Estruturas de Dados

Victor Machado da Silva, MSc
victor.silva@professores.ibmec.edu.br

ibmec

Índice

Apresentação do curso

Apresentação do curso

- Contato: victor.silva@professores.ibmec.edu.br
- Grupo no Whatsapp: https://chat.whatsapp.com/IqnSl5PTiVBI1WNwMpXxSS
- Material: www.victor0machado.github.io

Apresentação do curso

- Avaliação
  - Proporção
    - AC (20%): atividades em sala
    - AP1 (40%): projeto + prova objetiva (sem consulta)
    - AP2 (40%): projeto + prova objetiva (sem consulta)
  - Detalhes das entregas
    - Atividades da AC individuais ou em dupla
    - Projetos da AP1 e AP2 em grupos, mínimo 2 e máximo 3 pessoas

Algoritmos

Introdução

Um algoritmo é um processo sistemático para a resolução de um problema. O desenvolvimento de algoritmos é particularmente importante para problemas a serem solucionados em um computador, pela própria natureza do instrumento utilizado.

Um algoritmo computa uma saída, o resultado do problema, a partir de uma entrada, as informações inicialmente conhecidas e que permitem encontrar a solução do problema. Durante o processo de computação o algoritmo manipula dados, gerados a partir da sua entrada.

O estudo de estruturas de dados não pode ser desvinculado de seus aspectos algorítmicos. A escolha correta da estrutura adequada a cada caso depende diretamente do conhecimento de algoritmos para manipular a estrutura de maneira eficiente.

Apresentação dos algoritmos

As convenções seguintes serão utilizadas com respeito à linguagem:

- O início e o final de cada bloco são determinados por indentação, isto é, pela posição da margem esquerda. Se uma certa linha do algoritmo inicia um bloco, ele se estende até a última linha seguinte, cuja margem esquerda se localiza mais à direita do que a primeira do bloco;
- A declaração de atribuição é indicada pelo símbolo :=;
- As declarações seguintes são empregadas com significado semelhante ao usual:

```
se... então
se... então... senão
enquanto... faça
para... faça
pare
```

- Variáveis simples, vetores, matrizes e registro são considerados como tradicionalmente em linguagens de programação. Os elementos de vetores e matrizes são identificados por índices entre colchetes.

```
para i := 1, ..., |__n/2__|
    temp := S[i]
    S[i] := S[n - i + 1]
    S[n - i + 1] := temp
```

Recursividade

Um tipo especial de procedimento será utilizado, algumas vezes, ao longo do curso. É aquele que contém, em sua descrição, uma ou mais chamadas a si mesmo. Um procedimento dessa natureza é denominado **recursivo**.

Naturalmente, todo procedimento, recursivo ou não, deve possuir pelo menos uma chamada proveniente de um local exterior a ele. Essa chamada é denominada **externa**.

Um procedimento não recursivo é, pois, aquele em que todas as chamadas são externas.

Recursividade

O exemplo clássico mais simples de recursividade é o cálculo do fatorial de um inteiro $n \geq 0$:

```
função fat(i)
    fat(i) := se i <= 1 então 1 senão i * fat(i - 1)
```

```
fat[0] := 1
para j := 1, ..., n faça
    fat[j] := j * fat[j - 1]
```

Um exemplo conhecido, onde a solução recursiva é natural e intuitiva, é o do Problema da Torre de Hanói.

Recursividade

A solução do problema é descrita a seguir. Para $n > 1$, o pino-trabalho deve ser utilizado como área de armazenamento temporário. O raciocínio utilizado para resolver o problema é semelhante ao de uma prova matemática por indução. Suponha que se saiba como resolver o problema até $n - 1$ discos, $n > 1$, de forma recursiva. A extensão para n discos pode ser obtida pela realização dos seguintes passos:

- Resolver o problema da Torre de Hanói para os $n - 1$ discos do topo do pino-origem A, supondo que o pino-destino seja C e o trabalho seja B;
- Mover o n-ésimo pino (maior de todos) de A para B;
- Resolver o problema da Torre de Hanói para os $n - 1$ discos localizados no pino C, suposto origem, considerando os pinos A e B como trabalho e destino, respectivamente.

```
procedimento hanoi(n, A, B, C)
se n > 0 então
    hanoi(n - 1, A, C, B)
    mover o disco do topo de A para B
    hanoi(n - 1, C, B, A)
```

Complexidade de algoritmos

Conforme já mencionado, uma característica muito importante de qualquer algoritmo é o seu tempo de execução. Naturalmente, é possível determiná-lo através de métodos empíricos, isto é, obter o tempo de execução através da execução propriamente dita do algoritmo, considerando-se entradas diversas.

Ao contrário do método empírico, o método analítico visa aferir o tempo de execução de forma independente do computador utilizado, da linguagem e dos compiladores empregados e das condições locais de processamento.

Complexidade de algoritmos

As seguintes simplificações serão introduzidas para o modelo proposto:

- Suponha que a quantidade de dados a serem manipulados pelo algoritmo seja suficientemente grande. Somente o comportamento assintótico será avaliado.
- Não serão consideradas constantes aditivas ou multiplicativas na expressão matemática obtida. Isto é, a expressão matemática obtida será válida, a menos de tais constantes.

O processo de execução de um algoritmo pode ser dividido em etapas elementares, denominadas *passos*. Cada passo consiste na execução de um número fixo de operações básicas cujos tempos de execução são considerados constantes.

```
para i := 1, ..., |__n/2__|
    temp := S[i]
    S[i] := S[n - i + 1]
    S[n - i + 1] := temp
```

Complexidade de algoritmos

Como exemplos adicionais, considere os problemas de determinar as matrizes soma C e produto D de duas matrizes dadas.

```
para i := 1, ..., n faça
    para j := 1, ..., n faça
        C[i][j] := A[i][j] + B[i][j]
```

```
para i:= 1, ..., n faça
    para j := 1, ..., n faça
        C[i][j] := 0
        para k := 1, ..., n faça
            C[i][j] := C[i][j] + A[i][k] * B[k][j]
```

Complexidade de algoritmos

Noção de complexidade:

- Seja A um algoritmo, $\{E_1, \dots, E_m\}$, o conjunto de todas as entradas possíveis de A. Denote por t_i o número de passos efetuados por A, quando a entrada for E_i. Definem-se, com p_i sendo a probabilidade de ocorrência da entrada E_i:
  - Complexidade do pior caso: $\max_{E_{i \in E}}\{t_i\}$;
  - Complexidade do melhor caso: $min_{E_{i \in E}}\{t_i\}$;
  - Complexidade do caso médio: $\sum_{1 \leq i \leq m}(p_i \times t_i)$.
- As complexidades têm por objetivo avaliar a eficiência de tempo ou espaço. A complexidade de tempo de pior caso corresponde ao número de passos que o algoritmo efetua no seu pior caso de execução, isto é, para a entrada mais desfavorável. De certa forma, a complexidade de pior caso é a mais importante das três mencionadas.

A Notação O

Quando se considera o número de passos efetuados por um algoritmo, podem-se desprezar constantes aditivas ou multiplicativas.

Por exemplo, um valor de número de passos igual a 3n será aproximado para n.

Além disso, como o interesse é restrito a valores assintóticos, termos de menor grau também podem ser desprezados. Assim, um valor de número de passos igual a $n^2 + n$ será aproximado para n^2. O valor $6n^3 + 4n - 9$ será transformado em n^3.

Torna-se útil, portanto, descrever operadores matemáticos que sejam capazes de representar situações como essas. A notação O será utilizada com essa finalidade.

A Notação O

Sejam f, h funções reais positivas de variável inteira n. Diz-se que f é $O(h)$, escrevendo-se $f = O(h)$, quando existir uma constante $c > 0$ e um valor inteiro n_o, tal que:

$$n > n_o \Rightarrow f(n) \leq c \times h(n)$$

Ou seja, a função h atua como um limite superior para valores assintóticos da função f. Em seguida são apresentados alguns exemplos da notação O.

$$f = n^2 - 1 \Rightarrow f = O(n^2)$$
$$f = n^3 - 1 \Rightarrow f = O(n^3)$$
$$f = 403 \Rightarrow f = O(1)$$
$$f = 5 + 2\log n + 3\log^2 n \Rightarrow f = O(\log^2 n)$$

Listas Lineares

Introdução

- Dentre as estruturas de dados não primitivas, as listas lineares são as de manipulação mais simples. Iremos discutir seus algoritmos e estruturas de armazenamento.

- Uma lista linear agrupa informações referentes a um conjunto de elementos que, de alguma forma, se relacionam entre si. Ela pode se constituir, por exemplo, de informações sobre os funcionários de uma empresa, sobre notas de compras, itens de estoque, notas de alunos, etc.

- Uma *lista linear*, ou *tabela*, é então um conjunto de $n \geq 0$ nós L[1], L[2], ..., L[n] tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear.

Introdução

- As operações mais frequentes em listas são a *busca*, a *inclusão* e a *remoção* de um determinado elemento, o que, aliás, ocorre na maioria das estruturas de dados. Tais operações podem ser consideradas como básicas e, por essa razão, é necessário que os algoritmos que as implementem sejam eficientes.

- Outras operações também são relevantes, porém não serão estudadas a fundo neste curso:
  - Alteração de um elemento da lista;
  - Combinação de duas ou mais listas lineares.
  - Ordenação dos nós segundo um determinado campo;
  - Determinação do primeiro (ou do último) nó da lista;
  - etc.

Introdução

- As operações mais frequentes em listas são a *busca*, a *inclusão* e a *remoção* de um determinado elemento, o que, aliás, ocorre na maioria das estruturas de dados. Tais operações podem ser consideradas como básicas e, por essa razão, é necessário que os algoritmos que as implementem sejam eficientes.

- Outras operações também são relevantes, porém não serão estudadas a fundo neste curso:
  - Alteração de um elemento da lista;
  - Combinação de duas ou mais listas lineares.
  - Ordenação dos nós segundo um determinado campo;
  - Determinação do primeiro (ou do último) nó da lista;
  - etc.

Introdução

- Casos particulares de listas são de especial interesse:
  - Se as inserções e remoções são permitidas apenas nas extremidades da lista, ela recebe o nome de **deque** (uma abreviatura do inglês *double ended queue*);
  - Se as inserções e as remoções são realizadas somente em um extremo, a lista é denominada **pilha**;
  - A lista é denominada **fila** no caso em que as inserções são realizadas em um extremo e remoções em outro.

Introdução

- O tipo de armazenamento de uma lista linear pode ser classificado de acordo com a posição relativa na memória de dois nós consecutivos na lista:
  - Quando dois nós consecutivos na lista são alocados contiguamente na memória do computador, a lista é classificada como de *alocação sequencial de memória*;
  - Quando dois nós consecutivos não são alocados contiguamente, a lista é classificada como de alocação encadeada de memória.
- A escolha de um ou outro tipo depende essencialmente das operações que serão executadas sobre a lista, do número de listas envolvidas na operação, bem como de características particulares.

Alocação sequencial

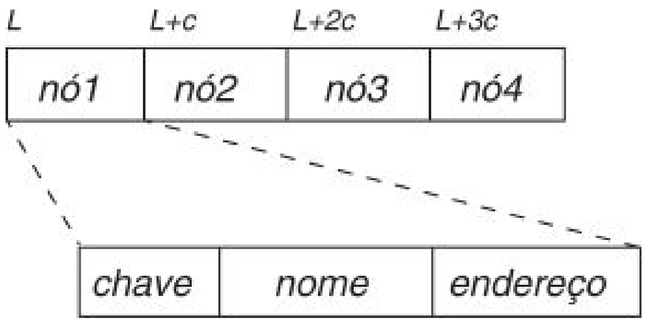
- A maneira mais simples de se manter uma lista linear na memória do computador é colocar seus nós em posições contíguas.

- O armazenamento sequencial é particularmente atraente no caso de filas e pilhas porque, nessas estruturas, as operações básicas podem ser implementadas de forma bastante eficiente.

- Esse tratamento pode, contudo, se tornar oneroso em termos de memória quando se empregam diversas estruturas simultaneamente. Nesse caso, a utilização ou não do armazenamento sequencial dependeria de um estudo cuidadoso das opções existentes.

Listas lineares em alocação sequencial

- Seja uma lista linear. Cada nó é formado por *campos*, que armazenam as características distintas dos elementos da lista. Além disso, cada nó da lista possui, geralmente, um identificador, denominado *chave*. A chave, quando presente, se constitui em um dos campos do nó. Os nós podem se encontrar ordenados, ou não, segundo os valores de suas chaves. No primeiro caso a lista é denominada *ordenada*, e *não ordenada* no caso contrário.



Listas lineares em alocação sequencial

- Observe que, para cada elemento da tabela referenciado na busca, o algoritmo realiza dois testes.

- A complexidade de pior caso é $O(n)$.

```
função busca1(x)
    i := 1
    busca1 := 0
    enquanto i <= n faça
        se L[i].chave = x então
            % chave encontrada
            busca1 := i
            i := n + 1
        senão
            % pesquisa prossegue
            i:= i + 1
```

Listas lineares em alocação sequencial

- Quando a lista está ordenada, pode-se tirar proveito desse fato. Se o número procurado não pertence à lista, não há necessidade de percorrê-la até o final.

