



Algoritmos e Estruturas de Dados

MEEC – 2014/15



Recursividade e Árvores

Recursividade

- ▶ O conceito de recursividade é fundamental em matemática e ciências da computação.
- ▶ A definição mais simples de um programa recursivo numa linguagem de programação é um programa que se chama a si próprio.
- ▶ Naturalmente que um programa recursivo não se pode chamar a si próprio para sempre, senão nunca pararia.
- ▶ Por isso, o segundo ingrediente essencial é que tem de existir uma condição de paragem, em que o programa cessa de se chamar a si próprio.
- ▶ Praticamente todos os programas podem ser concebidos e desenhados em formato recursivo.

Recursividade e Árvores

- ▶ O estudo da recursividade está intimamente ligado com o estudo de estruturas de dados definidas recursivamente conhecidas como árvores.
- ▶ Iremos usar árvores tanto para nos ajudar a compreender e analisar programas recursivos como também enquanto estruturas de dados abstractas explícitas.
 - ▶ Ou seja, iremos usar árvores para entender programas recursivos;
 - ▶ Iremos usar programas recursivos para construir árvores;
 - ▶ E iremos socorrer-nos da íntima relação que possuem entre si para analisar algoritmos.
- ▶ A recursividade em programação permite o desenvolvimento de estruturas de dados elegantes e eficientes, assim como algoritmos para uma grande variedade de aplicações.

Algoritmos Recursivos

- ▶ Um algoritmo recursivo é aquele que resolve um dado problema através da resolução de uma ou mais instâncias mais pequenas do mesmo problema.
- ▶ Para implementar algoritmos recursivos em C, usamos naturalmente funções recursivas.
 - ▶ Notar que funções recursivas em C correspondem a definições recursivas de funções matemáticas.
 - ▶ Assim, no nosso estudo da recursividade, iremos começar por examinar programas que avaliam directamente funções matemáticas
 - ▶ Como se verá, o mecanismo básico fornecerá um paradigma de programação muito mais geral do que a simples avaliação ou cálculo de funções matemáticas.
 - ▶ O primeiro exemplo mais óbvio e familiar de uma função matemática recursiva é a função factorial.

A função factorial

- ▶ A função factorial pode ser definida pela seguinte relação recursiva
$$N! = N(N - 1)!, \quad \text{para } N > 0 \text{ com } 0! = 1.$$
- ▶ Esta definição corresponde directamente a função recursiva em C que se apresenta abaixo

```
int factorial(int N)
{
    if (N == 0) return 1;
    return N*factorial(N-1);
}
```

- ▶ Esta função é perfeitamente equivalente a um ciclo **for** simples, como se pode ver

```
for (t = 1, i = 1; i <= N; i++) t *= i;
```

Programas recursivos e não recursivos

- ▶ É sempre possível transformar um programa recursivo num programa não recursivo que execute os mesmos cálculos.
- ▶ Também é sempre possível expressar sem ciclos qualquer cálculo que os envolva utilizando recursividade.
- ▶ A utilização de recursividade permite frequentemente expressar cálculos complexos numa forma compacta, sem sacrificar eficiência.
 - ▶ No exemplo da função factorial, a versão recursiva dispensou a necessidade de utilizar variáveis locais.
- ▶ No entanto, existe também um custo associado às implementações recursivas, que se prende com a manutenção de uma pilha de recursão (pushdown stack).

Características básicas e demonstrações (1)

- ▶ Apesar das vantagens das implementações recursivas, é relativamente fácil escrever funções recursivas que são extremamente ineficientes, pelo que é necessário tomar as devidas precauções para evitar implementações intratáveis.
- ▶ O programa recursivo que vimos para a função factorial possui as características básicas de um qualquer programa recursivo:
 - ▶ Chama-se a si próprio para valores menores do seu argumento;
 - ▶ E possui uma condição de paragem em que calcula o resultado directamente.
- ▶ É relativamente fácil utilizar a indução matemática para nos convencermos que o código funciona como pretendido:
 - ▶ Calcula $0!$ (valor base)
 - ▶ Assumindo que calcula correctamente $(N-1)!$ (hipótese de indução), calcula correctamente $N! = N * (N-1)!$

Características básicas e demonstrações (2)

- ▶ Qualquer programa/função recursivo que se desenvolva necessita satisfazer duas propriedades básicas:
 - ▶ Têm que resolver explicitamente um caso base;
 - ▶ Cada chamada recursiva deverá envolver valores menores para os seus argumentos.
- ▶ Estes dois pontos não deixam de ser vagos – no essencial estão a implicar que deveremos ser capazes de estabelecer uma demonstração indutiva para cada programa recursivo que escrevamos.
- ▶ É preciso não esquecer que a utilização de recursividade significa que se está a incluir sucessivas chamadas de funções dentro de outras chamadas, pelo que existe o perigo de se exceder o limite de profundidade de recursão.

Características básicas e demonstrações (3)

- ▶ O programa abaixo é um exemplo em que se viola uma das propriedades, concretamente a que estabelece que cada chamada recursiva tem que ser feita para instâncias menores do problema original
- ▶ Assim, não é possível a utilização de indução matemática para entender o seu funcionamento.

```
int puzzle(int N)
{
    if (N == 1) return 1;
    if (N % 2 == 0)
        return puzzle(N/2);
    else return puzzle(3*N+1);
}
```

- ▶ De facto não é possível saber se o programa termina para todos os valores de N .

Dividir para conquistar (1)

- ▶ Muitos dos programas recursivos usam duas chamadas recursivas, cada uma delas operando em cerca de metade dos dados iniciais.
- ▶ Este esquema de recursão é, por certo, a mais importante instância do conhecido paradigma para desenvolvimento de algoritmos que dá pelo nome de divide-and-conquer, que serve de base a muitos e importantes algoritmos.
- ▶ Como exemplo, considere-se a solução recursiva para determinar o valor máximo numa tabela de inteiros

```
int max(int a[], int l, int r)
{int u, v, m = (l+r)/2;
 if (l == r) return a[l];
 u = max(a, l, m);
 v = max(a, m+1, r);
 if (u > v) return u; else return v;
}
```

Dividir para conquistar (2)

- ▶ Propriedade I – Uma função recursiva que divide um problema de tamanho N em duas partes independentes (não vazias) que depois resolve recursivamente, chama-se a si própria menos que N vezes.
- ▶ Demonstração – Se uma das partes tiver tamanho k e a outra tiver tamanho $N-k$, então o número total de chamadas recursivas feitas é

$$T_N = T_k + T_{N-k} + \text{I}, \quad \text{para } N > 0 \text{ com } T_1 = 0.$$

- ▶ A solução $T_N = N-\text{I}$ obtém-se facilmente por indução (verifique que assim é).
- ▶ A função **max** é representativa de muitos algoritmos divide-and-conquer, mas os outros podem ser diferentes desta em dois aspectos essenciais.

Dividir para conquistar (3)

- ▶ A função **max** executa sempre uma quantidade constante de cálculo em cada uma das chamadas, pelo que o seu tempo de execução é linear.
 - ▶ Outros algoritmos divide-and-conquer poderão realizar mais cálculos em cada chamada, pelo que a determinação do tempo de execução poderá requerer uma análise mais complexa.
- ▶ Outro aspecto é que na função **max** as duas partes formam o todo inicial.
 - ▶ Outros algoritmos divide-and-conquer podem dividir em partes mais pequenas que somam menos que o todo ou partes sobrepostas que somam mais que o todo.
- ▶ O tempo de execução destes algoritmos depende da forma como se faz a divisão em partes e do que é feito em cada parte.

Exemplo (1)

TINY EXAMPLE

```
max(0, 10)
  max(0, 5)
    max(0, 2)
      max(0, 1)
        max(0, 0) -> T
        max(1, 1) -> I
      max(2, 2) -> N
    max(3, 5)
      max(3, 4)
        max(3, 3) -> Y
        max(4, 4) -> E
      max(5, 5) -> X
  ...
  ...
```

TINY EXAMPLE

```
...
  max(6, 10)
    max(6, 8)
      max(6, 7)
        max(6, 6) -> A
        max(7, 7) -> M
      max(8, 8) -> P
    max(9, 10)
      max(9, 9) -> L
    max(10, 10) -> E
```

Exemplo (2)

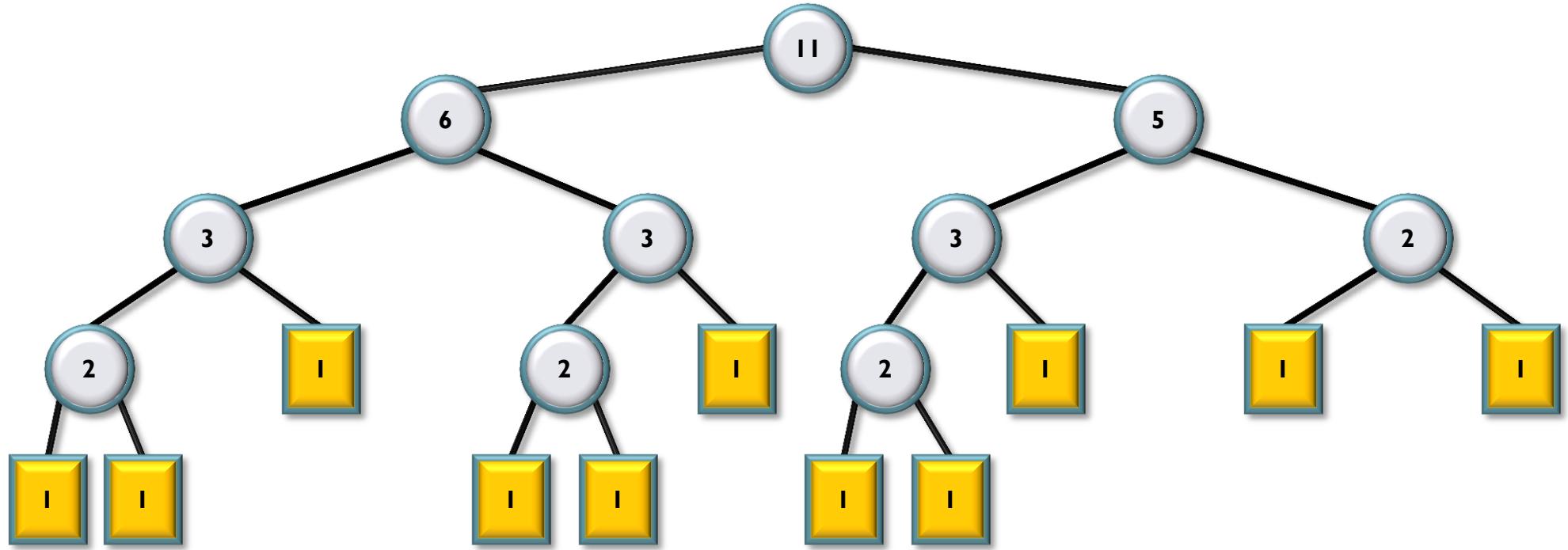
TINY EXAMPLE

```
max(0, 10)  
  Y max(0, 5)  
    T max(0, 2)  
      T max(0, 1)  
        max(0, 0) -> T  
        max(1, 1) -> I  
        max(2, 2) -> N  
  Y max(3, 5)  
    Y max(3, 4)  
      max(3, 3) -> Y  
      max(4, 4) -> E  
      max(5, 5) -> X  
  
...
```

TINY EXAMPLE

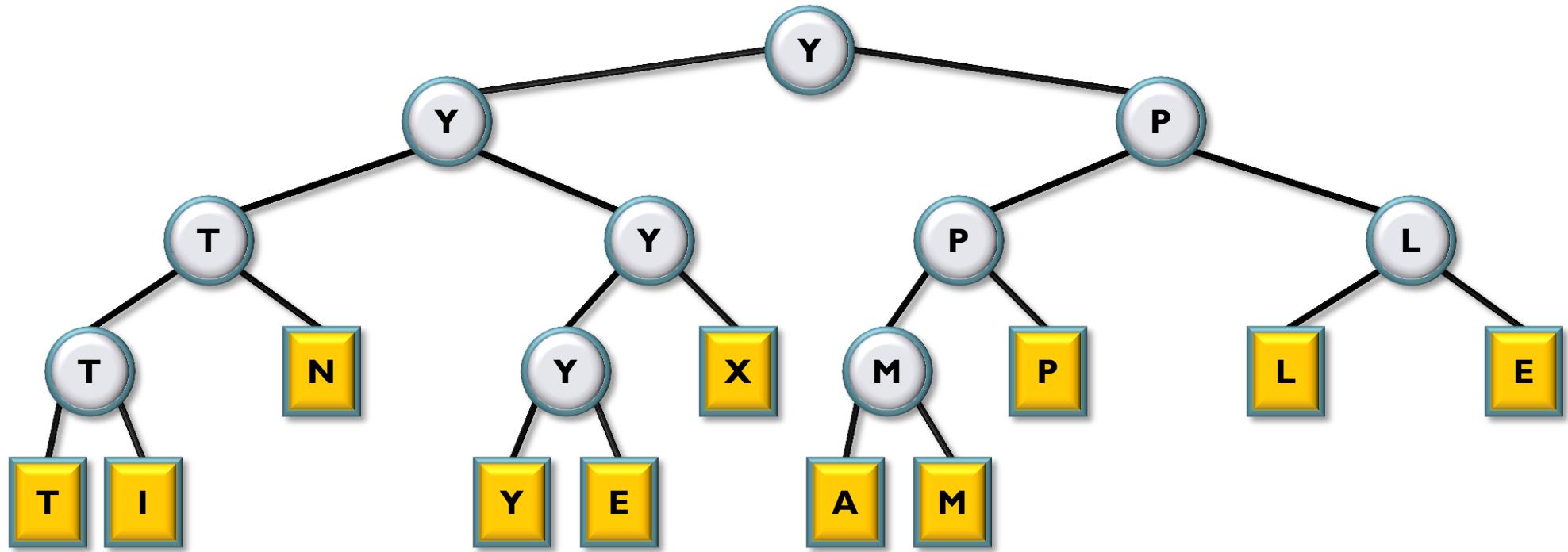
```
...  
  P max(6, 10)  
    P max(6, 8)  
      M max(6, 7)  
        max(6, 6) -> A  
        max(7, 7) -> M  
        max(8, 8) -> P  
  L max(9, 10)  
    max(9, 9) -> L  
    max(10, 10) -> E
```

Exemplo (3)



Tamanho dos problemas em cada chamada

Exemplo (4)



Valor devolvido em cada chamada

Torres de Hanoi (1)

- ▶ Qualquer discussão sobre recursividade e algoritmos divide-and-conquer não fica completa sem o velho problema das Torres de Hanoi.
- ▶ Existem três postes e N discos com um orifício central que permite que sejam colocados em qualquer dos postes.
- ▶ Os discos são todos de tamanhos diferentes e estão inicialmente num dos postes com o maior dos discos em baixo e os seguintes dispostos por ordem decrescente do seu raio.
- ▶ O problema consiste em mover os N discos para o poste à direita daquele em que estão originalmente obedecendo às seguintes regras:
 - ▶ Apenas se pode mover um disco de cada vez;
 - ▶ Nenhum disco pode ser colocado sobre outro de menor raio;
 - ▶ Os discos têm sempre que ser transferidos de um poste para outro.

Torres de Hanoi (2)

- ▶ A lenda diz que o mundo acabará quando um conjunto de monjes que habitam um convento na zona de Hanoi for capaz de realizar a tarefa para 40 discos de ouro usando três postes de diamante.
- ▶ O código abaixo apresenta uma solução recursiva para este problema.

```
void hanoi(int N, int +d)
{
    if (N == 0) return;
    hanoi(N-1, -d);
    shift(N, +d)
    hanoi(N-1, -d);
}
```

Primeiro mover os $N-1$ discos de cima para o poste à esquerda;
Depois, deslocar para o poste da direita o maior disco;
Finalmente, deslocar os $N-1$ mais pequenos para a esquerda mais uma vez.

- ▶ Onde “-” significa mover para a esquerda e “+” mover para a direita, circulando nos postes quando se atinge um dos extremos.

Torres de Hanoi (3)

- ▶ Propriedade 2 – O algoritmo recursivo divide-and-conquer para as torres de Hanoi produz uma solução com $2^N - 1$ movimentos.
- ▶ Demonstração – Basta escrever a recorrência que descreve a função apresentada. Para resolver um problema com N discos é preciso resolver um problema com $N-1$ discos, depois mover um disco para o poste final e no fim resolver um problema com $N-1$ discos de novo.

$$T_N = T_{N-1} + 1 + T_{N-1}, \quad \text{para } N > 1 \text{ com } T_1 = 1$$

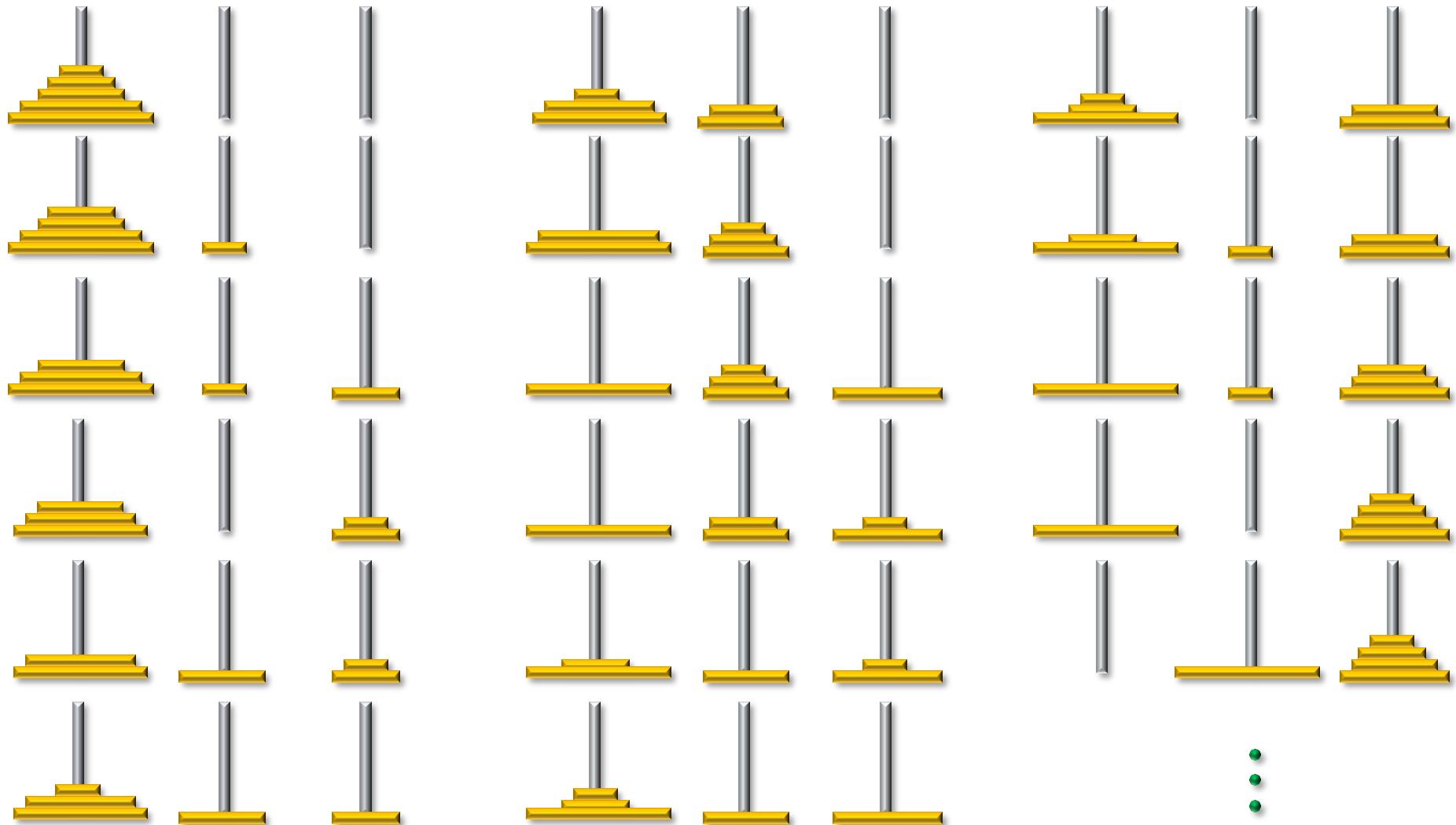
O resultado é verdadeiro para $N = 1$.

Assumindo, para todo o $k < N$, que o número de movimentos é de $2^k - 1$, aplicando a hipótese de indução, vem que para N se terá

$$T_N = 2(2^{N-1} - 1) + 1 = 2^N - 1$$

QED

Torres de Hanoi – Exemplo



Exemplos de algoritmos divide-and-conquer

- ▶ O algoritmo da procura binária é um caso em que cada chamada recursiva pega apenas numa das metades do problema original, deixando a outra metade fora através de uma operação instantânea

$$T_N = T_{N/2} + 1$$

- ▶ O algoritmo de ordenação quicksort divide a tabela em duas realizando primeiro um conjunto de operações $\mathcal{O}(N)$

$$T_N = N + T_k + T_{N-k}$$

- ▶ O algoritmo de ordenação mergesort chama-se a si próprio para cada uma das metades da tabela e depois combina as duas soluções obtidas através de um conjunto de operações $\mathcal{O}(N)$

$$T_N = 2T_{N/2} + N$$

Outras variantes de divide-and-conquer

- ▶ Outras variantes desta estratégia de desenho de algoritmos que são relevantes consistem em:
 - ▶ Divisão em partes de tamanhos variáveis;
 - ▶ Divisão em mais que duas partes;
 - ▶ Divisão em partes que se sobrepõem (as torres de Hanoi pode ser visto como um exemplo disto);
 - ▶ Realizam diversas quantidades de cálculos na componente não recursiva do algoritmo.
- ▶ Em geral, algoritmos divide-and-conquer realizam cálculos para dividir a entrada em partes ou para combinar os resultados obtidos em cada parte, ou ainda para facilitar os cálculos entre as duas chamadas
 - ▶ Ou seja, pode haver código antes, depois ou entre as duas chamadas recursivas que necessite ser analisado para concluir da complexidade global do algoritmo.

