Árvores

objetivos

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos relativos a árvores:
  - ❒ Nó
  - ❒ Raiz
  - ❒ Folha
  - ❒ Filho de nó
  - ❒ Grau de nó
  - ❒ Grau
  - ❒ Nó interno
  - ❒ Nível de nó
  - ❒ Floresta
  - ❒ Ancestral de nó
  - ❒ Descendente de nó
  - ❒ Profundidade ou altura
- Discernir entre estruturas de dados linear e hierárquica
- Fazer distinção entre árvore binária e árvore ordinária
- Exibir diferenças entre os seguintes tipos de árvores binárias:
  - ❒ Completa
  - ❒ Repleta
  - ❒ Perfeita
  - ❒ 0/2
  - ❒ Inclinada
  - ❒ Patológica
  - ❒ Estritamente binária
  - ❒ Com e sem costura
- Conhecer e demonstrar as principais proposições referentes a árvores binárias
- Descrever a abordagem de numeração de nós para árvores binárias
- Implementar árvores binárias usando ponteiros ou arrays
- Descrever e implementar os caminhamentos infixo, prefixo, sufixo e por nível em árvores binárias
- Representar uma expressão aritmética usando árvore binária
- Construir uma árvore binária com base em dois de seus caminhamentos
- Converter árvores ordinárias e florestas em árvores binárias
- Explicar o algoritmo de codificação de Huffman

UMA ESTRUTURA DE DADOS HIERÁRQUICA é uma forma de organizar dados de tal modo que eles denotem uma hierarquia entre si. Ou seja, um item de uma estrutura de dados hierárquica pode ser considerado superior, inferior ou do mesmo nível que outro. Um exemplo de organização hierárquica aparece em sistemas de arquivos e diretórios (pastas), como aquele usado no sistema operacional Unix apresentado de forma simplificada na **Figura 12–1**. Essa organização hierárquica facilita sobremaneira a localização de arquivos e diretórios. Já imaginou como seria complicado encontrar um dentre milhares de arquivos se todos eles fossem organizados como uma única lista? Existem outras inúmeras aplicações de árvores, dentre as quais se destacam: tomada de decisões (v. **Seção 12.8.1**), jogos, busca (v. **Volume 2**) e codificação (v. **Seção 12.8.2**).

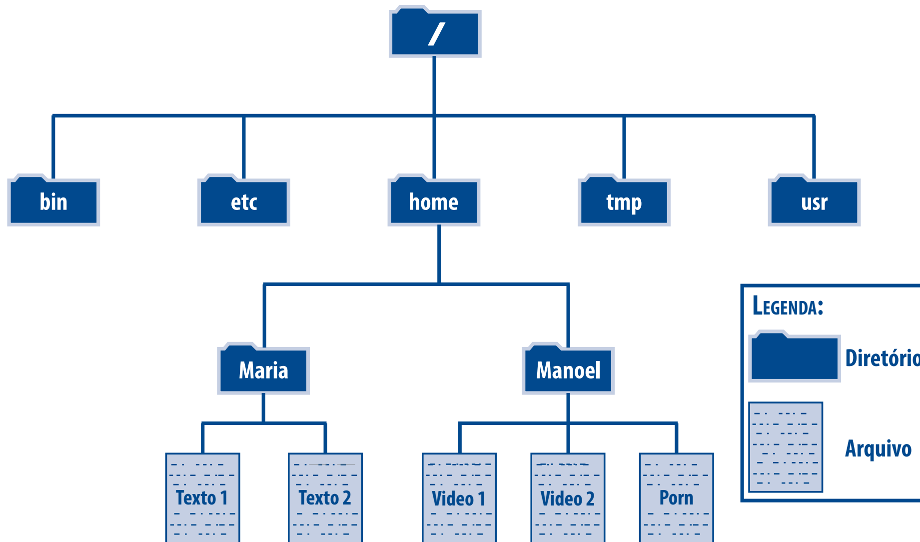


Figura 12–1: Estrutura Hierárquica Simplificada de Arquivos do Sistema Unix

Estruturas de dados hierárquicas contrastam com **estruturas de dados lineares**. Numa estrutura de dados linear, cada elemento, com exceção do primeiro, tem um único antecessor e cada elemento, com exceção do último, tem um único sucessor. Esse é o caso de todas as estruturas de dados apresentadas nos capítulos anteriores. Este capítulo é dedicado ao estudo introdutório de árvores, que são as estruturas de dados hierárquicas mais utilizadas em programação.

12.1 Conceitos Fundamentais

Definição 12.1 Uma **árvore** (**ordinária**) é um conjunto finito constituído de um ou mais nós, de modo que há um nó especial chamado **raiz** e os nós remanescentes são divididos em $n \geq 0$ conjuntos disjuntos $A_1, A_2, \ldots, A_n$, onde cada um desses conjuntos é uma árvore (ou, mais apropriadamente, uma **subárvore**).

Uma árvore pode ser representada esquematicamente como na **Figura 12–2**. Nessa figura, `A` é a raiz da árvore, que, por sua vez, possui três subárvores cujas raízes são `B`, `C` e `D`. Como você deve ter percebido, normalmente, a raiz de uma árvore é desenhada no topo da figura que representa a árvore.

A exigência de que os conjuntos (subárvores) na definição de árvore sejam disjuntos impede que haja ligação entre uma subárvore e outra (por exemplo, uma ligação entre `D` e `F` na **Figura 12–2** não é permitida).

Algumas terminologias frequentemente utilizadas com árvores serão apresentadas a seguir, mas antes disso, um alerta se faz necessário. A terminologia usada com árvores é bem variada e, infelizmente, não é padronizada, de modo que ela varia muito entre autores. Aqui, será adotada a nomenclatura que se julga ser a mais comum.

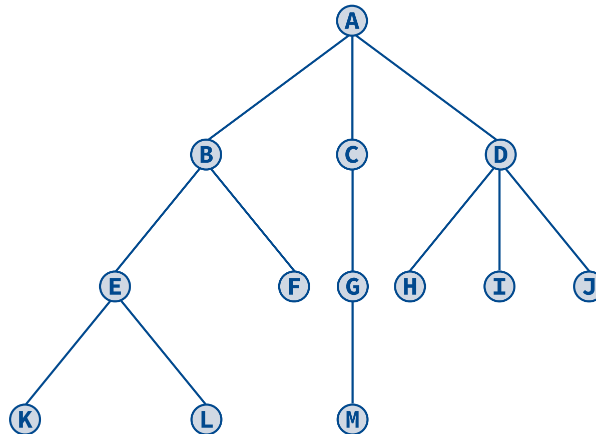


Figura 12–2: Representação Gráfica de Árvore Ordinária

Um **nó** de uma árvore consiste num item de informação mais as ramificações para outros itens. A árvore da **Figura 12–2** possui 13 nós, cujos campos de informação são representados por letras e as ramificações por segmentos de reta. O **grau de um nó** é o número de subárvores do nó. Na **Figura 12–2**, o grau do nó `A` é 3, o grau do nó `B` é 2 e o grau do nó `J` é zero. Os nós cujos graus são iguais a zero (i.e., aqueles que não possuem nenhuma subárvore) são denominados **nós folha** (ou, apenas, **folhas**) ou **nós terminais**. Os nós que não são folhas são denominados de **nós internos** (ou **nós não terminais**). As raízes das subárvores de um nó `X` são denominadas **filhos** de `X`. Neste último caso, `X` é chamado de **pai** de seus nós filhos. Assim, no exemplo acima, os filhos de `B` são `E` e `F`, e o pai de `B` é `A`. Os filhos de um mesmo pai são ditos serem **irmãos**. O **grau de uma árvore** é o maior dentre os graus de todos os nós da árvore. O grau da árvore do exemplo acima é 3. Os **ancestrais** de um nó são todos os nós ao longo do caminho que vai da raiz até esse nó (sem incluí-lo). No exemplo acima, os ancestrais de `L` são `A`, `B` e `E`. Os **descendentes** de um nó são todos os nós que o têm como ancestral. Por exemplo, os descendentes do nó B na árvore da **Figura 12–2** são os nós `E`, `F`, `K` e `L`.

Definição 12.2 O **nível** de um nó x é definido conforme aparece a seguir, mas alguns autores consideram o nível da raiz como sendo *0*.

$$nível(x) = \begin{cases} 1, \text{ se } x \text{ é a raiz} \\ n+1, \text{ se o nível do pai de } x \text{ é } n \end{cases}$$

Na árvore da **Figura 12–2**, o nível do nó `A` é 1, o nível dos nós `B`, `C` e `D` é 2, e assim por diante. A **profundidade** ou **altura** de uma árvore é o nível máximo encontrado em algum nó da árvore. A altura da árvore da **Figura 12–2** é 4 (correspondendo ao nível dos nós `K`, `L` e `M`).

Definição 12.3 Uma **floresta** é definida como sendo um conjunto de n ≥ 0 árvores disjuntas. Observe que a noção de floresta é muito próxima da própria noção de árvore. Por exemplo, se a raiz da árvore do exemplo acima fosse removida, ter-se-ia uma floresta com três árvores, cujas raízes seriam `B`, `C` e `D`. Inversamente, se as raízes das árvores que formam uma floresta forem unidas por um nó adicional ter-se-ia uma árvore.

Há várias formas de se representar graficamente uma árvore, além daquela vista na **Figura 12–2**. Uma forma útil é a representação em forma de lista. Neste esquema, a árvore dessa figura seria representada como:

```
(A(B(E(K,L),F),C(G),D(H(M),I,J)))
```

Outra forma menos utilizada para representar graficamente uma árvore é usando endentação para refletir as relações entre os nós. Usando esse tipo de representação a árvore do exemplo seria exibida como na **Figura 12–3**.

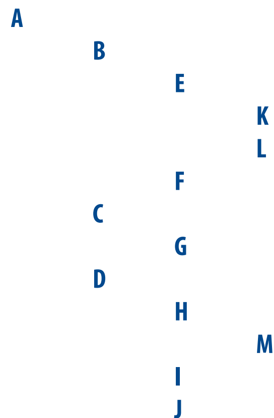


Figura 12–3: Representação de Árvore Ordinária em Forma de Lista

Construir a representação da **Figura 12–3** usando um editor de texto é relativamente trivial. Usa-se tabulação para representar o nível de cada nó, considerando a raiz como nível 0. Assim, considerando a árvore do exemplo, o nó `A` não terá tabulação; os nós `B`, `C` e `D` terão uma tabulação; os nós `E`, `F`, `G`, `H`, `I` e `J` terão duas tabulações, e assim por diante.

12.2 Árvores Binárias

Informalmente, uma **árvore binária** é uma estrutura de dados do tipo árvore (i.e., hierárquica), onde o grau de cada nó é no máximo igual a dois (em outras palavras, cada nó tem no máximo duas ramificações). Além disso, numa árvore binária, faz-se distinção entre a **subárvore da esquerda** e a **subárvore da direita** de um nó. Uma árvore binária também pode ser vazia, i.e., não possuir nenhum nó. A definição formal de árvore binária será apresentada a seguir.

Definição 12.4 Uma árvore binária é um conjunto de nós que é vazio ou consiste numa raiz e duas árvores binárias disjuntas denominadas de subárvore direita e subárvore esquerda.

Compare as definições de árvore ordinária e de árvore binária e observe que existem diferenças entre elas. Note que pode existir árvore binária vazia, enquanto não pode existir árvore ordinária vazia. Note ainda que as duas árvores binárias apresentadas na **Figura 12–4** são diferentes. A primeira delas tem uma subárvore direita vazia e a segunda tem uma subárvore esquerda vazia. Se estas estruturas fossem consideradas como árvores ordinárias, em vez de árvores binárias, elas seriam iguais.

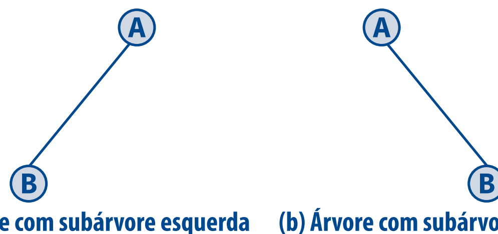


Figura 12–4: Duas Árvores Binárias Diferentes com Dois Nós

A terminologia vista antes para árvores (grau, nível, altura, pai, filho etc.) aplica-se naturalmente às árvores binárias.

Alguns resultados importantes relativos a árvores binárias serão vistos em seguida.

Teorema 12.1 **(i)** O número máximo de nós no nível i de uma árvore binária é 2^{i-1}, $i \geq 1$. (ii) O número máximo de nós numa árvore binária de altura p é $2^p - 1$, $p \geq 1$.

Prova: A prova deste e dos demais teoremas enunciados neste capítulo encontram-se no **Apêndice B**.

Teorema 12.2 Para qualquer árvore binária, se n_0 é o número de nós terminais (folhas) e n_2 é o número de nós de grau 2, então $n_0 = n_2 + 1$.

12.2.1 Degeneradas (ou Patológicas)

Definição 12.5 Uma árvore binária **degenerada** ou **patológica** é aquela na qual cada nó possui um filho e só há uma folha.

Um exemplo de árvore binária degenerada é mostrada na **Figura 12–5**. Essa é uma árvore binária **inclinada à esquerda**.

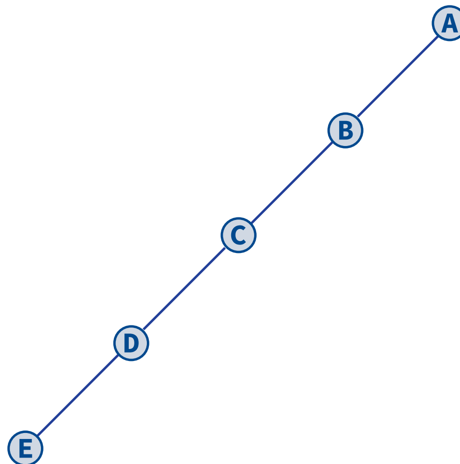


Figura 12–5: Árvore Binária Inclinada à Esquerda

12.2.2 Estritamente Binárias

Definição 12.6 Uma **árvore estritamente binária** (ou **árvore 0/2**) é aquela na qual cada nó tem zero ou dois filhos.

Teorema 12.3 Se n é o número de nós de uma árvore estritamente binária e p é a sua profundidade, então a seguinte relação é válida: $2p - 1 \leq n \leq 2^p - 1$ $(p > 0)$.

Teorema 12.4 Numa árvore estritamente binária, o número total de nós n é dado por: $n = 2n_0 - 1$, sendo n_0 o número de folhas da árvore

12.2.3 Perfeitas (ou Repletas)

Definição 12.7 Uma árvore binária de profundidade p que possui o número máximo possível de nós (dado pelo **Teorema 12.1** por $2^p - 1$) é denominada de árvore binária **perfeita** (ou **repleta**).

Uma definição alternativa para árvore binária perfeita é que, numa árvore dessa natureza, qualquer nó interno possui dois filhos e todas as folhas estão no mesmo nível.

Teorema 12.5 O número de nós de grau dois (n_2) de uma árvore binária perfeita mantém a seguinte relação com o número total de folhas (n_0): $n_2 = n_0 - 1$.

Corolário 12.1 Numa árvore binária perfeita, o número de folhas (n_0) mantém a seguinte relação com o número total de nós (n):

$$n_0 = \frac{n+1}{2}$$

Observação: Não confunda árvore estritamente binária com árvore binária perfeita. Toda árvore binária perfeita é uma árvore estritamente binária, mas a recíproca não é verdadeira.

12.2.4 Completas

Uma representação elegante para árvores binárias perfeitas é obtida a partir da numeração sequencial dos seus nós, começando com o nó do nível 1, depois os nós do nível 2, e assim por diante. Em cada nível, os nós são numerados da esquerda para a direita. Uma árvore binária perfeita, de altura 4, com numeração sequencial é apresentada na **Figura 12–6**.

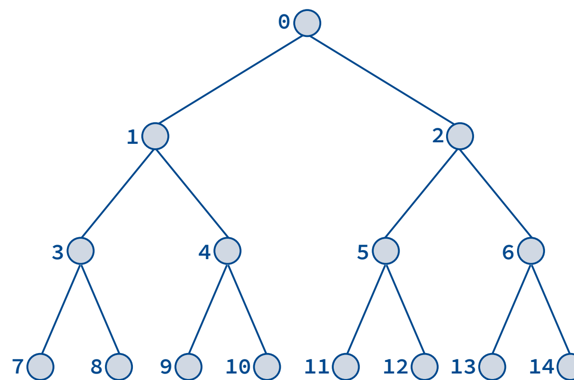


Figura 12–6: Árvore Binária Perfeita

Esse esquema de numeração de nós serve para definir uma árvore binária completa, como se vê a seguir.

Definição 12.8 Uma árvore binária com n nós e profundidade p é **completa** se e somente se seus nós correspondem aos nós que são numerados de 0 até $n - 1$ numa árvore binária perfeita de profundidade p.

Numa árvore binária completa, todos os níveis, exceto talvez o último, são completamente preenchidos. Isto é, de acordo com a última definição, pode-se dizer que uma árvore binária completa é uma árvore binária perfeita com suas folhas mais à direita removidas. Note que toda árvore repleta é uma árvore completa, mas a recíproca não é verdadeira. Por exemplo, a árvore binária apresentada na **Figura 12–7** é uma árvore binária completa, mas ela não é perfeita.

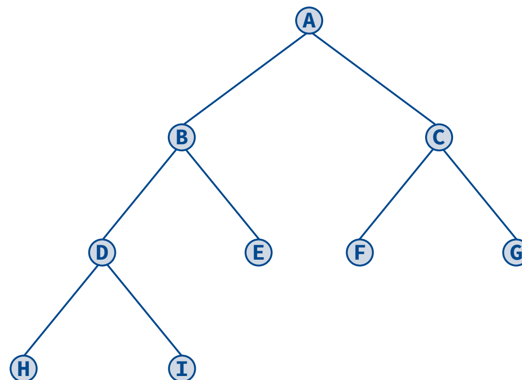


Figura 12–7: Árvore Binária Completa

Também, não confunda árvore estritamente binária com árvore binária completa. Existem árvores estritamente binárias que não são completas e existem árvores binárias completas que não são estritamente binárias.

Teorema 12.6 A profundidade p de uma árvore binária completa com n nós é dada por $p = \lfloor log_2\, n + 1 \rfloor$.

Os nós de uma árvore binária completa podem ser armazenados num array unidimensional `arvore[]`, com o nó numerado por `i` sendo armazenado em `arvore[i]`. O **Teorema 12.7**, a seguir, permite que se determinem as localizações do pai, do filho da esquerda e do filho da direita de um nó de uma árvore binária completa representada neste esquema.