- Podemos criar um novo nó ao final da lista para evitar a dupla comparação do algoritmo anterior. No entanto, a complexidade de pior caso ainda é $O(n)$.

```
função busca-ord(x)
    L[n + 1].chave := x
    i := 1
    enquanto L[i].chave < x faça
        i := i + 1
    se i = n + 1 ou L[i].chave != x então
        busca-ord := 0
    senão
        busca-ord := i
```

Listas lineares em alocação sequencial

- Ainda no caso das listas ordenadas, um algoritmo diverso e bem mais eficiente pode ser apresentado: a *busca binária*. Em tabelas, o primeiro nó pesquisado é o que se encontra no meio; se a comparação não é positiva, metade da tabela pode ser abandonada na busca, uma vez que o valor procurado se encontra ou na metade inferior, ou na superior. Esse procedimento, aplicado recursivamente, esgota a tabela.

```
função busca-bin(x)
    inf := 1
    sup := n
    busca-bin := 0
    enquanto inf <= sup faça
        % índice a ser buscado
        meio := |__(inf + sup) / 2__|
        se L[meio].chave = x então
            % elemento encontrado
            busca-bin := meio
            inf := sup + 1
        senão se L[meio].chave < x então
            inf := meio + 1
        senão
            sup := meio - 1
```

Listas lineares em alocação sequencial

- A complexidade do algoritmo de busca binária pode ser avaliada da seguinte forma. O pior caso acontece quando o elemento procurado é o último a ser encontrado, ou mesmo não é encontrado. Na primeira iteração, a dimensão da tabela é n, e algumas operações são realizadas para situar o valor procurado. Na segunda, a dimensão se reduz a $\lfloor n/2 \rfloor$, e assim sucessivamente. Ao final, a dimensão da tabela é 1. Então, no pior caso:
  - 1ª iteração: a dimensão da tabela é n;
  - 2ª iteração: a dimensão da tabela é $\lfloor n/2 \rfloor$;
  - 3ª iteração: a dimensão da tabela é $\lfloor \lfloor n/2 \rfloor /2 \rfloor$;
  - mª iteração: a dimensão da tabela é 1.

- Portanto, o número máximo de iterações é $1 + \log_2 n$, tendo, portanto, complexidade $O(\log_2 n)$.

Listas lineares em alocação sequencial

- Ambas as operações de inserção e remoção utilizam o procedimento de busca. No primeiro caso, o objetivo é evitar chaves repetidas e, no segundo, a necessidade de localizar o elemento a ser removido.

- Os dois algoritmos a seguir consideram tabelas não ordenadas. A memória pressuposta disponível tem M posições. Devem-se levar em conta as hipóteses de se tentar fazer inserções numa lista que já ocupa M posições (situação conhecida como **overflow**), bem como a tentativa de remoção de um elemento de uma lista vazia (**underflow**). A atitude a ser tomada em cada um desses casos depende do problema tratado.

- Ambos as operações possuem complexidade $O(n)$, apesar do algoritmo de remoção ser mais lento que o de inserção.

Listas lineares em alocação sequencial

Algoritmo 11. Inserção de um nó na lista L

```
se n < M então
    se busca(x) = 0 então
        L[n + 1] := novo-valor
        n := n + 1
    senão
        "elemento já existe na tabela"
senão
    "overflow"
```

Algoritmo 12. Remoção de um nó na lista L

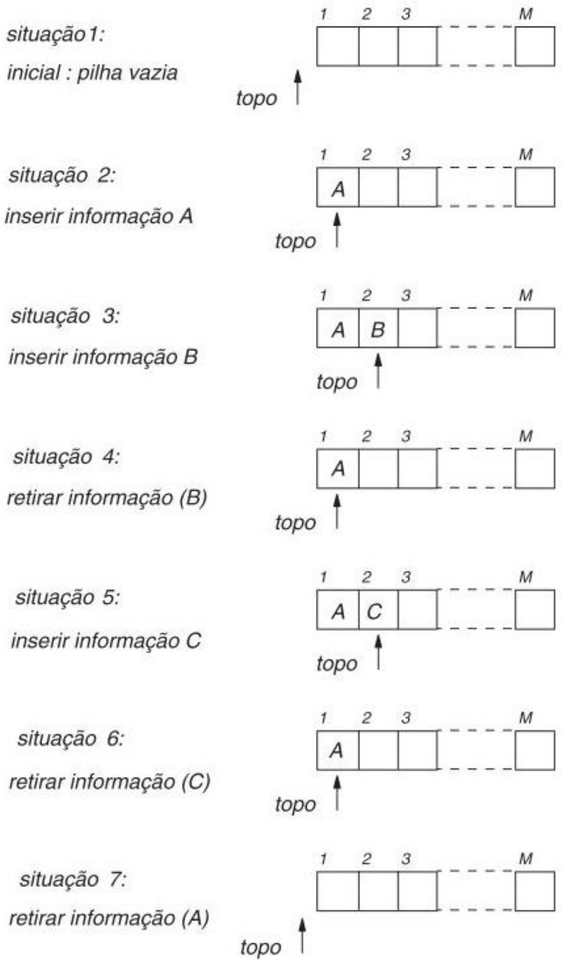
```
se n != 0 então
    indice := busca(x)
    se indice != 0 então
        valor-recuperado := L[indice]
        para i := indice, n - 1 faça
            L[i] := L[i + 1]
        n := n - 1
    senão "elemento não se encontra na tabela"
senão
    "underflow"
```

Pilhas e Filas

- Em geral, o armazenamento sequencial de listas é empregado quando as estruturas, ao longo do tempo, sofrem poucas remoções e inserções. Em casos particulares de listas, esse armazenamento também é empregado. Nessa caso, a situação favorável é aquela em que inserções e remoções não acarretam movimentação de nós, o que ocorre se os elementos a serem inseridos e removidos estão em posições especiais, como a primeira ou a última posição. Deques, pilhas e filas satisfazem tais condições.

- Na alocação sequencial de listas genéricas, considera-se sempre a primeira posição da lista no endereço 1 da memória disponível. Uma alternativa a essa estratégia consiste na utilização de indicadores especiais, denominados **ponteiros**, para o acesso a posições selecionadas.

Pilhas e Filas



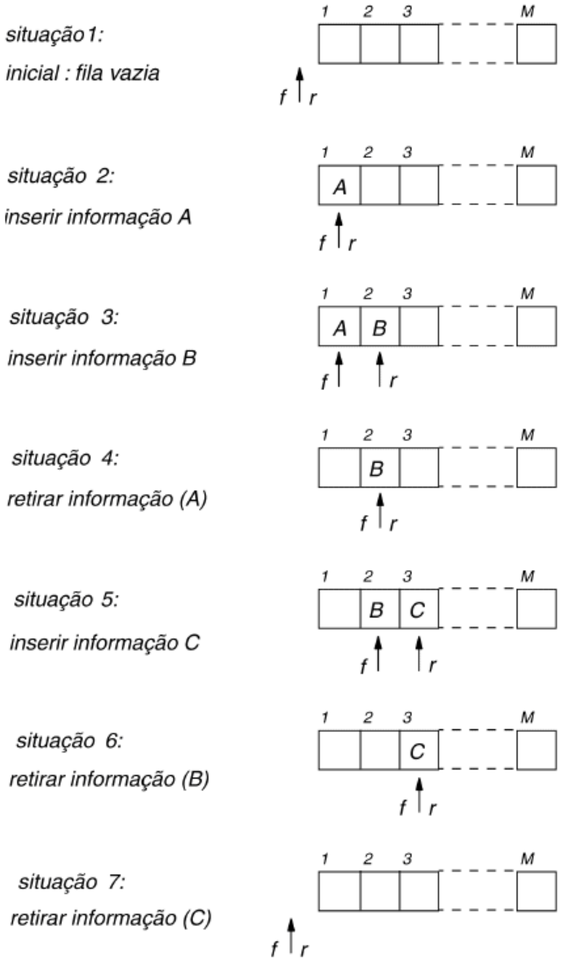
Algoritmo 13. Inserção na pilha P

```
se topo != M então
    topo := topo + 1
    P[topo] := novo-valor
senão
    "overflow"
```

Algoritmo 14. Remoção da pilha P

```
Se topo != 0 então
    valor-recuperado := P[topo]
    topo := topo - 1
senão
    "underflow"
```

Pilhas e Filas



Algoritmo 15. Inserção na fila F

```
prov := r mod M + 1
se prov != f então
    r := prov
    F[r] := novo-valor
    se f = 0 então
        f := 1
senão
    "overflow"
```

Algoritmo 16. Remoção da fila F

```
se f != 0 então
    valor-recuperado := F[f]
    se f = r então
        f := 0
        r := 0
    senão
        f := f mod M + 1
senão
    "underflow"
```

Alocação encadeada

- O desempenho dos algoritmos que implementam operações realizadas em listas com alocação sequencial, mesmo sendo estes muito simples, pode ser bastante fraco. E mais, quando está prevista a utilização concomitante de mais de duas listas a gerência de memória se torna mais complexa.

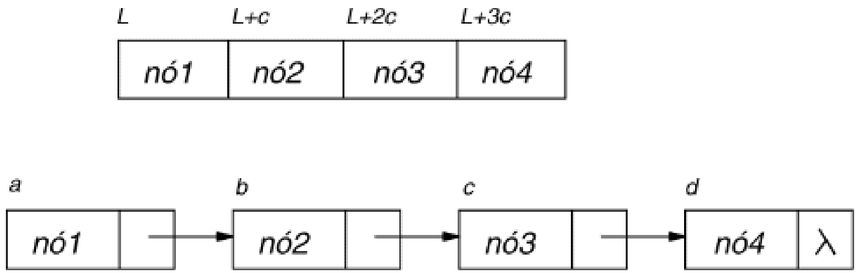
- Nesses casos se justifica a utilização da alocação encadeada, também conhecida por alocação dinâmica, uma vez que posições de memórias são alocadas (ou desalocadas) na medida em que são necessárias (ou dispensadas).

- Os nós de uma lista encontram-se então aleatoriamente dispostos na memória e são interligados por ponteiros, que indicam a posição do próximo elemento da tabela. É necessário o acréscimo de um campo a cada nó, justamente o que indica o endereço do próximo nó da lista.

Alocação encadeada

- Há vantagens e desvantagens associadas a cada tipo de alocação. Estas, entretanto, só podem ser precisamente medidas ao se conhecerem as operações envolvidas na aplicação desejada.

- De maneira geral pode-se afirmar que a alocação encadeada, a despeito de um gasto de memória maior em virtude da necessidade de um novo campo no nó (o campo do ponteiro), é mais conveniente quando o problema inclui o tratamento de mais uma lista.



Listas simplesmente encadeadas

- Qualquer estrutura, inclusive listas, que seja armazenada em alocação encadeada requer o uso de um ponteiro que indique o endereço de seu primeiro nó. O percurso de uma lista é feito então a partir desse ponteiro. A ideia consiste em seguir consecutivamente pelos endereços existentes no campo que indica o próximo nó, da mesma forma que na alocação sequencial se acrescentava uma unidade ao índice do percurso.

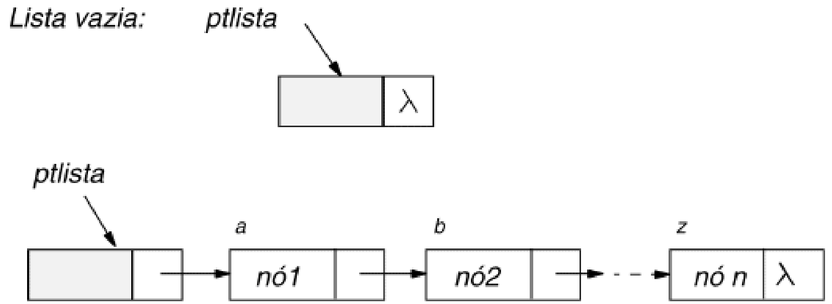
Algoritmo 17. Impressão da lista apontada por ptlista

```
pont := ptlista
enquanto pont != lambda faça
    imprimir(pont*.info)
    pont := pont*.prox
```

Listas simplesmente encadeadas

- O algoritmo de busca em listas encadeadas possui mais restrições, além da eficiência: por exemplo, a existência de um ponteiro indicando o primeiro nó da lista obriga os algoritmos de inserção e remoção a apresentarem testes especiais para verificar se o nó desejado é o primeiro da lista.

- Isto pode ser resolvido por uma pequena variação na estrutura de armazenamento: a criação de um nó especial, chamado **nó-cabeça**, nunca removido, que passa a ser o nó indicado pelo ponteiro de início de lista.



Listas simplesmente encadeadas

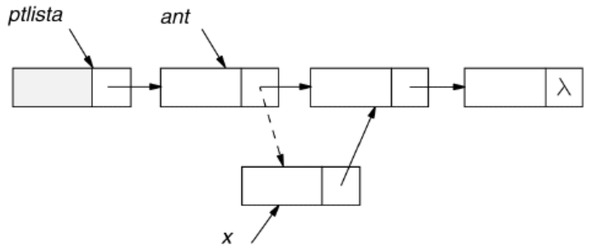
Algoritmo 18. Busca em uma lista ordenada

```
procedimento busca-enc(x, ant, pont)
    ant := ptlista
    pont := lambda
    % ponteiro de percurso
    ptr := ptlista*.prox
    enquanto ptr != lambda faça
        se ptr*.chave < x então
            % atualiza ant e ptr
            ant := ptr
            ptr := ptr*.prox
        senão se ptr*.chave = x então
            % chave encontrada
            pont := ptr
            ptr := lambda
```

Listas simplesmente encadeadas

- Após a realização da busca, as operações de inserção e remoção em uma lista encadeada são triviais.

