

## Licenciatura em Engenharia Informática – 2023/24

# Programação

## 4: Gestão Dinâmica de Memória

### 4.3: Outras Estruturas Dinâmicas

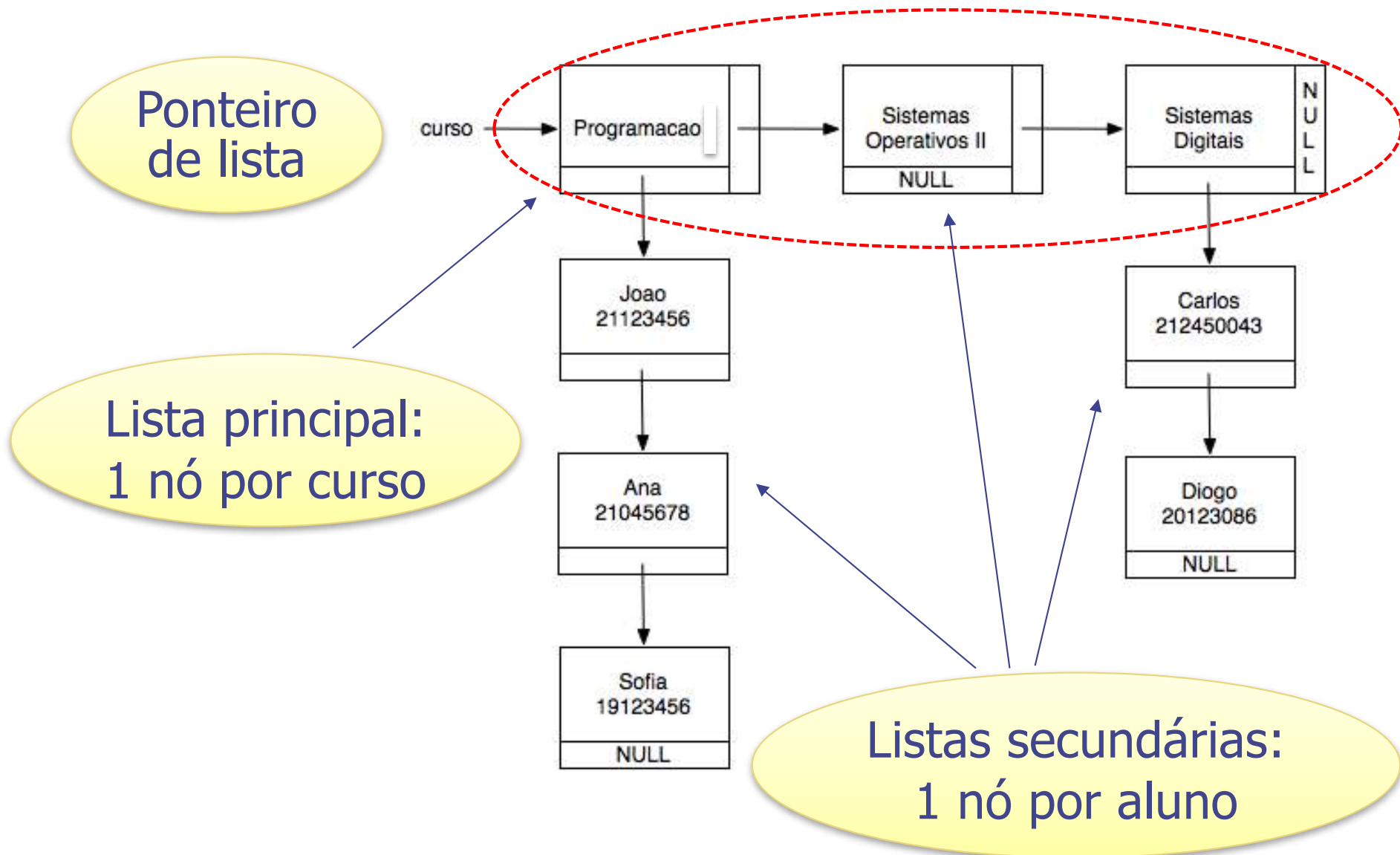
Francisco Pereira (xico@isec.pt)