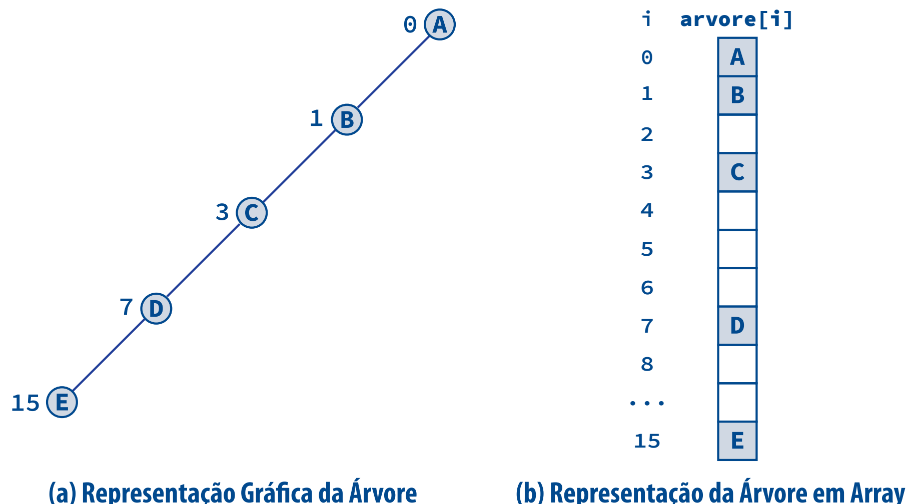
Teorema 12.7 Se uma árvore binária completa com n nós for representada sequencialmente conforme foi descrito acima, então, para qualquer nó numerado por i, $0 \leq i \leq n - 1$, tem-se:

(i) $Pai(i)$ é numerado como $\lfloor (i - 1)/2 \rfloor$, se $i \neq 0$. Se $i = 0$, o nó i é a raiz da árvore, que não possui pai.

(ii) $FilhoEsquerdo(i)$ é numerado como $2 \cdot i + 1$, se $2 \cdot i + 1 \leq n$. Se $2 \cdot i + 1 > n$, então o nó i não possui filho da esquerda.

(iii) $FilhoDireito(i)$ é numerado como $2 \cdot i + 2$, se $2 \cdot i + 2 \leq n$. Se $2 \cdot i + 2 > n$, então o nó i não possui filho da direita.

A representação sequencial proposta pode ser utilizada para qualquer árvore binária (e não apenas para árvores binárias completas), embora, em muitos casos, possa haver desperdício de espaço. Para árvores binárias completas essa representação é ideal, pois não há desperdício de espaço. O pior caso é o de uma árvore binária inclinada à direita de profundidade p, que requer um array com $2^p - 1$ elementos dos quais apenas p elementos são efetivamente usados. A **Figura 12–8** e a **Figura 12–9**, apresentam, respectivamente, as representações por array da árvore inclinada à esquerda e da árvore completa vistas na **Figura 10-5** e na **Figura 10-6**.



(a) Representação Gráfica da Árvore (b) Representação da Árvore em Array

Figura 12–8: Árvore Binária Inclinada Representada em Array

Além da desvantagem de um possível desperdício de memória, a representação sequencial também dificulta as operações de remoção e inserção de nós. De fato, inserção e remoção de nós no interior de uma árvore requer movimentações de muitos nós para refletir a mudança de nível desses nós. Estes problemas podem ser superados com a utilização de uma representação encadeada. Nessa representação, cada nó seria representado por uma estrutura com três campos: `esquerda`, `conteudo` e `direita`, como ilustrado na **Figura 12–10**.

Embora essa estrutura de nós torne difícil determinar o pai de um nó, ela é adequada para a maioria das aplicações de árvores binárias. Além disso, pode-se acrescentar um quarto campo, denominado `pai`, se for rotineira a determinação dos pais dos nós.

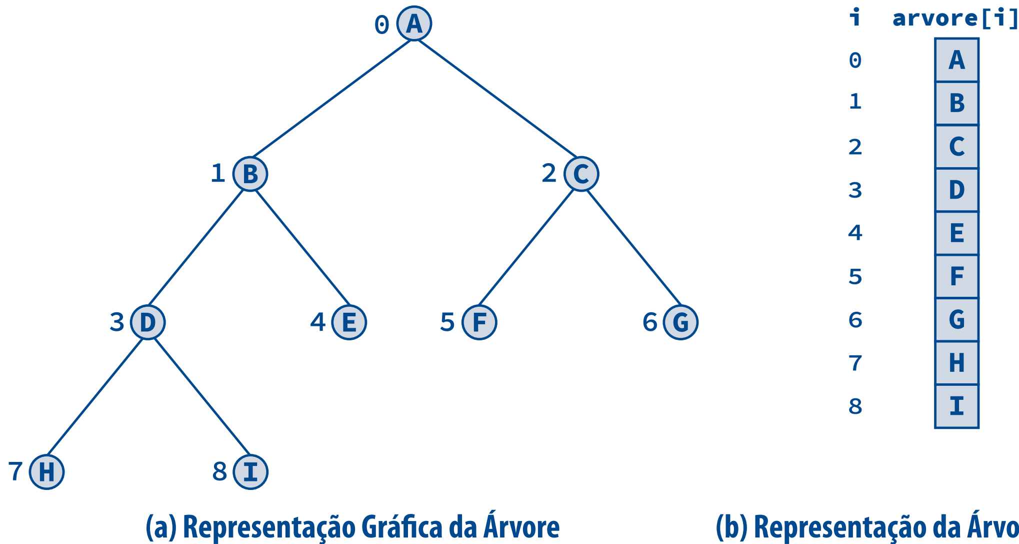


Figura 12–9: Árvore Binária Completa Representada em Array

Figura 12–10: Representação Esquemática de Nó de Árvore Binária

12.3 Caminhamentos em Árvores Binárias

Visitar um nó de uma árvore significa acessá-lo com o objetivo de realizar alguma operação sobre seu conteúdo. Por exemplo, pode-se visitar um nó com o objetivo de exibir seu conteúdo na tela, consultar ou alterar esse conteúdo. Por outro lado, entende-se por **caminhamento** a ação de visitar todos os nós de uma árvore.

Os conceitos de visitação e caminhamento também se aplicam a listas encadeadas. Numa lista encadeada, existe uma ordem natural para percorrer todos os seus nós, que é partir do início da lista e seguir na direção do encadeamento até o final da lista (por esse motivo, listas encadeadas são denominadas **estruturas lineares**). No entanto, não há nenhuma ordem linear natural para visitar os nós de uma árvore (as árvores são conhecidas como **estruturas não lineares**).

Podem-se definir várias ordens de caminhamento em árvores binárias. Aqui, serão descritos os três tipos de caminhamento mais comuns, que são: caminhamento em **ordem prefixa** (abreviadamente, **caminhamento prefixo**), caminhamento em **ordem infixa** (**caminhamento infixo**) e caminhamento em **ordem sufixa** (**caminhamento sufixo**). Em cada um desses métodos de caminhamento, nada precisa ser feito para caminhar sobre uma árvore binária vazia. Os métodos serão definidos recursivamente, de forma que executar o caminhamento sobre uma árvore binária envolve visitar a raiz e *caminhar* em suas subárvores esquerda e direita. Estes métodos são descritos em seguida.

12.3.1 Caminhamento Prefixo

A **Figura 12–11** apresenta o algoritmo de caminhamento em ordem prefixa numa árvore binária.

ALGORITMO CAMINHAMENTOPREFIXO

ENTRADA: Uma árvore binária

SAÍDA: O resultado de um caminhamento em ordem prefixa na árvore

1. Visite a raiz
2. Caminhe na subárvore esquerda em ordem prefixa
3. Caminhe na subárvore direita em ordem prefixa

FIGURA 12–11: ALGORITMO DE CAMINHAMENTO PREFIXO

12.3.2 Caminhamento Infixo

O algoritmo de caminhamento em ordem infixa numa árvore binária é exposto na **Figura 12–12**.

ALGORITMO CAMINHAMENTOINFIXO

ENTRADA: Uma árvore binária

SAÍDA: O resultado de um caminhamento em ordem infixa na árvore

1. Caminhe na subárvore esquerda em ordem infixa
2. Visite a raiz
3. Caminhe na subárvore direita em ordem infixa

FIGURA 12–12: ALGORITMO DE CAMINHAMENTO INFIXO

12.3.3 Caminhamento Sufixo

A **Figura 12–13** apresenta o algoritmo de caminhamento em ordem prefixa numa árvore binária.

ALGORITMO CAMINHAMENTOSUFIXO

ENTRADA: Uma árvore binária

SAÍDA: O resultado de um caminhamento em ordem sufixa na árvore

1. Caminhe na subárvore esquerda em ordem sufixa
2. Caminhe na subárvore direita em ordem sufixa
3. Visite a raiz

FIGURA 12–13: ALGORITMO DE CAMINHAMENTO SUFIXO

12.3.4 Exemplos

É fácil memorizar esses caminhamentos utilizando as seguintes observações:

- Em todos os caminhamentos, o caminhamento na subárvore esquerda ocorre antes do caminhamento na subárvore direita. Assim como se escreve em português.
- O que diferencia os caminhamentos é a ordem na qual a raiz é visitada com relação ao caminhamento na subárvore esquerda. No caminhamento prefixo, o prefixo *pre* indica que a visita à raiz *pre*cede os caminhamentos nas subárvores.
- No caminhamento infixo, o prefixo *in* indica que a visita à raiz ocorre *entre* os caminhamentos nas duas subárvores.
- Finalmente, no caminhamento em sufixo, o prefixo *su* indica que a visita à raiz *su*cede os caminhamento nas subárvores.

Considere a árvore binária da **Figura 12–14**. Os caminhamentos prefixo, infixo e sufixo aplicados a essa árvore produziriam as seguintes sequências de visitação dos nós:

- ❒ Prefixo: `ABDGCEHIF` (v. **Figura 12–15**)
- ❒ Infixo: `DGBAHEICF` (v. **Figura 12–16**)
- ❒ Sufixo: `GDBHIEFCA` (v. **Figura 12–17**)

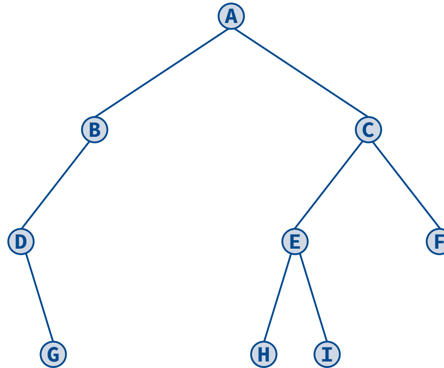


Figura 12–14: Caminhamentos em Árvore Binária

Note que as folhas de uma árvore binária são visitadas na mesma ordem em todos os caminhamentos.

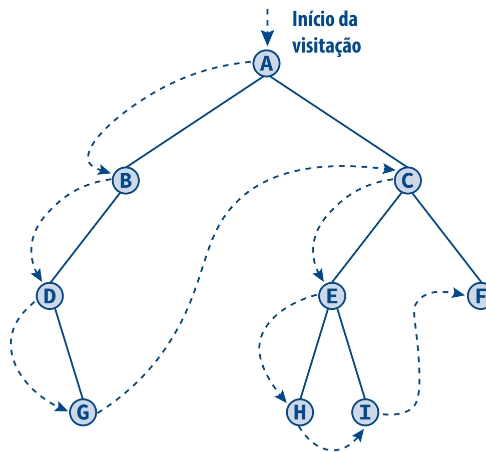


Figura 12–15: Caminhamento Prefixo em Árvore Binária

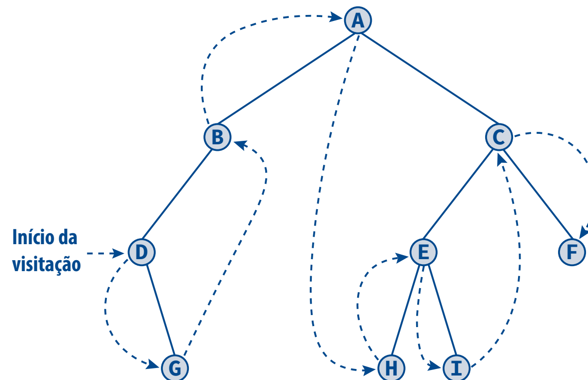


Figura 12–16: Caminhamento Infixo em Árvore Binária

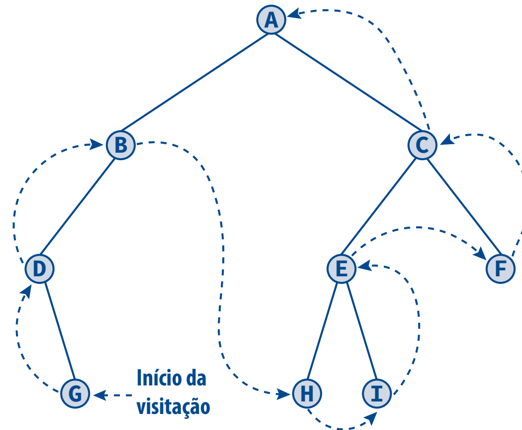


Figura 12–17: Caminhamento Sufixo em Árvore Binária

Árvores binárias podem ser utilizadas para representar expressões aritméticas. **Árvore de expressão** é uma árvore binária contendo operadores (nós internos) e operandos (folhas) e que não armazena parênteses. Neste método de representação, a raiz da árvore contém o operador que deve ser aplicado aos resultados das avaliações das expressões aritméticas representadas pelas subárvores esquerda e direita. Um nó que representa um operador (binário) possui duas subárvores não vazias, enquanto um nó que representa um operando não possui subárvores (i.e., é um nó folha). Assim a expressão `A + B*C` seria representada pela árvore da **Figura 12–18**.

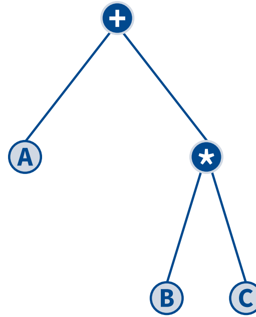


Figura 12–18: Representação de Expressão Aritmética em Árvore Binária 1

Enquanto a expressão `(A + B)*C` seria representada como na **Figura 12–19**.

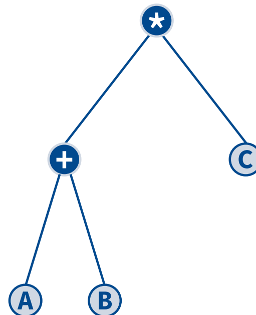


Figura 12–19: Representação de Expressão Aritmética em Árvore Binária 2

O que acontece quando se executa sobre estas árvores cada uma das três formas de caminhamento definidas acima? O caminhamento prefixo para a árvore do primeiro exemplo acima produz `+A*BC`, enquanto, para a árvore do segundo exemplo, resultaria em `*+ABC`. Observe que estas sequências representam justamente as

formas prefixas das expressões dadas. Na realidade, esta regra é geral para expressões representadas em forma de árvores binárias e sugere a denominação dada a esse tipo de caminhamento. É fácil verificar que as sequências de nós obtidas com os caminhamentos sufixos resultam nas formas sufixas das expressões representadas. Para as árvores dos exemplos acima, estes caminhamentos produziriam as sequências: `ABC*+` e `AB+C*`, respectivamente para a primeira e para a segunda árvore. Será, então, que o caminhamento infixo numa árvore que representa uma expressão produz a forma infixa da expressão? A resposta é *nem sempre*! Se, sobre a árvore do primeiro exemplo, fosse executado o caminhamento infixo obter-se-ia a sequência `A+B*C`, que, de fato, é a forma infixa da expressão representada. Entretanto, uma árvore binária que representa uma expressão não contém parênteses, uma vez que a ordem das operações é implícita na estrutura da árvore. Assim uma expressão cuja forma infixa necessita de parênteses para indicar explicitamente a ordem das operações não pode ter sua forma infixa recuperada por um caminhamento infixo convencional. O caminhamento infixo da árvore do segundo exemplo acima resultaria em `A+B*C`, que não corresponde à forma infixa da expressão original.

12.3.5 Caminhamento em Largura (ou por Nível)

Nesse tipo de caminhamento, os nós são visitados nível a nível da árvore. Em cada nível, os nós são visitados da esquerda para a direita. Diferentemente dos demais caminhamentos, o caminhamento por nível não é inerentemente recursivo, mas sua implementação, tipicamente, requer uma fila para armazenar os endereços dos nós que ainda não foram visitados em cada nível (v. **Seção 12.4.3**). Esse tipo de caminhamento aplicado sobre a árvore da **Figura 12–14** resulta em `ABCDEFGHI`, como ilustra a **Figura 12–20**.

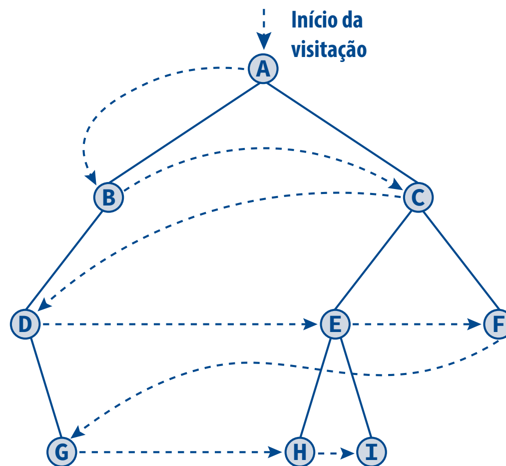


Figura 12–20: Caminhamento por Nível em Árvore Binária

12.4 Implementação de Árvores Binárias

Serão apresentadas a seguir algumas funções em C para construção de árvores binárias e para caminhamentos sobre as mesmas. Ponteiros e estruturas serão utilizados na implementação de árvores, pois estes constituem a forma mais elegante e prática de representar árvores em linguagens que proveem estas facilidades. No entanto, é oportuno frisar que esse não o único modo de representação para árvores (da mesma forma que listas encadeadas não precisam, necessariamente, ser implementadas com ponteiros e estruturas).

12.4.1 Definição de Tipo

As funções apresentadas adiante assumem a existência da seguinte definição dos tipos `tNoArvoreBin` e `tArvoreBin` que representam, respectivamente, nós e ponteiros para nós de árvores binárias:

```
typedef struct rotNoArvoreBin {
           struct rotNoArvoreBin *esquerda;
           tConteudo              conteudo;
           struct rotNoArvoreBin *direita;
        } tNoArvoreBin, *tArvoreBin;
```

Na definição de tipos acima, `tConteudo` é um tipo de dado definido anteriormente pelo programador.

