Estruturas de dados básicas

Estruturas de dados

- Programas operam sobre dados
- Dados são relacionados e possuem estrutura
- **Como representar e manipular dados em um computador**

Exemplo – Baralho

- Para representar um baralho precisamos:
  - Representar cartas: naipe e valor
    - `struct carta { char naipe; char valor; };`
  - Representar a ordem das cartas
- Operações de manipulação:
  - Comprar a carta no topo do baralho
  - Colocar uma carta no fundo do baralho
  - Embaralhar
  - Retirar uma carta aleatória do meio do baralho

Escolhendo a estrutura adequada

- Estruturas de dados têm vantagens e desvantagens
- Devemos escolher a estrutura de dados depois de analisar as operações que vamos realizar sobre os dados

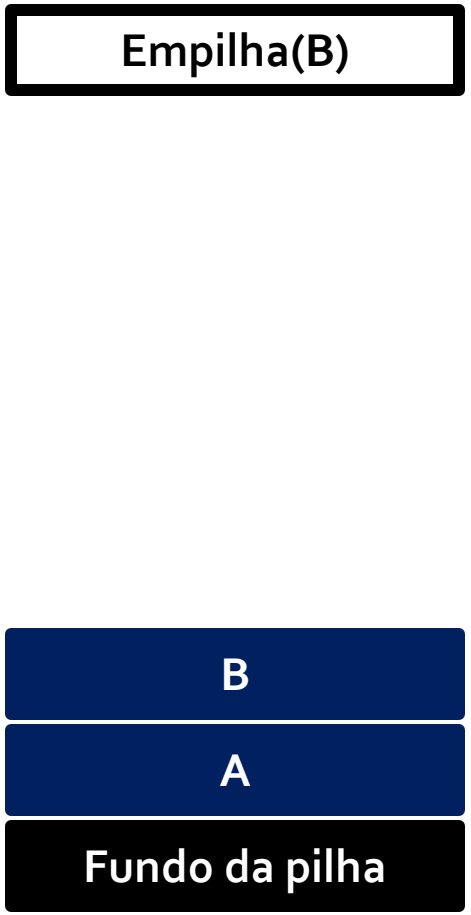
Pilhas

Pilhas

- Operações:
  - Criar uma pilha
  - Empilhar um elemento
  - Desempilhar um elemento
  - Recuperar o tamanho da pilha
  - Destruir uma pilha
- Último elemento a entrar é o primeiro elemento a sair

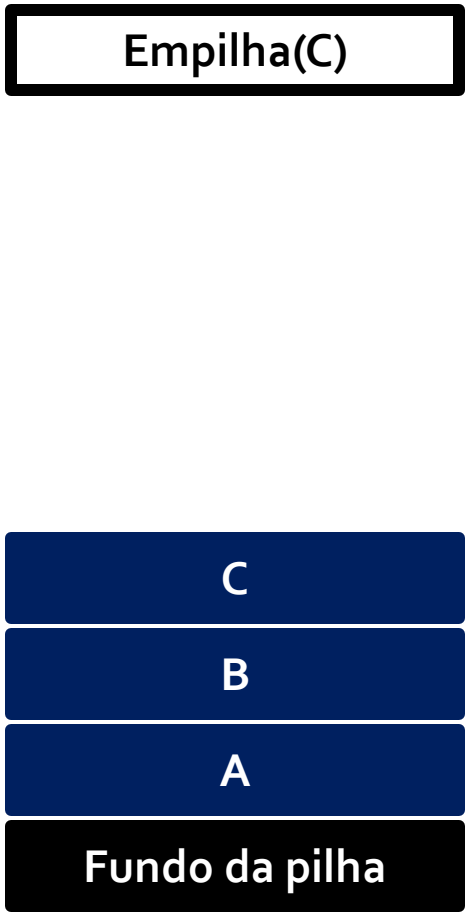
Pilhas

- Operações:
  - Criar uma pilha
  - Empilhar um elemento
  - Desempilhar um elemento
  - Recuperar o tamanho da pilha
  - Destruir uma pilha
- Último elemento a entrar é o primeiro elemento a sair



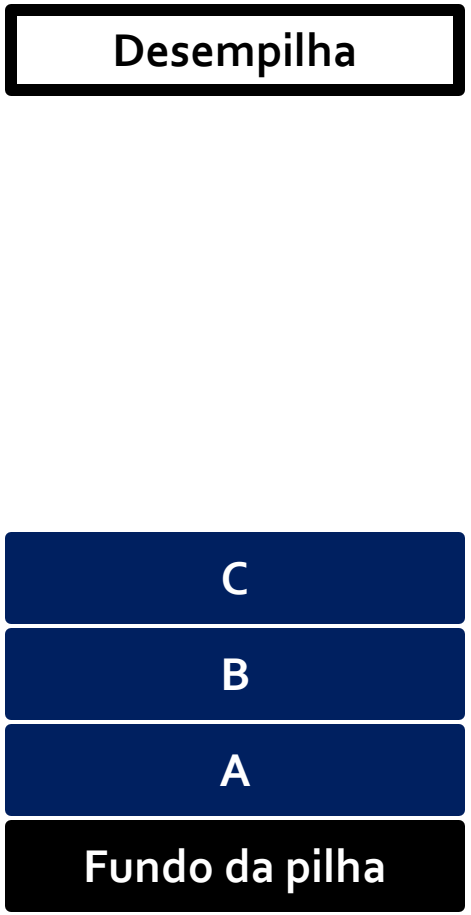
Pilhas

- Operações:
  - Criar uma pilha
  - Empilhar um elemento
  - Desempilhar um elemento
  - Recuperar o tamanho da pilha
  - Destruir uma pilha
- Último elemento a entrar é o primeiro elemento a sair



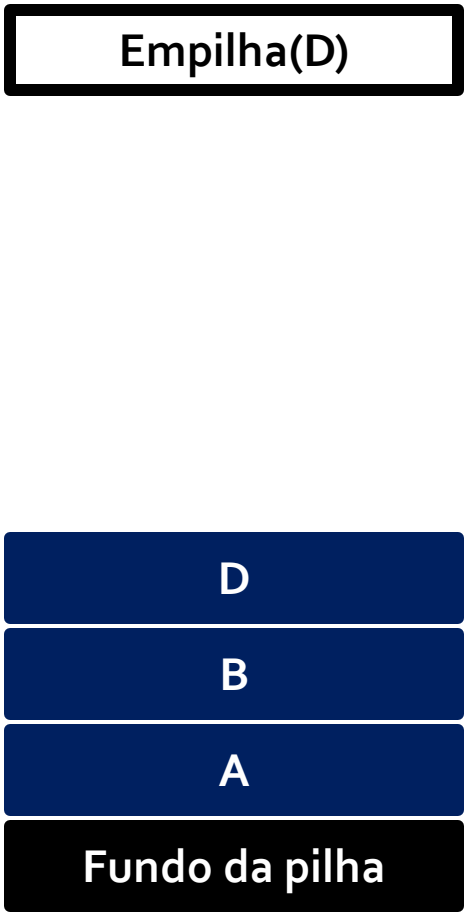
Pilhas

- Operações:
  - Criar uma pilha
  - Empilhar um elemento
  - Desempilhar um elemento
  - Recuperar o tamanho da pilha
  - Destruir uma pilha
- Último elemento a entrar é o primeiro elemento a sair



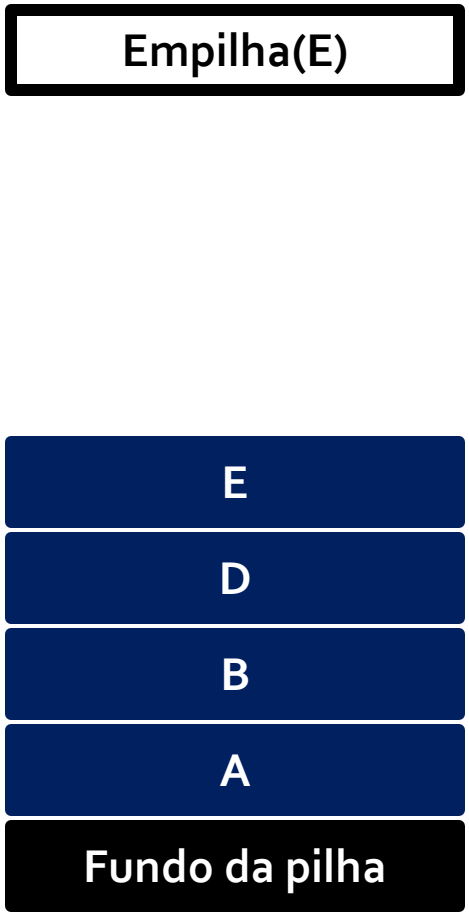
Pilhas

- Operações:
  - Criar uma pilha
  - Empilhar um elemento
  - Desempilhar um elemento
  - Recuperar o tamanho da pilha
  - Destruir uma pilha
- Último elemento a entrar é o primeiro elemento a sair



Pilhas

- Operações:
  - Criar uma pilha
  - Empilhar um elemento
  - Desempilhar um elemento
  - Recuperar o tamanho da pilha
  - Destruir uma pilha
- Último elemento a entrar é o primeiro elemento a sair



Usos de pilha

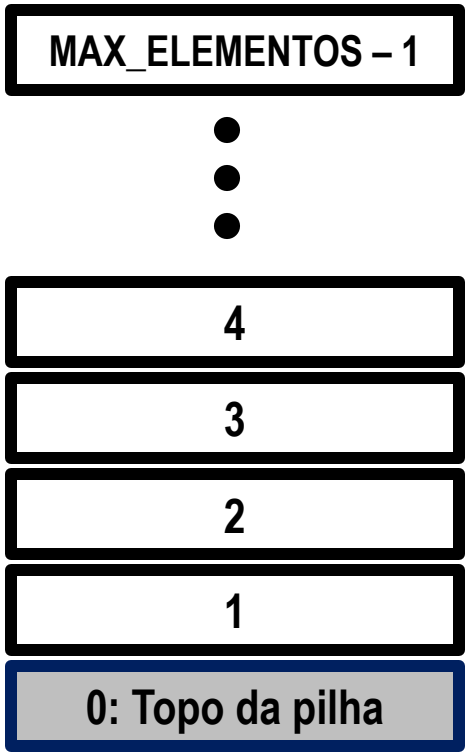
- Solução de expressões matemáticas
  - Calculadora HP-12c
- Guardar variáveis locais em chamadas recursivas de função

Implementação de pilha com arranjo

- Armazenamento em posições contíguas de um arranjo
- Como só precisamos saber a posição do elemento que está no topo, usamos um inteiro para armazenar seu índice no arranjo

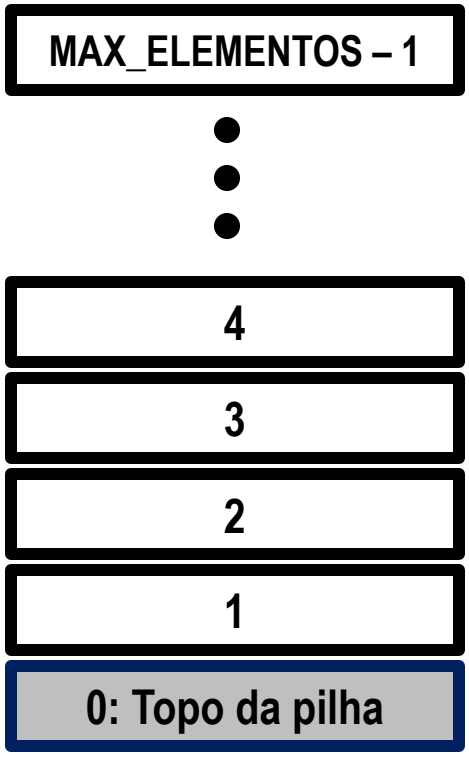
```
struct pilha {
    int elementos[MAX_ELEMENTOS];
    int topo;
}
```

Criação de uma pilha



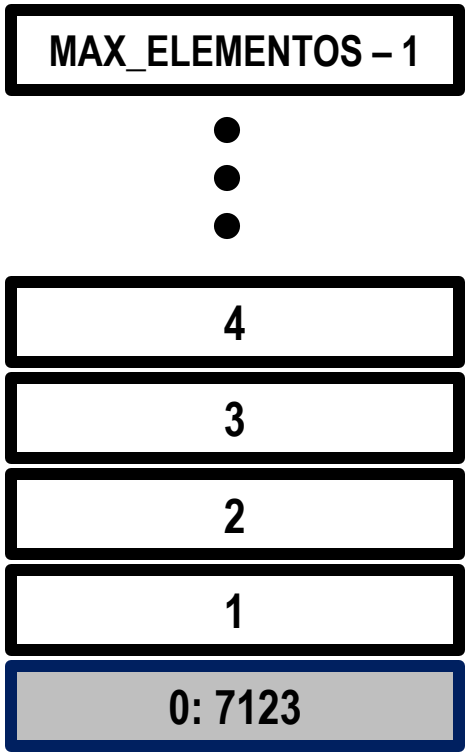

```
struct pilha {
    int elementos[MAX_ELEMENTOS];
    int topo;
}

struct pilha * cria(void) {
    struct pilha *p;
    p = malloc(sizeof(struct pilha));
    if(!p) { perror(NULL); exit(1); }
    /* IMPORTANTE: */
    p->topo = 0;
    /* Não precisa alocar o vetor
     * por que ele está estático na
     * struct pilha. */
}
```

Empilhando um elemento



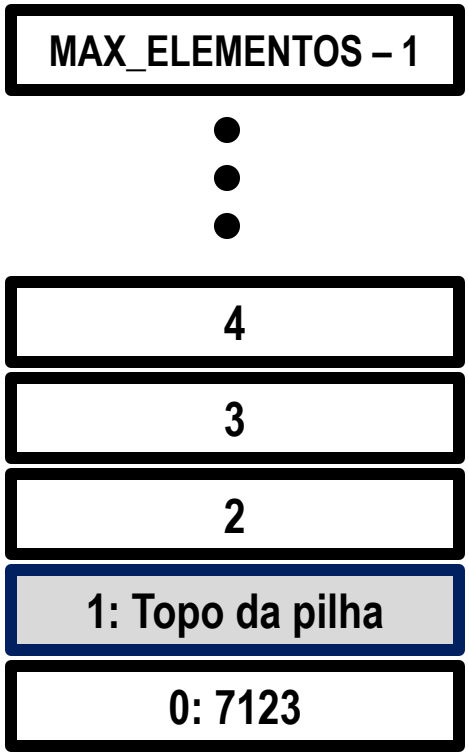

```
struct pilha {
    int elementos[MAX_ELEMENTOS];
    int topo;
}

void empilha(struct pilha *p, int A){
    p->elementos[p->topo] = A;
    p->topo = p->topo + 1;
}

...
    empilha(p, 7123);
...
```