---

- A organização de uma estrutura dinâmica baseada em listas ligadas é flexível:
  - Informação a armazenar (pode ser heterogénea)
  - Operações a efetuar
- Alguns exemplos:
  1. Lista de listas
  2. Array de listas
  3. Lista duplamente ligada

- Armazenar informação sobre a estrutura curricular de um curso:
  - Várias disciplinas
  - Cada disciplina tem vários alunos inscritos
- Porquê recorrer a uma representação dinâmica?
  - Número de disciplinas pode variar
  - Número de alunos inscritos em cada disciplina pode variar

# Representação dinâmica do curso




# Definição de tipos

```
typedef struct dados disc, *pdisc;  
typedef struct pessoa aluno, *paluno;
```

```
struct dados {  
    char nome[50];  
    int ano, semestre, n_alunos;  
    paluno lista;  
    pdisc prox;  
};
```

Nós da lista  
principal



```
struct pessoa {  
    char nome[100];  
    char numero[15];  
    paluno prox;  
};
```

Nós da lista  
secundária



1. Listagem total ou parcial
2. Adicionar informação:
  - Nova disciplina
  - Inscrição de um aluno numa disciplina do curso
3. Eliminar informação:
  - Retirar uma disciplina do plano curricular
  - Remover a inscrição de um aluno de uma disciplina

# Operação 1: Listagem completa

---

```
void escreve_tudo (pdisc curso) ;
```

- Argumento: Ponteiro para o início da estrutura dinâmica (para a 1ª disciplina)
- Estratégia a seguir:
  - Percorrer a lista de disciplinas até ao final. Em cada nó:
    - Escrever informação sobre a disciplina
    - Aceder à lista de alunos inscritos
    - Percorrer toda a lista de alunos escrevendo os seus nomes

# Operação 1: Listagem completa

---

```
void escreve_tudo(pdisc p) {
    paluno aux;

    while(p != NULL)
    {
        printf("%s\t%2dA\t%2dS\t%d Alunos\n",
               p->nome, p->ano, p->semestre, p->n_alunos);
        aux = p->lista;
        while(aux != NULL)
        {
            printf("%s\t%s\n", aux->nome, aux->numero);
            aux = aux->prox;
        }
        p = p->prox;
    }
}
```



## Operação 2: Criar uma nova disciplina

---

```
pdisc cria_disc(pdisc p, char *st, int ano, int s);
```

- Argumentos: Ponteiro para início da estrutura dinâmica, nome, ano e semestre da nova disciplina
  - Devolve ponteiro para início da estrutura dinâmica modificada
- 
- Estratégia a seguir:
    - Criar e preencher um nó com informação relativa à nova disciplina.
    - Inserir o novo nó no início da lista de disciplinas.

## Operação 2: Criar uma nova disciplina

---

```
pdisc cria_disc(pdisc p, char *st, int ano, int s){
    pdisc novo;

    novo = malloc(sizeof(disc));
    if(novo == NULL) return p;

    strcpy(novo->nome, st);
    novo->ano = ano;
    novo->semestre = s;
    novo->n_alunos = 0;
    novo->lista = NULL;

    novo->prox = p;
    p = novo;

    return p;
}
```

# Operação 3: Inscrever um aluno

---

```
void adic_al(pdisc p, char *n_d, char *n_al, char *id);
```

- Argumentos: Ponteiro para a estrutura dinâmica, nome da disciplina, nome e número do aluno
- Estratégia a seguir:
  - Percorrer a lista principal até encontrar o nó com a disciplina pretendida
  - Se a disciplina existir:
    - Criar e preencher nó com informação relativa ao novo aluno
    - Inserir o nó no início da lista de alunos da disciplina em causa

## Operação 3: Inscrever um aluno

```
void adic_al(pdisc p, char *n_d, char *n_al, char *id){
    paluno novo;

    while(p != NULL && strcmp(p->nome, n_d) != 0)
        p = p->prox;

    if(p != NULL)
    {
        novo = malloc(sizeof(aluno));
        if(novo == NULL) return;

        strcpy(novo->nome, n_al);
        strcpy(novo->numero, id);
        novo->prox = p->lista; /*insere no inicio*/
        p->lista = novo;
        p->n_alunos++;
    }
}
```