Programação Dinâmica

- ▶ Os algoritmos divide-and-conquer considerados até aqui dividem um problema em dois sub-problemas independentes.
- ▶ Quando os sub-problemas não são independentes a situação é mais complicada, porque implementações recursivas directas poderão conduzir a implementações que requerem enormes quantidades de tempo.
- ▶ Por exemplo, o programa abaixo calcula os números de Fibonacci recursivamente (NUNCA USAR ESTE PROGRAMA).

```
int F(int i)
{
    if (i < 1) return 0;
    if (i == 1) return 1;
    return F(i-1) + F(i-2);
}
```

Números de Fibonacci (1)

- ▶ O algoritmo recursivo apresentado é espectacularmente ineficiente.
- ▶ De facto, o número de chamadas recursivas para calcular F_N é exactamente F_{N+1} .
- ▶ Como F_{N+1} é da ordem de ϕ^{N+1} , com ϕ aproximadamente igual a 1,618 (número de ouro), o número de chamadas recursivas é exponencial em N .
- ▶ Ou seja, aquela implementação tem complexidade exponencial para determinar uma coisa tão simples como os números da série de Fibonacci.
 - ▶ Uma análise simples permite concluir que todas as chamadas acabarão por retornar os valores de F_0 e F_1 , a partir das quais se constroem os seguintes.
 - ▶ Em alternativa, é possível realizar aqueles cálculos num tempo proporcional a N , como se pode ver no algoritmo seguinte.

Números de Fibonacci (2)

```
F[0] = 0; F[1] = 1;  
for (i = 2; i <= N; i++)  
    F[i] = F[i-1] + F[i-2];
```

- ▶ Os números da série crescem exponencialmente, mas a tabela necessária é pequena em tamanho.
 - ▶ Por exemplo, $F_{45} = 1836311903$ é o maior número de Fibonacci que pode ser representado como um inteiro em 32 bits.
 - ▶ Assim, uma tabela de tamanho 46 é suficiente.
- ▶ Este exemplo serve para ilustrar uma outra técnica para determinar soluções numéricas para qualquer relação definida recursivamente.
- ▶ No caso dos números de Fibonacci, podemos até dispensar toda a tabela, dado ser apenas necessário manter em memória os dois últimos números.

Programação Dinâmica Ascendente

- ▶ Para a maior parte das recorrências numéricas existentes é necessário manter uma tabela com todos os valores conhecidos.
- ▶ Uma recorrência é uma função recursiva que envolva números inteiros.
- ▶ O exemplo dos números de Fibonacci permite-nos concluir que é possível avaliar estas funções calculando todos os seus valores por ordem crescente da sua ordem.
- ▶ A esta técnica dá-se o nome de programação dinâmica ascendente.
- ▶ Aplica-se a qualquer cálculo recursivo em que se pode guardar todos os valores calculados previamente.

Programação Dinâmica Descendente

- ▶ Como vimos, foi possível com a técnica anterior melhorar o tempo de execução de um algoritmo de exponencial para linear!!!
 - ▶ Merece pois a nossa melhor atenção.
- ▶ Uma outra técnica, denominada programação dinâmica descendente, permite executar chamadas recursivas com o mesmo (ou menor) custo que a programação dinâmica ascendente.
- ▶ A ideia é permitir que as sucessivas chamadas recursivas guardem os seus resultados para evitar mais chamadas recursivas, na medida em que cada chamada verifica primeiro se o valor que procura já foi calculado.

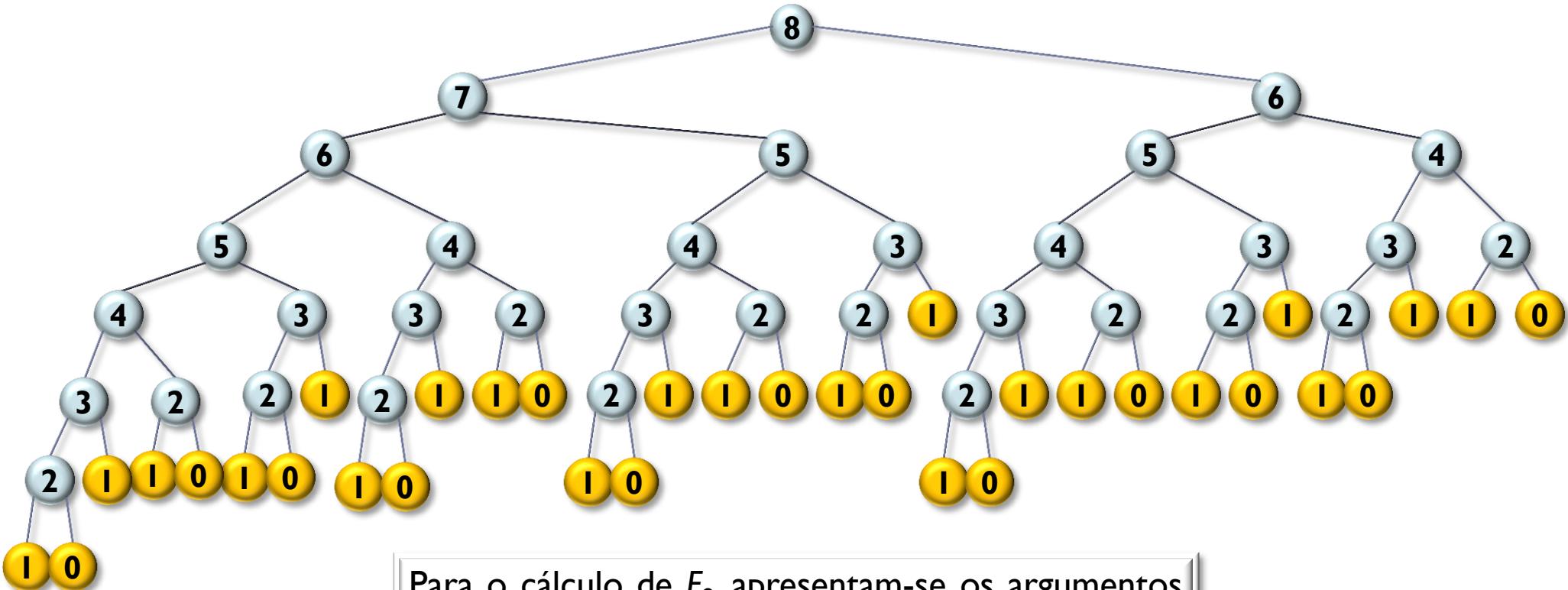
Números de Fibonacci com memorização

- ▶ Esta função, apesar de ser também recursiva, calcula os números de Fibonacci em tempo linear, na medida em que vai guardando na tabela `knownF` os valores que “descobre”.

```
int F(int i)
{
    int t;
    if (knownF[i] != unknown) return knownF[i];
    if (i == 0) t = 0;
    if (i == 1) t = 1;
    if (i > 1) t = F(i-1) + F(i-2);
    return knownF[i] = t;
}
```

Números de Fibonacci

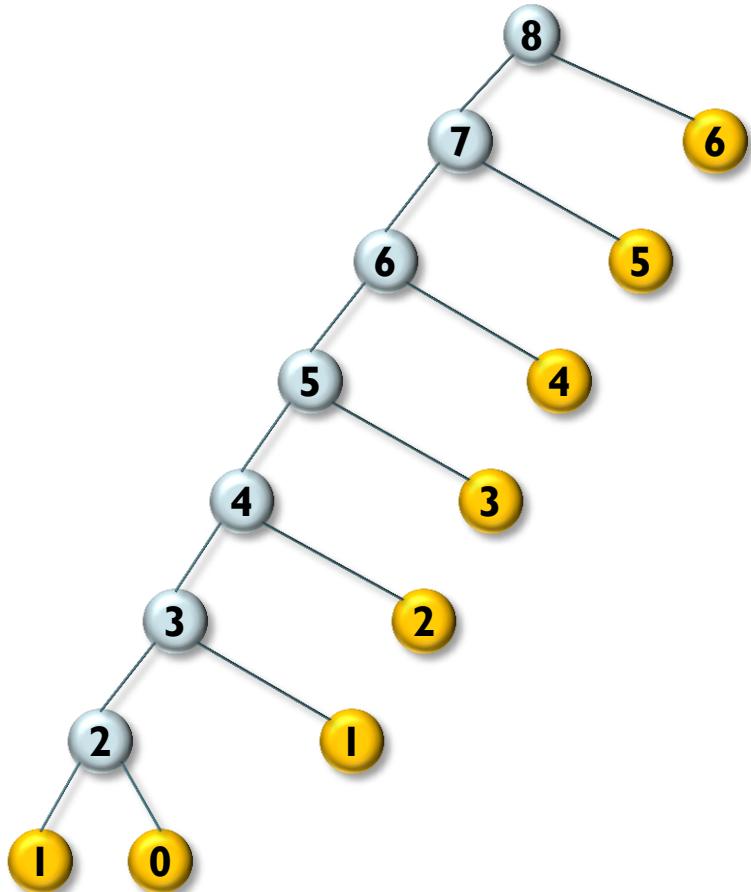
Primeira implementação



Para o cálculo de F_8 , apresentam-se os argumentos de cada uma das chamadas recursivas realizadas.
Como se vê, os mesmos problemas são resolvidos em diferentes chamadas sem ligação entre si.

Números de Fibonacci

Segunda implementação



Para o cálculo de F_8 , apresentam-se os argumentos de cada uma das chamadas recursivas realizadas.

Quando se concretiza a segunda chamada em cada nível já o problema foi resolvido como subproblema da primeira chamada.

Programação Dinâmica - Propriedade

- ▶ Propriedade 3 – A Programação Dinâmica reduz o tempo de computação de uma função recursiva para ser no máximo igual ao tempo necessário para avaliar a função para todos os argumentos iguais ou menores que o argumento dado, considerando o tempo de uma chamada recursiva como constante.
- ▶ Esta propriedade é estabelecida assumindo que é possível guardar em memória todos os valores da função a calcular para os argumentos menores que o da chamada em questão.

Programação Dinâmica – Síntese (1)

- ▶ Dada uma recorrência definida sobre inteiros, é possível aplicar PD ascendente, calculando e armazenando as soluções para as pequenas instâncias construtivamente até atingir a dimensão desejada.
 - ▶ Tipicamente em implementações não recursivas.
 - ▶ Pre-cálculo de soluções para instâncias menores.
- ▶ Ou então, é possível aplicar a PD descendente deixando ao processo recursivo o papel de ir registando as soluções para os problemas de menor dimensão à medida que são encontrados e utilizando-os como primeira opção em cada chamada recursiva, quando já existem.
 - ▶ Aqui faz-se uso da recursividade.
 - ▶ Salvam-se e usam-se valores conhecidos de instâncias menores.

Programação Dinâmica – Síntese (2)

- ▶ Também é possível estender este tipo de metodologias para recorrências que tenham mais que um argumento inteiro.
 - ▶ Neste caso ter-se-á que construir tabelas multidimensionais onde armazenar os valores intermédios.
- ▶ Um exemplo clássico é o cálculo de $C(n, k)$, combinações de k objectos diferentes dentro de um conjunto de n objectos, com $0 \leq k \leq n$.
- ▶ Esta função pode ser definida de duas formas

$$C(n, k) = n!/[k!(n-k)!]$$

$$C(n, k) = C(n-1, k-1) + C(n-1, k), \text{ com } C(n,n) = C(n,0) = 1 \text{ para todo o } n$$

- ▶ Problemas numéricos na primeira forma.
- ▶ Recursão directa da segunda forma é exponencial, mas a PD conduz ao triângulo de Pascal, cuja complexidade é nk .

Síntese da Aula 1

▶ Recursividade

- ▶ Recursividade e árvores
- ▶ Algoritmos recursivos
- ▶ Programas recursivos e não recursivos
- ▶ Características básicas e demonstrações
- ▶ Exemplos

▶ Divide-and-conquer

- ▶ Decomposição em sub-problemas independentes
- ▶ Torres de Hanoi
- ▶ Exemplos e variantes

▶ Programação dinâmica

- ▶ Programação dinâmica ascendente
- ▶ Programação dinâmica descendente

Árvores – Introdução (1)

- ▶ Árvores são uma abstração matemática que desempenha um papel central no projecto e análise de algoritmos:
 - ▶ Usamos árvores para descrever as propriedades dinâmicas de algoritmos;
 - ▶ Construimos e usamos estruturas de dados explícitas que são implementações concretas de árvores.
- ▶ Já vimos alguns exemplos:
 - ▶ Os algoritmos para o problema da conectividade que estudámos assentam em árvores e foi dessa forma que interpretámos os resultados;
 - ▶ Descrevemos a estrutura de chamadas de algoritmos recursivos através de árvores.

Árvores – Introdução (2)

- ▶ Árvores, enquanto estruturas para representação de informação, fazem parte da nossa vida diária:
 - ▶ Estão presentes quando se traçam árvores genealógicas;
 - ▶ Usam-se para representar um processo de eliminatórias em torneios desportivos;
 - ▶ Estão presentes em organigramas de empresas;
 - ▶ Em parsers para linguagens;
 - ▶ Etc.
- ▶ Em aplicações em computadores, um dos mais familiares usos de árvores surge na organização de sistemas de ficheiros:
 - ▶ Os ficheiros são guardados em directorias, por vezes também chamados pastas, que são definidos recursivamente como sequências de directorias e ficheiros;
 - ▶ Esta definição recursiva reflecte uma decomposição recursiva natural que é característica de um certo tipo de árvores.

Tipos de árvores

- ▶ Existem vários tipos de árvores, pelo que é necessário e importante compreender a distinção entre o conceito abstracto e uma representação concreta que se pretenda, ou necessite, usar.
- ▶ Dos vários tipos de árvores existentes destacam-se, por ordem decrescente de generalidade:
 - ▶ Árvores;
 - ▶ Árvores com raíz (rooted trees);
 - ▶ Árvores ordenadas;
 - ▶ Árvores M-árias e binárias.

Árvores – Definições (1)

- ▶ Uma **árvore** é uma colecção de vértices e arestas que satisfazem um determinado conjunto de propriedades e requisitos.
- ▶ Um **vértice** é um objecto simples (também referido como nó) que pode possuir um nome e ter associado consigo um conjunto de informações complementares de caracterização.
- ▶ Uma **aresta** é uma ligação entre dois vértices.
- ▶ Um **caminho**, ou percurso, numa árvore é uma lista de vértices distintos em que quaisquer dois vértices sucessivos estão ligados por arestas pertencentes à árvore.

Árvores – Definições (2)

- ▶ Uma das propriedades estruturalmente definidores de uma árvore é o facto de existir apenas um caminho ligando quaisquer dois vértices.
- ▶ Se existir mais que um caminho entre algum par de vértices tem-se um **grafo**; não uma árvore.
- ▶ A um conjunto disjunto de árvores dá-se o nome de **floresta**.

Árvores com raíz (1)

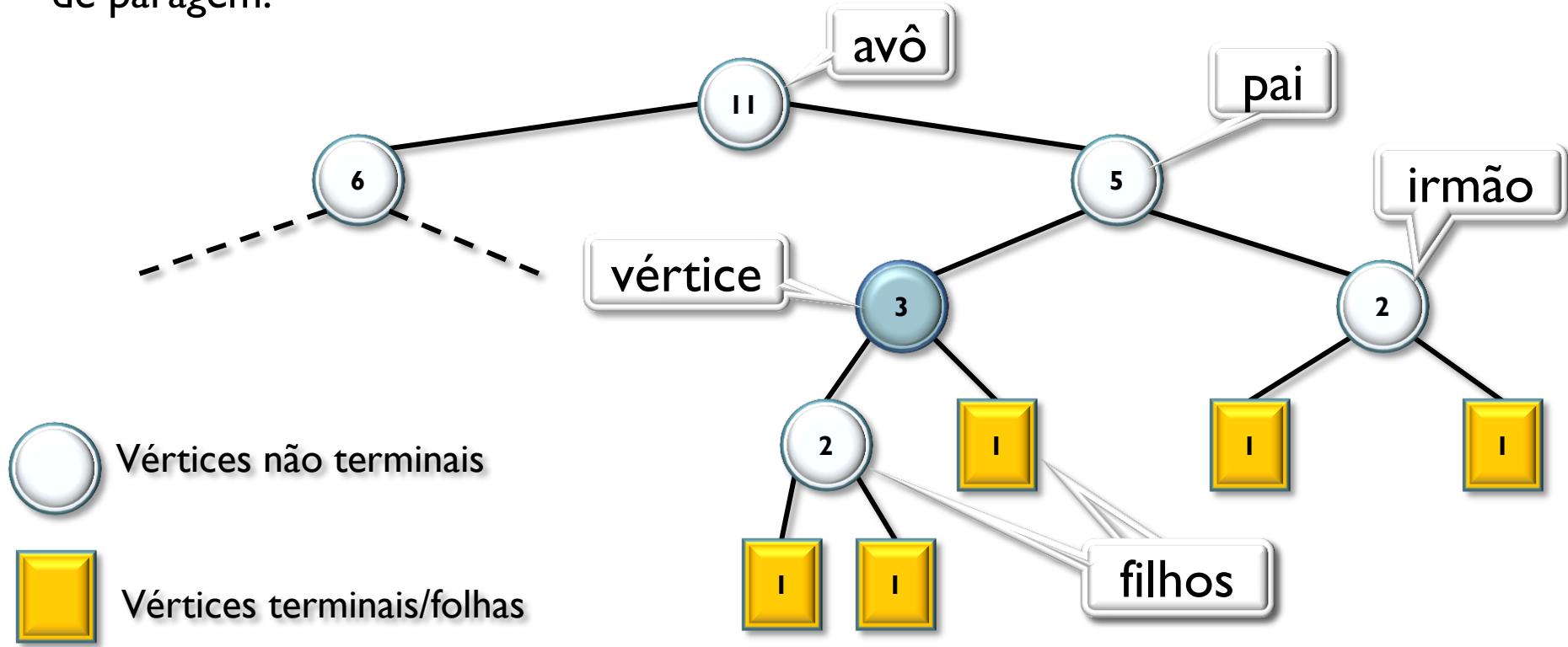
- ▶ Uma **árvore com raíz** – rooted tree –, é tal que um dos vértices é designado como raiz.
 - ▶ Em ciências da computação é habitual estar-se a falar de árvores com raíz quando se usa apenas o termo árvore e para as restantes usa-se a expressão árvore livre – free tree.
- ▶ Numa árvore com raíz qualquer vértice é a raiz da **sub-árvore** constituída por si próprio e pelo vértices que lhe estão abaixo.
- ▶ Existe exactamente um caminho entre o vértice raiz e qualquer vértice na árvore.
 - ▶ A definição implica que não existe sentido atribuído às arestas; mas costuma-se pensar na existência de um sentido descendente a partir da raiz ou inverso, dependendo das aplicações.

Árvores com raiz (2)

- ▶ Numa árvore com raiz cada vértice, excepto a raiz, possui apenas um vértice acima de si, a que se dá o nome de pai.
- ▶ Os vértices directamente abaixo de um vértice são designados como filhos.
- ▶ Por vezes esta analogia familiar é levada mais longe, usando os termos irmão e avô de um vértice.
- ▶ Vértices sem filhos designam-se folhas, ou vértices terminais.
- ▶ Vértices que possuam pelo menos um filho são por vezes designados como não terminais.

Árvores com raiz (3)

- Nas árvores que traçámos para ilustrar o funcionamento dos algoritmos recursivos os vértices não terminais (círculos azuis) representam chamadas que implicam nova chamada recursiva, enquanto que os vértices terminais (quadrados ou círculos amarelos) representam chamadas que se concretizam nas condições de paragem.



Árvores ordenadas

- ▶ Em algumas aplicações, a forma como os filhos de cada vértice estão ordenados é relevante; noutras aplicações não é.
- ▶ Uma **árvore ordenada** é uma árvore com raiz em que a ordem dos filhos em cada nó é especificada.

Árvores binárias (1)

- ▶ Se cada vértice tem que possuir um número específico de filhos numa ordem específica, tem-se uma árvore **M-ária**.
- ▶ Em tais árvores é por vezes necessário definir **vértices externos** especiais que não possuam filhos.
- ▶ O tipo mais simples de árvore M-ária é uma árvore binária.
- ▶ Uma **árvore binária** é uma árvore ordenada com dois tipos de vértices:
 - ▶ Vértices internos com exactamente dois filhos;
 - ▶ Vértices externos sem filhos.
- ▶ Como os dois filhos de cada vértice interno estão ordenados, referimo-nos a cada um deles como o **filho esquerdo** e o **filho direito**.

Árvores binárias (2)

- ▶ Cada vértice interno tem que possuir um filho esquerdo e um filho direito, mas um ou ambos poderão ser um vértice externo.
- ▶ Uma **folha** num árvore M-ária é um vértice interno cujos filhos são todos vértices externos (definição alterada para o caso de árvores M-árias).
- ▶ Definição I – Uma **árvore binária** é um vértice externo ou um vértice interno ligado a um par de árvores binárias, que se designam como a sub-árvore esquerda e a sub-árvore direita desse vértice.
- ▶ Existem várias formas de representar ou implementar uma árvore binária em programação.
- ▶ A mais frequente é uma estrutura com dois ponteiros (ponteiro esquerdo e ponteiro direito) para vértices internos.

Definição de tipo para Árvores Binárias (1)

```
typedef struct node link;
struct node {Item item; link *l, *r;}
```

- ▶ O código acima mais não faz do que concretizar a definição de árvore binária que vimos atrás.
- ▶ Variáveis do tipo **link** serão usadas apenas como ponteiros para vértices e um vértice consiste em um **Item** e um par de ponteiros do tipo **link**.
 - ▶ Por exemplo, implementamos a operação abstracta “mover para a sub-árvore da esquerda” com a instrução **x = x->l**.
 - ▶ Devido a todas as diferentes representações possíveis para árvores binárias, seria normal definir-se uma ADT (abstract data type) que incluísse as operações mais importantes, como se tem feito até aqui, para separar a utilização da implementação.