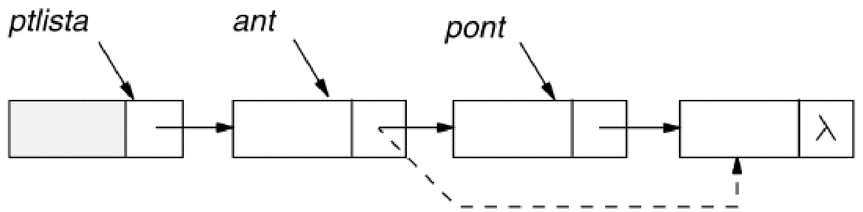
- Há três fases a serem cumpridas: a ocupação do espaço na memória, o acesso ao campo de informações e o acerto da estrutu



Algoritmo 19. Inserção de um nó após o nó apontado por busca-enc(x, ant, pont)

```
se pont = lambda então
    % solicitar nó
    ocupar(pt)
    % inicializar nó
    pt*.info := novo-valor
    % acertar lista
    pt*.chave := x
    pt*.prox := ant*.prox
    ant*.prox := pt
senão
    "elemento já está na tabela"
```

Listas simplesmente encadeadas



Algoritmo 20. Remoção do nó apontado por pont na lista

```
busca-enc(x, ant, pont)
se pont != lambda então
    % acertar lista
    ant*.prox := pont*.prox
    % utilizar nó
    valor-recuperado := pont*.info
    % devolver nó
    desocupar(pont)
senão
    "nó não se encontra na tabela"
```

Pilhas e Filas encadeadas

- Como casos particulares, algumas modificações são necessárias para implementar operações eficientes em pilhas e filas. No caso de pilhas, as operações são muito simples. Considerando-se listas simplesmente encadeadas (sem nó-cabeça), o topo da pilha é o primeiro nó da lista, apontado por uma variável ponteiro $topo$. Se a pilha estiver vazia então $topo = lambda$. Filas exigem duas variáveis do tipo ponteiro: $inicio$, que aponta para o primeiro nó da lista, e fim, que aponta para o último. Na fila vazia, ambos apontam para $lambda$. Os algoritmos que se seguem implementam essas operações.

Pilhas e Filas encadeadas

Algoritmo 21. Inserção na pilha

```
% solicitar nó
ocupar(pt)
% inicializar nó
pt*.info := novo-valor
pt*.prox := topo
% acertar pilha
topo := pt
```

Algoritmo 22. Remoção da pilha

```
se topo != lambda então
    % acertar pilha
    pt := topo
    topo := topo*.prox
    % utilizar nó
    valor-recuperado := pt*.info
    % devolver nó
    desocupar(pt)
senão
    "underflow"
```

Pilhas e Filas encadeadas

Algoritmo 23. Inserção na fila

```
% solicitar nó
ocupar(pt)
% inicializar nó
pt*.info := novo-valor
pt*.prox := lambda
% acertar fila
se fim != lambda então
    fim*.prox := pt
senão inicio := pt
fim := pt
```

Algoritmo 24. Remoção da fila

```
se inicio != lambda então
    pt := inicio
    % acertar fila
    inicio := inicio*.prox
    se inicio = lambda então
        fim := lambda
    % utilizar nó
    valor-recuperado := pt*.info
    % devolver nó
    desocupar(pt)
senão
    "underflow"
```

Árvores

Introdução

- Em diversas aplicações necessita-se de estruturas mais complexas do que as puramente sequenciais, examinadas no capítulo anterior. Entre essas, destacam-se as árvores, por existirem inúmeros problemas práticos que podem ser modelados através delas. Além disso, as árvores, em geral, admitem um tratamento computacional simples e eficiente. Isso não pode ser dito de estruturas mais gerais do que as árvores, como os grafos, por exemplo.

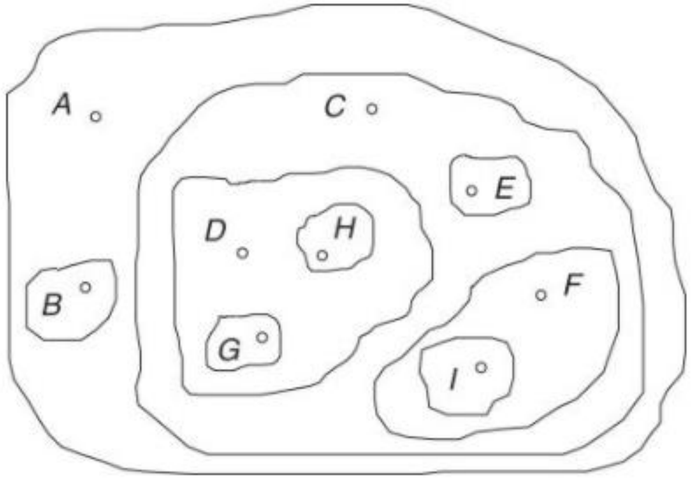
- Neste capítulo são apresentados os conceitos iniciais relativos às árvores, bem como os algoritmos para sua manipulação computacional básica.

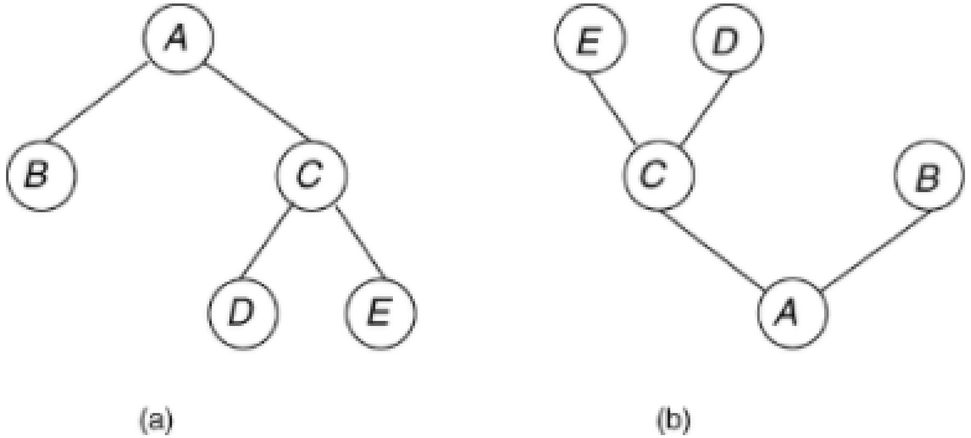
Definições e representações básicas

- Uma **árvore enraizada T**, ou simplesmente **árvore**, é um conjunto finito de elementos denominados *nós* ou *vértices* tais que:
  - $T = \emptyset$, e a árvore é dita *vazia*; ou
  - Existe um nó especial chamado *raiz* de $T\big(r(T)\big)$; os restantes constituem um único conjunto vazio ou são divididos em $m \geq 1$ conjuntos disjuntos não vazios, as subárvores de $r(T)$, ou simplesmente **subárvores**, cada qual, por sua vez, uma árvore.

- Uma floresta é um conjunto de árvores. Se v é um nó de T, a notação $T(v)$ indica a subárvore de T com raiz v.

Definições e representações básicas





(a) (b)

Definições e representações básicas

- Seja v o nó raiz da subárvore $T(v)$ de T. Os nós raízes $w_1, w_2, \ldots, w_j$ das subárvores de $T(v)$ são chamados **filhos** de v; v é chamado pai de $w_1, w_2, \ldots, w_j$. Os nós $w_1, w_2, \ldots, w_j$ são **irmãos**. Se z é filho de w_1, então w_2 é **tio** de z e v é **avô** de z. O número de filhos de um nó é chamado de **grau de saída** desse nó. Se x pertence à subárvore $T(v)$, x é **descendente** de v, e v é **ancestral** de x. Nesse caso, sendo x diferente de v, x é **descendente próprio** de v, e v é **ancestral próprio** de x.

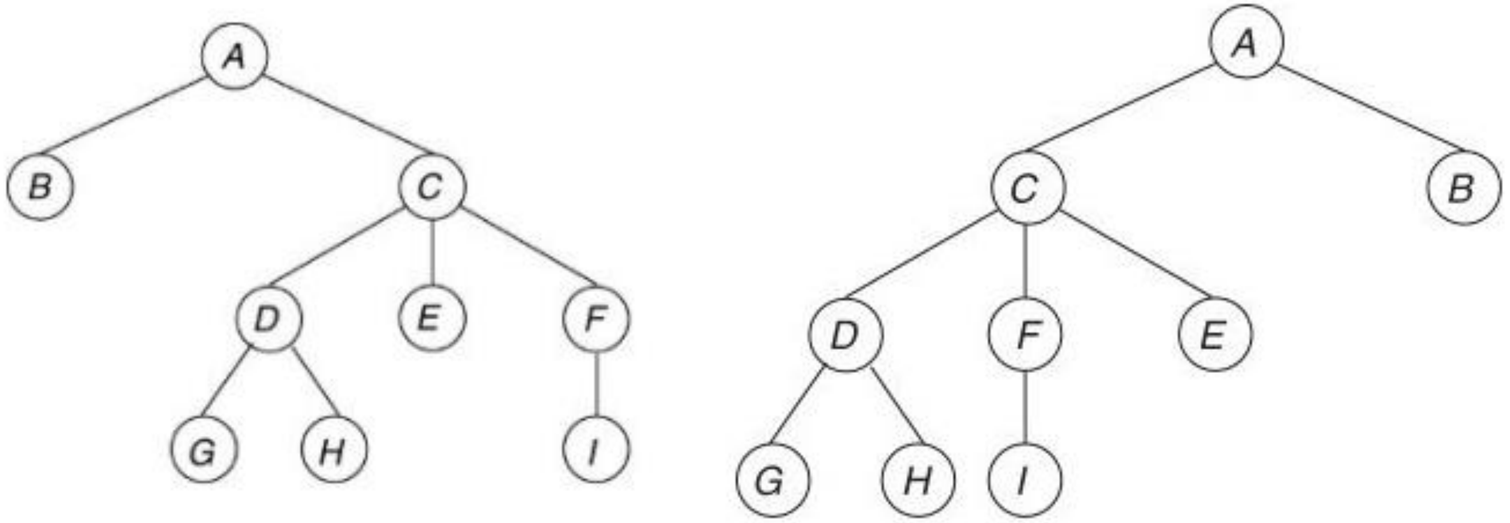
- Um nó que não possui descendentes próprios é chamado de **folha**. Toda árvore com $n > 1$ nós possui no mínimo 1 e no máximo $n - 1$ folhas. Um nó não folha é dito **interior**.

Definições e representações básicas

- Uma sequência de nós distintos $v_1, v_2, \ldots, v_k$, tal que existe sempre entre nós consecutivos (v_1 e v_2, v_2 e v_3, ..., v_{k-1} e v_k) a relação "é filho de" ou "é pai de", é denominada **caminho da árvore**. Diz-se que v_1 alcança v_k e vice-versa. Um caminho de k vértices é obtido pela sequência de $k-1$ pares da relação. O valor $k-1$ é o comprimento do caminho. **Nível de um nó** v é o número de nós do caminho da raiz até o nó v. O nível da raiz é, portanto, igual a 1. A **altura** de um nó v é o número de nós do maior caminho de v até um de seus descendentes. As folhas têm altura 1. A altura da árvore T é igual ao nível máximo de seus nós. Representa-se a altura de T por $h(T)$, enquanto $h(v)$ é a altura da subárvore de raiz v.

Definições e representações básicas

- Uma **árvore ordenada** é aquela na qual os filhos de cada nó estão ordenados. Assume-se que tal ordenação se desenvolva da esquerda para a direita. Assim, as árvores da figura abaixo são distintas se consideradas como ordenadas. Contudo, elas podem se tornar coincidentes mediante uma reordenação de nós irmãos.



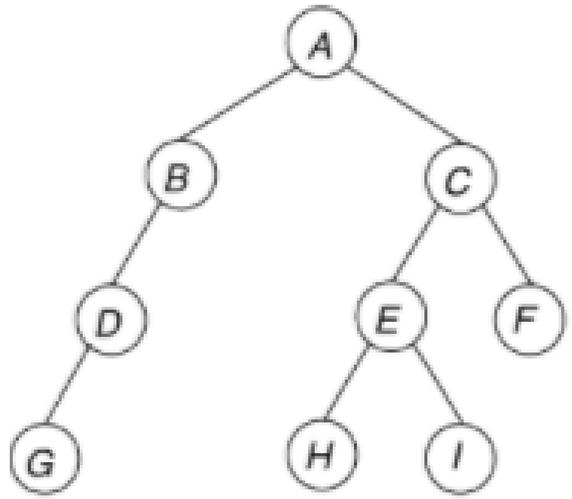
Árvores binárias

- Conforme já mencionado, as árvores constituem as estruturas não sequenciais com maior aplicação em computação. Dentre as árvores, as binárias são, sem dúvida, as mais comuns.

- Uma **árvore binária** T é um conjunto finito de elementos denominados nós ou vértices, tal que:
  - $T = \emptyset$, e a árvore é dita *vazia*; ou
  - Existe um nó especial chamado *raiz* de $T\big(r(T)\big)$; e os restantes podem ser divididos em dois subconjuntos disjuntos, $T_E\big(r(T)\big)$ e $T_D\big(r(T)\big)$, a subárvore esquerda e a direita da raiz, respectivamente, as quais são também árvores binárias.

Árvores binárias

- A raiz da subárvore esquerda (direita) de um nó v, se existir, é denominada **filho esquerdo (direito)** de v. Naturalmente, o esquerdo pode existir sem o direito e vice-versa. Analogamente à seção anterior, a notação $T(v)$ indica a (sub) árvore binária, cuja raiz é v e cujas subárvores esquerda e direita de T são $T_E(v)$ e $T_D(v)$, respectivamente.

