Capítulo 8: Estruturas

Grande parte da informática lida com a representação e manipulação de informação. Para tal, foram concebidas várias estruturas para organizar blocos de dados de uma forma que facilite o seu armazenamento, pesquisa e recuperação. Há uma disciplina completa em boa parte dos currículos de computação chamada "estruturas de dados" que cobre como implementar estas estruturas em código. Neste livro, não vamos falar do código, mas sim das próprias estruturas abstratas. Este capítulo tem muitas imagens, que descrevem exemplos das várias estruturas de uma forma muito geral. Os conceitos aqui descritos mapeiam diretamente para código, quando for necessário pô-los em prática.

Há todo o tipo de estruturas de dados — vetores, listas ligadas, filas, pilhas, hashtables, e heaps, para citar alguns — mas quase todos eles resumem-se a um de dois tipos fundamentais de coisas: **grafos**, e **árvores**. Estas são as duas estruturas em que nos vamos concentrar neste capítulo. Um grafo é praticamente a estrutura mais geral que se pode imaginar: um monte de elementos de dados dispersos que estão de alguma forma relacionados uns com os outros. Quase todas as estruturas de dados imagináveis podem ser reformuladas como um tipo de grafo. As árvores são uma espécie de caso especial de grafos, mas também uma espécie de tópico em si próprias, assim como funções eram um tipo especial de relação, mas também um pouco diferentes. Uma árvore pode ser vista como um tipo de grafo que impõe condições extras especiais que fornecem alguns benefícios para percorrê-las.

8.1. Grafos

Em muitos aspectos, a forma mais elegante, simples e poderosa de representar conhecimento é por meio de um **grafo**. Um grafo é composto por um monte de pequenos pedaços de dados, cada um dos quais pode (ou não) ser ligado a cada um dos outros. Um exemplo está na Figura 8.1. Cada uma das ovais rotuladas é chamada um **vértice**, e as linhas entre elas são chamadas **arestas**. Cada vértice contém, ou não, uma aresta que o liga a cada um dos outros vértices. Poder-se-ia imaginar cada um dos vértices contendo vários atributos descritivos — talvez a oval de Matheus Nachtergaele tivesse informação sobre a sua data de nascimento, e a oval de Rio Grande do Sul informação sobre a sua área, número de municípios e população — mas estes não são tipicamente mostrados no diagrama. Tudo o que realmente importa, em termos de grafos, é que vértices ele contém, e quais são os que se conectam a que outros.

Os psicólogos cognitivos, que estudam os processos mentais internos da mente, já há muito tempo identificaram este tipo de estrutura como a principal forma em que as pessoas mentalmente armazenam e trabalham com informação. Afinal, se refletirmos um momento e nos perguntarmos "quais são as 'coisas' que estão na minha memória", uma resposta razoável é "bem, eu sei sobre um monte de coisas, e as propriedades dessas coisas, e as relações entre essas coisas". Se as "coisas" são vértices, e as "propriedades" são atributos desses vértices, e as "relações" são as arestas, temos precisamente a estrutura de um grafo. Os psicólogos deram a isto outro nome: uma **rede semântica**. Pensa-se que a miríade de conceitos que você registrou na memória

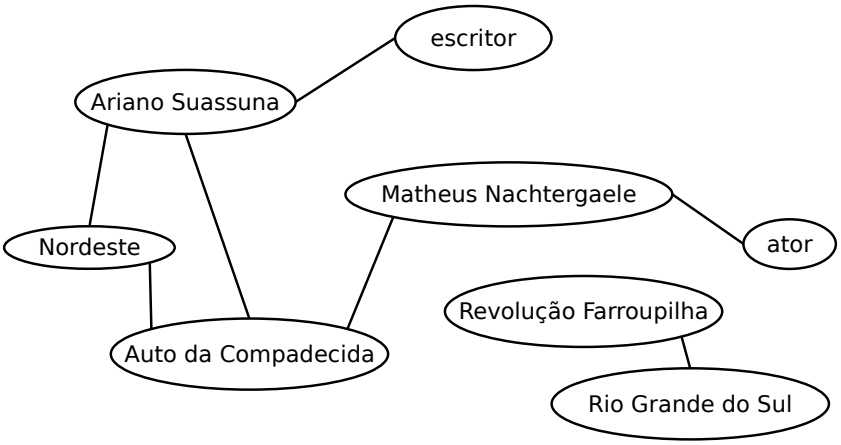

Figura 8.1: Um grafo (não direcionado).

— Ariano Suassuna, o desodorante, e seu calendário de aulas, e talvez milhões de outros — estão todos associados na sua mente numa vasta rede semântica que liga os conceitos relacionados entre si. Quando a sua mente recorda informações, ou deduz fatos, ou mesmo se desloca aleatoriamente em momentos de inatividade, está essencialmente a percorrer este grafo passando pelas várias arestas que existem entre os vértices.

Isso é profundo. Mas não é preciso ir tão fundo para ver o aparecimento de estruturas de grafos em toda a informática. O que é um mapa de um aplicativo de GPS, se não um grafo gigantesco onde os vértices são localizações transitáveis e as arestas são rotas entre eles? O que é uma rede social, se não um grafo gigante onde os vértices são pessoas e as arestas são amizades? O que é a World Wide Web, se não um grafo gigantesco onde os vértices são páginas e as arestas são hiperligações? O que é a Internet, se não um grafo enorme onde os vértices são computadores ou roteadores e as arestas são ligações de comunicação entre eles? Este esquema simples de vértices ligados é suficientemente poderoso para acomodar toda uma série de aplicações, e é por isso que vale a pena estudá-lo.

8.1.1 Termos de Grafos

O estudo dos grafos traz consigo toda uma série de novos termos que são importantes usar com precisão:

- **Vértice**. Cada grafo contém zero ou mais vértices (estes também são por vezes chamados de *nós*, *conceitos*, ou *objetos*)[1].
- **Aresta**. Cada grafo contém zero ou mais arestas (estas também são por vezes chamadas de *links*, *conexões*, *associações*, ou *relacionamentos*). Cada aresta liga exatamente dois vértices, a menos que a aresta ligue um vértice a si próprio, o que é possível, acredite ou não. Uma aresta que liga um vértice a si própria é chamada de *loop* ou *laço*.

1 A frase “zero ou mais” é comum em matemática discreta. Neste caso, ela indica que o grafo vazio, que não contém quaisquer vértices, continua a ser um grafo legítimo.

- **Caminho**. Um caminho é uma sequência de arestas consecutivas que o leva de um vértice para um outro. Na Figura 8.1, existe um caminho entre Nordeste e Matheus Nachtergaele (passando pelo Auto da Compadecida), embora não exista uma aresta direta entre os dois. Pelo contrário, não existe um caminho entre Ariano Suassuna e a Revolução Farroupilha. Não confunda os dois termos aresta e caminho: o primeiro é uma ligação direta entre dois nós, enquanto que o segundo pode ser uma travessia passo-a-passo inteira (uma aresta única, entretanto, conta como um caminho).
- **Direcionado/não direcionado** (ou **dirigido/não dirigido**). Em alguns grafos, os relacionamentos entre nós são inerentemente bidirecionais: se A está ligado a B, então B está ligado a A, e não faz sentido de outra forma. Pense numa rede social: a amizade vai sempre nos dois sentidos. Este tipo de grafo é chamado de grafo não direcionado, e tal como o exemplo de Ariano Suassuna na Figura 8.1, as arestas são mostradas como linhas retas. Noutras situações, uma aresta de A para B não implica necessariamente uma aresta também na direção inversa. Na World Wide Web, por exemplo, só porque a página A tem um link para a página B, não significa que o inverso seja verdade (normalmente não é). Neste tipo de grafo direcionado, desenhamos pontas de seta nas linhas para indicar para que sentido vai a ligação. Um exemplo é a Figura 8.2: os vértices representam pugilistas famosos, e as arestas direcionadas indicam que pugilista derrotou que outro(s). É possível que um par de vértices tenha arestas em ambas as direções — Muhammad Ali e Joe Frazier derrotaram cada um o outro (em momentos distintos, obviamente) — mas esta não é a norma, e certamente não é a regra, com um grafo direcionado.

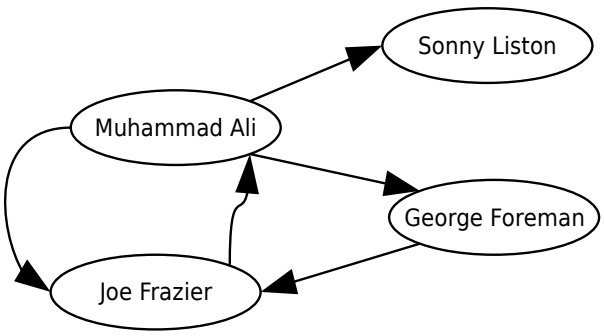

Figura 8.2: Um grafo direcionado.