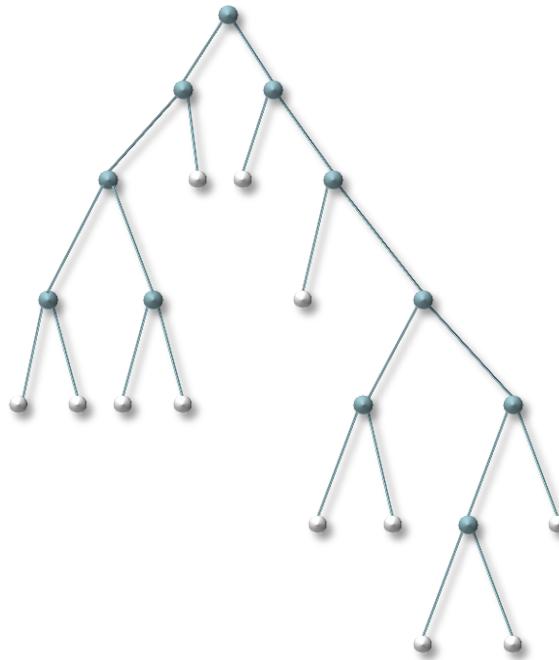
Definição de tipo para Árvores Binárias (2)

- ▶ Muitas vezes isso não é feito porque:
 - ▶ Na maior parte dos casos usa-se a representação com dois ponteiros;
 - ▶ Usam-se árvores para implementar outros ADT de ordem superior, onde o enfoque principal é nestes últimos;
 - ▶ Determinados algoritmos têm uma eficiência que depende da representação particular – esse facto poderia perder-se usando uma ADT genérica.
- ▶ Estas são as mesmas razões que nos levam a usar as representações explícitas para tabelas e listas simples.
- ▶ A representação para árvores binárias agora apresentada é um instrumento fundamental que agora adicionamos àquelas duas.
- ▶ Para listas simples enunciámos um conjunto de operações elementares para inserção e remoção de elementos.

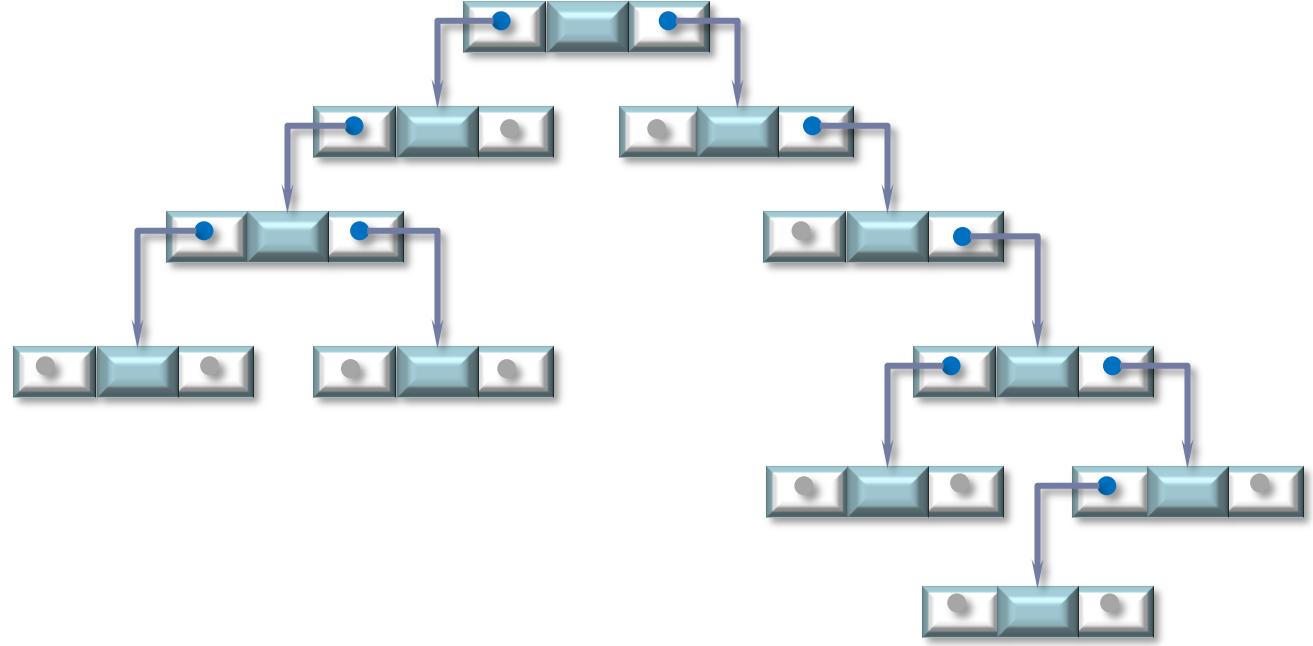
Definição de tipo para Árvores Binárias (3)

- ▶ Nem todas estas operações podem ser consideradas elementares para árvores binárias, por causa da existência de dois ponteiros.
- ▶ Se pretendermos remover um vértice de uma árvore binária temos que ter em conta que existem dois filhos depois de retirá-lo mas apenas um pai.
- ▶ Algumas operações naturais não possuem esta dificuldade:
 - ▶ Inserir um vértice novo no fundo;
 - ▶ Remover uma folha;
 - ▶ Combinar duas árvores, criando uma nova raiz.

Exemplo



Uma instância de
árvore binária



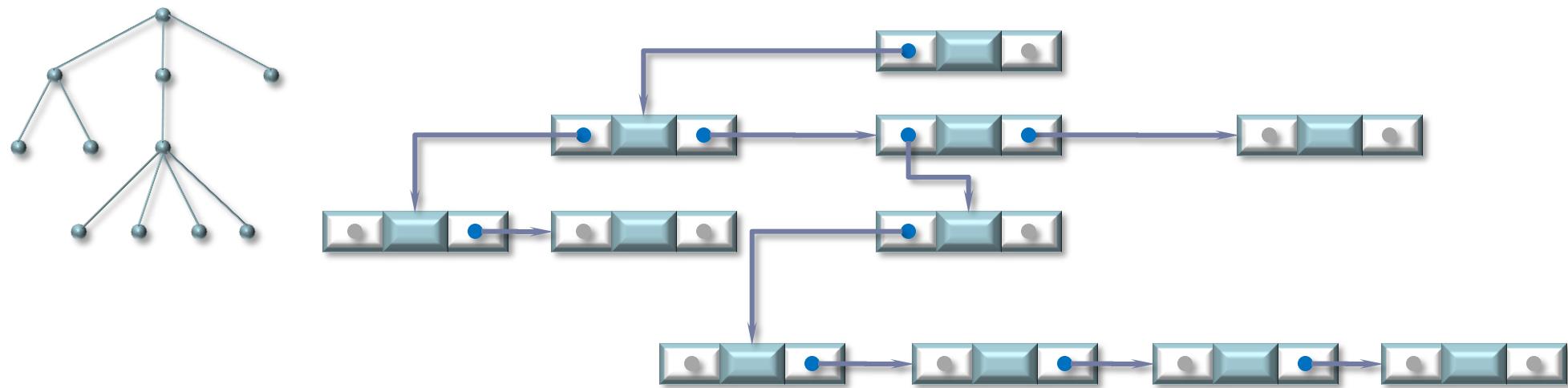
Correspondente representação
em computador

Árvores – Definições e representações (1)

- ▶ **Definição 2** – Uma **árvore M-ária** é um vértice externo ou um vértice interno ligado a uma sequência ordenada de M árvores que também são M -árias.
- ▶ Normalmente representam-se os vértices em árvores M -árias como estruturas com M ponteiros individualmente nomeados ou através de uma tabela de M ponteiros.
- ▶ **Definição 3** – Uma **árvore** (também designada como **árvore ordenada**) é um vértice (chamado raiz) ligado a uma sequência de árvores disjuntas. A essa sequência dá-se o nome de floresta.
- ▶ A distinção essencial entre uma árvore ordenada e uma árvore M -ária é que na primeira qualquer vértice pode possuir um número arbitrário de filhos.

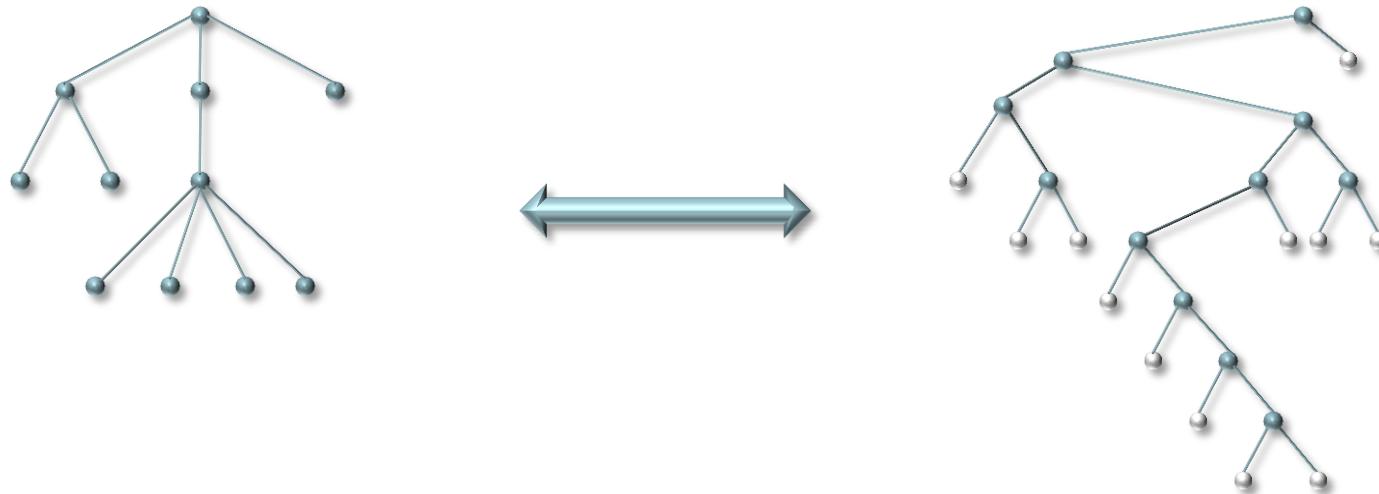
Árvores – Definições e representações (2)

- ▶ Porque cada vértice de uma árvore ordenada pode possuir um número arbitrário de filhos, a representação natural é uma lista simples de ponteiros, em vez de uma tabela, para os filhos.



Árvores – Definições e representações (3)

- ▶ Propriedade 4 – Existe uma correspondência de um para um entre árvores binárias e florestas ordenadas.
- ▶ Podemos representar qualquer floresta como uma árvore binária fazendo o ponteiro da esquerda de cada vértice apontar para o seu filho mais à esquerda e o ponteiro da direita apontar para o seu irmão à direita.



Árvores – Definições e representações (4)

- ▶ **Definição 4** – Uma **árvore com raíz** (ou **árvore não ordenada**) é um vértice (denominado raiz) ligado a um conjunto múltiplo de árvores com raiz. (Esse conjunto múltiplo é designado como uma floresta não ordenada)
- ▶ As árvores que encontrámos quando estudámos o problema da conectividade eram não ordenadas.
- ▶ Estas podem ser representadas da mesma forma que as árvores ordenadas sem que se atribua significado particular à ordem pela qual se organizam os filhos de cada vértice.
- ▶ O tipo mais geral de árvores são aquelas em que não existe a necessidade de, ou não faz sentido, definir um vértice como raiz.
 - ▶ Este tipo mais geral de árvores será discutido no contexto dos grafos.

Propriedades das Árvores Binárias (1)

- ▶ Propriedade 5 – Uma árvore binária com N vértices internos possui $N+1$ vértices externos.
- ▶ Demonstração – O resultado estabelece-se por indução. Uma árvore binária sem vértices internos possui exactamente um vértice externo, pelo que propriedade se verifica para $N = 0$.

Para $N > 0$, qualquer árvore binária com N vértices internos possui k vértices internos na sub-árvore da esquerda e $N-1-k$ vértices internos na sub-árvore da direita, para algum k entre 0 e $N-1$, porque a raíz é um vértice interno.

Pela hipótese de indução a sub-árvore da esquerda possui $k+1$ vértices externos e a sub-árvore da direita possui $N-k$ vértices externos, o que perfaz um total de $N+1$.

QED

Propriedades das Árvores Binárias (2)

- ▶ Propriedade 6 – Uma árvore binária com N vértices internos possui $2N$ arestas: $N-1$ arestas para vértices internos e $N+1$ arestas para vértices externos.
- ▶ Demonstração – Em qualquer árvore com raíz qualquer vértice, excepto a raiz, tem apenas um pai e cada aresta liga um vértice ao seu pai.

Portanto existem $N-1$ arestas ligando vértices internos.

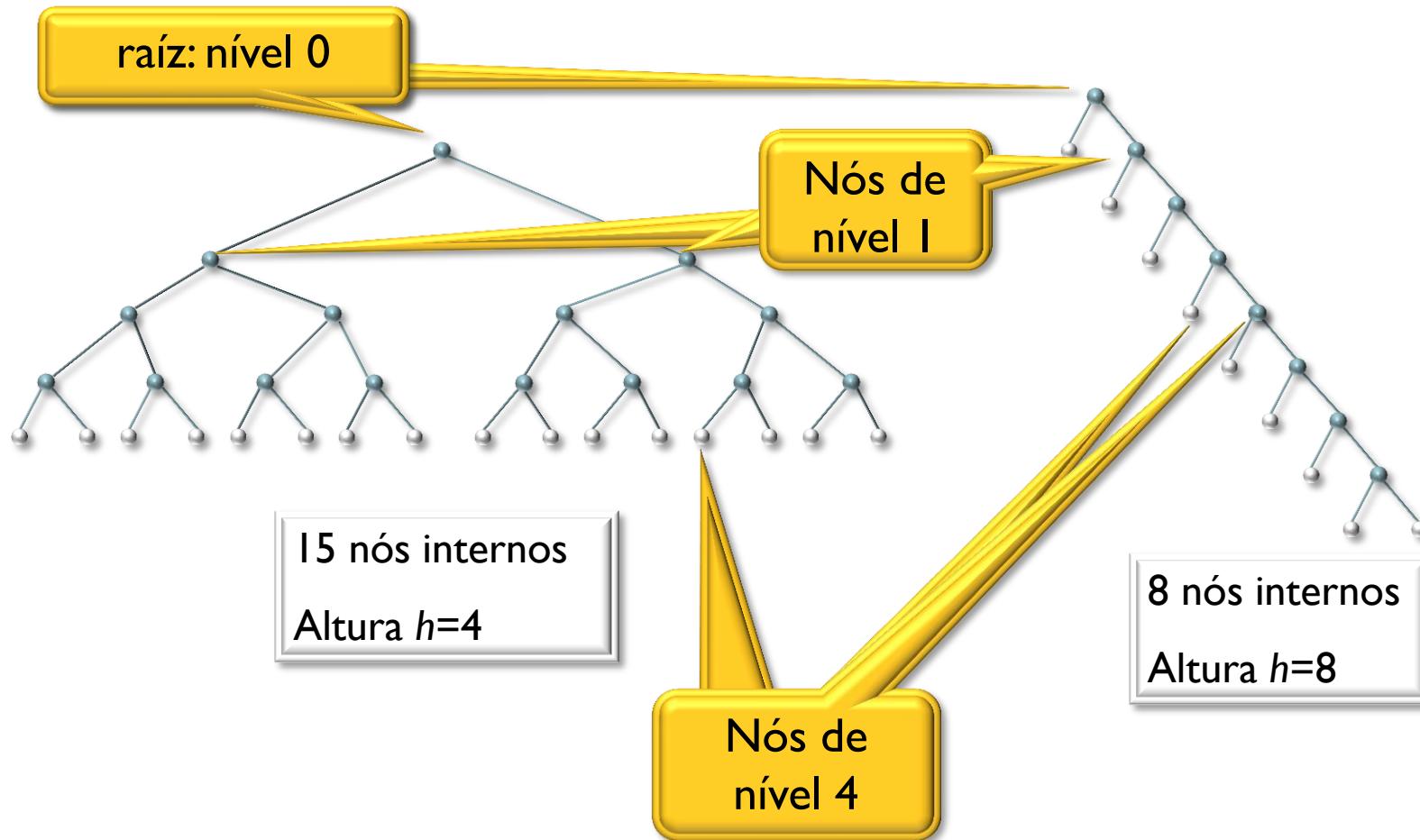
E cada um dos $N+1$ vértices externos possui uma aresta para o seu pai.

QED

Propriedades das Árvores Binárias (3)

- ▶ **Definição 5** – O **nível** de um vértice numa árvore é uma unidade superior ao nível do seu pai, sendo que a raiz tem nível 0.
A **altura** de uma árvore é o máximo dos níveis de todos os vértices que a constituem.
O **comprimento de percurso** (path length) de uma árvore é a soma dos níveis de todos os vértices da árvore.
O **comprimento de percurso interno** de uma árvore binária é a soma do níveis de todos os vértices internos.
O **comprimento de percurso externo** de uma árvore binária é a soma dos níveis de todos os seus vértices externos.
- ▶ **Propriedade 7** – A altura de uma árvore binária com N vértices internos é pelo menos $\lg N$ e no máximo $N-1$.

Propriedades das Árvores Binárias (4)



Síntese da Aula 2

- ▶ Introdução às árvores
 - ▶ Tipos de árvores;
 - ▶ Definições;
 - ▶ Árvores com raiz; árvores ordenadas; árvores M-árias; árvores binárias.
- ▶ Definições de tipos e representações
 - ▶ Árvores binárias;
 - ▶ Árvores M-árias;
 - ▶ Árvores ordenadas;
 - ▶ Árvores com raiz;
 - ▶ Árvores gerais
- ▶ Propriedades das Árvores Binárias

Varrimento em árvores binárias (1)

- ▶ Antes de avançarmos para algoritmos de construção de árvores, vamos analisar algoritmos para a mais elementar das operações em árvores: **varrimento**.
- ▶ Dado um ponteiro para uma árvore, pretende-se processar cada vértice da árvore de forma sistemática.
- ▶ Vamos concentrar-nos em árvores binárias.
- ▶ Assim, como cada vértice possui dois ponteiros, para a sub-árvore esquerda e para a sub-árvore direita, existem três formas diferentes de executar o varrimento:
 - ▶ **Pré-fixado** (Preorder) – em que se processa primeiro um vértice e depois as sub-árvores esquerda e direita;
 - ▶ **In-fixado** (Inorder) – em que se processa a sub-árvore esquerda, depois o vértice e depois a sub-árvore direita;
 - ▶ **Pós-fixado** (Postorder) – em que se processam as sub-árvores esquerda e direita e depois o vértice.

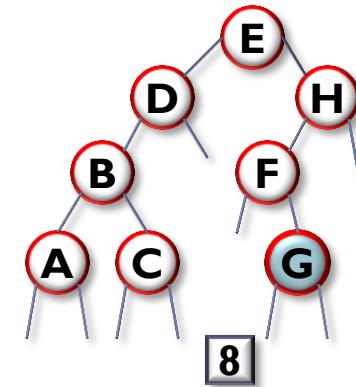
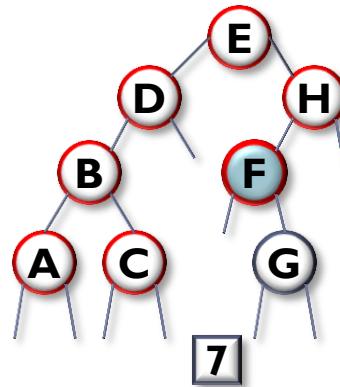
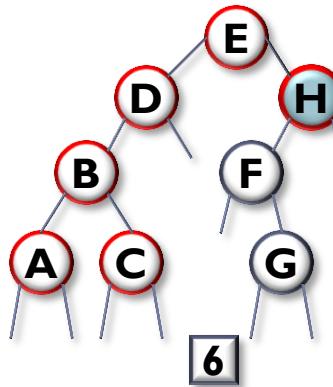
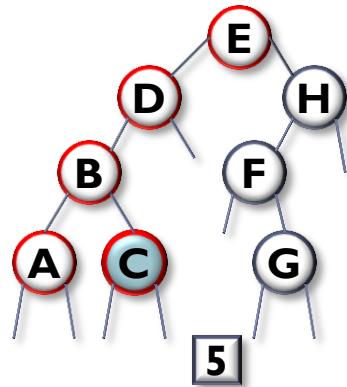
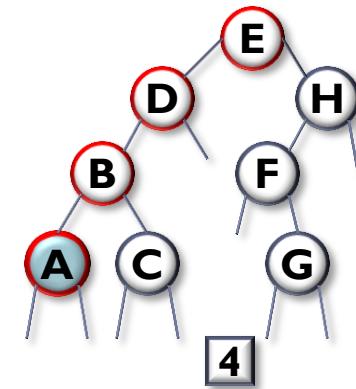
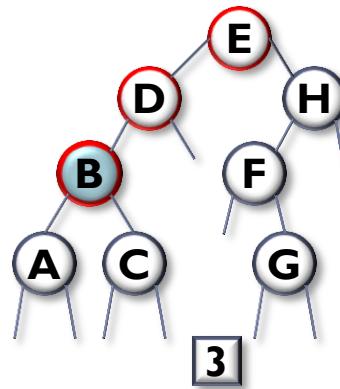
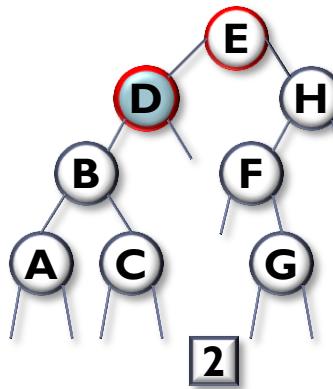
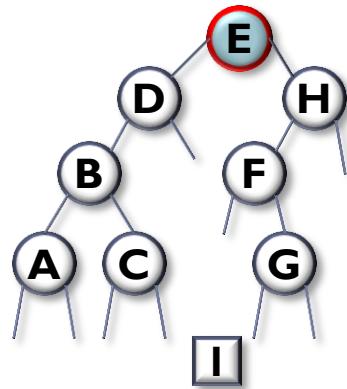
Varrimento em árvores binárias (2)

- ▶ A implementação destes três mecanismos é muito simples quando se usam funções recursivas.
- ▶ O código abaixo implementa o mecanismo pré-fixado

```
void traverse(link *h, void (*visit) (link))
{
    if (h == NULL) return;
    (*visit) (h);
    traverse(h->l, visit);
    traverse(h->r, visit);
}
```

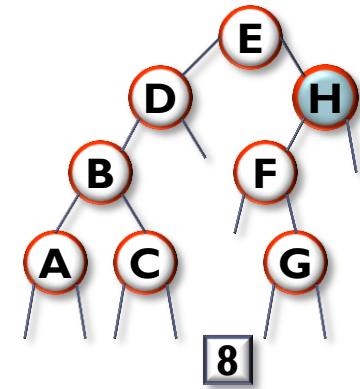
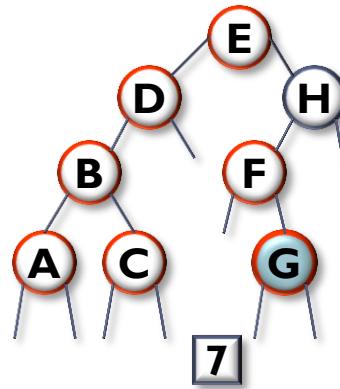
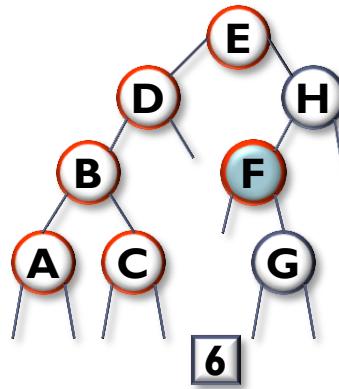
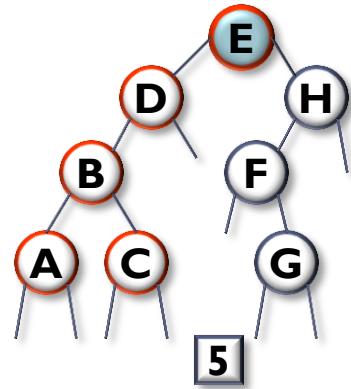
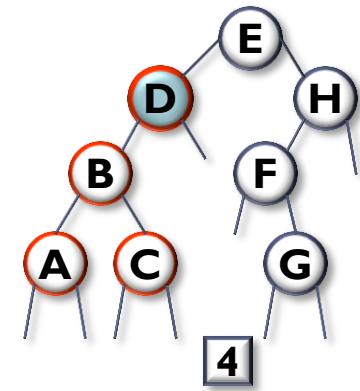
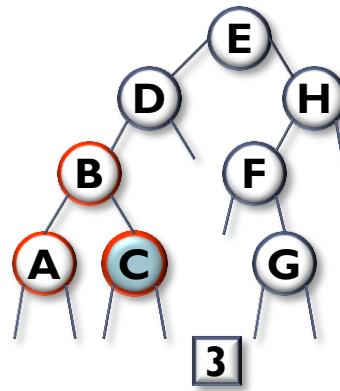
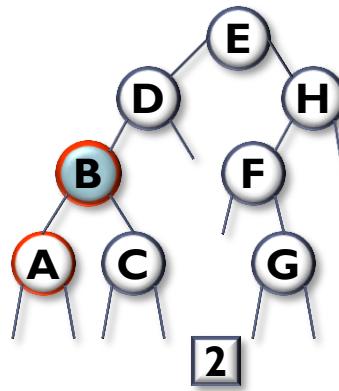
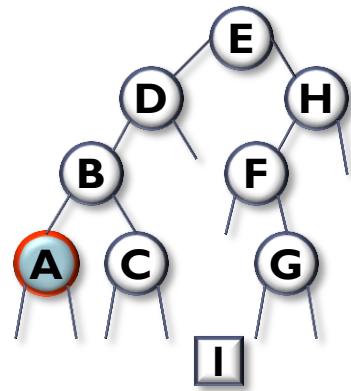
- ▶ Para implementar os outros dois mecanismos basta alterar a posição em que se realiza a chamada da função **visit**.
- ▶ Notar que este esquema de varrimento assenta na estratégia *divide-and-conquer*.

