

Listas Encadeadas

Algoritmo e Estrutura de dados

Prof. Sílvia Campos

(Ref. Estrutura de dados e algoritmos em C++ - Adam Drosdek)

Estrutura de dados

- Explorar as estruturas e investigar seus comportamentos em termos de exigência de tempo (executar operações rapidamente) e espaço (não ocupar muita memória).
- Com a orientação a objetos, iremos encapsular as estruturas de dados dentro de uma classe e tornar público somente o que é necessário para o uso apropriado da classe.

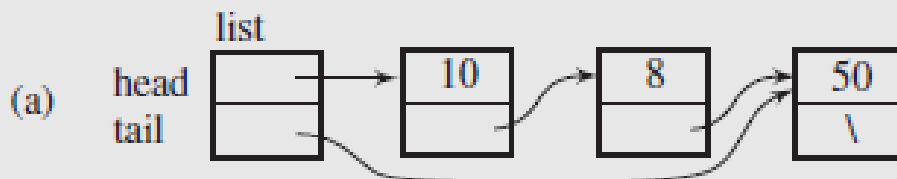
Estrutura de dados

- **Matrizes** – limitações: (1) o tamanho deve ser conhecido no momento da compilação; (2) dados separados na memória do computador pela mesma distância – inserir novos elementos requer a movimentação de outros dados.
- **Estrutura ligada** – coleção de **nós** que armazenam dados, e de **ligações** com outros nós.
 - > os **nós** podem estar em qualquer lugar na **memória**;
 - > passagem de um nó para outro através do **armazenamento dos endereços** de outros nós.
 - > geralmente implementadas por ponteiros.
- **Tipos mais comuns:** Listas simplesmente encadeadas; listas duplamente encadeadas e listas circulares.

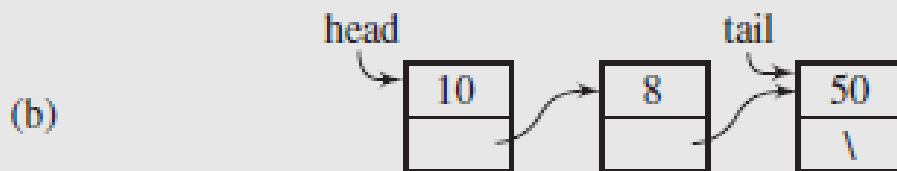
Lista singularmente encadeada

(SLL – Singly Linked List)

- Estrutura de dados composta de nós, cada **nó** contendo alguma **informação** e um **ponteiro** para outro nó na lista.
- Cada nó tem um vínculo somente para o seu sucessor na sequência. ***Exemplo de uma lista de inteiros:***




O primeiro objeto não é parte da lista; ele permite ter acesso a ela.



Apenas os nós pertencentes à lista são mostrados. O nó de acesso é omitido e os membros *head* e *tail* são apresentados.

- Classe para os nós da lista

Cada nó é
uma
instância da
classe:

```
class IntSLLNode {  
public:  
    IntSLLNode() {  
        next = 0;  
    }  
    IntSLLNode(int el, IntSLLNode *ptr = 0) {  
        info = el; next = ptr;  
    }  
    int info;   
    IntSLLNode *next;  
};
```

- Um nó inclui dois membros de dados: *info* e *next*:
 - > *info* : guarda a informação
 - > *next* : vincula os nós da lista. É um ponteiro para um nó do tipo que está sendo definido. Os objetos que incluem este membro de dados são chamados autorreferenciais.

- **Dois Construtores:**

```
class IntSLLNode {  
public:  
    IntSLLNode() {  
        next = 0;  
    }  
    IntSLLNode(int el, IntSLLNode *ptr = 0) {  
        info = el; next = ptr;  
    }  
    int info;  
    IntSLLNode *next;  
};
```

Inicializa **next** para nulo e deixa o valor de **info** não especificado

Toma dois argumentos.
Inicializa **info** e **next**.

Também aceita um argumento numérico. **info** é inicializado para o argumento e **next** para nulo.

- Cada nó deve ser inicializado apropriadamente e incorporado na lista.