# Operação 4: Eliminar uma disciplina

---

```
pdisc elimina_disc(pdisc p, char *nome);
```

- Argumentos: Ponteiro para a estrutura dinâmica e nome da disciplina a eliminar
  - Devolve ponteiro para estrutura dinâmica modificada
- 
- Estratégia a seguir:
    - Percorrer a lista principal até encontrar o nó com a disciplina pretendida.
    - Se a disciplina existir:
      - Eliminar todos os alunos associados à disciplina
      - Retirar o nó da disciplina da lista principal

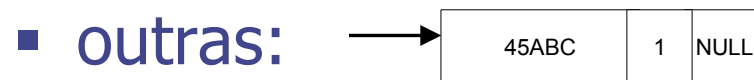
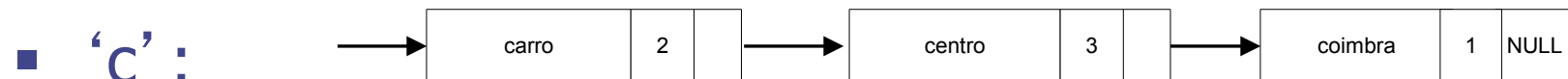
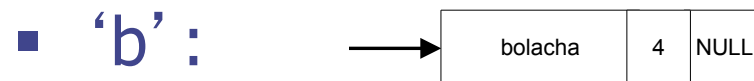
## Exemplo 2: Conjunto de palavras

---

- Representação dinâmica:
  - Número de palavras desconhecido
    - Só letras minúsculas
  - Varia ao longo do tempo
- Acesso eficiente:
  - Conjunto ordenado
  - Divisão em função da 1ª letra