Varrimento Pré-fixado



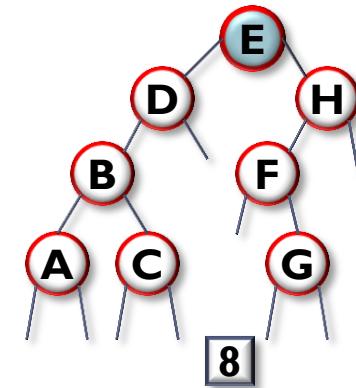
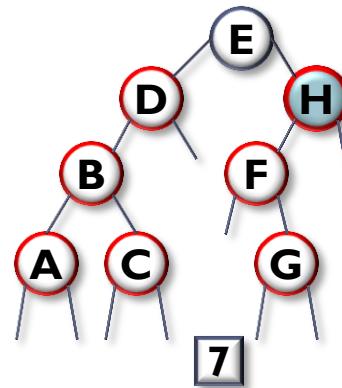
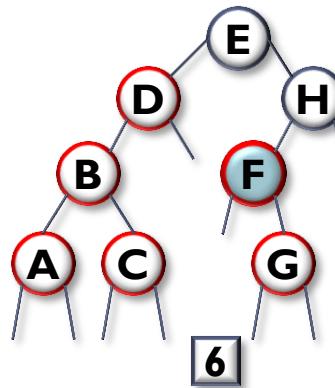
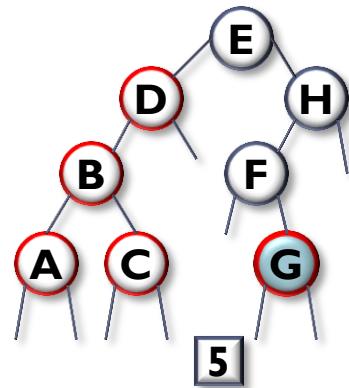
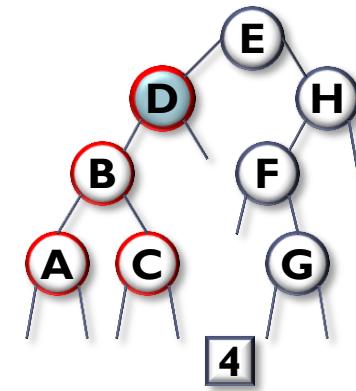
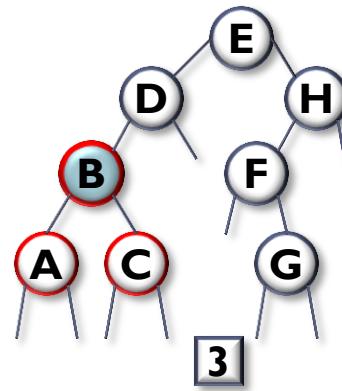
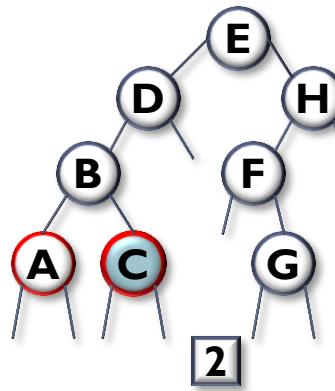
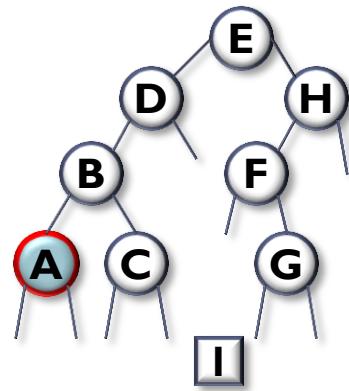
E D B A C H F G

Varrimento In-fixado



A B C D E F G H

Varrimento Pós-fixado



A C B D G F H E

Versões não recursivas (1)

- ▶ Em algumas circunstâncias pode ser útil considerar implementações não recursivas que fazem uso explícito de uma pilha.
- ▶ Considere-se uma pilha que tanto pode armazenar itens como árvores, inicializada com a árvore que se pretende processar.
- ▶ O algoritmo prossegue num ciclo em que se retira o elemento do topo da pilha para processamento, continuando enquanto a pilha não estiver vazia.
 - ▶ Se o elemento retirado da pilha for um item, visita-se;
 - ▶ Se o elemento retirado da pilha for uma árvore então a sequência de operações de inserção na pilha depende do tipo de varrimento desejado.

Versões não recursivas (2)

- ▶ Se a ordem desejada for pré-fixada, insere-se primeiro a sub-árvore da direita, depois a da esquerda e depois o vértice raiz.
- ▶ Se a ordem desejada for in-fixada, insere-se primeiro a sub-árvore direita, depois o vértice raiz e depois a sub-árvore esquerda.
- ▶ Para a ordem pós-fixada, insere-se primeiro o vértice raiz, depois a sub-árvore direita e depois a esquerda.
- ▶ Não se inserem vértices nulos na pilha.
- ▶ Deverá ser fácil verificar por indução que as sequências acima produzem os mesmos resultados que as versões recursivas.

Versões não recursivas (3)

```
void traverse(link *h, void (*visit)(link))
{
    STACKinit(max);
    STACKpush(h);
    while (!STACKempty())
    {
        (*visit) (h=STACKpop());
        if (h->r != NULL) STACKpush(h->r);
        if (h->l != NULL) STACKpush(h->l);
    }
}
```

- ▶ Acima encontra-se uma versão não recursiva do varrimento pré-fixado.
- ▶ É possível implementar uma versão mais eficiente, evitando inserir os objectos que vão ser removidos imediatamente.

Outros varrimentos e outras árvores

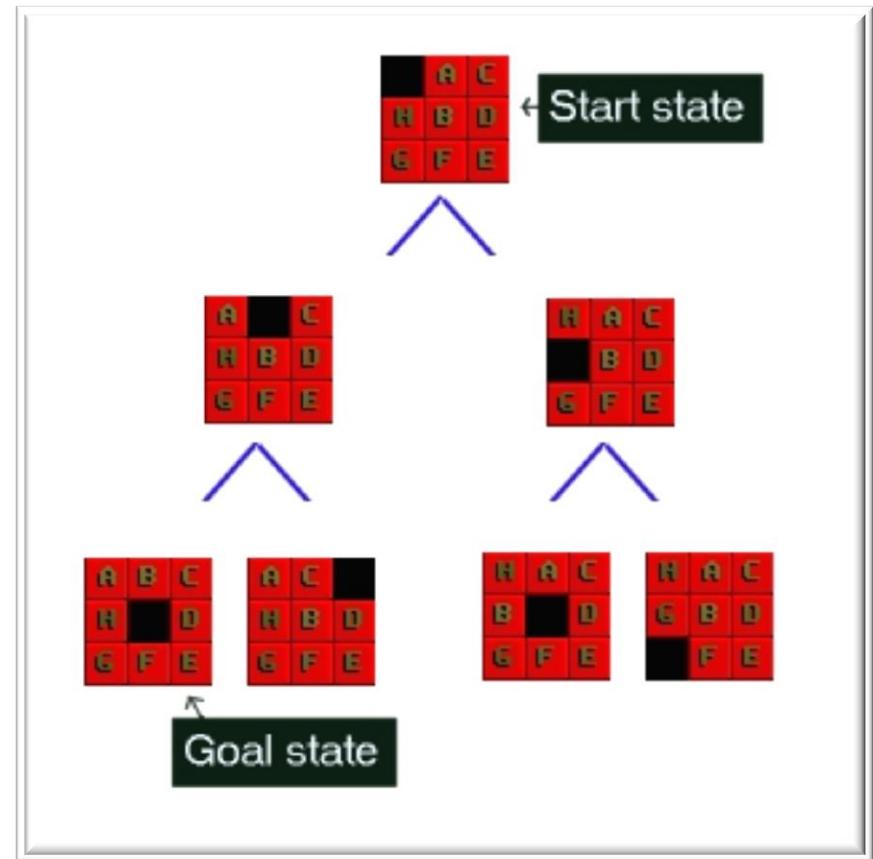
- ▶ Existe uma outra possibilidade para varrimento de árvores, que consiste em processá-los por níveis e em cada nível da esquerda para a direita.
- ▶ Este é o varrimento em largura, também conhecido como level-order.
- ▶ Para implementar este tipo de varrimento, basta usar uma fila em vez da pilha.
- ▶ Para outras árvores não binárias os varrimentos pré-fixado, pós-fixado e em largura são bem definidos:
 - ▶ Pré-fixado – visitar primeiro a raiz e depois cada uma das sub-árvores.
 - ▶ Pós-fixado – visitar cada uma das sub-árvores e depois a raiz.
 - ▶ Largura – tem a mesma caracterização que para árvores binárias.
- ▶ Para as implementações destas basta generalizar as versões não recursivas que estudámos para as árvores binárias.

Árvores e algoritmos de procura (1)

- ▶ Em muitos contextos torna-se necessário procurar um determinado conjunto de acções que, quando executadas em sequência, permitem atingir algum objectivo.
- ▶ Exemplos:
 - ▶ 8-puzzle
 - ▶ Sudoku
 - ▶ Mahjong
 - ▶ Etc.
- ▶ Existe uma íntima relação entre os algoritmos de procura para resolver este tipo de problemas e árvores genéricas.
- ▶ Abordaremos este tema com mais detalhe no capítulo de Grafos
 - ▶ Aqui faz-se apenas uma breve discussão introdutória

Árvores e algoritmos de procura (2)

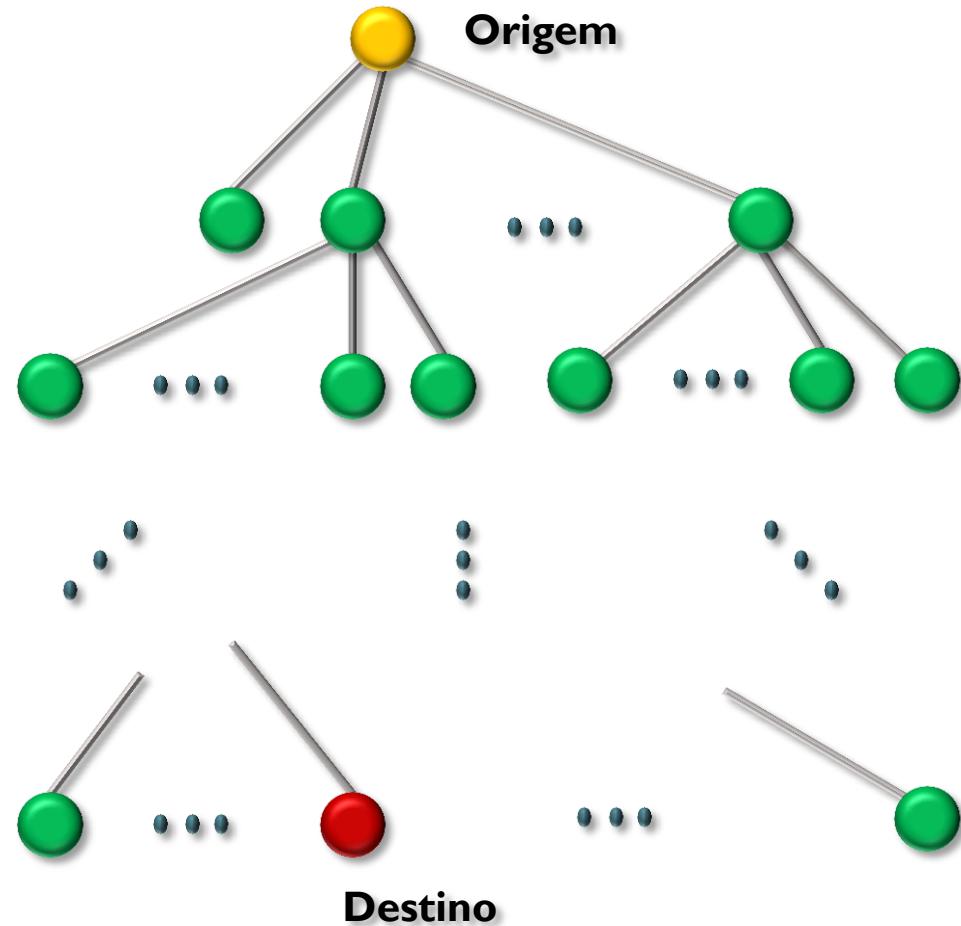
- ▶ Tomemos o 8-puzzle como exemplo
- ▶ Para cada matriz existe um conjunto de acções legais
 - ▶ Baixo, Cima, Esquerda, Direita
 - ▶ Para algumas configurações nem todas as 4 acções são legais
- ▶ Se a partir de uma dada matriz explorarmos todas as alternativas, acabaremos por encontrar a solução, se existir
 - ▶ Durante a procura poderemos construir uma árvore



Retirado de <http://www.heyes-jones.com/astar.html>

Árvores e algoritmos de procura (3)

- ▶ Assim, tomando uma dada posição de partida, torna-se necessário encontrar um caminho executando apenas acções legais.
- ▶ Em cada ponto do caminho poderá existir mais que uma acção legal
 - ▶ Conceptualmente, é como se existisse de facto uma árvore
 - ▶ No entanto, a estrutura arbórea serve apenas para ilustrar o espaço de procura e como se navega nele



Algoritmos de Procura (1)

- ▶ Um algoritmo de procura consiste na definição do processo de navegação naquelas árvores conceptuais
 - ▶ Notar que não é necessário, nem sequer desejável, que toda a árvore esteja representada em memória
 - ▶ O que se pretende é ser-se capaz de navegar no espaço de acções legais até que se encontre o objectivo (Destino) e que depois seja possível indicar como se chegou até esse ponto
- ▶ Existem três procedimentos de procura gerais, que se inspiram nos procedimentos de varrimento já discutidos
 - ▶ Procura em profundidade, Procura em largura e Procura generalizada
 - ▶ Notar que falamos de varrimento para árvores em memória e de procura para árvores conceptuais

Algoritmos de Procura (2)

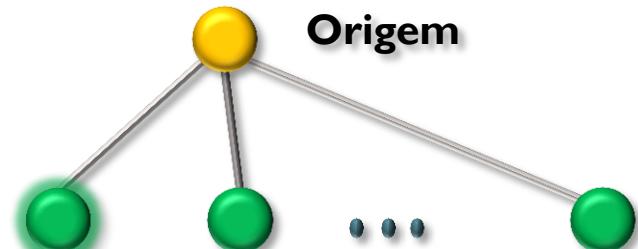
- ▶ Para qualquer um dos três procedimentos é suposto ser-se capaz de identificar todas as acções legais aplicáveis em cada um dos pontos da viagem
 - ▶ Produzida para uma lista, tabela, etc.
- ▶ Também é suposto ser-se capaz de identificar quais os caminhos já percorridos na procura
 - ▶ Isto é, quais as acções legais que já foram testadas
- ▶ No essencial, cada um dos três procedimentos de procura define a política de processamento da lista e/ou tabela que contém as acções legais

Algoritmos de Procura (3)

▶ Procura em Profundidade (DFS)

– *Depth First Search*)

- ▶ Para cada ponto escolher a primeira das acções legais ainda não usada
- ▶ Prosseguir da mesma forma em cada nível, enquanto for possível avançar

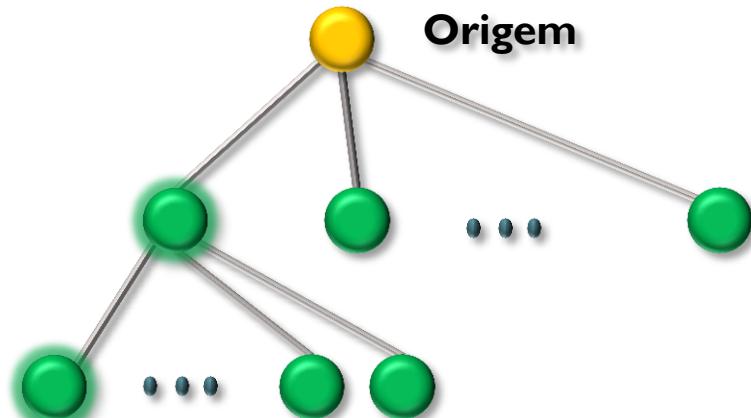


Algoritmos de Procura (3)

▶ Procura em Profundidade (DFS)

– *Depth First Search*)

- ▶ Para cada ponto escolher a primeira das acções legais ainda não usada
- ▶ Prosseguir da mesma forma em cada nível, enquanto for possível avançar

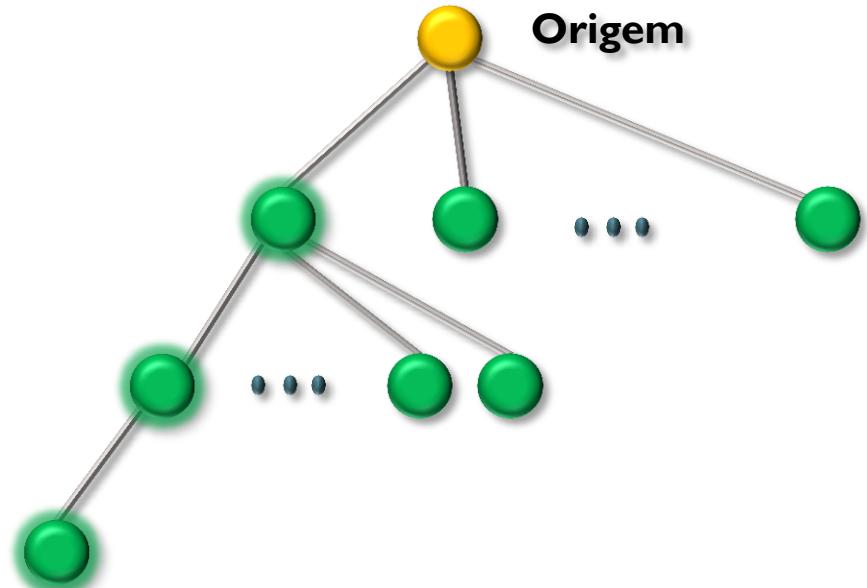


Algoritmos de Procura (3)

▶ Procura em Profundidade (DFS)

– *Depth First Search*)

- ▶ Para cada ponto escolher a primeira das acções legais ainda não usada
- ▶ Prosseguir da mesma forma em cada nível, enquanto for possível avançar