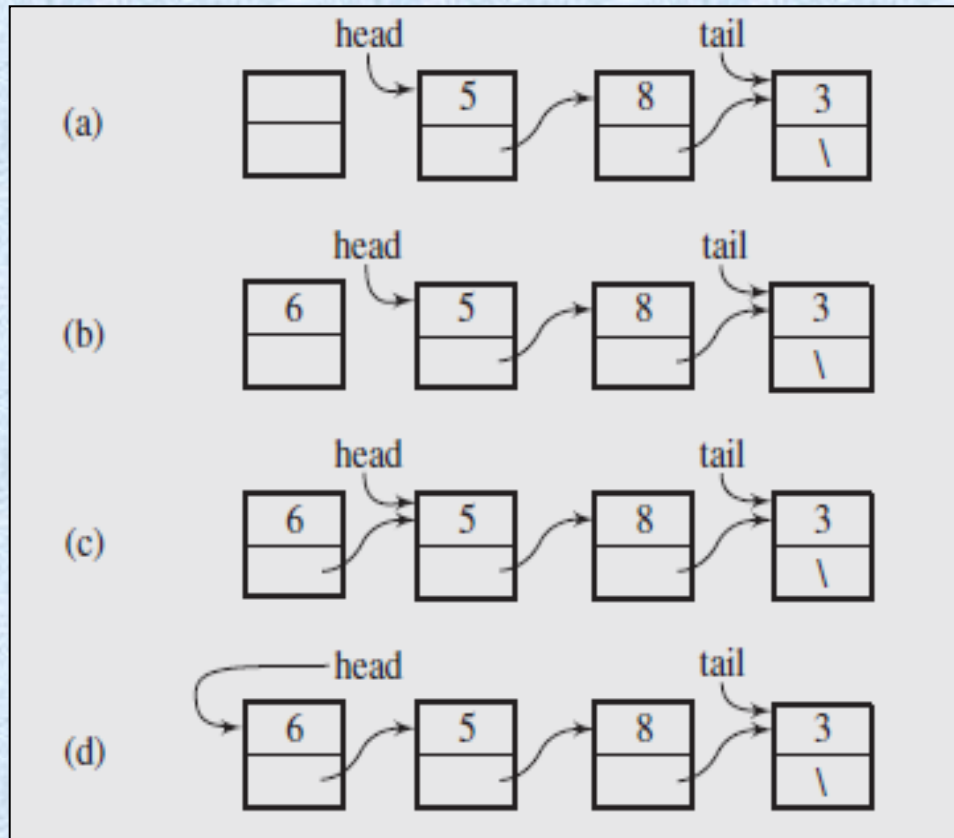
- **Classe para acesso à lista:**

```
class IntSLList {  
public:  
    IntSLList() {  
        head = tail = 0;  
    }  
    ~IntSLList();  
    int isEmpty() {  
        return head == 0;  
    }  
    void addToHead(int);  
    void addToTail(int);  
    int deleteFromHead(); // delete the head and return its info;  
    int deleteFromTail(); // delete the tail and return its info;  
    void deleteNode(int);  
    bool isInList(int) const;  
  
private:  
    IntSLLNode *head, *tail;  
};
```

Funções membro para manipulação da lista

Ponteiros para o próximo e último nós da lista.

Inserindo um novo nó no início da lista:



(a) Um nó vazio é criado

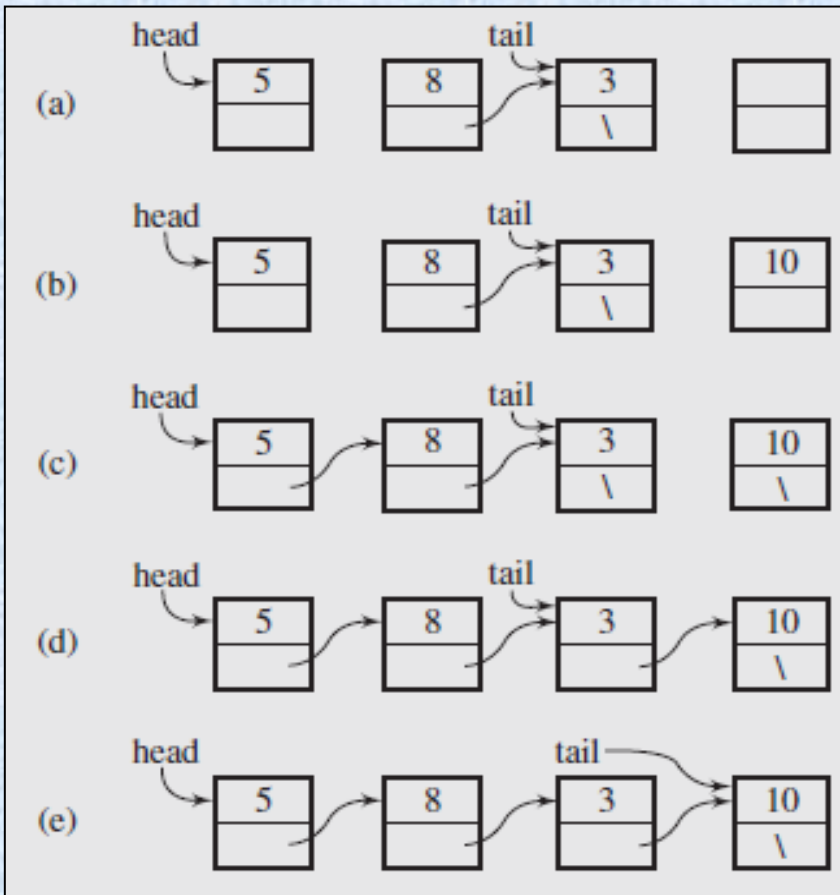
(b) O membro **info** do nó é inicializado para um inteiro em particular

