# Índice

## Capítulo 3 – Estrutura de Dados sequencial com armazenamento não sequencial

# Capítulo 3 – Estrutura de Dados sequencial com armazenamento não sequencial

## 1. Listas ligadas – armazenamento não sequencial

A ideia é colocar em cada registo de uma ***EAD Lista*** um ponteiro que indique a localização do próximo registo. Esta situação é ilustrada pela *Figura 1*.

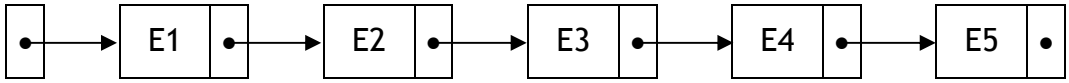



**Figura 1 - Representação esquemática duma lista ligada.**

Portanto uma lista ligada é construída usando registos, em que cada registo contém a informação armazenada (elemento) e um ponteiro a indicar onde encontrar o próximo registo (e elemento) da lista.

As características fundamentais das listas são as seguintes:

- Tamanho: variável
- Acesso: sequencial.

### 1.1. Nodos e ligações

É comum designar os registos por *nodos* (ou nós) e os ponteiros por *ligações*. Como a *ligação* em cada nodo indica onde encontrar o próximo elemento, pode-se usar o nome *Prox* para designar a *ligação*. Traduzindo estas definições para a linguagem C obtemos:

```
struct  Nodo {
        Info      Elemento;
        struct   Nodo   *Prox;
};
typedef  struct  Nodo  *PNodo;
```

Note-se que se está perante um problema de circularidade nesta declaração. O campo **Prox* é do tipo *Nodo* e faz parte do registo *Nodo*. A linguagem C requer que seja identificada a estrutura logo no início, para que se possa utilizar na declaração de *Prox*, tal como foi feito. Para contornar este problema de declarações circulares e apenas para o caso dos ponteiros, a linguagem C relaxa um pouco a regra da obrigatoriedade de declarar todo o identificador antes de este poder ser usado.

A razão da linguagem C poder relaxar as suas regras desta forma, e mesmo assim compilar eficientemente, é que todos os ponteiros ocupam a mesma quantidade de memória (a mesma quantidade de memória que um inteiro), independentemente do tipo a que ele se refere.

A partir daqui neste documento, usa-se a denominação Lista para se fazer referência a uma EAD Lista.

## 1.2. Início de uma Lista

Na lista ligada é usado o campo *Prox* para passar de um *Nodo* para o seguinte. Contudo, existe um pequeno problema: onde é que se encontra o início da lista?

Este problema pode ser ultrapassado se for usada uma variável estática para guardar a localização do primeiro nodo da Lista. Esta variável é uma variável ponteiro e é comum denominá-la por ***Lista***, ***Head*** (cabeça) ou simplesmente ***L***. Esta variável identifica completamente a Lista.

Quando começar a execução do programa pretende-se marcar a Lista como vazia. Esta tarefa é agora muito fácil. Sendo *L* uma variável estática, já existe quando a execução começar. Assim sendo, para marcar a lista como vazia é suficiente a atribuição seguinte:

```
L = NULL;
```

## 1.3. Fim da Lista

Encontrar o fim duma Lista é uma tarefa muito fácil. Cada *Nodo* contém uma *ligação* para o próximo *Nodo* da Lista. O campo associado à *ligação* do último *Nodo* da Lista não aponta para lado algum, donde é comum atribuir-lhe o valor *NULL*. Desta forma um dado *Nodo* é o último da Lista se o seu campo associado à *ligação* possui o valor *NULL*.

## 1.4. Movimentos sobre a Lista

Por vezes é necessário processar outros *Nodos* da Lista para além do primeiro. Para realizar esta tarefa são necessárias algumas variáveis auxiliares do tipo ponteiro. A *Figura*

*2* mostra uma destas variáveis, chamada *Aux*. Note que, embora *Aux* seja um ponteiro, é uma variável estática, isto é, possui um nome e deve aparecer na declaração de variáveis tal como qualquer outra variável.

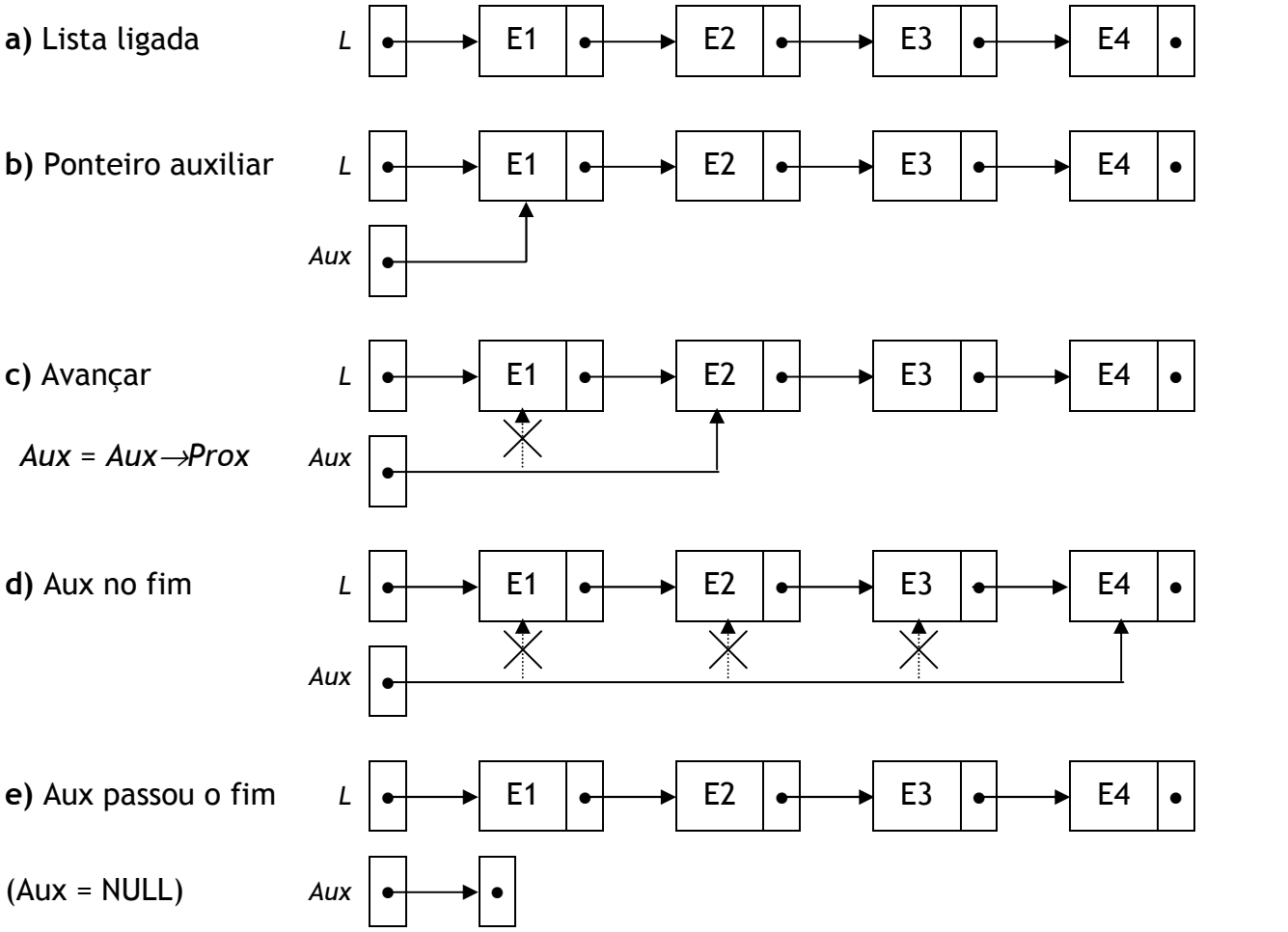



**Figura 2 - Ponteiros e listas ligadas.**

Para *Aux* começar no início da lista usa-se a atribuição Aux = L. Esta instrução copia o valor de *L* para *Aux*, o que faz com que *Aux* aponte para o mesmo *nodo* que *L*, isto é, para o início da Lista.

A seguir pretende-se mover *Aux* uma posição na Lista, como está ilustrado na *Figura 2.c*. Se Aux aponta para um *nodo* da Lista, então este *nodo* é **Aux* e o campo *Prox* de **Aux* aponta para o próximo nó da Lista. Este é o valor que se atribui a *Aux* para o mover uma posição na Lista. Por outras palavras: a atribuição "Aux = Aux→Prox" (que é igual à atribuição "Aux = *Aux.Prox") avança o ponteiro Aux uma posição na Lista (ligada). Esta é uma das operações fundamentais quando se manipulam Lista. Note-se a analogia entre a atribuição "*Aux = Aux→Prox*" (para ponteiros) e "*i = i +1*" (para índices, que avança o índice uma posição no *array* ou na lista contínua).

Repetindo a atribuição "*Aux = Aux→Prox*" pode-se avançar *Aux* uma posição na Lista de cada vez, até se encontrar o último *nodo* da Lista (*Figura 2.d*). Se *Aux* avançar mais uma posição então passa o fim da Lista. Quando tal acontece, é atribuído a *Aux* o valor do campo *Prox* do último *nodo* e este campo contém o valor *NULL*. Esta situação é reconhecida pela condição "*Aux = NULL*" (*Figura 2.e*)).

Quando se trabalha com listas ligadas, precisa-se com muita frequência de fazer testes tais como "*if (Aux != NULL) ...*" ou "*while (Aux != NULL) ...*" para se assegurar que se está a trabalhar com um *nodo* da Lista.

Por exemplo, um ciclo para percorrer a Lista L:

```
Aux = L;
while (Aux != NULL) {
    Visitar (Aux→Elemento);
    Aux = Aux→Prox;
};
```

De notar que é fácil mover para a frente, mas, no entanto, não existe nenhuma forma fácil de recuar, pois as *ligações* só permitem o movimento numa única direção (indicam qual é o próximo elemento). A forma mais óbvia de recuar um *nodo* é começar de novo no início (*L*) da Lista e percorrer todos os nodos até chegar a um que aponte para aquele de onde se partiu.

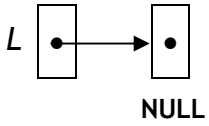
## 1.5. Operações sobre a EAD Lista

### 1.5.1. Criar uma Lista

*Entrada*: nada (void)

*Saída*: devolve um ponteiro para uma lista vazia (sem elementos)

```
PNodo CriarLista () {
    PNodo L;
    L = NULL;
    return L;
}
```

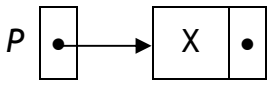


### 1.5.2. Criar um nodo duma Lista

*Entrada*: a informação que se pretende tratar, e que faz parte de um nodo (X)

*Saída*: um ponteiro para um nodo (informação + ligação)

```
PNodo CriarNodo (Info X) {
    PNodo  P;
    P = (PNodo) malloc(sizeof(struct Nodo));
    if (P == NULL)
        return  NULL;
    P→Elemento = X;
    P→Prox = NULL;
    return P;
}
```

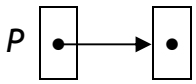



### 1.5.3. Libertar/destruir um nodo duma Lista

*Entrada*: um ponteiro para o nodo que se pretende destruir (P)

*Saída*: o ponteiro a apontar para *NULL*

```
PNodo LibertarNodo (PNodo P) {
    free(P);
    P = NULL;
    return P;
}
```




### 1.5.4. Verificar se uma Lista está vazia

*Entrada*: um ponteiro para o início da Lista (L)

*Saída*: 1 (se a Lista está vazia) ou 0 (se a Lista não está vazia)

```
int ListaVazia (PNodo L) {
    if (L == NULL)
        return  1;
    return  0;
}
```

### 1.5.5. Mostrar/listar os elementos duma Lista

*Entrada*: um ponteiro para o início da Lista (L)

*Saída*: nada (apenas mostra os elementos da Lista no écran)

Mostrar do início para o fim (versão iterativa)

```
void MostrarLista (PNodo L) {
    PNodo P = L;
    while (P != NULL) {
        printf(P→Elemento);
        P = P→Prox;
    }
}
```

Mostrar do início para o fim (versão recursiva)

```
void MostrarListaRec (PNodo L) {
    if (L != NULL) {
        printf(L→Elemento);
        MostrarListaRec(L→Prox);
    }
}
```

Mostrar do fim para o início (versão recursiva)

```
void MostrarListaContrarioRec (PNodo L) {
    if (L != NULL) {
        MostrarListaContrarioRec(L→Prox);
        printf(L→Elemento);
    }
}
```

### 1.5.6. Determinar o tamanho duma Lista

*Entrada*: um ponteiro para o início da Lista (L)

*Saída*: o tamanho da Lista

Versão iterativa:

```
int TamanhoLista (PNodo L) {
    PNodo P = L;
    int  tam = 0;
    while (P != NULL) {
        tam++;
        P = P→Prox;
    }
    return tam;
}
```

Versão recursiva:

```
int TamanhoListaRec (PNodo L) {
    if (L == NULL)
        return 0;
    return (1 + TamanhoListaRec(L→Prox));
}
```

### 1.5.7. Consultar um elemento duma Lista

*Entrada*: um ponteiro para o início da Lista (L) e o elemento a consultar (X)

*Saída*: 0 (não existe o elemento) ou 1 (o elemento existe)

Versão iterativa:

```
int ConsultarLista (PNodo L, Info X) {
    PNodo  P = L;
    if (L == NULL)
        return 0;
    while (P != NULL)
        if (P→Elemento == X)
            return 1;
        else
            P = P→Prox;
    return 0;
}
```

Versão recursiva:

```
int ConsultarListaRec (PNodo L, Info X) {
    if (L == NULL)
        return 0;
    if (L→Elemento == X)
        return 1;
    return ConsultarListaRec(L→Prox, X);
}
```

### 1.5.8. Procurar o nodo anterior dum elemento duma Lista

Em muitas operações, tais como remover um nodo e trocar dois nodos, é necessário executar uma operação que permita localizar o nodo anterior ao nodo associado a um dado elemento da Lista.