Empilhando um elemento



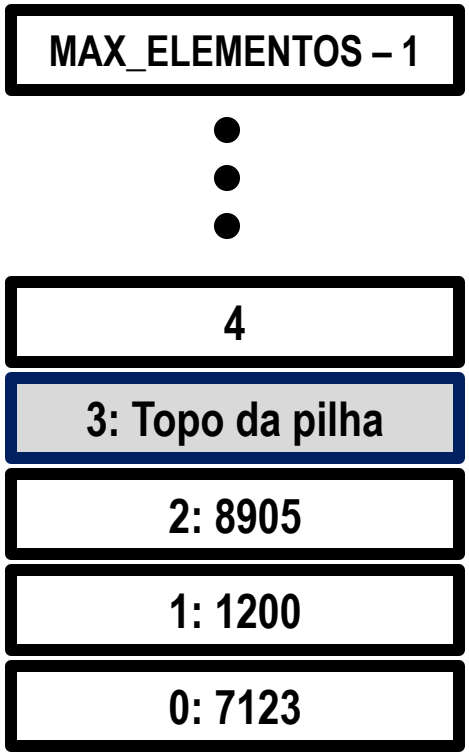

```
struct pilha {
    int elementos[MAX_ELEMENTOS];
    int topo;
}

void empilha(struct pilha *p, int A){
    p->elementos[p->topo] = A;
    p->topo = p->topo + 1;
}

...
    empilha(p, 7123);
...
```

Empilhando um elemento



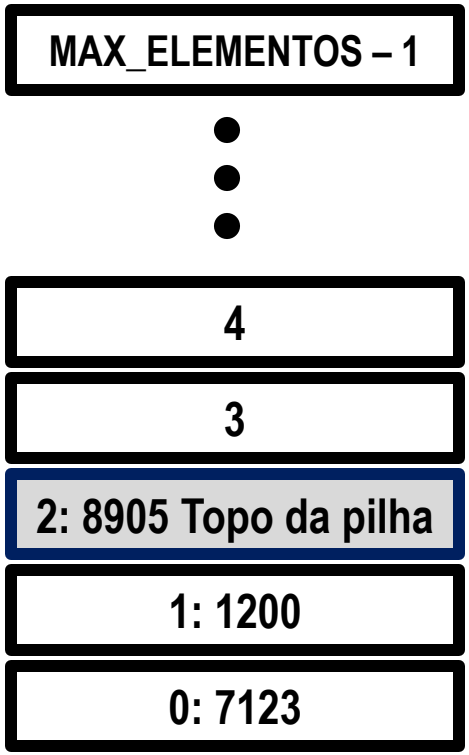

```
struct pilha {
    int elementos[MAX_ELEMENTOS];
    int topo;
}

void empilha(struct pilha *p, int A){
    p->elementos[p->topo] = A;
    p->topo = p->topo + 1;
}

...
    empilha(p, 7123);
...
```

Desempilhando um elemento



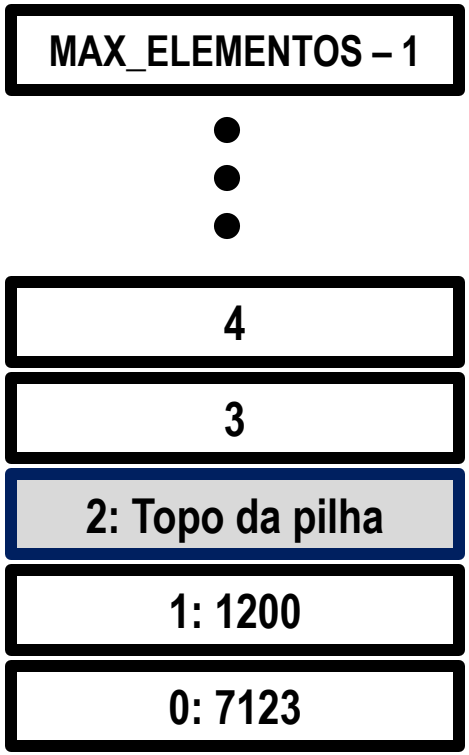

```
struct pilha {
     int elementos[MAX_ELEMENTOS];
     int topo;
}

int desempilha(struct pilha *p) {
     p->topo = p->topo - 1;
     return p->elementos[p->topo];
}

...
     int t = desempilha(p);
...
```

Desempilhando um elemento



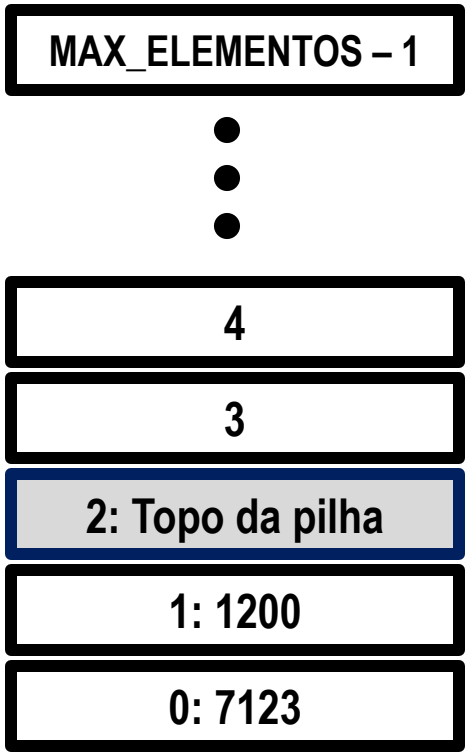

```
struct pilha {
     int elementos[MAX_ELEMENTOS];
     int topo;
}

int desempilha(struct pilha *p) {
     p->topo = p->topo - 1;
     return p->elementos[p->topo];
}

...
     int t = desempilha(p);
...
```

Desempilhando um elemento




```
struct pilha {
    int elementos[MAX_ELEMENTOS];
    int topo;
}

int desempilha(struct pilha *p) {
    p->topo = p->topo - 1;
    return p->elementos[p->topo];
}

...
    int t = desempilha(p);
    /* t == 8905 */
...
```

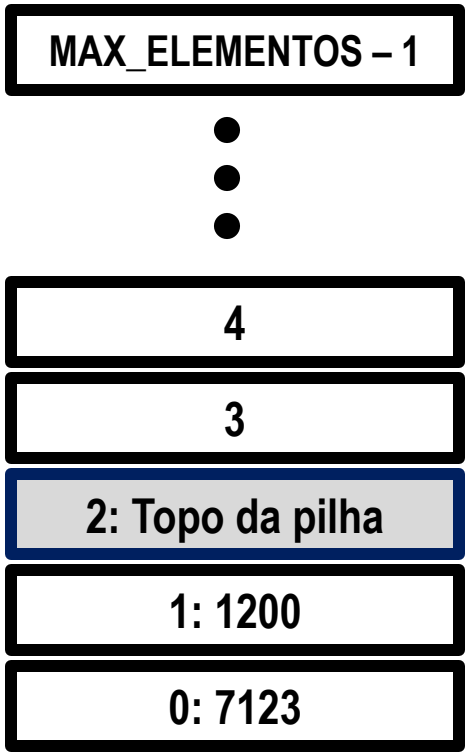
Calculando o tamanho da pilha




```
struct pilha {
     int elementos[MAX_ELEMENTOS];
     int topo;
}

int tamanho(struct pilha *p) {
     return p->topo;
}
```

Destruindo uma pilha

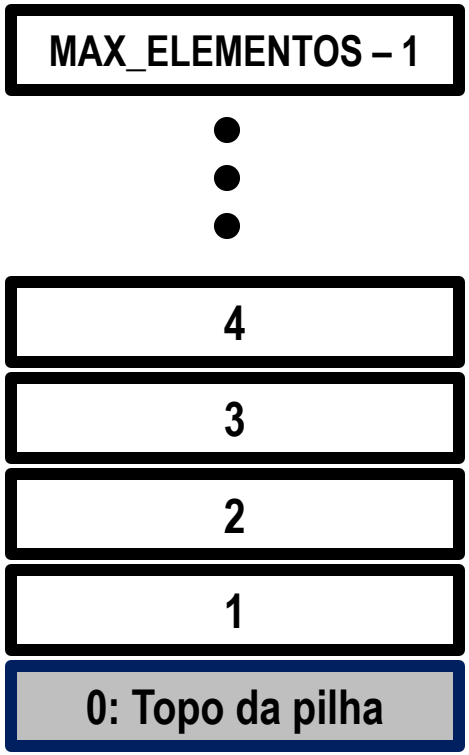
```
struct pilha {
     int elementos[MAX_ELEMENTOS];
     int topo;
}

int destroi(struct pilha *p) {
     free(pilha);
}
```



Exercício

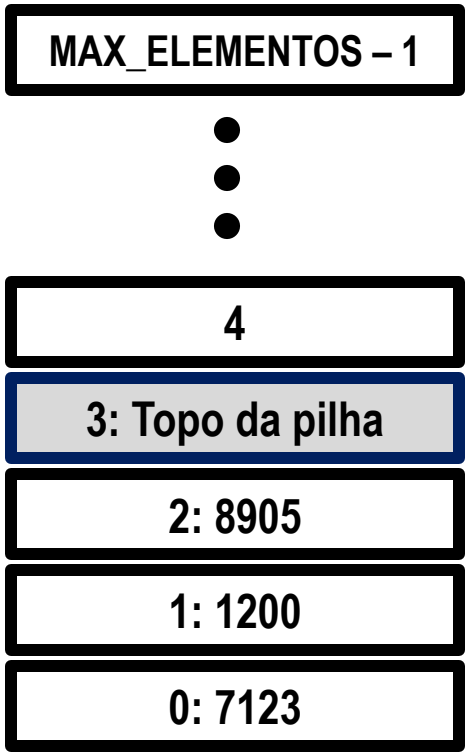
- O código apresentado não testa se a pilha está cheia antes de empilhar um elemento. Modifique a função **`empilha`** para que ela imprima um erro se a pilha estiver cheia.
- O código apresentado não testa se a pilha está vazia antes de desempilhar. Modifique a função **`desempilha`** pra que ela imprima um erro se a pilha estiver vazia.

Empilhando um elemento XP




```
struct pilha {
    int elementos[MAX_ELEMENTOS];
    int topo;
}

void empilha(struct pilha *p, int A){
    if(p->topo == MAX_ELEMENTOS-1) {
        printf("pilha cheia.");
        exit(1);
    }
    p->elementos[p->topo] = A;
    p->topo = p->topo + 1;
}
```

Desempilhando um elemento




```
struct pilha {
    int elementos[MAX_ELEMENTOS];
    int topo;
}

int desempilha(struct pilha *p) {
    if(p->topo == 0) {
        printf("pilha vazia.");
        exit(1);
    }
    p->topo = p->topo - 1;
    return p->elementos[p->topo];
}
```

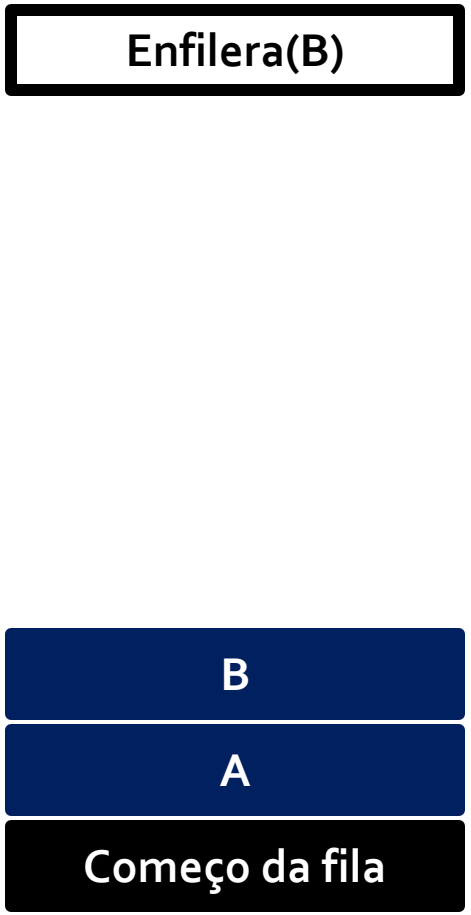
Filas

Filas

- Operações:
  - Criar uma fila
  - Enfileirar um elemento
  - Desenfileirar o primeiro elemento
  - Recuperar o tamanho da fila
  - Destruir uma fila
- Primeiro elemento a entrar é o primeiro elemento a sair

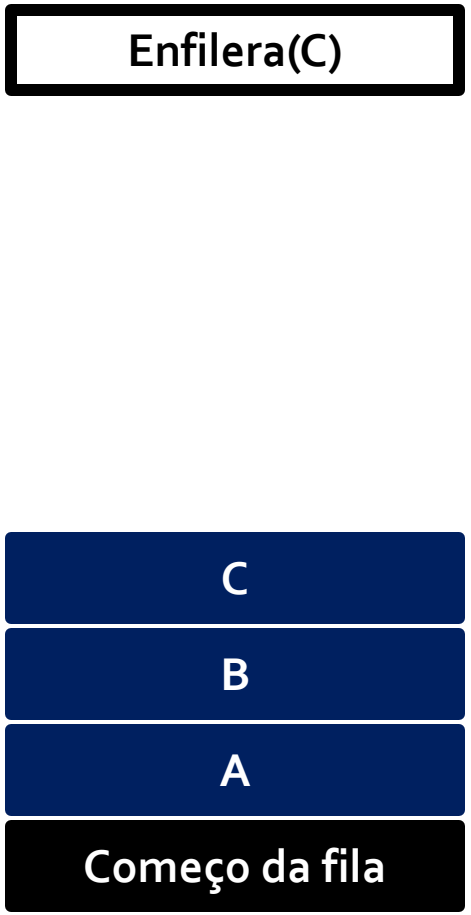
Filas

- Operações:
  - Criar uma fila
  - Enfileirar um elemento
  - Desenfileirar o primeiro elemento
  - Recuperar o tamanho da fila
  - Destruir uma fila
- Primeiro elemento a entrar é o primeiro elemento a sair



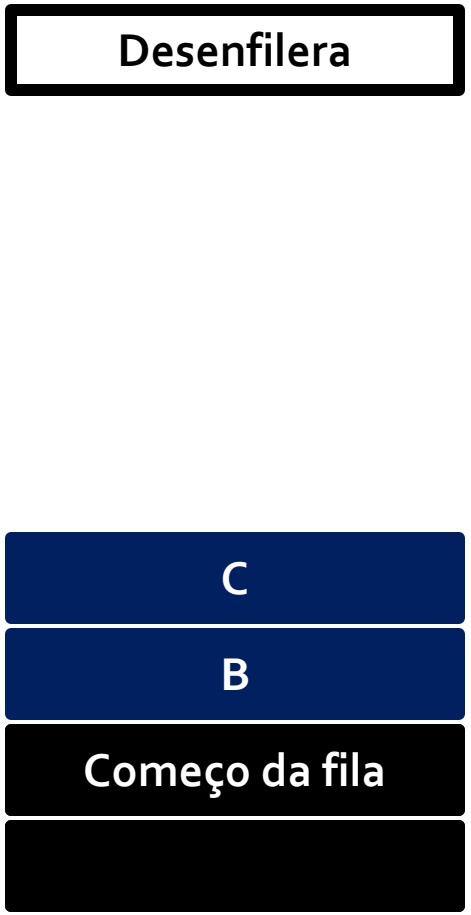
Filas

- Operações:
  - Criar uma fila
  - Enfileirar um elemento
  - Desenfileirar o primeiro elemento
  - Recuperar o tamanho da fila
  - Destruir uma fila
- Primeiro elemento a entrar é o primeiro elemento a sair



