

# Estruturas de Dados Encadeadas

- Vetores são usados como **estrutura de dados** em muitos problemas computacionais.

**Exemplo:** manter uma lista de valores ordenada.

- No caso dos valores da lista serem complexos, pode-se definir um **novo tipo de dados** e então utilizar um vetor deste novo tipo.

**Exemplo:**

```
typedef struct lista
{
    int cliente;
    float valor;
} lista;

lista v[MAX];
```

0003	0018	0505	2122	0018			
95.00	84.50	71.90	65.00	58.00			

- Mas para alguns problemas, o vetor não é conveniente.

# Estruturas de Dados Encadeadas


- Imagine que a lista de clientes deve ser mantida ordenada pelo valor do pedido:

0003	0018	0505	2122	0018							
95.00	84.50	71.90	65.00	58.00							

Considere um novo pedido a ser inserido:

- número do cliente: 0101
- valor do pedido: 99.00

O que deve ser feito para manter a lista ordenada?



0003	0018	0505	2122	0018							
95.00	84.50	71.90	65.00	58.00							

	0003	0018	0505	2122	0018						
	95.00	84.50	71.90	65.00	58.00						

0101	0003	0018	0505	2122	0018						
99.00	95.00	84.50	71.90	65.00	58.00						

## Problemas:

- Muitas movimentações de dados.
- O esforço computacional depende dos dados a serem inseridos.

## Estruturas de Dados Encadeadas

- No caso de **exclusão**, as movimentações de dados também são necessárias. Por exemplo: excluir o pedido do cliente 0003:

0101	0003	0018	0505	2122	0018	0505					
99.00	95.00	84.50	71.90	65.00	58.00	42.00					

0101	0018	0505	2122	0018	0505						
99.00	84.50	71.90	65.00	58.00	42.00						

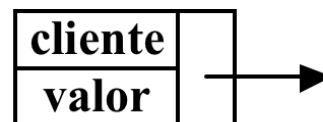
- Estruturas de dados encadeadas** são mais convenientes para estes tipos de problemas.
- Com uma estrutura de dados encadeada:
  - Evitam-se as movimentações de dados nas operações de inclusão e exclusão de dados.
  - O esforço computacional destas operações é, praticamente, constante.

# Listas encadeadas

- Uma lista encadeada é uma estrutura de dados que, além dos campos necessários para armazenar a informação do problema (como **cliente** e **valor**, por exemplo), contém também um **ponteiro** para a própria estrutura.

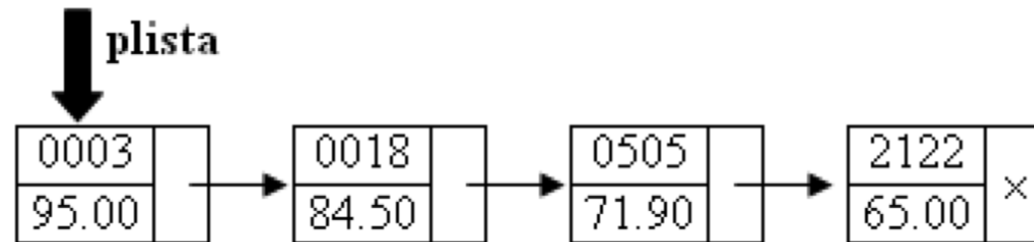
```
typedef struct lista
{
    int cliente;
    float valor;
    struct lista *prox;
} lista;
```

- O campo **prox** desta estrutura é do tipo **struct lista \***, ou seja, corresponde a um **ponteiro** para a própria estrutura **lista**. Assim, uma variável do tipo **lista** pode ser vista como:



# Listas encadeadas

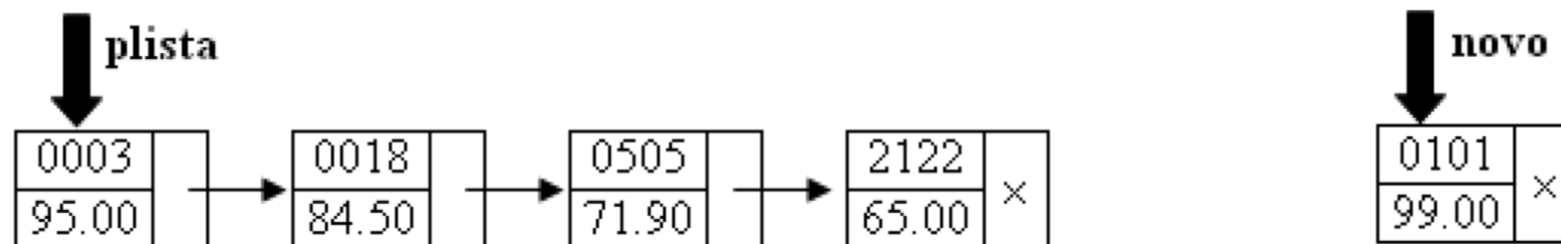
- Lista de pedidos de clientes:



- O ponteiro **prox** pode ser entendido como o **endereço do próximo elemento** na lista. O ponteiro NULL indica o final da lista (ou seja, indica que não existe um “próximo” elemento). Este tipo de estrutura de dados é conhecido como **lista encadeada simples**.
- Por que usar um ponteiro para indicar o próximo elemento da lista? A vantagem está na facilidade em realizar as operações de inserção e exclusão de elementos da lista.

# Listas encadeadas

- **Vetor**: existe uma **ordenação física** dos elementos (posições consecutivas de memória). As movimentações de dados são necessárias para manter o vetor ordenado "sem buracos".
- **Lista encadeada**: existe uma **ordenação lógica** dos elementos da lista (os elementos da lista ocupam posições quaisquer de memória, não necessariamente consecutivas). É preciso indicar (por meio de um **outro ponteiro**) qual é o "primeiro" elemento da lista.
- Inclusão na lista:

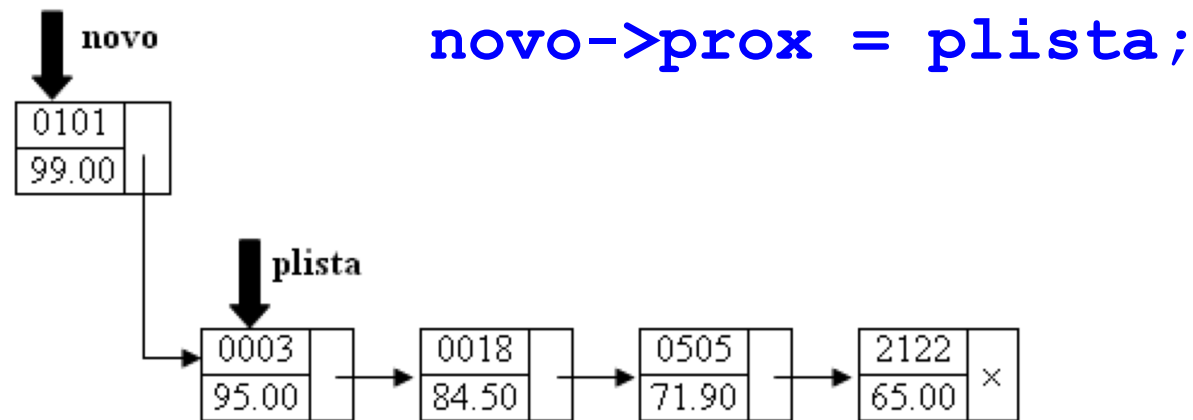


- O que fazer para incluir o novo elemento na lista?