*Entrada*: o elemento a procurar (X) e uma Lista (L)

*Saída*: ponteiro para o nodo anterior ao nodo com X (NULL se X é o elemento inicial)

*Pré-requisitos*: Lista L não vazia e X pertence à Lista L

```
PNodo ProcurarAnterior (PNodo L, Info X) {
    PNodo  P = L, Ant = NULL;
    while (P→Elemento != X) {       // é garantido que X ∈ L
        Ant = P;
        P = P→Prox;
    }
    return  Ant;
}

PNodo ProcurarAnteriorRec (PNodo L, Info X) {
    if (L→Elemento == X)
        return NULL;  // só acontece se X estiver no início da lista inicial
    if (L→Prox→Elemento == X)
        return L;
    return  ProcurarAnteriorRec(L→Prox, X);
}
```

*Entrada:* uma Lista (L) e o elemento a pesquisar (X)

*Saída:* o ponteiro para o elemento que ficará antes de X (*NULL*, se X for inserido no início de L).

```
PNodo ProcurarAnteriorOrdem (PNodo L, Info X) {
    PNodo P = L, Ant = NULL;
    while (P != NULL  &&  P→Elemento < X) {
            Ant = P;
            P = P→Prox;
        }
    return Ant;
}
```

### 1.5.9. Inserir um elemento numa Lista

*Entrada*: um ponteiro para o início da Lista (L) e o elemento a inserir (X)

*Saída*: a Lista L atualizada (com mais um elemento)

Inserir no início da Lista:

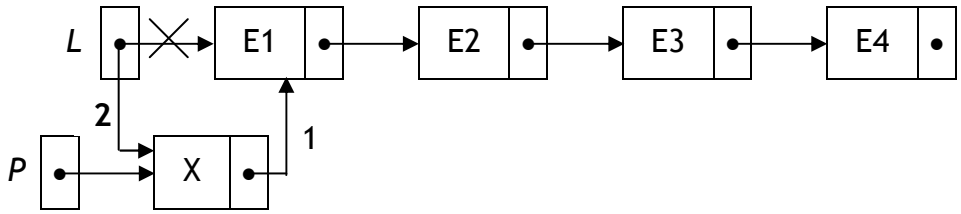



**Figura 3 - Inserir um elemento no início duma Lista.**

```
PNodo InserirInicio (PNodo L, Info X) {
        PNodo P;
        P = CriarNodo(X);
        if (P == NULL)
                return L;
        P→Prox = L;     (1)
        L = P;          (2)
        return L;
}
```

Inserir no fim da Lista:

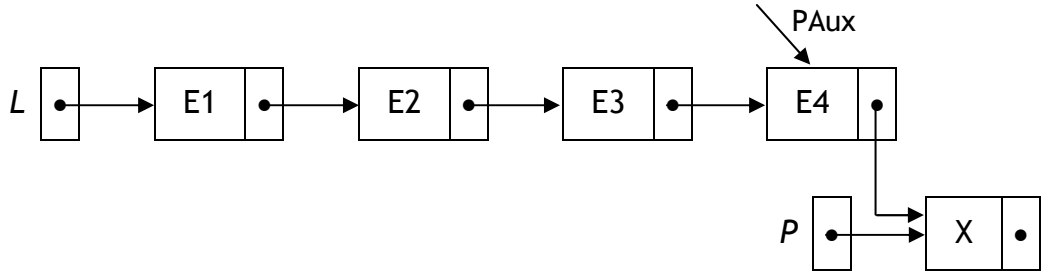



**Figura 4 - Inserir um elemento no fim duma lista.**

```
PNodo InserirFim (PNodo L, Info X) {
        PNodo P, PAux;
        P = CriarNodo(X);
        if (P == NULL)
                return L;
        if (L == NULL) {  // a lista encontra-se vazia
                L = P;
                return L;
        }
        PAux = L;
```

```
        while (PAux→Prox != NULL)
            PAux = PAux→Prox;
        PAux→Prox = P;
        return L;
    }
```

Inserir no fim da lista (versão recursiva)

```
    PNodo InserirFimRec (PNodo L, Info X) {
        PNodo  P;
        if (L == NULL) {   // lista inicial vazia
            P = CriarNodo(X);
            if (P != NULL)
                L = P;
            return L;
        }
        if (L→Prox == NULL) {   // lista apenas com 1 elemento
            P = CriarNodo(X);
            if (P != NULL)
                L→Prox = P;
            return L;
        }
        return InserirFimRec(L→Prox, X);  // lista com mais do que 1 elemento
    }
```

Inserir numa Lista ordenada (ordem crescente)

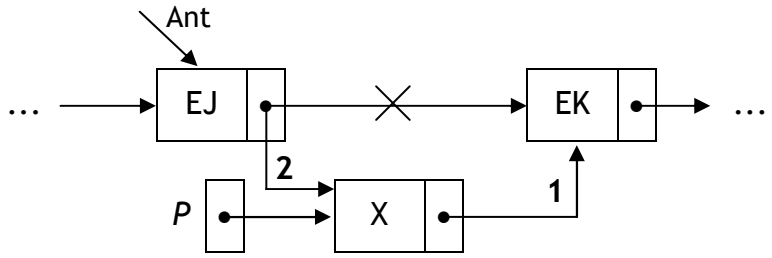



**Figura 5 - Inserir um elemento numa lista ordenada entre dois elementos.**

Esta operação necessita de uma operação auxiliar para procurar o local onde inserir o elemento: a função *ProcurarAnteriorInserirOrdem*.

*Entrada:* uma Lista (L) e o elemento a pesquisar (X)

*Saída:* o ponteiro para o elemento que ficará antes de X (*NULL*, se X for inserido no início de L).

```
PNodo ProcurarAnteriorInserirOrdem (PNodo L, Info X) {
        PNodo P = L, Ant = NULL;
        while (P != NULL  &&  P→Elemento < X) {
                Ant = P;
                P = P→Prox;
            }
        return Ant;
    }
PNodo InserirOrdem (PNodo L, Info X) {
        PNodo  P, Ant;
        P = CriarNodo(X);
        if (P == NULL)
            return L;
        if (L == NULL) {        // a lista L está vazia
            L = P;
            return L;
        }
        Ant = ProcurarAnteriorInserirOrdem(L, X);
        if (Ant == NULL) {     // X deve ser inserido no início da lista L
            P→Prox = L;
            L = P;
            return L;
        }
        P→Prox = Ant→Prox;      (1)
        Ant→Prox = P;           (2)
        return L;
}
```

### 1.5.10. Remover um elemento duma Lista

Na implementação desta operação admite-se que o elemento X existe na Lista L (não há perda de generalidade, uma vez que só depois de se verificar se o elemento X existe na Lista L é que se aplica a operação remover).

Na versão iterativa desta operação necessita de uma operação auxiliar para procurar o local onde se encontra o elemento a remover. Esta é a função ***ProcurarAnterior***.

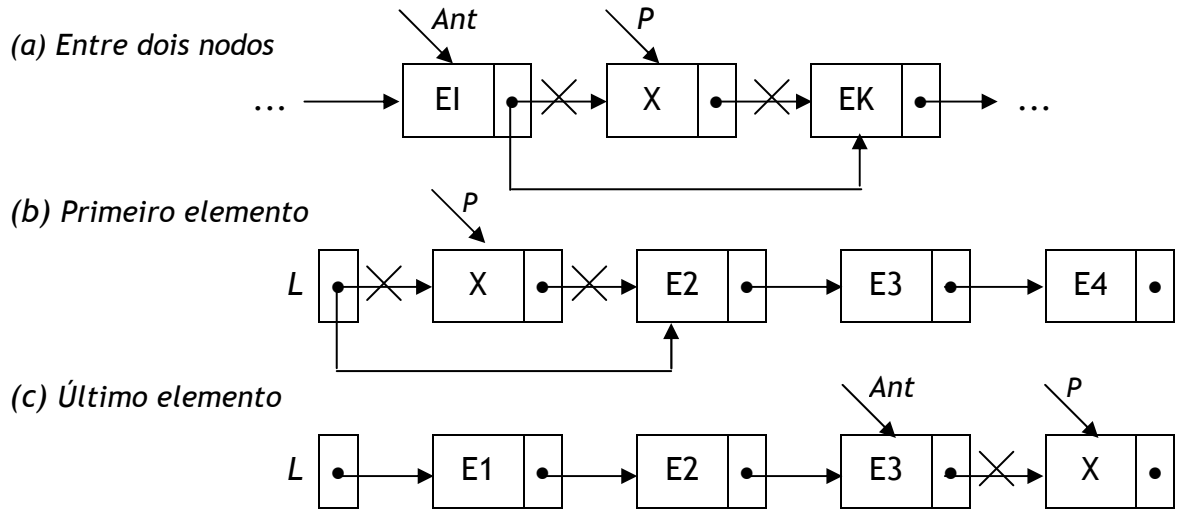



**Figura 6 - Remoção dum elemento duma lista.**

*Entrada*: o elemento X a remover e uma Lista não vazia L (X ∈ L)

*Saída*: a Lista atualizada (sem o elemento X)

*Pré-requisitos*: Lista L não vazia e X pertencente a L

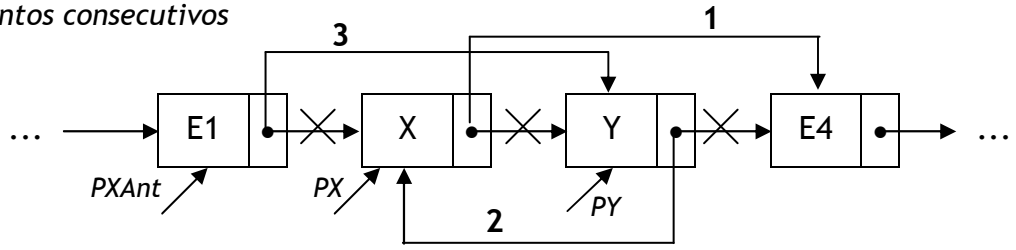
```
PNodo RemoverLista (PNodo L, Info X) {
    PNodo  Ant, P;
    Ant = ProcurarAnterior(L, X);
    if (Ant == NULL) {     // X está no início da Lista L
        P = L;
        L = L→Prox;
    }
    else {       // X está no meio ou no fim da Lista L
        P = Ant→Prox;
        Ant→Prox = P→Prox;
    }
    P = LibertarNodo(P);
    return L;
}
```

A versão recursiva é a seguinte:

```
PNodo RemoverListaRec (PNodo L, Info X) {
    PNodo  P;
    if (L→Elemento == X) {
        P = L;
        L = L→Prox;
        P = LibertarNodo(P);
        return L;
    }
    if ((L→Prox)→Elemento == X) {
        P = L→Prox;
        L→Prox = L→Prox→Prox;
        P = LibertarNodo(P);
        return L;
    }
    return RemoverListaRec(L→Prox, X);
}
```

### 1.5.11. Trocar dois elementos numa Lista

*(a) Elementos consecutivos*




*(b) Elementos distantes*

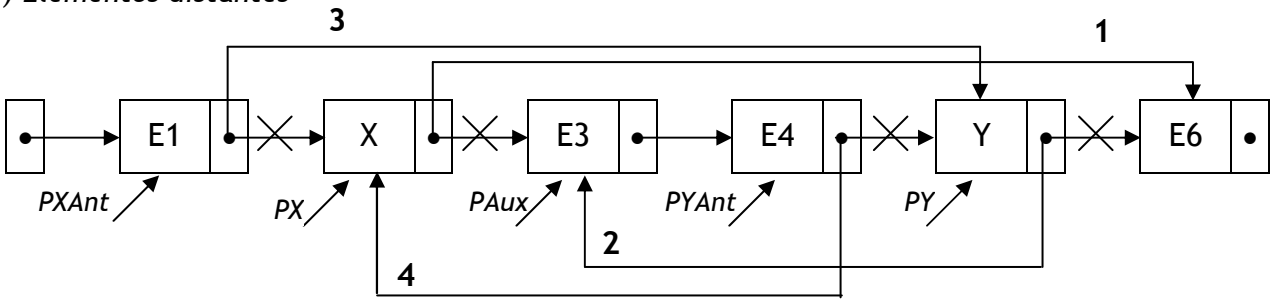



**Figura 7 - Troca de dois elementos duma Lista (X com Y).**

```
PNodo TrocarLista (PNodo L, Info X, Info Y) {
    PNodo  PX, PY, PXAnt, PYAnt, PAux;
    PXAnt = ProcurarAnterior(L, X);
    if (PXAnt == NULL)
        PX = L;
    else
        PX = PXAnt→Prox;
    PYAnt = ProcurarAnterior(L, Y);
    if (PYAnt == NULL)
        PY = L;
    else
        PY = PYAnt→Prox;
    PAux = PX→Prox;
    PX→Prox = PY→Prox;      (1)
    PY→Prox = PAux;         (2)
    if (PXAnt == NULL)
        L = PY;
    else
        PXAnt→Prox = PY;    (3)
    if (PYAnt == NULL)
        L = PX;
    else
        PYAnt→Prox = PX;    (4)
    return L;
}
```

# 2. Listas com ligações duplas – armazenamento não sequencial

## 2.1. Introdução

Uma ***EAD Lista*** pode ser implementada através de uma lista duplamente ligada, em que cada um dos seus elementos é composto por três campos:

1. Os dados ou informação propriamente dita;
2. Um ponteiro para o próximo elemento da Lista (sucessor);
3. Um ponteiro para o elemento anterior (antecessor).

Estes dois ponteiros facilitarão a tarefa de percorrer toda a Lista em qualquer direção, isto é da cabeça para a cauda e vice-versa, ou alterando a direção daquela tarefa em qualquer momento.

Neste tipo de implementação de uma Lista é habitual usar-se dois ponteiros fixos: uma a apontar para a cabeça (***Head***) e outro para a cauda (***Tail***).

Assim, uma possível definição para a ***EAD Lista*** com ligações duplas é a seguinte:

```
struct  NodoLD {
        Info     Elemento;
        struct   NodoLD  *Prox;
        struct   NodoLD  *Ant;
};
typedef   struct   NodoLD   *PNodoLD;
PNodoLD  Head, Tail;
```

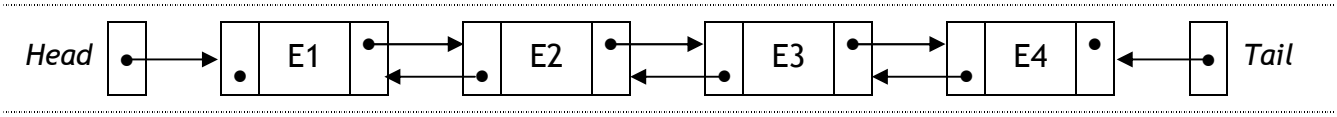



**Figura 8 - Representação esquemática duma lista duplamente ligada.**

## 2.2. Operações sobre a Lista

Qualquer operação que necessite percorrer a **Lista**, isto pode ser feito de duas formas: da cabeça para cauda, ou vice-versa. Se a **Lista** for percorrida da cabeça para cauda, então deve-se fornecer o ponteiro para a cabeça (*Head*) e usar o campo "*Prox*" para caminhar pela Lista. Se a Lista for percorrida da cauda para a cabeça, então deve-se fornecer o ponteiro para a cauda (*Tail*) e usar o campo "*Ant*" para caminhar pela Lista.