(c) O membro **next** se torna um ponteiro para o primeiro nó da lista: o valor corrente de **head**.

(d) O novo nó precede todos os outros nós da lista. Assim, **head** é atualizado para se tornar um ponteiro para o novo nó.

```
void IntSLList::addToHead(int el) {  
    head = new IntSLLNode(el, head);  
    if (tail == 0)  
        tail = head;  
}
```


Inserindo um novo nó no final da lista:



(a) Um nó vazio é criado

(b) O membro **info** do nó é inicializado para um inteiro em particular

(c) O membro **next** é ajustado para nulo porque o nó é incluído no final da lista.

(d) O novo nó é incluído na lista, fazendo o membro **next** do último nó da lista um ponteiro para o nó recém-criado.

(e) O novo nó é incluído no final da lista. O valor de **tail** se torna o ponteiro para o novo nó.

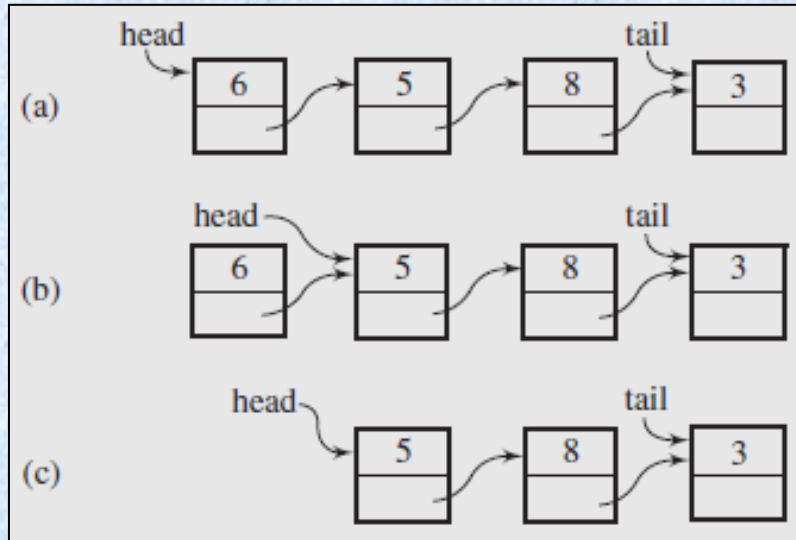
```
void IntSLList::addToTail(int el) {  
    if (tail != 0) { // if list not empty;  
        tail->next = new IntSLLNode(el);  
        tail = tail->next;  
    }  
    else head = tail = new IntSLLNode(el);  
}
```

tail: possibilita acesso imediato ao último nó da lista, sem precisar percorrer toda a lista.

Processo de inserção na lista

- O processo de inserir um novo nó no início ou no final de uma lista é semelhante, pois a implementação de *IntSLLList* usa dois membros ponteiros: **head** e **tail**. Logo,
- `addtoHead()` e `addtoTail()` podem ser executados em tempo constante **$O(1)$** , ou seja, **independente do número de nós na lista**, o número de operações por essas funções membro não excedem algum número constante c .
- Se o ponteiro **tail** não fosse usado, adicionar um nó no final da lista seria mais complicado pois seria necessário varrer toda a lista para atingir o último nó e então adicionar o novo nó. Tempo $O(n)$: linearmente proporcional ao comprimento da lista.