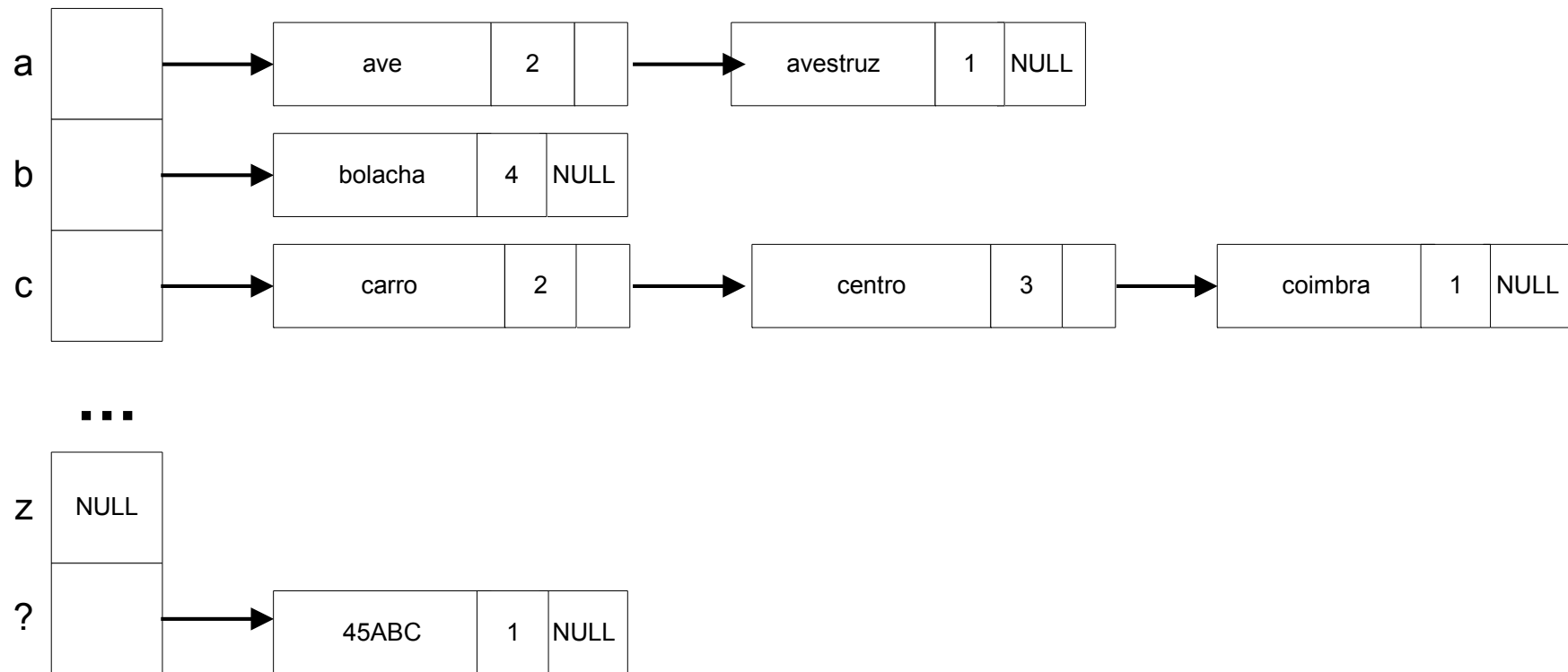
# Representar um conjunto de palavras

- Conjunto de 27 listas ligadas ordenadas:



# Agrupar ponteiros de lista

- Como?
  - Array de ponteiros:





# Definição de tipos

---

```
typedef struct palavra no, *pno;  
  
struct palavra{  
    char pal[50];  
    int conta;  
    pno prox;  
};
```

```
#define N 27  
  
int main(){  
  
    pno texto[N] = {NULL};  
    ...  
}
```

1. Consulta de uma palavra, verificando quantas vezes já surgiu
2. Atualizar a ED com uma nova ocorrência de uma palavra

# Operação 1: Consulta de uma palavra

---

```
int consulta_palavra(pno d[], char *pal);
```

- Argumentos: Endereço do array de ponteiros e palavra a pesquisar
  - Devolve número de ocorrências
- 
- Estratégia a seguir:
    - Determinar qual a lista a pesquisar
    - Percorrer a lista até encontrar o local onde a palavra deverá estar
      - Se existir, devolver o número de ocorrências
      - Se não existir, devolver 0

# Operação 1: Consulta de uma palavra

```
int consulta_palavra(pno d[], char *pal){
    pno aux;
    int index;

    if(*pal >= 'a' && *pal <= 'z')
        index = *pal - 'a';
    else
        index = N-1;

    aux = d[index];

    while(aux != NULL && strcmp(aux->pal, pal) < 0)
        aux = aux->prox;

    if(aux == NULL || strcmp(aux->pal, pal) > 0)
        return 0;
    else
        return aux->conta;
}
```

## Operação 2: Atualizar ED com palavra

```
void atualiza_ED(pno d[], char *pal);
```

- Argumentos: Endereço do array de ponteiros e palavra a atualizar
- Estratégia a seguir:
  - Determinar qual a lista a alterar
  - Percorrer a lista e verificar se a palavra já existe:
    - Se existir, atualiza contador
    - Se não existir, adiciona um novo nó no local adequado
      - Caso particular: Inserção no início da lista

## Operação 2: Atualizar ED com palavra

- Função auxiliar para criar e preencher um novo nó com uma palavra passada por argumento:

```
pno cria_preenche(char *pal)
{
    pno novo;

    novo = malloc(sizeof(novo));
    if(novo == NULL)
        return NULL;

    novo->conta = 1;
    strcpy(novo->pal, pal);
    novo->prox = NULL;
    return novo;
}
```

## Operação 2: Atualizar ED com palavra

```
void atualiza_ED(pno d[], char *pal){
    pno atual, ant, novo;
    int index;

    if(*pal >= 'a' && *pal <= 'z')
        index = *pal - 'a';
    else
        index = N-1;

    atual = d[index];
    ant = NULL;

    while(atual!=NULL && strcmp(atual->pal, pal)<0)
    {
        ant = atual;
        atual = atual->prox;
    }

    ...
}
```

## Operação 2: Atualizar ED com palavra

```
if(atual!=NULL && strcmp(atual->pal, pal) == 0)
{
    atual->conta++;
}
else
{
    novo = cria_preenche(pal);
    if(novo == NULL)
        return;
    if(ant == NULL)
    {
        novo->prox = atual;
        d[index] = novo;
    }
    else
    {
        novo->prox = atual;
        ant->prox = novo;
    }
}
}
```