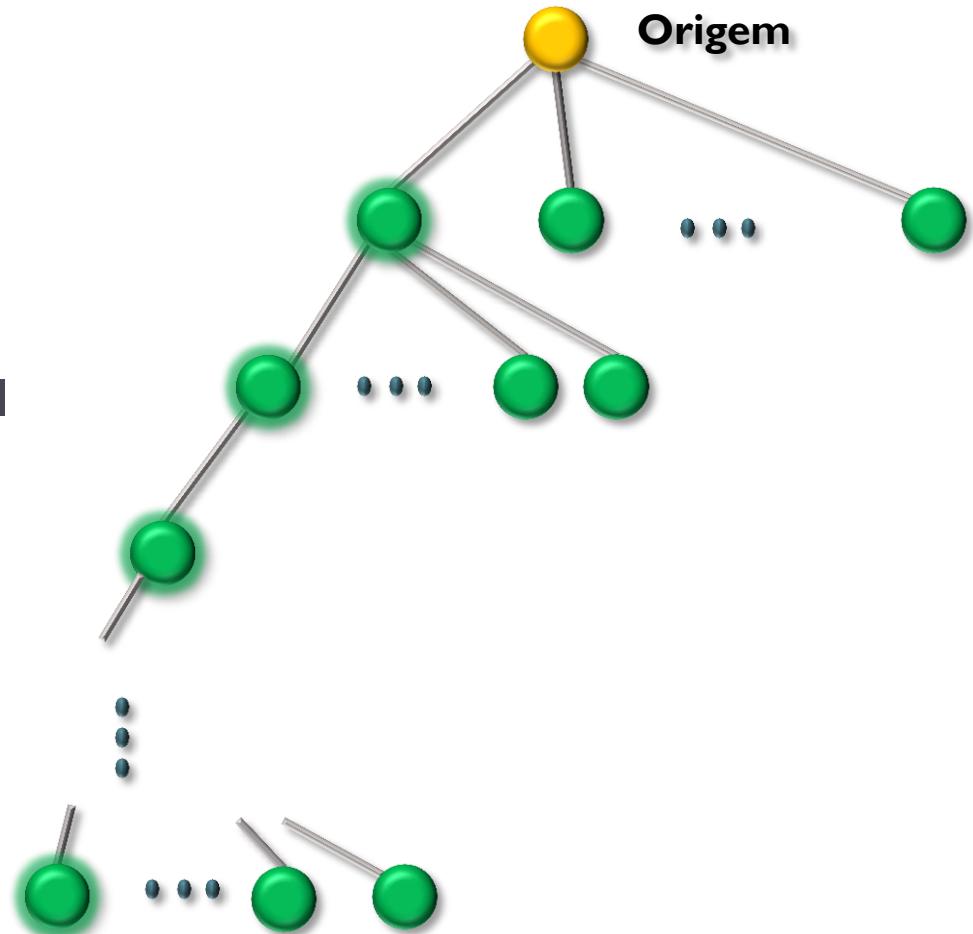


Algoritmos de Procura (3)

▶ Procura em Profundidade (DFS)

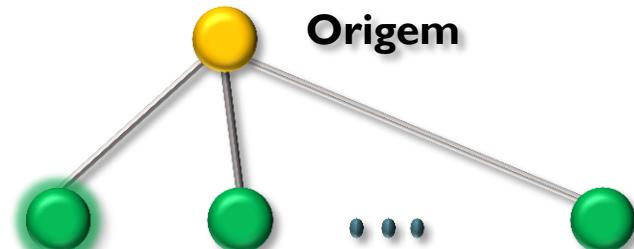
– *Depth First Search*)

- ▶ Para cada ponto escolher a primeira das acções legais ainda não usada
- ▶ Prosseguir da mesma forma em cada nível, enquanto for possível avançar



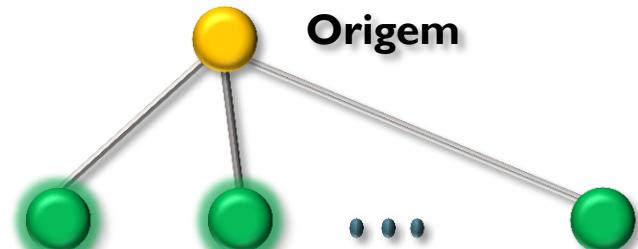
Algoritmos de Procura (4)

- ▶ Procura em Largura (BFS – *Breadth First Search*)
 - ▶ Para cada ponto inspecionar primeiro todas as acções legais disponíveis
 - ▶ Prosseguir para o nível seguinte apenas depois de esgotar todas as alternativas no presente nível



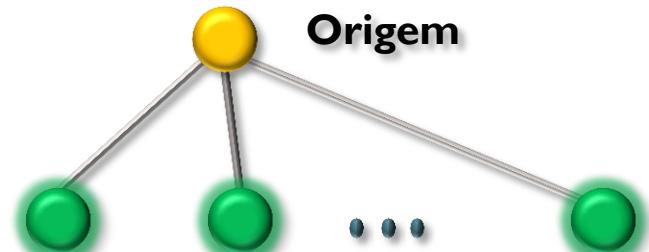
Algoritmos de Procura (4)

- ▶ Procura em Largura (BFS – *Breadth First Search*)
 - ▶ Para cada ponto inspecionar primeiro todas as acções legais disponíveis
 - ▶ Prosseguir para o nível seguinte apenas depois de esgotar todas as alternativas no presente nível



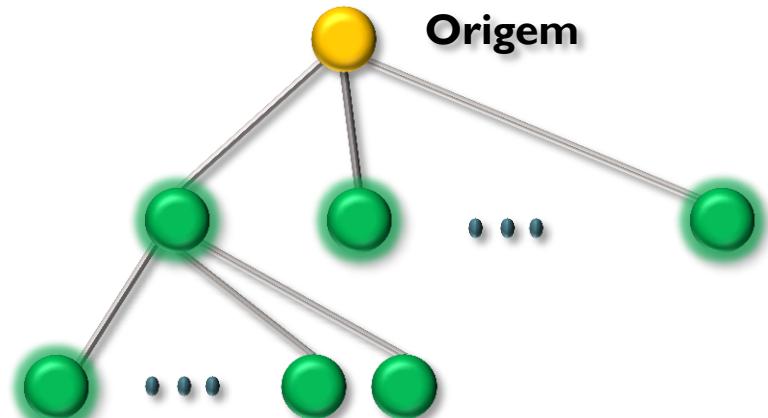
Algoritmos de Procura (4)

- ▶ Procura em Largura (BFS – *Breadth First Search*)
 - ▶ Para cada ponto inspecionar primeiro todas as acções legais disponíveis
 - ▶ Prosseguir para o nível seguinte apenas depois de esgotar todas as alternativas no presente nível



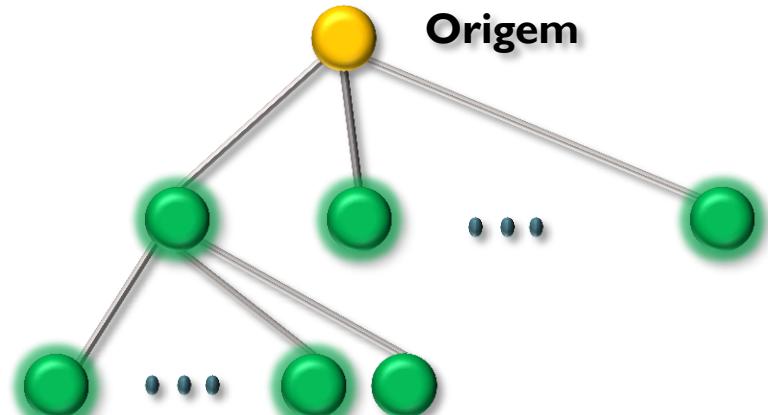
Algoritmos de Procura (4)

- ▶ Procura em Largura (BFS – *Breadth First Search*)
 - ▶ Para cada ponto inspecionar primeiro todas as acções legais disponíveis
 - ▶ Prosseguir para o nível seguinte apenas depois de esgotar todas as alternativas no presente nível



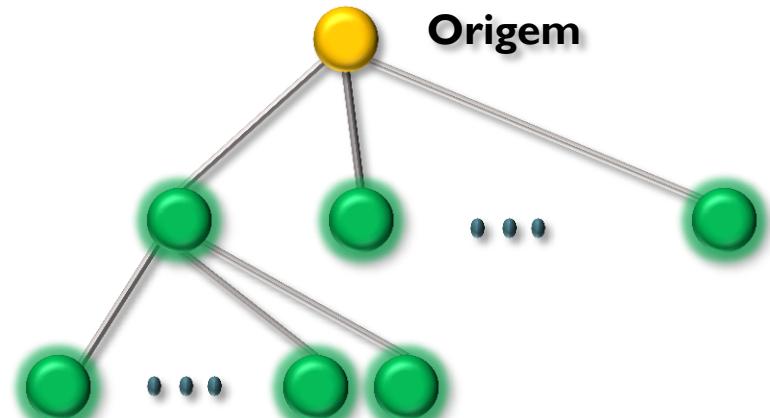
Algoritmos de Procura (4)

- ▶ Procura em Largura (BFS – *Breadth First Search*)
 - ▶ Para cada ponto inspecionar primeiro todas as acções legais disponíveis
 - ▶ Prosseguir para o nível seguinte apenas depois de esgotar todas as alternativas no presente nível



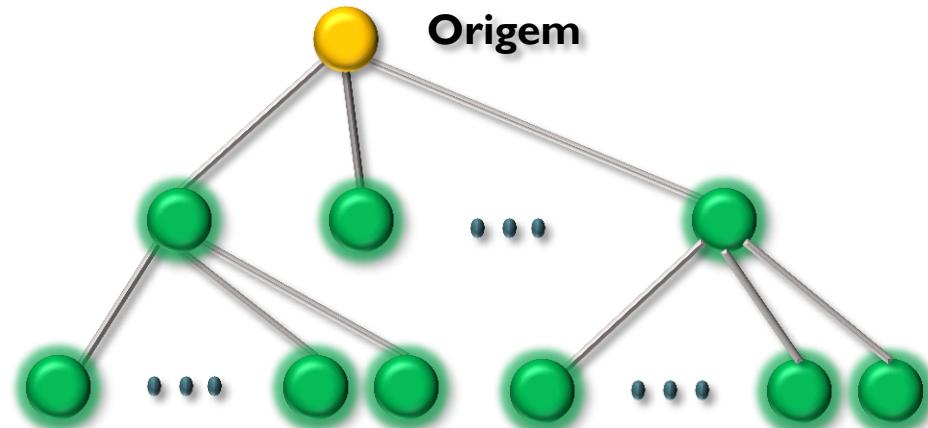
Algoritmos de Procura (4)

- ▶ Procura em Largura (BFS – *Breadth First Search*)
 - ▶ Para cada ponto inspecionar primeiro todas as acções legais disponíveis
 - ▶ Prosseguir para o nível seguinte apenas depois de esgotar todas as alternativas no presente nível



Algoritmos de Procura (4)

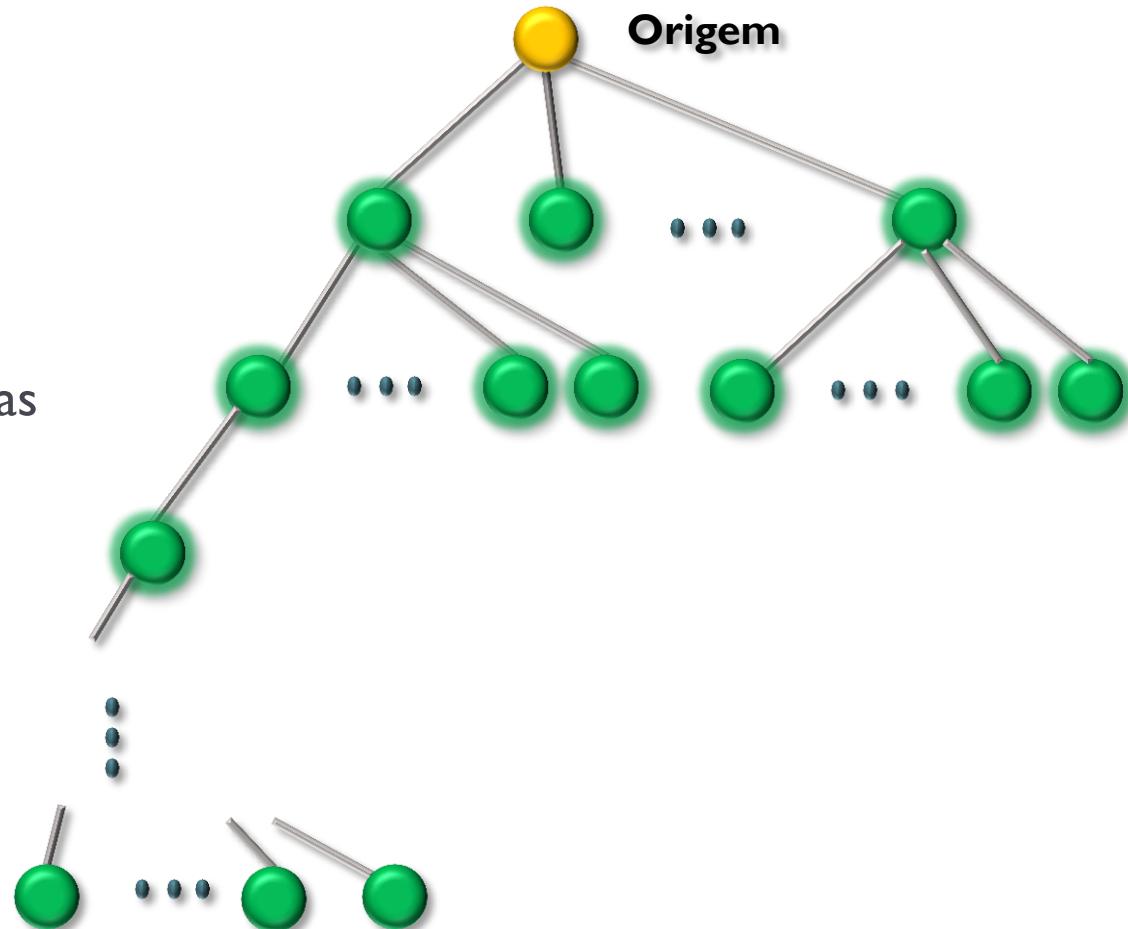
- ▶ Procura em Largura (BFS – *Breadth First Search*)
 - ▶ Para cada ponto inspecionar primeiro todas as acções legais disponíveis
 - ▶ Prosseguir para o nível seguinte apenas depois de esgotar todas as alternativas no presente nível



Algoritmos de Procura (4)

▶ Procura em Largura (BFS – *Breadth First Search*)

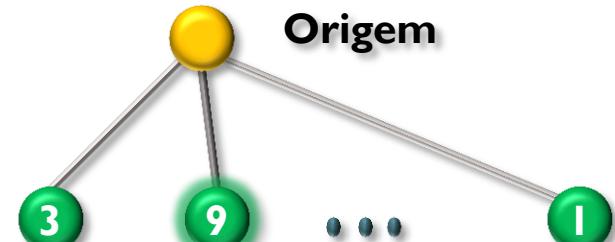
- ▶ Para cada ponto inspecionar primeiro todas as acções legais disponíveis
- ▶ Prosseguir para o nível seguinte apenas depois de esgotar todas as alternativas no presente nível



Algoritmos de Procura (5)

▶ Procura em Generalizada (GS – Generalised Search)

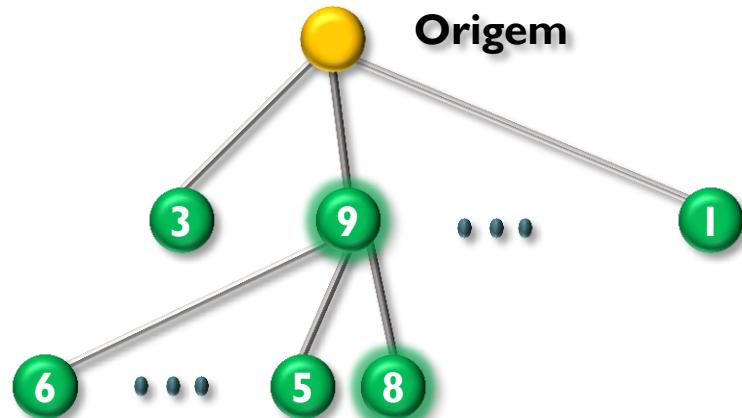
- ▶ Para cada ponto avaliar a qualidade de cada acção e/ou do local a que se chega mediante essa acção
- ▶ Tomar o caminho da acção que em cada instante possui e/ou promete o melhor valor



Algoritmos de Procura (5)

▶ Procura em Generalizada (GS – Generalised Search)

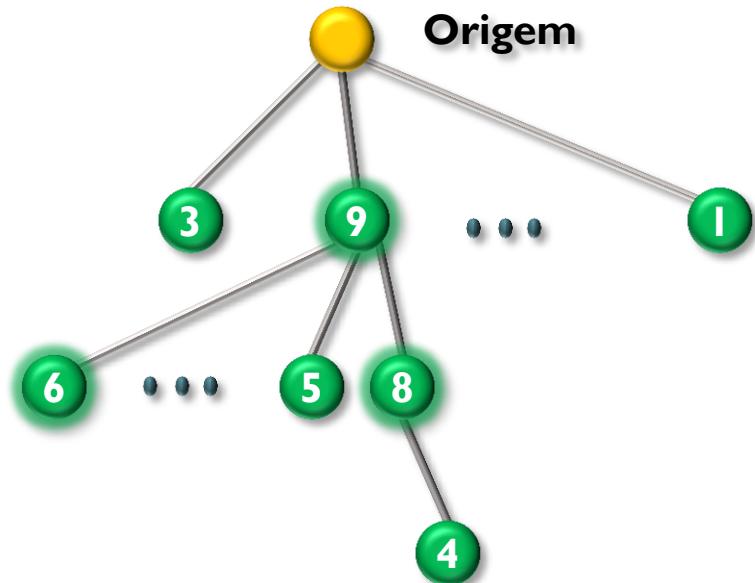
- ▶ Para cada ponto avaliar a qualidade de cada acção e/ou do local a que se chega mediante essa acção
- ▶ Tomar o caminho da acção que em cada instante possui e/ou promete o melhor valor



Algoritmos de Procura (5)

▶ Procura em Generalizada (GS – Generalised Search)

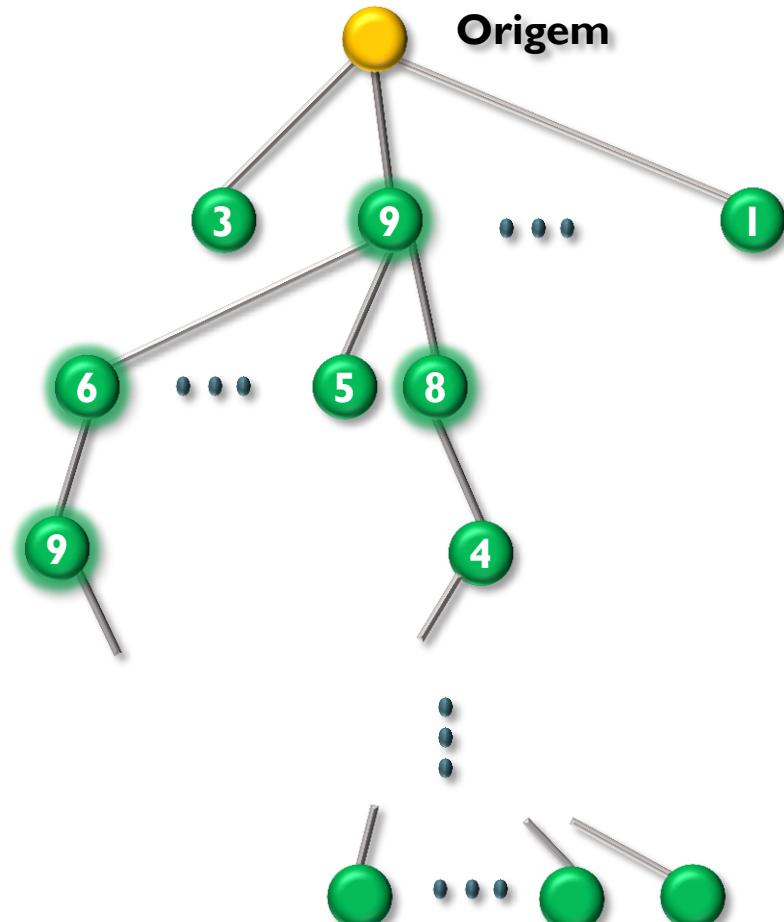
- ▶ Para cada ponto avaliar a qualidade de cada acção e/ou do local a que se chega mediante essa acção
- ▶ Tomar o caminho da acção que em cada instante possui e/ou promete o melhor valor



Algoritmos de Procura (5)

▶ Procura em Generalizada (GS – Generalised Search)

- ▶ Para cada ponto avaliar a qualidade de cada acção e/ou do local a que se chega mediante essa acção
- ▶ Tomar o caminho da acção que em cada instante possui e/ou promete o melhor valor



Algoritmos recursivos em árvores binárias (1)

- ▶ Os algoritmos de varrimento que vimos fornecem uma boa base para outros tipos de problemas que poderemos necessitar de resolver.
- ▶ Muitas tarefas em árvores binárias admitem soluções recursivas do tipo divide-and-conquer como as que vimos para varrimento.
- ▶ É frequentemente necessário calcular o valor de parâmetros estruturais de uma árvore, recebendo como argumento um ponteiro para essa árvore.

```
int count(link *h)
{
    if (h == NULL) return 0;
    return count(h->l) + count(h->r) +1;
}
```

- ▶ A função acima calcula o número de vértices existentes na árvore binária apontada por h.

Algoritmos recursivos em árvores binárias (2)

- ▶ A função abaixo calcula a altura de uma árvore binária.

```
int height(link *h)
{
    int u, v;
    if (h == NULL) return -1;
    u = height(h->l); v = height(h->r);
    if (u > v) return u+1; else return v+1;
}
```

- ▶ A altura de uma árvore binária, como vimos antes, define-se como sendo a maior das alturas das sub-árvores esquerda e direita mais um, para incluir a raíz nos cálculos.
- ▶ Nem todos os parâmetros estruturais admitem implementações tão simples como estes dois exemplos.
 - ▶ Por exemplo, calcular o comprimento de percurso de uma árvore é um pouco mais complexo.

Algoritmos recursivos em árvores binárias (3)

- ▶ A nossa primeira função para construção explícita de uma árvore binária está associada com a função max que estudámos anteriormente.
- ▶ O objectivo é construir um torneio: uma árvore binária em que cada item em cada vértice interno é uma cópia do maior dos itens dos seus dois filhos.
 - ▶ Os itens nas folhas é que constituem a informação de interesse;
 - ▶ O resto da árvore permite-nos calcular o maior dos items de forma eficiente.

Construção de uma árvore

```
typedef struct node link;
struct node {Item item; link *l; link *r};

link *NEW(Item item, link *l, link *r)
{ link *x = (link *) malloc(sizeof(link));
  x->Item = item; x->l = l; x->r = r;
  return x;
}

link *max(Item a[], int l, int r)
{ int m = (l+r)/2; Item u, v;
  link *x = NEW(a[m], NULL, NULL);
  if (l == r) return x;
  x->l = max(a, l, m);
  x->r = max(a, m+1, r);
  u = x->l->Item; v = x->r->Item;
  if (u > v)
    x->Item = u; else x->Item = v;
  return x;
}
```

Exemplo (1)

- ▶ Considere-se a tabela $a = [3 \ 4 \ 1 \ 8 \ 9 \ 2 \ 7 \ 6 \ 5]$ e a chamada à função $\max(a, 0, 8)$.