Removendo um nó no início da lista:



(a) A informação do primeiro nó é temporariamente armazenada em uma variável local e1.

(b) *head* é reajustado para que o segundo nó se torne o primeiro

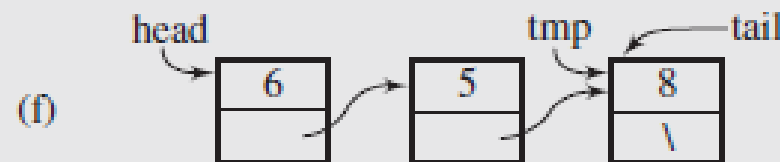
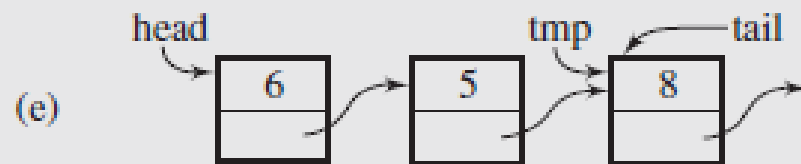
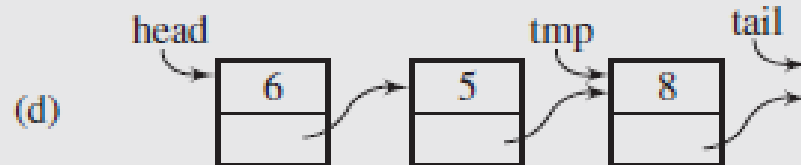
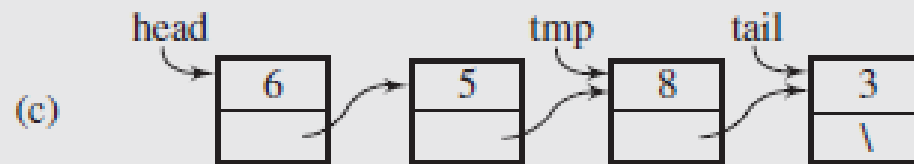
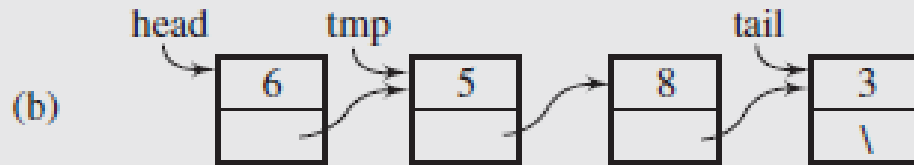
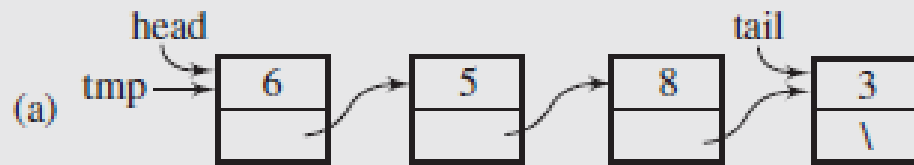
(c) O valor do item excluído é retornado.

e retornando o valor

```
int IntSLList::deleteFromHead() {  
    int e1 = head->info;  
    IntSLLNode *tmp = head;  
    if (head == tail)    // if only one node in the list;  
        head = tail = 0;  
    else head = head->next;  
    delete tmp;  
    return e1;  
}
```

***E se a lista
estiver vazia?***

Removendo um nó no final da lista:



(a) A informação do último nó é armazenada em uma variável local. E um novo ponteiro **tmp** é criado com o valor de **head**.

(b) e (c) Se há mais de um nó na lista, encontrar o predecessor de **tail**. Para isto, deve-se percorrer a lista desde o início com o ponteiro **tmp** enquanto **tmp->next** ser diferente de **tail**.

(d) O nó apontado por **tail** é excluído.

(e) O predecessor de **tail** se torna o novo **tail**.

(f) Ajusta-se o ponteiro **next** de **tail** para 0.