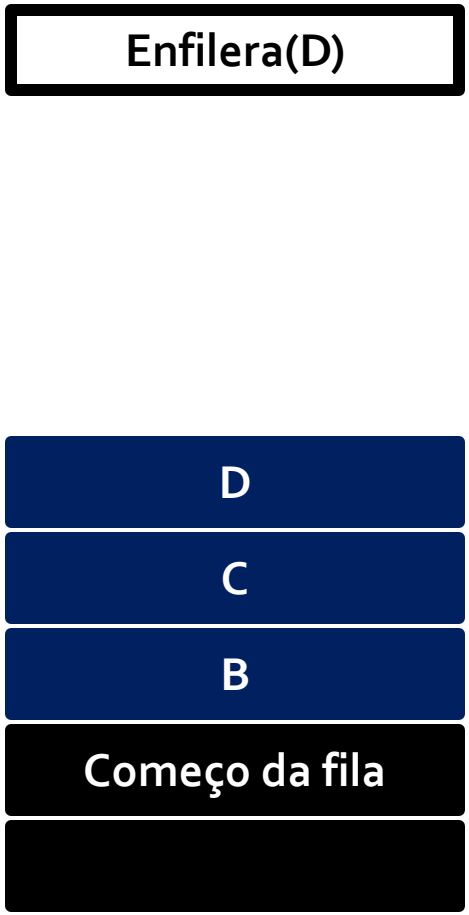
Filas

- Operações:
  - Criar uma fila
  - Enfileirar um elemento
  - Desenfileirar o primeiro elemento
  - Recuperar o tamanho da fila
  - Destruir uma fila
- Primeiro elemento a entrar é o primeiro elemento a sair



Filas

- Operações:
  - Criar uma fila
  - Enfileirar um elemento
  - Desenfileirar o primeiro elemento
  - Recuperar o tamanho da fila
  - Destruir uma fila
- Primeiro elemento a entrar é o primeiro elemento a sair

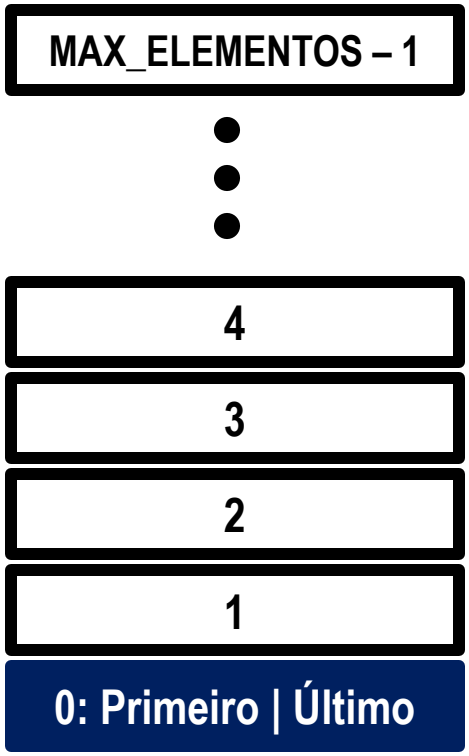


Implementação de filas com arranjo

- Itens são armazenados em posições contíguas de um arranjo
- Operação enfilera expande a parte de trás da fila: marcar o índice do último elemento
- Operação desenfilera retrai a parte da frente da fila: marcar o índice do primeiro elemento

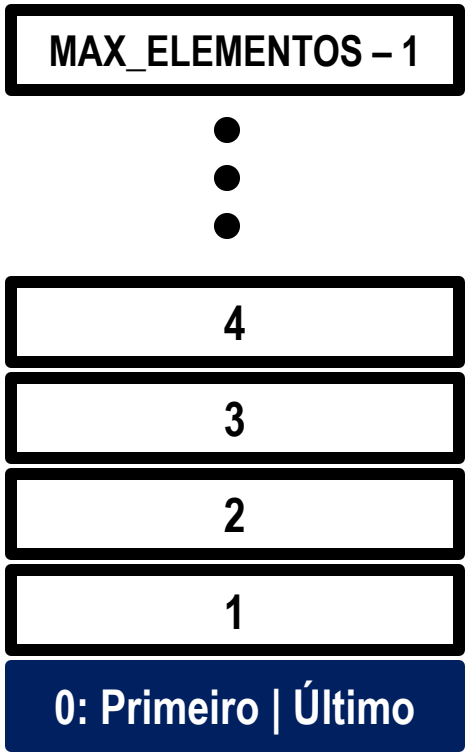
```
struct fila {
    int elementos[MAX_ELEMENTOS];
    int primeiro;
    int ultimo;
}
```

Criando uma fila



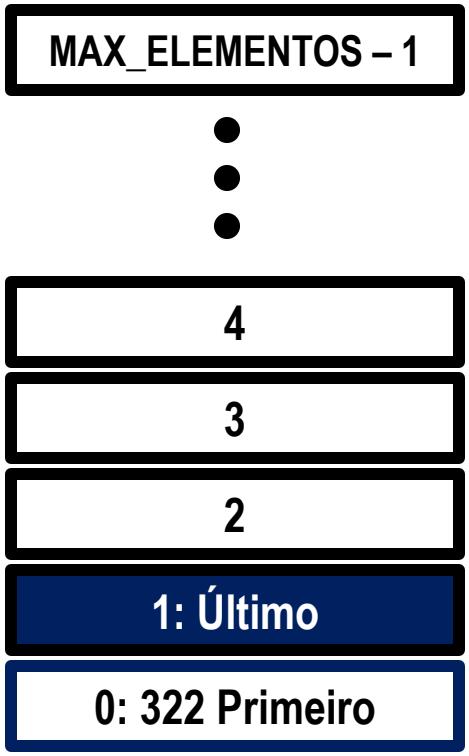

```
struct fila {
    int elementos[MAX_ELEMENTOS];
    int primeiro; int ultimo;
}

struct fila * cria(void) {
    struct fila *f;
    f = malloc(sizeof(struct fila));
    if(!f) { perror(NULL); exit(1); }
    /* IMPORTANTE: */
    f->primeiro = 0;
    f->ultimo = 0;
}
```

Enfileirando um elemento



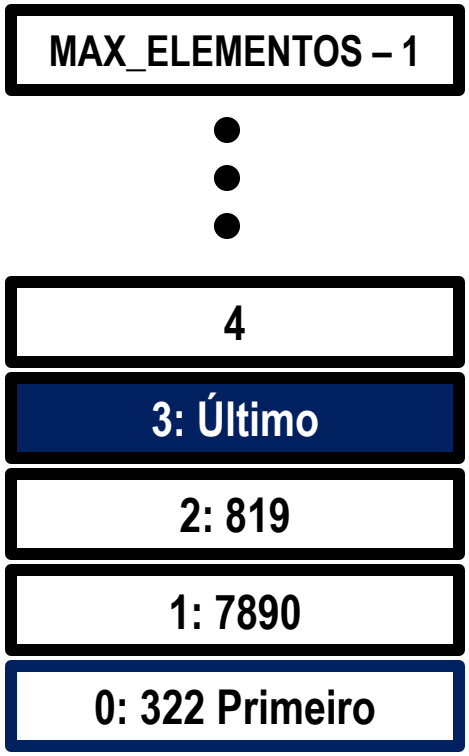

```
struct fila {
    int elementos[MAX_ELEMENTOS];
    int primeiro; int ultimo;
}

void enfilera(struct fila *f, int A){
    f->elementos[f->ultimo] = A;
    f->ultimo += 1;
}


...
    enfilera(f, 322);
...
```

Enfileirando um elemento



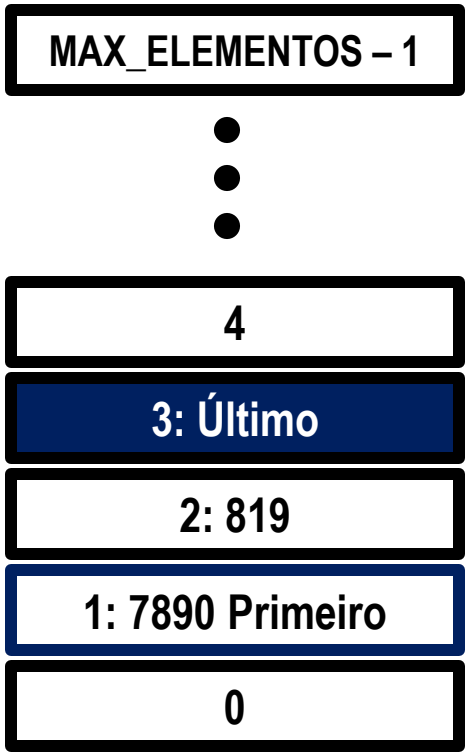

```
struct fila {
    int elementos[MAX_ELEMENTOS];
    int primeiro; int ultimo;
}

void enfilera(struct fila *f, int A){
    f->elementos[f->ultimo] = A;
    f->ultimo += 1;
}


...
    enfilera(f, 322);
...
```

Desenfileirando um elemento



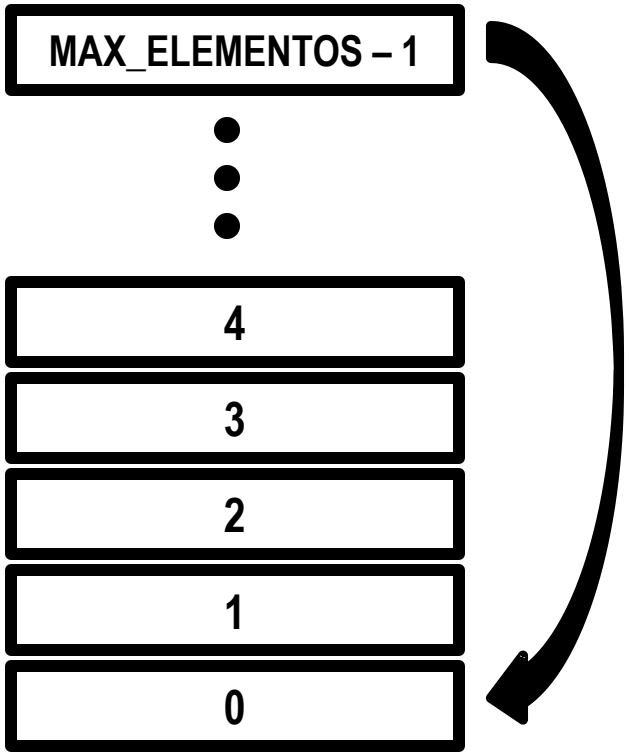

```
struct fila {
     int elementos[MAX_ELEMENTOS];
     int primeiro; int ultimo;
}

int desenfilera(struct fila *f) {
    int r = f->elementos[f->primeiro];
    f->primeiro += 1;
    return r;
}

...
     int p = desenfilera(f);
...
```

Desenfileirando um elemento



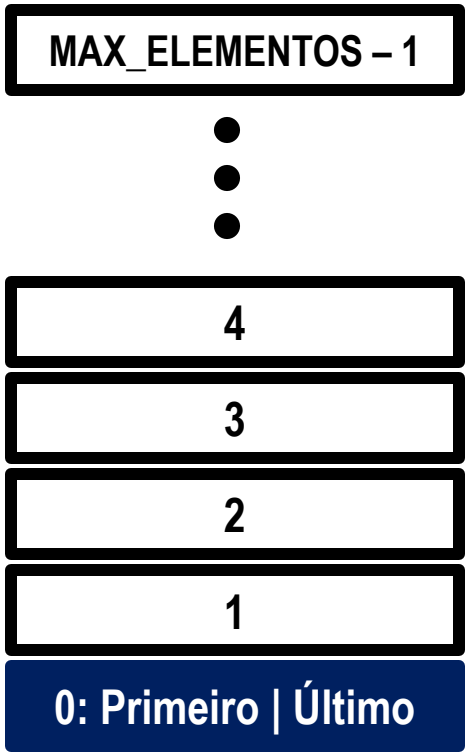

```
struct fila {
     int elementos[MAX_ELEMENTOS];
     int primeiro; int ultimo;
}

int desenfilera(struct fila *f) {
    int r = f->elementos[f->primeiro];
    f->primeiro += 1;
    return r;
}

...
     int p = desenfilera(f);
...
```

Ooops, limite de memória



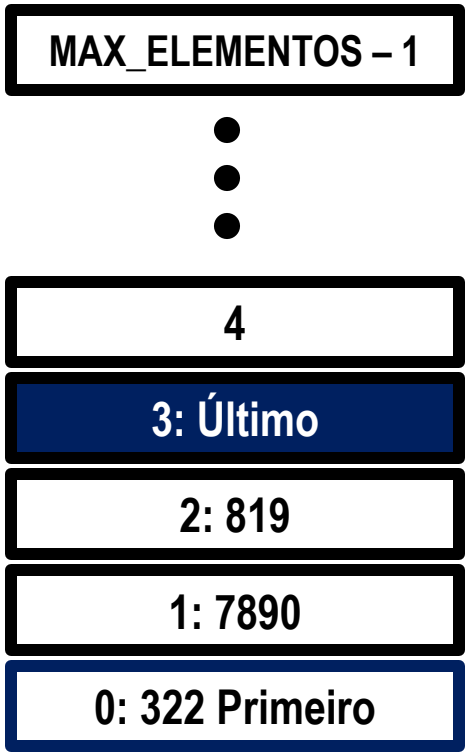
- Depois de várias inserções e remoções, a fila pode ultrapassar a memória dos elementos!
- Solução: Imaginar o arranjo como um círculo. A posição que vem depois de **`(ELEMENTOS_MAX - 1)`** é a posição zero

Enfileirando um elemento XP



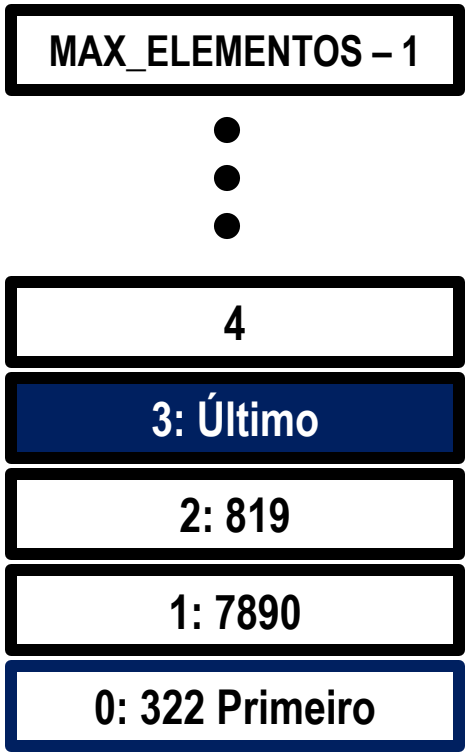

```
struct fila {
    int elementos[MAX_ELEMENTOS];
    int primeiro; int ultimo;
}

void enfilera(struct fila *f, int A){
    f->elementos[f->ultimo] = A;
    f->ultimo += 1;
    if(f->ultimo == MAX_ELEMENTOS) {
       f->ultimo = 0;
    }
}
```