- **Com pesos**. Alguns grafos, além de conterem apenas a presença (ou ausência) de uma aresta entre cada par de vértices, têm também um número em cada aresta, chamado de peso da aresta. Dependendo do grafo, este pode indicar a distância, ou custo, entre os vértices. Um exemplo está na Figura 8.3: à moda de aplicativos de navegação, este grafo contém as localizações, e a quilometragem entre elas. Um grafo pode também ser ao mesmo tempo direcionado e com pesos, a propósito. Se um par de vértices em tal grafo estiver ligado "nos dois sentidos", então cada uma das duas arestas terá o seu próprio peso.
- **Adjacente**. Se dois vértices tiverem uma aresta entre si, diz-se que são adjacentes.

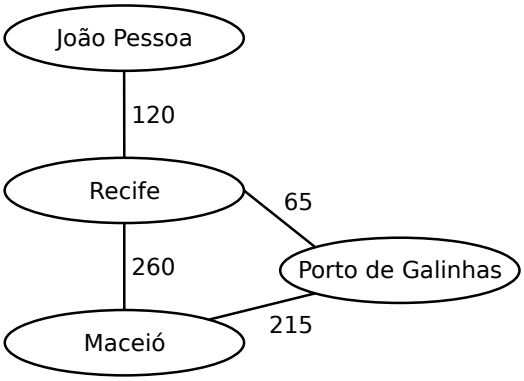

Figura 8.3: Um grafo (não direcionado) com pesos.

- **Conectividade**. A conectividade pode ter dois significados: aplica-se tanto a *pares de vértices* como a *grafos inteiros*. Dizemos que dois vértices estão **conectados** *se houver pelo menos um caminho entre eles*. Cada um destes vértices é, portanto, "*alcançável*" a partir do outro. Na Figura 8.1, Ariano Suassuna e ator estão conectados, mas o Auto da Compadecida e a Revolução Farroupilha não estão. Podemos também falar da conectividade de um grafo inteiro. Dizemos que um grafo *não direcionado* é **conexo**, se cada nó puder ser alcançado a partir de todos os outros. É fácil ver que a Figura 8.3 é um grafo conexo, enquanto que a Figura 8.1 não é (porque a Revolução Farroupilha e o Rio Grande do Sul estão isoladas dos outros nós). Para um grafo *direcionado*, dizemos que ele é conexo se a substituição de todas as suas arestas direcionadas por arestas não direcionadas produz um grafo (não-direcionado) conexo. Assim, o grafo da Figura 8.2 é conexo. Contudo, nem sempre é trivial determinar se um grafo é conexo: imagine um emaranhado de um milhão de vértices, com dez milhões de arestas, e ter que descobrir se cada vértice é ou não alcançável a partir de todos os outros (e se isso parecer irrealisticamente grande, considere uma rede social como o Facebook, que tem mais de um bilhão de nós).
- **Grau**. O grau de um vértice é simplesmente o número de arestas que se ligam a ele (loops contam duas vezes). Na Figura 8.3, Porto de Galinhas tem grau 2, e Recife, 3. No caso de um grafo dirigido, distinguimos por vezes entre o número de arestas de entrada que um vértice tem (chamado o seu **grau de entrada**) e o número de arestas de saída (o seu **grau de saída**). Muhammad Ali teve um grau de saída (3) superior ao grau de entrada (1), uma vez que ganhou a maior parte do tempo (cf. Figura 8.2).
- **Ciclo**. Um ciclo é um caminho que começa e termina no mesmo vértice.[2] Na Figura 8.3, Maceió—Porto de Galinhas—Recife—Maceió é um ciclo. Qualquer laço é um ciclo por si só. Para os grafos dirigidos, todo o ciclo deve compreender arestas na direção "para a frente": não há possibilidade de andar para trás.

2 Diremos também que um ciclo não pode repetir quaisquer arestas ou vértices pelo caminho, de modo que não possa andar para trás e para a frente repetidamente e sem sentido entre dois nós adjacentes. Alguns matemáticos chamam isto de um *ciclo simples* para o distinguir do ciclo mais geral, mas diremos apenas que nenhum ciclo pode repetir-se desta maneira.

Na Figura 8.2, Frazier—Ali—Foreman—Frazier é um ciclo, tal como o ciclo mais simples Ali—Frazier—Ali.

- **DAG** (grafo acíclico direcionado, do inglês *directed acyclic graph*). Um uso comum dos grafos é representar fluxos de dependências, por exemplo, os pré-requisitos que as diferentes disciplinas universitárias têm umas com as outras. Outro exemplo são os fluxos de tarefas na gestão de projetos: as tarefas necessárias para completar um projeto tornam-se vértices, e então as dependências que elas têm umas com as outras tornam-se arestas. O grafo da Figura 5.4 mostra os passos para fazer um bolo, e como estes passos dependem uns dos outros. Os ovos têm de ser quebrados antes dos ingredientes poderem ser misturados, e o forno tem de ser preaquecido antes de assar, mas a forma pode ser untada em qualquer ocasião, desde que seja feita antes de se despejar a massa nela.

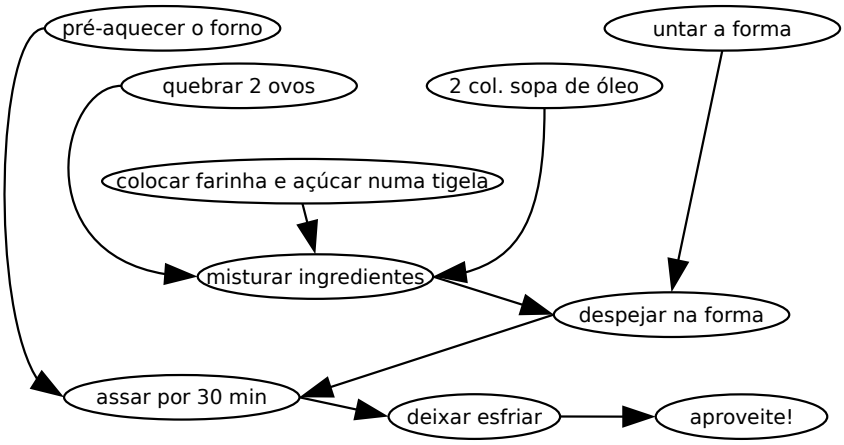

Figura 8.4: Um DAG.

Um grafo de dependências como este deve ser tanto **dirigido** como **acíclico**, ou não faria sentido. Dirigido, claro, significa que a tarefa X pode exigir que a tarefa Y seja concluída antes dela, sem que o inverso também seja verdadeiro. Se ambos dependessem um do outro, teríamos um loop infinito, e nenhum bolo poderia ser assado! Acíclico significa que *nenhum* tipo de ciclo pode existir no grafo, mesmo um que passe por múltiplos vértices. Tal ciclo resultaria novamente num loop infinito, tornando o projeto inútil. Imagine se houvesse uma seta de *assar por 30 minutos* de volta para *untar a forma* na Figura 8.4. Então, teríamos de untar a forma antes de despejar a massa nela, e teríamos que despejar a massa antes de assar, mas também teríamos de assar antes de untar a forma! Ficaríamos logo presos no trabalho: não haveria maneira de completar nenhuma dessas tarefas, uma vez que todas elas dependeriam indiretamente umas das outras. Um grafo que é tanto direcionado como acíclico (e portanto livre destes problemas) é por vezes chamado **DAG** para abreviar.

8.1.2 Posicionamento espacial

Uma coisa importante a compreender sobre os grafos é quais os aspectos de um diagrama que são relevantes. Especificamente, o *posicionamento espacial dos vértices não importa*. Na Figura 8.2 desenhamos Muhammad Ali do meio para a esquerda, e Sonny Liston no extremo superior direito. Mas esta foi uma escolha arbitrária, e irrelevante. Mais especificamente, isto não faz parte da informação que o diagrama pretende representar. Poderíamos ter posicionado os vértices de forma diferente, como na Figura 5.5, e obter o mesmo grafo. Em ambos os diagramas, há os mesmos vértices, e as mesmas arestas entre eles (verifique). Portanto, estes são matematicamente o mesmo grafo.

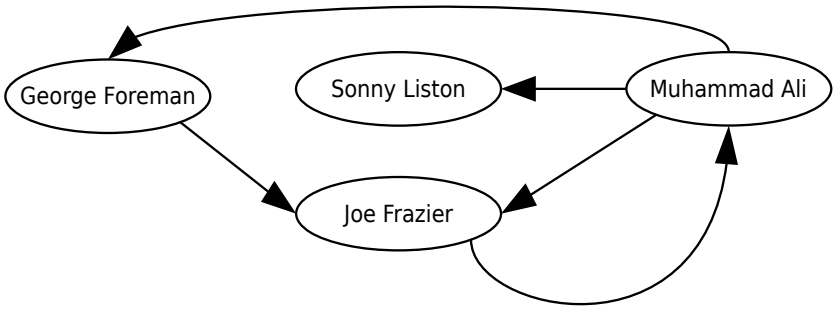

Figura 8.5: Uma visão diferente do mesmo grafo de boxeadores.