Removendo um nó no final da lista:

```
int IntSLList::deleteFromTail() {
    int el = tail->info;
    if (head == tail) { // if only one node in the list;

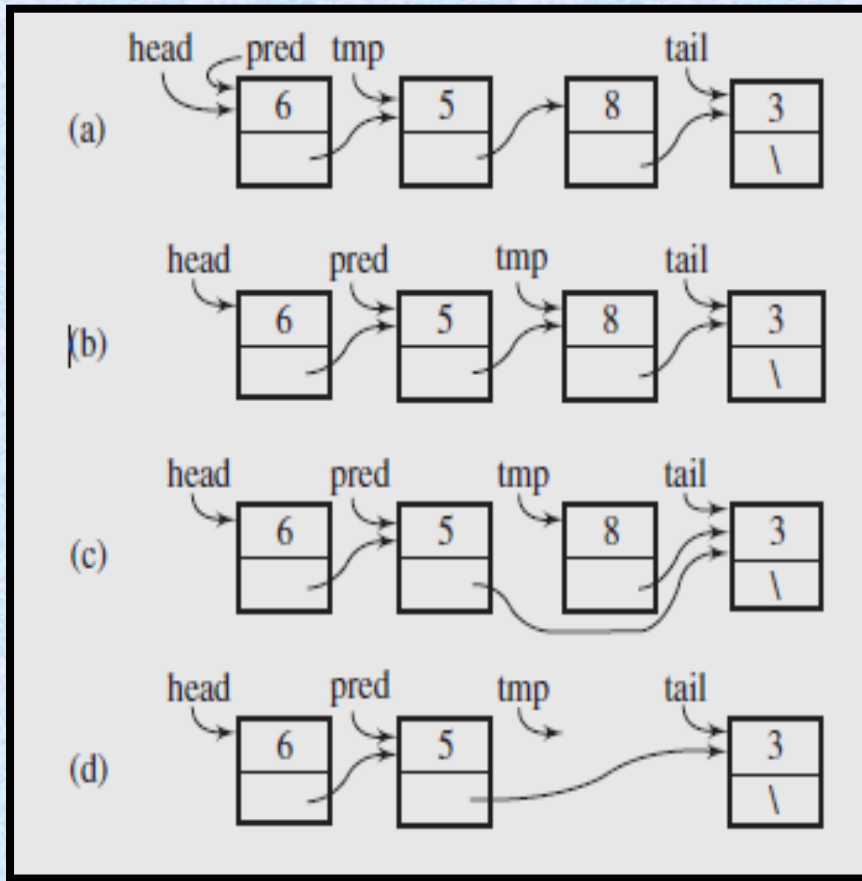
        delete head;
        head = tail = 0;
    }
    else {                // if more than one node in the list,
        IntSLLNode *tmp; // find the predecessor of tail;
        for (tmp = head; tmp->next != tail; tmp = tmp->next);
        delete tail;
        tail = tmp;      // the predecessor of tail becomes tail;
        tail->next = 0;
    }
    return el;
}
```

E se a lista estiver vazia?

e retornando o valor.

Obs: O que consome mais tempo é encontrar o nó próximo ao último, realizado pelo laço *for* . O laço realiza $n - 1$ iterações em uma lista de n nós, logo, tempo $O(n)$ para remover o último nó.

Removendo um nó qualquer da lista



(a) Ponteiro **pred** aponta para o primeiro nó e o ponteiro **tmp** aponta para o segundo nó.

(b) **pred** e **tmp** são avançados para os próximos nós.

(c) Como o nó 8 foi encontrado, fazer **pred -> next = tmp -> next**

(d) O nó 8 é excluído.

O nó pode estar no início, no fim ou em qualquer lugar da lista. Ele deve ser localizado primeiro e depois desligado da lista, **vinculando-se** o seu predecessor diretamente ao seu sucessor.

Removendo um nó qualquer da lista

Casos abordados:

- 1- Tentativa para remover o nó de uma lista vazia: a função é imediatamente abandonada;
- 2- Remoção do único nó de uma lista ligada de apenas um nó: *head* e *tail* ajustados para 0;
- 3- Remoção do primeiro nó da lista com pelo menos dois nós, o que exige atualizar *head*. Tempo de execução: $O(1)$.
- 4- Remoção do último nó da lista com pelo menos dois nós: atualiza *tail*. Tempo de execução: $O(n)$.
- 5- Tentativa de remover um nó com um número que não há na lista: não faz nada.