Desenfileirando um elemento XP



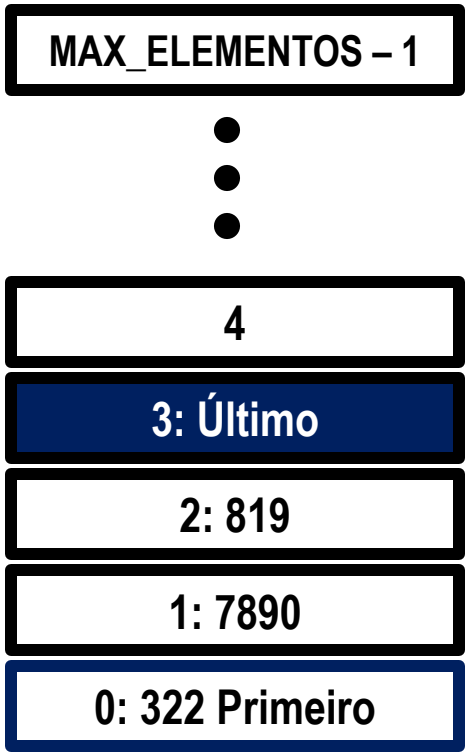

```
struct fila {
     int elementos[MAX_ELEMENTOS];
     int primeiro; int ultimo;
}

int desenfilera(struct fila *f) {
    int r = f->elementos[f->primeiro];
    f->primeiro += 1;
    if(f->primeiro == MAX_ELEMENTOS) {
       f->primeiro = 0;
    }
    return r;
}
```

Contando elementos




```
struct fila {
     int elementos[MAX_ELEMENTOS];
     int primeiro; int ultimo;
}

int tamanho(struct fila *f) {
    int t = f->ultimo - f->primeiro;
    if(t < 0) { t += MAX_ELEMENTOS; }
    return t;
}
```

Destruindo uma fila

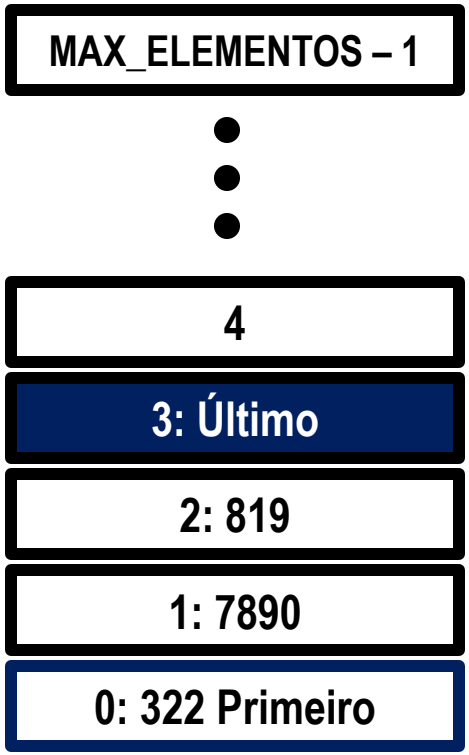



```
struct fila {
     int elementos[MAX_ELEMENTOS];
     int primeiro; int ultimo;
}

void destroi(struct fila *f) {
     free(f);
}
```

Exercício

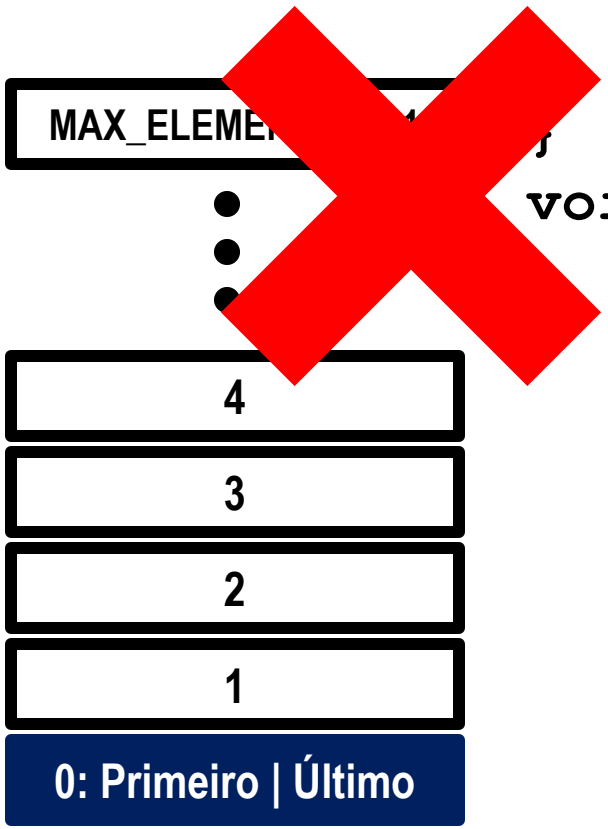
- O código apresentado não checa se a fila está cheia antes de enfileirar um elemento. Modifique a função **`enfilera`** para que ela imprima um erro se a fila estiver cheia.
- O código apresentado não checa se a fila está vazia antes de desenfileirar um elemento. Modifique a função **`desenfilera`** para que ela imprima um erro se a fila estiver vazia.

Desenfileirando um elemento XP



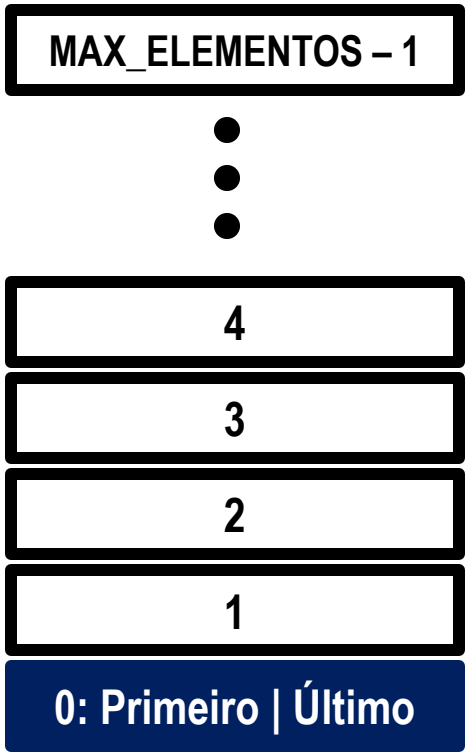

```
struct fila {
     int elementos[MAX_ELEMENTOS];
     int primeiro; int ultimo;
}
int desenfilera(struct fila *f) {
    if(f->primeiro == f->ultimo) {
        printf("fila vazia.\n");
        exit(1);
    }
    int r = f->elementos[f->primeiro];
    f->primeiro += 1;
    if(f->primeiro == MAX_ELEMENTOS) {
       f->primeiro = 0;
    }
    return r;
}
```

Enfileirando um elemento XP SP3




```
struct fila {
    int elementos[MAX_ELEMENTOS];
    int primeiro; int ultimo;
};
void enfilera(struct fila *f, int A){
    if(f->ultimo + 1 == f->primeiro){
        printf("fila cheia.\n");
        exit(1);
    }
    f->elementos[f->ultimo] = A;
    f->ultimo += 1;
    if(f->ultimo == MAX_ELEMENTOS) {
        f->ultimo = 0;
    }
}
```

Enfileirando um elemento XP SP3




```
struct fila {
    int elementos[MAX_ELEMENTOS];
    int primeiro; int ultimo;
}
void enfilera(struct fila *f, int A){
    if((f->ultimo + 1 == f->primeiro)
       || (f->ultimo == MAX_ELEM-1
           && f->primeiro == 0)) {
        printf("fila cheia.\n");
        exit(1);
    }
    ...
```

Listas

Listas

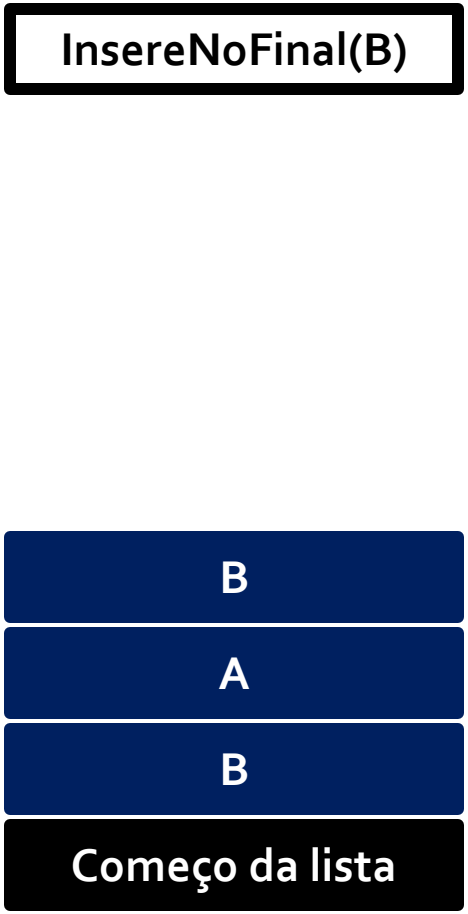
- Operações:
  - Criar uma fila
  - Inserir um elemento na posição x
  - Remover o elemento na posição y
  - Recuperar o elemento na posição z
  - Inserir ou remover elementos no início ou no fim da lista
  - Recuperar o tamanho da lista
  - Destruir uma fila

Listas

- Operações:
  - Criar uma fila
  - Inserir um elemento na posição x
  - Remover o elemento na posição y
  - Recuperar o elemento na posição z
  - Inserir ou remover elementos no início ou no fim da lista
  - Recuperar o tamanho da lista
  - Destruir uma fila

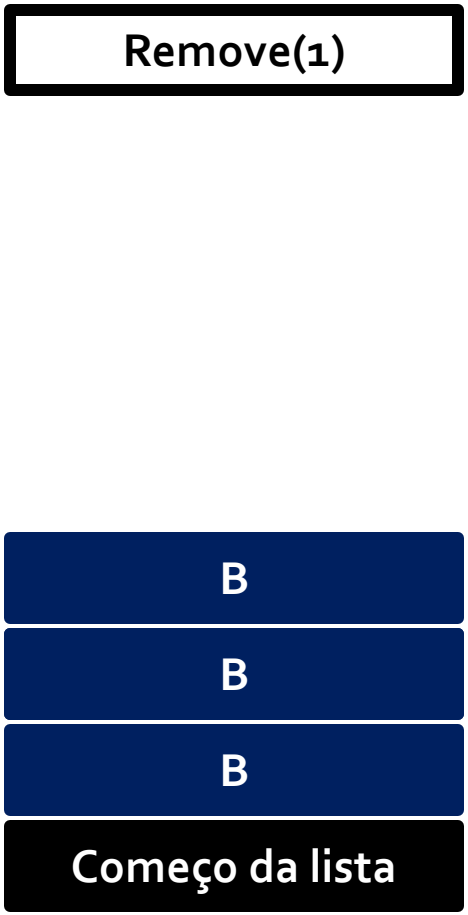
Listas

- Operações:
  - Criar uma fila
  - Inserir um elemento na posição x
  - Remover o elemento na posição y
  - Recuperar o elemento na posição z
  - Inserir ou remover elementos no início ou no fim da lista
  - Recuperar o tamanho da lista
  - Destruir uma fila



Listas

- Operações:
  - Criar uma fila
  - Inserir um elemento na posição x
  - Remover o elemento na posição y
  - Recuperar o elemento na posição z
  - Inserir ou remover elementos no início ou no fim da lista
  - Recuperar o tamanho da lista
  - Destruir uma fila

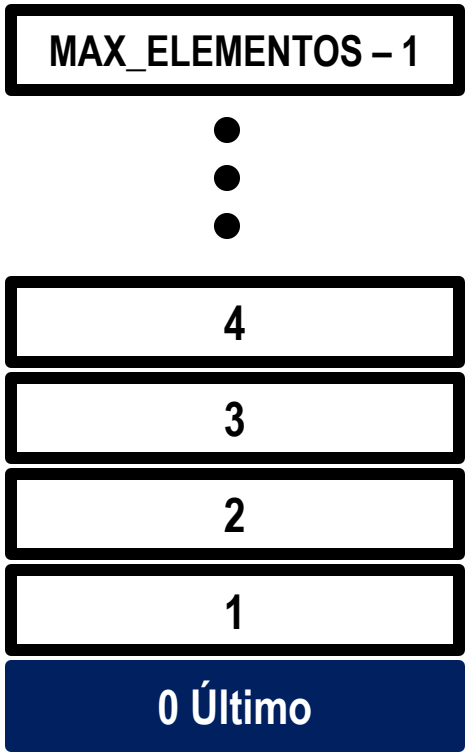


Implementação de lista com arranjo

- Itens são armazenados em posições contíguas de um arranjo
- Precisamos guardar o índice do último elemento para inserir e remover elementos no final da lista

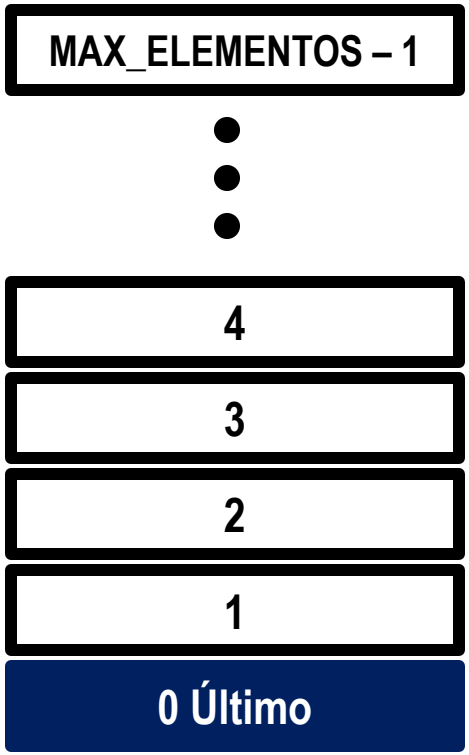
```
struct lista {
     int elementos[MAX_ELEMENTOS];
     int ultimo;
}
```

Criando uma lista




```
struct lista {
     int elementos[MAX_ELEMENTOS];
     int ultimo;
}

struct lista * cria(void) {
     struct lista *f;
     f = malloc(sizeof(struct lista));
     if(!f) { perror(NULL); exit(1); }
     /* IMPORTANTE: */
     f->ultimo = 0;
}
```

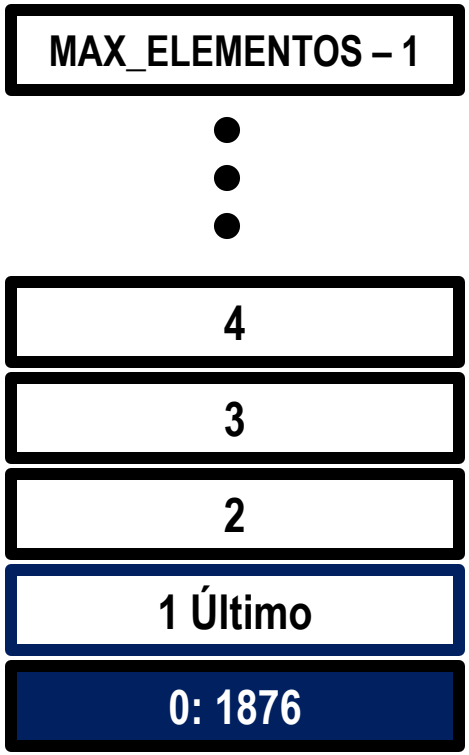
Inserindo um elemento no final




```
struct lista {
     int elementos[MAX_ELEMENTOS];
     int ultimo;
}
void insere_final(struct lista *f,
                  int A){




     f->elementos[f->ultimo] = A;
     f->ultimo += 1;
}
```