### 2.2.1. Criar uma lista

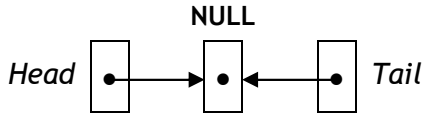



**Figura 9 - Representação duma Lista duplamente ligada vazia.**

*Entrada*: os ponteiros para a cabeça (*Head*) e para a cauda (*Tail*) da Lista

*Saída*: os mesmos ponteiros a apontarem para *NULL* (Lista vazia)

```
void CriarLD (PNodoLD *Head, PNodoLD *Tail) {
    *Head = NULL;
    *Tail = NULL;
}
```

### 2.2.2. Criar um nodo duma Lista

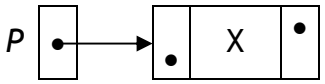



**Figura 10 - Representação esquemática dum nodo duma Lista duplamente ligada.**

*Entrada*: um elemento X

*Saída*: um ponteiro para um nodo (informação + ligação)

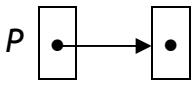
```
PNodoLD CriarNodoLD (Info X) {
    PNodoLD  P = (PNodoLD) malloc (sizeof (struct NodoLD));
    if (P == NULL)
        return  NULL;
    P→Elemento = X;
    P→Prox = NULL;
    P→Ant = NULL;
    return  P;
}
```

### 2.2.3. Libertar/destruir um nodo duma Lista (P)

*Entrada*: um ponteiro para o nodo que se pretende destruir

*Saída*: o ponteiro P a apontar para *NULL*

```
PNodoLD LibertarNodoLD (PNodoLD P) {
        free(P);
        P = NULL;
        return P;
}
```




### 2.2.4. Verificar se uma Lista está vazia

*Entrada*: um ponteiro para a cabeça ou para a cauda da lista (P)

*Saída*: 1 (Lista vazia) ou 0 (Lista não vazia)

```
int LDVazia (PNodoLD P) {
        if (P == NULL)
                return 1;
        return 0;
}
```

### 2.2.5. Mostrar/listar os elementos duma Lista

Listar da cabeça para a cauda (versão iterativa):

*Entrada*: um ponteiro para a cabeça da Lista (*Head*)

*Saída*: nada (apenas mostra os elementos da Lista no écran)

```
void ListarLD (PNodoLD Head) {
        PNodoLD  P = Head;
        while (P != NULL) {
                printf (P→Elemento);
                P  =  P→Prox;
        }
}
```

Listar da cauda para a cabeça (versão iterativa):

*Entrada*: um ponteiro para a cauda da Lista (*Tail*)

*Saída*: nada (apenas mostra os elementos da Lista no écran)

```
void ListarLDContrario (PNodoLD Tail) {
        PNodoLD  P = Tail;
        while (P != NULL) {
                printf (P→Elemento);
                P = P→Ant;
        }
}
```

### 2.2.6. Determinar o tamanho duma lista

Para se determinar o tamanho de uma Lista duplamente ligada, deve-se percorrer toda a Lista e contar os seus elementos. Esta operação pode ser feita percorrendo a Lista da cabeça para cauda, ou vice-versa. A função que se segue usa estas duas formas de percorrer a Lista, mas deve-se apenas escolher um delas para execução.

*Entrada*: um ponteiro para a cabeça/cauda da lista (*Head*/*Tail*)

*Saída*: o tamanho da Lista

```
int TamanhoLD (PNodoLD Head/Tail) {
        PNodoLD  P = Head/Tail;
        int  tam = 0;
        while (P != NULL) {
                tam = tam + 1;
                P = P→Prox/Ant;
        }
        return tam;
}
```

### 2.2.7. Consultar um elemento numa Lista

Realizar consulta da cabeça para a cauda da lista:

*Entrada*: um ponteiro para a cabeça da Lista (*Head*) e o elemento a consultar (*X*)

*Saída*: 1 (existe o elemento na Lista) ou 0 (não existe o elemento na Lista)

```
int ConsultarLDCabeca (PNodoLD Head, Info X) {
        PNodoLD  P = Head;
        if (Head == NULL)
                return 0;
        while (P != NULL && P→Elemento != X)
                P = P→Prox;
        if (P != NULL && P→Elemento == X)
                return 1;
        return 0;
}
```

```
int ConsultarLDCabecaOrdem (PNodoLD Head, Info X) {    // Lista ordenada
    PNodoLD  P = Head;
    if (Head == NULL)
        return 0;
    while (P != NULL && P→Elemento <= X)
        P = P→Prox;
    if (P != NULL && P→Elemento == X)
        return 1;
    return 0;
}
```

Realizar consulta da cauda para a cabeça da Lista:

*Entrada*: um ponteiro para a cauda da Lista (*Tail*) e o elemento a consultar (X)

*Saída*: 1 (existe o elemento na Lista) ou 0 (não existe o elemento na Lista)

```
int ConsultarLDCauda (PNodoLD Tail, Info X) {
    PNodoLD  P = Tail;
    if (Tail == NULL)
        return 0;
    while (P != NULL && P→Elemento != X)
        P = P→Ant;
    if (P != NULL && P→Elemento == X)
        return 1;
    return 0;
}

int ConsultarLDCaudaOrdem (PNodoLD Tail, Info X) {    // com Lista ordenada
    PNodoLD  P = Tail;
    if (Tail == NULL)
        return 0;
    while (P != NULL && P→Elemento <= X)
        P = P→Ant;
    if (P != NULL && P→Elemento == X)
            return 1;
    return 0;
}
```

### 2.2.8. Procurar o nodo que contém um elemento duma Lista

Em muitas operações, tal como remover um nodo, é necessário executar uma operação que permita localizar o nodo anterior ao nodo que contém uma dada informação.

*Entrada*: um ponteiro para a cabeça da Lista (*Head*) e o elemento a procurar (X)

*Saída*: o ponteiro do nodo que contém o elemento a procurar (*NULL*, se não existe)

```
PNodoLD ProcurarElementoLD (PNodoLD Head, Info X) {
    PNodoLD  P = Head;
    while (P != NULL && P→Elemento != X)
        P = P→Prox;
    return P;
}
```

### 2.2.9. Inserir um elemento numa Lista

Inserir o primeiro elemento numa Lista:

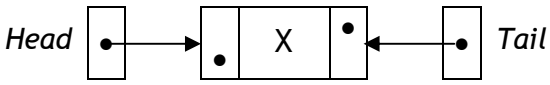



**Figura 11 - Inserir o primeiro elemento numa Lista duplamente ligada.**

*Entrada*: os ponteiros para a cabeça (*Head*) e para a cauda (*Tail*) da Lista, e o elemento a inserir (X)

*Saída*: a Lista atualizada (com um elemento)

```
void InserirPrimeiroElementoLD (PNodoLD *Head, PNodoLD *Tail, Info X) {
    PNodoLD  P = CriarNodoLD(X);
    if (P != NULL) {
        *Head = P;
        *Tail = P;
    }
}
```

Inserir à cabeça da Lista (supõe-se existir pelo menos um elemento na Lista):

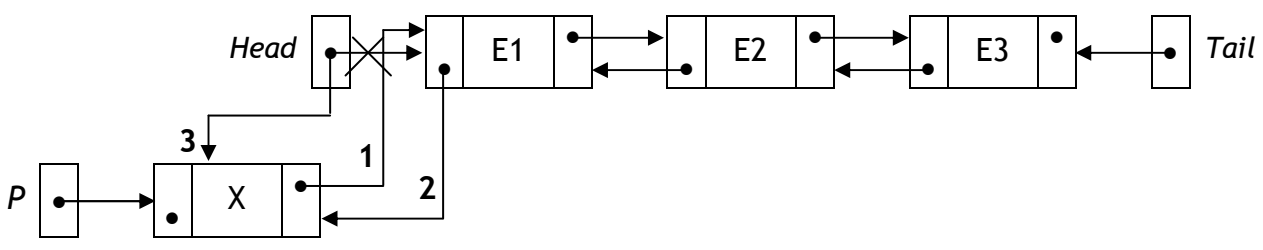



**Figura 12 - Inserir um elemento na cabeça duma Lista duplamente ligada.**

*Entrada*: um ponteiro para a cabeça da Lista (*Head*) e o elemento a inserir (X)

*Saída*: a Lista atualizada (com mais um elemento)

```
PNodoLD InserirCabecaLD (PNodoLD Head, Info X) {
    PNodoLD  P = CriarNodoLD(X);
    if (P == NULL)
        return Head;
    P→Prox = Head;      (1)
    Head→Ant = P;       (2)
    Head = P;           (3)
    return Head;
}
```

Inserir na cauda da Lista (supõe-se existir pelo menos um elemento na Lista):

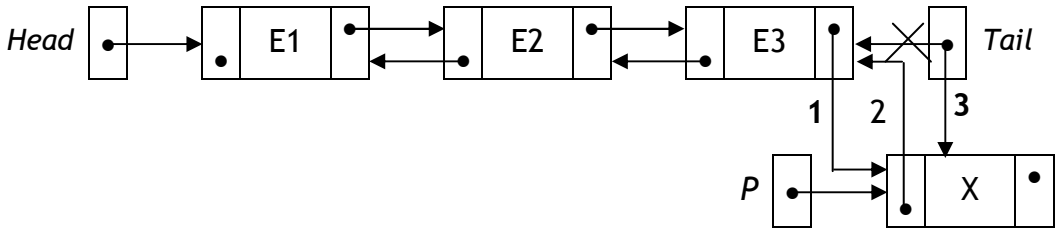



**Figura 13 - Inserir um elemento na cauda duma Lista duplamente ligada.**

*Entrada*: um ponteiro para a cauda da Lista (*Tail*) e o elemento a inserir (X)

*Saída*: a Lista atualizada (com mais um elemento)

```
PNodoLD InserirCaudaLD (PNodoLD Tail, Info X) {
    PNodoLD  P = CriarNodo(X);
    if (P == NULL)
        return Tail;
    Tail→Prox = P;      (1)
    P→Ant = Tail;       (2)
    Tail = P;           (3)
    return Tail;
}
```

Inserir numa Lista ordenada (ordem crescente):

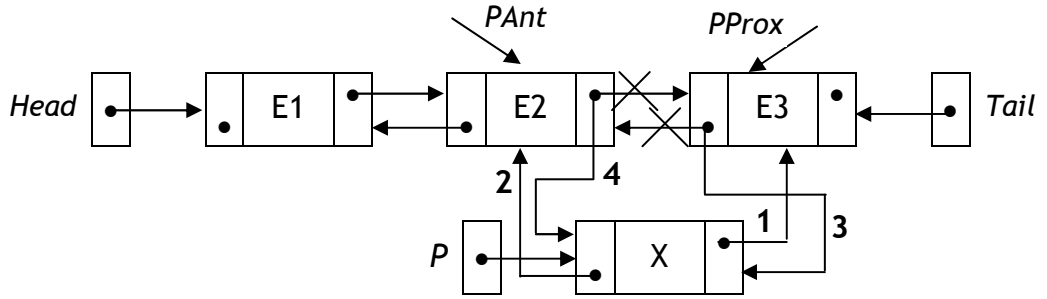



**Figura 14 - Inserir um elemento no meio duma Lista duplamente ligada.**

Esta operação necessita de uma outra operação que permita localizar o nodo que ficará antes do nodo a inserir, tendo em conta que a Lista se encontra ordenada por ordem crescente.

Procurar o nodo anterior ao nodo a inserir:

*Entrada*: um ponteiro para a cabeça da Lista (*Head*) e o elemento a pesquisar (X)

*Saída*: ponteiro para o nodo que ficará antes do nodo com o elemento a procurar

```
PNodoLD ProcurarAnteriorInserirOrdemLD (PNodoLD Head, Info X) {
        PNodoLD  P = Head, PAnt = NULL;
        while (P != NULL && P→Elemento < X) {
                PAnt = P;
                P = P→Prox;
            }
        return PAnt;
}
```

*Entrada*: ponteiros para a cabeça (*Head*) e a cauda (*Tail*) da Lista, e o elemento a inserir (X)

*Saída*: a Lista atualizada (com mais um elemento)

```
int InserirOrdemLD (PNodoLD *Head, PNodoLD *Tail, Info X) {
        PNodoLD  P, PAnt, PProx;
        P = CriarNodoLD(X);
        if (P == NULL)
                return  0;
        if (*Head == *Tail) {   // inserir X numa lista vazia
                InserirPrimeiroElementoLD(*Head, *Tail, X);
                return 1;
        }
        PAnt = ProcurarAnteriorInserirOrdemLD(*Head, X);
```

```
        if (PAnt == NULL) {    // inserir X na cabeça da Lista
            *Head = InserirCabecaLD(*Head, X);
            return 1;
        }
        if (PAnt == *Tail) {    // inserir X na cauda da Lista
            *Tail = InserirCaudaLD(*Tail, X);
            return 1;
        }
        PProx = PAnt→Prox;
        P→Prox = PAnt→Prox;      (1)
        P→Ant = PAnt;            (2)
        PAnt→Prox→Ant = P;       (3)
        PAnt→Prox = P;           (4)
        return 1;
}
```

### 2.2.10. Remover um elemento duma Lista

Remover o único elemento que se encontra numa Lista:

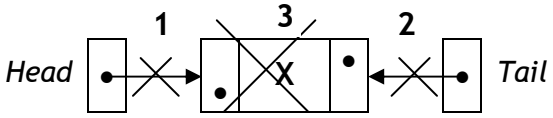



**Figura 15 - Remove o único elemento duma Lista duplamente ligada.**

*Entrada*: ponteiros para a cabeça (*Head*) e a cauda (*Tail*) da Lista

*Saída*: a Lista atualizada (vazia)

```
void RemoverUnicoElementoLD (PNodoLD *Head, PNodoLD *Tail) {
        PNodoLD P = *Head;
        *Head = NULL;            (1)
        *Tail = NULL;            (2)
        P = LibertarNodoLD(P);   (3)
}
```

Remover o elemento que está à cabeça da Lista:

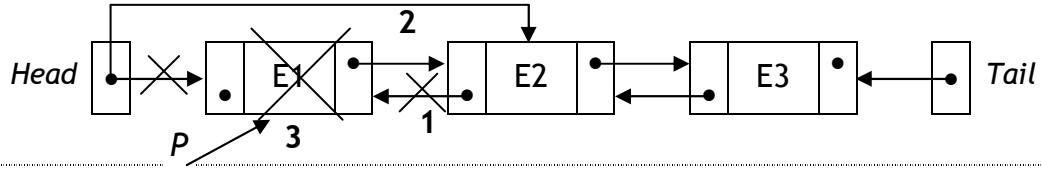



**Figura 16 - Remover o elemento à cabeça duma Lista duplamente ligada.**

*Entrada*: um ponteiro para a cabeça da Lista (*Head*)