Removendo um nó qualquer da lista

```
void IntSLList::deleteNode(int el) {
    if (head != 0) // if nonempty list;
        if (head == tail && el == head->info) { // if only one
            delete head; // node in the list;
            head = tail = 0;
        }
        else if (el == head->info) { // if more than one node in the list
            IntSLLNode *tmp = head;
            head = head->next;
            delete tmp; // and old head is deleted;
        }
        else { // if more than one node in the list
            IntSLLNode *pred, *tmp;
            for (pred = head, tmp = head->next; // and a nonhead node
                tmp != 0 && !(tmp->info == el); // is deleted;
                pred = pred->next, tmp = tmp->next);
            if (tmp != 0) {
                pred->next = tmp->next;
                if (tmp == tail)
                    tail = pred;
                delete tmp;
            }
        }
    }
}
```

Busca binária

Operações de inserção e remoção: modificam as listas ligadas.

Operação de busca: varre uma lista existente para informar se um número está nela.

```
bool IntSLList::isInList(int e1) const {  
    IntSLLNode *tmp;  
    for (tmp = head; tmp != 0 && !(tmp->info == e1); tmp = tmp->next);  
    return tmp != 0;  
}
```

- Usa uma variável temporária ***tmp*** para percorrer a lista, começando do nó do topo, o nó ***head***.
- O número armazenado em cada nó é comparado com o número procurado. Se são iguais, o laço é abandonado. Caso contrário, ***tmp*** é atualizado e o próximo nó é verificado;
- Se ***tmp*** é não nulo, *e1* foi encontrado e retorna verdadeiro (1), caso contrário, retorna falso (0);
- *Melhor caso: $O(1)$. Pior caso: $O(n)$*

Exibir a informação da lista

```
void IntSLList::printSLLList()
{
    IntSLLNode *p=head;
    while (p!=NULL)
    {
        cout << "\n\n p->info=" << p->info;
        //cout << "\n p=" << p;
        //cout << "\n p->next=" << p->next;
        p = p->next;
    }
}
```

```
p->info=5
p=0x3512f0
p->next=0x351160
```

```
p->info=8
p=0x351160
p->next=0x351270
```

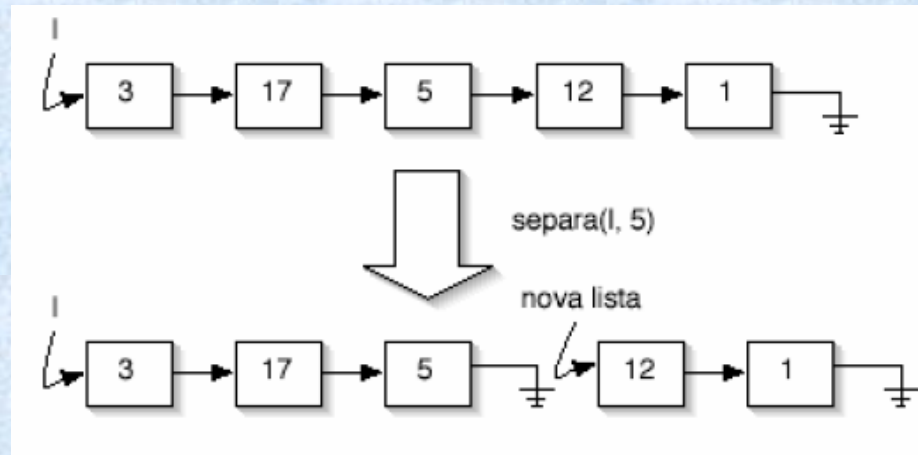
```
p->info=3
p=0x351270
p->next=0
```


Exercícios de SLL

Considerando listas de valores inteiros, escreva funções membro para:

- 1) Informar o maior e o menor elemento da lista;
- 2) Retornar a média aritmética dos valores da lista;
- 3) Anexar uma SLL ao fim de outra SLL;
- 4) Indicar quantos elementos existem na lista;
- 5) Inserir um novo nó no meio da lista (caso o número de elementos seja par e superior a dois);
- 6) Verificar se duas SLL tem o mesmo conteúdo;
- 7) Retornar apenas os valores pares da lista;
- 8) Inverter uma lista SLL.