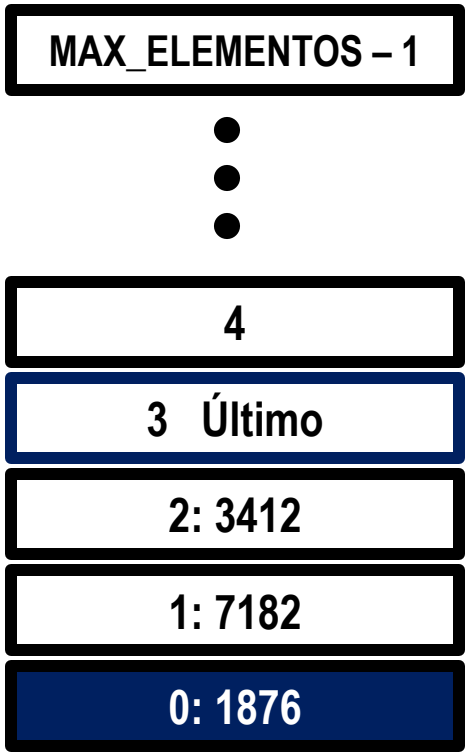
Inserindo um elemento no final

O(1)




```
struct lista {
     int elementos[MAX_ELEMENTOS];
     int ultimo;
}
void insere_final(struct lista *f,
                  int A){
     if(f->ultimo == MAX_ELEMENTOS-1){
          printf("lista cheia.\n");
          exit(1);
     }
     f->elementos[f->ultimo] = A;
     f->ultimo += 1;
}
```

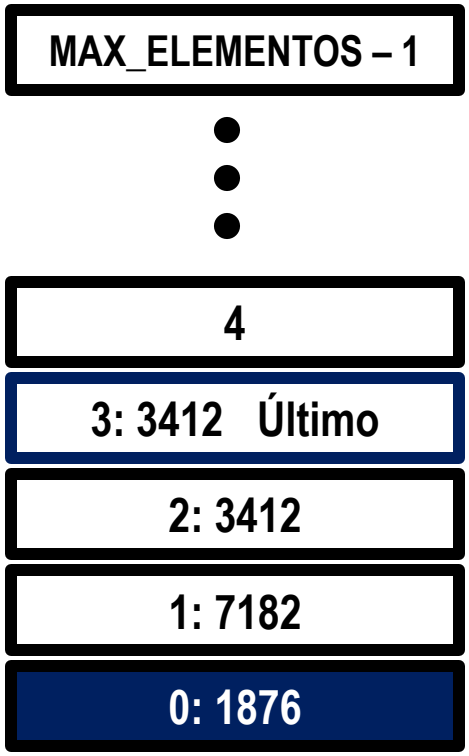
Inserindo um elemento na posição X




```
struct lista {
     int elementos[MAX_ELEMENTOS];
     int ultimo;
}
void insere(struct lista *f,
                int A, int posicao){
   if(f->ultimo == MAX_ELEMENTOS-1){
      printf("lista cheia"); exit(1);
   }
   for(int i = f->ultimo; i > x; i--){
      f->elemtos[i] = f->elemtos[i-1];
   }
   f->ultimo += 1;
   f->elementos[x] = A;
}
```

Inserindo um elemento na posição X

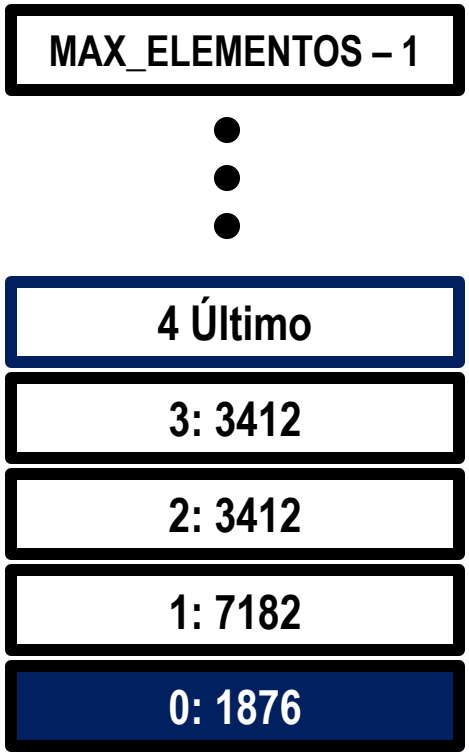
```
insere(f, 8818, 2);
```




```
void insere(struct lista *f,
                int A, int posicao){
    if(f->ultimo == MAX_ELEMENTOS-1){
        printf("lista cheia"); exit(1);
    }
    for(int i = f->ultimo; i > x; i--){
        f->elemtos[i] = f->elemtos[i-1];
    }
    f->ultimo += 1;
    f->elementos[x] = A;
}
```

Inserindo um elemento na posição X

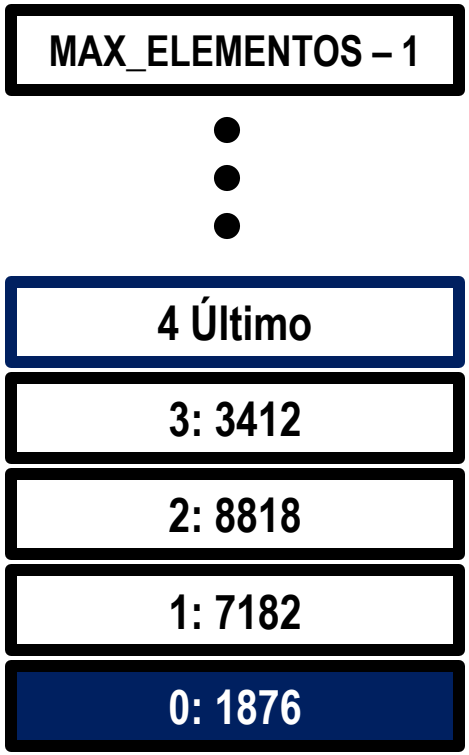
```
insere(f, 8818, 2);
```




```
void insere(struct lista *f,
                int A, int posicao){
    if(f->ultimo == MAX_ELEMENTOS-1){
        printf("lista cheia"); exit(1);
    }
    for(int i = f->ultimo; i > x; i--){
        f->elemtos[i] = f->elemtos[i-1];
    }
    f->ultimo += 1;
    f->elementos[x] = A;
}
```

Inserindo um elemento na posição X

O(n)




```
insere(f, 8818, 2);
```

```
void insere(struct lista *f,
                int A, int posicao){
    if(f->ultimo == MAX_ELEMENTOS-1){
        printf("lista cheia"); exit(1);
    }
    for(int i = f->ultimo; i > x; i--){
        f->elemtos[i] = f->elemtos[i-1];
    }
    f->ultimo += 1;
    f->elementos[x] = A;
}
```

Exercício

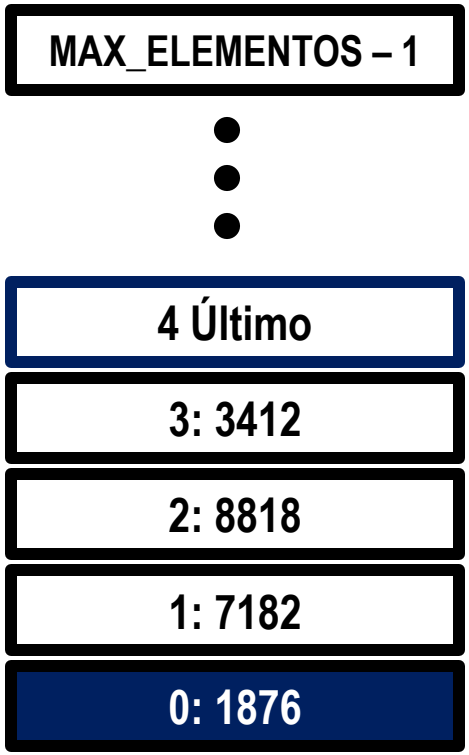
- Implemente as funções

```
int remover(struct lista *f,
            int elemento, int posicao);

int remover_final(struct lista *f);
```

- Qual a complexidade de cada função?
- O que elas fazem quando a lista está vazia?

Recuperando o elemento na posição X

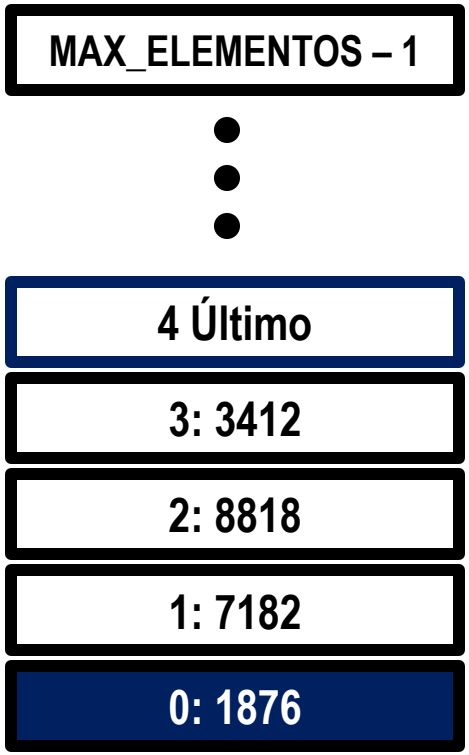
O(1)




```
struct lista {
     int elementos[MAX_ELEMENTOS];
     int ultimo;
}
void recupera(struct lista *f,
                    int posicao) {
     return f->elementos[posicao];
}
```

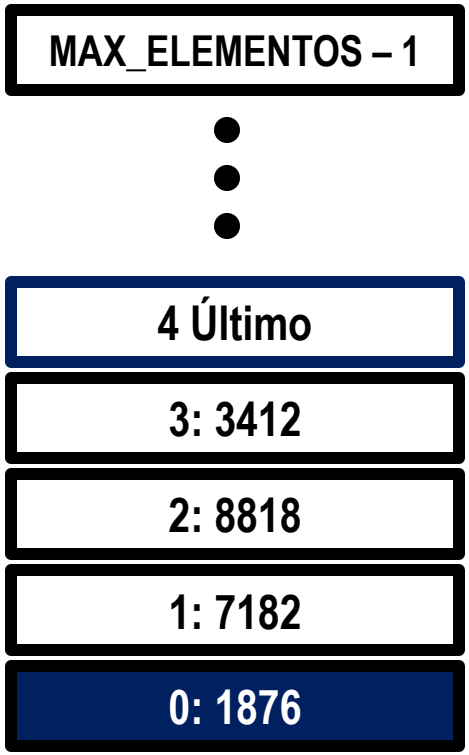
Recuperando o tamanho da lista

O(1)




```
struct lista {
     int elementos[MAX_ELEMENTOS];
     int ultimo;
}
void tamanho(struct lista *f) {
     return f->ultimo;
}
```

Destruir uma lista




```
struct fila {
     int elementos[MAX_ELEMENTOS];
     int ultimo;
}
void destroi(struct lista *f) {
     free(f);
}
```

Listas implementadas com arranjo

- Criar uma lista $O(1)$
- Inserir/remover no final $O(1)$
- Inserir/remover na posição X $O(N)$
- Recuperar o elemento na posição X $O(1)$
- Recuperar o tamanho da lista $O(1)$
- Destruir a lista $O(1)$

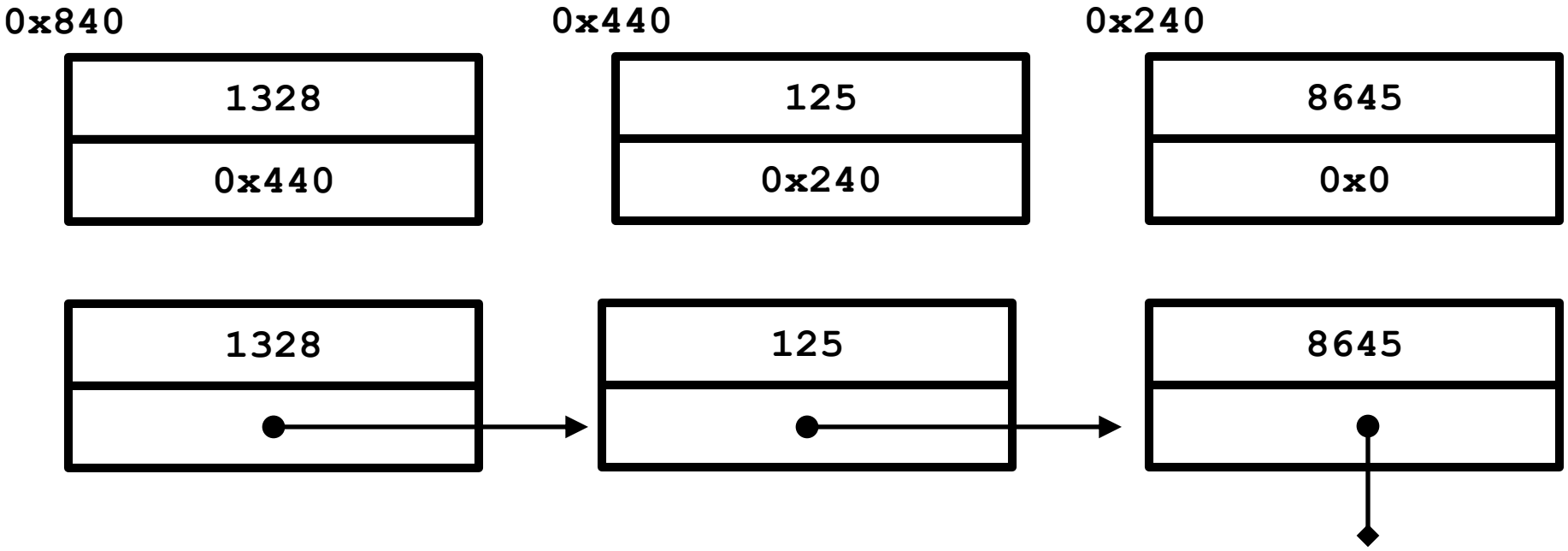
Listas implementadas com ponteiros

- Suportam as mesmas operações de criar, inserir/remover no final, inserir/remover no meio, recuperar o valor de um elemento, etc
- Possuem modo de operação diferente
- Elementos são armazenados em nós, e cada nó tem um apontador para o próximo nó na lista

Ilustração – nós de uma lista

```
struct no {
    int dado;
    struct no *proximo;
};
```

Exemplo de uma sequência de nós na memória:



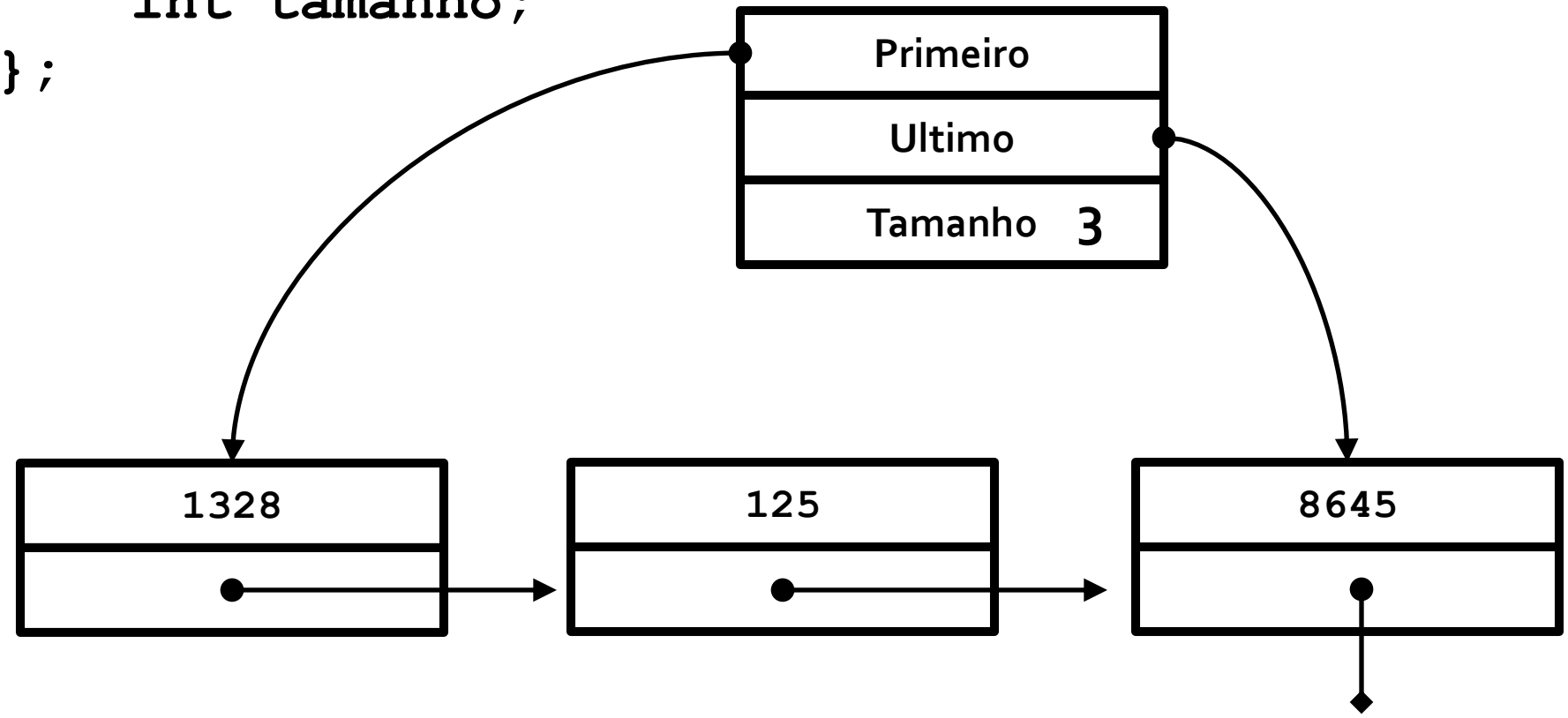
Implementação de lista com ponteiros

- Elementos são armazenados em nós, e cada nó aponta para o próximo nó da lista
- Precisamos guardar um apontador para o primeiro elemento (aponta pro resto da lista)

- *Podemos* guardar um apontador para o último elemento (para inserir/remover no final)
- *Podemos* guardar o tamanho da lista num inteiro

Ilustração – lista

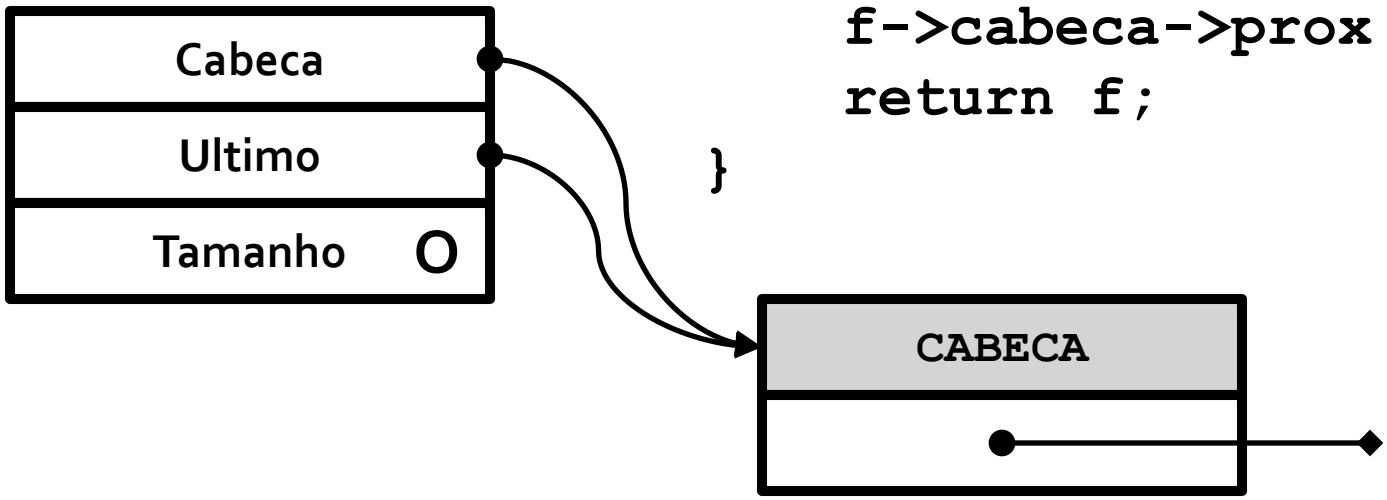
```
struct lista {
    struct no *primeiro;
    struct no *ultimo;
    int tamanho;
};
```



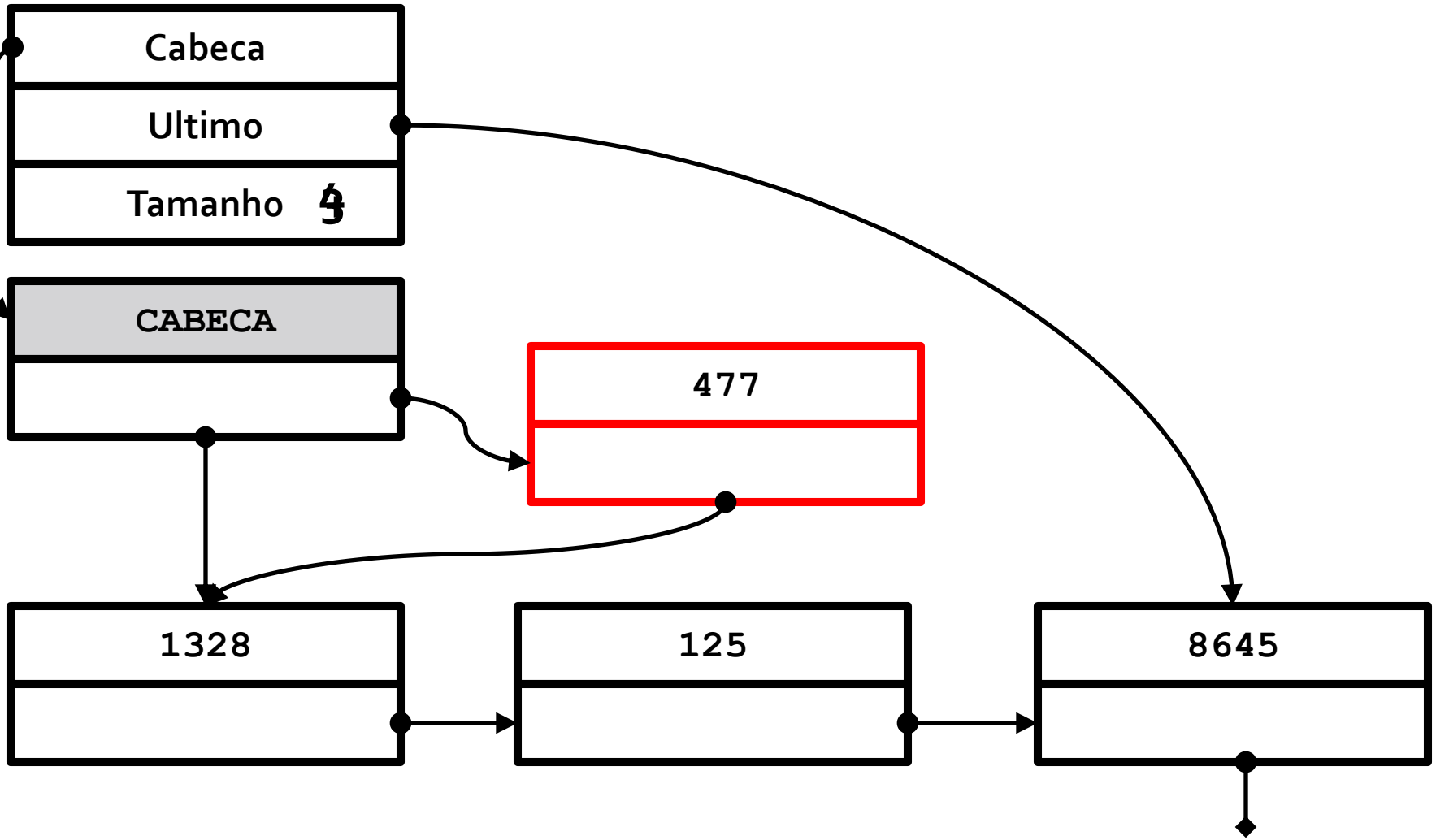
Criando uma lista

O(1)

```
struct lista * cria() {
    struct lista *f;
    f = malloc(sizeof(...));
    if(!f){ exit(1); }
    f->cabeca = malloc(...);
    if(!f->primeiro) { exit(1); }
    f->ultimo = f->primeiro;
    f->tamanho = 0;
    f->cabeca->prox = NULL;
    return f;
}
```



Inserindo no início

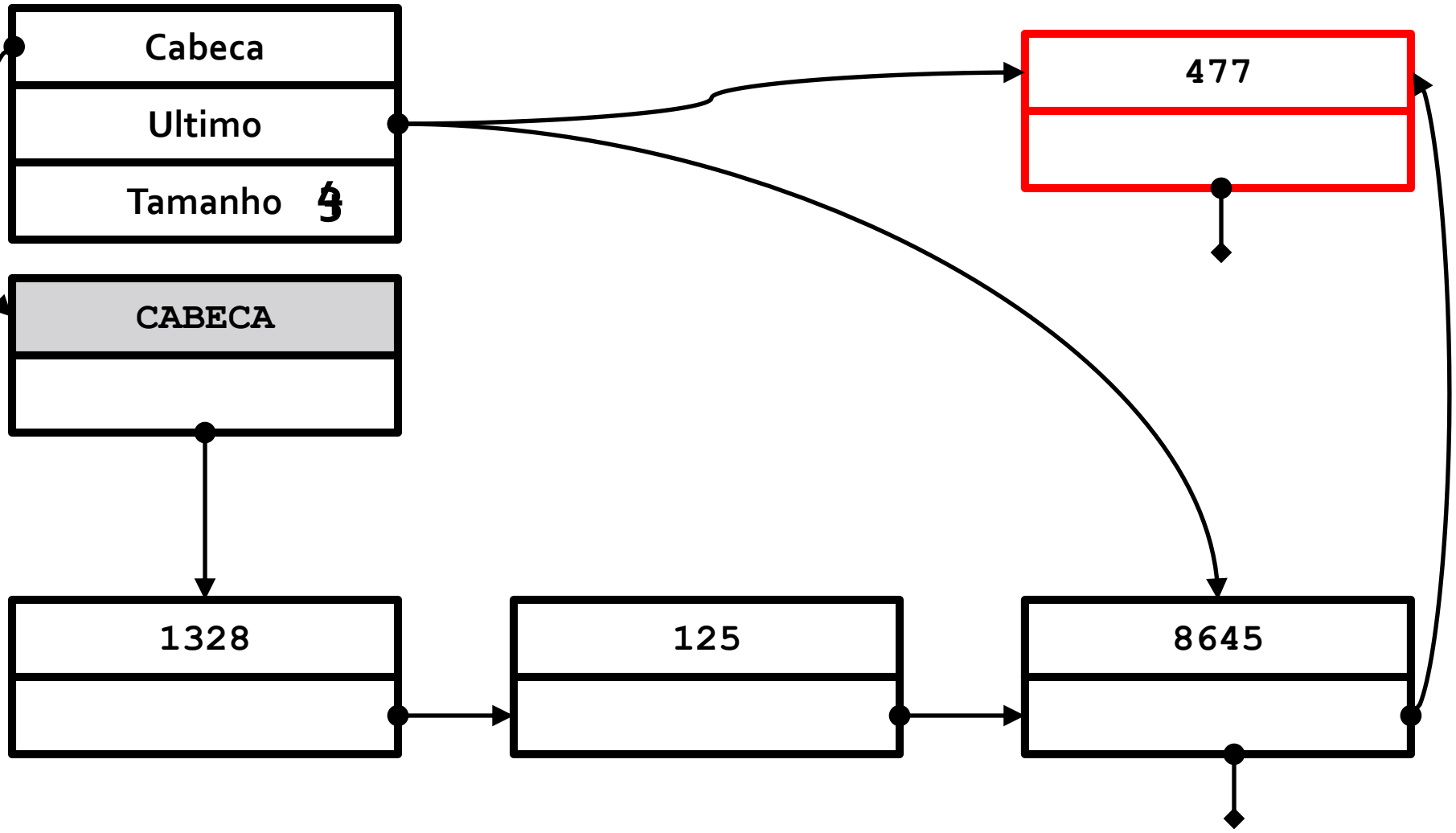


Inserindo no início

```
void insere_inicio(struct lista *f, int A) {
    struct no *novo;
    novo = malloc(sizeof(struct no));
    if(!novo) { perror(NULL); exit(EXIT_FAILURE); }
    novo->dado = A;
    novo->prox = f->cabeca->prox;
    /* NOTA: se a lista estiver vazia, novo->prox
     * fica NULL por que f->primeiro é o nó cabeça.
     * se nó cabeça tem que testar se a lista está
     * vazia e tratar caso especial. */
    f->cabeca->prox = novo;
    f->tamanho += 1;
}
```

O(1)