12.4.2 Criação de Nós

A função seguinte implementa a operação de construção de um nó de uma árvore binária. O parâmetro dessa função é o conteúdo efetivo do nó e ela retorna o endereço do nó criado.

```
tArvoreBin ConstroiNoArvoreBin(tConteudo item)
{
   tArvoreBin no;

   no = malloc(sizeof(tNoArvoreBin)); /* Tenta alocar o novo nó */

      /* Se não houve alocação, aborta o programa */
   ASSEGURA(no, "Erro: Nao foi possivel alocar no'");

      /* Preenche o conteúdo do nó */
   no->conteudo = item;
   no->esquerda = NULL; /* Este nó ainda não tem filho */
   no->direita = NULL;

   return no;
}
```

As funções `FilhoEsquerdo()` e `FilhoDireito()`, apresentadas a seguir, criam, respectivamente, os filhos da esquerda e da direita de um nó recebido como parâmetro.

```
void FilhoEsquerdo(tNoArvoreBin *pNo, tConteudo item)
{
      /* O ponteiro para o nó não pode ser NULL */
   ASSEGURA(pNo, "Erro: Insercao invalida.");

      /* O nó não deve ter filho à esquerda ainda */
   ASSEGURA(!pNo->esquerda, "Erro: Insercao invalida.");

      /* Constrói um novo nó e faz com que ele seja     */
      /* filho à esquerda do nó recebido como parâmetro */
   pNo->esquerda = ConstroiNoArvoreBin(item);
}

void FilhoDireito(tNoArvoreBin *pNo, tConteudo item)
{
      /* O ponteiro para o nó não pode ser NULL */
   ASSEGURA(pNo, "Erro: Insercao invalida.");

      /* O nó não deve ter filho à direita ainda */
   ASSEGURA(!pNo->direita, "Erro: Insercao invalida.");

      /* Constrói um novo nó e faz com que ele seja    */
      /* filho à direita do nó recebido como parâmetro */
   pNo->direita = ConstroiNoArvoreBin(item);
}
```

As funções apresentadas acima fazem uso da macro `ASSEGURA()` discutida na Seção 7.4.

12.4.3 Caminhamentos

Nas funções seguintes, são implementados (recursivamente) os caminhamentos prefixo, infixo e sufixo. O parâmetro `op` utilizado nessas funções é um ponteiro para função (v. **Seção Figura 12–34**) definido como:

```
typedef void (*tOperacao) (tConteudo);
```

Ponteiros desse tipo representam operações a ser efetuadas sobre a informação (do tipo `tConteudo`) contida em cada nó da árvore. Por exemplo, se a informação contida em cada nó fosse um string e a operação requerida no caminhamento fosse exibir na tela as informações a função **puts()** da biblioteca padrão de C (`#include <stdio.h>`) poderia ser utilizada como respectivo parâmetro real numa chamada a qualquer dessas funções.

```
void CaminhamentoPrefixo(tArvoreBin arvore, tOperacao op)
{
   if (arvore) {
      op(arvore->conteudo); /* Visita a raiz */

         /* Caminha na subárvore da esquerda */
      CaminhamentoPrefixo(arvore->esquerda, op);

         /* Caminha na subárvore da direita */
      CaminhamentoPrefixo(arvore->direita, op);
   }
}
```

```
void CaminhamentoInfixo(tArvoreBin arvore, tOperacao op)
{
   if (arvore) {
      CaminhamentoInfixo(arvore->esquerda, op); /* Caminha subárvore esquerda */

      op(arvore->conteudo); /* Visita a raiz */

      CaminhamentoInfixo(arvore->direita, op); /* Caminha subárvore direita */
   }
}
```

```
void CaminhamentoSufixo(tArvoreBin arvore, tOperacao op)
{
   if (arvore) {
      CaminhamentoSufixo(arvore->esquerda, op); /* Caminha na subárvore esquerda */

      CaminhamentoSufixo(arvore->direita, op); /* Caminha na subárvore direita */

      op(arvore->conteudo); /* Visita raiz */
   }
}
```

12.4.4 Altura

O conceito de altura (ou profundidade) foi definido na **Seção 12.1**. A função `AlturaArvoreBin()`, apresentada a seguir, calcula e retorna a altura de uma árvore binária.

```
int AlturaArvoreBin(tArvoreBin p)
{
   int profEsq, profDir;

   if (!p) {
      return 0;
   } else {
        /* Calcula a altura da subárvore esquerda */
      profEsq = 1 + AlturaArvoreBin(p->esquerda);
```

```
        /* Calcula a altura da subárvore direita */
      profDir = 1 + AlturaArvoreBin(p->direita);

        /* A altura da árvore é a maior */
      return profEsq > profDir ? profEsq : profDir;
   }
}
```

12.4.5 Semelhança

Duas árvores binárias são **semelhantes** se elas são ambas vazias ou suas subárvores esquerdas são semelhantes e suas subárvores direitas são semelhantes. O conceito de semelhança entre árvores binárias não deve ser confundido com o de **equivalência** (ou **igualdade**) entre elas. De acordo com a definição, duas árvores binárias são semelhantes (ou iguais) se elas possuem a mesma estrutura (ou mesmo formato). Por outro lado, duas árvores binárias são equivalentes quando elas são semelhantes e os respectivos conteúdos dos seus nós são os mesmos. Por exemplo, as árvores binárias da **Figura 12–21** são semelhantes, mas elas não são equivalentes. Por outro lado, as árvores da **Figura 12–22** não são semelhantes nem são equivalentes.

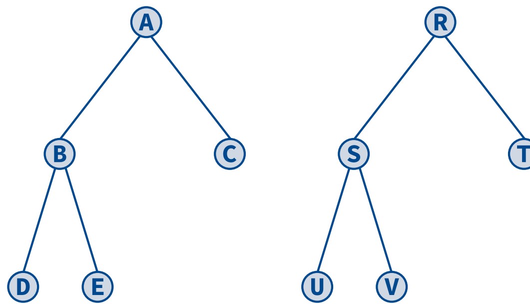


Figura 12–21: Duas Árvores Binárias Semelhantes

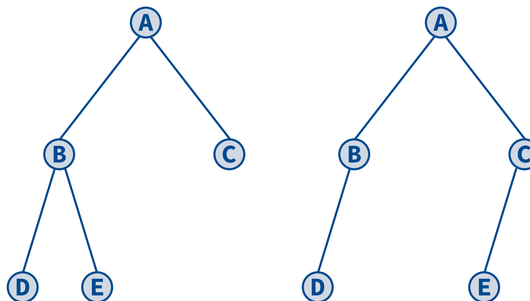


Figura 12–22: Duas Árvores Binárias Não Semelhantes

A função `SaoSemelhantesArvoresBin()`, apresentada a seguir, verifica se duas árvores binárias são semelhantes. Ela retorna `1`, se as árvores recebidas como parâmetros são semelhantes, ou `0`, em caso contrário

```
int SaoSemelhantesArvoresBin(tArvoreBin arvore1, tArvoreBin arvore2)
{
   if (!arvore1 && !arvore2)
      return 1;
   else if (arvore1 && arvore2)
      return SaoSemelhantesArvoresBin( arvore1->esquerda, arvore2->esquerda ) &&
             SaoSemelhantesArvoresBin( arvore1->direita, arvore2->direita );
   else
      return 0;
}
```

12.4.6 Clonagem

A função `CopiaArvoreBin()`, apresentada a seguir, retorna um ponteiro para uma nova árvore binária que é uma cópia da árvore recebida como parâmetro.

```
tArvoreBin CopiaArvoreBin(tArvoreBin arvore)
{
   tArvoreBin copia;

   if (!arvore) {
      return NULL; /* A árvore é vazia e não há o que copiar */
   } else {
      copia = ConstroiNo(arvore->conteudo); /* Cria a raiz da árvore */

         /* Chama esta função recursivamente para copiar */
         /* os filhos da esquerda e da direita           */
      copia->esquerda = CopiaArvoreBin(arvore->esquerda);
      copia->direita = CopiaArvoreBin(arvore->direita);

      return copia;
   }
}
```

12.4.7 Contagem de Nós

A função `NumeroDeNosArvoreBin()`, apresentada a seguir, calcula o número de nós de uma árvore binária.

```
int NumeroDeNosArvoreBin(tArvoreBin arvore)
{
   if (!arvore)
      return 0; /* O número de nós de uma árvore vazia é 0 */
   else
         /* O número de nós de uma árvore não vazia é igual a 1 mais */
         /* o número de nós de sua subárvore esquerda mais o número  */
         /* de nós de sua subárvore direita                          */
      return 1 + NumeroDeNosArvoreBin(arvore->esquerda)
               + NumeroDeNosArvoreBin(arvore->direita);
}
```

12.4.8 Destruição

A função `DestroiArvoreBin()` apresentada a seguir libera o espaço ocupado por cada nó de uma árvore binária por meio de um caminhamento em ordem sufixa.

```
void DestroiArvoreBin(tArvoreBin arvore)
{
   if (arvore) {
      DestroiArvoreBin(arvore->esquerda); /* Destrói subárvore esquerda */

      DestroiArvoreBin(arvore->direita); /* Destrói subárvore direita */

      free(arvore); /* Libera a raiz */
   }
}
```

12.5 Árvores Binárias Baseadas em Caminhamentos

Suponha que a única informação disponível sobre uma árvore binária seja um dos seus caminhamentos. Será possível construir a árvore que deu origem a esse caminhamento com base nessa única informação?

A resposta à questão acima é *não*. Para convencer-se da validade dessa resposta, suponha que se saiba que `BFDAEGC` representa o caminhamento infixo de uma árvore binária. A **Figura 12–23** mostra que existem pelo menos duas árvores que apresentam esse mesmo caminhamento.

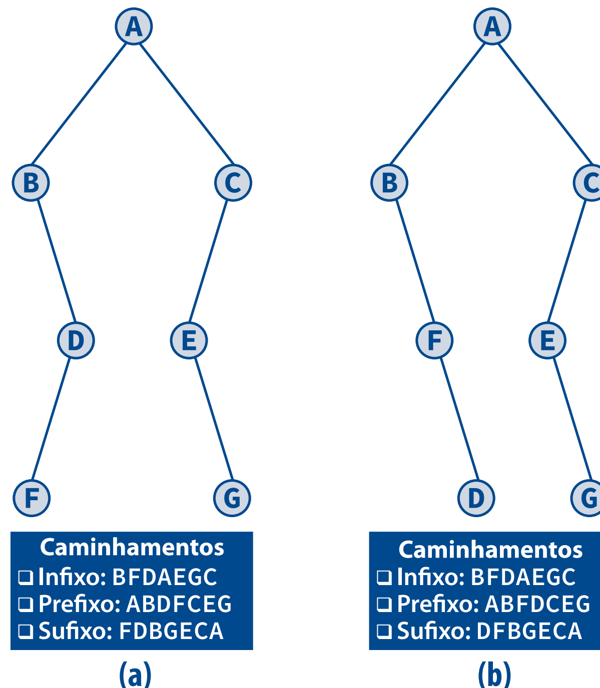


Figura 12–23: Duas Árvores Binárias com o Mesmo Caminhamento Infixo

Agora, considere novamente as árvores binárias representadas na **Figura 12–23** e note que elas possuem diferentes caminhamentos prefixo e sufixo. Esse fato sugere que é possível construir uma árvore binária levando em consideração apenas seus caminhamentos infixo e prefixo ou seus caminhamentos infixo e sufixo. Com efeito, essa constatação é verdadeira, mas existe uma ressalva: a árvore em questão não pode possuir dois ou mais nós com o mesmo conteúdo. Para entender a razão para essa restrição, suponha que uma árvore binária possua `BAA` como caminhamento infixo e `BAA` como caminhamento prefixo. Nesse exemplo, esses caminhamentos poderiam dar origem a duas árvores binárias distintas, como mostra a **Figura 12–24**.

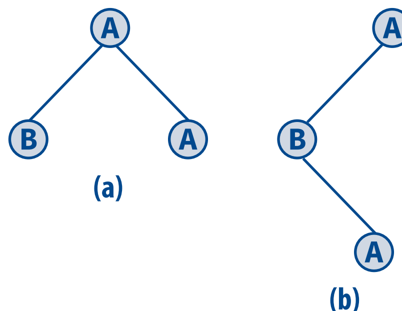


Figura 12–24: Árvores Binárias com os Mesmos Caminhamentos Infixo e Prefixo

Para ilustrar o processo de construção de uma árvore a partir de dois caminhamentos, suponha que se tenham os caminhamentos infixo e prefixo iguais, respectivamente, a `BFDAEGC` e `ABDFCEG`. Esses caminhamentos correspondem exatamente aos caminhamentos prefixo e infixo da árvore da **Figura 12–23 (a)**. Portanto a missão neste exemplo é construir essa árvore a partir desses caminhamentos como se a estrutura da árvore fosse desconhecida.

O primeiro valor encontrado num caminhamento prefixo é a raiz da árvore da qual esse caminhamento se origina. Portanto, considerando o caminhamento prefixo em questão, conclui-se que a raiz da árvore é **A**. Agora, localiza-se essa raiz no caminhamento infixo e divide-se esse caminhamento em duas partições: uma à esquerda da raiz e outra à direita da raiz. Essas partições correspondem aos caminhamentos infixos das subárvores da esquerda e da direita da raiz, respectivamente, como mostra a **Figura 12–25**.

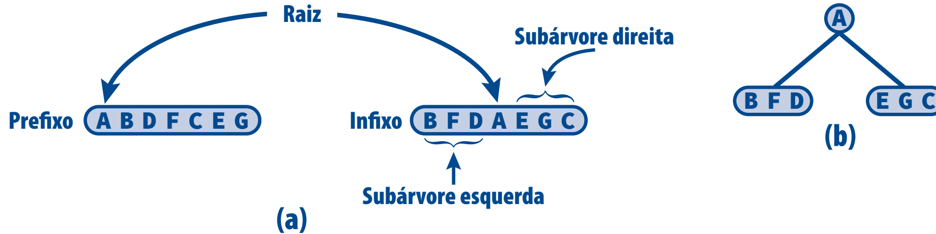


Figura 12–25: Obtendo uma Árvore Binária Baseada em Caminhamentos 1

Neste ponto, o procedimento seguido acima se repete para as subárvores **DFD** e **ECG**, de modo que o resultado é mostrado na **Figura 12–26**.

Novamente, aplicando o mesmo procedimento para a construção das subárvores formadas pelos caminhamentos **FD** e **EG**, obtém-se finalmente a última expansão de nós, que é mostrada na **Figura 12–27**.

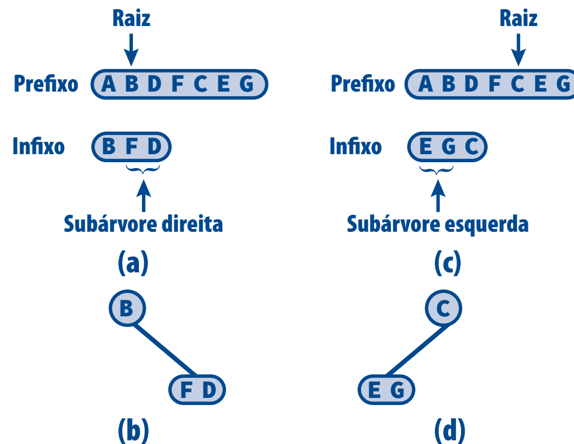


Figura 12–26: Obtendo uma Árvore Binária Baseada em Caminhamentos 2

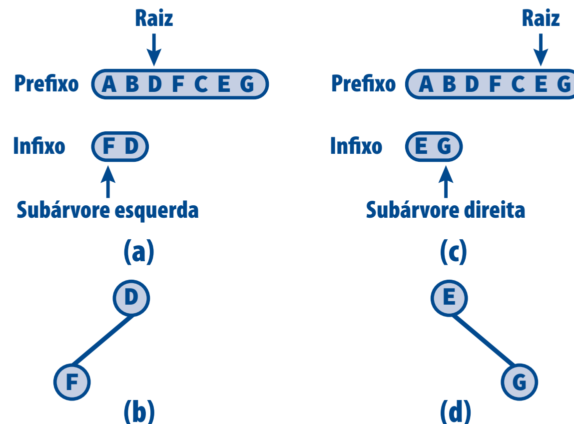


Figura 12–27: Obtendo uma Árvore Binária Baseada em Caminhamentos 3

Observando-se o exemplo acima, pode-se obter insight para implementar uma função que construa uma árvore baseada em seus caminhamentos prefixo e infixo.

A função `ReconstroiArvoreBin()` apresentada a seguir reconstrói uma árvore binária a partir de seus caminhamentos prefixo e infixo de acordo com o raciocínio discutido acima. Essa função retorna o endereço da raiz da árvore reconstruída e seus parâmetros são:

- `prefixo[]` (entrada) — array que armazena o caminhamento prefixo
- `infixo[]` (entrada) — array que armazena o caminhamento infixo
- `infInfixo` (entrada) — índice inferior do array infixo[]
- `supInfixo` (entrada) — índice superior do array infixo[]
- `*proximoNo` (entrada e saída) — o próximo nó a ser levado em consideração no caminhamento prefixo

```
tArvoreBin ReconstroiArvoreBin( const tConteudo prefixo[], const tConteudo infixo[],
                                int infInfixo, int supInfixo, int *proximoNo )
{
   tNoArvoreBin *no; /* Ponteiro para um novo nó */
   int           indiceNo; /* Índice de um nó no array que */
                           /* contém o caminhamento infixo */

      /* Se o índice inferior do array infixo[] for maior do */
      /* que o seu índice superior, não há mais o que fazer  */
   if(infInfixo > supInfixo)
      return NULL;

     /* Obtém o conteúdo do nó corrente no caminhamento prefixo e constrói um novo nó */
   no = ConstroiNo(prefixo[*proximoNo]);

   (*proximoNo)++; /* Passa para o próximo nó no caminhamento prefixo */

      /* Se existir apenas um nó no caminhamento infixo, esse */
      /* nó não possui filhos e não há mais nada o que fazer  */
   if(infInfixo == supInfixo)
      return no;

      /* Obtém o índice do nó corrente no caminhamento infixo */
   indiceNo = IndiceElemento( no->conteudo, infixo, infInfixo, supInfixo );

      /*Constrói a subárvore esquerda deste nó */
   no->esquerda = ReconstroiArvoreBin( prefixo,infixo,infInfixo,
                                       indiceNo - 1,proximoNo );

      /*Constrói a subárvore direita deste nó */
   no->direita = ReconstroiArvoreBin( prefixo, infixo,indiceNo + 1,
                                      supInfixo, proximoNo );
   return no; /* Serviço completo */
}
```

A função `ReconstroiArvoreBin()` chama a função `IndiceElemento()` para encontrar o índice de um nó no caminhamento infixo. Essa função é definida como:

```
int IndiceElemento( tConteudo valor, const tConteudo ar[], int inf, int sup )
{
      /* Procura o elemento no array */
   for (int i = inf; i <= sup; ++i)
      if (ar[i] == valor)
         return i; /* Encontrado */

   return -1; /* Elemento não foi encontrado */
}
```

É importante observar que a função `ReconstroiArvoreBin()` não efetua nenhuma verificação para determinar se os caminhamentos recebidos como parâmetros realmente correspondem a caminhamentos válidos sobre uma mesma árvore binária. Portanto, se esse não for o caso, o resultado será imprevisível.