9) Dividir uma lista em duas, recebendo como **parâmetros** uma **lista encadeada** e um **valor inteiro n**, de tal forma que a segunda lista comece no primeiro nó logo após a primeira ocorrência de **n** na lista original. A figura a seguir ilustra essa separação:



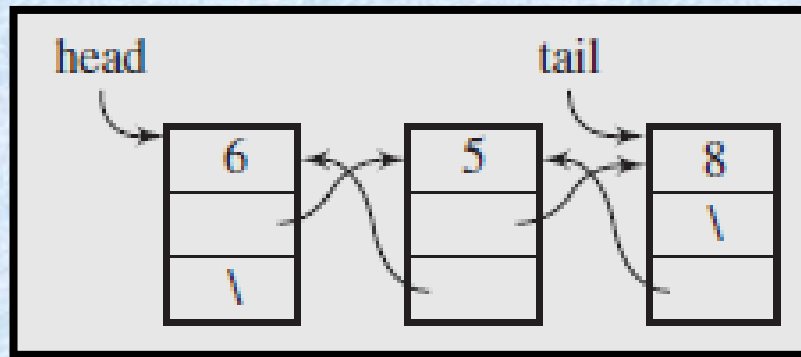
Essa função deve obedecer o protótipo:

Lista* separa (Lista* l, int n);

- A função deve retornar um ponteiro para a segunda subdivisão da lista original, enquanto l deve continuar apontando para o primeiro elemento da primeira subdivisão da lista.

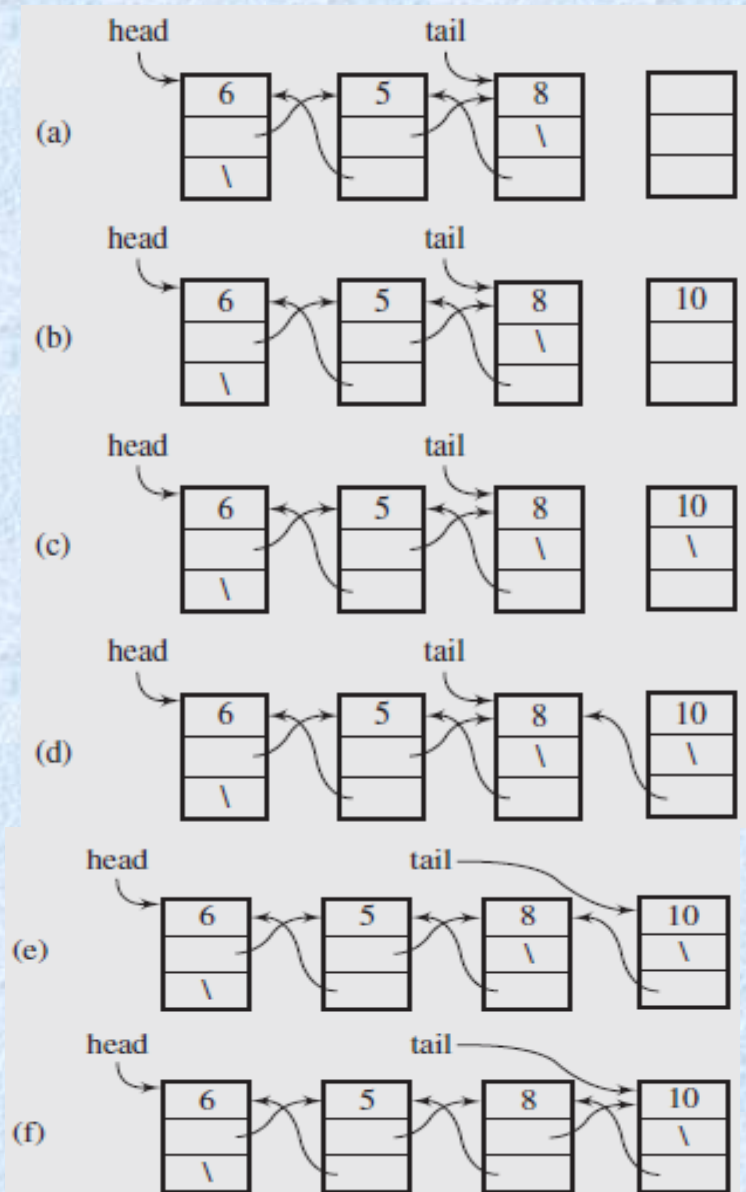
Lista Duplamente ligada (DLL: Doubly linked list)

- Cada nó agora tem dois ponteiros: um para o sucessor e outro para o predecessor.



- Como executar as operações de inserção, remoção e busca?
- Inserir e remover no final da lista: Tempo constante $O(1)$.

Inserindo um novo nó no final da lista DLL



(a) Um novo nó é criado e seus três membros de dados inicializados (*info*, *next*, *prev*).

(b) O membro *info* para o número e1 é inserido.