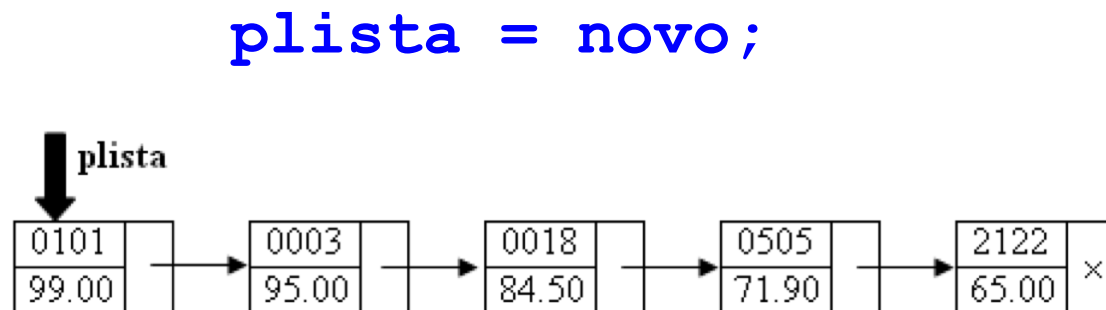
# Listas encadeadas

## Operações:

- Fazer o ponteiro **prox** do elemento apontado por **novo** apontar para o elemento apontado por **plista**:

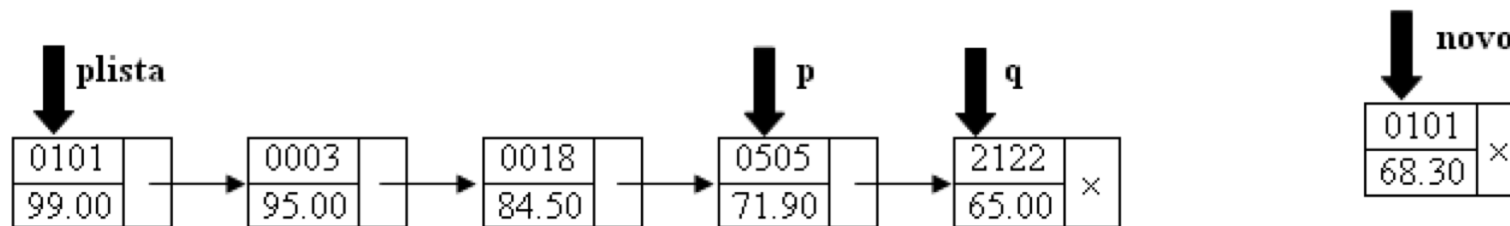


- Movimentar o ponteiro **plista** para o “início” da lista:



# Listas encadeadas

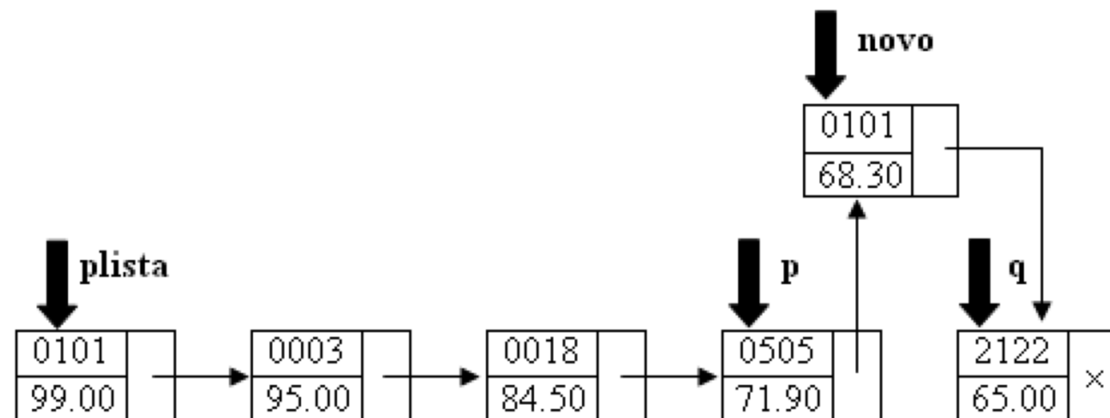
- E a inclusão no “meio” da lista?



- Algoritmo de inclusão:

**p->prox = novo;**

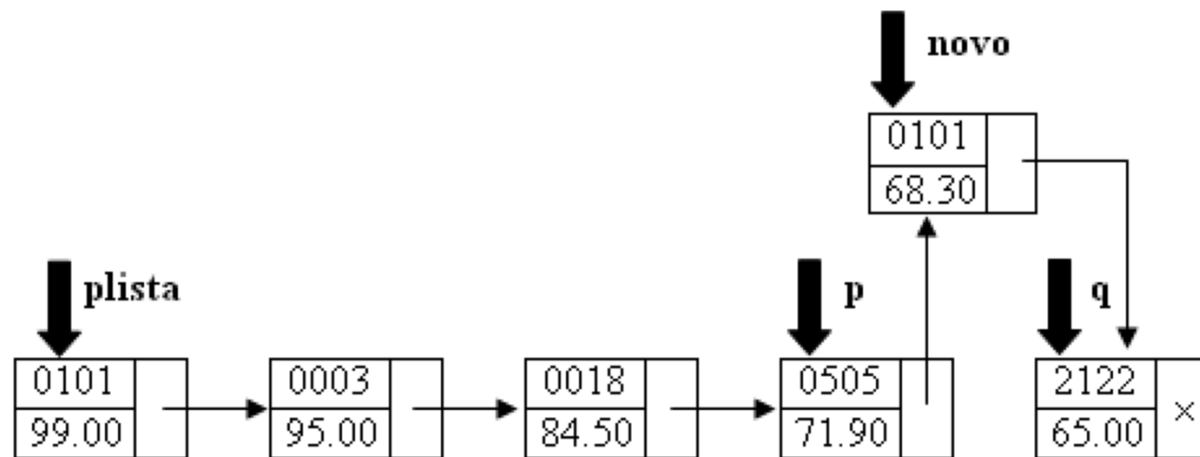
**novo->prox = q;**



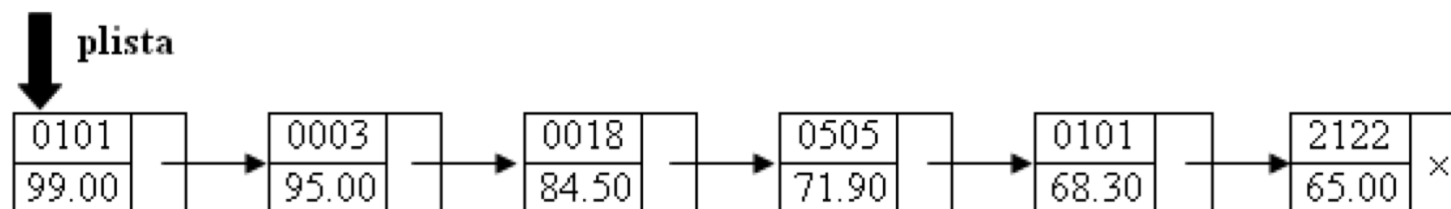


# Listas encadeadas

- Do ponto de vista lógico, a lista:



é equivalente a:



# Listas encadeadas

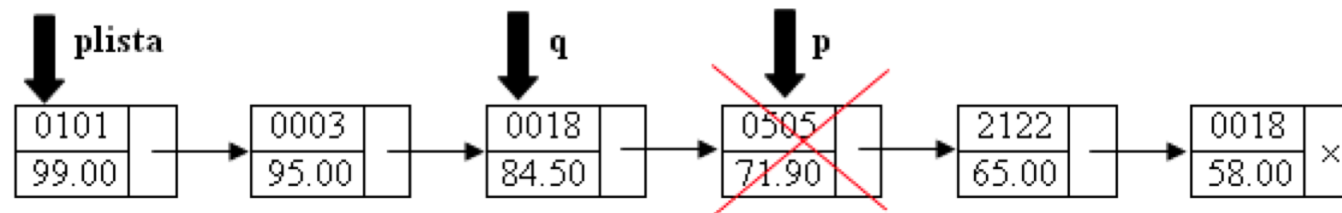
- O algoritmo de inclusão é o mesmo, se o novo elemento tiver que ser incluído no “final” da lista ( $q = \text{NULL}$ ).
- Um cuidado especial deve ser tomado quando o novo elemento tiver que ser incluído no “início” da lista ( $p = \text{NULL}$  e  $q = \text{plista}$ ).
- Algoritmo de **inclusão em lista encadeada**:

```
void InclusaoLista(lista novo, lista p, lista q)
{
    if (p == NULL)
        plista = novo;
    else
        p->prox = novo;
    novo->prox = q;
}
```

- E a exclusão?