$\max(a, 0, 8)$

$\max(a, 0, 4)$

$\max(a, 0, 2)$

$\max(a, 0, 1)$

$\max(a, 0, 0) \rightarrow 3$

$\max(a, 1, 1) \rightarrow 4$

$\max(a, 2, 2) \rightarrow 1$

$\max(a, 3, 4)$

...

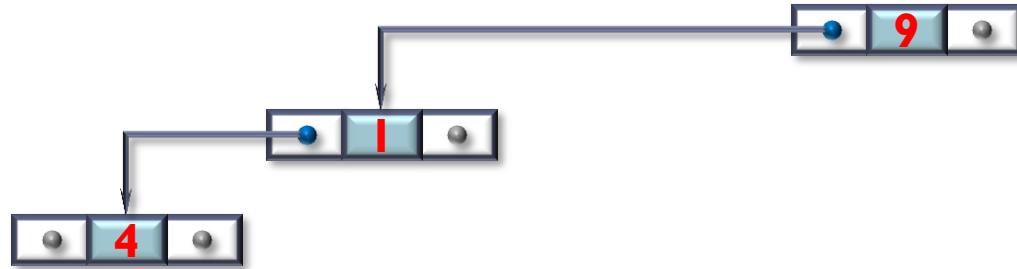
Exemplo (2)



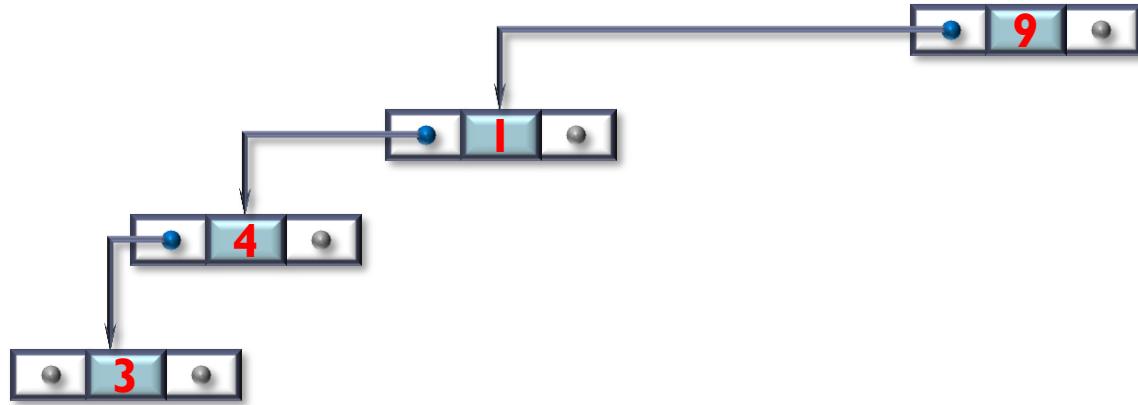
Exemplo (2)



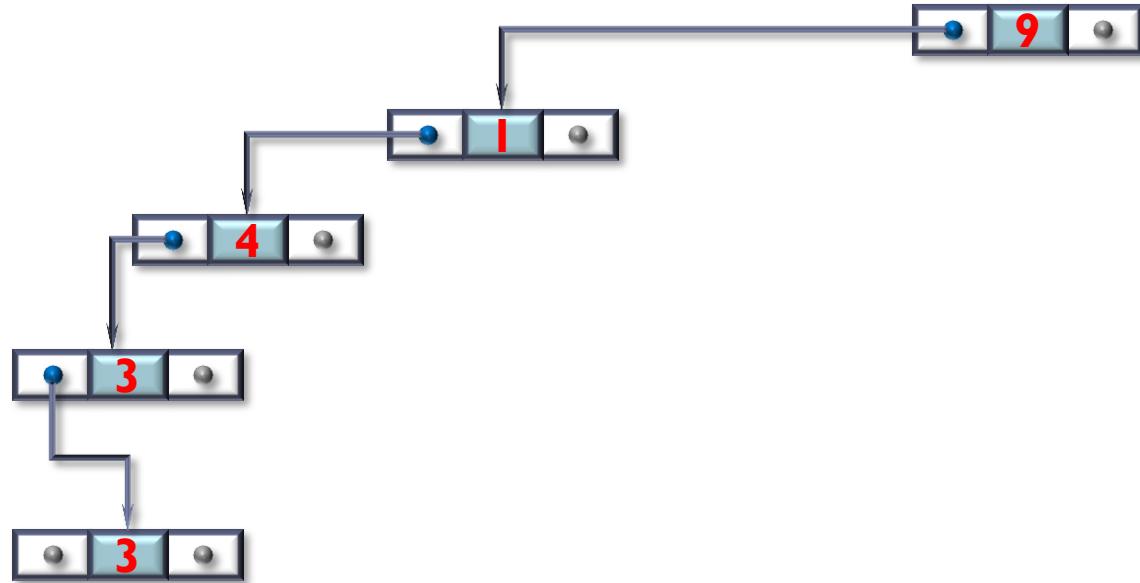
Exemplo (2)



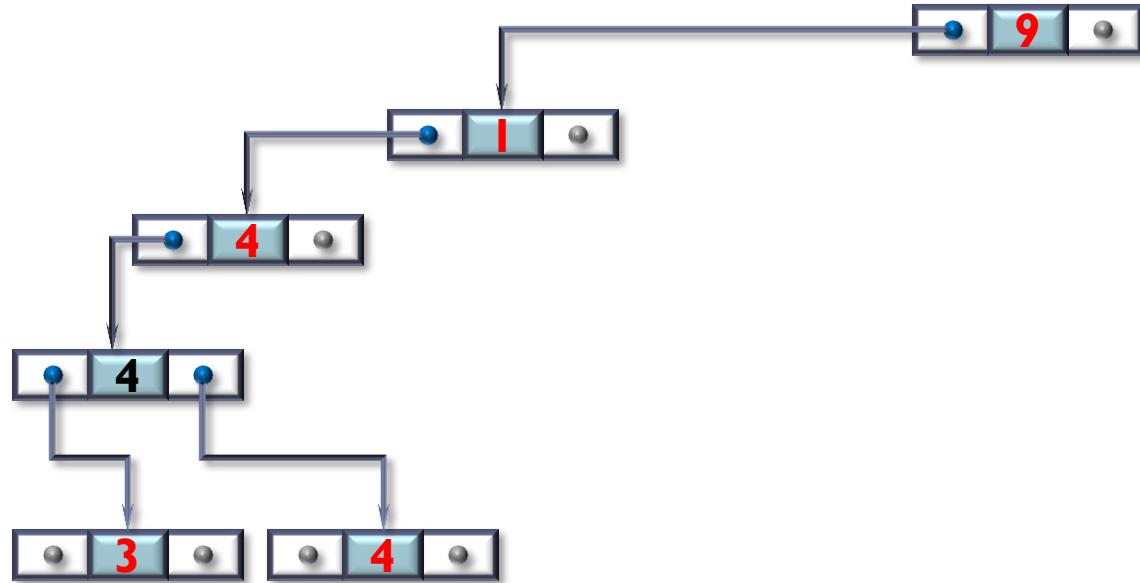
Exemplo (2)



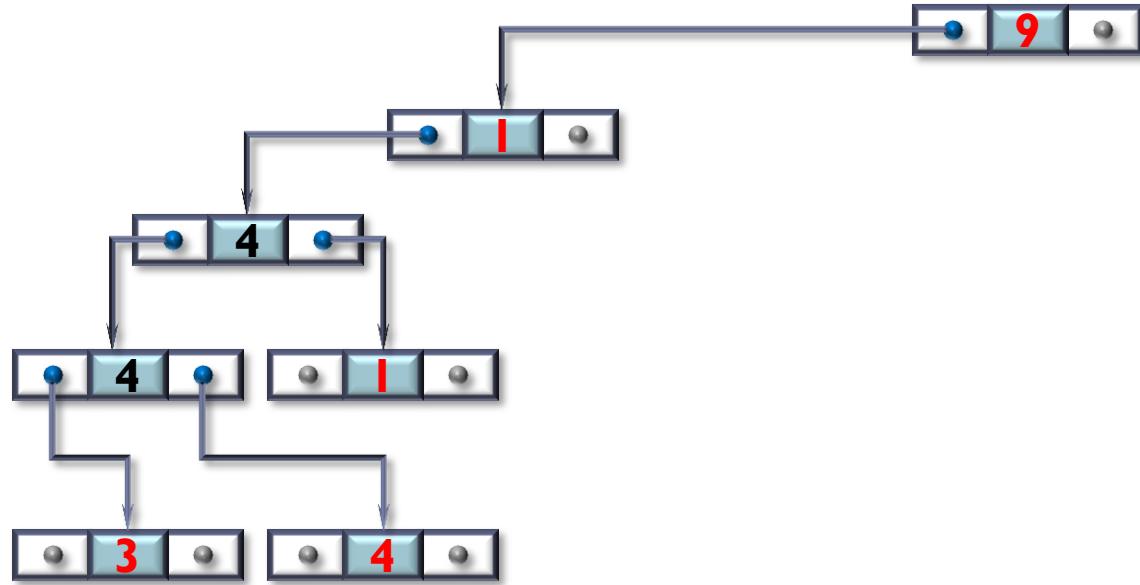
Exemplo (2)



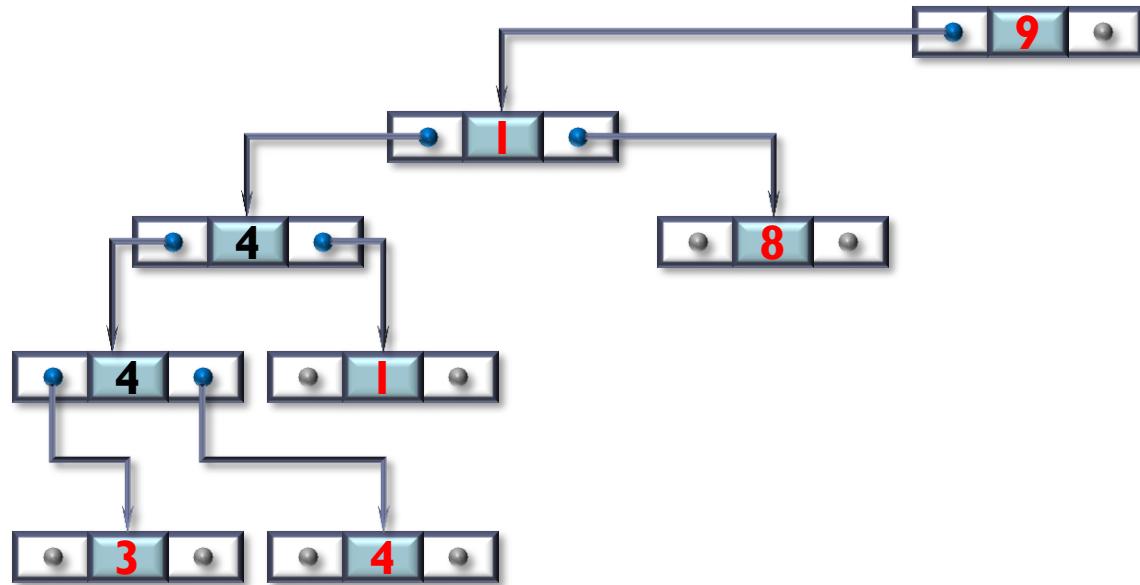
Exemplo (2)



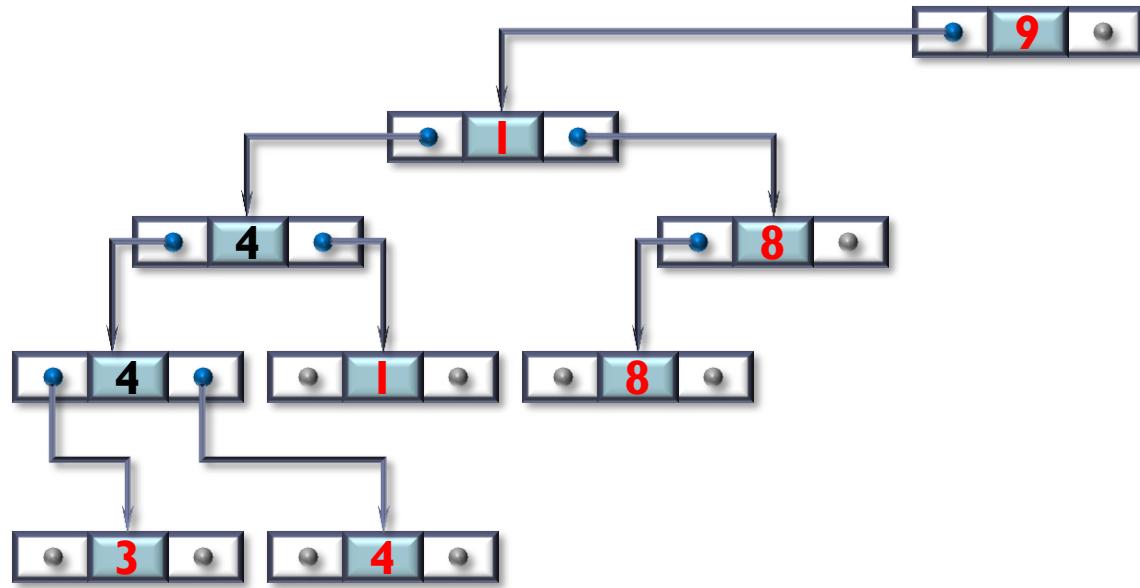
Exemplo (2)



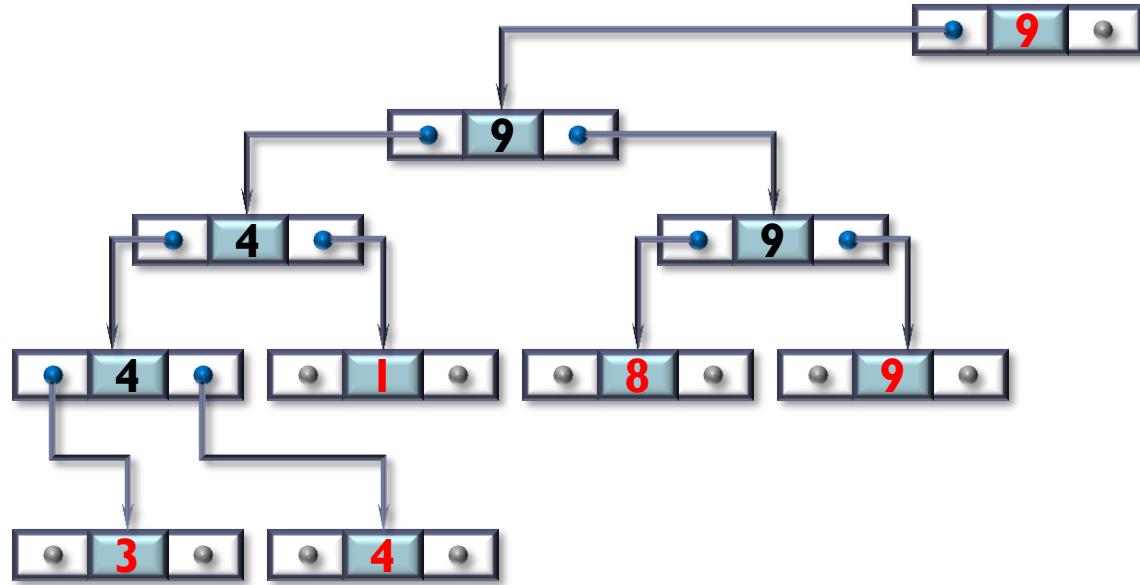
Exemplo (2)



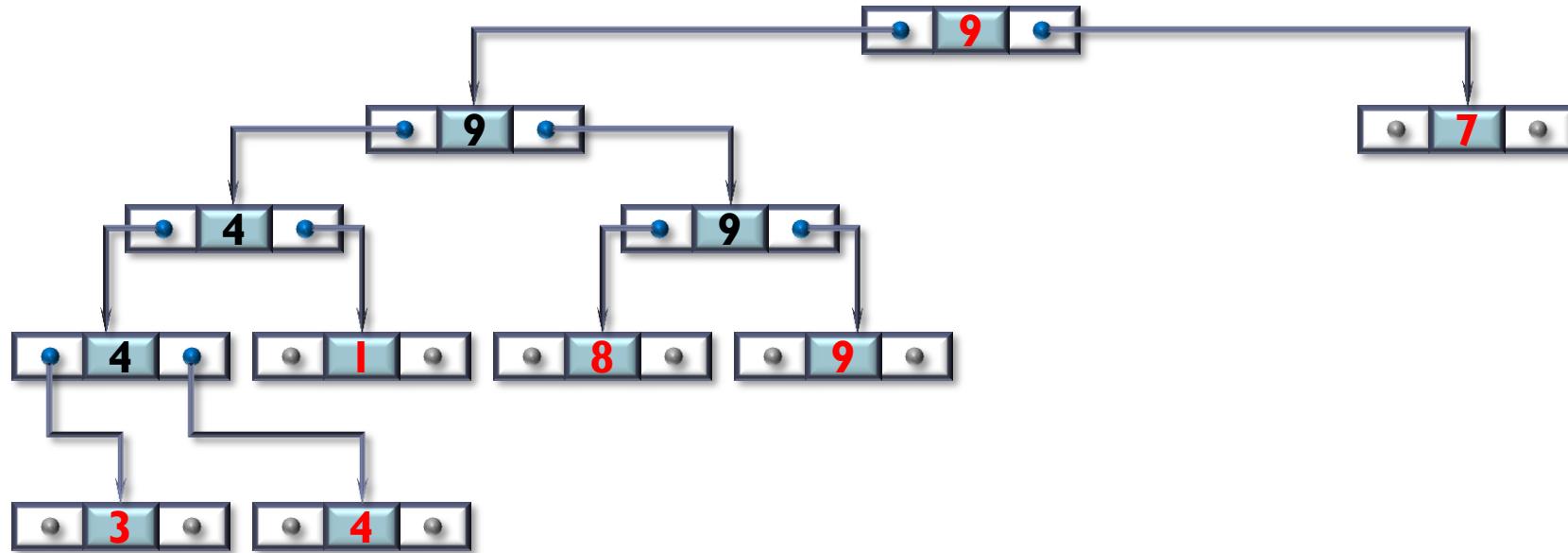
Exemplo (2)



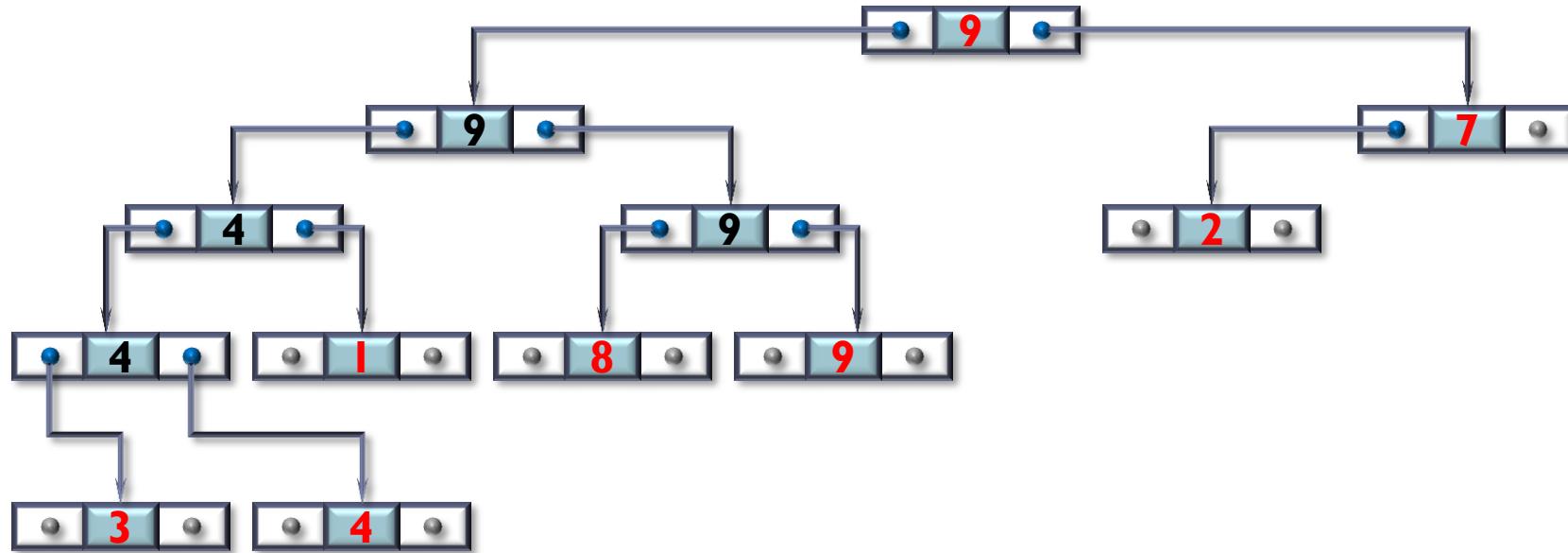
Exemplo (2)