Isto pode não parecer surpreendente para o grafo dos boxeadores, mas para grafos como os de aplicações de GPS, em que nós na realidade representam localizações físicas, pode parecer chocante. Na Figura 8.3 poderíamos ter desenhado Recife ao norte de Maceió, e Porto de Galinhas num lugar qualquer do diagrama, e ainda ter o mesmo grafo, desde que todos os nós e ligações fossem os mesmos. Basta lembrar que o posicionamento espacial é concebido para conveniência humana, e não faz parte da informação matemática. É semelhante a como não há ordem para os elementos de um conjunto, embora quando especificamos uma extensão do conjunto, tenhamos de os listar em alguma ordem para evitar escrever todos os nomes dos elementos em cima uns dos outros. Num diagrama de um grafo, temos de desenhar cada vértice em algum lugar, mas onde os colocamos é simplesmente estético.

8.1.3 Relação com conjuntos

Parece que nos afastamos muito de conjuntos com todo este material sobre grafos. Mas na verdade, há algumas ligações importantes a serem feitas com esses conceitos iniciais. Recordemos o conjunto A de amigos do capítulo 2 que estendemos para conter {Helena, Ronaldo, Heitor, Nélson}. Consideremos agora a seguinte endorrelação em A:

(Helena, Ronaldo)
(Ronaldo, Helena)
(Ronaldo, Heitor)
(Ronaldo, Nélson)
(Heitor, Heitor)
(Nélson, Helena)

Esta relação, e tudo o que ela contém, é fielmente representada pelo grafo da Figura 8.6. Os elementos de A são os vértices, claro, e cada par ordenado da relação é refletido em uma aresta do grafo. Você consegue ver como exatamente a mesma informação é representada por ambas as formas?

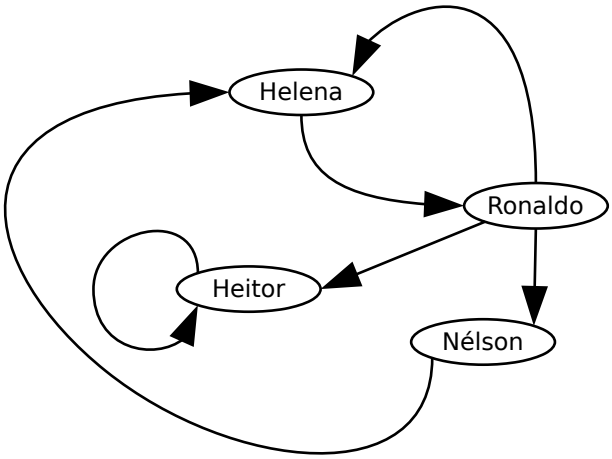

Figura 8.6: Um grafo representando uma endorrelação.

A Figura 8.6 é um grafo direcionado, é claro. E se fosse um grafo não direcionado? A resposta é que a relação correspondente seria *simétrica*. Um grafo não direcionado implica que se houver uma aresta entre dois vértices, ela "vai para os dois lados". Isto é de fato idêntico a dizer que uma relação é simétrica: se um (x, y) está na relação, então o correspondente (y, x) também deve estar. Um exemplo é a Figura 8.7, que retrata a seguinte relação simétrica:

(Helena, Ronaldo)
(Ronaldo, Helena)
(Ronaldo, Heitor)
(Heitor, Ronaldo)
(Helena, Helena)
(Nélson, Nélson)

Observe como os loops (arestas de um nó de volta para si próprio) nestes diagramas representam pares ordenados em que ambos os elementos são iguais.

Outra ligação entre grafos e conjuntos tem a ver com partições. A Figura 8.7 não era um grafo conexo: Nélson não podia ser alcançado a partir de nenhum dos outros nós. Agora considere: um grafo como este não é semelhante de alguma forma a uma partição de A — concretamente, esta?

{ Helena, Ronaldo, Heitor } e { Nélson }.

Simplesmente particionamos os elementos de A nos grupos que estão conectados. Se removermos a aresta entre Helena e Ronaldo nesse grafo, teremos:

{ Helena }, { Ronaldo, Heitor }, e { Nélson }.

Depois acrescente uma aresta entre Heitor e Nélson, e agora temos:

{ Helena } e { Ronaldo, Heitor, Nélson }.

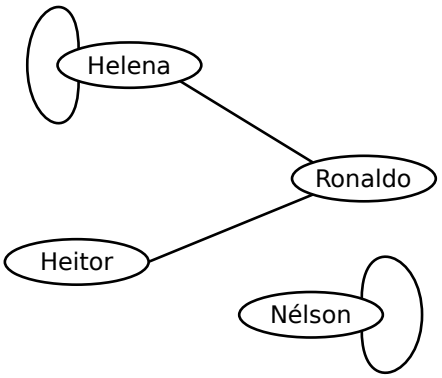

Figura 8.7: Um grafo representando uma endorrelação simétrica.

Em outras palavras, a "conectividade" de um grafo pode ser representada precisamente como uma partição do conjunto de vértices. Isto porquê cada subconjunto conectado, ou **componente conexo** do grafo, é um grupo, e cada vértice está num e apenas num grupo: por conseguinte, estes grupos isolados são mutuamente exclusivos e coletivamente exaustivos, ou seja, formam uma partição do conjunto de vértices. Legal.

8.1.4 Percursos em grafos

Se você tivesse uma longa lista — talvez de números de telefone, nomes, ou ordens de compra — e precisasse percorrê-la e fazer algo a cada elemento da lista — ligar para todos os números, procurar um determinado nome na lista, somar todas as ordens — seria bastante óbvio como fazê-lo. Basta começar pelo topo e trabalhar até o fim. Pode ser enfadonho, mas não é confuso.

Iterar pelos elementos desta forma chama-se **percorrer** a estrutura de dados. Você quer ter a certeza de encontrar cada elemento uma vez (e apenas uma vez) para poder fazer o que for preciso com ele. É evidente como se percorre uma lista. Mas como percorrer um grafo? Não há um "primeiro" ou "último" nó óbvio, e cada um deles está potencialmente ligado a muitos outros. E como vimos, os vértices podem nem sequer estar totalmente conectados, pelo que um caminho que percorre todos os nós pode nem sequer existir.

Há duas formas diferentes de percorrer um grafo: em largura, e em profundidade. Elas fornecem formas diferentes de explorar os nós, e como efeito secundário, cada uma é capaz de descobrir se o grafo é ou não conexo. Vamos examinar cada uma delas a seguir.

Travessia em Largura

Com a travessia em largura, começamos num vértice inicial (não importa qual) e exploramos o gráfico cautelosamente e delicadamente. Investigamos igualmente a fundo em todas as direções, certificando-nos de que olhamos um pouco para cada caminho possível antes de explorarmos cada um desses caminhos um pouco mais longe.

Para tal, usamos uma estrutura de dados muito simples chamada **fila**. Uma fila é simplesmente uma lista de nós que estão à espera na fila. Quando inserimos um nó na fi-

nal da fila, dizemos que “**enfileiramos o nó**”, e quando retiramos um nó da frente da fila, dizemos que “**desenfileiramos o nó**”. Os nós do meio esperam pacientemente chagar sua vez de serem tratados, aproximando-se da frente da fila cada vez que o nó da frente é desenfileirado.

Um exemplo desta estrutura de dados em ação é mostrado na Figura 5.8. Note cuidadosamente que inserimos sempre os nós numa extremidade (à direita) e os retiramos da outra extremidade (à esquerda). Isto significa que o primeiro item a ser enfileirado (neste caso, o triângulo) será o primeiro a ser desenfileirado. “As chamadas serão respondidas na ordem em que foram recebidas”. Este fato deu origem a outro nome para uma fila: uma “**FIFO**”, que significa “primeiro a entrar e primeiro a sair, do inglês *first-in-first-out*”..

Comece com uma fila vazia:	|
Enfileire um triângulo, então temos:	|△
Enfileire uma estrela, então temos:	|△✶
Enfileire um coração, então temos:	|△✶♡
Desenfileire (o triângulo), então temos:	|✶♡
Enfileire um trevo, então temos:	|✶♡♣
Desenfileire (a estrela), então temos:	|♡♣
Desenfileire (o coração), então temos:	|♣
Desenfileire (o trevo). Estamos vazios novamente:	|

Figura 8.8: Uma fila em ação. A barra vertical marca a "frente da fila", e os elementos estão à espera de serem desenfileirados, em ordem, da esquerda para a direita.

Agora, eis como utilizamos uma fila para percorrer um grafo em largura. Vamos começar por um nó em particular, e colocar todos os seus nós adjacentes numa fila. Isto faz com que todos eles fiquem “à espera na fila” em segurança até os explorarmos. Depois, repetidamente pegamos o primeiro nó da fila, fazemos o que precisamos fazer com ele, e depois colocamos todos os seus nós adjacentes na fila. Continuamos a fazer isto até que a fila esteja vazia.

Agora pode ter-lhe ocorrido que podemos ter problemas se encontrarmos o mesmo nó várias vezes enquanto estamos a percorrer. Isto pode acontecer se o grafo tiver um ciclo: haverá mais de um caminho para chegar a alguns nós, e podemos ficar presos num loop infinito se não tivermos cuidado. Por esta razão, introduzimos o conceito de **marcação de nós**. Isto é como deixar um rastro de migalhas de pão: se alguma vez estivermos prestes a explorar um nó, mas descobrirmos que ele está marcado, então sabemos que já lá estivemos, e é inútil pesquisá-lo novamente.

Portanto, há duas coisas que vamos fazer aos nós enquanto buscamos:

- **Marcar** um nó significa recordar que já o encontramos no processo da nossa pesquisa.