# Listas encadeadas

- Para a exclusão devemos ter um ponteiro para o elemento a ser excluído (ponteiro **p**) e um outro ponteiro para o elemento anterior (ponteiro **q**):

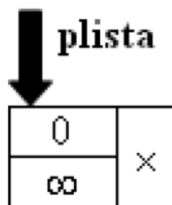


- A exclusão do “último” elemento pode ser feita sem problemas, mas deve-se ter um cuidado especial ao excluir o primeiro elemento ( $q = \text{NULL}$  e  $p = \text{plista}$ ).
- Algoritmo de exclusão em lista encadeada:

```
void ExclusaoLista(lista p, lista q)
{
    if (q == NULL)
        plista = p->prox;
    else
        q->prox = p->prox;
    free(p);
}
```

# Listas encadeadas

- Com a estrutura de lista encadeada, para manter a lista ordenada, o esforço computacional:
  - É pequeno (movimentação de um ou dois ponteiros).
  - É praticamente constante (independente do valor a ser inserido ou excluído).
- Tanto na inclusão como na exclusão, um cuidado especial deve ser tomado com o "início" da lista.
- Para simplificar os algoritmos: incluir na lista um elemento especial como cabeça da lista ("*list head*"). Com isso, uma **lista vazia** é uma lista que contém apenas o "*list head*" (e não com `plista = NULL`):

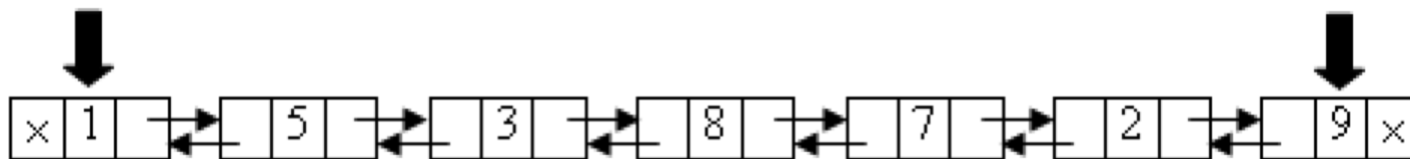


```
lista CriarLista()
{
    plista = (lista *)calloc(1, sizeof(lista));
    plista->cli = 0;
    plista->val = INFINITO;
    plista->prox = NULL;
    return plista;
}
```

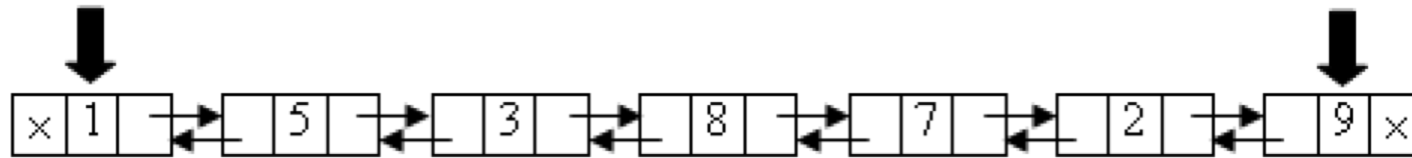
Com o "list head", os "casos especiais" não serão mais necessários.

## Listas duplamente encadeadas

- Em algumas situações, uma lista encadeada precisa ser percorrida tanto do primeiro ao último elemento, como no sentido inverso (do último para o primeiro elemento).
- **Exemplo:** um programa que armazena os dígitos de números inteiros arbitrariamente grandes como células de uma lista encadeada. Para a leitura dos dígitos é interessante percorrer a lista do início para o fim, mas para a operação de soma é mais interessante percorrer a lista do fim para o início.
- **Lista duplamente encadeada:** contém não apenas ponteiros para o “próximo” elemento, como também ponteiros para o elemento “anterior”.



# Listas duplamente encadeadas



- Esta estrutura de dados pode ser definida como:

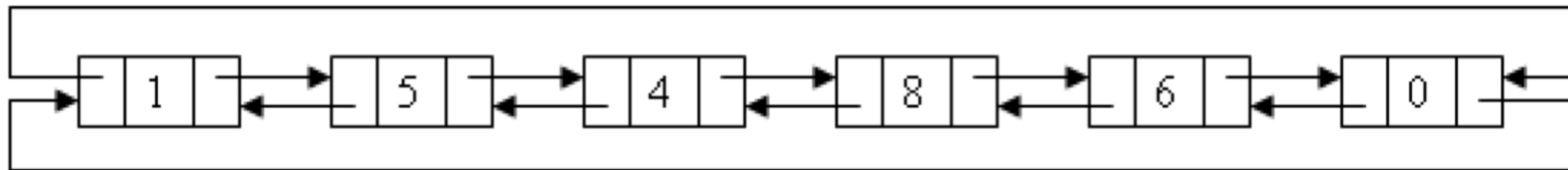
```
typedef struct lista
{
    int digito;
    struct lista *prox;
    struct lista *prev;
} lista;
```

onde **prox** aponta para o próximo elemento e **prev** aponta para o elemento anterior da lista.

- Para listas duplamente encadeadas ordenadas, as operações de inclusão e exclusão também são facilmente implementadas. Neste caso, pode-se usar dois elementos especiais, apontados por: **LH** (“*list head*”) e **LT** (“*list tail*”).

# Listas circulares

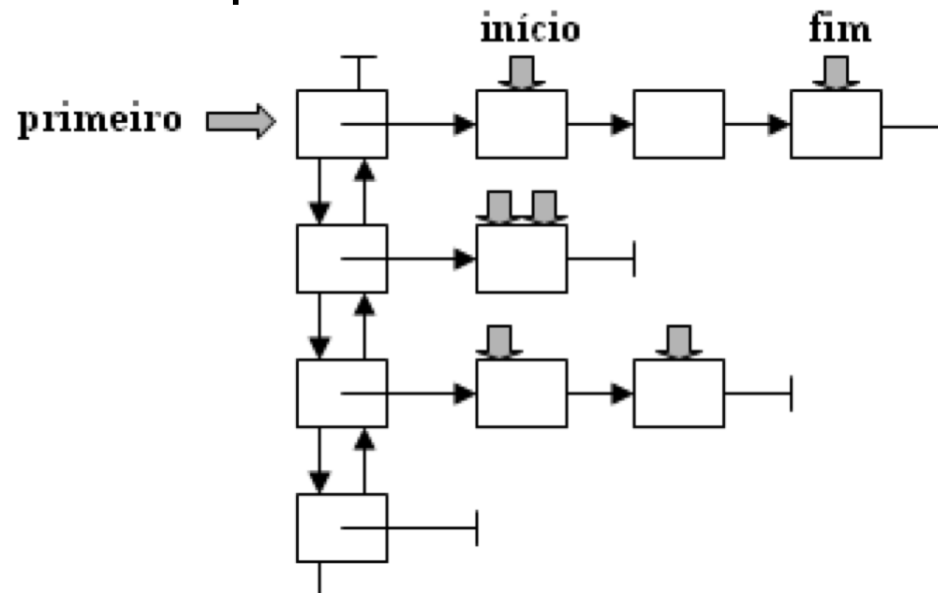
- Caso especial de lista duplamente encadeada.
- Numa lista circular, o ponteiro **prox** do “último” elemento aponta para o “primeiro” elemento da lista e o ponteiro **prev** do “primeiro” elemento aponta para o “último” elemento da lista.



- No caso de listas circulares ordenadas, basta ter um ponteiro para o "início" da lista (que pode ser um "*list head*").

# Estruturas de dados encadeadas não-lineares

- As estruturas de dados encadeadas que vimos até agora são **listas lineares**. Há variações quanto ao número de ponteiros, quanto à existência de “elementos especiais” mas, do **ponto de vista lógico**, os elementos estão dispostos linearmente na estrutura.
- Evidentemente, estas estruturas lineares podem ser combinadas das mais diversas formas, conforme a necessidade. Exemplo:



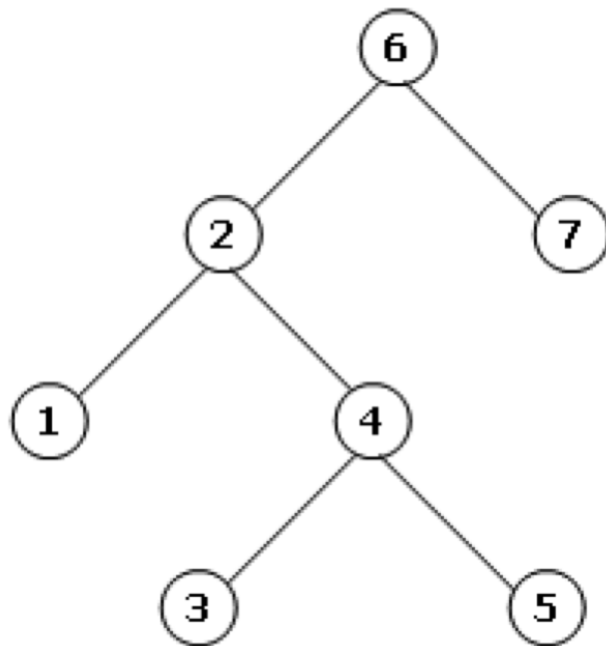


# Árvores binárias

- Uma das estruturas de dados encadeadas **não-lineares** mais importantes é a **árvore binária**.
- Uma árvore binária **A** pode ser definida recursivamente como:
  - **A** é um conjunto finito de elementos denominados **nós** ou **vértices**, tal que:
    - $A = \emptyset$  (“**árvore vazia**”), ou
    - Existe um nó especial  $r \in A$ , denominado “**raiz de A**” e os nós restantes podem ser divididos em dois subconjuntos disjuntos, **AE** e **AD**, que são denominados, respectivamente, “**subárvore esquerda de r**” e “**subárvore direita de r**”, os quais, por sua vez, também são árvores binárias.