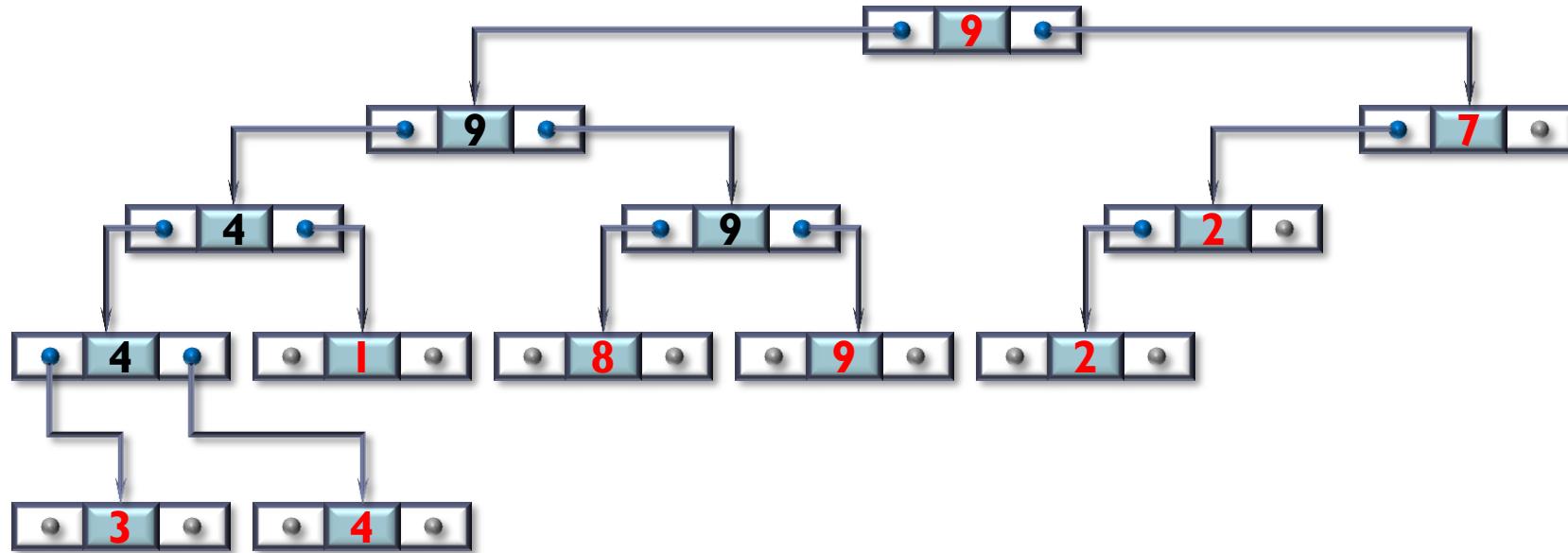
Exemplo (2)



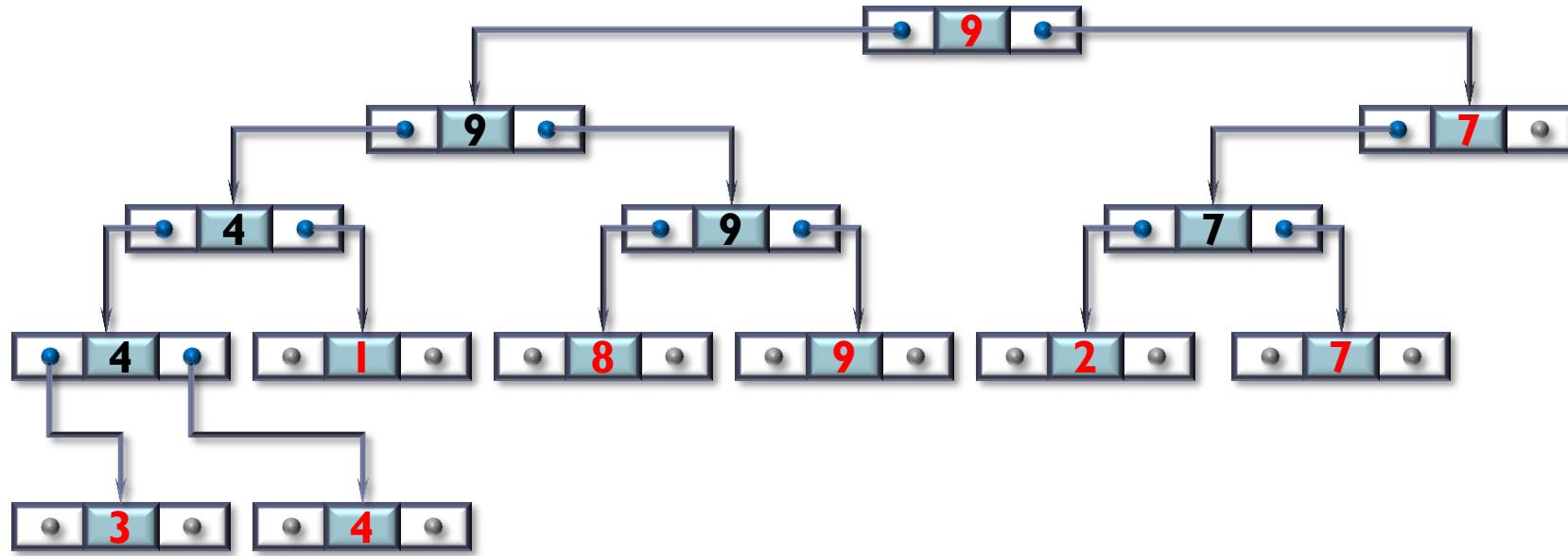
Exemplo (2)



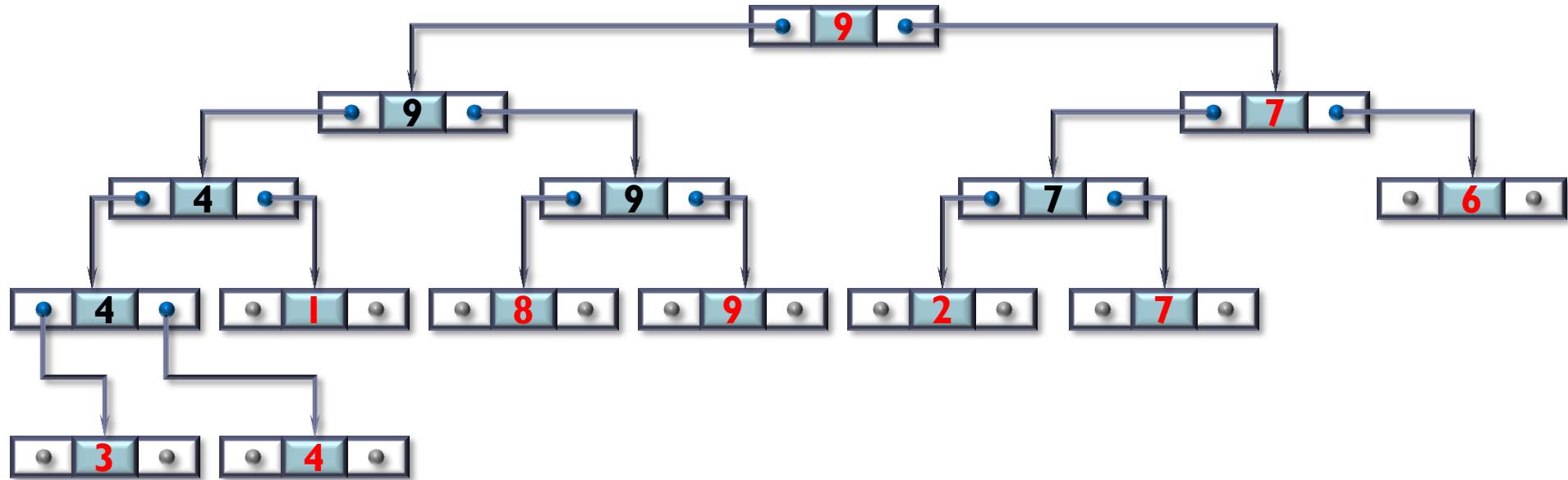
Exemplo (2)



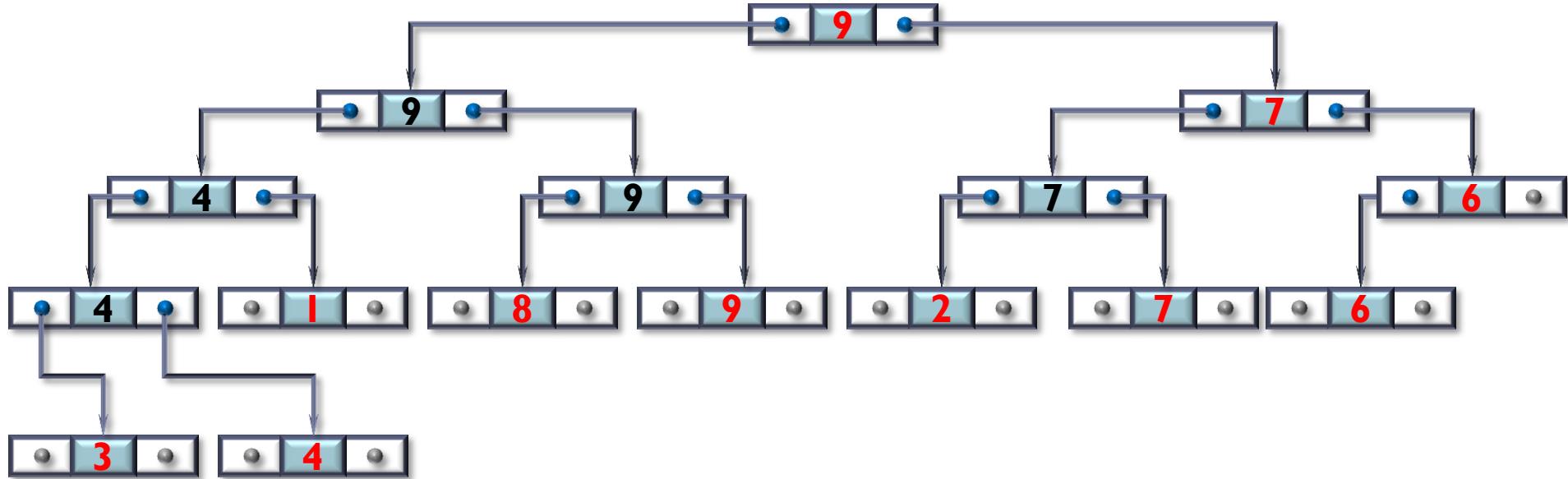
Exemplo (2)



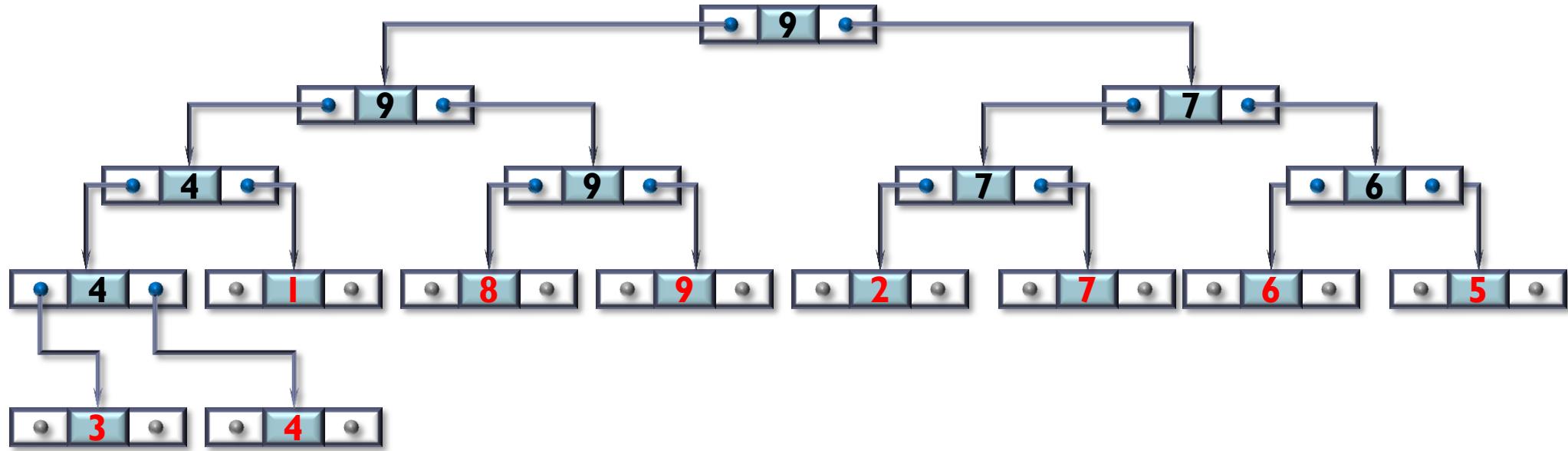
Exemplo (2)



Exemplo (2)



Exemplo (2)



Construção de árvores para notação pré-fixada (1)

- ▶ Dada uma sequência de caracteres compondo uma expressão aritmética em notação pré-fixada, é possível construir uma árvore binária que a represente.
 - ▶ Exemplo: * + a * * b c + d e f
- ▶ Vamos criar uma nova árvore para cada caracter na expressão:
 - ▶ Vértices correspondentes a operadores possuem ponteiros para os operandos;
 - ▶ Os vértices folha conterão as variáveis, ou constantes, que são entradas para a expressão aritmética.
- ▶ Programas que executam traduções como compiladores frequentemente usam este tipo de representação para os programas.

Construção de árvores para notação pré-fixada (2)

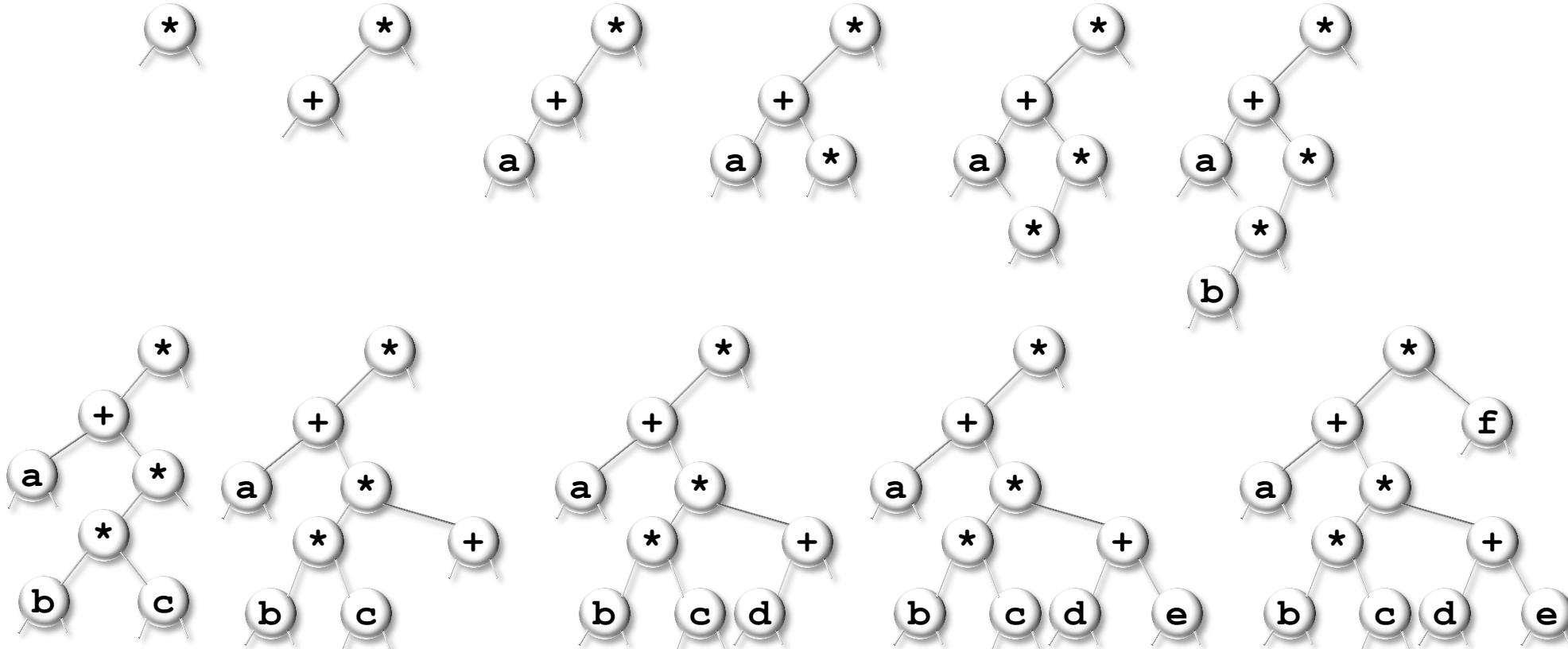
```
char *a; int i;
typedef struct Tnode link;
struct Tnode {char token; link *l, *r;};

link * NEW(char token, link *l, link *r)
{
    link *x = (link *) malloc(sizeof(link));
    x->token = token; x->l = l; x->r = r;
    return x;
}

link * parse()
{
    char t = a[i++];
    link *x = NEW(t, NULL, NULL);
    if ((t == '+') || (t == '*'))
        { x->l = parse(); x->r = parse(); }
    return x;
}
```

Exemplo

- ▶ Seja $a = [* + a * * b c + d e f]$

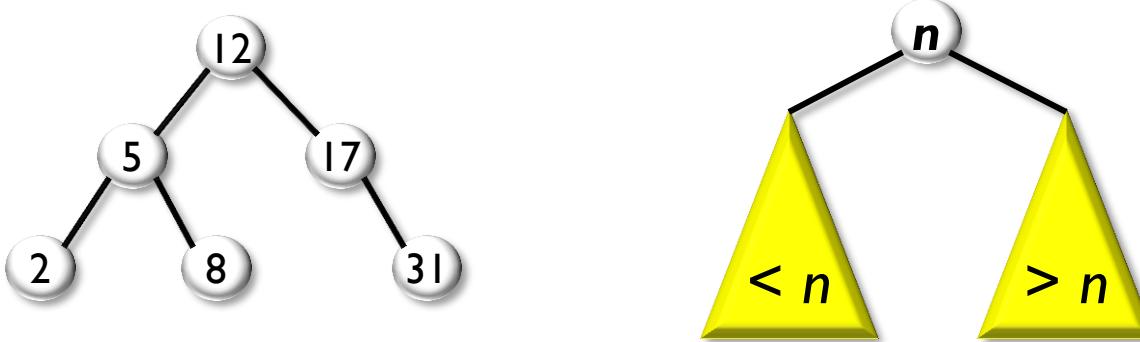


Síntese da Aula 3

- ▶ Varrimento em árvores binárias
 - ▶ Pré-fixado, In-fixado, Pós-fixado
 - ▶ Exemplos;
 - ▶ Largura.
- ▶ Outros varrimentos e outras árvores
- ▶ Árvores e algoritmos de procura
- ▶ Algoritmos de procura
- ▶ Algoritmos recursivos em árvores binárias
- ▶ Construção de árvores binárias
 - ▶ Torneio e parsing;
 - ▶ Exemplos.

Árvores Ordenadas (1)

- ▶ Exemplo de uma árvore ordenada de pesquisa binária¹

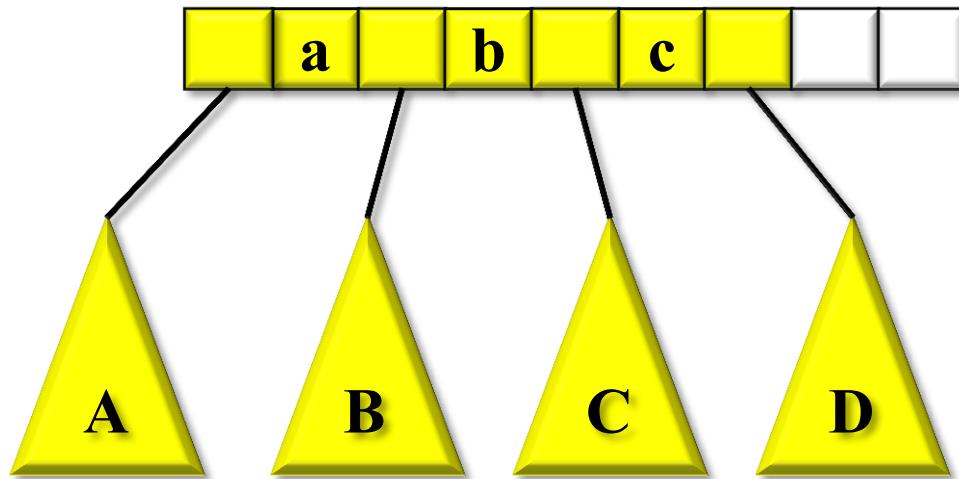


- ▶ Sub-árvore direita (esquerda) armazena números maiores (menores) que o vértice raiz. Se os dados forem cadeias de caracteres, usa-se a ordem lexicográfica ($cf > cb$ e $cf > az$).
 - ▶ Vantagens: pesquisa mais rápida – $\mathcal{O}(\log N)$;
 - ▶ Inconvenientes: pode degenerar em lista com pesquisa – $\mathcal{O}(N)$, a resolução deste problema torna inserção mais complicada.

¹ BST - Binary Search Tree

Árvores Ordenadas (2)

- As árvores BB (Balanced Binary) possuem, em cada nó, uma tabela para as sub-árvore. São usadas em Bases de Dados.



- $A < a < B < b < C < c < D$;
- Todos os caminhos da raíz para as folhas possuem o mesmo comprimento.

Árvores – Inserção (1)

- ▶ Sendo as árvores recursivas, as funções de manipulação podem ser recursivas: a instrução recursiva é aplicada nos nós intermédios e a instrução terminal nas folhas.
- ▶ As árvores são, usualmente, construídas de forma descendente (top-down).
- ▶ Nos algoritmos de inserção consideramos o item um inteiro (**typedef int Item;**) e uma árvore BST.
 - ▶ Se a informação a armazenar for uma estrutura, um dos campos designado **chave** (key) tem de ser tipo ordenado.
 - ▶ Por exemplo, no bilhete de identidade, a chave é o número do BI.

Árvores – Inserção (2)

▶ Inserção directa

- ▶ A inserção efectuada em dois casos, conforme situação da árvore:
 - ▶ vazia: insere directamente novo dado;
 - ▶ não vazia: procura, em primeiro lugar, o nó onde inserir o novo dado

```
link *insert(int i, link *tree)
{
    link *upper = tree,  *child = tree,  *leaf;
    if (tree == NULL)           /* árvore vazia */
    {
        tree = (link *) malloc(sizeof(link));
        tree->data = i; /* insere dado */
        tree->left = tree->right = NULL; /* sub-árvores vazias */
        return tree;
    }
    /* continua */
}
```

Árvores – Inserção (3)

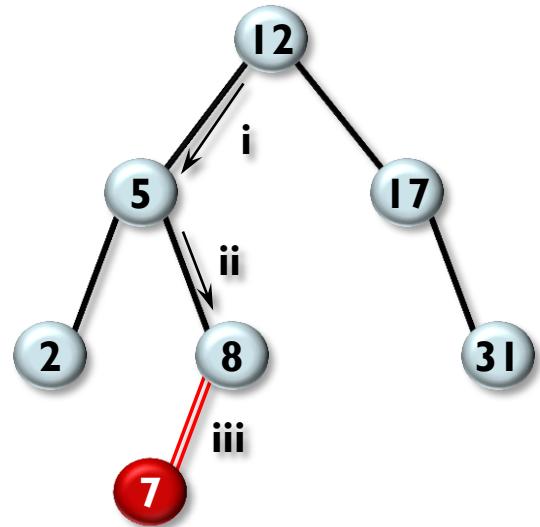
```
/* continuação */

while (child != NULL) {                                /* procura nó onde inserir dado */
    upper = child;
    if (child->data == i) return tree;                  /* já existe! */
    child = (child->data > i) ? child->left : child->right;
}
                                /* cria e inicializa nova folha */
leaf = (link *) malloc(sizeof(link));
leaf->data = i;
leaf->left = leaf->right = NULL;
                                /* upper passa a nó intermédio e aponta para nova folha */
if (upper->data > i) upper->left = leaf;
else upper->right = leaf;
return tree;
}
/* rotina insert retorna nova árvore (i.e. ponteiro para raiz), porque a
inicial foi alterada */
```

Árvores – Inserção (4)

- ▶ Exemplo: inserção de 7 na árvore do slide 96

- i) $7 < 12$: insere na sub-árvore esquerda
- ii) $7 > 5$: insere na sub-árvore direita
- iii) 8 é folha:
 - cria 7 nova folha.
 - $7 < 8$: a nova folha é sub-árvore esquerda



O algoritmo degenera numa lista, se os dados inseridos forem sucessivamente crescentes (ou sucessivamente decrescentes).