12.6 Árvores Binárias Costuradas

Podem-se construir funções iterativas (i.e., não recursivas) mais eficientes para os caminhamentos vistos anteriormente. Por exemplo, uma função iterativa para executar o caminhamento infixo numa árvore binária pode ser escrita como a função `CaminhamentoInfixo2()` apresentada adiante. Nessa função, supõe-se a existência do tipo `tPilha` e das funções `Empilha()`, `Desempilha()`, `CriaPilha()` e `PilhaVazia()` que implementam uma pilha cujos elementos são do tipo `tArvoreBin`.

```
void CaminhamentoInfixo2(tArvoreBin arvore, tOperacao op)
{
   tPilha     pilha;
   tArvoreBin pArvore;

   CriaPilha(&pilha);
   pArvore = arvore;

   do {
         /* Desce até o nó mais à esquerda da árvore */
         /* e empilha os nós encontrados no caminho  */
      while (pArvore) {
         Empilha(pArvore, &pilha);
         pArvore = pArvore->esquerda;
      }

         /* Se a pilha não estiver vazia, desempilha um nó, */
         /* visita-o e caminha em sua subárvore direita     */
      if (!PilhaVazia(pilha)) {
         pArvore = Desempilha(&pilha);
         op(pArvore->conteudo);
         pArvore = pArvore->direita;
      }
   } while (!PilhaVazia(pilha) || pArvore);
}
```

Existe uma maneira ainda mais eficiente do que a precedente para implementar-se o caminhamento infixo. Examinando-se a função anterior, percebe-se que a pilha é desempilhada quando `pArvore` é igual a **NULL**. Isto pode acontecer em duas situações. Uma situação é aquela em que se sai da instrução **while** após esta ter sido executada uma ou mais vezes. Isto significa que se *caminhou* para baixo e à esquerda até que se atingiu um ponteiro **NULL**, empilhando-se cada nó à medida que se passava por ele. Assim o elemento do topo da pilha é o valor de `pArvore` antes de ele tornar-se **NULL**. Se um ponteiro auxiliar `q` for mantido num nível acima de `pArvore`, o valor de `q` pode ser utilizado diretamente e o valor de `pArvore` não precisa ser empilhado. A outra situação na qual `pArvore` é **NULL** é quando a instrução `while` não é executada. Isto ocorre após ser atingido um nó com uma subárvore direita vazia; i.e., após a execução da instrução `pArvore = pArvore->direita` e o retorno ao início do laço **do-while**. Neste ponto, ter-se-ia perdido a ordem de caminhamento se não fosse o fato de o nó, em cuja subárvore esquerda foi feito o caminhamento justamente anterior, estar no topo da pilha.

Então, suponha que, em vez de um ponteiro igual a **NULL**, um nó com subárvore direita vazia possuísse um ponteiro para o nó que deveria estar no topo da pilha neste ponto da função. Então não haveria mais necessidade da pilha, pois o nó apontaria diretamente para seu sucessor no caminhamento infixo. Tal ponteiro é denominado **costura** e deve ser diferenciado de um ponteiro normal da árvore, utilizado para ligar um nó às

suas subárvores esquerda e direita. A **Figura 12–28** mostra uma árvore com costuras (linhas pontilhadas), conforme aqui descrito.

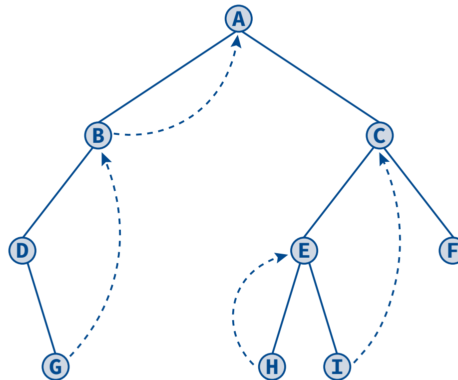


FIGURA 12–28: ÁRVORE BINÁRIA COSTURADA EM ORDEM INFIXA À DIREITA

Observe que o nó mais à direita na **Figura 12–28** tem seu ponteiro direito igual a **NULL**, uma vez que ele não possui nenhum sucessor infixo. Árvores com costuras conforme descritas aqui são denominadas árvores **costuradas em ordem infixa à direita**. Uma árvore binária **costurada em ordem infixa à esquerda** também pode ser similarmente definida como sendo aquela na qual cada ponteiro esquerdo igual a **NULL** é alterado para conter uma costura para seu antecessor no caminhamento infixo. Pode-se ainda ter uma árvore binária que seja simultaneamente costurada em ordem infixa à esquerda e à direita. Neste último caso, diz-se que a árvore é **costurada em ordem infixa**. As árvores costuradas em ordem infixa à esquerda não resultam em tantas vantagens quanto as costuradas à direita e, por isso, não serão consideradas aqui.

Para a implementação de árvores binárias costuradas ordem infixa à direita e para as funções seguintes, são supostas as seguintes definições:

```
typedef enum {SIM, NAO} tCostura;

typedef struct rotNoArvoreCost {
           struct rotNoArvoreCost *esquerda;
           tConteudo               conteudo;
           struct rotNoArvoreCost *direita;
           tCostura                costura;
        } tNoArvoreCost, *tArvoreCost;
```

A função `ConstroiNoArvoreCost()` vista na **Seção 12.4** para árvores binárias não costuradas precisa ser levemente alterada para o caso de árvores costuradas, como mostrado a seguir:

```
tArvoreCost ConstroiNoArvoreCost(tConteudo item)
{
   tNoArvoreCost *pNo;

   pNo = malloc(sizeof(tNoArvoreCost));

   ASSEGURA(pNo, "Erro: Nao foi possivel alocar no'");

   pNo->conteudo = item;
   pNo->esquerda = NULL;
   pNo->direita = NULL;

   return pNo;
}
```

As funções `FilhoEsquerdoCost()` e `FilhoDireitoCost()` são também modificadas com relação àquelas vistas na **Seção 12.4**, conforme mostrado a seguir.

```
void FilhoEsquerdoCost(tNoArvoreCost *pNo, tConteudo item)
{
   tArvoreCost q;

      /* O ponteiro para o nó não pode ser NULL */
   ASSEGURA(pNo, "Erro: Insercao invalida.");

      /* O nó não deve ter filho à esquerda ainda */
   ASSEGURA(!pNo->esquerda, "Erro: Insercao invalida.");

      /* Cria o filho da esquerda do nó p e faz q apontar para ele */
   q = pNo->esquerda = ConstroiNoArvoreCost(item);

      /* O filho da direita do novo nó conterá uma costura para seu pai */
   q->costura = SIM;
   q->direita = pNo;
}

void FilhoDireitoCost(tNoArvoreCost *pNo, tConteudo item)
{
   tArvoreCost q, r;

      /* O ponteiro para o nó não pode ser NULL */
   ASSEGURA(pNo, "Erro: Insercao invalida.");

      /* O nó não deve ter filho à direita */
   ASSEGURA(!pNo->direita || pNo->costura == SIM, "Erro: Insercao direita invalida");

      /* Constrói um novo nó e faz q apontar para ele */
   q = ConstroiNoArvoreCost(item);

   r = pNo->direita; /* Guarda o sucessor em ordem infixa de p */

   pNo->direita = q; /* Torna o novo nó filho à direita de p */

      /* O ponteiro direito de p aponta para o  */
      /* novo nó e, portanto, não é uma costura */
   pNo->costura = NAO;

      /* Se o sucessor em ordem infixa de p não  */
      /* for NULL, o novo nó conterá uma costura */
   q->costura = r ? SIM : NAO;

      /* O ponteiro direito do novo nó apontará para */
      /* o sucessor em ordem infixa de seu pai       */
   q->direita = r;
}
```

A função `CaminhamentoInfixoCost()` a seguir implementa o caminhamento infixo para uma árvore costurada ordem infixa à direita.

```
void CaminhamentoInfixoCost(tArvoreCost arvore, tOperacao op)
{
   tArvoreCost p,q;

   p = arvore;

   do {
      q = NULL;

      while (p) {
         q = p;
         p = p->esquerda;
      }
```

```
      if (q) {
         op(q->conteudo);

         p = q->direita;

         while (q->costura == SIM) {
            op(p->conteudo);
            q = p;
            p = q->direita;
         }
      }
   } while (q);
}
```

Se as implementações das funções `CaminhamentoPrefixo()` e `CaminhamentoSufixo()` apresentadas na **Seção 12.4.3** forem desejadas para o tipo de árvore costurada implementada aqui, deve-se tomar o cuidado de não seguir as costuras como se estas fossem ramificações normais. Isto é, as funções correspondentes para árvores costuradas em ordem infixa devem ser implementadas como:

```
void CaminhamentoPrefixoCost(tArvoreCost arvore, tOperacao op)
{
   if (arvore) {
      op(arvore->conteudo); /* Visita a raiz */

         /* Caminha na subárvore da esquerda */
      CaminhamentoPrefixoCost(arvore->esquerda, op);

         /* Caminha na subárvore da direita, se o */
         /* ponteiro direto não for uma costura   */
      if (arvore->costura == NAO)
         CaminhamentoPrefixoCost(arvore->direita, op);
   }
}

void CaminhamentoSufixoCost(tArvoreCost arvore, tOperacao op)
{
   if (arvore) {
         /* Caminha na subárvore esquerda */
      CaminhamentoSufixoCost(arvore->esquerda, op);

         /* Caminha na subárvore direita se o      */
         /* ponteiro à direita não for uma costura */
      if (arvore->costura == NAO)
         CaminhamentoSufixoCost(arvore->direita, op);

      op(arvore->conteudo); /* Visita raiz */
   }
}
```

Outros tipos de costuras para os caminhamentos prefixo e sufixo podem ser definidos de modo similar ao que foi visto aqui e espera-se que com o conhecimento adquirido nesta seção o leitor seja capaz de implementá-los.

12.7 Conversões de Árvores Ordinárias e Florestas em Árvores Binárias

Esta seção mostra que qualquer árvore ordinária pode ser representada como uma árvore binária. Esse fato tem implicações importantes para a implementação de árvores ordinárias. Com efeito, a implementação de árvores, cujos nós podem ter seus graus variando muito de um nó para outro, é efetuada utilizando-se nós de tamanhos

variáveis ou utilizando-se nós de tamanhos fixos que contêm um número de campos de ponteiros igual ao grau da árvore (i.e., igual ao maior grau dentre todos os nós da árvore). Essa última forma de implementação dos nós de uma árvore gera um desperdício muito grande de memória, conforme será visto em seguida.

Suponha uma árvore de grau *k* com *n* nós de tamanho fixo, como o nó da **Figura 12–29**.

Figura 12–29: Nó de Árvore Ordinária de Grau k

Uma vez que cada ponteiro não nulo aponta para um nó e exatamente um ponteiro aponta para cada um dos nós (exceto para a raiz), o número de ramificações não nulas numa árvore com *n* nós é exatamente $n - 1$. Como o número total de campos de ponteiros numa árvore de grau *k* é *nk*, o número de ponteiros nulos na árvore é dado por: $nk - (n - 1) = n(k - 1) + 1$.

Uma consequência disso é que uma árvore de grau 3, por exemplo, possui mais de $2/3$ ($2/3 + n/3$, exatamente) de seus campos de ramificações nulos. A proporção de ramificações nulas aproxima-se de *1* quando o grau da árvore cresce. A vantagem de representarem-se árvores ordinárias por árvores binárias é que, nesse último caso, apenas cerca de metade das ramificações é nula.

Considere a árvore da **Figura 12–30**. Em qualquer árvore ordinária, cada nó tem no máximo um filho na extrema esquerda e no máximo um irmão mais próximo à direita. Na árvore da **Figura 12–30**, o filho mais à esquerda do nó B é E e seu irmão mais próximo à direita é o nó C. No método de transformação a ser adotado, cada nó da árvore binária terá como filho à esquerda o filho mais à esquerda do nó correspondente na árvore ordinária que ela representa, e como filho à direita o irmão mais próximo à direita do nó correspondente na árvore original. Uma vez que, numa árvore ordinária, a ordem dos filhos não é importante, qualquer filho de um nó poder ser seu filho mais à esquerda e qualquer de seus irmãos poder ser seu irmão mais próximo à direita. Apenas por uma questão de sistematização, os nós são escolhidos da forma em que eles são desenhados na árvore.

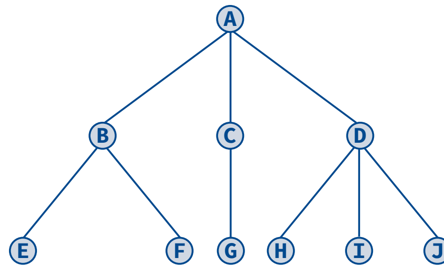


Figura 12–30: Árvore Ordinária de Grau 3

Uma forma prática e totalmente equivalente ao que foi descrito acima de obter a representação de uma árvore ordinária por árvore binária é unir todos os irmãos de um nó e remover todos os ponteiros de um nó para seus filhos, exceto aquele que o une a seu filho mais à esquerda. Utilizando esse artifício para a árvore da **Figura 12–30** obtém-se a árvore binária mostrada na **Figura 12–31 (a)**. Essa figura pode não parecer muito com uma árvore binária, mas se as ramificações horizontais nessa figura forem giradas no sentido horário obtém-se a árvore da **Figura 12–31 (b)**.

A **Figura 12–32** apresenta dois outros exemplos de transformação de árvores ordinárias em árvores binárias.

Note que as raízes das árvores binárias resultantes não possuem filhos à direita. Isto se deve ao fato de a raiz da árvore transformada não possuir nenhum irmão. Esse fato pode ser utilizado para transformarem-se florestas numa única árvore binária. Para tal, transforma-se inicialmente cada árvore da floresta em árvore binária e, depois, essas árvores binárias são unidas por meio dos ponteiros direitos dos nós-raízes. Utilizando-se esse método, a floresta da **Figura 12–33** **(a)** resulta na árvore binária **Figura 12–33** **(b)**.

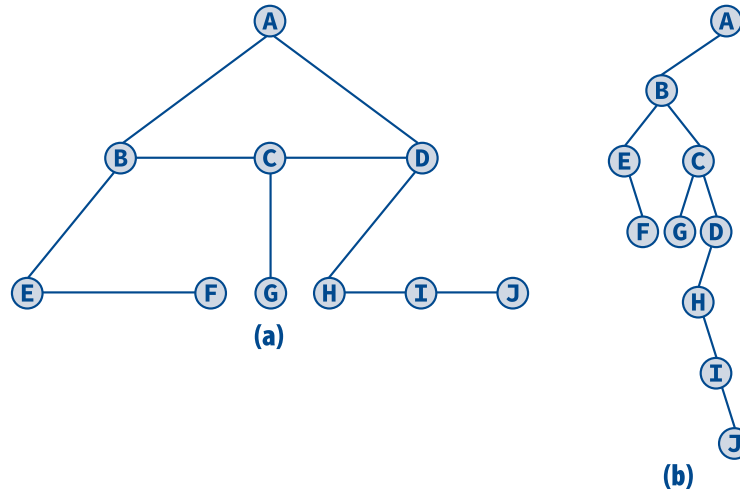


Figura 12–31: Árvore Ordinária de Grau 3 Transformada em Árvore Binária

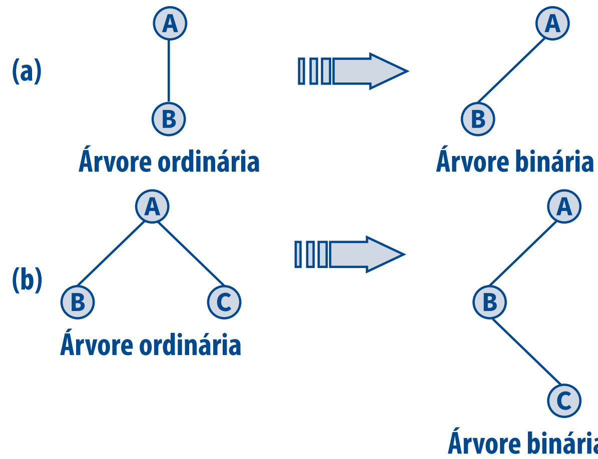


Figura 12–32: Transformações de Árvores Ordinárias em Árvores Binárias

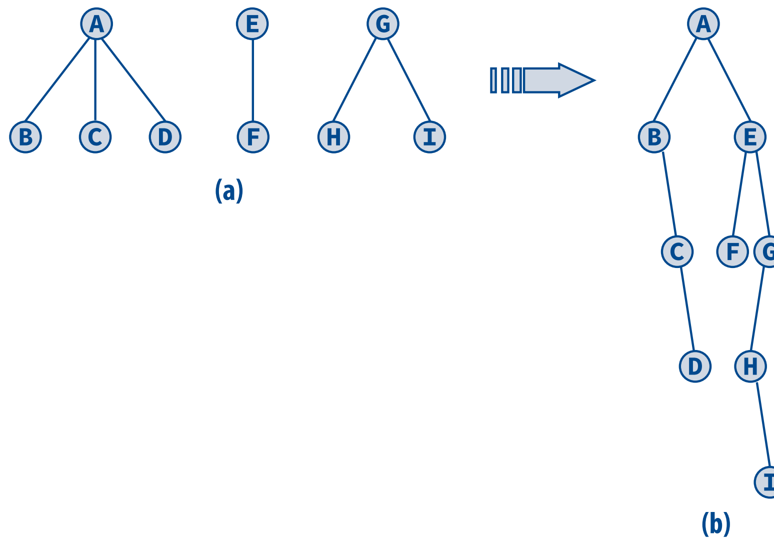


Figura 12–33: Transformação de Floresta em Árvore Binária

12.8 Exemplos de Programação

12.8.1 Problema das Oito Moedas

Preâmbulo: Uma aplicação prática de árvores é no processo de tomada de decisão. Considere o problema conhecido como **Problema das Oito Moedas**. Dadas as moedas A, B, C, D, E, F, G e H, sabe-se que uma delas é falsa e tem um peso diferente das demais. O objetivo é determinar qual das moedas é falsa utilizando uma balança de braços iguais. Deseja-se fazer isso utilizando o menor número de comparações e, ao mesmo tempo, determinando se a moeda falsa é mais pesada ou mais leve que as demais. A árvore da **Figura 12–34** representa um conjunto de decisões por meio das quais se pode obter a resposta do problema. Por isso, tal árvore é denominada de **árvore de decisão**. Uma letra P ou L numa folha da árvore da **Figura 12–34** indica que a moeda representada na respectiva folha é falsa e é mais pesada (P) ou mais leve (L) do que as demais.