- **Visitar** um nó significa realmente fazer o que quer que seja que precisamos fazer com o nó (ligar para o número de telefone, examinar o seu nome para um casamento de padrão, adicionar o número ao nosso total, o que quer que seja).

A travessia em largura (**BFT** — do inglês *breadth-first traversal*) é um algoritmo, o que significa simplesmente um procedimento passo-a-passo, fiável, que garante a produção de um resultado. Neste caso, é garantido que se visite todos os nós do grafo que são alcançáveis a partir do nó inicial, e que não se fique preso em quaisquer loops infinitos no processo. Aqui está ele:

Travessia em Largura (BFT)

1. Escolher um nó de partida.
2. Marque-o e enfileire-o numa fila até então vazia.
3. Enquanto a fila não estiver vazia, faça estes passos:
   a. Desenfileire o nó do início da fila.
   b. Visite-o.
   c. Marque e enfileire todos os *nós não marcados adjacentes a ele* (em qualquer ordem)

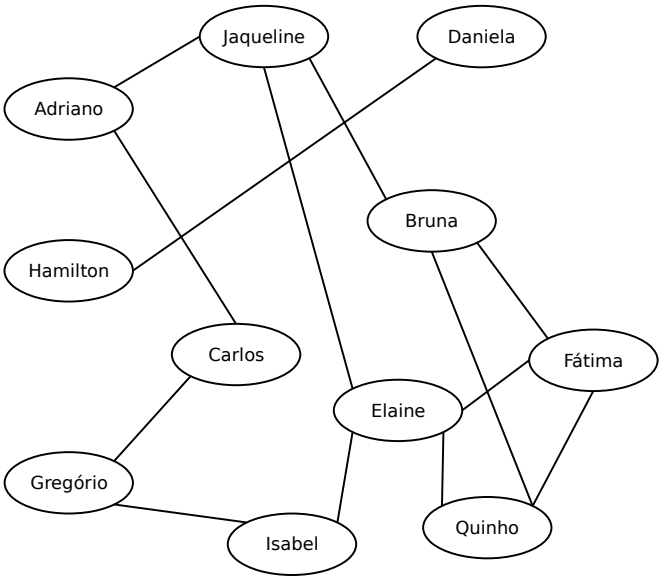

Figura 8.9: Grupo de pessoas numa rede social e suas relações de amizade.

Vamos executar este algoritmo e vê-lo em ação num conjunto de utilizadores de uma rede social. A Figura 8.9 retrata onze utilizadores, e as amizades entre eles. Agora acompanhemos a execução do algoritmo BFT neste grafo na Figura 8.10. Primeiro, escolhemos Gregório como o nó de partida (não por nenhuma razão em particular, apenas porque temos de começar em algum lugar). Marcamo-lo (cinza claro no diagrama) e colocamo-lo na fila (o conteúdo da fila é listado na parte inferior de cada quadro, com a frente da fila à esquerda). Depois, começamos o nosso laço. Quando ti-

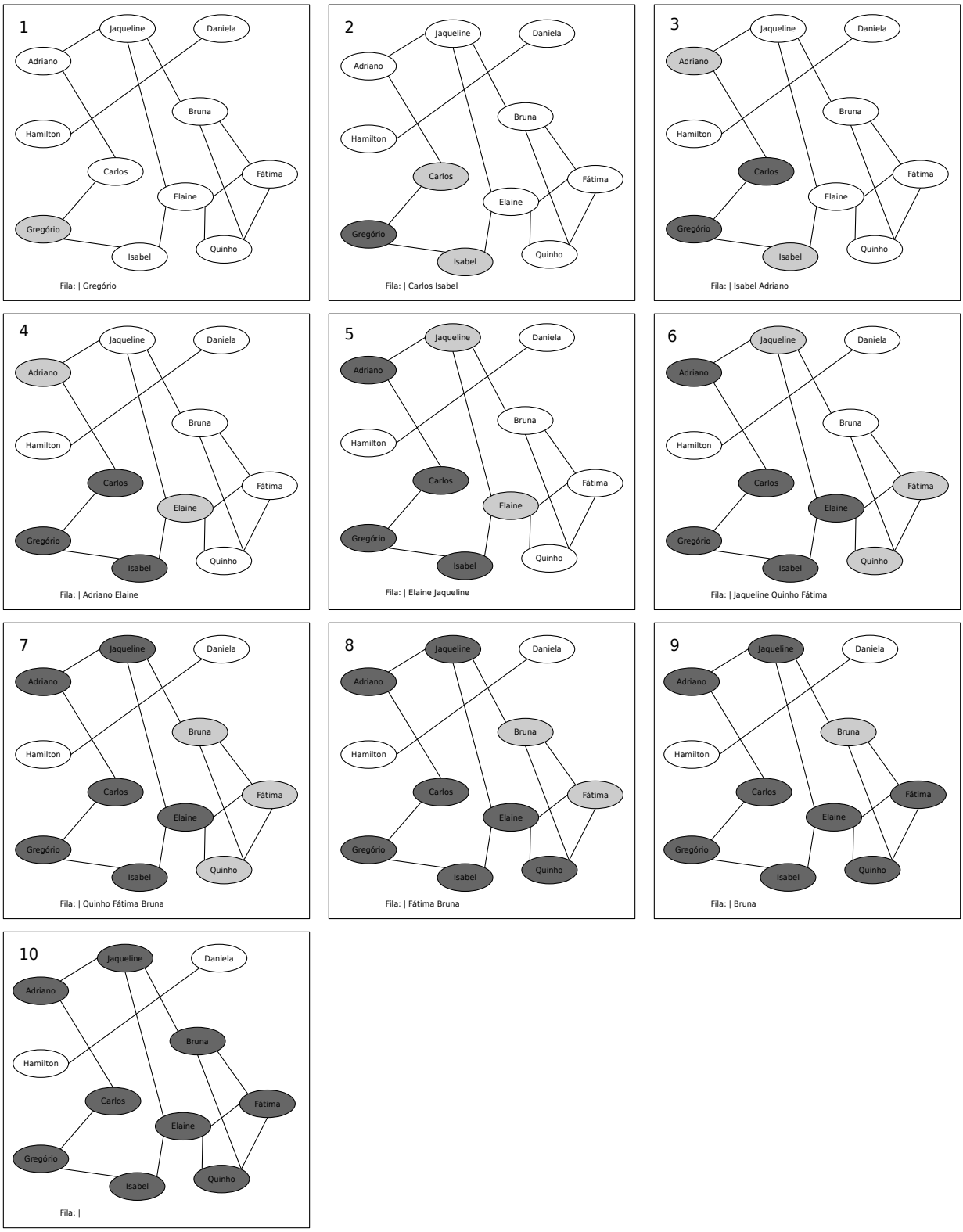

Figura 8.10: As etapas da travessia em largura. Os nós marcados são cinza claro, e os nós visitados são cinza escuro. A ordem de visita é: G, C, I, A, E, J, Q, F, B.

ramos o Gregório da fila, visitamo-lo (o que significa que "fazemos o que for preciso fazer ao Gregório") e depois marcamos e enfileiramos os seus nós adjacentes Carlos e

Isabel. Não importa em que ordem os colocamos na fila, tal como não importava com que nó começamos. No quadro 3, Carlos foi desenfileirado, visitado, e os seus nós adjacentes colocados na fila. Apenas um nó é enfileirado aqui — Adriano — porque obviamente Gregório já foi marcado (e mesmo visitado, não menos) e esta marcação permite-nos ser inteligentes e não o reenfileiramos.

É neste ponto que a característica "em largura" se torna aparente. Acabamos de terminar com Carlos, mas em vez de explorarmos Adriano a seguir, retomamos com Isabel. Isto porque ela tem estado pacientemente à espera na fila, e chegou a sua vez. Assim, colocamos Adriano de lado (na fila, claro) e visitamos Isabel, enfileirando a sua vizinha Elaine no processo. Depois, voltamos a Adriano. O processo segue, seguindo a ordem em que os elementos tornam-se presentes na fila. A visita a Jaqueline leva-nos a enfileirar Bruna, e depois, quando tiramos Quinho da fila, não voltamos a enfileirar Bruna porque ela foi marcada e por isso sabemos que ela já tinha sido tratada.

Continuamos a visitar os nós da fila até a fila ficar vazia. Como podem ver, Hamilton e Daniela não serão visitados neste processo: isto porque aparentemente ninguém que eles conhecem conhece ninguém da turma de Gregório, e por isso não há maneira de os alcançar a partir de Gregório. Foi isto que quis dizer anteriormente ao falar que, como efeito secundário, o algoritmo BFT diz-nos se o gráfico é ou não conexo. Tudo o que temos de fazer é começar em algum lugar, rodar o BFT, e depois ver se algum nó não foi marcado e visitado. Se houver algum, e quisermos visitar todos os nós, podemos continuar com outro ponto de partida, e depois repetir o processo.

Travessia em Profundidade

Com a **travessia em profundidade**, ou **DFT** (do inglês *depth-first traversal*), exploramos o gráfico de forma arrojada e imprudente. Escolhemos a primeira direção que vemos, e mergulhamos até as suas profundezas, antes de recuarmos relutantemente e tentarmos os outros caminhos a partir do início.