*Saída*: a Lista atualizada (com menos um elemento)

```
PNodoLD RemoverCabecaLD (PNodoLD Head) {
    PNodoLD P = Head;
    Head→Prox→Ant = NULL;        (1)
    Head = Head→Prox;            (2)
    P = LibertarNodoLD(P);       (3)
    return Head;
}
```

Remover o elemento da cauda da Lista:

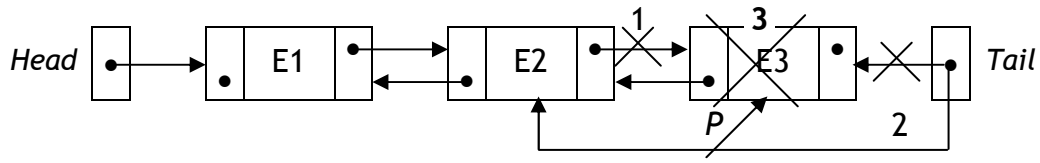



**Figura 17 - Remover o elemento da cauda duma Lista duplamente ligada.**

*Entrada*: um ponteiro para a cauda da Lista (*Tail*)

*Saída*: a Lista atualizada (com menos um elemento)

```
PNodoLD RemoverCaudaLD (PNodoLD Tail) {
    PNodoLD  P = Tail;
    Tail→Ant→Prox = NULL;        (1)
    Tail = Tail→Ant;             (2)
    P = LibertarNodoLD(P);            (3)
    return Tail;
}
```

Remover um elemento duma Lista (entre dois elementos):

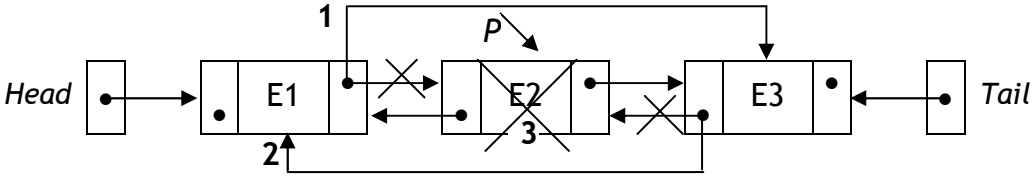



**Figura 18 - Remover um elemento duma Lista duplamente ligada (entre dois elementos).**

Esta operação necessita de uma operação auxiliar para procurar o elemento a remover (usar a função *ProcurarElemento* — ver secção 2.2.8, pág. 20). Com a execução desta função, os ponteiros para a cabeça e a cauda não sofrem alteração, pois o elemento a remover não se encontra à cabeça, nem na cauda da lista.

*Entrada*: um ponteiro para a cabeça da Lista (*Head*)

*Saída*: a Lista atualizada (com menos um elemento)

```
void RemoverEntreDoisElementosLD (PNodoLD Head, Info X) {
    PNodoLD  P = ProcurarElementoLD(Head, X);
    if (P != NULL) {
        P→Ant→Prox = P→Prox;          (1)
        P→Prox→Ant = P→Ant;           (2)
        P = LibertarNodoLD(P);        (3)
    }
}
```

Remover um elemento duma Lista (global):

A operação para remover um elemento duma Lista consiste no seguinte: dado uma lista e um elemento a remover, verificar em que caso se enquadra esta situação: a) se a Lista está vazia, b) se o elemento está na cabeça da Lista, c) se o elemento está na cauda da Lista e, d) se o elemento está entre dois elementos.

*Entrada*: ponteiros para a cabeça (*Head*) e a cauda (*Tail*) da Lista, e o elemento a remover (X)

*Saída*: a Lista atualizada (com menos um elemento)

```
int Remover (PNodoLD *Head, PNodoLD *Tail, Info X) {
    if (ConsultarCabecaLD(*Head, X) == 0)     // X não pertence à lista
        return 0;
    if (*Head→Elemento == X) {      // X está na cabeça da lista
        *Head = RemoverCabecaLD(*Head);
        return 1;
    }
    if (*Tail→Elemento == X) {      // X está na cauda da lista
        *Tail = RemoverCaudaLD(*Tail);
        return 1;
    }
    RemoverEntreDoisElementosLD(*Head, X);
    return 1;
}
```

# 3. Listas circulares – armazenamento não sequencial

Quando a *EAD Lista* é definida através de uma lista circular, o nodo seguinte (*Prox*) ao último é o primeiro. Isto é, o último nodo aponta para o primeiro.

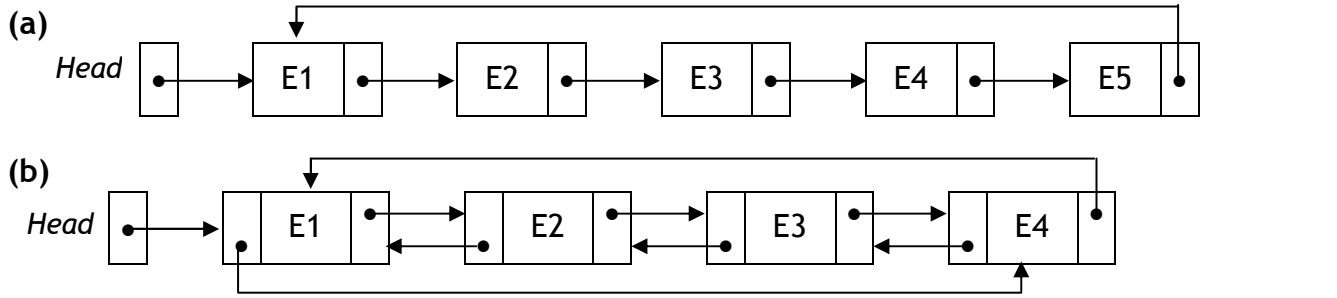



**Figura 19 - Representação esquemática de Listas ligadas circulares: (a) simples e (b) duplas.**

Agora, todas as travessias, em vez de "*while (P != NULL)* … " passam a ser da forma "*while (P != Head)* … ". No entanto, esta situação provoca um problema na entrada no ciclo (inicialmente P = *Head*), para além de não funcionar para o caso de Listas vazias.

Poder-se-ia resolver este problema, aplicando este processo apenas a Listas não vazias (o que implicaria verificação prévia) e passando a condição de paragem para o final do ciclo. Isto é, alterar-se o tipo de ciclo a aplicar, de "*while* …" para "*do* … *while*", o que não acarreta problemas pois, à partida, sabe-se que a Lista não está vazia. A seguir apresenta-se uma possível função para mostrar todos os elementos duma Lista circular simples.

*Entrada*: um ponteiro para a cabeça da Lista (*Head*)

*Saída*: nada (mostra no écran todos os elementos da Lista)

```
void Listar (PNodo Head) {      //  para Listas não vazias
    PNodo P = Head;
    do {
        printf (P→Elemento);
        P = P→Prox;
    } while (P != Head);
}
```

No entanto, apresentam-se a seguir duas possíveis soluções, para evitar a aplicação do teste de verificação de Lista vazia:

**1.** Utilizar-se um nodo inicial auxiliar chamado ***Base***;

**2.** Em vez de se guardar a cabeça da Lista (*Head*), guardar a cauda (*Tail*).

## 3.1. Listas circulares com Base

O elemento que funcionar como ***Base*** deve ser tal que fique sempre "posicionado" à cabeça da Lista; isto é, que seja o elemento apontado pela cabeça da Lista (*Head*) — Figura 20 e Figura 21.

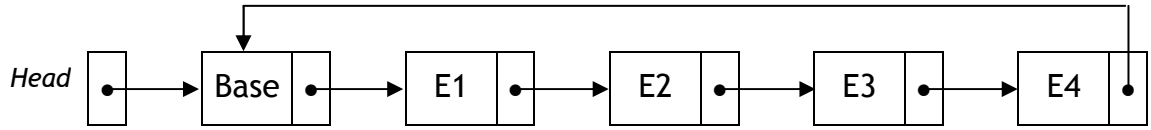



**Figura 20 - Representação esquemática duma Lista ligada circular.**

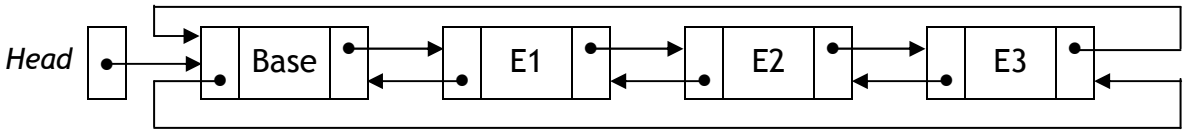



**Figura 21 - Representação esquemática duma Lista duplamente ligada circular.**

A lista propriamente dita começa no elemento *E1*. Para se percorrer toda a Lista, o primeiro elemento a ser visitado será o E1 (*P* = *Head→Prox*); o processo termina quando *P* = *Head*. Por exemplo, se para uma lista de valores inteiros positivos, a base poderia ser o elemento que tivesse um valor negativo (-1, por exemplo). A seguir apresenta-se uma possível função para mostrar todos os elementos duma Lista circular.

*Entrada*: um ponteiro para a cabeça da Lista (*Head*)

*Saída*: nada (mostra no écran todos os elementos da Lista circular)

```
void Listar (PNodo Head) {
    PNodo P = Head→Prox;
    while (P != Head) {
        printf (P→Elemento);
        P = P→Prox;
    }
}
```

Por outro lado, quando se cria uma Lista circular com base, esta já contém um elemento, que é a base, como se pode verificar pela figura seguinte:

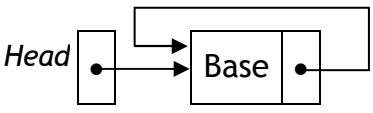



**Figura 22 - Representação duma Lista ligada circular vazia.**

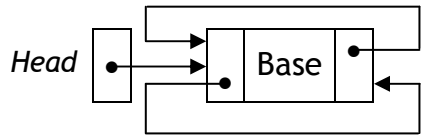



**Figura 23 - Representação duma Lista duplamente ligada circular vazia.**

Neste caso, considera-se a Lista vazia quando tem apenas um elemento, que é a base.

## 3.2. Listas circulares com ponteiro para a cauda

Para o caso de listas ligadas com apenas uma *ligação*, temos, por exemplo se pretendêssemos uma lista de valores inteiros positivos, o seguinte:

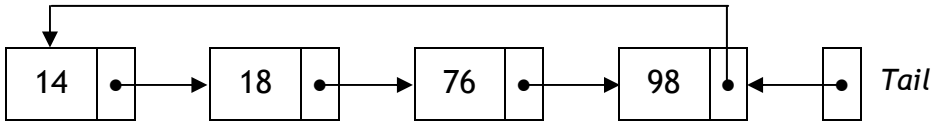



**Figura 24 - Representação esquemática duma Lista ligada circular.**

Para se percorrer toda a lista, o primeiro elemento a ser visitado será o 14 (*P = Tail→Prox*) e o critério de paragem é quando *P = Tail*. Por exemplo, repare-se na função que se segue para listar todos os elementos duma lista circular.

*Entrada*: um ponteiro para a cauda da Lista (*Tail*)

*Saída*: nada (mostra no écran todos os elementos da Lista circular)

```
void Listar (PNodo Tail) {
    PNodo P = Tail→Prox;
    while (P != Tail) {
        printf(P→Elemento);
        P = P→Prox;
    }
}
```

# 4. EAD Pilha

Na ***EAD Pilha*** (denominada por ***Pilha*** no restante documento) os seus elementos são processados pela ordem inversa à ordem de chegada: o último elemento a entrar na ***Pilha*** é o primeiro a sair (LIFO - "*Last In First Out*"). Na ***Pilha*** qualquer operação que se pretenda efetuar será realizada no topo da Pilha (ver Figura 25). A inserção de um novo elemento na ***Pilha*** consiste em acrescentar este elemento ao cimo do topo da ***Pilha***; por outro lado, a remoção de um elemento da ***Pilha*** consiste em retirar o elemento que se encontra no topo da ***Pilha***, passando o elemento que está antes, caso exista, a ser o topo da ***Pilha***. Quando é removido o último elemento da ***Pilha***, esta fica vazia, sendo apenas possível realizar operações de inserção de elementos na ***Pilha***.

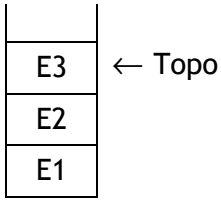



**Figura 25 - Representação esquemática duma Pilha.**

## 4.1. Definição axiomática

**Estrutura Pilha (Info)**

Declarar

| | | |
|---|---|---|
| Criar() | → | Pilha |
| Vazia(Pilha) | → | 1 (verdadeiro) ou 0 (falso) |
| Push(Info, Pilha) | → | Pilha |
| Pop(Pilha) | → | Pilha |
| Topo(Pilha) | → | Info |

tais que $\forall$ S $\in$ Pilha, $\forall$ X $\in$ Info sejam

Vazia(Criar()) = 1 (verdadeiro)

Vazia(Push(X, S)) = 0 (falso)

Pop(Criar ()) = ERRO

Pop(Push (X, S)) = S

Topo(Criar ()) = ERRO

Topo(Push (X, S)) = X

**Fim Pilha**

## 4.2. Operações

As operações que compõem a ***Pilha*** são as seguintes:

1. Criar uma Pilha (vazia): Criar(S)
2. Verificar se uma Pilha está vazia: Vazia(S)
3. Colocar um elemento X (no topo) numa Pilha: Push(X, S)
4. Remover um elemento (que está no topo) de uma Pilha: Pop(S)
5. Fornecer o elemento do topo de uma Pilha: Topo(S).

Uma ***Pilha*** pode ser implementada usando listas, sejam elas de armazenamento sequencial (usando *array's*) ou não sequencial (usando listas ligadas). Neste documento apenas se irá fazer referência à implementação usando listas ligadas.

## 4.3. Implementação usando listas ligadas simples

Considere as seguintes declarações:

```
struct NodoPilha {
        Info  Elemento;
        struct  NodoPilha  *Ant;
};
typedef  struct  NodoPilha  *PNodoPilha;
```

Na implementação de uma ***Pilha*** usando listas ligadas simples, cada nodo aponta para o nodo anterior da lista, apontando o primeiro nodo da lista para *NULL*, para servir de indicador de início da ***Pilha***.