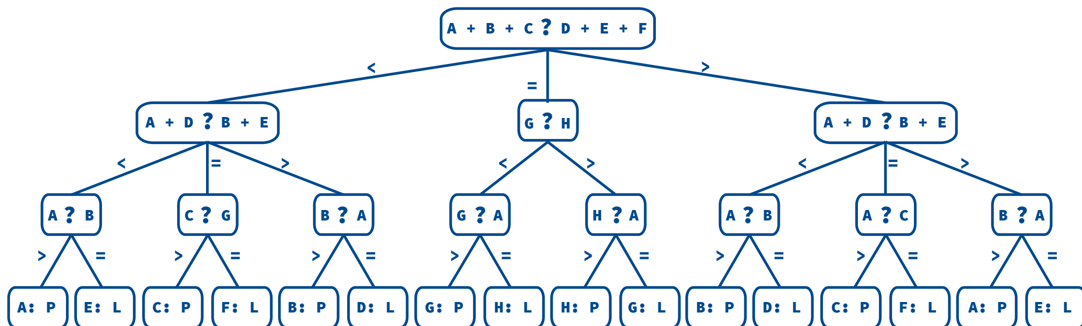


Figura 12–34: Árvore de Decisão do Problema das Oito Moedas

Observe que se $A + B + C < D + E + F$, então se sabe que a moeda falsa está presente entre essas seis moedas e nem é G nem H. Suponha que, na próxima medida, encontra-se que $A + D < B + E$, e depois, trocando-se B por D, não se obtém nenhuma mudança na desigualdade. Isto significa duas coisas: **(1)** que nem C nem F é falsa e **(2)** que nem B nem D é falsa. Se $A + D$ fosse igual a $B + E$, então ou C ou F seria a moeda falsa. Sabendo neste ponto que A ou E é a moeda falsa, compara-se A com uma moeda boa, por exemplo, B. Se A for igual a B, então E é mais pesada; caso contrário, A deve ser mais leve do que as demais moedas.

Observando-se atentamente a árvore da **Figura 12–34**, vê-se que todas as possibilidades são cobertas, uma vez que há oito moedas que podem ser mais leves ou mais pesadas e há dezesseis nós terminais. Cada caminho da raiz até uma folha requer exatamente três comparações. Visualizar esse problema como uma árvore de decisão é muito útil, mas ela não resulta imediatamente num algoritmo. Para resolver o problema das oito moedas utilizando um programa, deve-se codificar uma série de testes que espelhem a estrutura da árvore.

Problema: Escreva um programa em C que reflita a solução do problema das oito moedas de acordo com a árvore de decisão da **Figura 12–34**.

Solução: O programa que resolve o problema proposto usa as seguintes definições de constante simbólica e de tipo:

```
#define NUM_MOEDAS 8 /* Número de moedas */
typedef enum {A, B, C, D, E, F, G, H} tMoeda;
```

A função **main()** a seguir resolve o problema das oito moedas.

```
int main(void)
{
   int    pesos[NUM_MOEDAS];
   tMoeda M;

   printf("Problema das Oito Moedas\n");
   printf("======== === ==== ======\n");

   for (M = A; M <= H; ++M) {
      printf("\nPeso da moeda %c: ", 'A' + M);
      pesos[M] = LeNaturalPositivo();
   }

   if (pesos[A] + pesos[B] + pesos[C] == pesos[D] + pesos[E] + pesos[F])
      if (pesos[G] > pesos[H])
         Compara(G, H, A, pesos);
      else
         Compara(H, G, A, pesos);
   else if ( pesos[A] + pesos[B] + pesos[C] > pesos[D] + pesos[E] + pesos[F] )
      if (pesos[A] + pesos[D] == pesos[B] + pesos[E])
         Compara(C, F, A, pesos);
      else if (pesos[A] + pesos[D] > pesos[B] + pesos[E])
         Compara(A, E, B, pesos);
      else
         Compara(B, D, A, pesos);
   else if ( pesos[A] + pesos[B] + pesos[C] < pesos[D] + pesos[E] + pesos[F] )
      if (pesos[A] + pesos[D] == pesos[B] + pesos[E])
         Compara(F, C, A, pesos);
      else if (pesos[A] + pesos[D] > pesos[B] + pesos[E])
         Compara(D, B, A, pesos);
      else
         Compara(E, A, B, pesos);

   return 0;
}
```

A função `Compara()`, chamada pela função **main()** e apresentada a seguir, é utilizada para executar a última comparação de um caminho da raiz até uma folha da árvore da **Figura 12–34**. Os parâmetros dessa função devem ser interpretados da seguinte maneira:

- `x` (entrada) — moeda a ser comparada
- `y` (entrada) — uma moeda
- `z` (entrada) — moeda padrão
- `p[]` (entrada) — array contendo os pesos das moedas

```
void Compara(tMoeda x, tMoeda y, tMoeda z, const int p[])
{
   if (p[x] > p[z])
      printf("\n>>> A moeda %c e' a mais pesada", 'A' + x);
   else
      printf("\n>>> A moeda %c e' a mais leve", 'A' + y);
}
```

12.8.2 Codificação de Huffman

Preâmbulo: A **codificação de Huffman** consiste numa técnica bastante utilizada para reduzir o espaço de armazenamento. Descrita formalmente, essa técnica parece complicada, mas sua utilização prática é bem simples, apesar de engenhosa. Para entender o funcionamento da codificação de Huffman e

sua importância prática, suponha que você tem um string que deseja, por alguma razão, armazenar no menor espaço possível em memória. Considere, a título de ilustração, que esse string seja **"bola"**. Obviamente, do ponto de vista pragmático, não há razão para você desperdiçar tempo imaginando uma maneira para reduzir o espaço de armazenamento para um string que ocupa tão pouco espaço, mas o raciocínio a ser seguido adiante se aplica também a strings de tamanho arbitrário. Por exemplo, um arquivo de texto com, digamos, 100 GB pode ser considerado como apenas um gigantesco string.

Para armazenar o string **"bola"** usando C ou muitas ouras linguagens de programação são necessários cinco bytes, visto que cada caractere é armazenado em um byte e deve-se levar em consideração o caractere terminal, como mostra a **Tabela 12–1**. Por outro lado, utilizando a codificação de Huffman, esse string pode ser armazenado utilizando-se apenas dois bytes, como mostra a **Tabela 12–2**.

Caractere	Bits Necessários
'b'	*8*
'o'	*8*
'l'	*8*
'a'	*8*
Total: 4 bytes (sem incluir '\0')	

Tabela 12–1: Espaço Necessário para Armazenar um String em C

Caractere	Bits Necessários
'b'	*3*
'o'	*2*
'l'	*3*
'a'	*1*
Total: 2 bytes	

Tabela 12–2: Espaço Necessário para Armazenar um String na Codificação de Huffman

A codificação de Huffman é o princípio mais básico de compressão de arquivos. Essa codificação atribui códigos (i.e., sequências de bits) de tamanhos variados a caracteres. Ou seja, o tamanho do código depende da frequência de cada caractere: quanto maior a frequência, menor é o código. Esses códigos são considerados **livres de prefixo**, pois o código atribuído a um caractere não é um prefixo do código de outro caractere. Por exemplo, suponha que $a = 00$, $b = 01$, $c = 0$ e $d = 1$, então c é prefixo de a e b; logo *0001* pode ser interpretado como *cccd*, *ccb*, *acd* ou *ab*. Consequentemente, se $a = 0$, nenhuma outra codificação pode começar com *0*. Portanto os códigos desse exemplo não são livres de prefixo.

A **Tabela 12–3** mostra frequências hipotéticas de uso de caracteres presentes no string **"bola"**, enquanto a **Tabela 12–4** mostra os códigos (sequências de bits) obtidos por meio da codificação de Huffman quando essas frequências são utilizadas[1].

[1] Essa associação entre letras e frequências foi obtida por meio da contagem de ocorrências de letras numa lista de palavras gentilmente cedida por Valdir Jorge, analista de sistemas e programador na Universidade Concórdia, em Montreal, a quem o autor deste livro penhoradamente

Caractere	Frequência (%)
'b'	*1.54*
'o'	*10.24*
'l'	*4.03*
'a'	*14.26*

Tabela 12–3: Frequência de Uso de Caracteres do String "bola"

Caractere	Codificação ASCII	Codificação de Huffman
'b'	01100010	100
'o'	01101111	11
'l'	01101100	101
'a'	01100001	0

Tabela 12–4: Sequências de Bits nas Codificações ASCII e de Huffman

A codificação de Huffman é representada por uma árvore estritamente binária e o algoritmo na **Figura 12–35** mostra como essa árvore é obtida:

Algoritmo CodificaçãoDeHuffman

Entrada: Um string

Saída: Uma árvore binária de codificação do string usando a codificação de Huffman

1. Crie nós folhas, cada um dos quais contendo um caractere e sua frequência
2. Insira os nós numa lista em ordem crescente de frequência
3. Remova os dois primeiros nós que, como a lista é ordenada em ordem crescente de frequência, possuem as menores frequências
4. Crie um nó cuja frequência seja a soma das frequências dos nós removidos
5. Insira o novo nó na mesma lista em ordem crescente de frequência
6. Repita os passos de **3** a **5** até que a lista esteja vazia

Figura 12–35: Algoritmo de Codificação de Huffman

No algoritmo da **Figura 12–35**, o último nó removido da lista é a raiz da árvore de codificação. Se o número de caracteres codificados for igual a n, a árvore de codificação conterá n folhas e $n - 1$ nós internos.

Na análise de custos do algoritmo de Huffman, o tamanho da entrada, tipicamente referido como n, é o número de símbolos codificados, que tipicamente é muito pequeno comparado com o tamanho dos strings a serem codificados. Por isso, na prática, essa análise raramente é importante, pois, como foi visto no **Capítulo 6**, ela refere-se ao comportamento de um algoritmo quando n se torna muito grande, o que, aqui, raramente é o caso.

Problema: Escreva um programa que lê strings contendo apenas letras introduzidos pelo usuário e apresenta na tela as sequências de bits associados às letras dos strings, sendo essas sequências de bits por meio do algoritmo de codificação de Huffman. [**NB:** O problema não solicita que o programa produza

agradece a gentileza. Essa lista de palavras e o programa que calcula as referidas frequências podem ser encontrados no site do livro. Mas, é imperativo notar que não se pretende que essas frequências tenham qualquer precisão. Na prática, essas frequências dependem do contexto da codificação e devem ser determinadas experimentalmente.

as sequências de bits em si, que podem ser obtidas utilizando-se técnicas de programação de baixo nível, que está além do escopo deste livro.]

Solução:

NB: Implementações mais eficientes do algoritmo de Huffman usam filas de prioridade, que serão estudadas no **Volume 2**, para armazenar os nós ordenados por frequência, como requer o algoritmo delineado acima. Mas, conforme foi discutido no final do preâmbulo, a eficiência assim obtida pode ser interpretada como excesso de preciosismo, de modo que aqui será utilizada uma lista simplesmente encadeada ordenada com o mesmo propósito.

A implementação do algoritmo de Huffman requer o uso de duas estruturas de dados:

- Árvore de codificação que é uma árvore binária da qual se obtêm as sequências de bits almejadas.
- Lista encadeada que armazena os nós da árvore de codificação em ordem crescente de frequência. Ou seja, os nós da árvore que armazenam as menores frequências se encontram no início da lista. No começo da execução do algoritmo, essa lista contém todas as folhas da árvore (e apenas elas), que são os nós que armazenam os caracteres a ser codificados. Após a execução do último passo do algoritmo, essa lista deverá estar vazia.

As estruturas de dados descritas acima, requerem o uso de dois tipos de nós. O primeiro deles usa as definições de tipo a seguir:

```
   /* Tipo do campo de informação dos nós da árvore de codificação */
typedef struct {
          char   letra;
          double frequencia;
        } tLetraFreq;

   /* Tipo de cada nó e tipo de ponteiro para nó da árvore de codificação */
typedef struct rotNoArvoreHuff {
          tLetraFreq              letraFreq;
          struct rotNoArvoreHuff *filhoEsquerdo;
          struct rotNoArvoreHuff *filhoDireito;
        } tNoArvoreHuff, *tArvoreHuff;
```

A primeira definição de tipo acima refere-se a estruturas contendo dois campos: o primeiro campo armazena uma letra e o segundo campo armazena a frequência associada à respectiva letra.

O segundo tipo necessário para o programa é aquele utilizado para definir nós da lista encadeada mencionada acima e usa a seguinte definição de tipo:

```
   /* Tipo de nó e tipo de ponteiro para nó da lista encadeada que */
   /* armazena os nós da árvore de codificação temporariamente     */
typedef struct rotNoLSE {
           tNoArvoreHuff    conteudo;
           struct rotNoLSE *proximo;
        } tNoListaSE, *tListaSE;
```

A **Figura 12–36** mostra esquematicamente como são as estruturas dos tipos `tNoArvoreHuff` e `tNoListaSE`.

Figura 12–36: Nó de Árvore e Nó de Lista Usados em Codificação de Huffman

O primeiro passo do algoritmo consiste em criar a assinalada lista encadeada ordenada, que, inicialmente, contém apenas as folhas da árvore de codificação.

Suponha, por exemplo, que as letras a serem codificadas são aquelas que constituem o string `"bola"`. Então, utilizando as frequências apresentadas na **Tabela 12–3**, a execução do primeiro passo do algoritmo deve resultar na lista encadeada ilustrada na **Figura 12–37**.

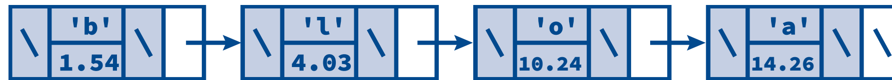


Figura 12–37: Composição Inicial da Lista na Codificação de Huffman

A função `CriaListaSEOrd()` apresentada a seguir implementa o primeiro passo do algoritmo de Huffman. Essa função cria uma lista contendo nós de uma árvore que armazenam caracteres e suas respectivas frequências. Seu único parâmetro é um string contendo os caracteres que serão armazenados com suas respectivas frequências. Essa função retorna o endereço da lista criada.

```
tListaSE CriaListaSEOrd(char *string)
{
   tListaSE    aLista; /* A lista que será criada */
   tNoListaSE *ptrNovoNo; /* Apontará para o novo nó alocado */
   char       *p = string; /* Ponteiro usado para acessar as letras   */
   int         guardaCaractere; /* Armazena um caractere */

   aLista = NULL; /* A lista deve ser iniciada com NULL */

      /* Acessa caracteres no string, determina sua frequência, cria */
      /* uma folha da árvore com esse conteúdo e armazena-a na lista */
   while (*p) {
         /* Verifica se a letra para a qual p aponta já foi levada em consideração */

      guardaCaractere = *p; /* Guarda o caractere para o qual p aponta */

      *p = '\0'; /* Termina temporariamente o string */

         /* Verifica se o caractere é encontrado na porção    */
         /* do string que antecede o local para onde p aponta */
      if (strchr(string, guardaCaractere)) {
         /* Este caractere já foi acrescentado à lista */

         *p = guardaCaractere; /* Restaura o string */

         ++p; /* Passa para o próximo caractere */

         continue; /* Salta o restante do laço */
      }

      *p = guardaCaractere; /* Restaura o string */

         /* Tenta alocar um novo nó. Se não for possível, aborta o programa. */
      ASSEGURA( ptrNovoNo = malloc(sizeof(tNoListaSE)),
                "Nao foi possível alocar no da lista" );

         /* Armazena a letra corrente e sua frequência no novo nó */
      ptrNovoNo->conteudo.letraFreq.letra = *p;
      ptrNovoNo->conteudo.letraFreq.frequencia = FrequenciasDeLetras(*p);

         /* Quando FrequenciasDeLetras() retorna um valor negativo, o caracter    */
         /* não possui frequência especificada. Nesse caso, o programa é abortado */
      ASSEGURA( ptrNovoNo->conteudo.letraFreq.frequencia >= 0,
                "Encontrado caractere sem frequencia" );
```

```
         /* Inicialmente, a lista só contém folhas da árvore */
      ptrNovoNo->conteudo.filhoEsquerdo = NULL;
      ptrNovoNo->conteudo.filhoDireito = NULL;

      InsereEmOrdemListaSE(&aLista, ptrNovoNo); /* Insere o nó na lista */

      p++; /* Avança para o próximo caractere */
   }

   return aLista; /* Retorna o endereço inicial da lista */
}
```

A função `CriaListaSEOrd()` é repleta de comentários que facilitam seu entendimento, mas as seguintes observações adicionais se fazem necessárias:

- A função examina cada caractere no string a ser codificado e acrescenta-o na lista levando em consideração que um mesmo caractere não pode ser incluído mais de uma vez. Em caso contrário, existiriam duas sequências de bits diferentes associadas a um mesmo caractere, o que é um disparate. Por exemplo, se o string a ser codificado fosse `"banana"` e essa precaução não fosse tomada, cada caractere `'a'` teria uma codificação diferente.
- A frequência de cada letra é obtida por meio de uma chamada da função `FrequenciasDeLetras()`.
- A função `CriaListaSEOrd()` chama `InsereEmOrdemListaSE()` para inserir cada nó criado em ordem crescente de frequência. Essa última função é similar à função `InsereEmOrdemLSE()` apresentada na **Seção 10.3.2**.

Antes que lhe seja apresentada a função que implementa os passos seguintes do algoritmo de codificação de Huffman, é importante que você realmente entenda o que será feito. Para facilitar o entendimento, suponha novamente que o string a ser codificado é `"bola"`. Após a execução da função `CriaListaSEOrd()`, a lista encadeada apresentada na **Figura 12–37** é criada. Então, os dois primeiros nós dessa lista são removidos e, utilizando os seus conteúdos, é criado um nó da árvore de codificação, de modo que a frequência armazenada nesse novo nó seja a soma das frequências armazenadas nos nós removidos. A **Figura 12–38** ilustra a criação desse nó. Note que a interrogação que aparece nesse novo nó indica que seu campo `letra` não foi alterado e, portanto, é indefinido. De fato, alterar o valor desse campo não tem nenhuma importância, já que ele jamais será utilizado.

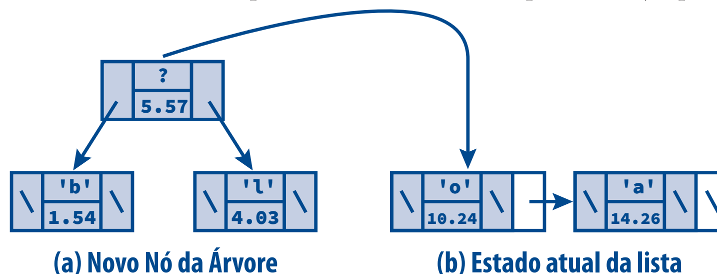


Figura 12–38: Criação do Primeiro Nó da Árvore na Codificação de Huffman

Prosseguindo com o exemplo de codificação do string `"bola"`, após a criação do nó da árvore ilustrada na **Figura 12–38**, insere-se esse nó como conteúdo de um novo nó da lista encadeada, como mostra a **Figura 12–39**. Como sempre, o novo nó da lista encadeada é inserido em ordem de frequência crescente.