Palavra  
já existe

Nova  
palavra

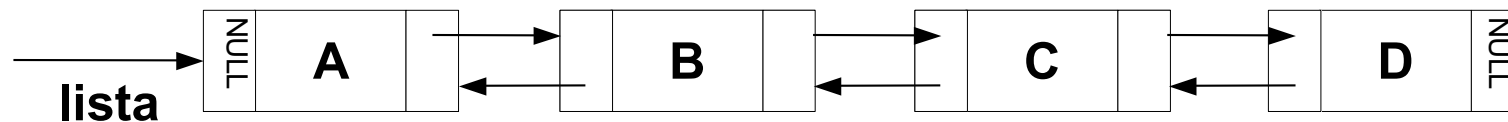
Inserção no início

Inserção no meio/final



## Exemplo 3: Listas duplamente ligadas

- Cada nó de uma lista duplamente ligada tem 2 ponteiros:
  - Um para o próximo nó
  - Um para o nó anterior



- Vantagens:
  - Lista percorrida nas duas direções
  - Acesso mais eficiente
- Desvantagens:
  - Nós ocupam mais espaço
  - Operações de manipulação da lista mais complexas

- Armazenar numa lista a informação relativa aos habitantes de uma rua:
  - Cada nó contém o nome da família e número de elementos
  - Vizinhos encontram-se em nós adjacentes

```
typedef struct elemento no, *pno;  
  
struct elemento{  
    char nome[100];  
    int numero;  
    pno ant;  
    pno prox;  
};
```

1. Adicionar uma nova família à lista.
2. Retirar uma família à lista.
3. Procurar os vizinhos de uma família.

# Operação 1: Adicionar um elemento

---

```
pno insere(pno p, int pos, pno novo);
```

- Argumentos: Ponteiro para início da lista, posição de inserção e ponteiro para nó a inserir (já preenchido)
- Devolve ponteiro para início da lista modificada
- Estratégia a seguir:
  - Tratar primeiro casos particulares: Lista vazia / Inserção no início
  - Caso geral: Encontrar elemento que vai ficar antes do novo nó

# Operação 1: Adicionar um elemento

```
pno insere(pno p, int pos, pno novo) {
    pno aux;
    int i;

    if (p == NULL) p = novo;
    else if (pos == 1)
    {
        novo->prox = p;
        p->ant = novo;
        p = novo;
    }
    else
    {
        aux = p;
        for (i=1; aux->prox!=NULL && i<pos-1; i++)
            aux = aux->prox;

        novo->prox = aux->prox;
        if (aux->prox != NULL)
            aux->prox->ant = novo;
        novo->ant = aux;
        aux->prox = novo;
    }
    return p;
}
```

## Operação 2: Eliminar um elemento

---

```
pno elimina(pno p, char *s);
```

- Argumentos: Ponteiro para início da lista e nome da família
- Devolve ponteiro para início da lista modificada
- Estratégia a seguir:
  - Encontrar elemento que vai ser eliminado
  - Se for encontrado, reorganizar a lista
    - Caso particular: primeiro elemento

## Operação 2: Eliminar um elemento

```
pno elimina(pno p, char *s){
    pno aux;

    aux = p;
    while(aux!=NULL && strcmp(aux->nome, s)!=0)
        aux = aux->prox;

    if(aux == NULL) return p;

    if(aux == p) /* o no' a eliminar e' o primeiro */
    {
        p = aux->prox;
        if(p != NULL) /* o no' nao e' o ultimo */
            p->ant = NULL;
    }
    else
    {
        aux->ant->prox = aux->prox;
        if(aux->prox != NULL) /* o no' nao e' o
ultimo */
            aux->prox->ant = aux->ant;
    }
    free(aux);
    return p;
}
```



## Operação 3: Pesquisar informação

---

```
void vizinhos(pno p, char *s);
```

- Argumentos: Ponteiro para início da lista e nome da família
- Escreve no monitor o nome dos vizinhos