# Árvores binárias

- Exemplo:



Os nós 7, 1, 3 e 5 são denominados **folhas da árvore**. Dizemos que os nós 2 e 7 são **filhos** de 6 (o nó 6 é o **pai** dos nós 2 e 7). E assim por diante...

$A = \{6, 2, 7, 1, 4, 3, 5\}$  onde o nó 6 é a **raiz da árvore A**, a subárvore esquerda de 6 é o conjunto  $AE = \{2, 1, 4, 3, 5\}$  e a subárvore direita de 6 é o conjunto  $AD = \{7\}$ . Os conjuntos AE e AD também são árvores binárias, sendo 2 a raiz de AE e 7 a raiz de AD. Para a árvore AE, a subárvore esquerda de 2 é o conjunto  $\{1\}$  e a subárvore direita de 2 é o conjunto  $\{4, 3, 5\}$ .

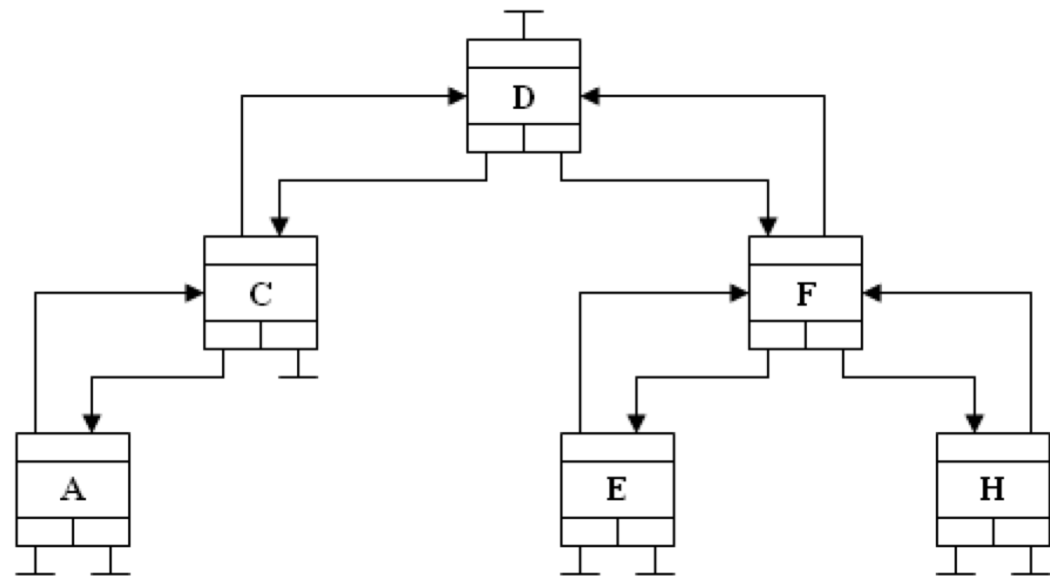
Para a árvore AD, a subárvore esquerda de 7 é o conjunto  $\emptyset$  e a subárvore direita de 7 é o conjunto  $\emptyset$ . E assim por diante...

# Árvores binárias

- A implementação de árvores binárias pode ser feita com uma estrutura de dados contendo três ponteiros:
  - um ponteiro para a “raiz”;
  - um ponteiro para a “subárvore esquerda”;
  - um ponteiro para a “subárvore direita”.
- Exemplo:

```
typedef struct arvore
{
    char info;
    struct arvore *pai;
    struct arvore *esquerda;
    struct arvore *direita;
} arvore;
```

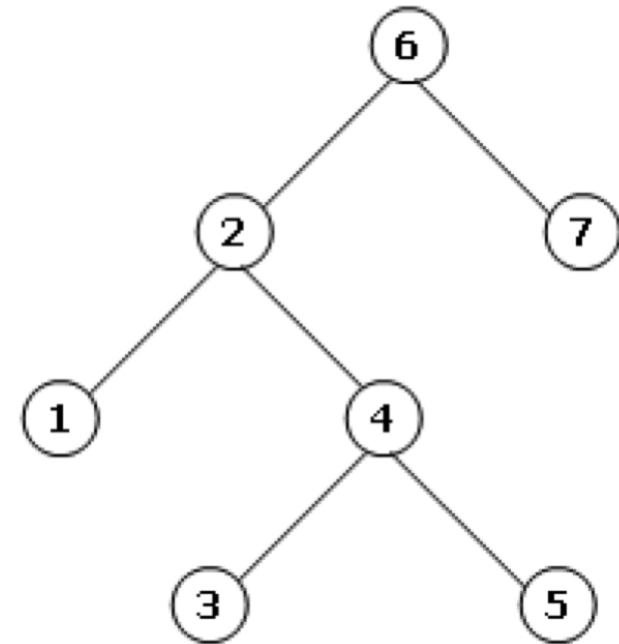
Note que a árvore binária pode ser **incompleta**, ou seja, nem todos os **nós internos** precisam ter os dois filhos.



# Árvores binárias

- O **nível** de um nó de uma árvore binária é definido como: a raiz da árvore tem nível 0 e o nível de qualquer outro nó da árvore é igual ao nível de seu pai + 1. Para a árvore ao lado tem-se:

nível(6) = 0, nível(2) = 1,  
nível(5) = 3.



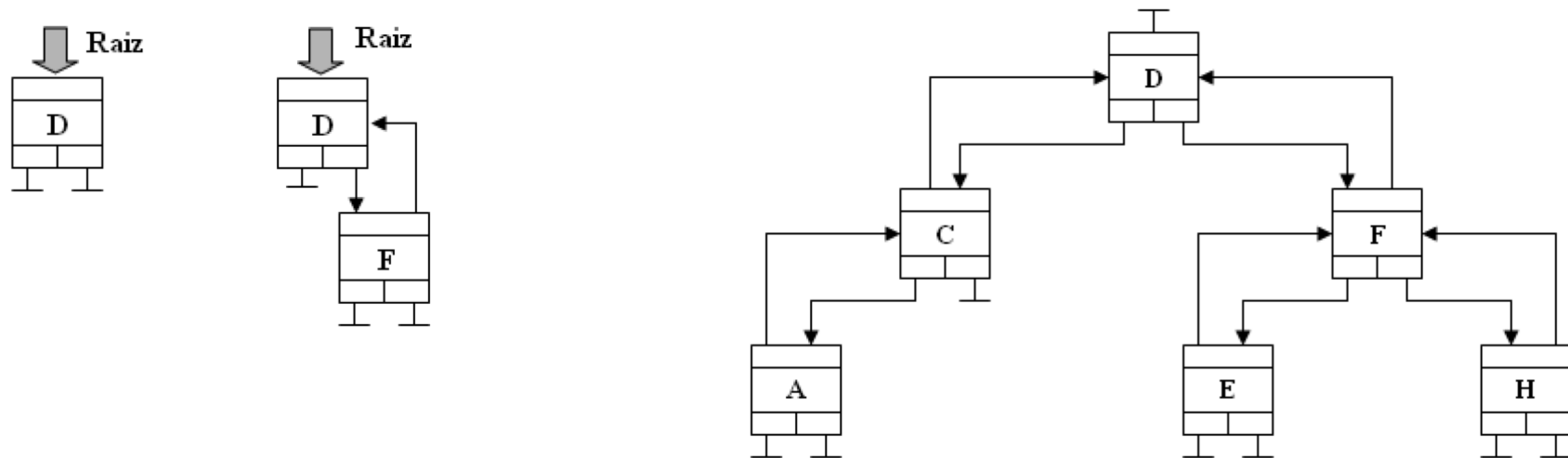
- A **profundidade** (ou **altura**) de uma árvore binária é definida como o nível máximo de qualquer folha da árvore (ou seja, o tamanho do percurso mais distante da raiz até uma folha da árvore). Por exemplo: a árvore acima tem profundidade igual a 3.

# Árvores binárias de procura

- As árvores binárias são estruturas de dados adequadas em muitas situações. Por exemplo: um catálogo telefônico sujeito a muitas **operações de consulta**.
- A **inclusão de novos nós** na árvore binária de procura segue as seguintes regras:
  - O primeiro nó que entra é colocado na raiz da árvore;
  - Os demais nós serão inseridos como folhas da árvore;
  - Para determinar qual é pai de um novo nó folha, compara-se o novo nó com o nó ativo. Inicialmente o nó ativo é a raiz da árvore. A cada comparação, se o novo nó é menor do que o nó ativo, o nó ativo passa a ser o seu filho da esquerda; caso contrário, o nó ativo passa a ser o seu filho da direita, até que o nó ativo seja NULL. O último nó ativo diferente de NULL é o pai do novo nó.