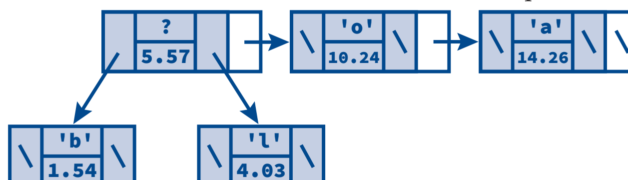


Figura 12–39: Lista após Inserção do Primeiro Nó Criado na Codificação de Huffman

Continuando com a codificação do string **"bola"**, novamente, os dois nós com menor frequência são removidos da lista e seus conteúdos dão origem a mais um nó da árvore de codificação, como mostra a **Figura 12–40**. Então, o conteúdo desse novo nó é usado para construir um novo nó da lista que, em seguida, é inserido em ordem nessa lista (v. **Figura 12–41**).

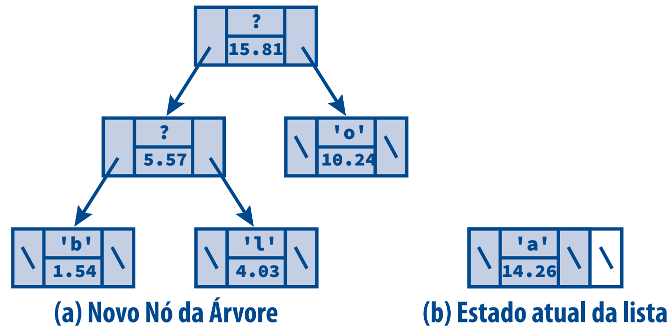


Figura 12–40: Criação do Segundo Nó da Árvore na Codificação de Huffman

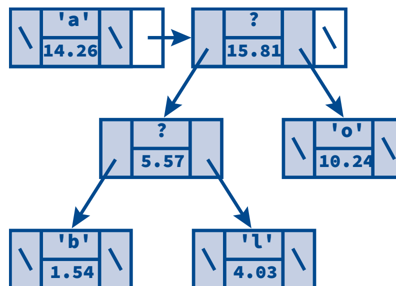


Figura 12–41: Lista após Inserção do Segundo Nó Criado na Codificação de Huffman

Na sequência de codificação do string **"bola"**, mais uma vez, os dois nós com menor frequência são removidos da lista e seus conteúdos são usados para construir um novo nó da árvore de codificação. Como mostra a **Figura 12–42**, neste instante a lista fica temporariamente vazia.

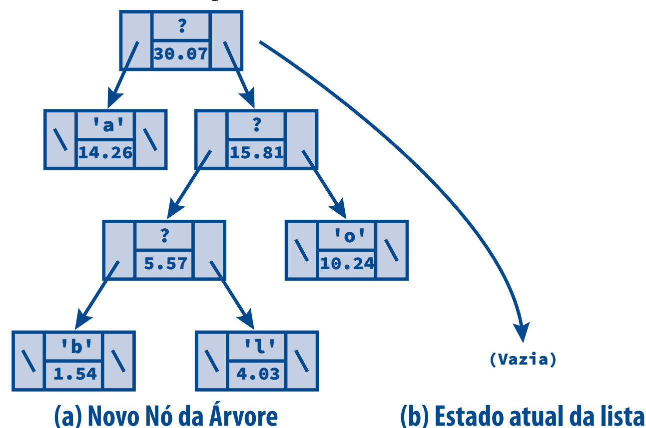


Figura 12–42: Criação do Terceiro Nó da Árvore na Codificação de Huffman

No passo seguinte, na codificação do string **"bola"**, o último nó da árvore que foi criado é usado como conteúdo de um novo nó a ser inserido na lista. Como a lista estava vazia (v. **Figura 12–42**), ela passa a contar com apenas um nó (v. **Figura 12–43**).

Finalmente, a codificação do string **"bola"** pode ser concluída com a remoção do último nó restante na lista e a obtenção da árvore de codificação apresentada na **Figura 12–44**. Você acha que a árvore apresentada nessa

figura não parece uma árvore de codificação? Aguarde mais um pouco e você verá que realmente essa árvore representa uma codificação.

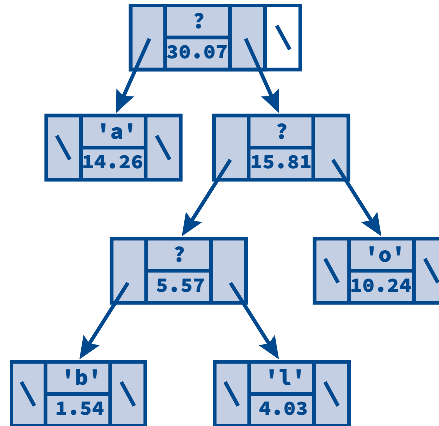


Figura 12–43: Lista após Inserção do Terceiro Nó Criado na Codificação de Huffman

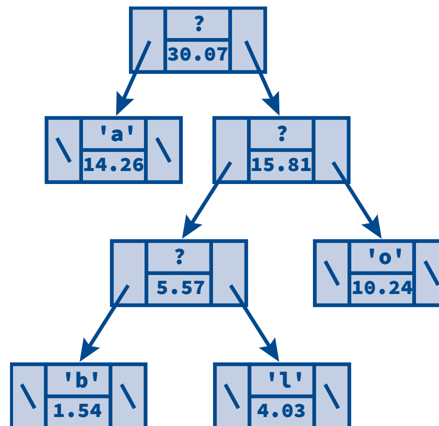


Figura 12–44: Árvore Resultante de Codificação de Huffman

A função `CodificaHuff()` implementa o que foi exposto acima e seu único parâmetro é o endereço de um ponteiro para a lista que armazena temporariamente os nós da árvore binária que representa uma codificação de Huffman. Essa função retorna o endereço da raiz da árvore binária que representa a codificação de Huffman criada.

```
tArvoreHuff CodificaHuff(tListaSE *aLista)
{
   tNoListaSE *no1, /* Ponteiro para o nó com a menor frequência na lista */
              *no2, /* Ponteiro para o nó com a segunda menor frequência na lista */
              *noNovo; /* Ponteiro para um novo nó da árvore a ser criado. */
                       /* Esse nó terá no1 e no2 como filhos.              */

      /*********************************************************************/
      /* O laço while a seguir é a essência do algoritmo de Huffman. Ele    */
      /* remove da lista os dois nós com menor frequência, cria um novo     */
      /* nó que tem como frequência a soma das frequências dos nós inserido */
      /* removidos e esses nós como filhos. O novo nó é lista. em ordem     */
      /* crescente de frequência na Quando resta apenas um nó na lista,     */
      /* esse último nó é a raiz da árvore de codificação.                  */
      /*********************************************************************/
```

```
   while (*aLista) {
         /* Remove o nó com a menor frequência da lista */
      no1 = RemovePrimeiroLSE(aLista);

         /* Se a lista ficou vazia, o último nó removido é */
         /* a raiz da árvore que representa a codificação  */
      if (!*aLista)
         return &no1->conteudo; /* Codificação terminada */

         /* Remove o nó com a segunda menor frequência da lista */
      no2 = RemovePrimeiroLSE(aLista);

         /* Tenta alocar um novo nó */
      ASSEGURA( noNovo = malloc(sizeof(tNoListaSE)),
                "Nao foi possível alocar no da lista" );

         /* Armazena a frequência do novo nó. O conteúdo do campo 'letra' */
         /* não tem importância porque este não é um nó terminal.         */
      noNovo->conteudo.letraFreq.frequencia =
         no1->conteudo.letraFreq.frequencia +
         no2->conteudo.letraFreq.frequencia;

         /* O filho esquerdo do novo nó é o primeiro nó removido da lista */
         /* e o filho direito é o segundo nó removido da lista            */
      noNovo->conteudo.filhoEsquerdo = &no1->conteudo;
      noNovo->conteudo.filhoDireito = &no2->conteudo;

      InsereEmOrdemListaSE(aLista, noNovo); /* Insere o nó na lista */
   }

   return NULL; /* Só para satisfazer o compilador... */
}
```

A função `CodificaHuff()` é relativamente simples e não requer comentários além daqueles que se encontram na própria função.

Neste ponto, já se tem a árvore de codificação, mas cadê a codificação (i. e., as almejadas sequências de bits)? Para obter as sequências de bits associadas aos caracteres codificados numa árvore de codificação, efetuam-se caminhamentos a partir da raiz dessa árvore até que os caracteres, que se encontram em folhas da árvore, sejam encontrados. Cada vez que se segue um filho esquerdo de um nó, acrescenta-se `0` à sequência de bits do caractere em consideração. Por outro lado, quando se segue um filho direito de um nó, acrescenta-se **1** à essa sequência. A **Figura 12–45** ilustra esse procedimento e a **Figura 12–46** mostra as sequências de bits obtidas para o string **"bola"**.

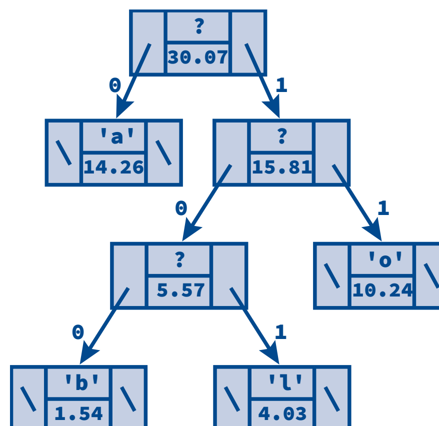


Figura 12–45: Como Códigos São Obtidos em Codificação de Huffman

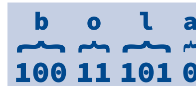


Figura 12–46: Sequências de Bits Obtidas na Codificação de Huffman

A função recursiva `ExibeCodigosHuff()` mostra como visitar os nós da árvore de codificação obtida no programa com o objetivo de apresentar as sequências de bits associadas aos caracteres que se encontram representados na árvore. Os parâmetros dessa função são:

- `raiz` (entrada) — ponteiro para a raiz da árvore contendo a codificação
- `codigo[]` (entrada e saída) — array que armazena os códigos dos caracteres
- `n` (entrada) — tamanho do array `codigo[]`

```
void ExibeCodigosHuff(tArvoreHuff raiz, char codigo[], int n)
{
      /* Armazena '0' no array quando encontra um filho esquerdo e chama */
      /* a função recursivamente para efetuar caminhamento na subárvore  */
      /* esquerda. Na prática, usa-se o bit 0 e não o caractere '0'      */
   if (raiz->filhoEsquerdo) {
      codigo[n] = '0';
      ExibeCodigosHuff(raiz->filhoEsquerdo, codigo, n + 1);
   }

      /* Armazena '1' no array quando encontra um filho direito e chama */
      /* a função recursivamente para efetuar caminhamento na subárvore */
      /* direita Na prática, usa-se o bit 1 e não o caractere '1'.      */
   if (raiz->filhoDireito) {
      codigo[n] = '1';
      ExibeCodigosHuff(raiz->filhoDireito, codigo, n + 1);
   }

      /* Se o nó corrente for uma folha,  ela armazena um caractere... */
   if (!(raiz->filhoEsquerdo) && !(raiz->filhoDireito)) {
         /* Exibe o caractere armazenado nesta folha */
      printf("%c: ", raiz->letraFreq.letra);
         /* Apresenta a codificação associada ao caractere */
      ExibeArrayDeChars(codigo, n);
   }
}
```

A função `ExibeCodigosHuff()` possui dois casos recursivos e um caso terminal e funciona assim:

- São efetuados caminhamentos na árvore de codificação, sendo que cada caminhamento começa sempre na raiz e termina numa folha dessa árvore. O número de caminhamentos é igual ao número de folhas da árvore de codificação.
- Quando, num desses caminhamentos, se segue um filho esquerdo, acrescenta-se o caractere `'0'` ao array `codigo[]`, que armazena a codificação de cada letra. Na prática, em vez de acrescentar esse caractere ao array, seria acrescentado o bit `0`, mas isso requer uso de técnicas de programação de baixo nível, que não fazem parte do escopo deste livro.
- Quando, num desses caminhamentos, se segue um filho direito, acrescenta-se o caractere `'1'` (na prática, usa-se o bit `1`) ao array `codigo[]`, que armazena a codificação de cada letra.
- Após armazenar o devido caractere no array, a função é chamada recursivamente para continuar o caminhamento a partir do nó mais recentemente visitado.
- A base da recursão é atingida quando se visita uma folha. Nesse caso, exibe-se na tela o caractere ali armazenado e chama-se a função `ExibeArrayDeChars()` para apresentar na tela a sequência de caracteres coletada no array `codigo[]` durante um dos caminhamentos descritos acima. A função

`ExibeArrayDeChars()` faz exatamente o que seu nome promete e é simples demais para precisar ser apresentada aqui.

A função **main()** que complementa o programa é apresentada a seguir:

```
int main(void)
{
   char       *string, /* String digitado pelo usuário */
              *caractereEstranho; /* Apontará para um caractere que não é letra */
   tListaSE    listaNos; /* Apontará para a lista que armazena    */
                        /* temporariamente os nós da codificação */
   tArvoreHuff arvore; /* A árvore que representa a codificação */
   char       *ar; /* Apontará para um array que armazenará o código de cada letra */
   int         op, /* Opção escolhida pelo usuário */
               nCaracteres, /* Número de caracteres armazenados */
                            /* no array apontado por 'ar'       */
               nFolhas; /* Número de folhas da árvore de codificação */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa codifica a sequencia"
           "\n\t>>> de letras que voce introduzir.\n" );

   do { /* Laço principal do programa */
         /* Lê o string */
      printf("\n\t>>> Digite uma sequencia de letras: ");
      string = LeLinhaIlimitada(NULL, stdin);

         /* Verifica se o usuário digitou apenas [ENTER] */
      if (!*string) {
         printf( "\n>>> Voce nao digitou nenhuma letra\n" );
         return 1; /* Este programa é intransigente */
      }

         /*****************************************************************/
         /* Verifica se o usuário digitou algum caractere que não é letra */
         /*****************************************************************/

      caractereEstranho = EhStringAlfa(string);

      if (caractereEstranho) {
         printf( "\n>>> Este programa nao pode processar o "
                 "caractere '%c'.\n", *caractereEstranho );
         return 1; /* Este programa é intransigente */
      }

         /* Cria a lista que armazena temporariamente os nós da codificação */
      listaNos = CriaListaSEOrd(string);

         /* O  número de folhas da árvore de codificação */
         /* é igual ao comprimento inicial da lista      */
      nFolhas = ComprimentoListaSE(listaNos);

      arvore = CodificaHuff(&listaNos); /* Cria a codificação de Huffman */

      /*******************************************************************/
      /* A função que exibe a codificação usa um array para armazenar    */
      /* os códigos à medida que visita os nós da árvore de codificação. */
      /* Portanto, o tamanho desse array deve ser pelo menos igual à     */
      /* profundidade da árvore. Como toda árvore de codificação de      */
      /* Huffmané estritamente binária essa profundidade deve ser pelo   */
      /* menos igual ao número de folhas da árvore.                      */
      /*******************************************************************/
```

```
      /* Tenta alocar o array que armazenará cada sequência de bits */
   ASSEGURA( ar = malloc(nFolhas), "Nao foi possivel"
             " alocar array para conter codificacao" );

      /* Inicialmente, não há nenhum caractere no array ar[] */
   nCaracteres = 0;

      /* Exibe na tela a codificação de Huffman obtida para cada caractere */
   ExibeCodigosHuff(arvore, ar, nCaracteres);

      /* Verifica se o usuário ainda deseja brincar */
   printf("\n\t>>> Deseja continuar codificando (s/n)? ");
   op = LeOpcao("sSnN");

   if (op == 's' || op == 'S') {
         /* A árvore criada na última codificação não é mais necessária */
      DestroiArvoreBin(arvore);

         /* O espaço ocupado pelo string lido foi alocado */
         /* dinamicamente e não é mais necessário         */
      free(string);

         /* O espaço ocupado pelo array ar[] não é mais necessário */
      free(ar);
   }
} while (op == 's' || op == 'S');

return 0;
}
```

Comentários adicionais para a função **main()**:

- A função **main()** lê strings introduzidos pelo usuário, certificando-se que strings que não contenham apenas letras não sejam aceitos. Isso ocorre porque o programa só contém frequências para letras, mas esse fato não constitui nenhuma limitação séria do programa.
- Em seguida, a função `CriaListaSEOrd()`, que já foi discutida, é chamada para criar a lista encadeada que armazena os nós da árvore de codificação.
- Logo após o retorno dessa função, a referida lista armazena apenas as folhas da árvore de codificação (v. **Figura 12–37**). Portanto a função `ComprimentoListaSE()` é chamada para calcular o comprimento da lista, que corresponde ao número de folhas da árvore. Esse valor será utilizado mais adiante para dimensionar um array a ser utilizado para coletar as sequências de bits associadas às letras codificadas.
- A função `CodificaHuff()`, discutida acima, é chamada para criar a árvore de codificação.
- O que resta a ser feito é chamar a função `ExibeCodigosHuff()`, que já foi discutida, para apresentar na tela as sequências de bits associadas às letras codificadas (v. **Figura 12–46**). Essa função utiliza um array para armazenar os bits à medida que visita nós. A pergunta mais pertinente neste ponto é: *qual deve ser o tamanho desse array*?

  Se você examinar com atenção a **Figura 12–45** que apresenta uma árvore de codificação, concluirá que o tamanho mínimo desse array deve ser igual à altura da árvore que, na referida figura é 4. Para não precisar chamar uma função que calcula essa altura (v. **12.4.4**), pode-se estimá-la levando em consideração algumas propriedades da árvore de codificação de Huffman.

  Conforme já foi antecipado, as árvores resultantes do algoritmo de Huffman são estritamente binárias (v. **Seção 12.2.2**), o que quer dizer que um nó dessa árvore ou é folha ou possui dois filhos (i.e., não

existe nó com apenas um filho nessas árvores). Portanto, numa árvore de codificação, tem-se, de acordo com o **Teorema 12.3**, que a seguinte relação é válida:

$$2p - 1 \leq n \Rightarrow p \leq \frac{n+1}{2} \qquad (\dagger)$$

Nessa relação, n é o número total de nós e p é a altura da árvore.

Agora, de acordo com o **Teorema 12.4**, o número total de nós de uma árvore estritamente binária é duas vezes o número de folhas menos um. Isto é:

$$n = 2n_0 - 1$$

Levando esse fato em consideração na relação marcada com **(†)**, tem-se que a altura máxima de uma árvore estritamente binária é igual ao seu número de folhas. Ou seja:

$$p \leq n_0$$

O restante da função **main()** tem pouco a ver com a codificação de Huffman e não requer comentários adicionais.

Exemplo de execução do programa:

```
        >>> Este programa codifica a sequencia
        >>> de letras que voce introduzir.

        >>> Digite uma sequencia de letras: amor
a: 0
o: 10
m: 110
r: 111

        >>> Deseja continuar codificando (s/n)? s

        >>> Digite uma sequencia de letras: roma
a: 0
o: 10
m: 110
r: 111
        >>> Deseja continuar codificando (s/n)? s

        >>> Digite uma sequencia de letras: AaBb
a: 0
B: 100
b: 101
A: 11
        >>> Deseja continuar codificando (s/n)? n
```

Adendo: Embora o enunciado do problema não solicite, o leitor se sentiria frustrado se não fosse discutido como uma sequência de bits pode ser decodificada de modo a resultar no caractere que lhe deu origem.