A memória para os elementos (ou nodos) da ***Pilha*** (lista ligada simples) é atribuída quando um elemento é inserido e é libertada quando um elemento é removido da pilha. O indicador de topo da ***Pilha*** é um ponteiro que aponta para o elemento que se encontra no início da lista, S, que corresponde ao último elemento que foi inserido na ***Pilha*** e será o primeiro elemento a ser removido. Será à frente deste elemento que será inserido um novo elemento. Quando o topo da ***Pilha*** é um ponteiro nulo (*NULL*) significa que a ***Pilha*** se encontra vazia. Uma ***Pilha*** nunca está cheia, mas pode acontecer que não haja memória suficiente para alocar novo elemento com o fim de ser inserido nela. A Figura 26 que corresponde a uma representação esquemática de uma ***Pilha*** de inteiros.

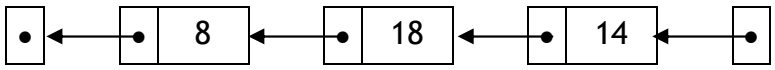


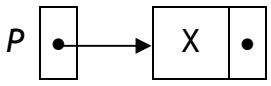

**Figura 26 – Representação de uma Pilha usando uma lista ligada simples.**

### 4.3.1. Criar um nodo de uma Pilha

*Entrada*: a informação que se pretende inserir num nodo (X)

*Saída*: um ponteiro para um nodo (informação + ligação)

```
PNodoPilha CriarNodoPilha (Info X) {
    PNodoPilha  P;
    P = (PNodoPilha) malloc(sizeof(struct NodoPilha));
    if (P == NULL)
        return  NULL;
    P→Elemento = X;
    P→Ant = NULL;
    return  P;
}
```

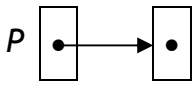



### 4.3.2. Libertar/destruir um nodo de uma Pilha

*Entrada*: um ponteiro para o nodo que se pretende destruir (P)

*Saída*: o ponteiro a apontar para *NULL*

```
PNodoPilha LibertarNodoPilha (PNodoPilha P) {
    free(P);
    P = NULL;
    return P;
}
```




### 4.3.3. Criar uma Pilha

A implementação da operação associada à criação de uma ***Pilha***, consiste apenas em fazer apontar o ponteiro para o topo da Pilha para *NULL* (ver Figura 27).

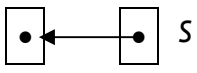



**Figura 27 – Criar uma EAD Pilha (apontar *S* para NULL).**

A seguir apresenta-se uma possível função para criar uma ***Pilha*** usando uma lista ligada simples.

*Entrada*: nada

*Saída*: ponteiro nulo (NULL)

```
PNodoPilha CriarPilha () {
        PNodoPilha  S;
        S = NULL;
        return S;
}
```

### 4.3.4. Verificar se uma Pilha está vazia

A implementação da operação associada à verificação se uma ***Pilha S*** está vazia, consiste apenas em verificar se ***S*** aponta para *NULL*. A seguir apresenta-se uma possível função para verificar se uma ***Pilha*** está vazia.

*Entrada*: ponteiro para o topo da ***Pilha*** (S)

*Saída*: 1 (se a ***Pilha*** está vazia); 0 (caso contrário)

```
int PilhaVazia (PNodoPilha S) {
        if  (S == NULL)
                return 1;
        return 0;
}
```

### 4.3.5. Inserir elemento numa Pilha

A operação de inserir um elemento numa ***Pilha*** consiste em acrescentar mais um elemento no topo desta e atualizar o ponteiro de topo da ***Pilha*** (ver Figura 28 — uma ***EAD Pilha*** de inteiros).

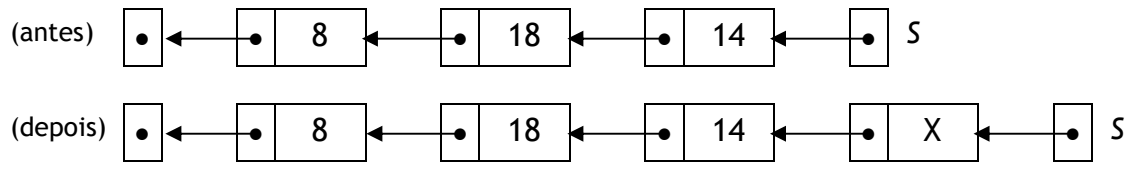



**Figura 28 – Representação da operação inserir o elemento X na Pilha.**

A seguir apresenta-se uma função possível para inserir um elemento X numa ***Pilha***.

*Entrada*: a informação a ser inserida (X) e um ponteiro para o topo da ***Pilha*** (S)

*Saída*: a ***Pilha*** com mais um elemento (X) e ponteiro para o topo da ***Pilha*** atualizado

```
PNodoPilha Push (Info X, PNodoPilha S) {
        PNodoPilha  P = CriarNodoPilha(X);
        if (P == NULL)
                return S;
        P→Ant = S;
```

```
        S = P;
        return S;
    }
```

### 4.3.6. Remover elemento duma Pilha

A operação para remover elemento duma ***Pilha*** consiste em remover o elemento que se encontra no topo da ***Pilha*** e atualizar o ponteiro de topo da ***Pilha*** (ver Figura 29 - uma ***Pilha*** de inteiros).

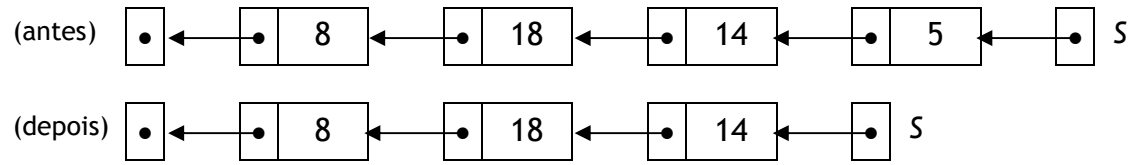



**Figura 29 – Representação da operação remover elemento da *Pilha*.**

A seguir apresenta-se uma possível função que permite remover o elemento do topo duma ***Pilha***.

*Entrada*: um ponteiro para o topo da ***Pilha*** (S)

*Saída*: remoção do elemento/nodo que se encontra no topo da ***Pilha*** e ponteiro de topo da ***Pilha*** atualizado (para o nodo anterior)

```
PNodoPilha Pop (PNodoPilha S) {
        PNodoPilha Paux = S;
        S = S→Ant;
        PAux = LibertarNodoPilha(PAux);
        return S;
}
```

### 4.3.7. Consultar elemento de uma Pilha

A operação para consultar uma ***Pilha*** consiste em devolver o elemento que se encontra no topo da ***Pilha*** (o ponteiro de topo da ***Pilha*** fica inalterado). A seguir apresenta-se uma possível função para consultar uma ***Pilha***.

*Entrada*: um ponteiro para o topo da ***Pilha*** (S)

*Saída*: o elemento que se encontra no topo da ***Pilha***

```
Info Topo (PNodoPilha S) {
        return (S→Elemento);
}
```

## 4.4. Exemplos

Considere a seguinte declaração:

```
typedef  struct {
        char car;
} Info;
```

Analisar dois problemas:

(1) verificar se uma expressão está equilibrada,

(2) inverter uma expressão.

### 4.4.1. Verificar se uma expressão está equilibrada

Dada uma expressão terminada com um ponto (.), verifica se os parêntesis estão bem colocados (equilibrados).

```
int Equilibrada (void) {
        PNodoPilha  S = CriarPilha();
        Info  X;
        char  ch;
        int  continua = 1;
        do {
                scanf("%c", &ch);
                X.car = ch;
                if (X.car == '(')
                        S = Push(X, S);
                else
                        if (X.car == ')')
                                if (!PilhaVazia(S))
                                        S = Pop(S);
                                else
                                        continua = 0;
        }
        while (ch != '.' && continua) ;
        if (continua && PilhaVazia(S))
                return 1;
        return 0;
}
```

### 4.4.2. Inverter uma expressão

Dada uma expressão terminada com um ponto (.), escrevê-la pela ordem inversa (do fim para o início).

```
void EscreverOrdemInversa () {
    PNodoPilha  S = CriarPilha();
    Info  X;
    char  ch;
    do {
        scanf("%c", &ch);
        if (ch != '.') {
            X.car = ch;
            S = Push(X, S);
        }
    }
    while (ch != '.')
        ;
    while (!PilhaVazia(S)) {
        X = Topo(S);
        printf("%c", X.car);
        S = Pop(S);
    }
}
```

# 5. EAD Fila

Na ***EAD Fila*** (ou simplesmente ***Fila***) os seus elementos são processados por ordem de chegada: o primeiro elemento a entrar na ***Fila*** é o primeiro a sair (FIFO - "*First In First Out*"). Na ***Fila*** algumas operações realizam-se na frente/cabeça e outras na cauda da ***Fila*** (ver Figura 30). As operações de inserção realizam-se na cauda da ***Fila***, enquanto que as operações de remoção realizam-se na frente. Portanto, a inserção de um novo elemento consiste em acrescentar este elemento na cauda da ***Fila***, tornando-se este a cauda; por outro lado, a remoção de um elemento da ***Fila*** consiste na eliminação do elemento que se encontra na frente/cabeça da ***Fila***, tornando o elemento que está a seguir na frente/cabeça da ***Fila***. Quando for retirado o último elemento da ***Fila***, esta fica vazia, sendo apenas possível efetuar a operação de inserção de elementos na ***Fila***.

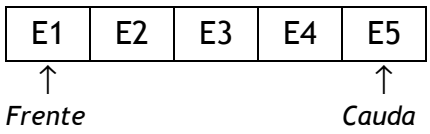



**Figura 30 - Representação esquemática duma Fila.**

## 5.1. Definição axiomática

**Estrutura Fila (Info)**

Declarar

| | | |
|---|---|---|
| Criar() | → | Fila |
| Vazia(Fila) | → | 1 (verdadeiro) ou 0 (falso) |
| Juntar(Info, Fila) | → | Fila |
| Remover(Fila) | → | Fila |
| Frente(Fila) | → | Info |

tais que $\forall$ Q $\in$ Fila, $\forall$ X $\in$ Info sejam

Vazia(Criar ()) = 1 (verdadeiro)

Vazia(Juntar(X, Q)) = 0 (falso)

Remover(Criar()) = ERRO

Remover(Juntar(X,Q)) = *Se* Vazia(Q) *Então* NULL *Senão* Juntar(X, Remover(Q))

Juntar(X, Remover(Q)) = *Se* Vazia(Q) *Então* ERRO *Senão* Remover(Juntar(X,Q))

Frente(Criar()) = ERRO

Frente(Juntar(X, Q)) = *Se* Vazia(Q) *Então* X *Senão* Frente(Q)

**Fim Fila**

## 5.2. Operações

As operações que compõem esta ***EAD Fila*** são as seguintes:

1. Criar uma ***Fila*** Q (vazia): Criar(Q)
2. Verificar se uma ***Fila*** Q está vazia: Vazia(Q)
3. Colocar um novo elemento X na cauda de uma ***Fila*** Q: Juntar(X, Q)
4. Remover o elemento que está à frente de uma ***Fila*** Q: Remover(Q)
5. Fornecer o elemento à frente de uma ***Fila*** Q: Frente(Q)

Tal como uma ***Pilha***, também uma ***Fila*** pode ser implementada usando listas, sejam elas de armazenamento sequencial (usando *arrays*) ou não sequencial (usando listas ligadas). Neste documento apenas se irá fazer referência à implementação usando listas ligadas simples e duplas.

Na implementação de uma ***Fila*** podem-se usar listas ligadas simples ou duplas. No entanto, enquanto que quando a ***Fila*** é implementada usando listas ligadas simples apenas se utiliza um ponteiro associado à frente da ***Fila*** (***Frente***); quando a ***Fila*** é implementada usando listas ligadas duplas utilizam-se dois ponteiros: um associado à frente/cabeça da ***Fila*** (***Frente***) e um outro associado à cauda da ***Fila*** (***Cauda***).

## 5.3. Implementação usando listas ligadas simples

Considere as seguintes declarações:

```
struct Nodo {
        Info  Elemento;
        struct  Nodo  *Prox;
};
typedef  struct  Nodo  *PNodo;
```

Na implementação de uma ***Fila*** usando listas ligadas simples, cada nodo aponta para o nodo seguinte, sendo que o último nodo da lista (nodo da cauda da fila) aponta para *NULL* para servir de indicador de fim da ***Fila*** (ver Figura 31).

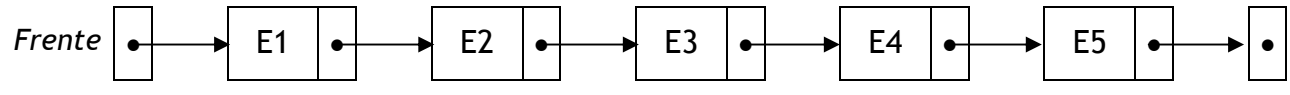



**Figura 31 – Representação de uma Fila usando uma lista ligada simples.**

Tal como acontece na ***Pilha***, também na ***Fila*** (implementação usando listas ligadas simples) a memória para os nodos e para os elementos é atribuída quando um elemento é inserido na ***Fila*** e é libertada quando um elemento é removido da ***Fila***. O indicador de

frente/cabeça da ***Fila*** é um ponteiro que aponta para o elemento mais antigo (em termos de inserção na ***Fila***) da ***Fila*** e que será o primeiro a ser removido. O acesso à cauda da ***Fila*** dá-se a partir do ponteiro que aponta para a frente da ***Fila***, sendo à frente deste que um novo elemento será inserido. Quando o ponteiro é nulo (*NULL*) significa que a ***Fila*** se encontra vazia. Uma ***Fila*** nunca está cheia, mas pode acontecer que não haja memória suficiente para alocar um novo elemento com o objetivo de ser inserido na ***Fila***.

### 5.3.1. Criar uma Fila (vazia)

A implementação da operação associada à criação de uma ***Fila***, consiste apenas em apontar o ponteiro *Q* para *NULL* (ver Figura 32).

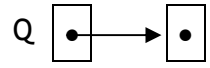



**Figura 32 – Criar uma Fila (apontar o ponteiro *Q* para NULL).**

A seguir apresenta-se uma possível função para criar uma ***Fila*** usando uma lista ligada simples.

*Entrada*: nada

*Saída*: ponteiro nulo (NULL)

```
PNodo CriarFila () {
    PNodo Q;
    Q = NULL;
    return Q;
}
```

### 5.3.2. Verificar se uma Fila está vazia

A implementação da operação associada à verificação se uma ***Fila*** está vazia, consiste apenas em verificar se Q aponta para *NULL*. A seguir apresenta-se uma possível função para verificar se uma ***Fila*** está vazia.