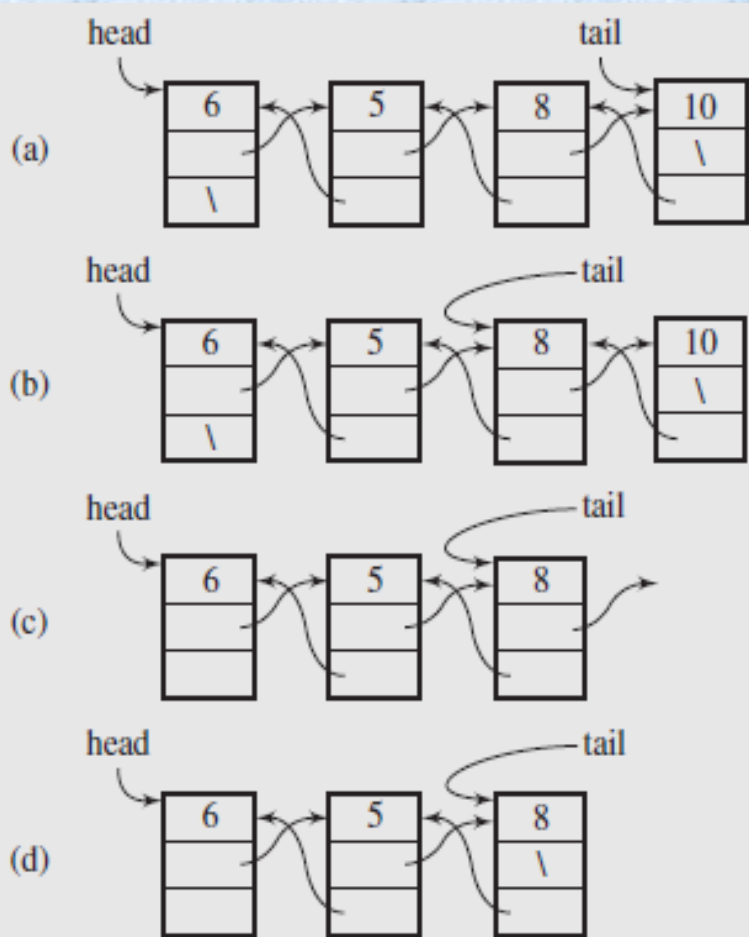
(c) O membro ***next*** é nulo.

(d) O membro ***prev*** vai para o valor de ***tail***, para que aponte para o último nó da lista.

(e) ***tail*** é ajustado para apontar para o novo nó. MAS o novo nó não é ainda acessível a partir de seu predecessor; para corrigir isto,

(f) O membro ***next*** do predecessor é ajustado para apontar para o novo nó.

Removendo um nó no final da lista DLL:



(a) A variável temporária *e1* é ajustada para o valor do nó.

(b) **tail** é ajustado para o seu predecessor.

(c) O último nó é removido.

(d) O nó próximo ao último se torna o último nó. O membro **next** do nó de **tail** é uma referência pendente; logo, **next** é ajustado para nulo

Obs: É necessário verificar: se a lista não está vazia e também se a lista possui apenas um nó (ajustar *head* e *tail* para nulo).

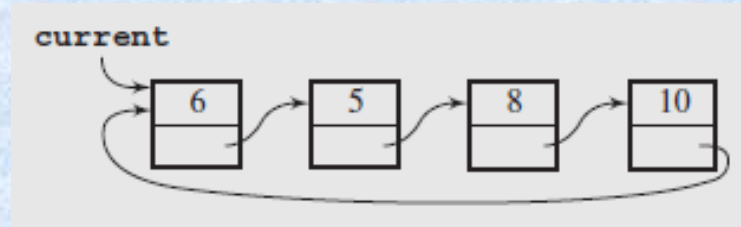
Exercício

- 1) Implemente uma DLL com as operações básicas de inserção, remoção e busca.
- 2) Insira os elementos 10, 20, 40, 30 e imprima na ordem inversa.
- 3) Remova o último elemento da lista.
- 4) Remova o *i-ésimo* nó em uma lista ligada. Tenha certeza que tal nó exista.
- 5) Insira um nó no meio da lista.

Lista circular

(Circular Singly Circular List)

- Os nós formam um anel: a lista é finita e cada nó tem um sucessor.



- Inserindo elementos na frente (a) e ao final da lista (b)