A decodificação de sequências de bits obtidas por meio de uma codificação de Huffman utiliza a mesma árvore resultante da codificação das mesmas sequências. Cada sequência é decodificada percorrendo-se um caminho na árvore que começa na raiz e termina num nó folha. Nesse processo, cada bit na sequência é analisado para determinar qual é o próximo nó a ser seguido, como ilustra a **Figura 12–47**. Nessa figura, nota-se que quando o bit ora em consideração é `0`, o caminhamento segue o filho esquerdo do nó corrente. Caso contrário, quando o bit corrente é `1`, o caminhamento segue o filho direito.

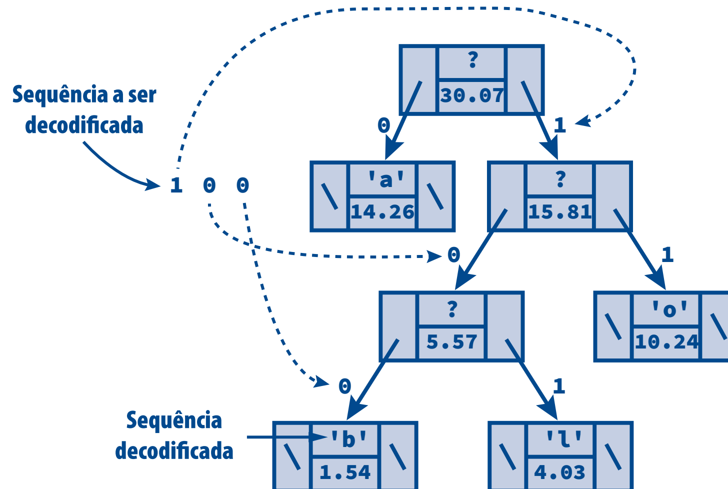


Figura 12–47: Decodificação de Bits na Codificação de Huffman

Para decodificar várias sequências resultantes da codificação de uma palavra inteira, como aquela mostrada na **Figura 12–46**, basta seguir o mesmo procedimento descrito acima a partir do início de cada subsequência que representa uma letra. Resta saber como se determina o início de cada subsequência de bits. Mas, a resposta a essa dúvida é fácil: o início da primeira subsequência coincide exatamente com o início da sequência completa de bits. A segunda subsequência começa logo após o final da primeira subsequência, a terceira subsequência começa logo após o final da segunda subsequência e assim por diante. Agora, como é que se sabe que uma subsequência terminou? Se não souber responder essa questão, você não entendeu bem o parágrafo anterior. Portanto leia-o novamente.

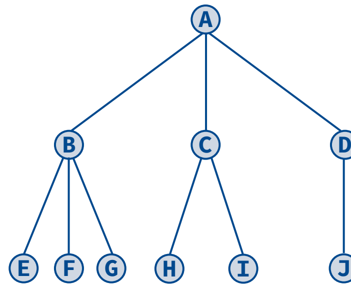
No site dedicado ao livro na internet (***http://www.ulysseso.com/ed1***), encontra-se um exemplo mais completo de codificação de Huffman.

12.9 Exercícios de Revisão

Conceitos Fundamentais (Seção 12.1)

1. (a) O que é uma estrutura de dados linear? (b) O que é uma estrutura de dados hierárquica?
2. O que é uma árvore ordinária?
3. Apresente três exemplos de aplicações práticas de árvores.
4. Por que árvores são consideradas estruturas hierárquicas?
5. Defina os seguintes conceitos relacionados à estrutura de dados árvore:
   (a) Nó
   (b) Raiz
   (c) Folha
   (d) Filho de um nó
   (e) Grau de um nó
   (f) Grau de uma árvore
   (g) Nó interno
   (h) Ancestral de um nó
   (i) Nível de um nó
   (j) Profundidade (ou altura) de uma árvore

Utilize a árvore representada na figura abaixo para responder as questões **6** e **7**.



6. (a) Qual é o grau da árvore? (b) Qual é a raiz da árvore? (c) Quais são as folhas da árvore? (d) Qual é a altura da árvore? (e) Quantos filhos possui o nó rotulado como `C`? (f) Quantos ancestrais possui o nó `J`?

7. Para cada nó da árvore, apresente:

(a) O pai do nó

(b) Os filhos do nó

(c) Os ancestrais do nó

(d) O nível do nó

8. Mostre que o número mínimo de nós de uma árvore com profundidade p é igual a p.

9. Mostre que a raiz de uma árvore é ancestral de qualquer nó da árvore, exceto de si própria.

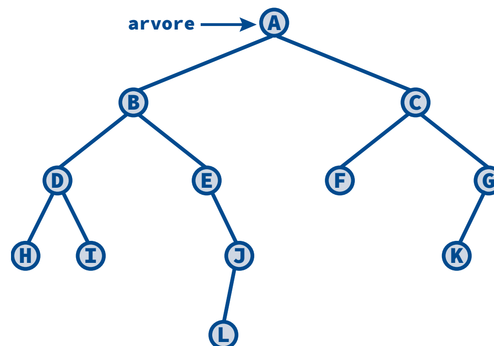
Árvores Binárias (Seção 12.2)

10. O que é uma árvore binária?

11. *Uma árvore binária é um caso particular de árvore*. Essa afirmativa é correta? Explique.

12. Qual é a diferença entre uma árvore binária e uma árvore ordinária na qual cada nó possui no máximo dois filhos.

13. Considere a árvore binária representada na figura a abaixo.



(a) Quais são os descendentes do nó `B`?

(b) Quais são os ancestrais do nó `K`?

(c) Qual é a altura da árvore?

(d) Qual é o nível do nó `J`?

(e) Qual é o maior número possível de nós no nível em que se encontra o nó `L`?

14. Quantos ancestrais possui um nó que se encontra no nível n de uma árvore binária? Justifique sua resposta.

15. Suponha que duas árvores binárias contenham apenas uma raiz com um filho e que o conteúdo desses filhos sejam iguais. Essas árvores podem ser consideradas equivalentes?

16. Suponha que uma árvore binária seja representada em memória por meio de três arrays integrados conforme mostrado a seguir (a primeira linha contém os índices dos arrays):

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Conteudo	10	20	30	40	50	60	70	80			25	32	44	83
Esquerdo	-1	0	-1	-1	1	-1	-1	6			-1	2	10	-1
Direito	-1	12	-1	-1	5	7	-1	13			11	3	-1	-1

Sabendo que a raiz da árvore encontra-se no índice `4` do array, apresente uma representação gráfica dessa árvore.

17. (a) Quantas árvores binárias estruturalmente diferentes contendo três nós existem? (b) Desenhe todas as árvores binárias estruturalmente diferentes possíveis contendo três nós.

18. (a) O que é uma árvore binária perfeita? (b) Apresente um exemplo de árvore binária perfeita.

19. (a) O que é uma árvore binária completa? (b) Apresente um exemplo de árvore binária completa.

20. Prove que cada nó de uma árvore binária possui no máximo um pai.

21. Defina os seguintes tipos de árvores:

(a) Árvore binária completa
(b) Árvore binária repleta
(c) Árvore binária perfeita
(d) Árvore binária 0/2
(e) Árvore binária inclinada
(f) Árvore estritamente binária
(g) Árvore binária patológica

22. Uma árvore binária perfeita contém n_0 folhas. Quantos nós essa árvore possui?

23. Uma árvore binária possui profundidade p. Qual é o número máximo de nós que essa árvore possui?

24. Qual é o número máximo de nós no nível k de uma árvore binária?

25. Suponha que a raiz de uma árvore binária possua nível 0 (o que não é o caso neste livro). (a) Qual seria o número máximo de nós que se encontram no nível i de uma árvore binária? (b) Qual seria o número máximo de nós que pode possuir uma árvore binária com n níveis?

26. Mostre que ao nó mais à esquerda no nível n de uma árvore binária completa é sempre atribuído o valor $2^{n-1} - 1$, segundo o esquema de numeração de nós visto na **Seção 12.2.4**.

27. Mostre que uma árvore binária com n nós possui $n + 1$ ramificações nulas.

28. (a) Descreva o procedimento utilizado para representar árvores binárias em arrays. (b) Utilizando essa forma de representação, em que circunstância ocorre o maior desperdício de memória? (c) Utilizando essa forma de representação, quando ocorre o menor desperdício de memória?

29. (a) Qual é a altura máxima que uma árvore binária com 100 nós pode ter? (b) Qual é a altura mínima que uma árvore binária com 100 nós pode ter?

30. Quantos nós internos existem numa árvore binária completa com 200 folhas?

31. Mostre que o número de ancestrais de uma folha de uma árvore binária perfeita com n nós é dado por $\lfloor log(n + 1) \rfloor - 1$.

32. Suponha que você tenha um array de elementos do tipo **int** contendo os seguintes valores (nesta ordem):

```
4  1  12  -3  5  16  6  18  0
```

Apresente uma árvore binária construída do seguinte modo:

(1) O primeiro elemento do array constitui o conteúdo da raiz da árvore.

(2) Se um elemento subsequente do array tiver um valor menor do que aquele armazenado na raiz da árvore, ele deve ser inserido na subárvore esquerda utilizando esse mesmo procedimento.

(3) Se um elemento subsequente do array tiver um valor maior do que aquele armazenado na raiz da árvore, ele deve ser inserido na subárvore direita utilizando esse mesmo procedimento.

(Uma árvore binária construída seguindo esse procedimento é denominada **árvore binária de busca** e será explorada em profundidade no **Volume 2** desta obra.)

33. (a) Mostre que se o procedimento da questão **32** for seguido com um array ordenado em ordem crescente, a árvore binária resultante será inclinada à direita. (b) Mostre que se o procedimento da questão **32** for seguido com um array ordenado em ordem decrescente, a árvore binária resultante será inclinada à esquerda.

34. Considerando a expressão *(–2x + 1)(3x – 5)* e utilizando símbolos de operadores aritméticos de C, desenhe uma árvore binária que represente essa expressão.

35. Um **nó completo** de uma árvore binária é um nó que possui dois filhos. Mostre que o número de folhas de uma árvore binária não vazia é igual ao número de nós completos da árvore mais um.

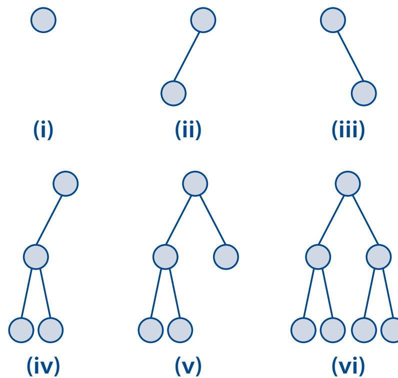
36. Numa representação de árvore binária por meio de array, como se encontra:

(a) O pai de um nó

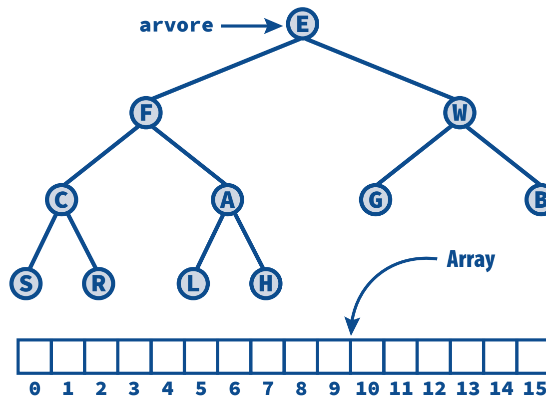
(b) O filho da esquerda de um nó

(c) O filho da direita de um nó

37. (a) Quais das árvores binárias a seguir são completas? (b) Quais delas são perfeitas? (c) Quais delas não são nem completas nem perfeitas? (d) Quais delas são estritamente binárias?



38. Mostre como a árvore binária da figura a seguir é representada no array dessa mesma figura.



39. Uma **árvore binária de Fibonacci** de ordem n é definida do seguinte modo: se $n = 0$ ou $n = 1$, a árvore consiste num único nó, e se $n > 1$, a árvore consiste numa raiz, tendo a árvore de Fibonacci de ordem $n - 1$ como sua subárvore esquerda e a árvore de Fibonacci de ordem $n - 2$ como sua subárvore direita. (a) Essa árvore é estritamente binária? (b) Qual é o número de folhas na árvore de Fibonacci de ordem n?

Caminhamentos em Árvores Binárias (Seção 12.3)

40. O que significa visitar um nó de uma árvore?

41. (a) O que é um caminhamento em árvore binária? (b) Para que servem caminhamentos em árvores binárias?

42. Descreva os seguintes caminhamentos em árvore binária:

(a) Caminhamento infixo

(b) Caminhamento prefixo

(c) Caminhamento sufixo

(d) Caminhamento por nível

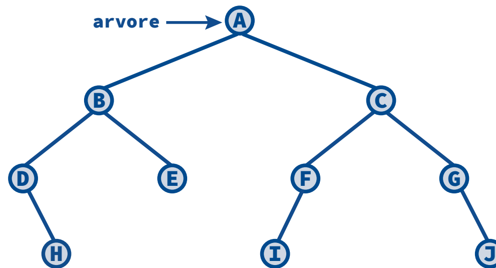
43. Mostre que as folhas de uma árvore binária são visitadas na mesma ordem em cada um dos três caminhamentos recursivos.

44. Quais são as sequências de visitação dos nós da árvore binária da figura abaixo quando são efetuados caminhamentos em:

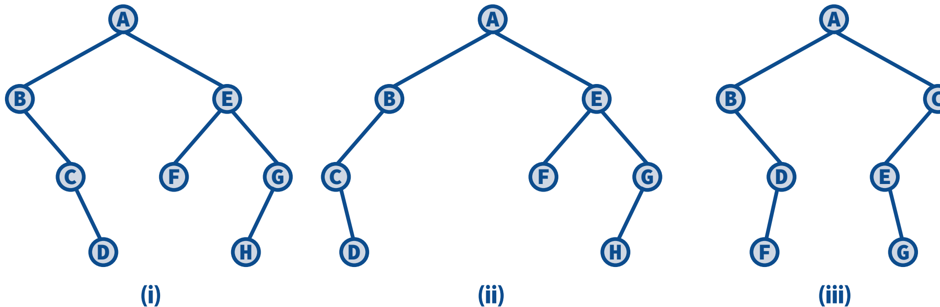
(a) Ordem infixa

(b) Ordem prefixa

(c) Ordem sufixa



45. Considere as árvores binárias (i), (ii) e (iii) apresentadas na figura a seguir.



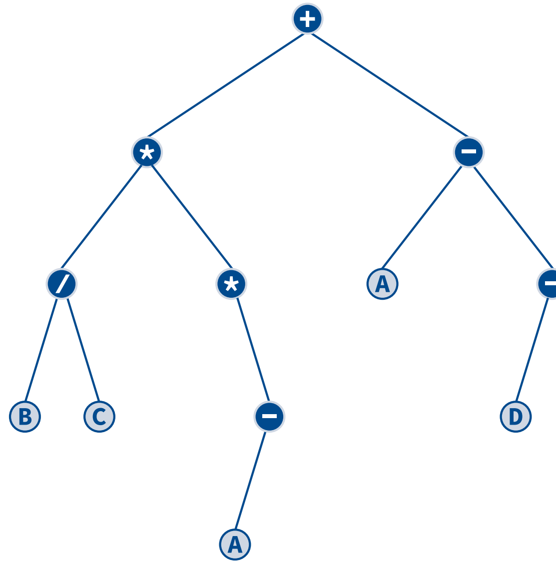
(a) Qual é a sequência de visitação de nós produzida por um caminhamento infixo em cada árvore?

(b) Qual é a sequência de visitação de nós produzida por um caminhamento prefixo em cada árvore?

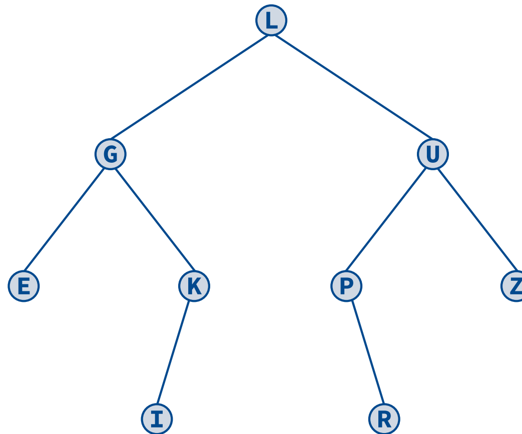
(c) Qual é a sequência de visitação de nós produzida por um caminhamento sufixo em cada árvore?

46. Desenhe uma árvore binária correspondente à expressão aritmética: $(A + B*C)/(A - C)$.

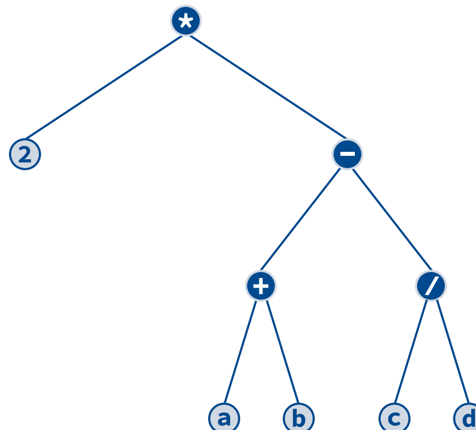
47. Considere a árvore binária ilustrada na figura a seguir. Essa árvore representa uma expressão aritmética? Explique seu raciocínio.



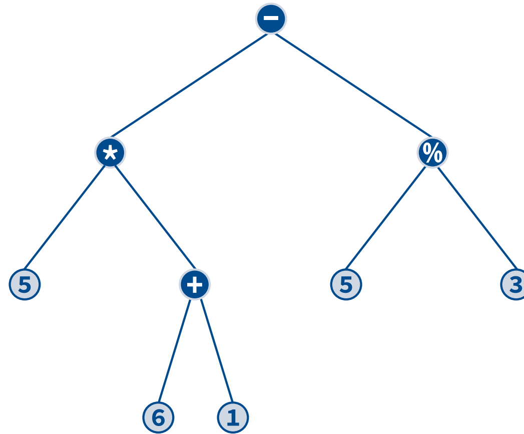
48. Onde deve ser inserido um nó contendo a letra `N` na árvore binária da figura abaixo, de tal forma que, no caminhamento infixo na árvore, os nós sejam visitados em ordem alfabética?