# Árvores binárias de procura

**Exemplo:** sejam os dados de entrada {D, F, H, C, E, A}.



- Para mostrar a informação da árvore em **ordem alfabética**, basta notar que em uma árvore binária de procura, uma subárvore esquerda contém somente valores menores do que o valor contido na raiz desta subárvore e que uma subárvore direita contém somente valores maiores (ou iguais, no caso de haver repetições) do que o valor contido na raiz desta subárvore.

# Árvores binárias de procura

- Portanto, para listar em ordem alfabética a informação contida em uma árvore binária de procura cuja raiz é apontada por **p** basta percorrer a árvore (recursivamente) como:
  - a) subárvore esquerda de **p**;
  - b) nó apontado por **p**;
  - c) subárvore direita de **p**.
- Algoritmo de percorrimento de árvore binária de procura:

```
void MostrarArvore(arvore *p)
{
    if (p == NULL)
        return;
    MostrarArvore(p->esquerda);
    printf("%s\n", p->nome);
    MostrarArvore(p->direita);
}
```

# Árvores binárias de procura

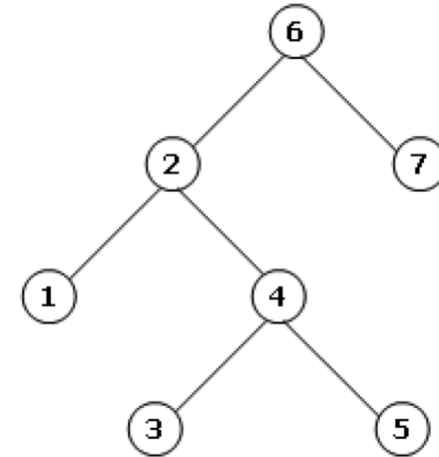
- O percorrimento de uma árvore binária consiste em **enumerar** cada um de seus nós. Dizemos que o nó é **visitado** à medida em que ele é enumerado.
- Três formas de visitar os nós de uma árvore binária:
  - Em **pré-ordem** (também conhecido como **percorrimento em profundidade**): visitar a raiz, visitar a subárvore esquerda em pré-ordem e visitar a subárvore direita em pré-ordem.
  - Em **ordem**: visitar a subárvore esquerda em ordem, visitar a raiz e visitar a subárvore direita em ordem.
  - Em **pós-ordem**: visitar a subárvore esquerda em pós-ordem, visitar a subárvore direita em pós-ordem e visitar a raiz.



# Árvores binárias de procura

## Exemplo:

- Pré-ordem: 6, 2, 1, 4, 3, 5, 7
- Ordem: 1, 2, 3, 4, 5, 6, 7
- Pós-ordem: 1, 3, 5, 4, 2, 7, 6



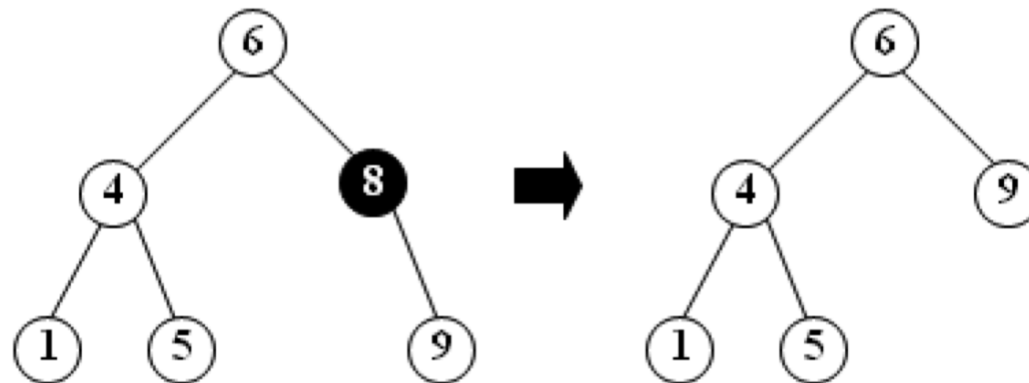
- Uma operação mais complicada é a **remoção de um nó** de uma árvore binária de procura.
- A complexidade da remoção depende do número de filhos do nó a ser removido. O caso mais fácil ocorre quando o nó a ser removido é uma folha.

**Exemplo:** removendo o nó 3 da árvore acima, tem-se ainda uma árvore binária de procura.

# Árvores binárias de procura

- Se o nó a ser removido tem somente um filho, o processo de remoção também não é complicado: o ponteiro (esquerdo ou direito) do pai deste nó deve ser ajustado para apontar para o filho do nó removido.

Exemplo:

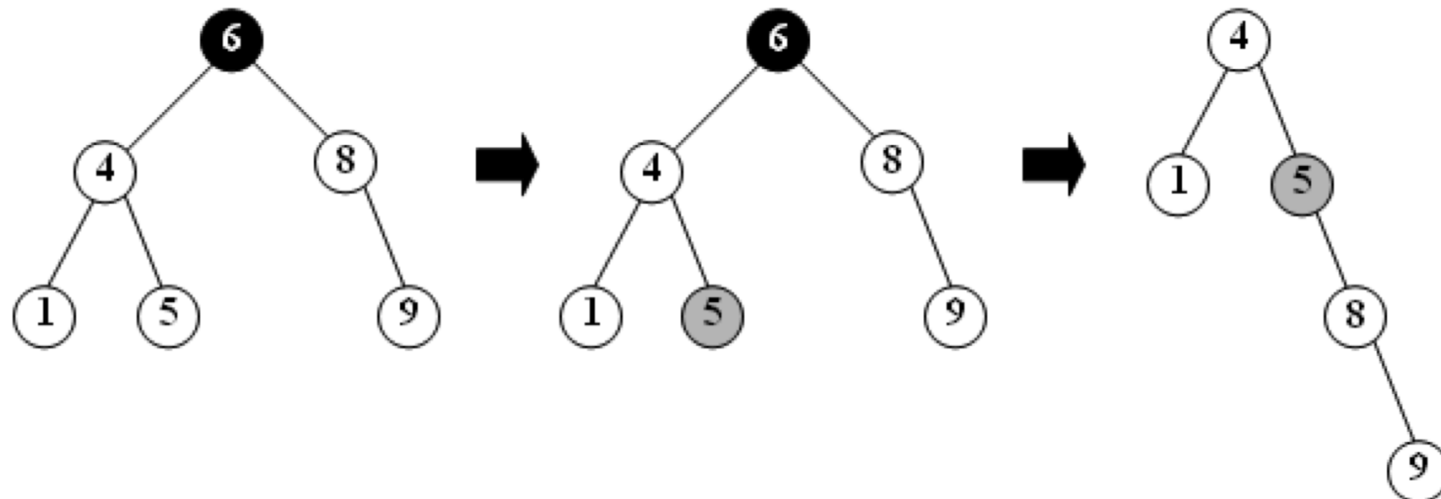


- O caso mais complicado ocorre quando o nó a ser removido tem dois filhos (ou seja, é pai de uma subárvore esquerda e de uma subárvore direita). Um algoritmo possível é conhecido como **remoção por fusão**.

# Árvores binárias de procura

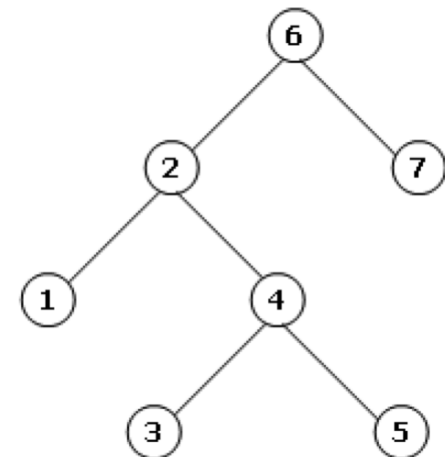
- A idéia do **algoritmo de remoção por fusão** é encontrar o nó de maior valor na subárvore esquerda do nó a ser removido (ou seja, o nó mais à direita da subárvore esquerda) e torná-lo pai da subárvore direita. Note que o algoritmo pode também ser aplicado de forma simétrica: encontrar o nó mais à esquerda da subárvore direita e torná-lo pai da subárvore esquerda do nó a ser removido.