## Operação 3: Pesquisar informação

```
void vizinhos(pno p, char *s){
    while(p != NULL && strcmp(p->nome, s) != 0)
        p = p->prox;
    if(p != NULL)
    {
        if(p->prox != NULL)
            printf("Posterior: %s\n", p->prox->nome);
        if(p->ant != NULL)
            printf("Anterior: %s\n", p->ant->nome);
    }
    else
        printf("A familia %s nao existe\n", s);
}
```

Criar uma lista ligada simples com elementos do tipo *no*

```
typedef struct info no, *pno;  
  
struct info{  
    char letra;  
    pno prox;  
};
```

Implementar uma função com a seguinte funcionalidade

- Recebe referências para duas listas ligadas L1 e L2
- As listas não estão vazias
- Retira o primeiro elemento de L1.
- Adiciona-o ao início de L2.



# Função A: Primeira Tentativa

---

```
void transfere(pno p1, pno p2){  
    pno aux;  
  
    if(p1 == NULL) return;  
  
    aux = p1;  
    p1 = aux->prox;  
    aux->prox = p2;  
    p2 = aux;  
}
```

Como ficam  
organizadas  
as listas?

# Primeira Tentativa - FALHADA

---



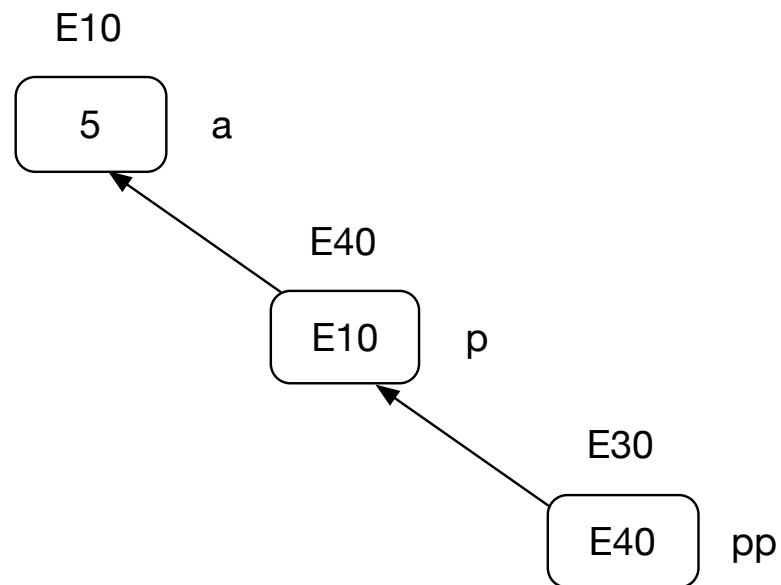
Porquê?

# Ponteiros para Ponteiro

Um ponteiro é uma variável

Tem um endereço

Esse endereço pode ser guardado num outro ponteiro



```
int a = 5;  
  
int *p;  
p = &a;  
  
int **pp;  
pp = &p;
```

# Função A: Solução Correta

---

```
void transfere(pno *p1, pno *p2) {  
    pno aux;  
  
    if(*p1 == NULL) return;  
  
    aux = *p1;  
    *p1 = aux->prox;  
    aux->prox = *p2;  
    *p2 = aux;  
}
```

**Argumentos:**  
referências para os dois  
ponteiros de lista



## Implementar a seguinte função

```
pno vogais(pno *p);
```

- Recebe endereço de um ponteiro de lista contendo nós do tipo *no*
- Transfere para uma nova lista todos os nós com vogais
- Devolve ponteiro para o primeiro elemento dessa nova lista

## Função auxiliar para verificar se um caracter armazena uma vogal

```
int eVogal(char c) {  
    return c=='A' || c=='E' || c=='I' || c=='O' || c=='U';  
}
```

# Função B

```
pno vogais(pno *p) {  
    pno v = NULL, lista=*p, ant=NULL, atual=*p;  
  
    while(atual != NULL) {  
        if(eVogal(atual->letra) == 1) {  
            if(ant == NULL)  
                lista = atual->prox;  
            else  
                ant->prox = atual->prox;  
            atual->prox = v;  
            v =atual;  
            if(ant == NULL) atual = lista;  
            else atual = ant->prox;  
        }  
        else{  
            ant = atual; atual = atual->prox;  
        }  
    }  
    *p = lista;  
    return v;  
}
```