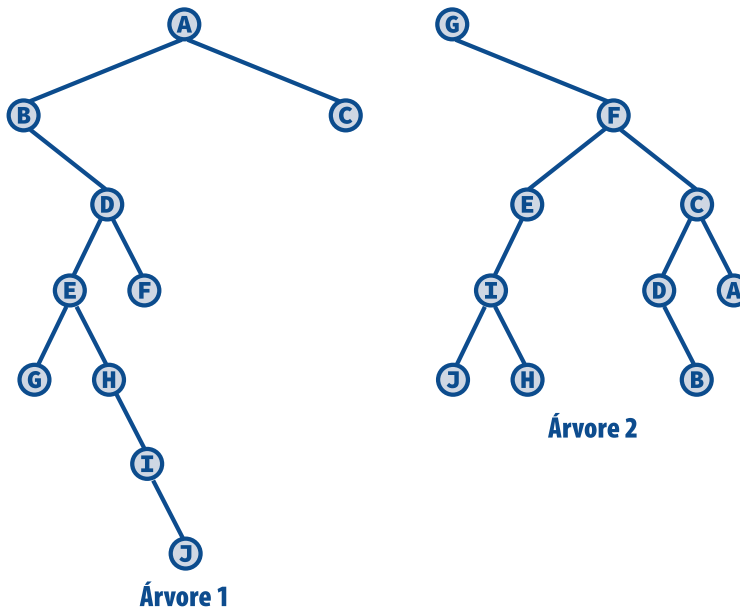
49. Qual é a expressão aritmética infixa representada pela árvore binária da figura a seguir?



50. Qual é o resultado da avaliação da expressão aritmética representada pela árvore da figura a seguir?



51. Considerando as árvores binárias da figura a seguir, quais caminhamentos produzem as mesmas sequências de visitação de nós nas duas árvores?



52. Os seguintes caminhamentos em árvores binárias sã\o denominados **caminhamentos árabes** devido à analogia com escrita árabe que é efetuada da direita para a esquerda:

Caminhamento Árabe Infixo	Caminhamento Árabe Prefixo
1. Caminhe na subárvore direita	1. Visite a raiz
2. Visite a raiz	2. Caminhe na subárvore direita
3. Caminhe na subárvore esquerda	3. Caminhe na subárvore esquerda

(a) Usando analogia com os caminhamentos apresentados acima, descreva o caminhamento árabe sufixo.

(b) Efetue um caminhamento árabe infixo na árvore do exercício 48.

(c) Efetue um caminhamento árabe prefixo na árvore do exercício 48.

Implementação de Árvores Binárias (Seção 12.4)

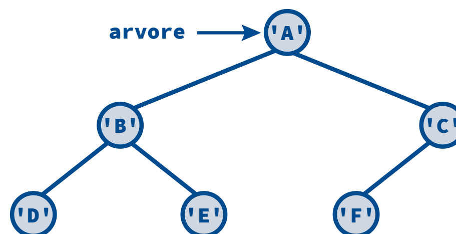
53. Suponha que se tenham as seguintes definições de tipos:

```
typedef struct noArvore {
           struct noArvore *esquerda;
           char             conteudo;
           struct noArvore *direita;
        } tNoArvore, *tArvore;

typedef struct noLista {
           tArvore         conteudo;
           struct noLista *proximo;
        } tNoLista, *tLista;

struct {
   tNoLista *frente, *fundo;
} *tFila;
```

Suponha ainda que `CriaFila()`, `FilaVazia()`, `Acrescenta()` e `Retira()` são funções que, respectivamente, iniciam, verificam se uma fila está vazia, acrescentam e retiram elementos de uma fila, como aquelas apresentadas na **Seção 8.2**. O que a função `Misteriosa()` apresentada a seguir realiza?

```
void Misteriosa(tArvore ptrArvore)
{
   tFila fila;

   CriaFila(&fila);

   Acrescenta(&fila, &ptrArvore);

   while (!FilaVazia(&fila)) {
      ptrArvore = Retira(&fila);

      if (ptrArvore) {
         putchar(ptrArvore->conteudo);
         Acrescenta(&fila, &ptrArvore->esquerda);
         Acrescenta(&fila, &ptrArvore->direita);
      }
   }
}
```

54. Se a função `Misteriosa()` da questão **53** for chamada recebendo como parâmetro o ponteiro `arvore` da figura a seguir, qual será o resultado?



55. Por que ponteiros para funções são usados como parâmetros de funções que implementam caminhamentos em árvores binárias?

56. Por que a maioria das operações sobre árvores binárias são implementadas recursivamente?

57. (a) O que são árvores binárias semelhantes? (b) Qual é a diferença entre árvores binárias semelhantes e árvores binárias equivalentes?

58. A função `DestroiArvoreBin()`, apresentada na **Seção 12.4.8**, poderia ter sido implementada por meio de um caminhamento em ordem prefixa ou infixa? Explique.

Árvores Binárias Baseadas em Caminhamentos (Seção 12.5)

59. Os caminhamentos infixo e prefixo de uma árvore binária produziram as sequências de visitação de nós vistas a seguir. Desenhe um diagrama mostrando a estrutura dessa árvore.

Caminhamento infixo: `F B C A G I E D H`

Caminhamento prefixo: `G B F A C I D E H`

60. (a) Uma árvore binária pode ser construída conhecendo-se apenas seu caminhamento infixo? (b) E se apenas o caminhamento prefixo ou sufixo estiver disponível?

61. Quais dos seguintes pares de caminhamentos permite a construção da árvore binária que dá origem a esses caminhamentos?

(a) Infixo e sufixo

(b) Infixo e prefixo

(c) Prefixo e sufixo

62. Suponha que se tenham disponíveis os caminhamentos prefixo e infixo de uma árvore binária. Em que situação, esses caminhamentos podem resultar numa árvore binária diferente da árvore que lhes deu origem?

63. Suponha que um caminhamento infixo seja efetuado numa árvore e que a operação efetuada sobre cada nó durante esse caminhamento seja copiar seu conteúdo efetivo para um array. Se a árvore original for destruída, será possível reconstruí-la usando os conteúdos dos nós armazenados no referido array?

64. (a) Desenhe um diagrama que represente a árvore binária completa cujo caminhamento por nível fornece a sequência de visitação de nós: `DGHCIBAEF`. (b) Se não fosse conhecido que a árvore é completa, seria possível construí-la?

65. Sabendo que uma árvore binária é completa, é possível construí-la conhecendo-se apenas seus caminhamentos prefixo e sufixo?

Árvores Binárias Costuradas (Seção 12.6)

66. Por que se utiliza pilha na implementação de um caminhamento iterativo em uma árvore binária?

67. (a) O que é uma árvore binária costurada? (b) Para que servem costuras numa árvore binária?

68. Descreva os seguintes tipos de árvores:

(a) Árvore binária costurada em ordem infixa à direita

(b) Árvore binária costurada em ordem infixa à esquerda

(c) Árvore binária costurada em ordem infixa

69. Uma árvore costurada em ordem infixa tem alguma utilidade quando se efetua um caminhamento prefixo?

70. Que cuidado deve ser tomado quando se efetua um caminhamento prefixo ou sufixo numa árvore costurada em ordem infixa?

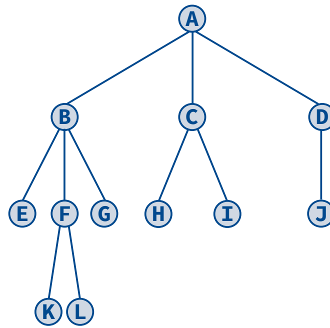
71. Por que uma enumeração é usada como um dos campos de um nó de uma árvore costurada?

72. O que ocorre quando uma costura é seguida em vez de um ponteiro que não é costura durante um caminhamento?

Conversões de Árvores Ordinárias e Florestas em Árvores Binárias (Seção 12.7)

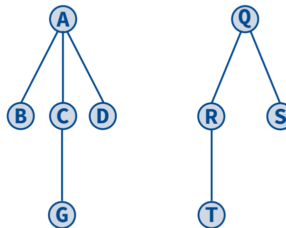
73. Descreva o procedimento utilizado para transformar uma árvore ordinária numa árvore binária.

74. Transforme a árvore ordinária da figura abaixo numa árvore binária.



75. Por que a raiz de qualquer árvore binária resultante de uma transformação de árvore ordinária não possui filho à direita?

76. Transforme a floresta representada na figura a seguir em árvore binária.



Exemplos de Programação (Seção 12.8)

77. O que é uma árvore de decisão?

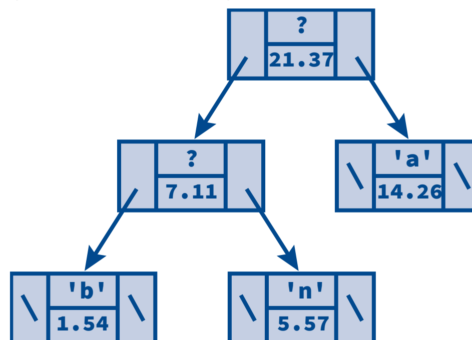
78. Descreva o algoritmo de codificação de Huffman.

79. Por que a árvore resultante de uma codificação de Huffman é estritamente binária?

80. Mostre que se o número de caracteres codificados for igual a n, a árvore de codificação de Huffman conterá n folhas e $n - 1$ nós internos.

81. A implementação do algoritmo de codificação de Huffman apresentada na **Seção 12.8.2** tem custo temporal $\theta(n^2)$. Existem implementações do mesmo algoritmo que apresentam custo temporal $\theta(log\ n)$, o que parece indicar que a implementação apresentada aqui não é a mais adequada. Que argumentos de natureza prática, você utilizaria para justificar o uso da implementação apresentada na **Seção 12.8.2**?

82. Suponha que a figura a seguir represente uma árvore codificação obtida seguindo-se o algoritmo de Huffman. (a) Quais são os caracteres codificados nessa árvore? (b) Qual é a sequência de bits associada a cada um desses caracteres? (c) O que significam as interrogações nessa árvore?



83. Considerando a árvore de codificação do exercício anterior, como a sequência de bits `001011011` seria decodificada?

12.10 Exercícios de Programação

Nos exercícios de programação a seguir, a não ser que seja especificado em contrário, considere o conteúdo efetivo de qualquer nó de árvore ou lista como sendo **int**.

EP12.1 Escreva uma função em C para determinar se uma árvore binária é repleta.

EP12.2 Duas árvores binárias são **semelhantes** se elas são ambas vazias, ou se suas subárvores esquerdas são semelhantes e suas subárvores direitas são semelhantes. Escreva uma função em C para determinar se duas árvores binárias são semelhantes.

EP12.3 Duas árvores binárias são **reflexivamente semelhantes** se são ambas vazias, ou se são ambas não vazias e a subárvore esquerda de uma árvore é reflexivamente semelhante à subárvore direita da outra e vice-versa. Escreva uma função em C que determina se duas árvores binárias são reflexivamente semelhantes.

EP12.4 Suponha que o conteúdo armazenado nos nós de uma árvore binária seja do tipo **int**. Escreva uma função recursiva que exibe o conteúdo de cada ancestral de um nó cujo endereço essa função recebe como parâmetro.

EP12.5 Suponha que o conteúdo armazenado nos nós de uma árvore binária seja do tipo **int**. Escreva uma função que retorna o número de nós que apresentam um valor menor do que o valor recebido como parâmetro.

EP12.6 Escreva uma função em C que receba como parâmetro um ponteiro para uma árvore binária e retorne o número de nós de grau dois da árvore.

EP12.7 Escreva uma função em C que receba como parâmetro um ponteiro para uma árvore binária e retorne o número de folhas da árvore.

EP12.8 Escreva uma função em C que receba como parâmetro um ponteiro para uma árvore binária e retorne o número de ramificações da árvore.

EP12.9 Escreva uma função em C que recebe um ponteiro para uma árvore binária e retorne um ponteiro para uma nova árvore binária que é a imagem da primeira. Isto é, todas as subárvores esquerdas da primeira serão subárvores direitas na segunda e vice-versa.

EP12.10 Escreva uma função em C que recebe como parâmetros um ponteiro para uma árvore binária e outro para um nó qualquer dessa árvore e retorne o nível desse nó na árvore.

EP12.11 Escreva uma função em C que recebe como parâmetro um ponteiro para uma árvore binária e retorna `1` se a árvore for estritamente binária ou `0` caso contrário.

EP12.12 Numa árvore binária de Fibonacci de ordem n, se $n = 0$ ou $n = 1$, a árvore consiste num único nó, e se $n > 1$, a árvore consiste numa raiz, tendo a árvore de Fibonacci de ordem $n - 1$ como sua subárvore esquerda e a árvore de Fibonacci de ordem $n - 2$ como sua subárvore direita. Escreva uma função em C que retorna um ponteiro para a árvore binária de Fibonacci de ordem n.

EP12.13 Escreva uma função recursiva que armazena numa lista simplesmente encadeada os conteúdos efetivos dos nós de uma árvore binária obtidos durante um caminhamento infixo nessa árvore. Suponha que o conteúdo efetivo de cada nó da árvore seja do tipo **int**.

EP12.14 Suponha que o tipo de valor armazenado como conteúdo efetivo em cada nó de uma árvore binária seja **int**. Dadas duas listas simplesmente encadeadas contendo os caminhamentos infixo e prefixo nessa árvore, escreva uma função que retorne o endereço da raiz da árvore construída utilizando esses caminhamentos.

EP12.15 Escreva uma função para determinar se uma árvore binária é completa.

EP12.16 Escreva uma função em C que remove todas as folhas de uma árvore binária.

EP12.17 Escreva uma função que retorna o menor valor armazenado numa árvore binária cujo campo de informação de cada nó é do tipo **int**.

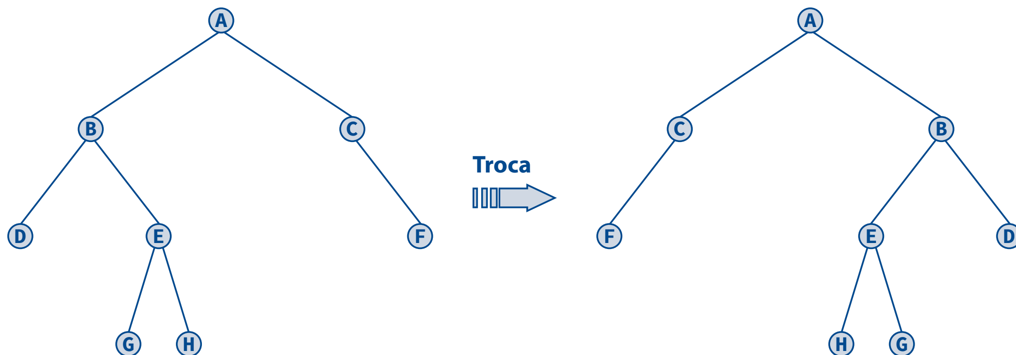
EP12.18 Escreva uma função que retorna uma lista encadeada contendo todos os ancestrais de um nó de uma árvore binária cujo endereço é recebido como parâmetro.

EP12.19 Escreva uma função que recebe o endereço de um nó de uma árvore binária e retorna o endereço do nó que segue imediatamente o nó recebido como parâmetro num caminhamento em ordem sufixa na árvore.

EP12.20 Escreva uma função que recebe o endereço de um nó de uma árvore binária e retorna o endereço do nó que segue imediatamente o nó recebido como parâmetro num caminhamento em ordem infixa na árvore.

EP12.21 Escreva uma função em C que exiba na tela o conteúdo do tipo **int** de cada ancestral de um determinado nó de uma árvore binária.

EP12.22 Escreva uma função em C que implemente a operação de **troca** sobre uma árvore binária. Essa operação consiste em trocar o filho direito com o filho esquerdo de um nó e vice-versa, como mostra a seguinte figura. (A árvore original e a árvore resultante dessa operação são denominadas **isomórficas**.)



EP12.23 Escreva uma função que recebe dois ponteiros para duas árvores binárias e retorna `1` se as árvores para as quais eles apontam são isomórficas e `0` em caso contrário. A definição de árvores isomórficas é apresentada no exercício **EP12.22**.

EP12.24 Escreva uma função iterativa para executar caminhamentos prefixos em árvores binárias. [**Sugestão:** Examine a implementação da função `CaminhamentoInfixo2()` implementada na **Seção 12.6**.]

EP12.25 Escreva uma função iterativa para executar caminhamentos sufixos em árvores binárias. [**Sugestão:** Examine a implementação da função `CaminhamentoInfixo2()` implementada na **Seção 12.6**.]

EP12.26 Escreva uma função em C para exibir os campos de informação dos nós de uma árvore binária por nível. Em cada nível, os nós devem ser listados da direita para a esquerda.

EP12.27 Apresente definições de tipo e funções que implementem árvores binárias costuradas em ordem prefixa de modo semelhante ao que foi realizado na **Seção 12.6**.

EP12.28 Apresente definições de tipo e funções que implementem árvores binárias costuradas em ordem sufixa de modo semelhante ao que foi realizado na **Seção 12.6**.

EP12.29 Escreva uma função para calcular a soma de todos os conteúdos efetivos de uma árvore binária supondo que esses conteúdos são do tipo **int**.

EP12.30 Escreva uma função para determinar se uma árvore binária é estritamente binária.

EP12.31 Escreva uma função que recebe como parâmetro um ponteiro para uma árvore binária e retorna o número de folhas dessa árvore.

EP12.32 Escreva uma função que recebe como parâmetro um ponteiro para uma árvore binária e retorna o número de filhos direitos dessa árvore.

EP12.33 Escreva uma função que recebe como parâmetro um ponteiro para uma árvore binária e retorna o número de nós completos (i.e., nós com dois filhos) dessa árvore.

EP12.34 Escreva uma função em C para determinar se uma árvore binária é perfeita.

EP12.35 Dadas duas árvores binárias A e B, diz-se que A **está contida** em B se e somente se A é igual a B ou A é igual a alguma subárvore de B. Construa uma função em C que teste se uma árvore binária A está contida em outra árvore binária B.

EP12.36 Escreva uma função em C que receba como parâmetro um ponteiro para uma árvore binária, e retorne `1` se a árvore for estritamente binária ou `0` caso contrário.

EP12.37 Escreva uma função em C que receba um ponteiro para uma árvore binária e retorne um ponteiro para uma nova árvore binária que seja a imagem da primeira. Isto é, todas as subárvores esquerdas da primeira serão subárvores direitas na segunda e vice-versa.

EP12.38 Incremente o programa apresentado na **Seção 12.8.2** de modo que ele apresente a codificação de cada string introduzido pelo usuário (e não simplesmente a codificação de cada letra individualmente).

EP12.39 Torne o programa solicitado no exercício **EP12.37** ainda mais sofisticado fazendo com que ele permita a seguinte interação com o usuário:

```
       Opcoes

1. Codifica caracteres
2. Decodifica caracteres
3. Exibe codigos
4. Encerra programa

        >>> Escolha sua opcao: 1

        >>> Digite uma sequencia de letras: banana

Codificacao: 1100010 100 11111 100 11111 100

        Opcoes

1. Codifica caracteres
2. Decodifica caracteres
3. Exibe codigos
4. Encerra programa

        >>> Escolha sua opcao: 2

        >>> Digite o codigo: 110001010011111100111111100

Decodificacao: banana

        Opcoes

1. Codifica caracteres
2. Decodifica caracteres
3. Exibe codigos
4. Encerra programa

        >>> Escolha sua opcao: 4

Bye
```