Exemplo:



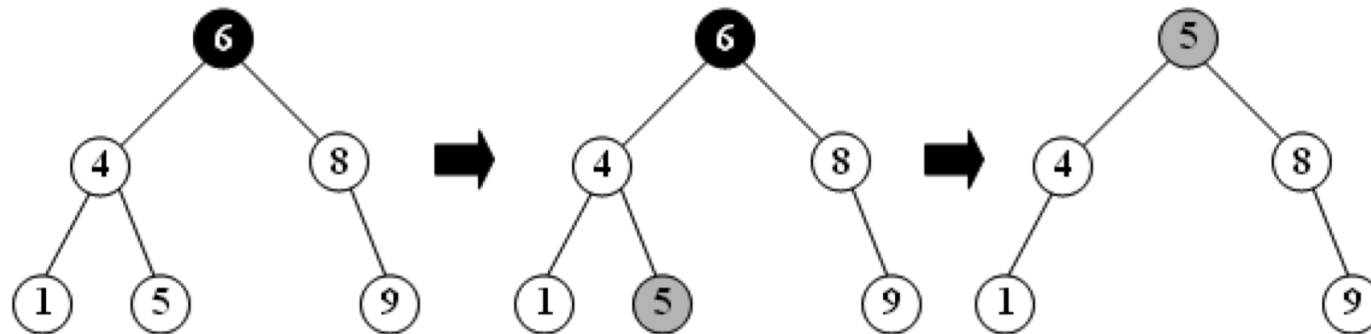
# Árvores binárias de procura

- A aplicação do algoritmo de remoção por fusão pode resultar, em alguns casos, no **aumento da profundidade** da árvore e em uma nova árvore altamente **desbalanceada**.
- Um outro algoritmo, que não implica em aumento da profundidade da árvore, é conhecido como **remoção por cópia**.
- O algoritmo consiste em encontrar o predecessor (ou sucessor) imediato do nó a ser removido e substituir este nó por seu predecessor (ou sucessor).
- O predecessor imediato do nó a ser removido é o nó mais à direita de sua subárvore esquerda (o sucessor é o nó mais à esquerda de sua subárvore direita).

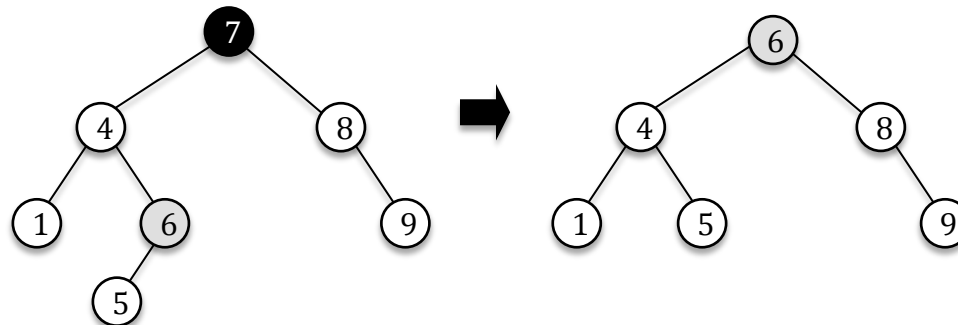


# Árvores binárias de procura

- Dois casos podem ocorrer:
  - O predecessor (ou sucessor) imediato é uma folha.



- O predecessor (sucessor) tem apenas um filho. Neste caso, o ponteiro para o filho direito (esquerdo) do pai do nó predecessor (sucessor) passa a apontar para o filho do predecessor (sucessor).



# Árvores binárias de procura

- O algoritmo de remoção por cópia não aumenta a altura da árvore, mas aplicado diversas vezes, pode resultar numa **árvore desbalanceada**, pois:
  - Usando o predecessor: a altura da subárvore esquerda da raiz pode diminuir, enquanto a altura da subárvore direita se mantém.
  - Usando o sucessor: o desbalanceamento é ao contrário, com a subárvore direita menor que a subárvore esquerda.
- Uma melhoria simples no algoritmo pode ajudar a manter a árvore balanceada: usar, alternativamente, o predecessor e o sucessor em aplicações repetidas do algoritmo.

## Aplicações de árvores binárias

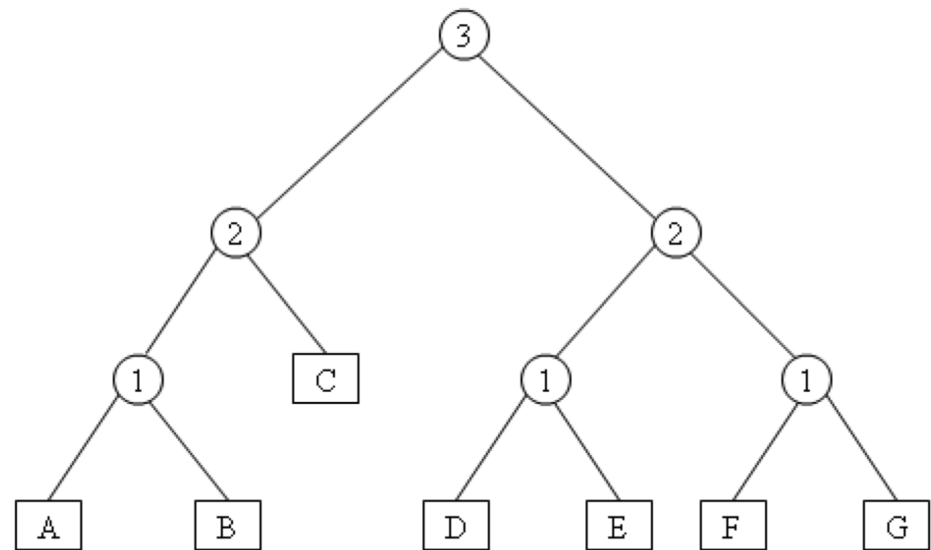
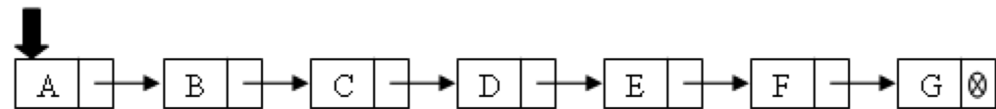
- Já vimos que as operações de inclusão e exclusão podem ser feitas eficientemente em listas encadeadas. Entretanto, recuperar o  $k$ -ésimo elemento de uma lista exige percorrer os primeiros  $k-1$  elementos da lista.
- Podemos **representar uma lista encadeada** como uma árvore binária, de modo que a recuperação do  $k$ -ésimo elemento possa ser feita de forma relativamente eficiente.
- Nesta representação, cada célula da lista corresponde a uma folha da árvore. Os demais nós da árvore contêm o número de folhas de sua subárvore esquerda.
- Várias árvores podem representar a mesma lista. Para que a recuperação seja eficiente, o ideal é que a árvore binária seja a mais completa (balanceada) possível.

# Listas encadeadas como árvores binárias

- Considere, por exemplo, uma lista com 1000 elementos. Uma árvore binária de profundidade 10 é suficiente para representar essa lista, pois  $\log_2 1000 < 10$  ( $2^{10} = 1024$ ).
- Algoritmo de recuperação:

```
int Recupera(arvore *p, int k)
{
    int r;

    r = k;
    while (!folha(p))
    {
        if (r <= p->folhas)
            p = p->esquerda;
        else
        {
            r = r - p->folhas;
            p = p->direita;
        }
    }
    return p->info;
}
```





## Aplicações de árvores binárias

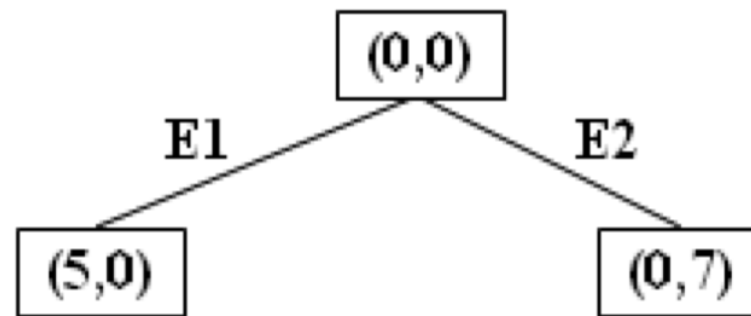
- Algumas técnicas de solução de problemas combinatoriais complexos baseiam-se na enumeração dos nós de uma árvore. Uma destas técnicas é conhecida como **procura em espaço de estados**.
- Nesta técnica, um estado do problema é transformado em diversos outros estados possíveis de serem alcançados a partir do estado original. Com isso, cria-se uma **árvore de estados**, em que a raiz é o estado inicial do problema e as folhas representam os estados terminais do problema, alguns dos quais correspondendo à sua solução.
- **Exemplo: O problema dos dois baldes.** Considere a existência de dois baldes, um de 5 litros e outro de 7 litros, inicialmente vazios, e uma fonte inesgotável de água. O objetivo é encontrar uma seqüência de ações que deixa exatamente 4 litros de água no balde maior.