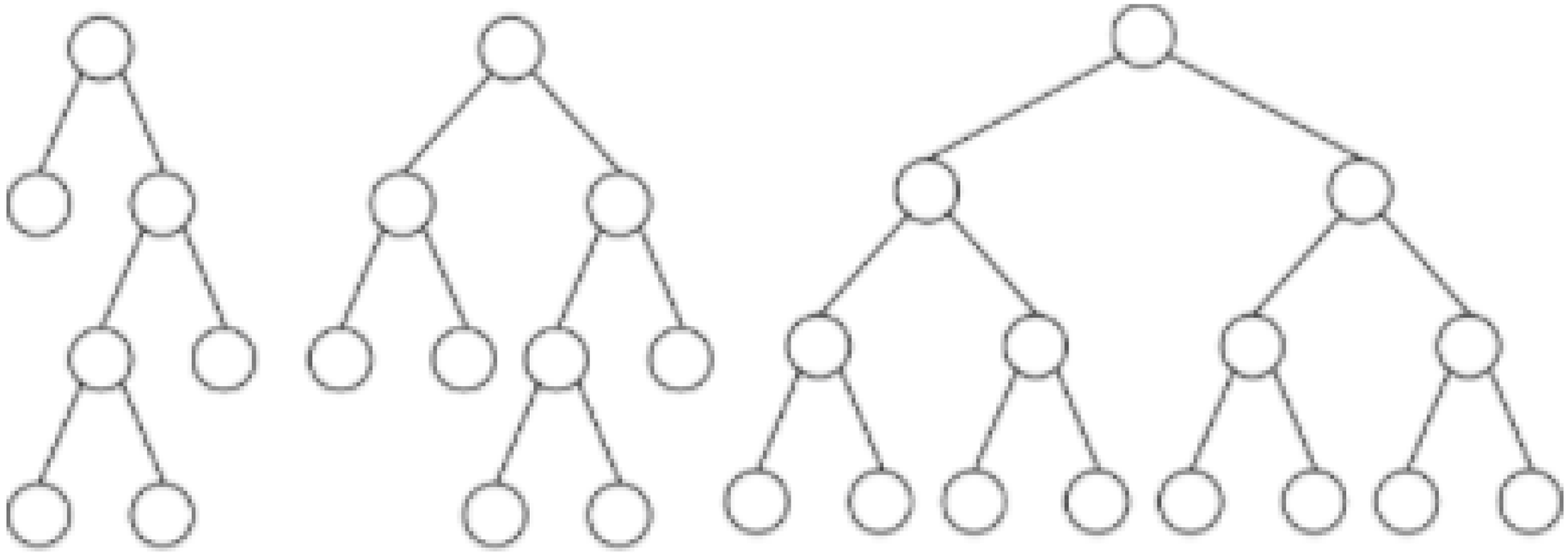


Árvores binárias

- Toda árvore binária com n nós possui exatamente $n + 1$ subárvores vazias entre suas subárvores esquerdas e direitas. Por exemplo, a árvore da figura anterior possui 9 nós e 10 subárvores vazias: as subárvores esquerda e direita dos nós F, G, H, I e as subárvores direitas de B e D.

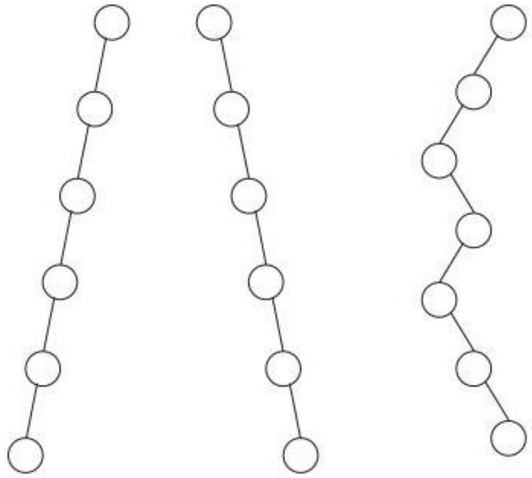
- Em seguida, são introduzidos alguns tipos especiais de árvores binárias:
  - Uma *árvore estritamente binária* é uma árvore binária em que cada nó possui 0 ou 2 filhos;
  - Uma *árvore binária completa* é aquela que apresenta a seguinte propriedade: se v é um nó tal que alguma subárvore de v é vazia, então v se localiza ou no último (maior) ou no penúltimo nível da árvore;
  - Uma *árvore binária cheia* é aquela em que, se v é um nó com alguma de suas subárvores vazias, então v se localiza no último nível. Segue-se que toda árvore binária cheia é completa e estritamente binária.

Árvores binárias



Árvores binárias

- A relação entre a altura de uma árvore binária e o seu número de nós é um dado importante para várias aplicações. Para um valor fixo de n, indagar-se-ia quais são as árvore binárias que possuem altura h máxima e mínima. A resposta ao primeiro problema é imediata. A árvore binária que possui altura máxima é aquela cujos nós interiores possuem exatamente uma subárvore vazia. Essas árvores são denominadas **zigue-zague**. Naturalmente, a altura de uma árvore zigue-zague é igual a n.



Altura de árvores binárias completas

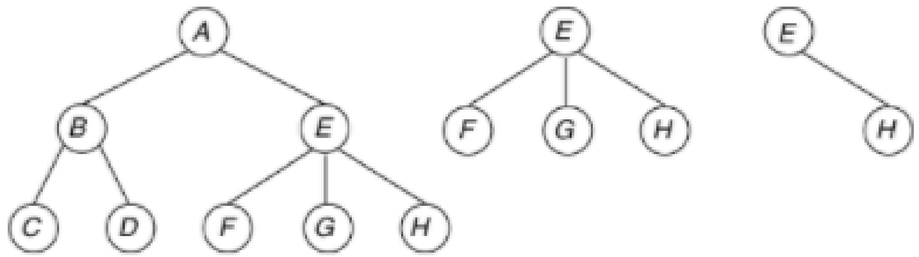
- Seja T uma árvore binária completa com $n > 0$ nós. Então T possui altura h mínima.

- A altura h mínima é dada por $h = 1 + \lfloor \log n \rfloor$, cuja demonstração, por indução, é dada abaixo:
  - Se $n = 1$, então $h = 1 + \lfloor \log n \rfloor = 1$, correto. Quando $n > 1$, suponha o resultado verdadeiro para todas as árvores binárias completas com até $n - 1$ nós. Seja T' a árvore obtida de T pela remoção de todos os nós, em número de k, do último nível. Logo, T' é uma árvore cheia com $n' = n - k$ nós. Pela hipótese de indução, $h(T') = 1 + \lfloor \log n' \rfloor$. Como T' é cheia, $n' = 2^m - 1$, para algum inteiro $m > 0$. Isto é, $h(T') = m$. Além disso, $1 \leq k \leq n' + 1$. Assim:

$$h(T) = 1 + h(T') = 1 + m = 1 + \log(n' + 1) = 1 + \lfloor \log(n' + k) \rfloor = 1 + \lfloor \log n \rfloor$$

Subárvore e subárvore parcial

- Seja T uma árvore (ou uma árvore binária) e v um nó de T. Seja $T(v)$ a subárvore de T de raiz v, e S um conjunto de nós $T(v)$ tal que $T(v) - S$ é uma árvore. A árvore $T' = T(v) - S$ é chamada **subárvore parcial** de raiz v. Observe, por exemplo, a árvore T abaixo, à esquerda. A árvore ao centro é subárvore de T de raiz E, enquanto a árvore à direita é uma subárvore parcial de T de raiz E, porém não é subárvore de T. Observe que a diferença entre uma subárvore de raiz v e uma subárvore parcial de raiz v é que a primeira contém obrigatoriamente todos os descendentes de v, enquanto a segunda, não necessariamente.



Armazenamento de árvores

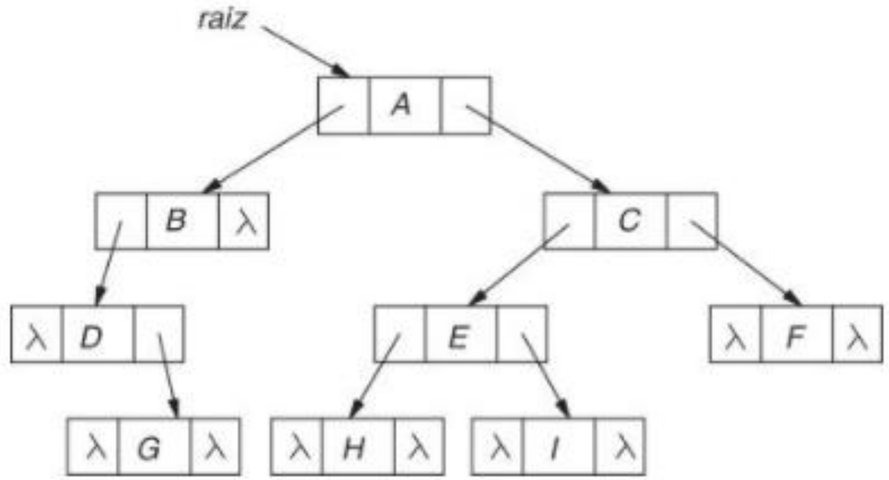
- O armazenamento de árvores pode utilizar alocação sequencial ou encadeada. As vantagens e desvantagens de uma e outra já foram discutidas. Sendo a árvore uma estrutura mais complexa do que listas lineares, as vantagens na utilização da alocação encadeada prevalecem.

- Não é difícil observar que a estrutura de armazenamento para árvores deve conter, em cada nó, ponteiros para seus filhos. A disposição mais econômica consiste em limitar o número de filhos a dois, exatamente o caso de árvores binárias. Note que o número de subárvores vazias cresce com o aumento do parâmetro m das árvores m-árias. Para um dado valor de n, a árvore binária é aquela que minimiza o número de ponteiros necessários.

Armazenamento de árvores

- O armazenamento de uma árvore binária surge naturalmente de sua definição. Cada nó deve possuir dois campos de ponteiros, **esq** e **dir**, que apontam para as suas subárvores esquerda e direita, respectivamente. O ponteiro **ptraiz** indica a raiz da árvore. Da mesma forma que na alocação encadeada de listas lineares, a memória é inicialmente considerada uma lista de espaço disponível. Os campos do nó da árvore que contém as informações pertinentes ao problema serão aqui representados como um só campo de nome **info**.

- A figura abaixo ilustra a estrutura de ponteiros usada no armazenamento de uma árvore binária.



Percurso em árvores binárias

- Por **percurso** entende-se uma visita sistemática a cada um de seus nós; esta é uma das operações básicas relativas à manipulação de árvores. Uma árvore é, essencialmente, uma estrutura não sequencial. Por isso mesmo, ela pode ser utilizada em aplicações que demandem acesso direto. Contudo, mesmo nesse tipo de aplicação é imprescindível conhecer métodos eficientes para percorrer toda a estrutura. Por exemplo, para listar o conteúdo de um arquivo é necessário utilizar algoritmos para percurso.

- Para percorrer uma árvore deve-se, então, visitar cada um de seus nós. Visitar um nó significa operar, de alguma forma, com a informação a ele relativa. Em geral, percorrer uma árvore significa visitar seus nós exatamente uma vez. Contudo, no processo de percorrer a árvore pode ser necessário passar várias vezes por alguns de seus nós sem visitá-los. A seguir são discutidas as ideias principais nas quais se baseiam alguns dos algoritmos de percurso em árvore.

Percurso em árvores binárias

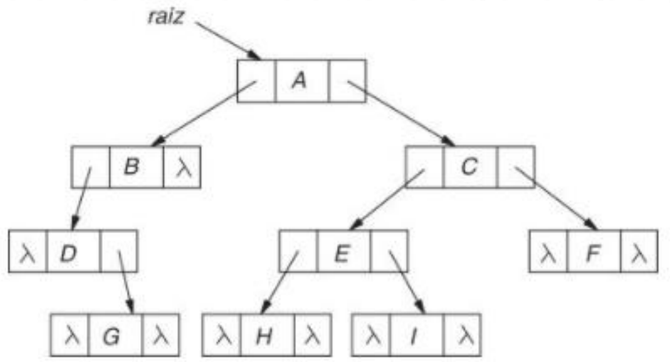
- Um dos passos de qualquer algoritmo de percurso é visitar a raiz v de cada subárvore da árvore T. Além disso, pode-se assumir que o algoritmo opere de tal forma que o percurso de T seja uma composição de percursos de suas subárvores.

- Nesse caso, poder-se-iam se identificar, no percurso de T, os percursos de suas subárvores em forma contígua. Esses percursos correspondem, no algoritmo, às operações de **percorrer subárvores esquerda** e **direita** de v, para cada nó v de T. Essas três operações (visitar e percorrer subárvores esquerda e direita) compõem um algoritmo.

- Cada um desses percursos pode ser mais ou menos adequado a um problema de aplicação dado. São apresentados a seguir três percursos diversos.

Percurso em árvores binárias

- O **percurso em pré-ordem** segue recursivamente os seguintes passos para cada subárvore da árvore:
  - Visitar a raiz;
  - Percorrer sua subárvore esquerda, em pré-ordem;
  - Percorrer sua subárvore direita, em pré-ordem.



- Para a árvore da figura ao lado, o percurso em pré-ordem para impressão de nós fornece a seguinte saída: A B D G C E H I F

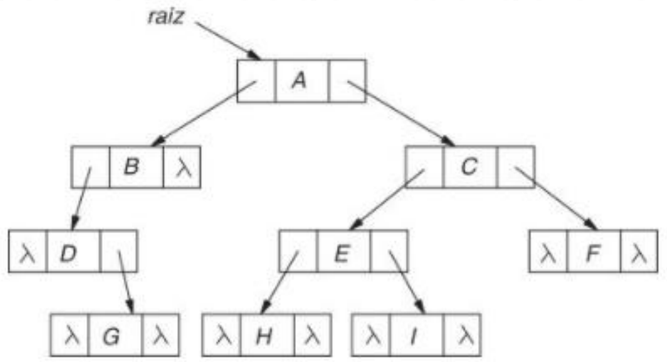
Algoritmo 25. Percurso em pré-ordem

```
procedimento pre(pt)
    visita(pt)
    se pt*.esq != lambda então pre(pt*.esq)
    se pt*.dir != lambda então pre(pt*.dir)

se prtaiz != lambda então pre(ptraiz)
```

Percurso em árvores binárias

- O **percurso em ordem simétrica** é composto dos passos seguin[illegible] subárvores:
  - Percorrer sua subárvore esquerda, em ordem simétrica;
  - Visitar a raiz;
  - Percorrer sua subárvore direita, em ordem simétrica.