Árvores – Inserção (5)

- ▶ A complexidade da inserção depende do tipo de árvore.
- ▶ Árvore degenerada (lista): $\mathcal{O}(N)$
- ▶ Árvore perfeitamente balanceada: complexidade determinada pela pesquisa do local de inserção $\lfloor \log_2 N \rfloor$, ou seja, $\mathcal{O}(\log N)$
- ▶ Árvore de inserção aleatória: considerando que a altura de cada sub-árvore possui uma distribuição uniforme $1 \dots N$, tem-se que a pesquisa do local de inserção custa $\ln N$, ou seja, $\mathcal{O}(\log N)$
- ▶ Estratégia de prova: simplesmente, efectuar os cálculos!
- ▶ O custo é I (visita à raiz) somado ao custo da visita à sub-árvore:
 - ▶ Esta pode ter entre 0 e $N-1$ nós, com distribuição uniforme
 - ▶ $C_N = I + I/N * \sum C_{k-1}$ $1 \leq k \leq N$, para $N \geq 2$
 - ▶ $C_I = I$

Árvores – Inserção (6)

- Para eliminar Σ , multiplica-se ambos os lados por N , e subtrai-se a fórmula para $N-1$ (nota: k varia até N nos dois Σ)

$$NC_N - (N-1)C_{N-1} = N + \sum C_{k-1} - (N-1 + \sum C_{k-2}) = 1 + C_{N-1}$$

$$NC_N = NC_{N-1} + 1$$

$$C_N = C_{N-1} + 1/N$$

- Por substituição telescópica

$$C_N = C_1 + \frac{1}{2} + \dots + 1/(N-1) + 1/N = \sum (1/k)$$

- Trata-se da série harmónica, pelo que

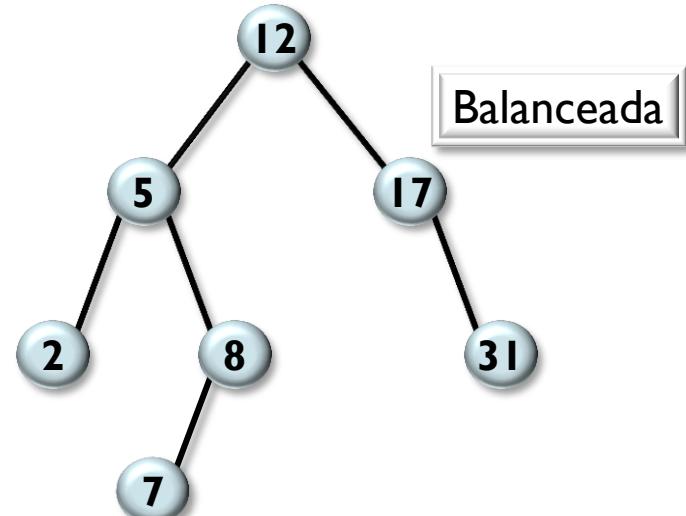
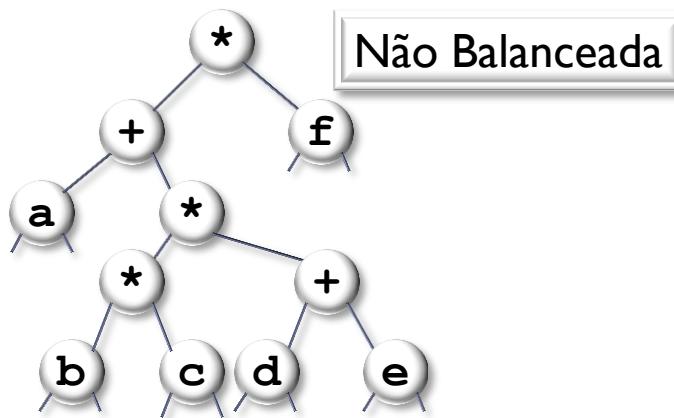
$$C_N \approx \ln N + \gamma + 1/(2N)$$

QED

- Conclusão: o custo da inserção é mínimo se a árvore for balanceada.
- O problema reside na maior dificuldade em manter a árvore balanceada (como se vai ver nos acetatos seguintes).

Árvores balanceadas AVL (1)

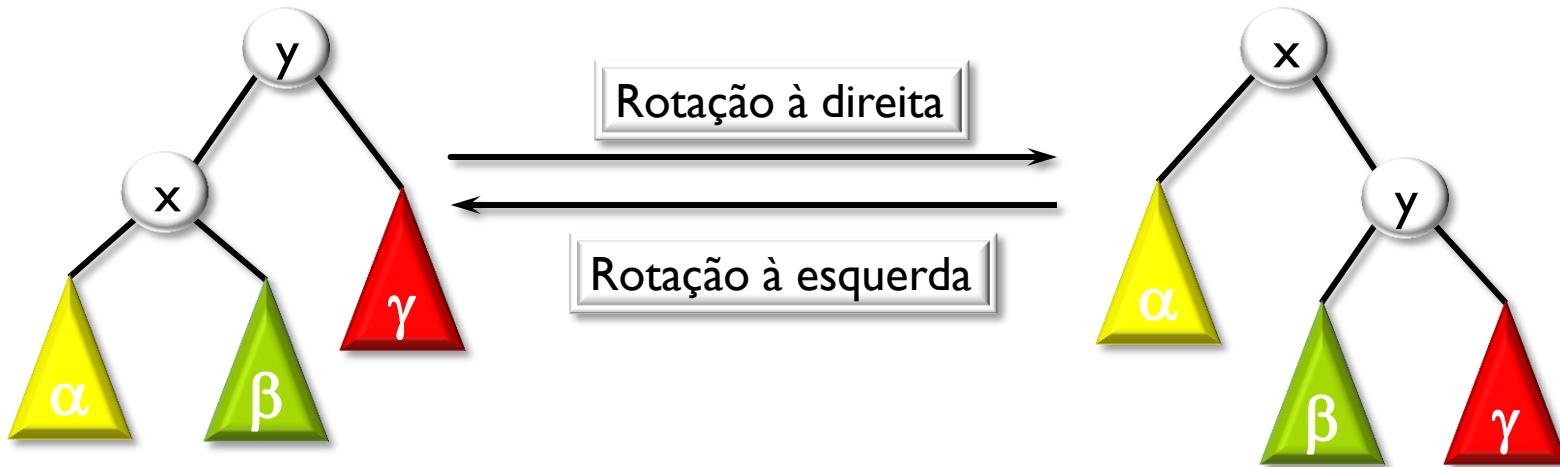
- ▶ Definição – Uma árvore diz-se balanceada AVL⁽²⁾ ou balanceada em altura, se só se em todos os nós a diferença entre as alturas das subárvore for igual ou inferior a 1.



- ▶ Para manter a árvore balanceada, respeitando a ordem, depois da inserção pode ser necessário rodar a configuração de nós (rotação simples e/ou rotação dupla).
- ▶ (2) Adel'son-Vel'skii e Landis

Árvores balanceadas AVL (2)

- Operações de rotação são classificadas de acordo com o sentido (à direita e à esquerda).



- Quando a diferença de alturas for igual a 2, a operação de rotação diminui o nível da folha mais afastada da raiz, mantendo a ordenação da árvore.

Árvores balanceadas AVL (3)

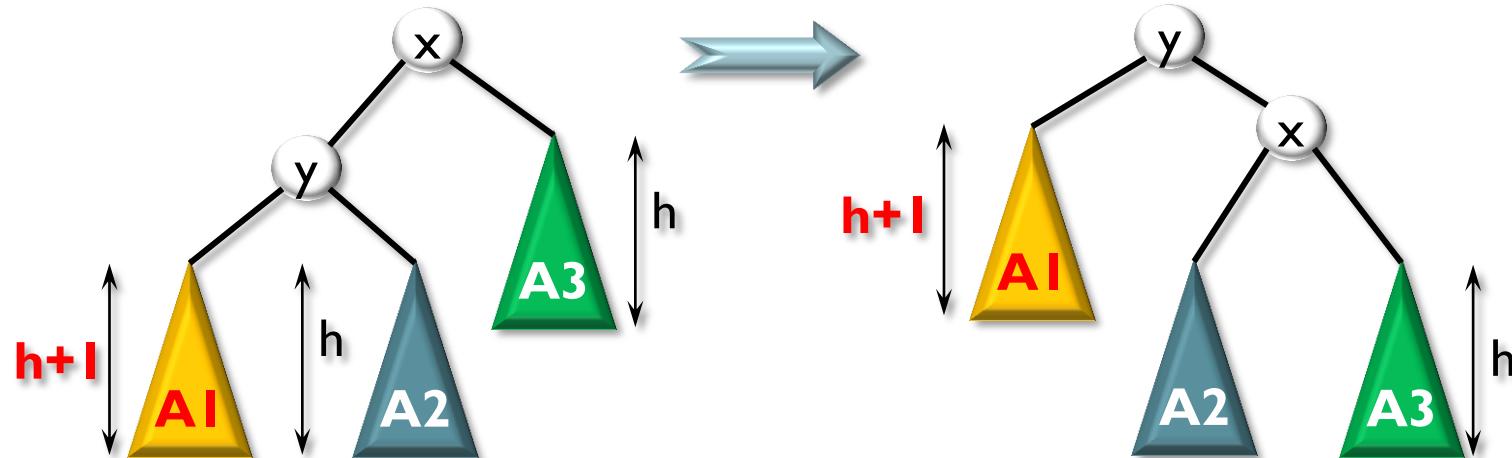
- ▶ A função de rotação apenas actualiza ponteiros (β é a única sub-árvore a mudar de ascendente).
- ▶ Assim, a rotação tem complexidade $\mathcal{O}(1)$.

```
link *rotate_right(link * tree)
{
    link *x, *y, *beta;
    /* inalterada se árvore vazia, ou sub-árvore esquerda vazia */
    if (tree == NULL) return tree;
    else if (tree->left == NULL) return tree;
    /* salva ponteiros */
    y = tree; x = tree->left; beta = x->right;
    /* actualiza ligações */
    x->right = y; y->left = beta;
    return x;
}
```

Árvores balanceadas AVL (4)

- ▶ Há dois pares de casos em que se torna necessário rodar subárvore
- ▶ Estes dependem do local onde um vértice acaba por estacionar no final da inserção (a vermelho identifica-se onde entrou o vértice inserido)

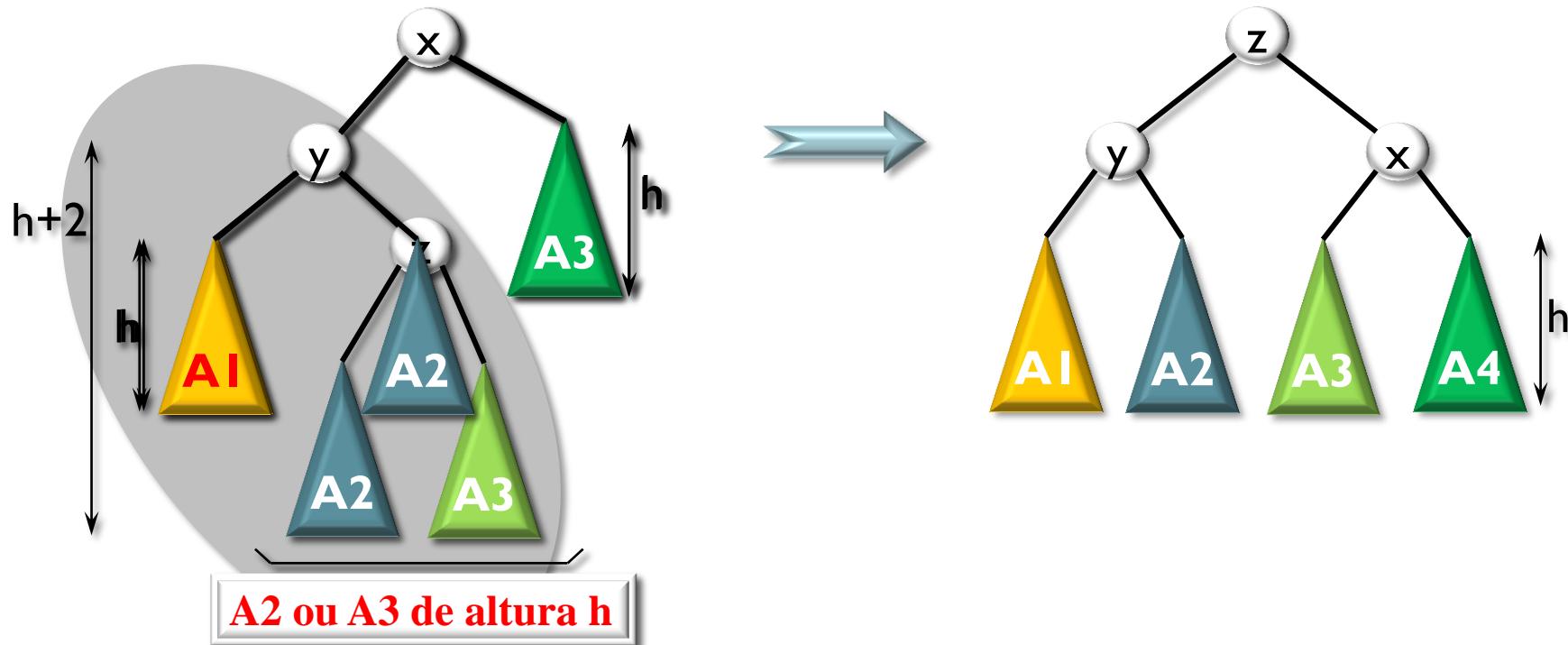
I: Rotação simples (à direita)



Árvores balanceadas AVL (5)

2: Rotação dupla (à direita)

- Consiste numa rotação simples à esquerda da sub-árvore esquerda seguida de uma rotação simples à direita da árvore.



Árvores balanceadas AVL (6)

▶ Código de inserção AVL

```
/* função que calcula a altura de uma árvore */
int height(link *tree)
{
    int hl, hr;

    if (tree == NULL) return 0;
    if (tree->left == NULL && tree->right == NULL)
        return 1;
    hl = height(tree->left); hr = height(tree->right);
    return ((hl>hr) ? 1+hl : 1+hr);
}
```

Árvores balanceadas AVL (7)

```
/* função que realiza a inserção */
link *insert(int i, link *tree) {
    /* tree é ponteiro para árvore: a alteração */
    /* é feita directamente na árvore que é dada */
    /* como parâmetro */
    int h1, h2, h3;

    if (tree == NULL) {           /* árvore vazia */
        tree = (link*) malloc(sizeof(link));
        tree->data = i;
        tree->left = tree->right = NULL;
        return tree;
    }

    /* continua */
}
```

Árvores balanceadas AVL (8)

```
/* continuaçāo */

if (i == tree->data) return tree; /* já inserido */
if (i < tree->data){
    /* insere na sub-árvore esquerda */
    tree->left = insert(i, tree->left);
    h1 = height(tree->left->left);
    h2 = height(tree->left->right);
    h3 = height(tree->right);
    if (h1 > h3)
        /* Caso 1, rotação simples à direita */
        tree = rotate_right(tree);
    if (h2 > h3) {
        /* Caso 2, rotação dupla à direita */
        tree->left = rotate_left(tree->left);
        tree = rotate_right(tree);
    }
}

/* continua */
```

Árvores balanceadas AVL (9)

```
/* continuação */

else {
    /* insere na sub-árvore direita */
    tree->right = insert(i, tree->right);
    h1 = height(tree->right->right);
    h2 = height(tree->right->left);
    h3 = height(tree->left);
    if (h1 > h3)
        /* Caso 1, rotação simples à esquerda */
        tree = rotate_left(tree);
    if (h2 > h3) {
        /* Caso 2, rotação dupla à esquerda */
        tree->right = rotate_right(tree->right);
        tree = rotate_left(tree);
    }
}

return tree;
}
```

Árvores balanceadas AVL (10)

- ▶ O algoritmo AVL não garante distribuição total pelos níveis intermédios (por exemplo, na série $S_0 + (-1)^n n$, a sub-árvore esquerda só deriva à esquerda e a sub-árvore direita só deriva à direita).
- ▶ O algoritmo de inserção listado é pesado devido à determinação das alturas. Uma versão mais eficiente (embora mais complexa e ocupando mais memória), guarda em cada nó o balanceamento das sub-árvores (maior à esquerda, iguais ou maior à direita).

Árvores Ordenadas - Combinação (1)

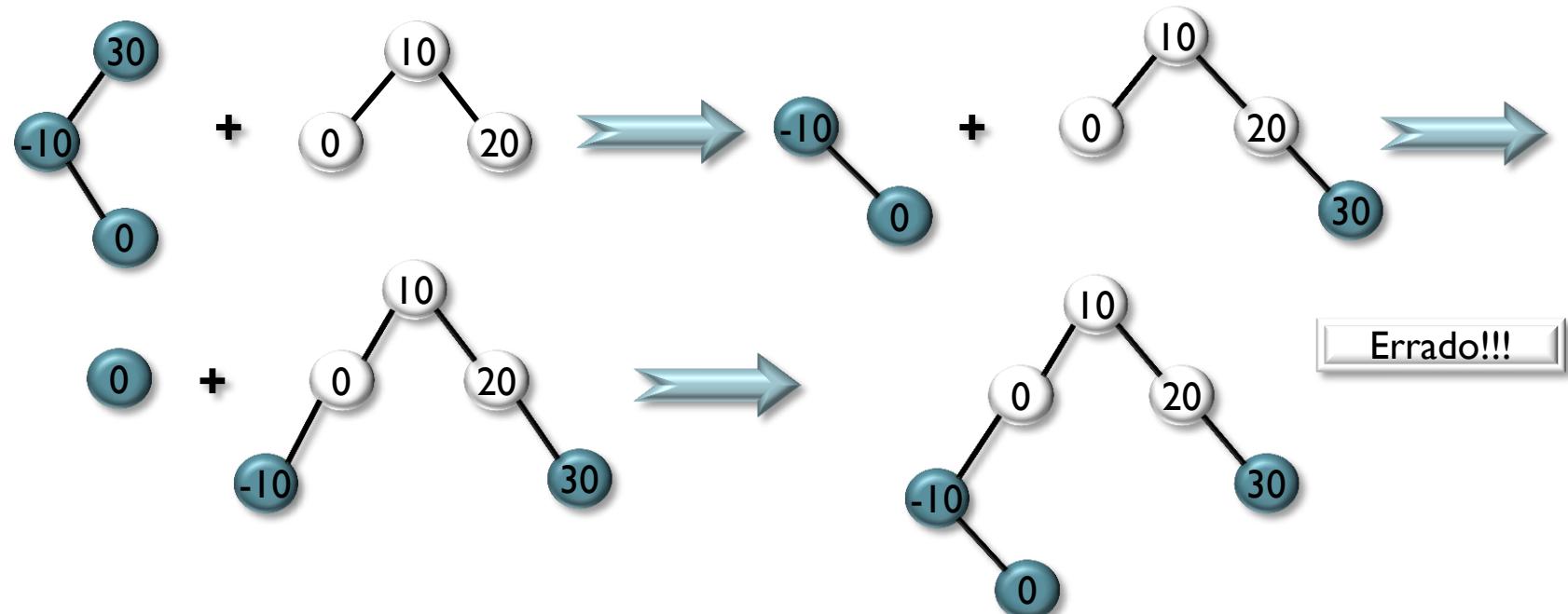
- ▶ Combinação permite juntar, numa única árvore, duas árvores (cada uma com dados parciais)
 - ▶ Se uma das árvores é nula, a combinação é apenas a outra árvore.
 - ▶ Senão, insere-se os elementos da primeira na segunda.

```
link *join(link *a, link *b)
{
    if (a == NULL) return b;
    if (b == NULL) return a;

    b = insert(a->data, b);
    b = join(a->left, b);
    /* Sedgewick tem join(a->left,b->left); */
    b = join(a->right, b);
    /* Sedgewick tem join(a->right,b->right); */
    free(a);
    /* Notar que se pode libertar a raiz a, */
    /* porque os filhos não se perderam */
    return b;
}
```

Árvores Ordenadas - Combinação (2)

- ▶ Solução proposta por Sedgewick (Prog 12.16) incorrecta!
 - ▶ Fazer join , logo nas sub-árvores esquerda e direita, impede verificar o mesmo dado em nós de alturas distintas!



Árvores Ordenadas - Remoção

- ▶ Um dado é eliminado da árvore, executando os passos
 - ▶ Procurar o nó de residência
 - ▶ Se existir, executar:
 - ▶ Combinar sub-árvores esquerda e direita
 - ▶ Substituir nó pelo resultado da combinação

```
link *delete(int i, link *tree)
{
    link *aux = tree;
    if (tree == NULL) return NULL;

    if (i < tree->data) tree->left = delete(i, tree->left);
    if (i > tree->data) tree->right = delete(i, tree->right);
    if(i == tree->data ) {
        tree = join(tree->left, tree->right);
        free(aux);
    }
    return tree;
}
```

Síntese da Aula 4

- ▶ Árvores Ordenadas
 - ▶ Conceito e exemplos
 - ▶ Inserção em árvores ordenadas
- ▶ Árvores Ordenadas balanceadas
 - ▶ Mecanismo de rotação
 - ▶ Exemplos
 - ▶ Inserção com balanceamento
- ▶ Combinação de árvores ordenadas
- ▶ Remoção de vértices em árvores ordenadas