# Árvores de estados

- Há três ações possíveis para alterar o conteúdo dos baldes:
  - Encher um balde (E1 ou E2)
  - Esvaziar um balde (V1 ou V2)
  - Transferir o conteúdo de um balde para outro até que um esteja vazio ou o outro esteja cheio (T12 ou T21)
- Considere que cada nó da árvore armazena um par de valores: (conteúdo do balde 1, conteúdo do balde 2).  
Vamos imaginar que as ações serão tomadas sempre na seguinte ordem: E1 (encher o balde menor), E2 (encher o balde maior), V1 (esvaziar o balde menor), V2 (esvaziar o balde maior), T12 (transferir o conteúdo do balde menor para o balde maior) e T21 (transferir o conteúdo do balde maior para o balde menor).

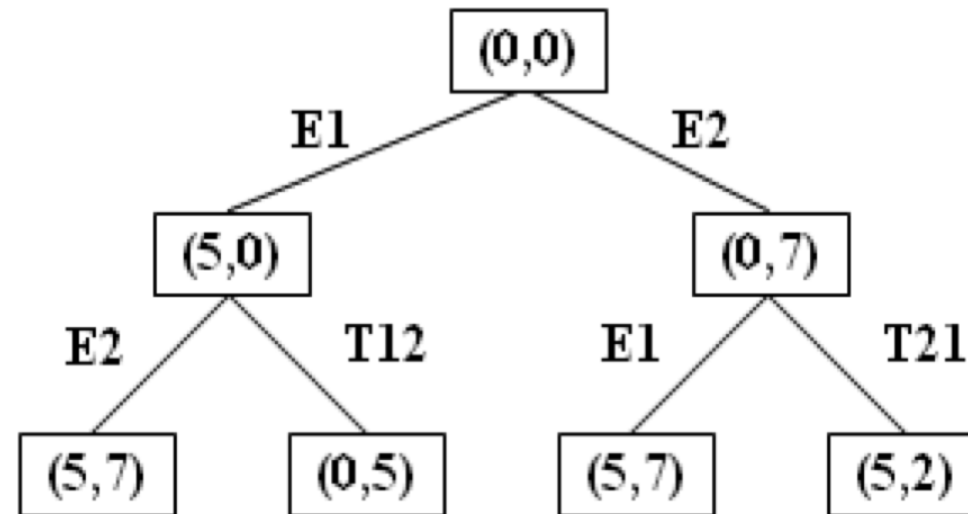
## Árvores de estados

- Inicialmente, tem-se o par **(0,0)** indicando que ambos os baldes estão vazios. Neste caso as ações possíveis são: E1 ou E2, que podem ser representadas pela árvore:



- Portanto, a ação E1 transforma o estado inicial **(0,0)** no estado **(5,0)** e a ação E2 transforma o estado **(0,0)** no estado **(0,7)**. Para a folha **(5,0)** as seguintes ações são possíveis: E2, V1 ou T12. Para a folha **(7,0)** as seguintes ações são possíveis: E1, V2, T21.

# Árvores de estados



- Note que as ações que levam a uma folha já existente no caminho até a raiz devem ser desconsideradas.
- Construindo uma árvore dessa forma, o problema estará resolvido quando for obtida uma folha  $(*,4)$ , ou seja, um conteúdo qualquer no balde menor e 4 litros no balde maior. Note, no entanto, que **a árvore não é necessariamente binária**. Por exemplo, para a folha  $(5,2)$  as seguintes ações são possíveis: E2, V1, V2 ou T12.

# Árvores de estados

- Para construir uma árvore geral, em que cada nó pode ter um **número qualquer de filhos**, pode-se definir uma estrutura de dados em que cada nó armazena três ponteiros:
  - um ponteiro para o nó “pai”;
  - um ponteiro para o nó “filho”;
  - um ponteiro para o nó “próximo irmão”.

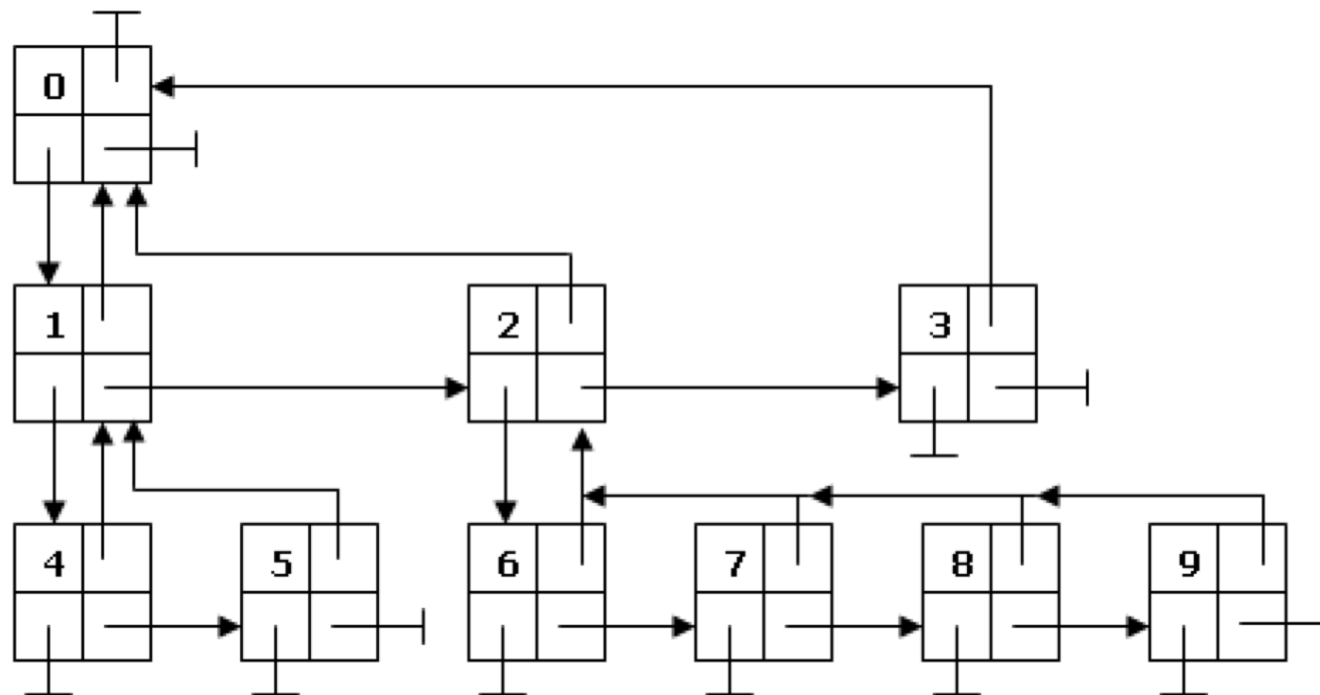
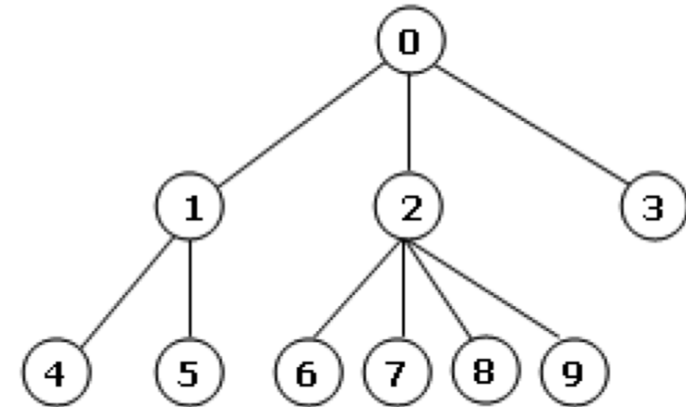
```
typedef struct tree
{
    int info;
    struct tree *pai;
    struct tree *filho;
    struct tree *irmao;
} tree;
```

info	pai
filho	irmao

# Árvores de estados

- Exemplo:

info	pai
filho	irmao



# Árvores de estados

- Um algoritmo para resolver o problema (conhecido como **busca em largura**) pode ser implementado como:

1. Expandir um nó considerando todas as ações possíveis;
2. Executar, recursivamente, o algoritmo para o próximo irmão deste nó;
3. Executar, recursivamente, o algoritmo para o filho deste nó.

- Neste algoritmo, pois cada nível da árvore é inteiramente construído (pelos passos 1 e 2) antes que qualquer nó do próximo nível seja acrescentado à árvore (pelo passo 3).
- Outro algoritmo é a **busca em profundidade**:

1. Expandir um nó considerando uma ação possível;
2. Executar, recursivamente, o algoritmo para o filho deste nó;
3. Se a expansão de um nó não for possível, voltar ao antecessor desse nó e tentar expandi-lo considerando uma ação possível ainda não tentada (o que irá corresponder a explorar um irmão desse nó).