- Para a árvore da figura ao lado, o percurso em ordem simétrica para impressão de nós fornece a seguinte[illegible]

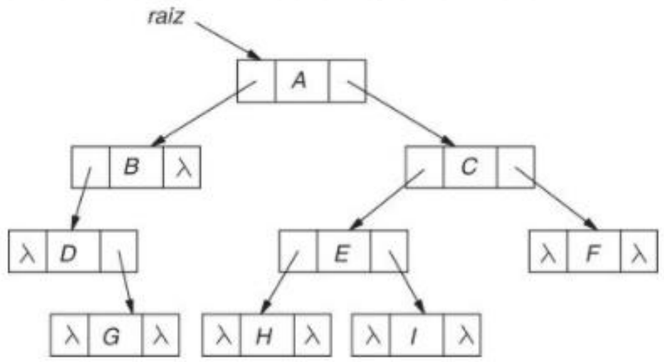
Algoritmo 26. Percurso em ordem simétrica

```
procedimento simet(pt)
    se pt*.esq != lambda então simet(pt*.esq)
    visita(pt)
    se pt*.dir != lambda então simet(pt*.dir)

se prtaiz != lambda então simet(ptraiz)
```

Percurso em árvores binárias

- O **percurso em pós-ordem** é composto dos passos seguintes, para cada uma de suas subárvores:
  - Percorrer sua subárvore esquerda, em pós-ordem;
  - Percorrer sua subárvore direita, em pós-ordem.
  - Visitar a raiz;



- Para a árvore da figura ao lado, o percurso em ordem simétrica para impressão de nós fornece a seguinte

Algoritmo 27. Percurso em pós-ordem

```
procedimento pos(pt)
    se pt*.esq != lambda então pos(pt*.esq)
    se pt*.dir != lambda então pos(pt*.dir)
    visita(pt)

se prtaiz != lambda então pos(ptraiz)
```

Cálculo da altura dos nós

- O cálculo da altura de todos os nós de uma árvore binária é uma aplicação do percurso em pós-ordem.

- A altura das folhas, pela própria definição, é 1. Para os outros nós, por exemplo v, é necessário conhecer o comprimento do maior caminho de v até um de seus descendentes. Isto equivale dizer que a altura de v deve ser calculada após a visita a seus descendentes.

- O algoritmo a seguir mostra a implementação do procedimento **visita(pt)**, que executa a tarefa de determinar a altura do nó apontado por **pt**. Considera-se **altura** um campo do nó da árvore. As variáveis auxiliares **alt1** e **alt2** armazenam, respectivamente, as alturas das subárvores esquerda e direita do nó em questão. A altura desejada corresponderá à maior altura dentre as de suas duas subárvores incrementada de um.

Cálculo da altura dos nós

Algoritmo 28. Cálculo da altura de um nó da árvore binária

```
procedimento visita(pt)
    se pt*.esq != lambda então
        alt1 := (pt*.esq)*.altura
    senão alt1 := 0

    se pt*.dir != lambda então
        alt2 := (pt*.dir)*.altura
    senão alt2 := 0

    se alt1 > alt2 então
        pt*.altura := alt1 + 1
    senão pt*.altura := alt2 + 1
```

Árvores Binárias de Busca

Introdução

- Dado um conjunto de elementos, onde cada um é identificado por uma chave, o objetivo é localizar nesse conjunto o elemento correspondente a uma chave específica procurada.

- No presente capítulo será visto um método de solução que emprega a árvore binária como estrutura na qual se processa a busca. Ou seja, os elementos do conjunto são previamente distribuídos pelos nós de uma árvore de forma conveniente. A localização da chave desejada é então obtida através de um caminhamento apropriado na árvore.

- É importante ressaltar, mais uma vez, a relevância desse problema na área de computação, em especial nas aplicações não numéricas. Sem dúvida, a operação de busca é uma das mais frequentemente realizadas.

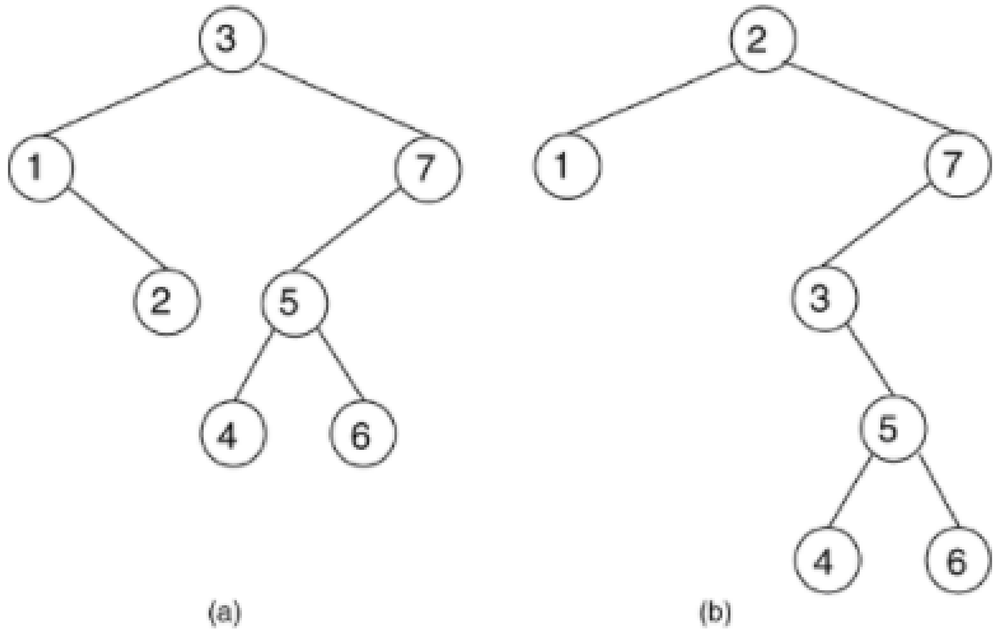
Conceitos básicos

- Seja $S = \{s_1, \ldots, s_n\}$ o conjunto de chaves satisfazendo $s_1 < \cdots < s_n$. Seja x um valor dado. O objetivo é verificar se $x \in S$ ou não. Em caso positivo, localizar x em S, isto é, determinar o índice j tal que $x = s_j$.

- Para resolver esse problema, emprega-se uma árvore binária rotulada T, com as seguintes características:
  - T possui n nós. Cada nó v corresponde a uma chave distinta $s_j \in S$ e possui como rótulo o valor $rt(v) = s_j$;
  - Seja um nó v de T. Seja também v_1, pertencente à subárvore esquerda de v. Então $rt(v_1) < rt(v)$. Analogamente, se v_2 pertence à subárvore direita de v, $rt(v_2) > rt(v)$.

Conceitos básicos

- A árvore T denomina-se **árvore binária de busca** para S. Naturalmente, se $|S| > 1$, existem várias árvores de busca para S. A figura abaixo ilustra duas dessas árvores para o conjunto $\{1, 2, 3, 4, 5, 6, 7\}$.



Busca

- O algoritmo seguinte implementa a ideia. Suponha que a árvore esteja armazenada da forma habitual, isto é, para cada nó v, esq e dir designam os campos que armazenam ponteiros para os filhos esquerdo e direito de v, respectivamente. A raiz da árvore é apontada por $ptraiz$. A variável f designa a natureza final da busca. Tem-se, então:
  - $f = 0$, se a árvore é vazia;
  - $f = 1$, se $x \in S$. Nesse caso, pt aponta para o nó procurado;
  - $f > 1$, se $x \notin S$.

Busca

Algoritmo 29. Busca em árvore binária de busca

```
procedimento busca-arvore(x, pt, f)
    se pt = lambda então f := 0
    senão se x = pt*.chave então f := 1
    senão se x < pt*.chave então
        se pt*.esq = lambda então f:= 2
        senão
            pt := pt*.esq
            busca-arvore(x, pt, f)
    senão
        se pt*.dir = lambda então f := 3
        senão
            pt := pt*.dir
            busca-arvore(x, pt, f)

pt := ptraiz
busca-arvore(x, pt, f)
```

Inserção

- Para resolver o problema de inserção de nós na árvore de busca T, utiliza-se também o procedimento *busca-arvore*. Seja x o valor da chave que se deseja inserir em T e $novo - valor$ a informação associada a x. A ideia inicial é verificar se $x \in S$. Em caso positivo, trata-se de uma chave duplicata e a inserção não pode ser realizada. Se $x \notin S$, a chave de valor x será o rótulo de algum novo nó w, situado à esquerda ou à direita de v, para $f = 2$ ou $f = 3$, respectivamente, de acordo com o procedimento *busca-arvore*. O algoritmo seguinte descreve o processo.

Inserção

Algoritmo 30. Inserção em árvore binária de busca

```
pt := ptraiz
busca-arvore(x, pt, f)
se f = 1 então
    "inserção inválida"
senão
    ocupar(pt1)
    pt1*.chave := x
    pt1*.info := novo-valor
    pt1*.esq := lambda
    pt1*.dir := lambda
    se f = 0 então
        ptraiz := pt1
    senão se f = 2 então
        pt*.esq := pt1
    senão pt*.dir := pt1
```

Árvores Balanceadas

Introdução

- Um aspecto fundamental do estudo de árvores de busca é, naturalmente, o custo de acesso a uma chave desejada. Com o intuito de minimizar esse custo, foram desenvolvidas as árvores binárias de busca e de partilha ótimas. Ambas, porém, se restringem a aplicações estáticas. Isto é, após um certo número de inserções e remoções as árvores deixam de ser ótimas. Além disso, a complexidade da árvore de partilha ótima é muito elevada.

- Para estrutura em que as probabilidades de acesso são idênticas entre si, há uma alternativa. A ideia é manter o custo de acesso na mesma ordem de grandeza de uma árvore ótima, ou seja, $O(\log n)$. Esse custo deve se manter ao longo de toda utilização da estrutura, inclusive após inclusões e remoções. Para alcançar essa finalidade, a estrutura deve ser alterada, periodicamente, de forma a se moldar aos novos dados, mantendo o custo em $O(\log n)$. Uma estrutura que opera com essas características é denominada **balanceada**.

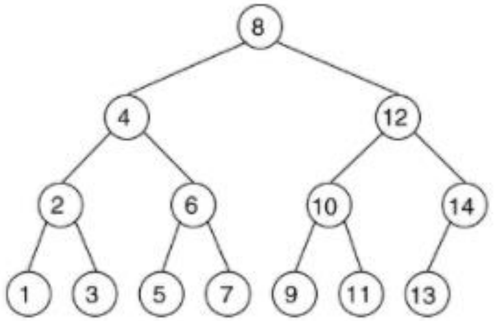
O Conceito de Balanceamento

- As árvores completas são aquelas que minimizam o número de comparações efetuadas no pior caso para uma busca com chaves de probabilidades de ocorrência idênticas.

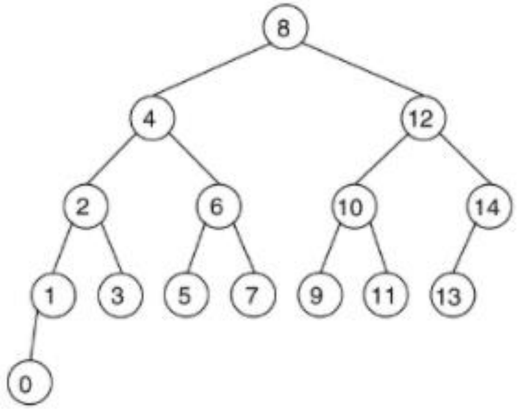
- Do ponto de vista das aplicações dinâmicas, contudo, o uso de árvores completas é, em geral, desaconselhável. Em um caso extremo, ela pode inclusive degenerar-se em uma lista.

- Para contornar esse problema, uma ideia seria aplicar um algoritmo que tornasse a árvore novamente completa, tão logo tal característica fosse perdida após uma inclusão ou exclusão. A dificuldade reside em como efetuar essa operação de forma ótima.

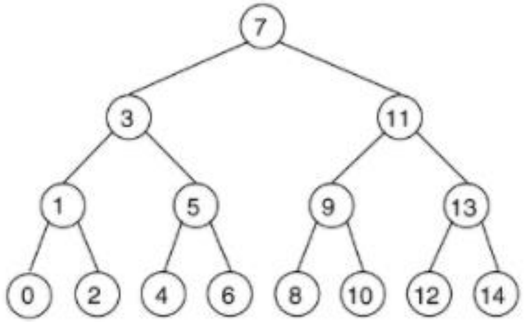
O Conceito de Balanceamento



(a)



(b)



(c)

O Conceito de Balanceamento

- Para efetuar essas transformações usuais de árvores binárias é necessário percorrer todos os nós da árvore. Isso implica que o algoritmo de restabelecimento da estrutura requer, pelo menos, $O(n)$ passos.

- Naturalmente, este custo é considerado excessivo, considerando que operações como inserção ou remoção seriam efetuadas em $O(\log n)$ passos. Por esse motivo, as árvores completas (e a busca binária) não são recomendadas para aplicações que requeiram estruturas dinâmicas.

- Além disso, é desejável que esta propriedade se estenda a todas as subárvores: cada subárvore que contém *m* nós deve possuir altura igual a $O(\log m)$. Uma árvore que satisfaça essa condição é denominada **balanceada**.

Árvores AVL

- O nome dessa estrutura deriva dos seus criadores, Adelson Velsky e Landis. Uma árvore binária *T* é denominada AVL quando, para qualquer nó de *T*, as alturas de suas duas subárvores, esquerda e direita, diferem em módulo de até uma unidade. Nesse caso, *v* é um nó **regulado**. Em contrapartida, um nó que não satisfaça essa condição de altura é denominado **desregulado**, e uma árvore que contenha um nó nessas condições é também chamada **desregulada**. Naturalmente, toda árvore completa é AVL, mas não necessariamente vale a recíproca.

- Em outras palavras, uma árvore AVL é tal que, nas inserções e remoções, procura-se executar uma rotina de balanceamento tal que as alturas das subárvores esquerda e direita tenham alturas bem próximas.