O algoritmo é quase idêntico ao BFT, exceto que, em vez de uma fila, utilizamos uma **pilha**. Uma pilha é semelhante a uma fila, exceto que em vez de colocar elementos numa extremidade e retirá-los da outra, adiciona-se e retira-se da mesma extremidade. Esta "extremidade" é chamada de **topo da pilha**. Quando adicionamos um elemento a este extremo, dizemos que o empurramos na pilha, e quando retiramos o elemento superior, dizemos que o tiramos do topo da pilha.

Pode-se pensar numa pilha como… bem, uma pilha, seja de livros ou de bandejas de cafeteria ou qualquer outra coisa. Não se pode tirar nada do meio de uma pilha, mas pode-se retirar itens e colocar mais itens, sempre do topo. A Figura 5.10 mostra um exemplo. O primeiro item que foi empurrado é sempre o último a ser retirado, e o mais recente está sempre pronto a ser tirado, e por isso uma pilha é por vezes também chamada "**LIFO**" (do inglês, *last-in-first-out*).

O próprio algoritmo de travessia em profundidade parece um déjà vu. Tudo o que se faz é substituir "fila" por "pilha":

Comece com uma pilha vazia:	__
Empurre um triângulo, então temos:	△ __
Empurre uma estrela, então temos:	✶ △ __
Empurre um coração, então temos:	♡ ✶ △ __
Retire (o coração), então temos:	✶ △ __
Empurre um trevo, então temos:	♣ ✶ △ __
Retire (o trevo), então temos:	✶ △ __
Retire (a estrela), então temos:	△ __
Retire (o triângulo). Estamos vazios novamente:	__

Figura 8.11: Uma pilha em ação. A barra horizontal marca o "fundo da pilha", e os elementos são colocados e tirados do topo.

Travessia em Profundidade (DFT)

1. Escolher um nó de partida.
2. Marque-o e empurre-o numa pilha até então vazia.
3. Enquanto a pilha não estiver vazia, faça estes passos:
   d. Retire o nó do topo da pilha.
   e. Visite-o.
   f. Marque e empurre na pilha todos os *nós não marcados adjacentes a ele* (em qualquer ordem)

O algoritmo em ação é mostrado na Figura 8.12. A pilha fez realmente a diferença! Em vez de explorar alternadamente os caminhos de Carlos e de Isabel, ele mergulha de cabeça pelo caminho de Carlos até onde pode ir, até atingir a porta dos fundos de Isabel. Só depois volta atrás e visita Isabel. Isto porque a pilha sempre tira o que acabou de empurrar, enquanto que o que foi empurrado primeiro tem de esperar até que todo o resto seja feito antes de ter a sua oportunidade. Esse primeiro par de empurrões foi crítico: se tivéssemos empurrado Carlos antes de Isabel logo no início, então teríamos explorado todo o mundo de Isabel antes de chegarmos à porta dos fundos

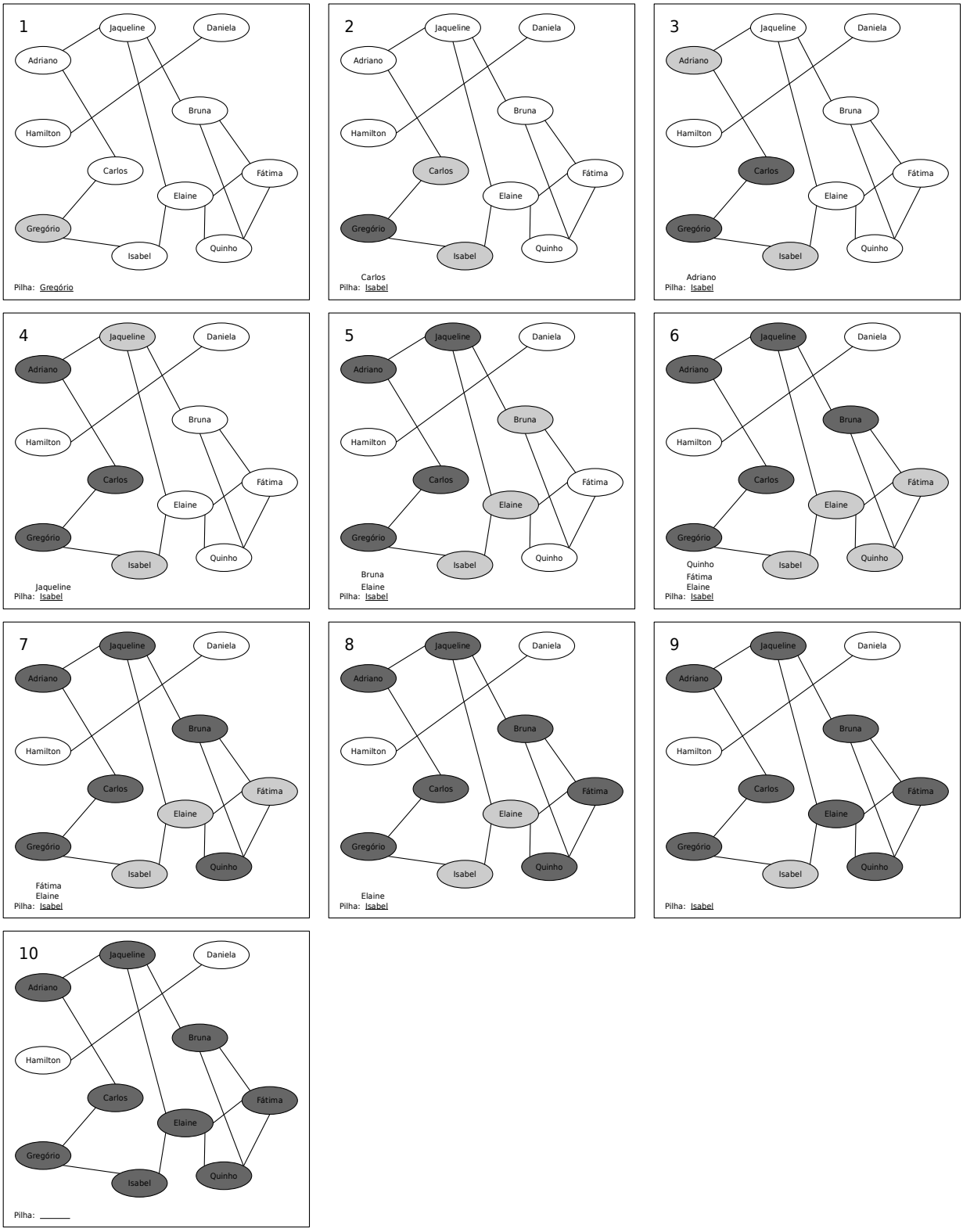

Figura 8.12: As etapas da travessia em profundidade. Os nós marcados são cinza claro, e os nós visitados são cinza escuro. A ordem de visita é: G, C, A, J, B, Q, F, E, I.

de Carlos, em vez do contrário. Tal como está, Isabel foi colocada no fundo, e assim ficou no fundo, o que é inevitável com uma pilha.

O DFT identifica gráficos desconexos da mesma forma que o BFT, e evita igualmente ficar preso em loops infinitos quando encontra ciclos. A diferença fundamental é a ordem em que visita os nós.

8.2. Árvores

Uma árvore não é nada mais do que uma simplificação de um grafo. Existem dois tipos de árvores no mundo: **árvores não-enraizadas** (também chamadas de **árvores livres**), e **árvores enraizadas**.[3]

8.2.1 Árvores Livres

Uma árvore livre é apenas um grafo conexo, sem ciclos. Cada nó é alcançável a partir dos outros, e só há uma maneira de chegar a qualquer ponto. Veja a Figura 8.13. Parece apenas um grafo (e é), mas ao contrário do grafo da Figura 8.9, é mais "esquelético". Isto porque, em certo sentido, uma árvore livre não contém nada "extra".

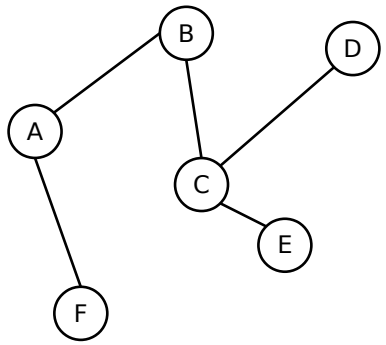

Figura 8.13: Uma árvore livre.

Se você tiver uma árvore livre, os seguintes fatos interessantes são verdadeiros:

1. Há exatamente um caminho entre quaisquer dois nós. (Verifique!)
2. Se você remover qualquer aresta, o gráfico torna-se desconexo. (Experimente!)
3. Se você adicionar qualquer nova aresta, você acaba de acrescentar um ciclo. (Experimente!)
4. Se houver n nós, há $n - 1$ arestas. (Pense sobre isso! Você conseguiria provar esse fato usando indução?)

Assim, basicamente, se o seu objetivo for conectar todos os nós, e você tiver uma árvore livre, está tudo pronto. Acrescentar qualquer coisa é redundante, e tirar qualquer coisa estraga tudo.