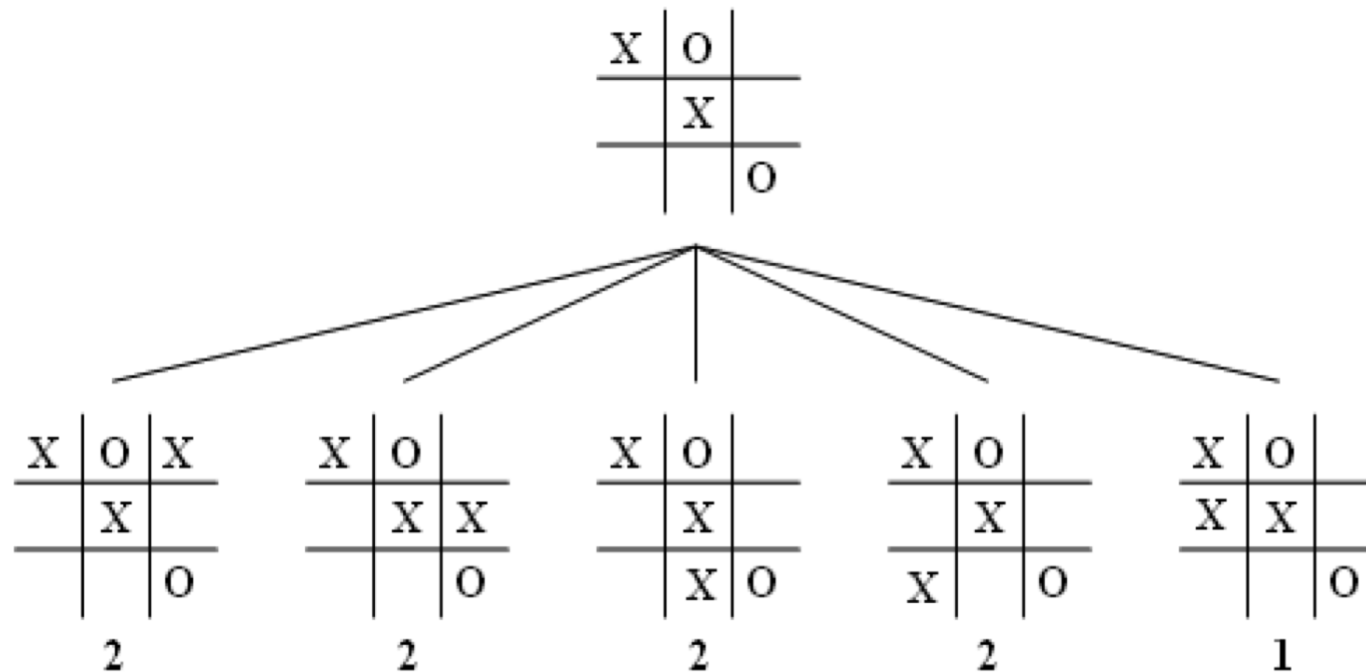
# Árvores de jogos

- Outra aplicação interessante de árvores refere-se à implementação de jogos por computador.
- No caso de jogos de tabuleiro envolvendo dois jogadores, cada nível da árvore corresponde aos movimentos de um jogador e nem sempre é factível expandir a árvore de todas as maneiras possíveis. Assim, é preciso dispor de uma função de avaliação capaz de determinar o “melhor” movimento possível em cada nível da árvore.
- **Exemplo:** o **jogo-da-velha**. Vamos imaginar os jogadores representados por **X** e **O**. Seja **chances**(  $j$  ) = número de linhas, colunas ou diagonais que permanecem abertas para o jogador  $j$ . Seja a seguinte função de avaliação:  
$$\text{avalia}(j) = \text{chances}(j) - \text{chances}(k)$$
onde  $k$  é o oponente do jogador  $j$ .



# Árvores de jogos

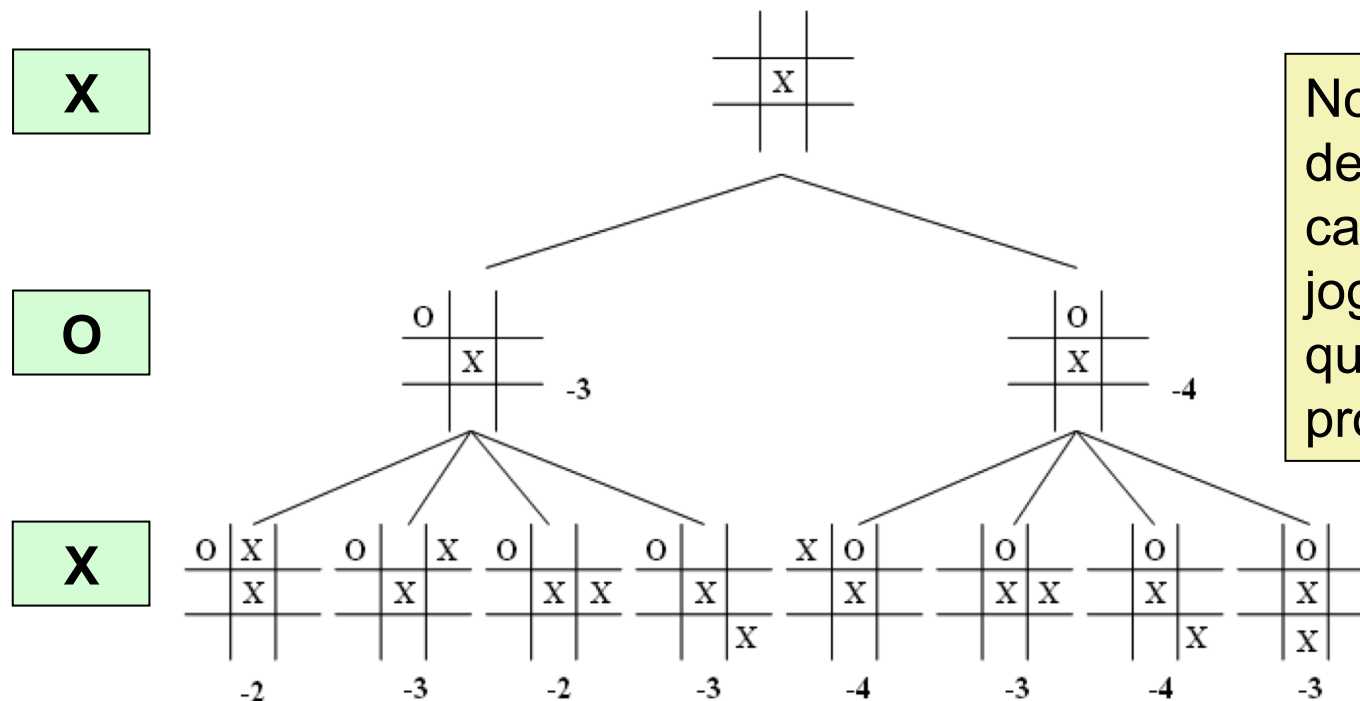
- Dada uma configuração do tabuleiro, o melhor movimento seguinte pode ser determinado considerando-se todos os movimentos possíveis e selecionando aquele movimento que resultar no maior valor da função de avaliação. A dificuldade é encontrar uma **boa** função de avaliação.



Note que esta função de avaliação não é boa. Por que?

# Árvores de jogos

- Uma avaliação melhor pode ser feita considerando-se um dado **nível de previsão de jogadas**. A árvore anterior considera apenas 1 jogada (árvore de profundidade = 1). Com um nível de previsão de jogadas maior a árvore irá conter jogadas dos dois jogadores.
- **Exemplo:** Qual deve ser a melhor jogada para **O**?



Note que a função de avaliação é calculada para o jogador **O** (o jogador que deve decidir a próxima jogada).

# Árvores de jogos

- A avaliação das jogadas possíveis de **O** é feita a partir das folhas da árvore. Nas folhas estão as jogadas possíveis de seu oponente.
- Imaginando-se que o oponente irá fazer sempre sua melhor jogada, neste nível da árvore o jogador **O** deve selecionar o **menor** valor da função de avaliação e levar esse valor para o pai destas folhas. No exemplo, os valores obtidos para as possíveis jogadas de **O** são -3 e -4.
- Para fazer sua melhor jogada, **O** deve selecionar, no seu nível, o **maior** valor da função de avaliação.
- Este algoritmo de solução do problema é conhecido como **método minimax** porque à medida que a árvore é percorrida das folhas para a raiz, as funções mínimo e máximo são usadas alternativamente até o jogador decidir sua melhor jogada.