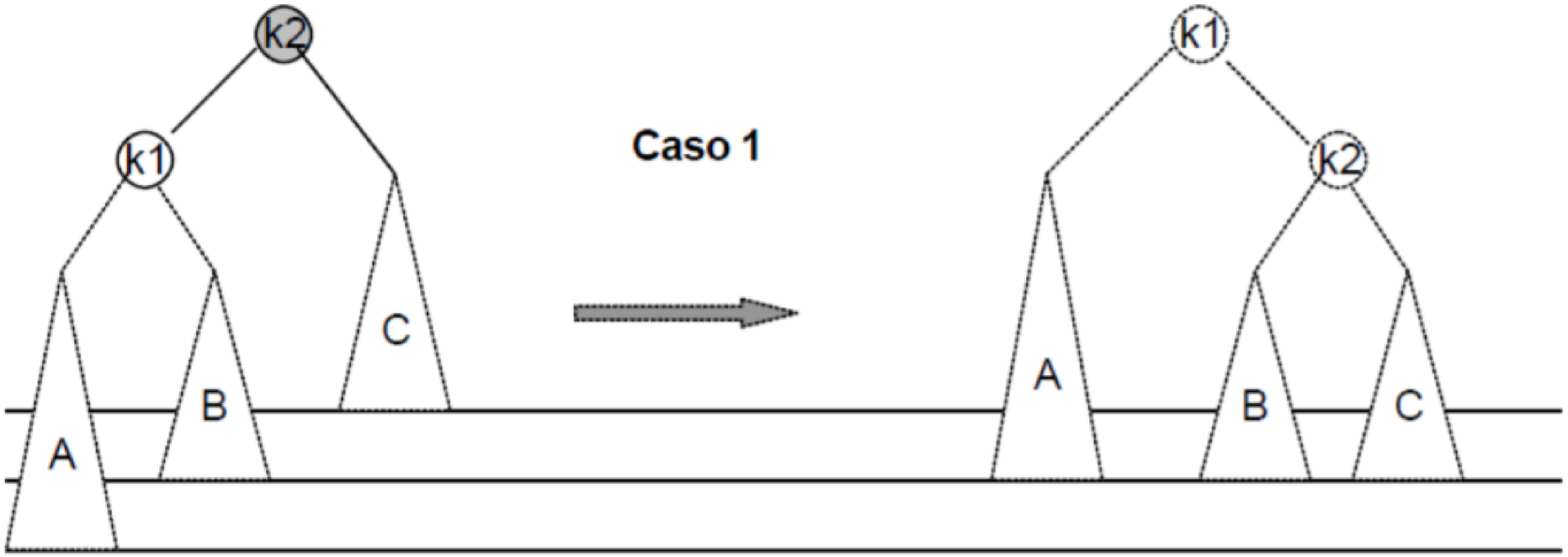
Árvores AVL

- Em uma árvore AVL, cada nó recebe um rótulo adicional que indica o seu fator de balanceamento. Esse rótulo pode ter os valores -1, 0 e 1. Se o fator ficar abaixo de -1 ou acima de 1, então a árvore precisa ser balanceada.

- No processo de inserção dos nós podem ser realizadas quatro tipos possíveis de transformações na árvore, para obter o seu balanceamento:
  - Rotação direita
  - Rotação esquerda
  - Rotação dupla direita
  - Rotação dupla esquerda

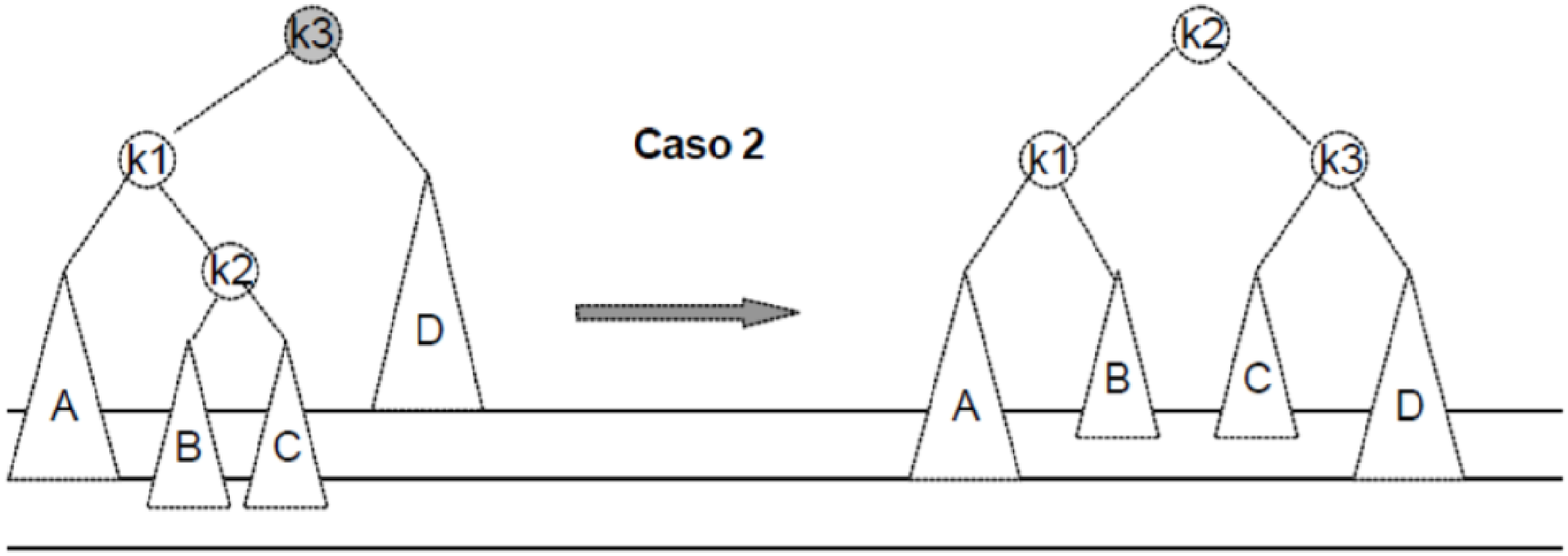
Árvores AVL

- Rotação simples
  - k2 é o nó mais profundo onde falha o equilíbrio
  - Subárvore esquerda está 2 níveis abaixo da direita



Árvores AVL

- Rotação dupla
  - Uma das subárvores B ou C está 2 níveis abaixo de D
  - k2, a chave intermédia, fica na raiz
  - Posições de k1, k3 e subárvores completamente determinadas pela rotação

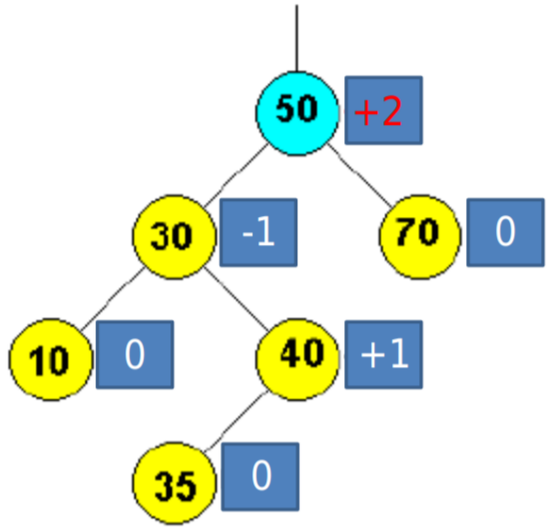


Fator de balanceamento

- Coeficiente que serve como referência para verificar se uma árvore AVL está ou não balanceada.

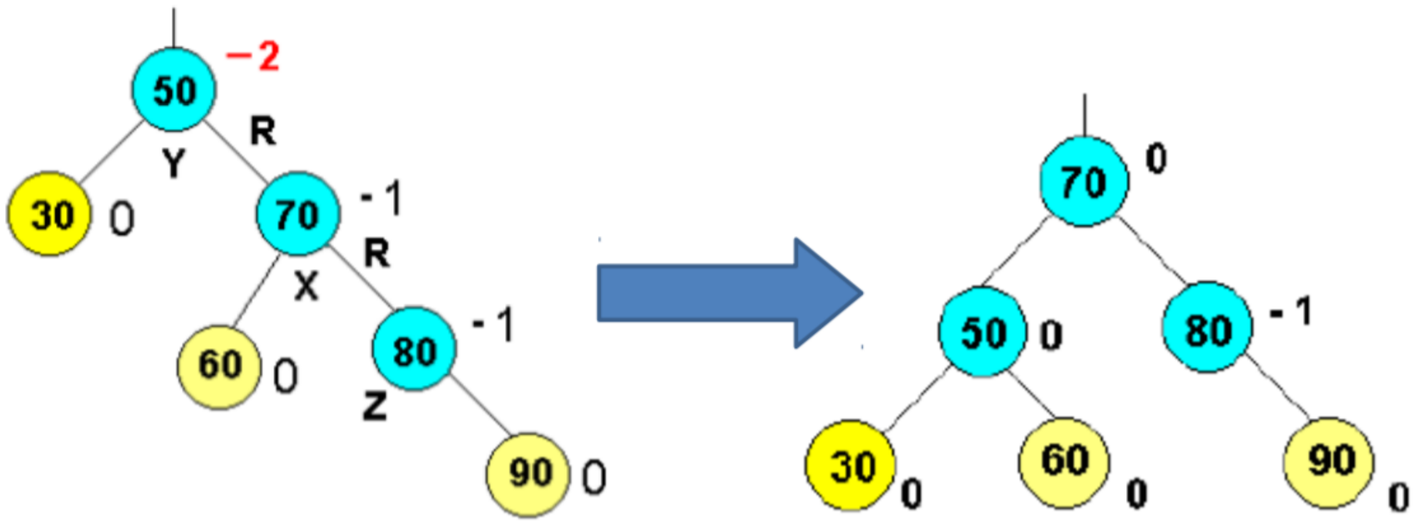
- O fator é calculado nó a nó e leva em consideração a diferença das alturas das subárvores da direita e da esquerda

- $FB = h_e - h_d$

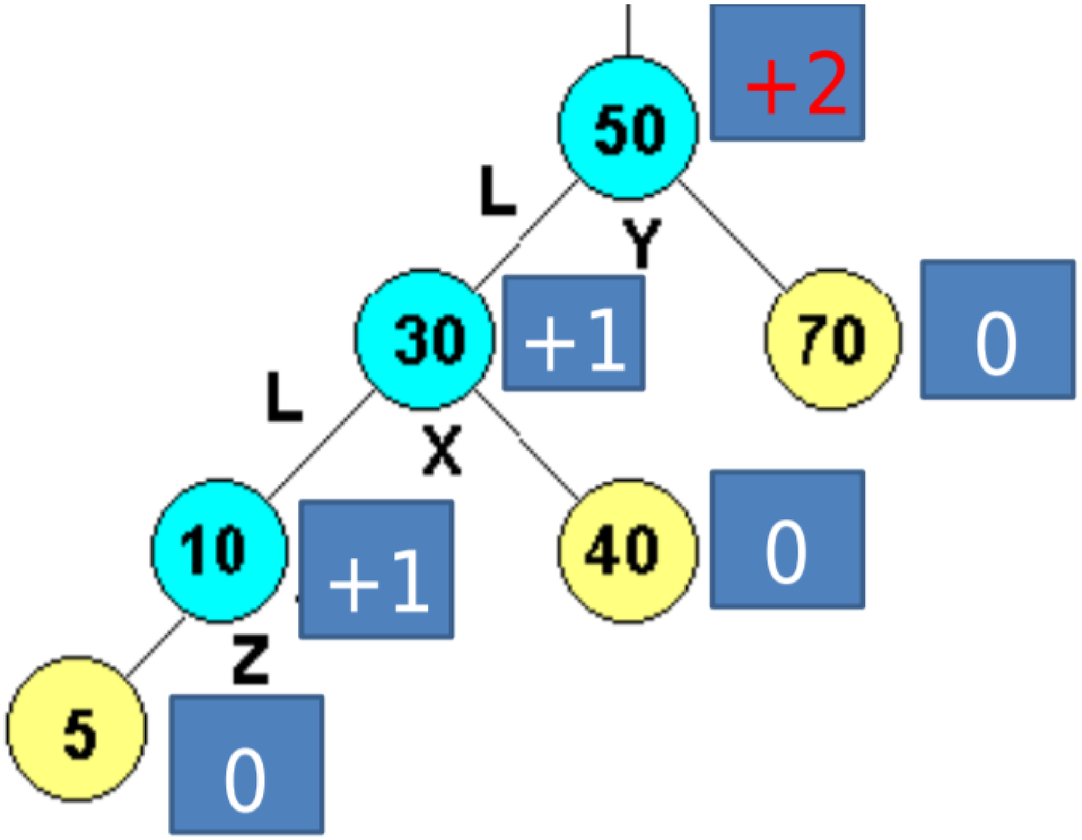


Quando balancear?