3 Parece não haver consenso quanto a qual destes conceitos é o mais básico. Alguns autores referem-se a uma árvore livre simplesmente como uma "árvore" — como se este fosse o tipo de árvore "normal" — e utilizam o termo árvore enraizada para o outro tipo. Outros autores fazem o oposto. Para evitar confusões, vou tentar usar sempre o termo completo (embora admita que sou daqueles que consideram as árvores enraizadas como sendo o conceito mais importante, padrão).

Tenha em mente que existem muitas árvores livres que podem ser feitas com o mesmo conjunto de vértices. Por exemplo, se você remover a aresta de A para F, e adicionar uma de qualquer outro ponto para F, você tem uma árvore livre diferente.

8.2.2 Árvores enraizadas

Agora uma árvore enraizada é a mesma coisa que uma árvore livre, exceto que nós elevamos um nó para se tornar a raiz. Acontece que isto faz toda a diferença. Suponha que escolhemos A como a raiz da Figura 8.13. Então teríamos a árvore enraizada no lado esquerdo da Figura 8.14. O vértice A foi posicionado no topo, e todo o resto passa para baixo dele. Penso nisso como se estivéssemos chegando na árvore livre, pegando cuidadosamente um nó, e depois levantando a mão para que o resto da árvore livre se pendurasse dali. Se tivéssemos escolhido (digamos) C como raiz, teríamos uma árvore enraizada diferente, representada na metade direita da figura. Ambas estas árvores enraizadas têm todas as mesmas arestas que a árvore livre tinha: B está ligado tanto a A quanto a C, F está ligado apenas a A, etc. A única diferença é qual o nó que é designado como a raiz.

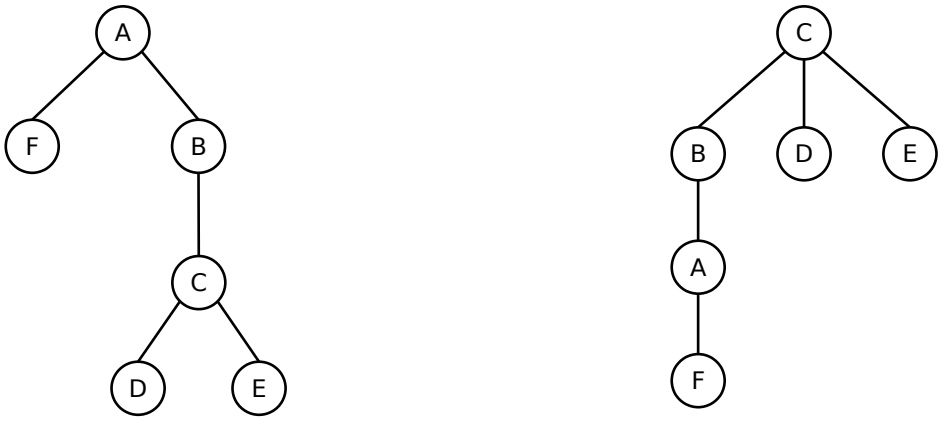

Figura 8.14: Duas árvores enraizadas diferentes com os mesmos vértices e arestas.

Até agora, temos dito que o posicionamento espacial nos grafos é irrelevante. Mas isto muda um pouco com as árvores enraizadas. O posicionamento vertical é a nossa única forma de mostrar que nós estão “acima” dos outros, e a palavra “acima” tem de fato um significado aqui: significa mais perto da raiz. A **altura** de um nó nos diz quantos passos ele está afastado da raiz. Na árvore enraizada à direita, os nós B, D, e E estão todos a um passo da raiz (C), portanto têm altura igual a um, enquanto o nó F está a três passos de distância da raiz, ou seja, sua altura é três. Entretanto, como veremos a diante, é mais comum utilizar o termo **profundidade do nó** (uma vez que nossa árvore é “de cabeça para baixo”), e reservamos o termo altura para se referir a uma característica da árvore como um todo.

O aspecto chave das árvores enraizadas — que é tanto a sua maior vantagem como a sua maior limitação — é que cada nó tem um e apenas um caminho para a raiz. Este comportamento é herdado de árvores livres: como notamos, cada nó tem apenas um caminho para cada outro.

As árvores têm uma miríade de aplicações. Pense nos arquivos e pastas no seu disco rígido: na parte superior está a raiz do sistema de arquivos (talvez "`/`" no Linux/Mac ou "`C:\`" no Windows) e abaixo disso encontram-se as chamadas pastas. Cada pasta pode conter arquivos assim como outras pastas com nomes, e assim por diante na hierarquia. O resultado é que cada arquivo tem um, e apenas um, caminho distinto até ele a partir do topo do sistema de arquivos. O arquivo pode ser armazenado, e mais tarde recuperado, exatamente de uma maneira.

Um "organograma" é assim: o CEO está no topo, depois abaixo dele estão os VP, os Diretores, os Gerentes, e finalmente os funcionários. Tal como a organização militar numa democracia presidencialista[4]: o Presidente é o comandante em chefe e comanda o Ministro da Defesa, que dirige generais, que comandam coronéis, que comandam majores, que comandam capitães, que comandam tenentes, que comandam sargentos, que comandam soldados.

O corpo humano é mesmo uma espécie de árvore enraizada: ele contém esqueleto, sistemas cardiovascular, digestório, e outros, cada um dos quais é composto por órgãos, depois tecidos, depois células, moléculas, átomos, e partículas subatômicas. De fato, qualquer coisa que tenha este tipo de hierarquia de contenção parte-todo está justamente pedindo para ser representada como uma árvore.

Na programação de computadores, as aplicações são muito numerosas para nomear. Os compiladores varrem o código-fonte e constroem uma "árvore de análise" do seu significado subjacente. O HTML é uma forma de estruturar texto simples numa hierarquia de elementos exibíveis em forma de árvore. Os programas de xadrez de Inteligência Artificial constroem árvores representando os seus possíveis movimentos futuros e as respostas prováveis do seu adversário, a fim de "ver muitos movimentos à frente" e avaliar as suas melhores opções. Os projetos orientados a objetos envolvem "hierarquias de herança" de classes, cada uma especializada a partir de uma outra específica. Etc. À exceção de uma simples sequência (como um vetor), as árvores são provavelmente a estrutura de dados mais comum em toda a computação.

Terminologia de árvores enraizadas

As árvores enraizadas trazem consigo uma série de termos. Vou usar a árvore do lado esquerdo da Figura 8.14 como ilustração de cada um:

- **Raiz**. O nó no topo da árvore, que é *A* no nosso exemplo. Note que, ao contrário das árvores no mundo real, as árvores da informática têm a sua raiz no topo e crescem para baixo. Cada árvore tem uma raiz, exceto a árvore vazia, que é a "árvore" que não tem nenhum nó nela. (É um pouco estranho pensar no "nada" como uma árvore, mas é como o conjunto vazio $\varnothing$, que ainda é um conjunto).

4 Pelo menos na teoria é assim. O presidente representa o poder popular, pois é o maior líder eleito por toda a população, e por isso é o comandante em chefe. No entanto, em democracias frágeis, por vezes as forças militares se aliam a poderes não eleitos, como a Mídia e o Sistema Judiciário para organizarem golpes de estado de acordo com os seus interesses. Isto geralmente acontece pois ao possuir o poder de fato da força (armas, controle sobre as liberdades individuais etc.), essas forças terminam considerando que têm o direito de ser uma casta privilegiada com poder superior ao poder democrático popular expresso pelo voto.

- **Pai**. Cada nó, exceto a raiz, tem um pai: o nó imediatamente acima dele. O pai de D é C, o pai de C é B, o pai de F é A, e A não tem pai.
- **Filho**. Alguns nós têm filhos, que são nós conectados diretamente abaixo dele. Os filhos de A são F e B, os de C são D e E, o único filho de B é C, e E não tem filhos.
- **Irmão**. Um nó que tem o mesmo pai. O irmão de E é D, o de B é F, e nenhum dos outros nós têm irmãos.
- **Ancestral**. Os seus pais, avós, bisavós, etc., até a raiz. O único ancestral de B é A, enquanto os ancestrais de E são C, B, e A. Note que F não é ancestral de C, apesar de estar acima dele no diagrama: não há ligação de C a F, exceto através da raiz (que não conta).
- **Folha**. Um nó sem filhos. F, D, e E são folhas. Note que numa árvore (muito) pequena, a própria raiz poderia ser uma folha.
- **Descendente**. Os seus filhos, netos, bisnetos, etc., até as folhas. Os descendentes de B são C, D e E, enquanto os de A são F, B, C, D, e E.
- **Nó interno**. Qualquer nó que não seja uma folha. A, B, e C são os nós internos no nosso exemplo.
- **Profundidade** (de um nó). A profundidade de um nó é a distância (em número de nós) a partir dele até a raiz. A própria raiz tem profundidade zero. No nosso exemplo, B tem profundidade 1, E tem profundidade 3, e A tem profundidade 0.
- **Altura** (de uma árvore). A altura de uma árvore enraizada é a profundidade máxima de qualquer dos seus nós; ou seja, a distância máxima da raiz a qualquer nó. O nosso exemplo tem altura 3, uma vez que os nós "mais profundos" são D e E, cada um com uma profundidade 3. Uma árvore com apenas um nó é considerada como tendo altura 0. Bizarro, mas para ser consistente, diremos que a árvore vazia tem altura -1! Estranho, mas que mais poderia ser? Dizer que tem altura 0 parece inconsistente com uma árvore de um nó também ter altura 0. Em todo o caso, isto não surgirá frequentemente.
- **Nível**. Todos os nós com a mesma profundidade são considerados no mesmo "nível". B e F estão no nível 1, e D e E estão no nível 3. Os nós no mesmo nível não são necessariamente irmãos. Se F tivesse uma criança chamada G no diagrama de exemplo, então G e C estariam no mesmo nível (2), mas não seriam irmãos porque têm pais diferentes. (Poderíamos chamar-lhes "primos" para continuar com a analogia familiar).
- **Subárvore**. Finalmente, muito do que dá às árvores o seu poder expressivo é a sua natureza **recursiva**. Isto significa que uma árvore é constituída por *outras árvores (menores)*. Consideremos o nosso exemplo. Trata-se de uma árvore com raiz A. Mas os dois filhos de A são, cada um por si, árvores de direito próprio! F em si é uma árvore com apenas um nó. B e os seus descendentes formam outra árvore com quatro nós. Consideramos que estas duas árvores são subárvores da árvore original. A noção de "raiz" muda um pouco à medida que consideramos subárvores—A é a raiz da árvore original, mas B é a raiz da se-