*Entrada*: ponteiro para a frente da ***Fila*** (Q)

*Saída*: 1 (se a fila está vazia); 0 (caso contrário)

```
int FilaVazia (PNodo Q) {
    if (Q == NULL)
        return 1;
    return 0;
}
```

### 5.3.3. Juntar elemento a uma Fila

A operação de inserir/juntar um elemento numa ***Fila*** consiste em acrescentar mais um elemento na cauda da ***Fila*** (ver Figura 33 — uma ***Fila*** de inteiros).

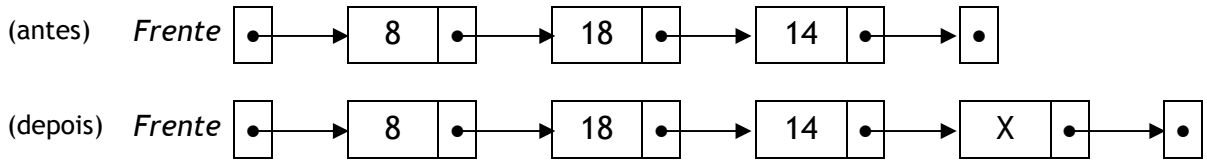



**Figura 33 - Representação da operação inserir o elemento X na Fila.**

A seguir apresenta-se uma função possível para inserir um elemento X numa ***Fila***.

*Entrada*: a informação a ser inserida (X) e um ponteiro para a frente da ***Fila*** (Q)

*Saída*: a ***Fila*** com mais um elemento (X)

```
PNodo Juntar (Info X, PNodo Q) {
    PNodo Qant = Q, QX = CriarNodo(X);
    if (QX == NULL)
        return Q;
    if (FilaVazia(Q))
        return QX;
    while (QAnt→Prox != NULL)
        QAnt = QAnt→Prox;
    QAnt→Prox = QX;
    return Q;
}
```

### 5.3.4. Remover elemento de uma Fila

A operação para remover elemento duma ***Fila*** consiste em remover o elemento que se encontra na frente/cabeça da ***Fila*** e atualizar o ponteiro de frente da v (ver Figura 34 — uma *Fila* de inteiros).

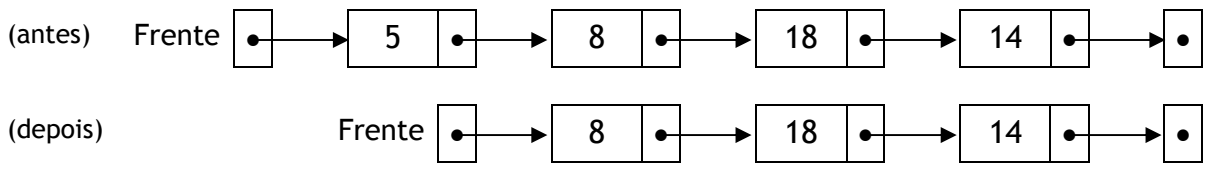



**Figura 34 - Representação da operação remover um elemento da Fila.**

A seguir apresenta-se uma possível função para remover o elemento da frente/cabeça duma ***Fila***.

*Entrada*: um ponteiro para frente da ***Fila*** (Q)

*Saída*: ponteiro de frente da ***Fila*** atualizado (para o nodo seguinte)

```
PNodo Remover (PNodo Q) {
        PNodo QAux = Q;
        QAux = QAux →Prox;
        QAux = LibertarNodo(QAux);
        return Q;
}
```

### 5.3.5. Consultar elemento de uma Fila

A operação para consultar uma ***Fila*** consiste em devolver o elemento da frente/cabeça da ***Fila*** (o ponteiro de frente de fila fica inalterado, pois não há remoção do elemento). A seguir apresenta-se uma possível função para consultar uma ***Fila***.

*Entrada*: um ponteiro para a frente/cabeça da ***Fila*** (Q)

*Saída*: o elemento que se encontra na frente/cabeça da ***Fila***

```
Info Frente (PNodo Q) {
        return (Q→Elemento);
}
```

## 5.4. Implementação usando listas ligadas duplas

Considere as seguintes declarações:

```
struct NodoLD {
        Info  Elemento;
        struct  NodoLD  *Ant;
        struct  NodoLD  *Prox;
};
typedef  struct  NodoLD  *PNodoLD;
```

Na implementação de uma ***Fila*** usando listas ligadas duplas são necessários dois ponteiros: um para a frente e um para a cauda da ***Fila***. Neste tipo de implementação, cada nodo aponta para o nodo seguinte e para o nodo anterior. O primeiro nodo (frente) aponta para *NULL* (ponteiro anterior) servindo de início da ***Fila***. O último nodo da ***Fila*** (cauda da ***Fila***) aponta para *NULL* (ponteiro seguinte) para servir de indicador de fim da ***Fila*** (ver Figura 35 — uma *Fila* de inteiros).

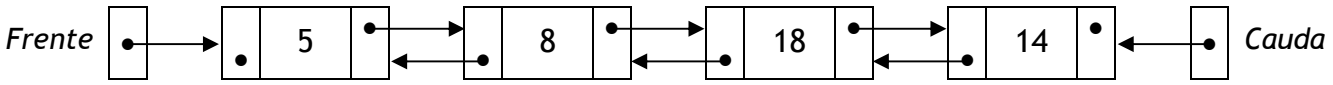



**Figura 35 – Representação de uma Fila usando uma lista ligada dupla.**

Na ***Fila*** com implementação usando listas ligadas simples a memória para os nodos e para os elementos é atribuída quando um elemento é inserido na ***Fila*** e é libertada quando um elemento é removido da ***Fila***. O indicador de frente/cabeça da ***Fila*** é um ponteiro que aponta para o elemento mais antigo (em termos de inserção na ***Fila***) da ***Fila*** e que será o primeiro a ser removido. O indicador de cauda da ***Fila*** é um ponteiro que aponta para a cauda da ***Fila***, sendo à frente deste que um novo elemento será inserido. Quando os ponteiros de frente e de cauda são nulos (*NULL*) significa que a fila se encontra vazia. Uma ***Fila*** nunca está cheia, mas pode acontecer que não haja memória suficiente para alocar um novo elemento com o objetivo de ser inserido na ***Fila***.

### 5.4.1. Criar uma Fila (vazia)

A implementação da operação associada à criação de uma ***Fila***, consiste apenas em fazer os ponteiros *Frente* e *Cauda* da ***Fila*** apontarem para *NULL* (ver Figura 36).

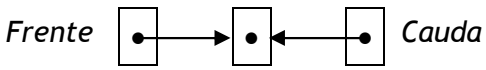



**Figura 36 – Criar uma Fila (apontar *Frente* e *Cauda* para NULL).**

A seguir apresenta-se uma possível função para criar uma ***Fila*** usando uma lista duplamente ligada.

*Entrada*: ponteiros para a frente (*Frente*) e a cauda (*Cauda*) da ***Fila***

*Saída*: ponteiros nulos (*NULL*)

```
void CriarFilaLD (PNodoLD *Frente, PNodoLD *Cauda) {
    *Frente = NULL;
    *Cauda = NULL;
}
```

### 5.4.2. Verificar se uma Fila está vazia

A implementação da operação associada à verificação se uma ***Fila*** está vazia, consiste apenas em verificar se algum dos ponteiros *Frente* e *Cauda* é nulo (aponta para *NULL*). A seguir apresenta-se uma possível função para verificar se uma ***Fila*** está vazia.

*Entrada*: ponteiro para a frente (*Frente*) ou para a cauda da fila (*Cauda*) da ***Fila***

*Saída*: 1 (se a pilha está vazia); 0 (caso contrário)

```
int FilaVaziaLD (PNodoLD Q) {
    if (Q == NULL)
        return 1;
    return 0;
}
```

### 5.4.3. Juntar um elemento a uma Fila

A operação de inserir/juntar um elemento numa ***Fila*** consiste em acrescentar mais um elemento na cauda da ***Fila*** (ver Figura 37 - uma ***Fila*** de inteiros).

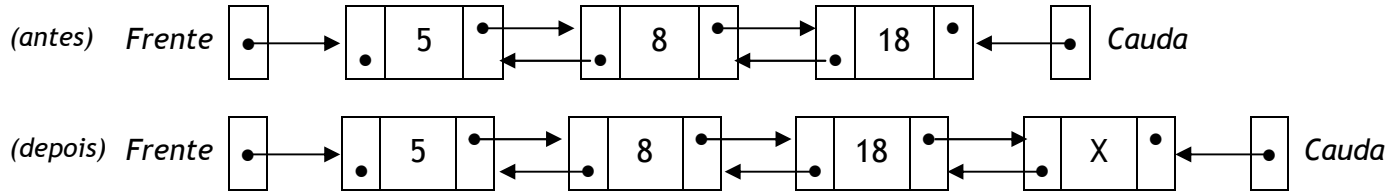



**Figura 37 – Representação da operação inserir o elemento X numa Fila.**

A seguir apresenta-se uma função possível para inserir um elemento X numa ***Fila***.

*Entrada*: a informação a ser inserida (X) e ponteiros para a frente (*Frente*) e a cauda (*Cauda*) da ***Fila***

*Saída*: 1 (inserção com sucesso) ou 0 (inserção sem sucesso); a ***Fila*** com mais um elemento (X), se inserção com sucesso

```
int JuntarLD (Info X, PNodoLD *Frente, PNodoLD *Cauda) {
    PNodoLD  QAux = CriarNodoLD(X);   // ver listas duplamente ligadas
    if (QAux == NULL)
        return 0;                     // inserção não realizada
    if (FilaVaziaLD(*Frente)) {       // primeiro elemento a entrar na Fila
        *Cauda = QAux;
        *Frente = QAux;
        return 1;
    }
    (*Cauda)→Prox = QAux;
    QAux →Ant = *Cauda;
    *Cauda = QAux;
    return 1;
}
```

### 5.4.4. Remover um elemento de uma Fila

A operação para remover elemento duma ***Fila*** consiste em remover o elemento que se encontra na frente/cabeça da ***Fila*** e atualizar o ponteiro de frente da ***Fila*** (ver Figura 38 - uma ***Fila*** de inteiros).

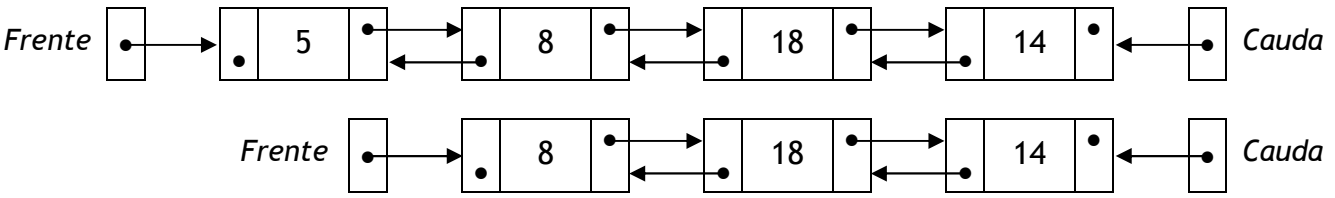



**Figura 38 – Representação da operação remover um elemento da Fila.**

A seguir apresenta-se uma possível função que permite remover o elemento da frente/cabeça duma ***Fila***.

*Entrada*: ponteiros para a frente (*Frente*) e a cauda (*Cauda*) da ***Fila***

*Saída*: ***Fila*** com menos um elemento (o da frente) e ponteiro de frente da ***Fila*** atualizado (para o nodo seguinte)

```
PNodoLD RemoverLD (PNodoLD Q) {
    PNodoLD  QAux = Q;
    Q = Q→Prox;
    Q→Ant = NULL;
    QAux = LibertarNodoLD(QAux);
    return Q;
}
```

### 5.4.5. Consultar o elemento à cabeça duma Fila

A operação para consultar uma ***Fila*** consiste em devolver o elemento que se encontra na frente/cabeça da ***Fila*** (o ponteiro de frente da ***Fila*** fica inalterado, pois não há remoção do elemento). A seguir apresenta-se uma possível função para esta operação.

*Entrada*: um ponteiro para a frente (*Frente*) da ***Fila***

*Saída*: o elemento que se encontra na frente da ***Fila***

```
Info FrenteLD (PNodoLD Frente) {
    return (Frente→Elemento);
}
```

# 6. Listas com saltos

As Listas com Saltos ("*Skip List*") foram propostas em 1989 por William Pugh, professor da Universidade de Maryland. Estas listas permitem operações de consulta, inserção e remoção mais eficientes do que as listas ligadas simples e duplas. No entanto, o aumento na eficiência é obtido à custa de um aumento de recursos (memória) necessários. A ideia base é a seguinte:

- Procurar o nome de "Manuel Silva" numa lista de nomes ordenados alfabeticamente, que constam num dado documento;
- Ao abrir o documento, verificamos que estamos apenas nos nomes cuja inicial é "A";
- Acção inteligente:
  - Avançar 1 página (provavelmente avançaremos para uma página com a mesma letra) - ERRADO
  - Avançar N páginas! (provavelmente avançaremos para a letra h?, j?) – CORRETO

## 6.1. Definição

Uma lista com saltos define-se como uma lista ligada simples em que cada nodo tem um número variável de *ligações*, sendo que cada nível de *ligação* implementa uma lista ligada simples formada por um número diferente de nodos. Está-se, pois, perante uma estrutura de listas paralelas, todas elas terminadas com o ponteiro nulo (*NULL*), que permitem implementar diferentes travessias, sendo que cada travessia avança sobre os nodos com um número de *ligações* inferior ao nível em que ela decorre. Quanto maior é o nível da travessia, maior é o espaçamento de nodos percorridos, sendo que uma travessia pode descer de nível para prosseguir de uma forma menos espaçada ao longo da lista. Logo, está-se perante uma lista ligada com atalhos. Apesar da travessia ao longo de um nível ser sequencial, o facto dos níveis terem espaçamentos diferentes, tanto maiores quanto maior é o nível, permite obter travessias mais eficientes do que a pesquisa sequencial.