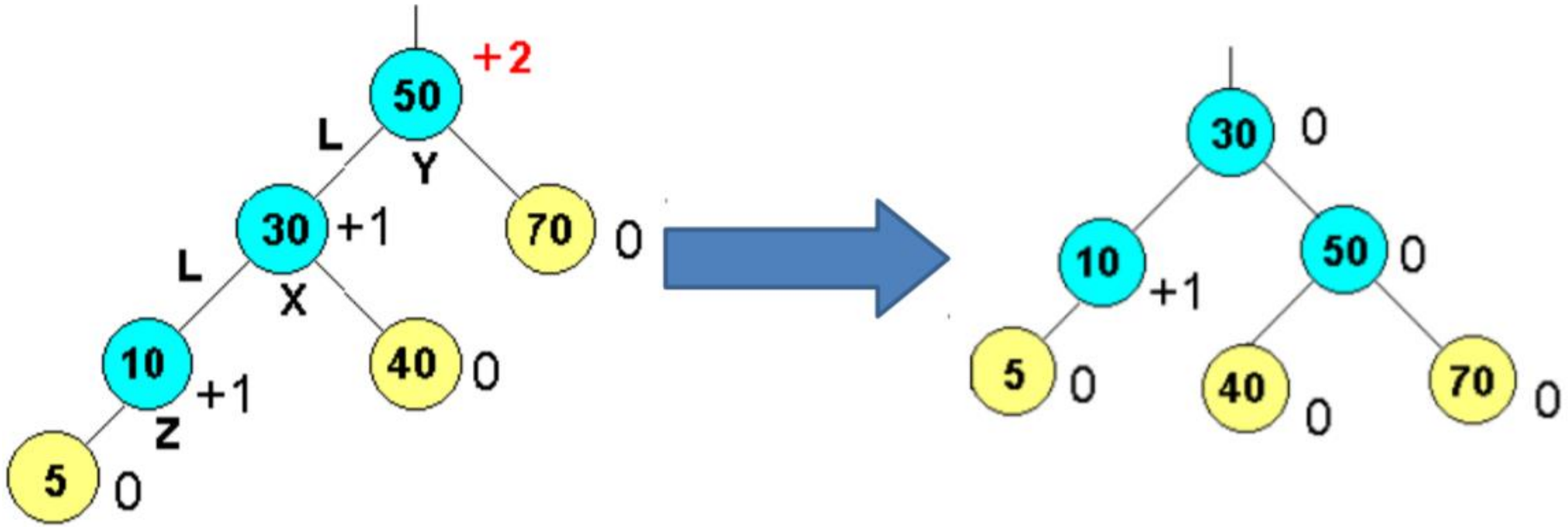
- Sempre que existir um fator de balanceamento superior a +1 ou inferior a -1
- Caso exista mais de um nó que se encaixe neste perfil deve-se sempre balancear o nó com o nível mais alto.
- Como balancear? Utilizando rotações simples ou duplas, à esquer



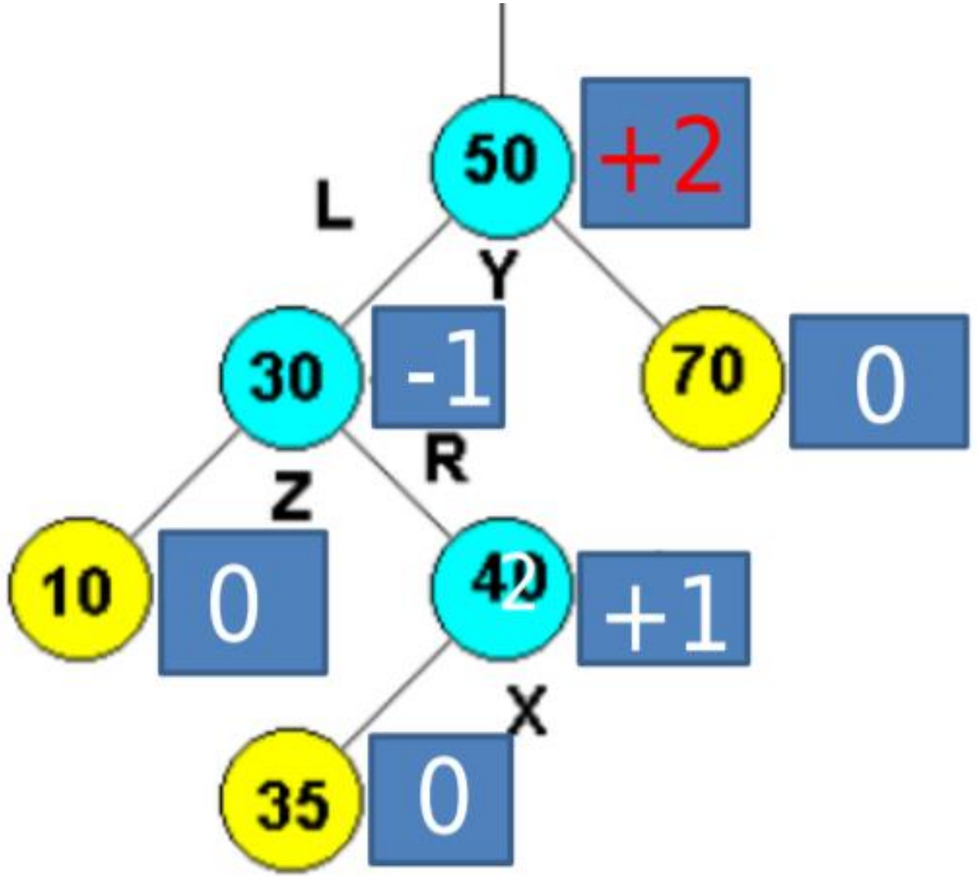
Como balancear?



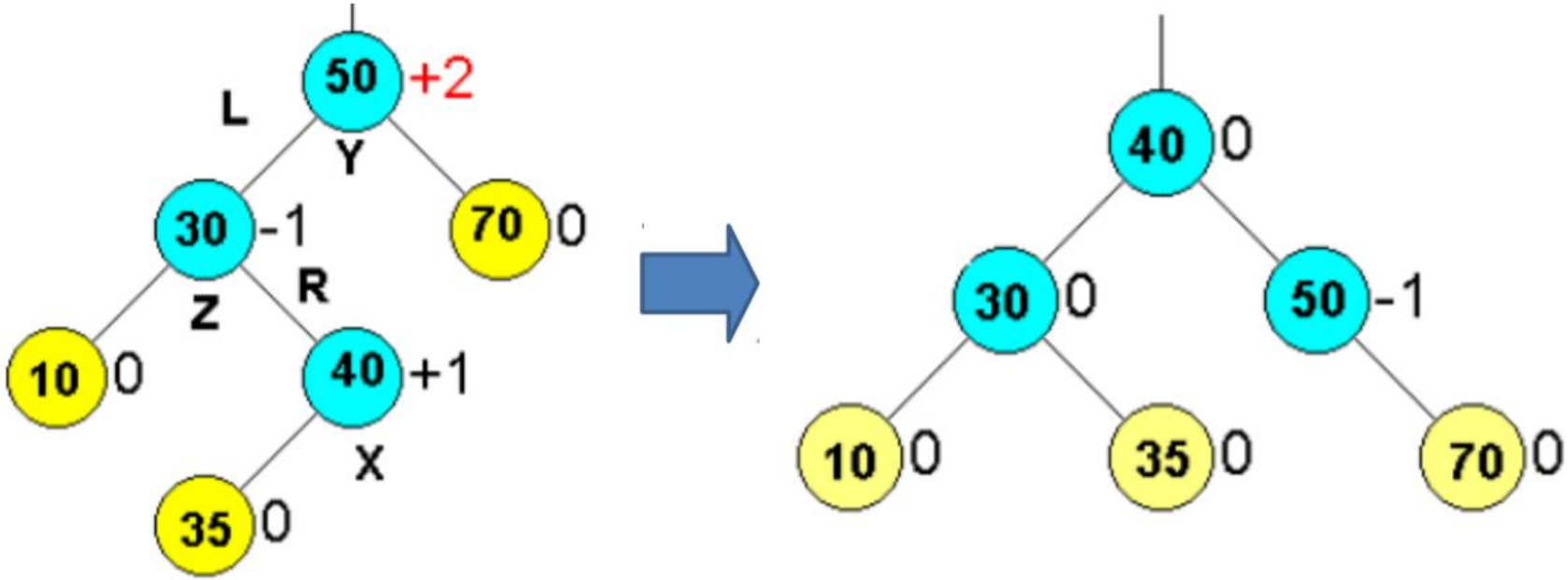
Como balancear?



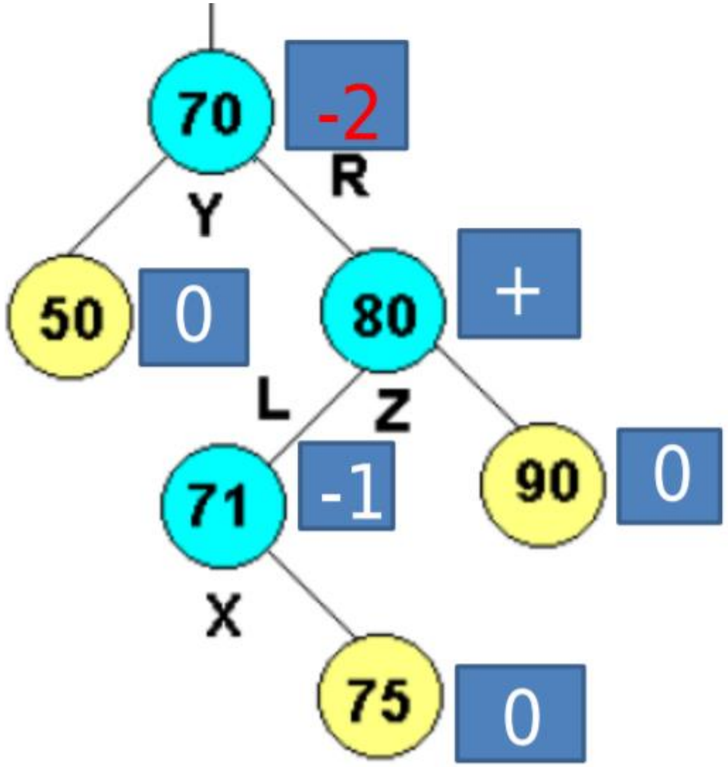
Como balancear?



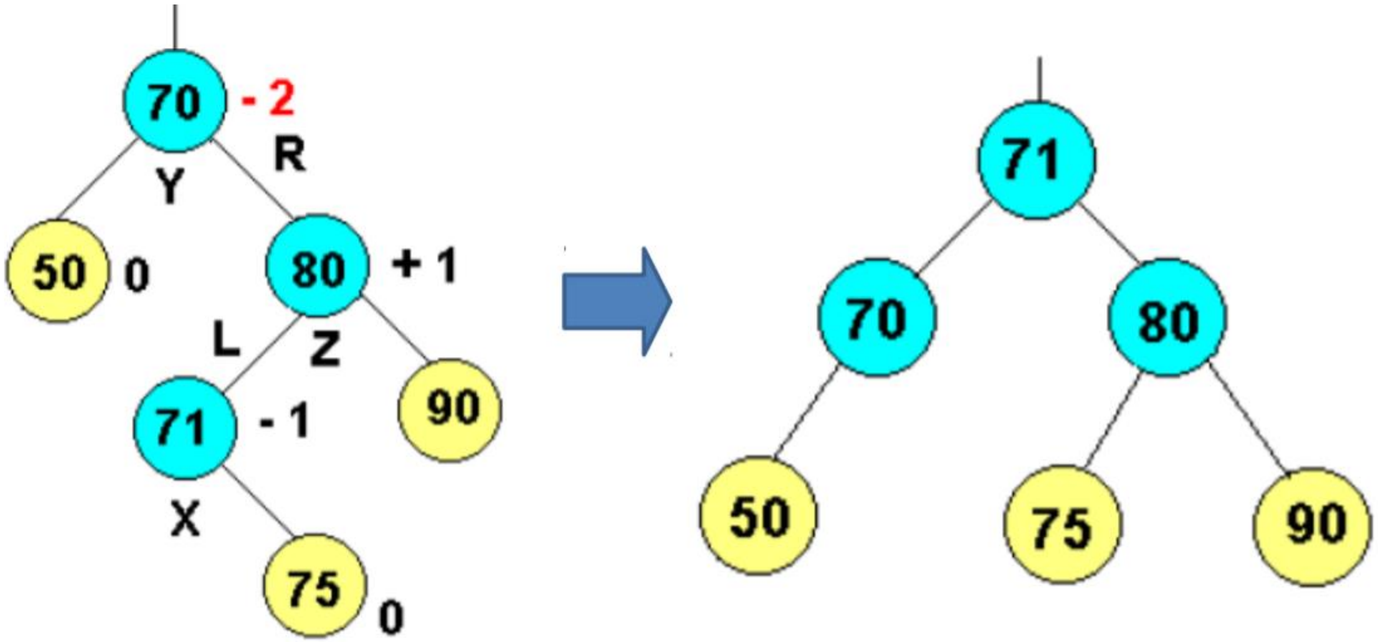
Como balancear?



Como balancear?



Como balancear?



Algoritmo

- Inserir nó com chave X numa árvore A
  - Recursivamente, inserir na subárvore conveniente de A, SA;
  - Se a altura de SA não se modifica, terminar;
  - Se a altura de SA é modificada: se ocorre desequilíbrio em A, fazer as rotações necessárias para reequilibrar.
- Comparação das alturas
  - Para evitar o cálculo repetido de alturas de subárvores, pode-se manter em cada nó o resultado da comparação das alturas das subárvores.

Importante

- Na inserção, caso a árvore AVL esteja desbalanceada, basta 1 operação de rotação para rebalanceá-la.

- Na remoção, caso a árvore esteja desbalanceada, pode ser necessário até $\log n$ operações de rotação.

Algoritmos de Ordenação

Introdução

- Os capítulos anteriores trataram, em geral, de estruturas genéricas, adequadas à representação de quaisquer massas de dados. Neste capítulo serão descritas técnicas distintas para solucionar um problema que aparece como pré-processamento em muitas aplicações que envolvam o uso de tabelas - a obtenção de uma tabela ordenada.

- O problema da ordenação foi um dos primeiros a gerar discussões sobre implementações eficientes ou não. Métodos mais simples, como a ordenação bolha e a ordenação por inserção, apesar de possuírem complexidade de pior caso ruim, são bastante utilizados em razão de sua extrema simplicidade de implementação. Também não pode ser esquecido que esses métodos podem ser convenientes quando a tabela é pequena ou está quase ordenada.

Bubblesort

- O método de ordenação bolha é bastante simples, e talvez seja o método de ordenação mais difundido. Uma iteração do mesmo se limita a percorrer a tabela do início ao fim, sem interrupção, trocando de posição dois elementos consecutivos sempre que estes se apresentarem fora de ordem. Intuitivamente percebe-se que a intenção do método é mover os elementos maiores em direção ao fim da tabela. Ao terminar a primeira iteração pode-se garantir que as trocas realizadas posicionam o maior elemento na última posição. Na segunda iteração, o segundo maior elemento é posicionado, e assim sucessivamente. O processamento é repetido então $n - 1$ vezes. O algoritmo que se segue implementa este método. A tabela se encontra armazenada na estrutura *L*. O algoritmo ordena *L* segundo valores não decrescentes do campo chave.

