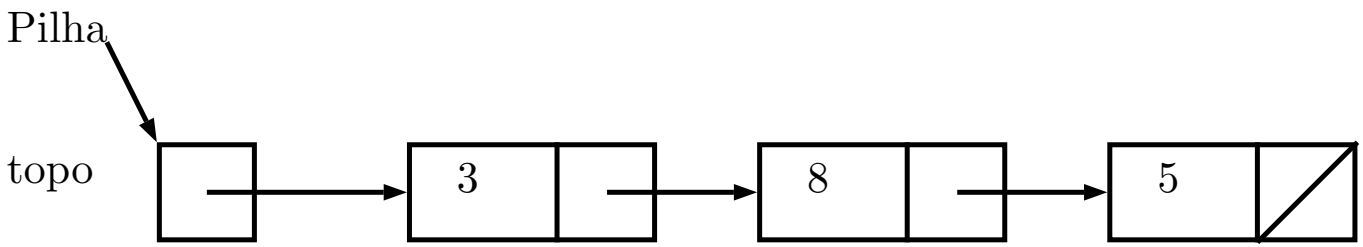
Pilha

3 empilhado:



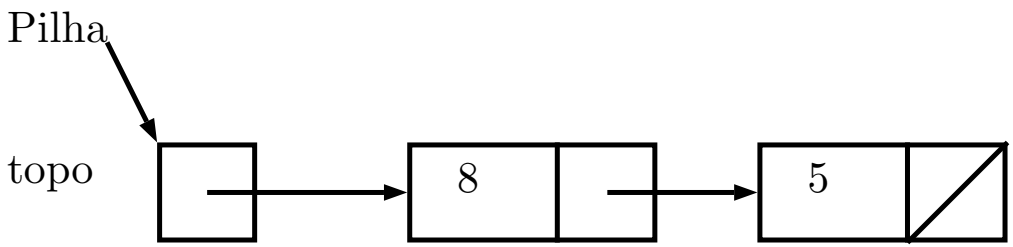
Pilha

Considere executar 2 operações de desempilhar (*pop*) na pilha abaixo:



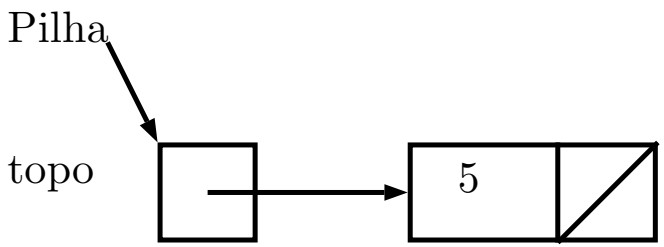
Pilha

Resultado do 1º *pop*:



Pilha

Resultado do 2º *pop*:



Implementação de Pilha usando lista encadeada

```
int vazia(Pilha* p) {
 return (p->topo == NULL);
}

Pilha* criar_pilha_vazia() {
  Pilha* aux = (Pilha*) malloc(sizeof(Pilha));
  aux->topo = NULL;
  return aux;
}
```

- A pilha é vazia se o `topo` é `NULL`

Implementação de Pilha usando lista encadeada

```
// Lê o topo da pilha sem desempilhar
TipoItem* topo(Pilha* p) {
  if(!vazia(p)) {
     TipoItem* x = (TipoItem*) malloc(sizeof(TipoItem));
     *x = p->topo->info;
     return x;
  }
  else return NULL;
}
```

- A função topo permite consultar o topo da pilha sem desempilhar

Implementação de Pilha usando lista encadeada

```
// Empilha um item
void push(Pilha* p, TipoItem x) {
   struct no* aux = (struct no*) malloc(sizeof(struct no));
   aux->info = x;
   aux->prox = p->topo;
   p->topo = aux;
}
```

Implementação de Pilha usando lista encadeada

```
//Desempilha um item
TipoItem* pop(Pilha* p) {
   if(!vazia(p)) {
     TipoItem* x = (TipoItem*) malloc(sizeof(TipoItem));
     struct no* aux = p->topo;
     *x = p->topo->info;
     p->topo = p->topo->prox;
     free(aux);
     return x;
   }
   else {
     printf("Pilha vazia!");
     return NULL;
   }
}
```

Observações quanto à complexidade de tempo

Sendo n o número de elementos já inseridos na estrutura:

	inserção na cabeça	inserção na cauda
lista em vetor	O(1)	O(1)
lista encadeada simples	O(1)	O(n)
lista encadeada com cabeça e cauda	O(1)	O(1)

Observações quanto à complexidade de tempo

A operação de busca de um elemento em uma lista, em qualquer uma das variantes é O(n) no pior caso, supondo-se que os elementos estejam desordenados e o elemento procurado é o último.

Observações quanto à complexidade de tempo

Sendo n o número de elementos já inseridos na estrutura:

	enqueue	dequeue
fila circular usando vetor	O(1)	O(1)
fila usando lista encadeada com cabeça e cauda	O(1)	O(1)

Observações quanto à complexidade de tempo

Sendo n o número de elementos já inseridos na estrutura:

	push	pop
pilha usando vetor	O(1)	O(1)
pilha usando lista encadeada com cabeça	O(1)	O(1)