gunda subárvore. Quando consideramos os filhos de *B*, vemos que há ainda outra subárvore, que está enraizada em *C*. E assim por diante. É fácil ver que qualquer subárvore cumpre todas as propriedades das árvores e, por isso, tudo o que dissemos acima aplica-se também a ela.

Árvores binárias (BT's)

Os nós de uma árvore enraizada podem ter qualquer número de filhos. Há um tipo especial de árvore enraizada, no entanto, chamada **árvore binária**, o qual restringimos dizendo simplesmente que cada nó pode ter, no máximo, dois filhos. Além disso, vamos rotular cada um destes dois filhos como o "filho esquerdo" e o "filho direito". (Note que um determinado nó pode muito bem ter apenas um filho esquerdo, ou apenas um filho direito, mas, ainda assim, é importante saber em que direção está esse filho).

O lado esquerdo da Figura 8.14 é uma árvore binária, mas o lado direito não é (C tem três filhos). Uma árvore binária maior (de altura 4) é mostrada na Figura 8.15.

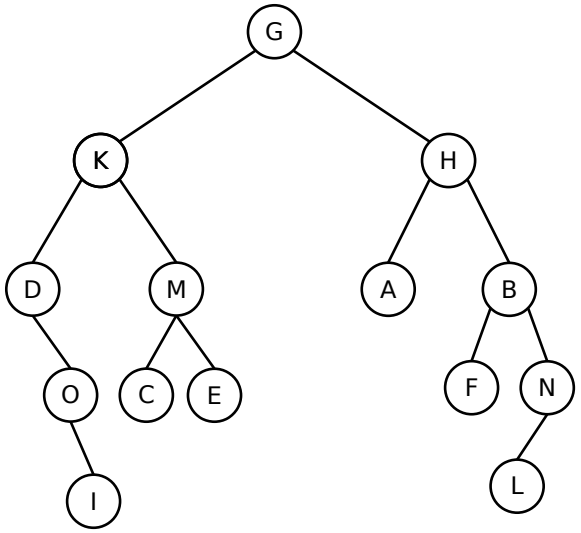

Figura 8.15: Uma árvore binária.

Percorrendo Árvores Binárias

Havia duas formas de percorrer um grafo: em largura, e em profundidade. Curiosamente, existem três formas de percorrer uma árvore: **pré-ordem**, **pós-ordem**, e **em ordem**. Todas as três começam pela raiz, e todas três consideram cada um dos filhos da raiz como subárvores. A diferença está na ordem de visitação.

Para percorrer uma árvore em *pré-ordem*, nós:

1. Visitamos a raiz.
2. Tratamos o filho esquerdo bem como todos os seus descendentes como uma subárvore, e percorremo-la (em pré-ordem) completamente.
3. Fazemos o mesmo com o filho direito.

É traiçoeiro porque você tem que se lembrar que cada vez que você "trata um filho como uma subárvore", você faz todo o processo de percorrer essa subárvore. Isto implica lembrar onde estava quando você terminar.

Siga este exemplo cuidadosamente. Para a árvore da Figura 8.15, começamos visitando *G*. Depois, atravessamos toda a "subárvore *K*". Isto envolve visitar o próprio *K*, e depois atravessar toda a sua subárvore esquerda (enraizada em *D*). Depois de visitarmos o nó *D*, descobrimos que na realidade não há nenhuma subárvore à esquerda dele, por isso avançamos e atravessamos toda a sua subárvore direita. Isto visita *O* seguido de *I* (já que *O* também não tem subárvore esquerda) e finalmente voltamos para cima.

É neste ponto que é fácil se perder. Acabamos de visitar *I*, e depois temos de perguntar "está bem, onde raio nós estávamos? Como é que chegamos aqui"? A resposta é que tínhamos acabado de estar no nó *K*, onde tínhamos atravessado a sua subárvore à esquerda (*D*). Então agora o que é que é tempo de fazer? Atravessar a subárvore direita, claro, que é *M*. Isto envolve visitar *M*, *C*, e *E* (nessa ordem) antes de voltar ao topo, *G*.

Agora estamos no mesmo tipo de situação em que podíamos ter nos perdido antes: passamos muito tempo na confusão da subárvore esquerda de *G*, e só temos de nos lembrar que agora é hora de fazer a subárvore direita de *G*. Siga este mesmo procedimento, e toda a ordem de visitação acaba por ser: *G*, *K*, *D*, *O*, *I*, *M*, *C*, *E*, *H*, *A*, *B*, *F*, *N*, *L*. (Ver Figura 8.16 para uma visualização).

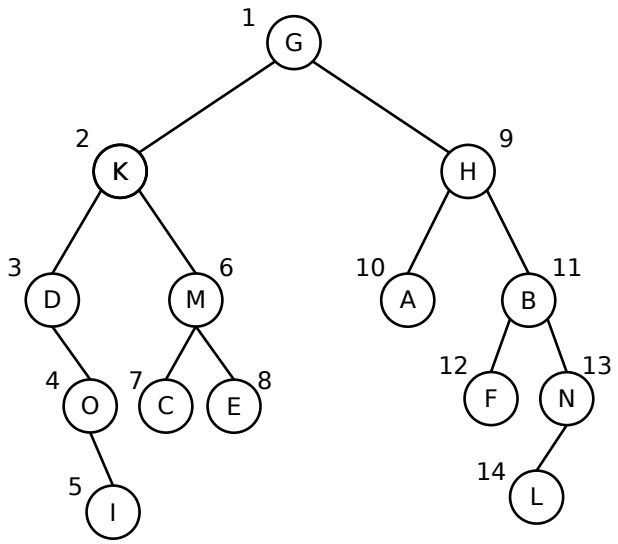

*Figura 8.16: A ordem de visitação dos nós num percurso em **pré-ordem**.*

Para percorrer uma árvore em *pós-ordem*, nós:

1. Tratamos o filho esquerdo bem como todos os seus descendentes como uma subárvore, e percorremo-la (em pós-ordem) completamente.
2. Fazemos o mesmo com o filho direito.
3. Visitamos a raiz.

É o mesmo que em pré-ordem, exceto que visitamos a raiz depois dos filhos, em vez de antes. Ainda assim, apesar da sua semelhança, esta foi sempre a mais complicada

para mim. Tudo parece adiado, e você tem que lembrar em que ordem fazê-lo mais tarde.

Para a nossa árvore de exemplo, o primeiro nó visitado acaba por ser *I*. Isto é porque temos de adiar a visita a *G* até terminarmos a sua subárvore esquerda (e direita); depois adiamos *K* até terminarmos a sua subárvore esquerda (e direita); adiamos *D* até terminarmos a subárvore de *O*, e adiamos *O* até fazermos *I*. Depois, finalmente, a coisa começa a desenrolar-se… até *K*. Mas ainda não podemos visitar o próprio *K*, porque temos de fazer a sua subárvore direita. Isto resulta em *C*, *E*, e *M*, nessa ordem. Então podemos fazer *K*, mas ainda não podemos fazer *G* porque temos todo o seu mundo de subárvore direita para enfrentar. A ordem inteira acaba sendo: *I*, *O*, *D*, *C*, *E*, *M*, *K*, *A*, *F*, *L*, *N*, *B*, *H*, e finalmente *G*. (Ver Figura 8.17 para uma visualização)

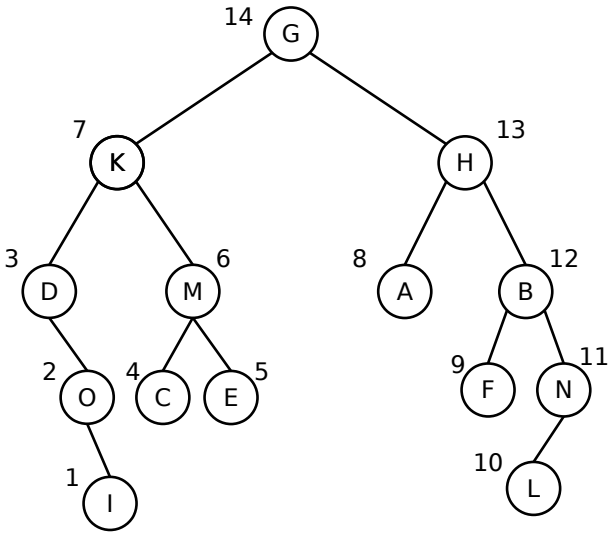

*Figura 8.17: A ordem de visitação dos nós num percurso em **pós-ordem**.*

Note que isto não é nem de longe o inverso da visita em pré-ordem, como você poderia esperar. *G* é o último em vez do primeiro, mas o resto fica todo embaralhado.