Bubblesort

Algoritmo 32. Ordenação bolha de uma tabela com n elementos

```
para i = 1, ..., n faça
    para j = 1, ..., n - 1 faça
        se L[j].chave > L[j + 1].chave então
            trocar(L[j], L[j + 1])
```

Bubblesort

- O algoritmo anterior é claramente ruim. Sua complexidade de pior caso é igual à de melhor caso, $O(n^2)$, devido aos percursos estipulados para as variáveis i e j. Pode-se, entretanto, pensar em alguns critérios de parada que levariam em consideração comparações desnecessárias, isto é, comparações executadas em partes da tabela sabidamente já ordenadas:
  - Uma variável lógica **mudou** é introduzida com a finalidade de sinalizar se pelo menos uma troca foi realizada. Caso isso não ocorra, o algoritmo pode ser encerrado. Essa simples alteração afeta a complexidade de melhor caso do algoritmo, que passa a ser O(n), uma vez que, se a tabela já está ordenada, apenas um percurso é realizado.
  - A posição da última troca (armazenada, no algoritmo, na variável guarda) indica que todos os elementos posteriores já estão ordenados. O algoritmo pode então utilizar esta posição para atualizar o limite superior da tabela, que inicialmente é o próprio número de elementos.

Bubblesort

Algoritmo 33. Ordenação bolha com critério de parada

```
mudou := Verdadeiro
n' := n - 1
guarda := n - 1
enquanto mudou faça
    j := 0
    mudou := Falso
    enquando j < n' faça
        se L[j].chave > L[j + 1].chave então
            trocar(L[j], L[j + 1])
            mudou := Verdadeiro
            guarda := j
        j := j + 1
    n' := guarda
```

Iteração	Tabela	Trocas
tabela inicial	40 37 95 42 39 51 60	
após 1ª iteração	37 40 42 39 51 60 95	5
após 2ª iteração	37 40 39 42 51 60 95	1
após 3ª iteração	37 39 40 42 51 60 95	1
após 4ª iteração	37 39 40 42 51 60 95	0

Insertion Sort

- O método de ordenação por inserção é também bastante simples, sendo sua complexidade equivalente à da ordenação bolha. Imagine uma tabela já ordenada até o i-ésimo elemento. A ordenação da tabela pode ser estendida até o $(i+1)$-ésimo elemento por meio de comparações sucessivas deste com os elementos anteriores, isto é, com o i-ésimo elemento, com o $(i+1)$-ésimo elemento etc., procurando sua posição correta na parte da tabela que já está ordenada. Pode-se então deduzir um algoritmo para implementar o método: considera-se sucessivamente todos os elementos, a partir do segundo deles, em relação à parte da tabela formada pelos elementos anteriores ao elemento considerado em cada iteração.

Insertion Sort

Iteração	Tabela	Trocas
tabela inicial	40 37 95 42 23 51 27	
após $i = 2$	37 40 95 42 23 51 27	1
após $i = 3$	37 40 95 42 23 51 27	0
após $i = 4$	37 40 42 95 23 51 27	1
após $i = 5$	23 37 40 42 95 51 27	4
após $i = 6$	23 37 40 42 51 95 27	1
após $i = 7$	23 27 37 40 42 51 95	5

Insertion Sort

- O algoritmo abaixo apresenta a ordenação por inserção da tabela *L*, de *n* elementos, segundo o seu campo chave. Para evitar erros de implementação, a tabela deve ser acrescida de uma posição $L[0]$, que pode receber em seu campo chave qualquer valor. Esse valor será testado no caso em que a posição definitiva do elemento que está sendo analisado seja a primeira. Essa comparação, entretanto, não tem efeito, uma vez que, nesse caso, $j < 1$ e o teste resulta falso.

Algoritmo 34. Ordenação por inserção de uma tabela com n elementos

```
para i = 1, ..., n - 1 faça
    aux := L[i]
    valor := L[i].chave
    j := i - 1
    enquanto j >= 0 e valor < L[j].chave faça
        L[j + 1] := L[j]
        j := j - 1
    L[j + 1] := aux
```

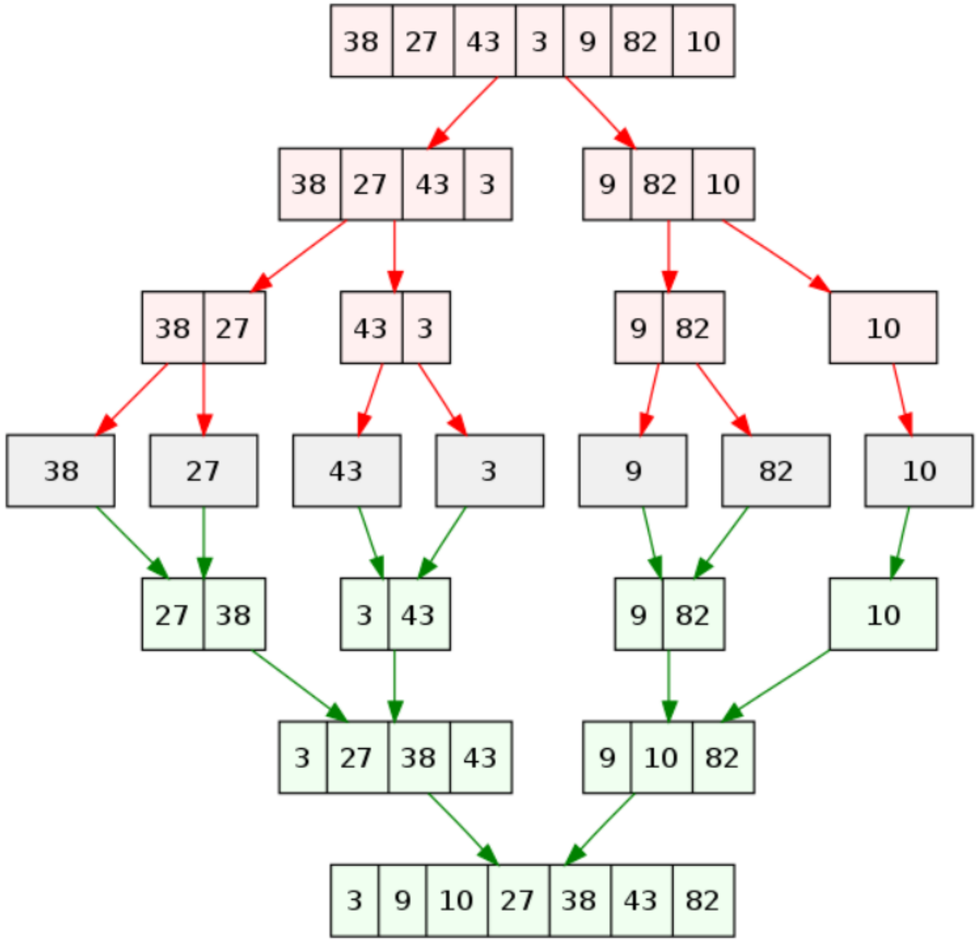
Mergesort

- Este método tem como procedimento básico o de intercalação de listas.

- A ideia básica do método é intercalar as duas metades da lista desejada quando estas já se encontram ordenadas. Deseja-se então ordenar primeiramente as duas metades, o que pode ser feito utilizando recursivamente o mesmo conceito.

- Sejam duas listas *A* e *B*, ordenadas, com respectivamente *n* e *m* elementos. As duas listas são percorridas por ponteiros *ptA* e *ptB*, armazenando o resultado da intercalação na lista *C*, apontada pelo ponteiro *ptC*. O primeiro elemento de *A* é comparado com o primeiro elemento de *B*; o menor valor é colocado em *C*. O ponteiro da lista onde se encontra o menor valor é incrementado, assim como o ponteiro da lista resultado; o processo se repete até que uma das listas seja esgotada. Neste ponto, os elementos restantes da outra lista são copiados na lista resultado.

Mergesort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Mergesort

Algoritmo 35. Ordenação por intercalação de uma tabela com n elementos

```
procedimento mergesort(esq, dir)
    se esq < dir então
        centro := |__(esq + dir) / 2__|
        mergesort(esq, centro)
        mergesort(centro + 1, dir)
        intercalar(esq, centro + 1, dir)
```

```
procedimento intercalar(L, ini1, ini2, fim2)
    fim1 := ini2 - 1
    ind := ini1
    tmp = COPIA(L)
    enquanto (ini1 <= fim1) e (ini2 <= fim2) faça
        se L[ini1].chave <= L[ini2].chave então
            tmp[ind] := L[ini1]
            ini1 := ini1 + 1
        senão
            tmp[ind] := L[ini2]
            ini2 := ini2 + 1
        ind := ind + 1
    enquanto ini1 <= fim1 faça
        tmp[ind] := L[ini1]
        ini1 := ini1 + 1
        ind := ind + 1
    enquanto ini2 <= fim2 faça
        tmp[ind] := L[ini2]
        ini2 := ini2 + 1
        ind := ind + 1
    para i := 0, ..., fim2 faça
        L[i] := tmp[i]
```

```
mergesort(1, n)
```

Quicksort

- O nome *quicksort* (ordenação rápida) já indica o que se deve esperar do método, que é, na realidade, um dos mais eficientes dentre os conhecidos. Dada uma tabela *L* com *n* elementos, o procedimento recursivo para ordenar *L* consiste nos seguintes passos:
  - Se $n = 0$ ou $n = 1$ então a tabela está ordenada;
  - Escolha qualquer elemento x em L – este elemento é chamado **pivô**;
  - Separe $L - \{x\}$ em dois conjuntos de elementos disjuntos: $S_1 = \{w \in L - \{x\} \mid w < x\}$ e $S_2 = \{w \in L - \{x\} \mid w \geq x\}$;
  - O procedimento de ordenação é chamado recursivamente para S_1 e S_2;
  - *L* recebe a concatenação de S_1, seguido de x, seguido de S_2.

Quicksort

- Dois pontos são decisivos para o bom desempenho do algoritmo:
  - **Escolha do pivô:** Uma solução utilizada com bons resultados é a escolha da mediana dentre três elementos: o primeiro, o último e o central.
  - **Particionamento da tabela:**
    - O pivô é afastado da tabela a ser percorrida; isto pode ser feito colocando-o na última posição e considerando somente o restante da tabela;
    - Em seguida, dois ponteiros são utilizados:
      - i é inicializado apontando para o primeiro elemento da tabela, percorrendo-a enquanto os valores apontados são menores do que o pivô;
      - j é inicializado na penúltima posição, efetuando a tarefa inversa.
    - Os percursos são interrompidos quando i aponta para um elemento maior do que o pivô e j aponta para um elemento menor do que o pivô.

Quicksort

- Dois pontos são decisivos para o bom desempenho do algoritmo:
  - **Particionamento da tabela (cont.):**
    - Duas situações podem ocorrer:
      - Se $i < j$, os elementos da tabela devem ser trocados e o procedimento deve prosseguir;
      - Se $i > j$, a partição já está determinada.
    - O pivô, que se encontra na última posição da tabela, deve ser trocado com o elemento de índice i.
    - Após a troca, os elementos de índice menor do que i formam o conjunto de elementos maiores do que o pivô. Note que o elemento de índice i foi trocado com o elemento que está na última posição da tabela (cuja chave é o pivô), ficando então na parte correta.

Quicksort

40	37	95	42	23	51	27
27	37	95	42	23	51	40
↑					↑	
i					*j*	
27	37	95	42	23	51	40
		↑		↑		
		i		*j*		
27	37	23	42	95	51	40
		↑		↑		
		i		*j*		
27	37	23	42	95	51	40
		↑	↑			
		j	*i*			
[27	37	23]	40	[95	51	42]

Quicksort

Algoritmo 36. Ordenação rápida de uma tabela com n elementos

```
procedimento quicksort(ini, fim)
    se fim - ini < 2 então
        se fim - ini = 1 então
            se L[ini].chave > L[fim].chave então
                trocar(L[ini], L[fim])
    senão
        PIVO(ini, fim, mediana)
        trocar(L[mediana], L[fim])
        i := ini
        j := fim - 1
        chave := L[fim].chave
        enquanto j >= i faça
            enquanto L[i].chave < chave faça
                i := i + 1
            enquanto L[j].chave > chave faça
                j := j - 1
            se j >= i então
                trocar(L[i], L[j])
                i := i + 1
                j := j - 1
        trocar(L[i], L[fim])
        quicksort(ini, i - 1)
        quicksort(i + 1, fim)

quicksort(1, n)
```

Listas de Prioridades

Introdução

- Em muitas aplicações, uma característica importante que distingue os dados de uma certa estrutura é uma prioridade atribuída a cada um deles. Nessas aplicações, em geral, determinar repetidas vezes o dado de maior prioridade é uma operação importante.

- Por exemplo, suponha que os dados de uma tabela correspondam a tarefas a serem realizadas com uma certa prioridade. Nesse caso, deseja-se que as tarefas sejam realizadas em ordem decrescente de prioridades. Além disso, é razoável supor que as prioridades das tarefas possam variar ao longo do tempo. Finalmente, novas tarefas podem ingressar na tabela a cada instante.

- Para encontrar a ordem desejada de execução das tarefas, um algoritmo deve, sucessivamente, escolher o dado de maior prioridade e retirá-lo da tabela. Além disso, o algoritmo deve ser capaz de introduzir novos dados, no momento adequado.

Introdução

- Com essa motivação, pode-se definir lista de prioridades como uma tabela na qual a cada um de seus dados está associada uma prioridade.

- Neste capítulo, será estudado um método considerado o mais eficiente para implementação de uma lista de prioridades, chamado *heap*. Analisaremos os algoritmos de alteração de prioridade, e inclusão e remoção do elemento de maior prioridade, além de um método de ordenação que utiliza essa estrutura de dados como base, o *heapsort*.

- É importante observar que, para maior simplicidade de not

IBMEC.BR

/IBMEC

IBMEC

@IBMEC_OFICIAL

@IBMEC

Ibmec