Inserindo no final

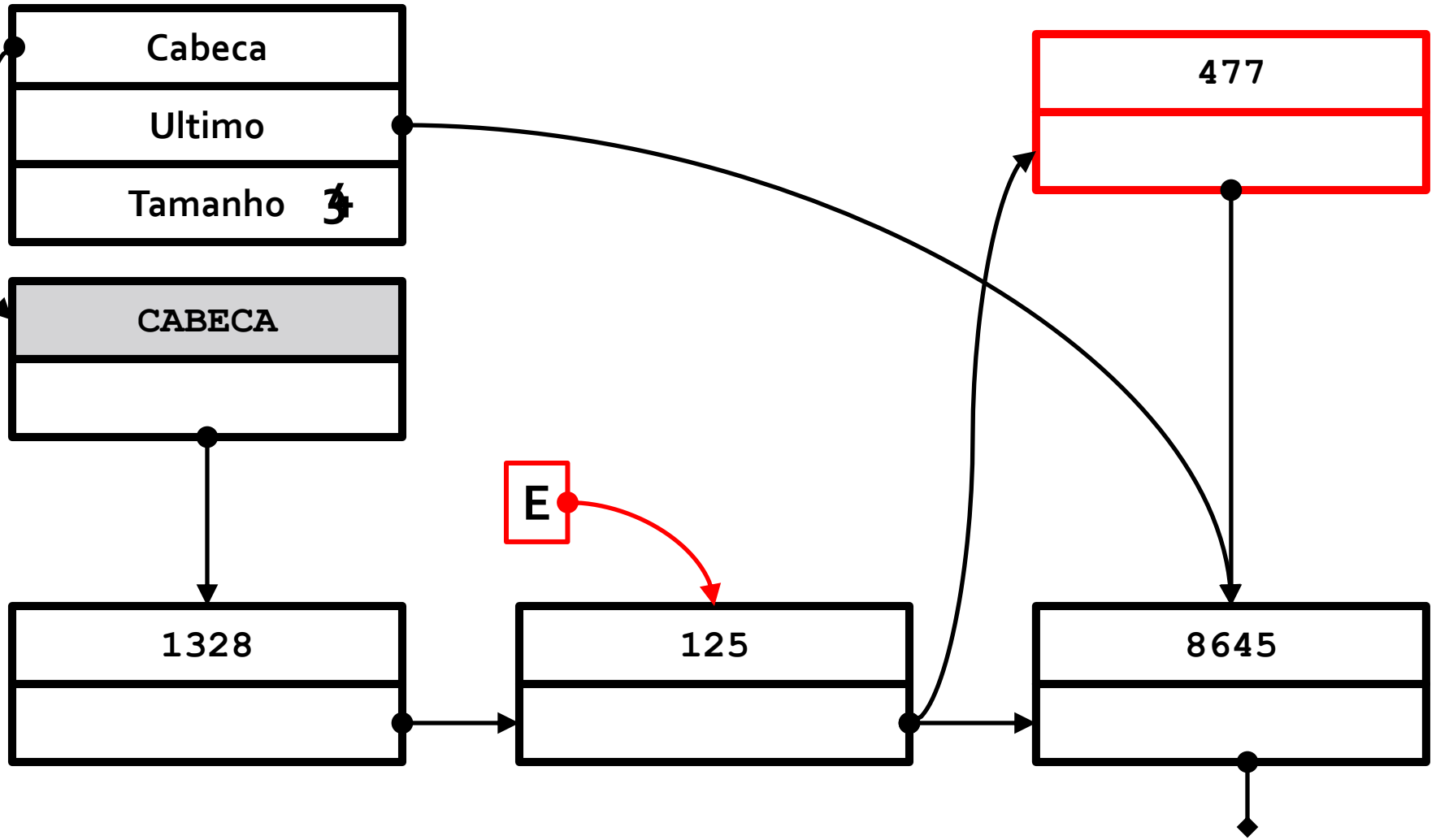


Inserindo no final

```
void insere_final(struct lista *f, int A) {
    struct no *novo = malloc(sizeof(struct no));
    if(!novo) { perror(NULL); exit(EXIT_FAILURE); }
    novo->dado = A;
    novo->prox = NULL;
    struct no *ultimo = f->ultimo;
    f->ultimo->prox = novo;
    /* ultimo->prox = novo; */
    f->ultimo = novo;
    f->tamanho += 1;
}
```

O(1)

Inserindo após um nó E



Inserindo após um nó E

```
void insere_atras(struct lista *f, int A,
                  struct no *E) {
    struct no *novo = malloc(sizeof(struct no));
    if(!novo) { perror(NULL); exit(EXIT_FAILURE); }
    novo->dado = A;
    novo->prox = E->prox;
    E->prox = novo;



}
```

O(1)

Equivalência

- Os tres métodos de inserção mostrados (no início, no final e após um elemento E) são equivalentes
  - Inserção no início é inserção após o **`f->cabeca`**
  - Inserção no final é inserção ápós o **`f->ultimo`**
  - Inserção após um nó E

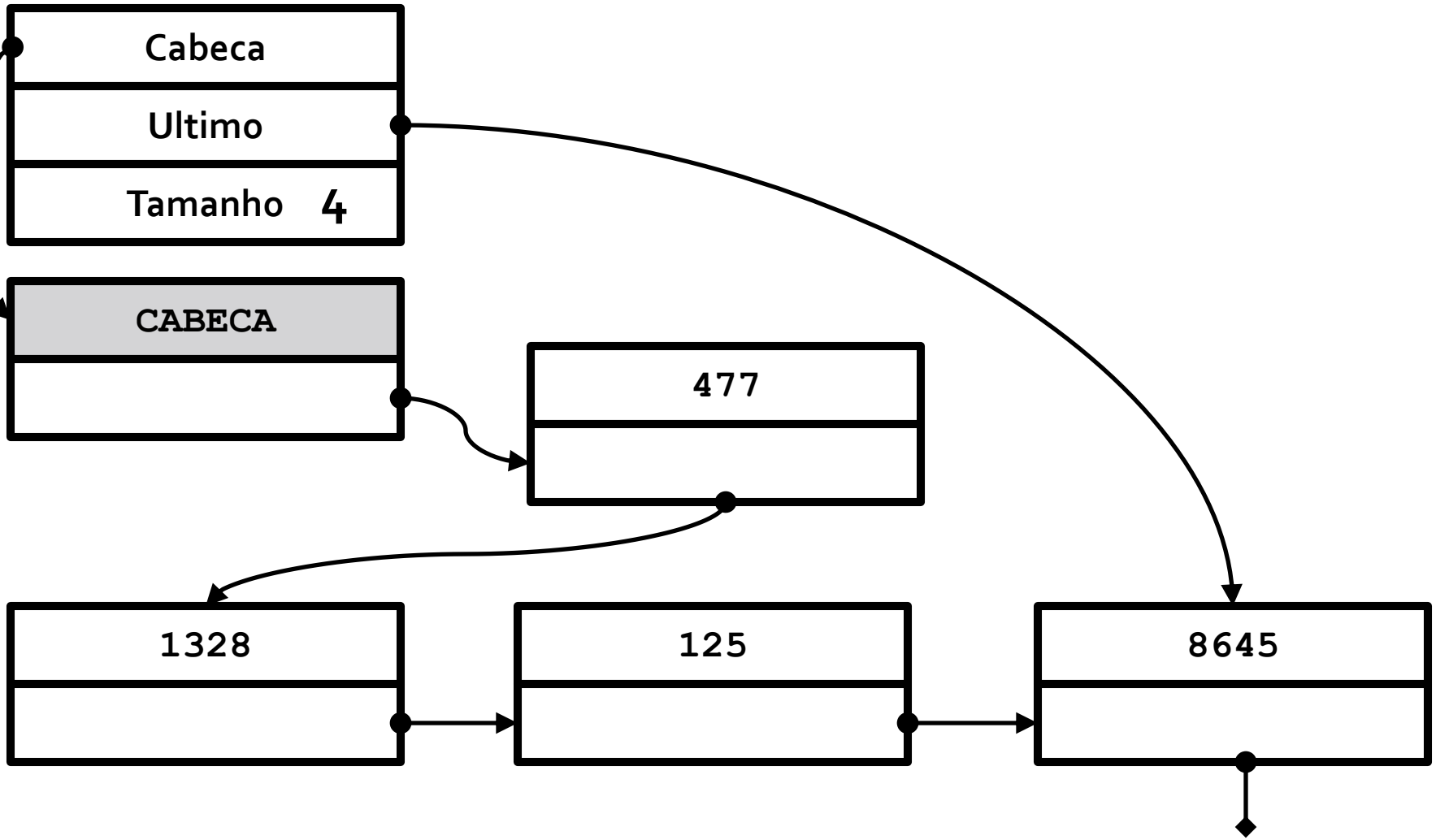
Recuperando o nó antecessor da posição X

- Tem de seguir os ponteiros **prox** X vezes

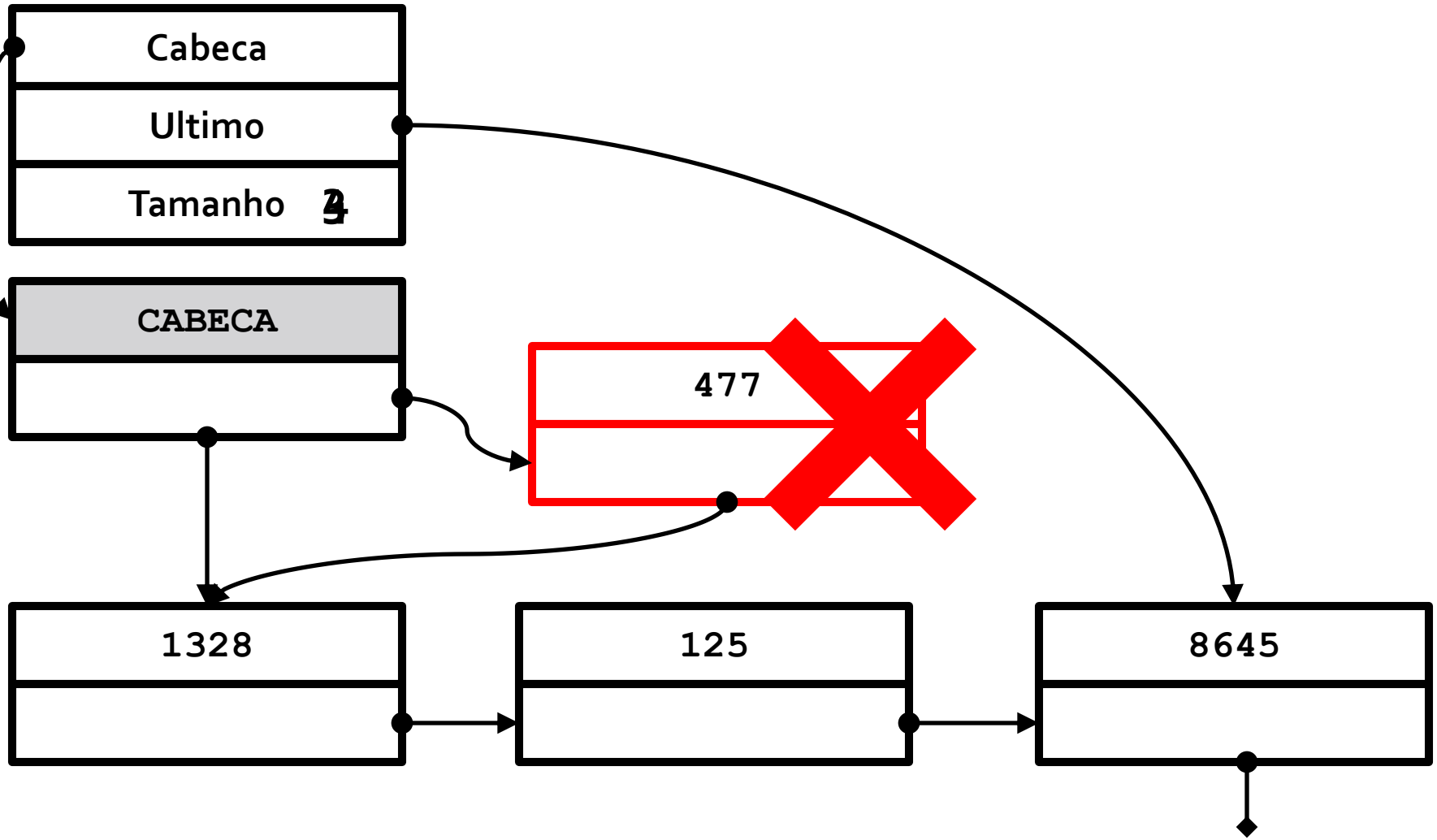
```
struct no * no_antecessor(struct lista *f, int pos){
    struct no *ante = f->cabeca;
    int i = 0;
    while(i < pos) {
        ante = ante->prox;
        i += 1;
    }
    return ante;
}
```

$O(X)$

Removendo no início

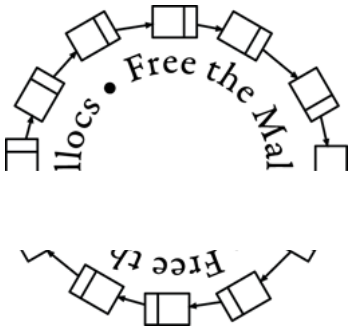


Removendo no início



Removendo no início

```
int remove_inicio(struct lista *f) {
    struct no *primeiro = f->cabeca->prox;



    f->cabeca->prox = primeiro->prox;
    f->tamanho -= 1;
    int dado = primeiro->dado;

    return dado;
}
```



O(1)

Destruindo uma lista

- Temos que destruir todos os nós da lista

```
void destroi(struct lista *f) {
     struct no *liberar;
     while(f->cabeca->prox) {
          liberar = f->cabeca;
          f->cabeca = f->cabeca->prox;
          free(liberar);
     }

     free(f);
}
```

Listas implementadas com ponteiros

- Criar uma lista $O(1)$
- Inserir/remover no final $O(1)$
- Inserir/remover atrás de antecessor $O(1)$
- Encontrar o anterecessor de X $O(N)$
- Recuperar o elemento na posição X $O(N)$
- Recuperar o tamanho da lista* $O(1)$
- Destruir a lista $O(N)$

Arranjos vs. ponteiros

ARRANJOS

- Limite máximo de número de elementos
- Acesso direto ao n-ésimo elemento da lista $O(1)$
- Inserção/remoção no meio implica realocar todos os elementos seguintes $O(n)$

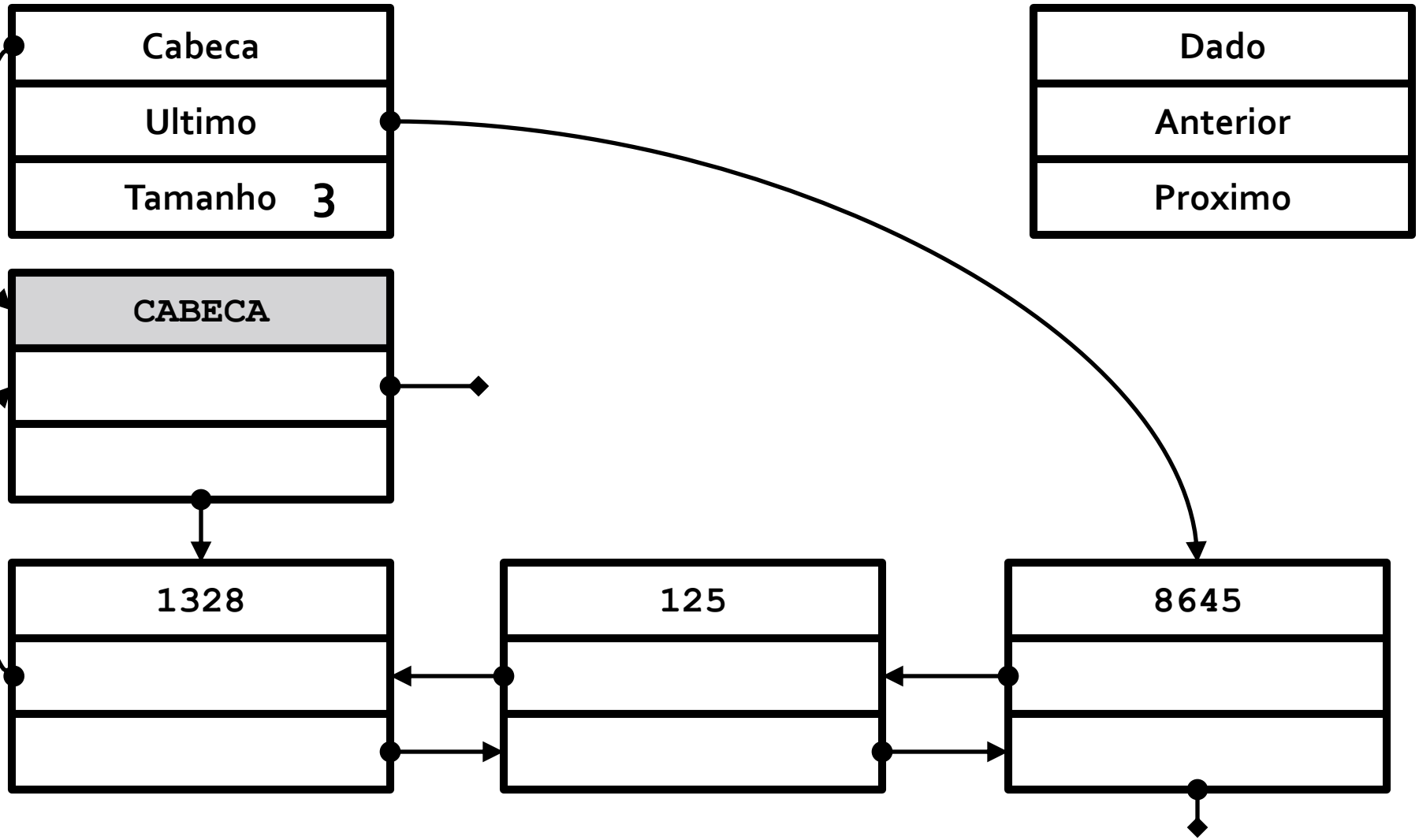
PONTEIROS

- Sem limite de elementos (enquanto tiver memória)
- Acesso ao n-ésimo elementos requer seguir vários apontadores $O(n)$
- Inserção/remoção com antecessor é fácil $O(1)$
- Ponteiros usam memória