Existem várias formas de implementar uma lista com saltos, dependendo da forma como se armazenam os ponteiros para cada nível. Desta forma, os ponteiros associados às listas com saltos podem ser guardados usando um *array* estático ou dinâmico, ou então listas ligadas. Quando se usa um *array* estático, a lista de ponteiros é definida com o máximo de elementos (que corresponde ao nível máximo de um nodo); isto significa que alguns ponteiros não irão ser usados (os elementos da lista não devem ser todos do mesmo

nível, pois este é escolhido aleatoriamente). No caso de usar-se um *array* dinâmico, não há desperdício de memória, pois o tamanho da lista de ponteiros é definida em função das suas necessidades (do nível determinado aleatoriamente). Neste texto, vai-se usar um *array* dinâmico, pelo que a sua declaração, em linguagem C, pode ser da seguinte forma:

```
typedef struct NodoSkip *PNodoSkip;
struct NodoSkip {
    Info     Elemento;
    int      Niveis;
    PNodoSkip  *PtProx;
};
```

Uma vez que o número de ponteiros é variável de nodo para nodo, então o nodo de uma lista com saltos precisa, para além de um campo ***Elemento*** associado aos dados que se pretendem guardar, de mais dois campos: o campo ***Niveis***, associado ao número de níveis do elemento (corresponde também ao número de ponteiros que tem que guardar) e o campo ***PtProx***, que é um ponteiro para uma sequência (*array*) de ponteiros para os nodos seguintes da lista com saltos, que é alocada aquando da criação do nodo (Figura 39). Os níveis são numerados de 1 até ***Niveis***.

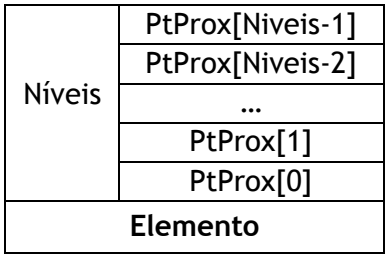



**Figura 39 – Representação esquemática dum nodo duma lista com saltos.**

A Figura 40 apresenta uma representação esquemática de uma lista com saltos cujos elementos são valores inteiros. O primeiro nodo/elemento é o que se encontra à cabeça da lista: valor 0, 5 níveis e, consequentemente, 5 ponteiros para nodos seguintes.

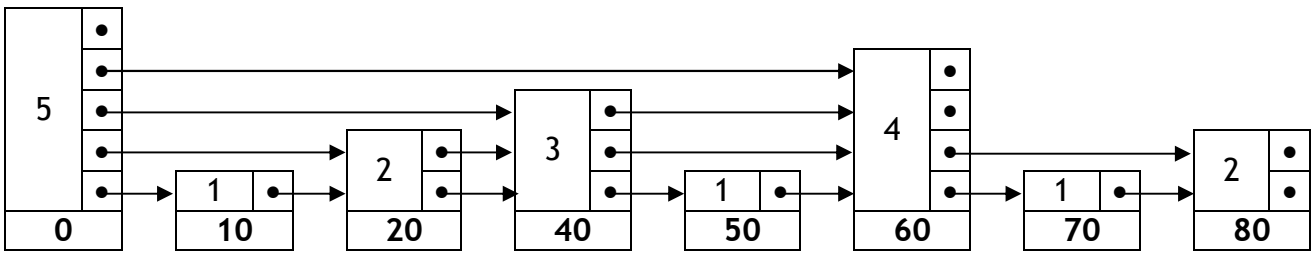



**Figura 40 - Representação esquemática duma lista com saltos (ordenada crescentemente).**

## 6.2. Operações sobre uma lista com saltos

Uma lista com saltos caracteriza-se pelos seguintes parâmetros:

- ***cabeça da lista*** (*PCab*), que é um nodo extra e que contém um valor sempre inferior a qualquer um dos valores contidos nos restantes nodos da lista;
- ***máximo de níveis*** da lista (*MaxNiv*), que corresponde ao número de níveis da cabeça.

### 6.2.1. Criar um nodo

*Entrada*: um elemento X e o seu nível (niv)

*Saída*: um ponteiro para um nodo (informação + ligações)

```
PNodoSkip CriarNodoSkip (Info X, int niv) {
    int  k;
    PNodoSkip  P = (PNodoSkip) malloc (sizeof(struct NodoSkip));
    if (P == NULL)
        return  NULL;
    P→PtProx = (PNodoSkip *) malloc (niv * sizeof(PNodoSkip));
    if (P→PtProx == NULL) {
        P = LibertarNodoSkip(P);
        return NULL;
    }
    P→Elemento = X;
    P→Niveis = niv;
    for (k = 0; k < niv; k++)
        P→PtProx[k] = NULL;
    return  P;
}
```

### 6.2.2. Determinar o nível dum elemento (nodo) duma lista

O desempenho de uma lista com saltos depende da composição das diferentes listas ligadas. Para se obter uma pesquisa com eficiência logarítmica, O(log n), a lista deve ter listas ligadas com espaçamentos do tipo $2^{MaxNiv}$. No entanto, como este critério rígido dificultaria a operação de inserção de elementos na lista, é preferível decidir o nível do nodo aleatoriamente, mas assegurando uma distribuição probabilística que distribua os elementos pela lista da seguinte forma: fator = $1/(2^{MaxNiv}-1)$ para o último nível (MaxNiv),

$2^1$xfator para o nível MaxNiv-1, $2^2$xfator para o nível MaxNiv-2, $2^3$xfator para o nível MaxNiv-3, … (ou seja, $2^k$xfator para o nível MaxNiv-k).

Apresenta-se, a seguir, uma possível função para gerar um número aleatório que cumpra com os requisitos referidos para a geração do nível de um elemento.

*Entrada*: nível máximo a considerar (*maxNiv*)

*Saída*: o nível do elemento (gerado aleatoriamente)

```
int GerarNivelAleatorio (int maxNiv) {
    int  i, Niv = maxNiv;
    float  t, fator = 1/(pow(2,maxNiv)-1);      // probabilidade do último nível
    srand(time(NULL));
    t = rand();
    for (i = 0; i < maxNiv-1; i++) {
        if (t <= pow(2,i)*fator)
            return Niv;
        Niv--;
    }
    return 1;
}
```

### 6.2.3. Criar uma lista

Criar uma lista com saltos consiste em criar um nodo associado à cabeça da lista, cujos valores a atribuir aos campos são os seguintes:

- Elemento = X, tal que X é menor/maior do que qualquer elemento da lista ordenada crescentemente/decrescentemente;
- Niveis = MaxNiv;
- PtProx[k] = NULL, k = 0,…,MaxNiv-1

Uma possível função para criar uma lista com saltos, é a seguinte:

*Entrada*: um elemento X (X menor da lista) e máximo de níveis a considerar (MaxNiv)

*Saída*: um ponteiro para um nodo (cabeça da lista)

```
PNodoSkip CriarListaSkip (Info X, int maxNiv) {
    PNodoSkip  P;
    P = CriarNodoSkip(X, maxNiv);
    return  P;
}
```

### 6.2.4. Libertar/destruir um nodo de uma lista

A destruição de um nodo de uma lista consiste em atribuir a todos os ponteiros dos campos associados aos nodos seguintes de cada nível o ponteiro nulo (*NULL*), libertar para o sistema o ponteiro para a sequência daqueles ponteiros e, por fim, libertar o ponteiro para o nodo a destruir. A seguir apresenta-se uma possível função para destruir um nodo.

*Entrada*: um ponteiro para o nodo que se pretende destruir

*Saída*: o ponteiro a apontar para *NULL*

```
PNodoSkip LibertarNodoSkip (PNodoSkip P) {
    int  k;
    for (k = 0; k < P→Niveis; k++)
        P→PtProx[k] = NULL;
    free(P→PtProx);
    free(P);
    P = NULL;
    return P;
}
```

### 6.2.5. Destruição de uma lista

No algoritmo proposto para destruir uma lista com saltos, a travessia da lista é feita no nível zero, para destruir todos os seus nodos, ficando apenas a cabeça da lista, de forma sequencial. Uma possível função para destruir uma lista com saltos é a que se segue.

*Entrada*: um ponteiro para a cabeça da lista

*Saída*: a lista vazia (cabeça da lista com todos os seus ponteiros seguintes nulos)

```
void DestruirListaSkip (PNodoSkip PCab) {
    int  k;
    PNodoSkip PAux, P = PCab→PtProx[0];
    while (P != NULL) {
        PAux = P;
        P = P→PtProx[0];
        PAux = LibertarNodoSkip(PAux);
    }
    for (k = 0; k < PCab→Niveis; k++)
        PCab→PtProx[k] = NULL;
}
```

### 6.2.6. Pesquisa dum elemento numa lista

A pesquisa de um elemento segue a seguinte estratégia:

- começar no ponteiro de maior nível na cabeça da lista; percorrer até ao nodo que seja igual ou que exceda o elemento que se procura;
- se encontro um elemento igual, terminar com sucesso; senão, descer de nível e usar a mesma estratégia;
- se depois de atingir o nível mais baixo (1) não encontrar o nodo com o elemento procurado, então concluir que ele não está presente.

Tome-se como exemplo a lista representada na Figura 40, na qual se pretende pesquisar os elementos com os valores 80 (Figura 41), 50 (Figura 42) e 30 (Figura 43).

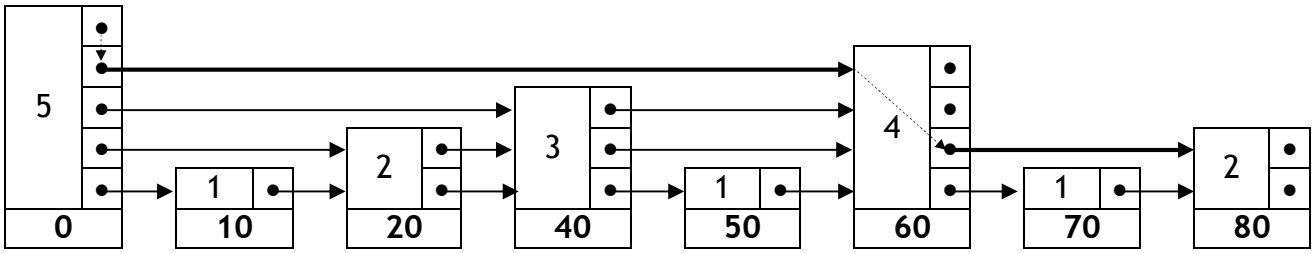



**Figura 41 - Pesquisa do elemento 80 numa lista ordenada crescentemente.**

- A travessia começa (sempre) pela cabeça da lista e no nível máximo (neste caso é 5);
- Como o ponteiro no nível 5 é nulo (PtProx[4] = NULL), passar para o nível 4;
- Como o ponteiro no nível 4 (PtProx[3]) aponta para um nodo com valor 60 e este valor é menor do que o valor procurado, 80, a travessia avança para nodo com valor 60;
- Como os ponteiros nos níveis 4 e 3 do nodo com 60 são nulos (PtProx[3] = PtProx[2] = NULL), passar para o nível 2;
- Como o ponteiro do nível 2 (PtProx[1]) aponta para um nodo com valor 80 e este é o valor procurado, termina a pesquisa com sucesso, após 2 comparações.

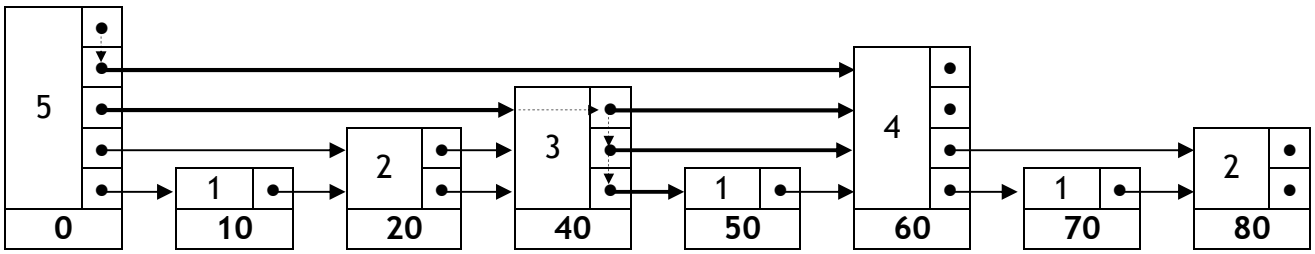



**Figura 42 - Pesquisa do elemento 50 numa lista ordenada crescentemente.**

- A travessia começa (sempre) pela cabeça da lista e no nível máximo (neste caso é 5);
- Como o ponteiro no nível 5 é nulo (PtProx[4] = NULL), passar para o nível 4;

- Como o ponteiro no nível 4 (PtProx[3]) aponta para um nodo com valor 60 e este valor é maior do que o valor procurado, 50, a travessia mantém-se na cabeça da lista e desce 1 nível, para o nível 3;
- Como o ponteiro no nível 3 (PtProx[2]) aponta para um nodo com valor 40, que é menor do que o valor procurado, 50, a travessia avança para o nodo com valor 40 e continua no mesmo nível 3;
- Como o ponteiro no nível 3 do nodo com 40 (PtProx[2]) aponta para um nodo com valor 60 e este valor é maior do que o procurado, 50, a travessia mantém-se no nodo com valor 40 e desce 1 nível, para o nível 2;
- Como o ponteiro no nível 2 do nodo com valor 40 (PtProx[1]) aponta de novo para um nodo com valor 60 e este valor é maior do que o procurado, 50, a travessia mantém-se no nodo com valor 40 e desce novamente 1 nível, para o nível 1;
- Como o ponteiro no nível 1 do nodo com 40 (PtProx[0]) aponta para um nodo com valor 50 e este é o valor procurado, termina a pesquisa com sucesso, após 5 comparações.

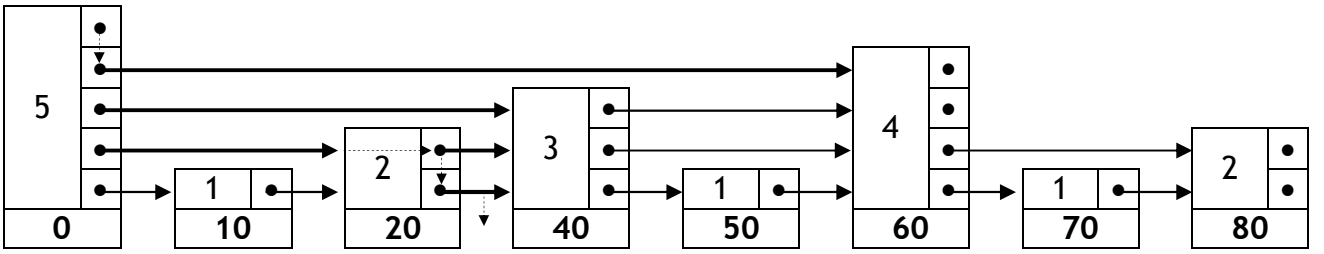



**Figura 43 - Pesquisa do elemento 30 numa lista ordenada crescentemente.**

- A travessia começa (sempre) pela cabeça da lista e no nível máximo (neste caso é 5).
- Como o ponteiro no nível 5 é nulo (PtProx[4] = NULL), passar para o nível 4.
- Como o ponteiro no nível 4 (PtProx[3]) aponta para um nodo com valor 60 e este valor é maior do que o valor procurado, 30, a travessia mantém-se na cabeça da lista e desce 1 nível, para o nível 3.
- Como o ponteiro no nível 3 (PtProx[2]) aponta para um nodo com valor 40, que é maior do que o valor procurado, 30, a travessia mantém-se na cabeça da lista e desce novamente 1 nível, para o nível 2.
- Como o ponteiro no nível 2 (PtProx[1]) aponta para um nodo com valor 20 e este valor é menor do que o procurado, 30, a travessia avança para o nodo com valor 20 e continua no mesmo nível 2.

- Como o ponteiro no nível 2 (PtProx[1]) do nodo com 20 aponta para um nodo com valor 40 e este valor é maior do que o procurado, 30, a travessia mantém-se no nodo com valor 20 e desce 1 nível, para o nível 1.
- Como o ponteiro no nível 1 (PtProx[0]) aponta de novo para um nodo com valor 40 e este valor é maior do que o procurado, 30, a travessia mantém-se no nodo com valor 20 e desce 1 nível, para o nível 0.
- Como se chegou a um nível não válido (0), a pesquisa termina sem sucesso, após 5 comparações.

A seguir apresenta-se uma possível função para pesquisar um elemento numa lista.

*Entrada*: um ponteiro para a cabeça da lista e o elemento a pesquisar

*Saída*: 1 (elemento está na lista); 0 (elemento não está na lista)

```
int PesquisarElementoListaSkip (PNodoSkip PCab, Info X) {
    int k;
    PNodoSkip P = PCab;
    for (k = PCab→Niveis; k > 0; k--) {
        while (P→PtProx[k-1] != NULL && (P→PtProx[k-1])→Elemento < X)
            P = P→PtProx[k-1];
        if (P→PtProx[k-1] != NULL && (P→PtProx[k-1])→Elemento == X)
            return 1;
    }
    return 0;
}
```

### 6.2.7. Pesquisa do nodo anterior dum elemento duma lista

As operações de inserir e remover elemento duma lista precisam de determinar quais os ponteiros para cada nível dos nodos anteriores ao elemento a inserir (quando este for colocado na lista) ou a remover. Uma possível função para esta operação suplementar apresenta-se a seguir, a qual tem como pré-requisito a existência do elemento na lista.

*Entrada*: um ponteiro para a cabeça da lista e o elemento a pesquisar

*Saída*: sequência de ponteiros para os elementos anteriores ao elemento a procurar

```
PNodoSkip PesquisarAnteriorListaSkip (PNodoSkip PCab, Info X) {
    int k;
    PNodoSkip *PAnt, P = PCab;
    PAnt = (PNodoSkip *) malloc(PCab→Niveis * sizeof(PNodoSkip));
    if (PAnt == NULL)
        return NULL;
    for (k = PCab→Niveis; k > 0; k--) {
        while (P→PtProx[k-1] != NULL && (P→PtProx[k-1])→Elemento < X)
            P = P→PtProx[k-1];
        PAnt[k-1] = P;
    }
    return PAnt;
}
```

### 6.2.8. Inserção de um elemento numa lista

A inserção de um nodo numa lista com saltos implica que todas as listas ligadas que são intercetadas pelo nodo a inserir, têm de ser atualizadas. A inserção de um elemento numa lista com saltos segue a seguinte estratégia:

1) alocar o novo nodo;
2) escolher aleatoriamente o nível do nodo seguindo uma distribuição de probabilidade;
3) determinar quais serão os nodos anteriores, para cada nível, ao novo nodo (a inserir);
4) inserir o novo nodo na lista.

O algoritmo proposto para a inserção de um elemento (nodo) numa lista com saltos ordenada crescentemente realiza uma travessia pela lista com saltos, começando na cabeça da lista e no nível máximo, até chegar ao local de inserção e ao nível mais baixo (1), sendo nesta altura que o processo de atualização das ligações se inicia; só após a conclusão daquelas ligações é que o nodo é efetivamente inserido e o algoritmo termina. Os ponteiros para cada nível (associados a cada lista ligada) determinam o local de inserção do novo nodo (isto é, o seu nodo anterior), fazendo depois as ligações do novo nodo ao nodo seguinte do seu nodo anterior, e do nodo anterior ao novo nodo. No caso de haver elementos repetidos na lista, este algoritmo insere o novo nodo antes de eventuais nodos com o mesmo valor (usa a comparação "<").

A Figura 44 apresenta a inserção numa lista - (a) - do elemento 40 (com 3 níveis) - (b); e a inserção na lista anterior - (b) - do elemento 60 (com 4 níveis) - (c).

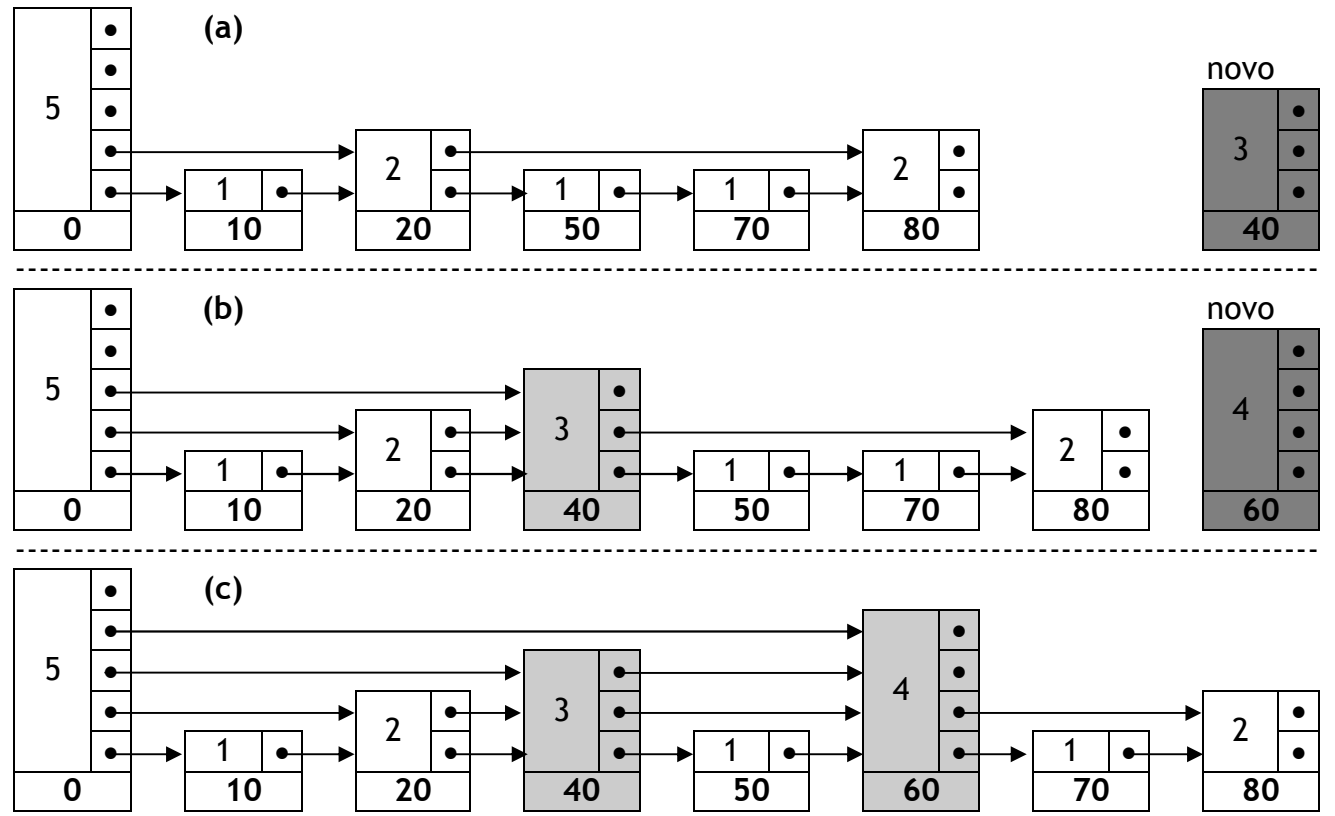



**Figura 44 – Inserir elementos (40 e 60) numa lista ordenada crescentemente.**

A seguir apresenta-se uma possível função para inserir um elemento numa lista.

*Entrada*: um ponteiro para a cabeça da lista e o elemento a inserir

*Saída*: a lista atualizada

```
void InserirElementoListaSkip (PNodoSkip PCab, Info X) {
    PNodoSkip  P, *PAnt;
    int  Nivel, k;
    if (PCab→Elemento < X)   // o valor da cabeça da lista tem de ser o menor
        PCab→Elemento = X – 100;       // atualizar o valor da cabeça da lista
    Nivel = GerarNivelAleatorio(MaxNiv);
    P = CriarNodoSkip(X, Nivel);
    if (P == NULL)   // criação de nodo sem sucesso - lista sem alteração
        return PCab;
    PAnt = ProcurarAnteriorListaSkip(PCab, X);
    for (k = Nivel; k > 0; k--) {
        P→PtProx[k-1] = PAnt[k-1]→PtProx[k-1];
        PAnt[k-1]→PtProx[k-1] = P;
    }
}
```

### 6.2.9. Remoção de um elemento da lista

A remoção de um elemento/nodo duma lista com saltos implica que todas as listas ligadas que são intercetadas pelo nodo a remover, têm de ser atualizadas.

A remoção de um elemento numa lista com saltos segue a seguinte estratégia:

1) determinar quais os nodos anteriores, para cada nível, do elemento a remover;
2) remover o nodo da lista.

O algoritmo proposto para a remoção de um elemento (nodo) duma lista com saltos ordenada crescentemente, realiza uma travessia pela lista, começando na cabeça da lista e no nível máximo, até chegar ao nível mais baixo (1) do nodo a remover, sendo nesta altura que o processo de atualização das ligações se inicia; só após a conclusão daquelas ligações é que o nodo é efetivamente removido e o algoritmo termina. Os ponteiros para cada nível (associados a cada lista ligada) determinam os nodos anteriores ao nodo a remover, fazendo depois as ligações destes nodos (anteriores) com os nodos seguintes do nodo a remover, e a libertação do nodo que se pretende remover. No caso de haver elementos repetidos na lista, este algoritmo remove o nodo mais próximo da cabeça da lista (usa a comparação "<").

A seguir apresenta-se uma possível função para remover um elemento duma lista.

*Entrada*: um ponteiro para a cabeça da lista e o elemento a remover

*Saída*: a lista atualizada

```
void RemoverElementoListaSkip (PNodoSkip PCab, Info X) {
    PNodoSkip  P, *PAnt;
    int  Nivel, k;
    PAnt = ProcurarAnteriorListaSkip(PCab, X);
    P = PAnt[0]→PtProx[0];  // nodo a remover
    Nivel = PAnt→Niveis;
    for (k = Nivel; k > 0; k--)
        PAnt[k-1]→PtProx[k-1] = P→PtProx[k-1];
    P = LibertarNodoSkip(P);
}
```

A Figura 45 apresenta a remoção duma lista - (a) - do elemento 20 (com 2 níveis) - (b); e a remoção da lista anterior - (b) - do elemento 80 (com 2 níveis) - (c).

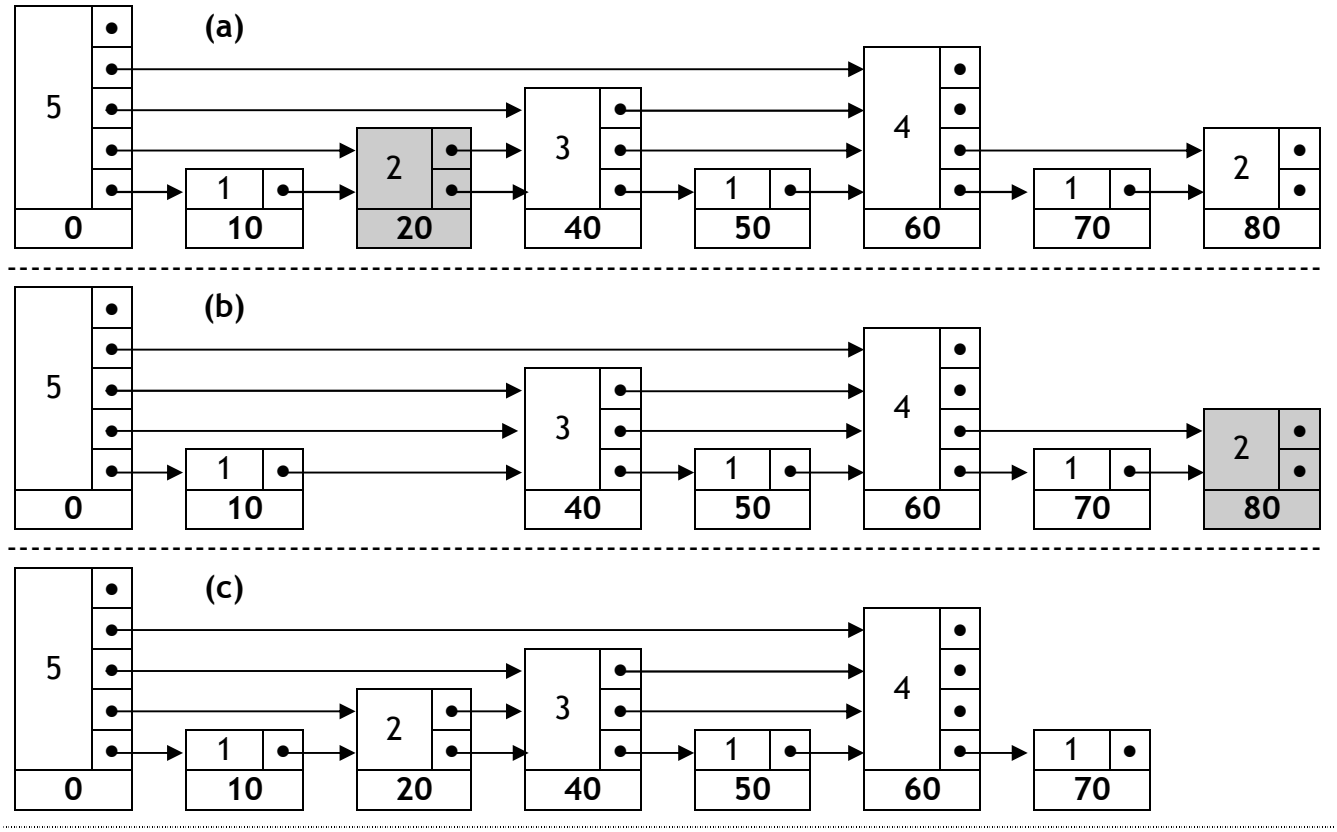



**Figura 45 – Remover elementos (20 e 80) duma lista ordenada crescentemente.**