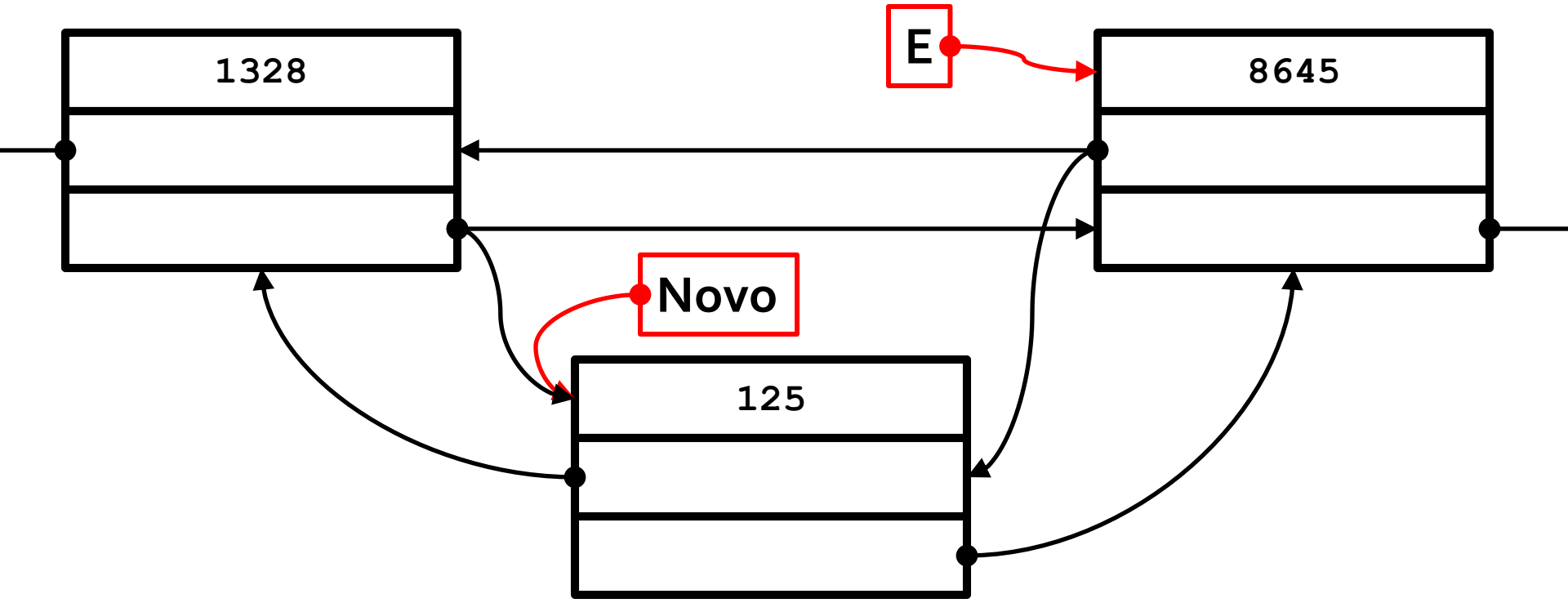
Variações sobre lista

- Na implementação de lista vista, só podemos inserir *atrás* de um nó *antecessor*. Por quê?
  - Por que temos de modificar o ponteiro prox do antecessor
- Como fazer uma lista onde podemos adicionar elementos na frente de um nó *sucessor*?

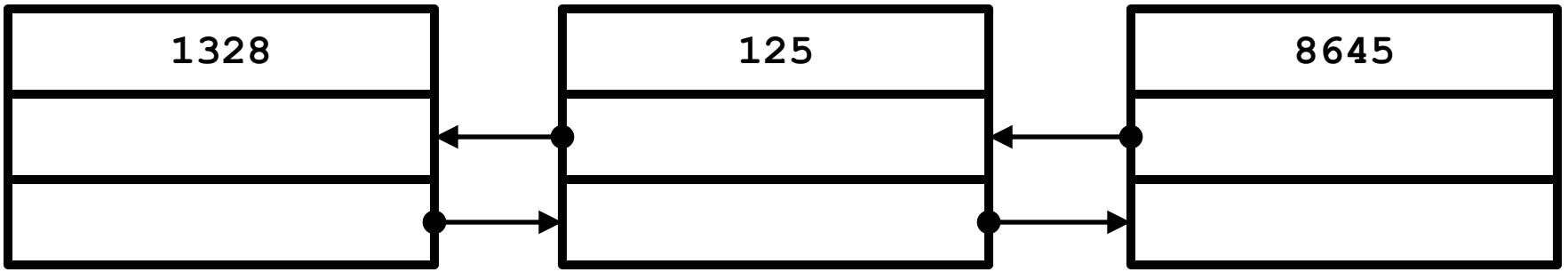
Listas duplamente encadeadas



Inserindo um elemento antes de um nó E

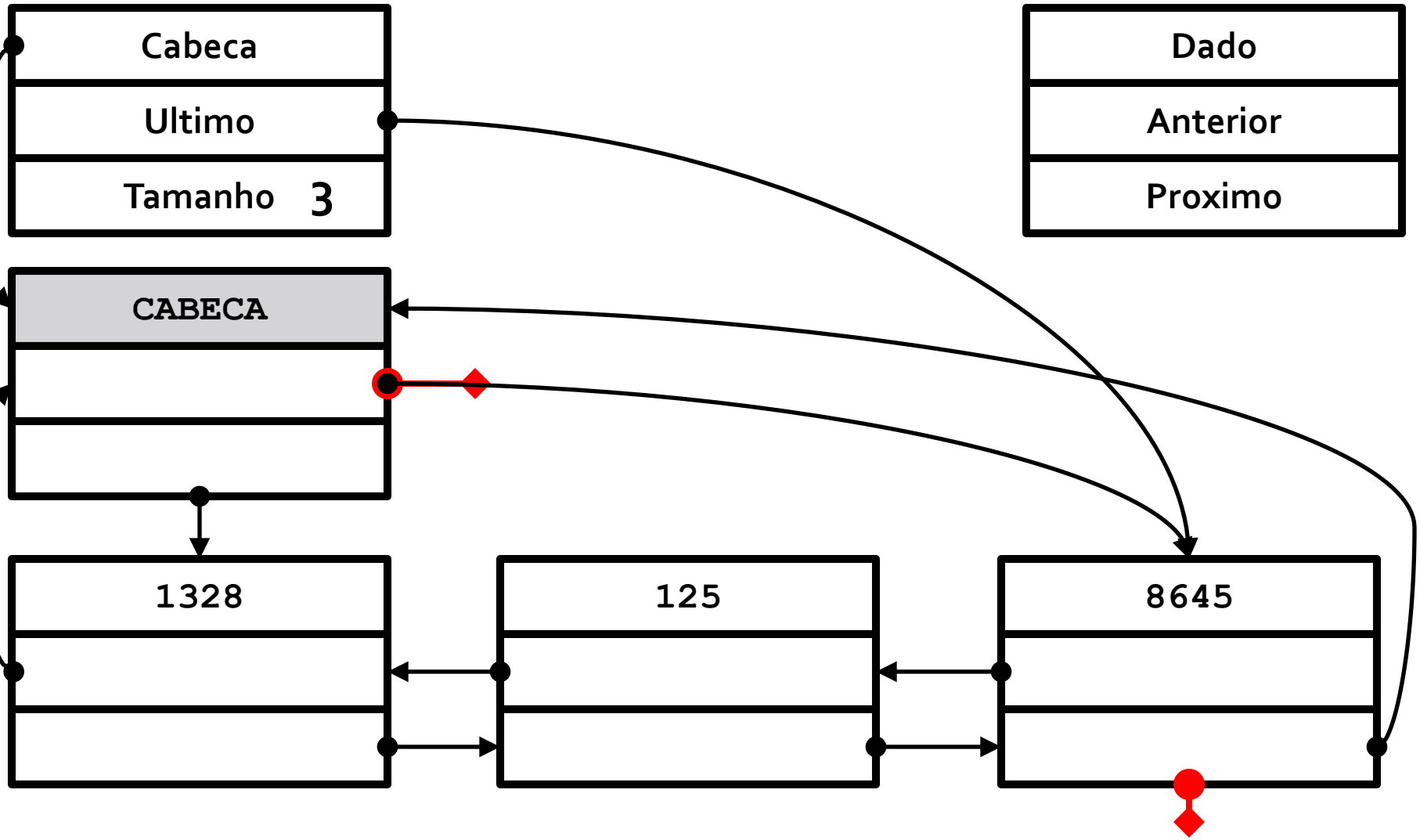


Inserindo um elemento antes de um nó E




```
void insere_antes(struct listadupla *f, int A,
                  struct no *E) {
    struct no *novo = malloc(sizeof(struct no));
    if(!novo) { perror(NULL); exit(EXIT_FAILURE); }
    novo->dado = A;
    novo->prox = E;
    novo->ante = E->ante;
    E->ante->prox = novo;
    E->ante = novo;
}
```

Listas circulares

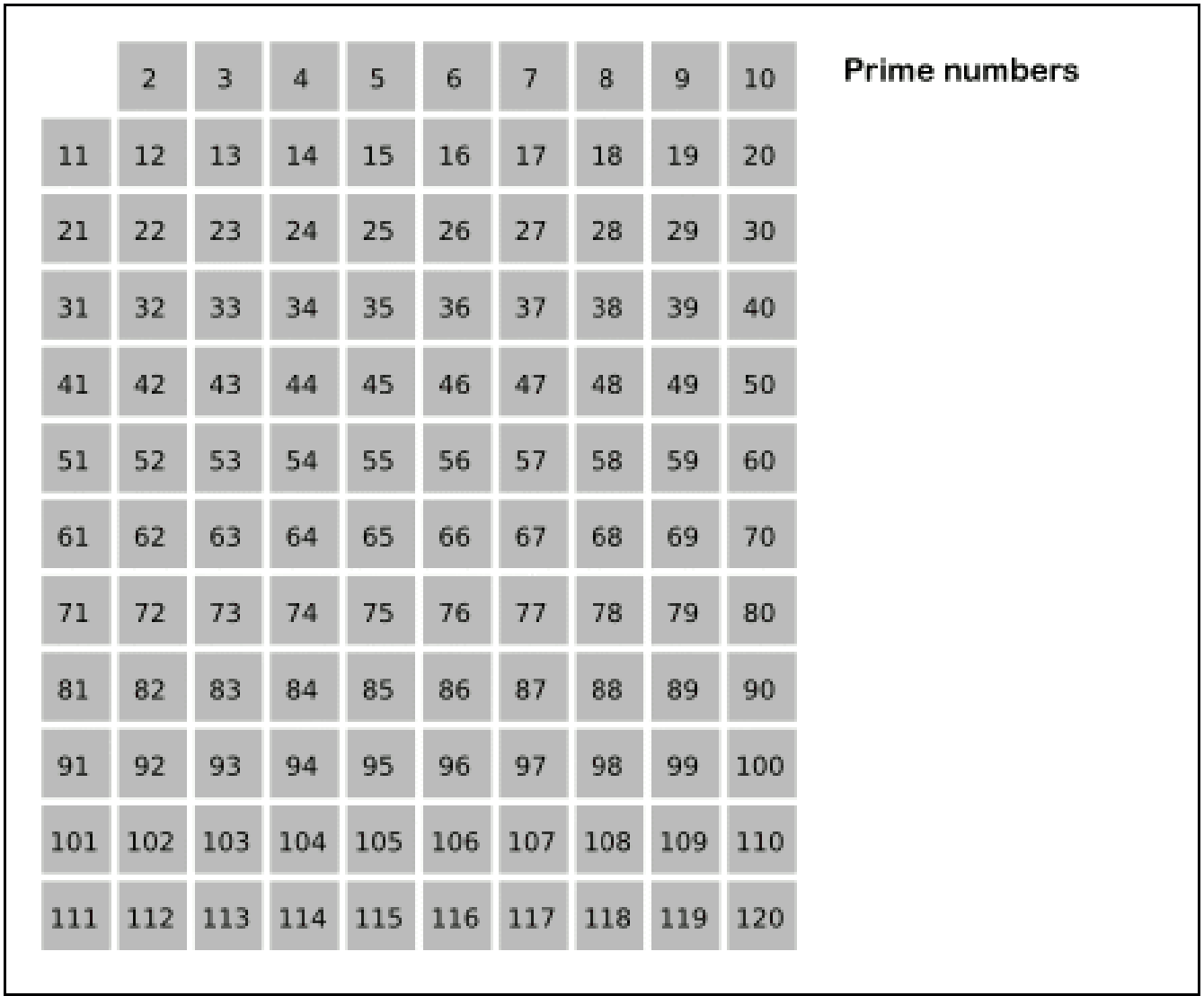


Exercícios

- Implemente a função
  ```
  int remove_final(struct lista *f);
  ```
- Implemente a função
  ```
  int remove_atras(struct lista *f, struct no *E);
  ```

- Suas funções funcionam quando a lista está vazia? E quando a removemos o primeiro ou último elementos?
- Qual a complexidade de suas funções? Elas devem ter custo $O(1)$.

Exercícios: Crivo de Erastotenes



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Listas de elementos genéricos

- As listas que implementamos até agora só armazenavam elementos do tipo **`int`**
- E se quisermos armazenar outros tipos de dados, ou armazenar estruturas complexas dentro de uma lista?

- Armazenar ponteiros para **`void`**
  - Eles podem apontar para *qualquer* tipo de dado

Listas de elementos genéricos

- Todos os algoritmos para criar e destruir uma lista, bem como para remover e inserir elementos continuam os mesmos
- A única mudança é no tipo do parâmetro recebido pelos métodos de inserção e no tipo do retorno dos métodos de remoção

Listas de elementos genéricos

```
struct no {
     struct no *ante;
     struct no *prox;
     void *dado;
}
void insere_inicio(struct listadupla *f, void *d);
```

Listas de elementos genéricos

```
struct no {
     struct no *ante;
     struct no *prox;
     void *dado;
}
void* recupera(struct listaduple *f, int posicao);
```

Exercícios

- Implemente pilha e fila com ponteiros.