Finalmente, para percorrer uma árvore *em ordem*, nós:

1. Tratamos o filho esquerdo bem como todos os seus descendentes como uma subárvore, e percorremo-la (em ordem) completamente.
2. Visitamos a raiz.
3. Tratamos o filho direito bem como todos os seus descendentes como uma subárvore, e percorremo-la (em ordem) completamente.

Assim, em vez de visitarmos a raiz primeiro (pré-ordem) ou por último (pós-ordem), tratamo-la entre os nossos filhos da esquerda e da direita. Isto pode parecer algo estranho para se fazer, mas há um método nesta loucura que se tornará claro na próxima seção.

Para a árvore do exemplo, o primeiro nó visitado é *D*. Isto é porque é o primeiro nó encontrado que não tem uma subárvore esquerda, o que significa que o passo 1 não precisa fazer nada. Isto é seguido por *O* e *I*, pela mesma razão. Visitamos então *K* an-

tes da sua subárvore direita, que por sua vez visita *C*, *M*, e *E*, nessa ordem. A ordem final é: *D*, *O*, *I*, *K*, *C*, *M*, *E*, *G*, *A*, *H*, *F*, *B*, *L*, *N*. (Ver Figura 5.20.)

Se os seus nós estiverem espaçados uniformemente, você pode ler a travessia em ordem do diagrama movendo os olhos da esquerda para a direita. Tenha cuidado com isto, no entanto, porque em última análise a posição espacial não importa, mas sim as relações entre nós. Por exemplo, se eu tivesse desenhado o nó *I* mais para a direita, a fim de tornar as linhas entre *D*–*O*–*I* menos íngremes, que o nó *I* poderia ter sido empurrado espacialmente para a direita de *K*. Mas isso *não* mudaria a ordem, não fazendo com que *K* fosse visitado mais cedo.

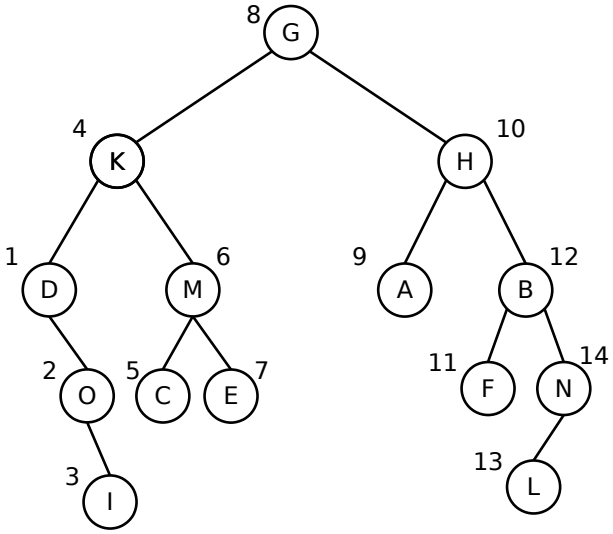

*Figura 8.18: A ordem de visitação dos nós num percurso **em ordem**.*

Por fim, vale a pena mencionar que todos estes métodos de travessia fazem um elegante uso da recursividade. A recursividade é uma forma de pegar um grande problema e dividi-lo em subproblemas semelhantes, mas menores. Então, cada um desses subproblemas pode ser atacado da mesma forma que se atacou o problema maior: dividindo-o em subproblemas. Tudo o que você precisa é de uma regra para, eventualmente, parar o processo de "quebrar", fazendo realmente alguma coisa.

Sempre que um destes processos de travessia trata um filho esquerdo ou direito como uma subárvore, ele está "recursivamente" reiniciando todo o processo de travessia numa árvore menor. A travessia em pré-ordem, por exemplo, depois de visitar a raiz, diz, "está bem, vamos fingir que começamos toda esta coisa de travessia com a árvore menor enraizada no meu filho esquerdo. Uma vez isso terminado, acorde-me para que eu possa iniciar de forma semelhante com o meu filho direito". A recursividade é uma forma muito comum e útil de resolver certos problemas complexos, e as árvores estão repletas de oportunidades.

Tamanhos de árvores binárias

As árvores binárias podem ter qualquer velho formato, como a nossa Figura 8.15, por exemplo. Por vezes, porém, queremos falar de árvores binárias com uma forma mais regular, que satisfaçam determinadas condições. Em particular, falaremos de três tipos especiais:

- **Árvore binária estrita**. Uma árvore binária estrita é aquela em que todo o nó (exceto as folhas) tem dois filhos. Dito de outra forma, cada nó tem ou dois filhos ou nenhum: nenhuma flexibilidade é permitida. A Figura 8.15 não é estrita, mas seria se acrescentássemos três nós em branco, como na Figura 8.19.

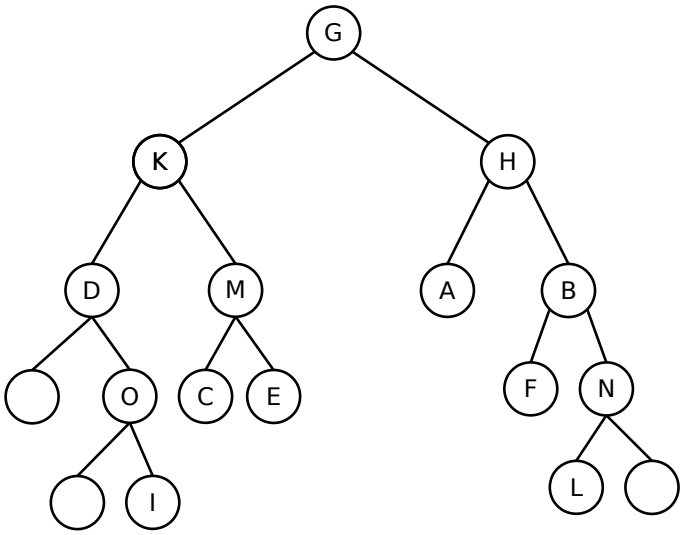

*Figura 8.19: Uma árvore binária **estrita**.*

A propósito, nem sempre é possível ter uma árvore binária estrita com um determinado número de nós. Por exemplo, uma árvore binária com dois nós não pode ser estrita, uma vez que inevitavelmente terá uma raiz com apenas um filho.

- **Árvore binária completa**. Uma árvore binária completa é aquela em que todos os níveis têm todos os nós possíveis presentes, à exceção talvez do nível mais profundo, que é preenchido a partir da esquerda. A Figura 8.19 não está completa, mas estaria se a consertássemos como na Figura 8.20.

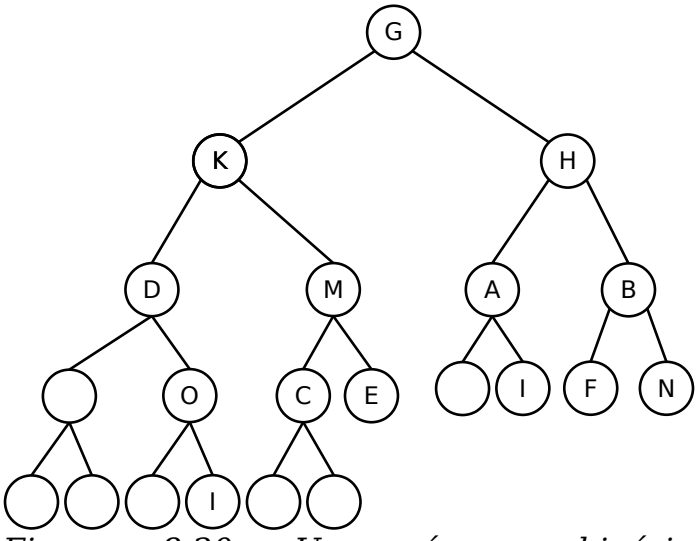

*Figura 8.20: Uma árvore binária **completa**.*

Ao contrário das árvores binárias estritas, é sempre possível ter uma árvore binária completa, para qualquer número de nós. Basta continuar a preencher da esquerda para a direita, nível após nível.

- **Árvore binária perfeita**. O nosso último tipo especial tem um título bastante audacioso, mas uma árvore "perfeita" é simplesmente uma que é exatamente balanceada: cada nível é completamente preenchido. A Figura 8.20 não é perfeita, mas seria se adicionássemos nós para preencher o nível 4, ou eliminássemos este nível inacabado (como na Figura 8.21).

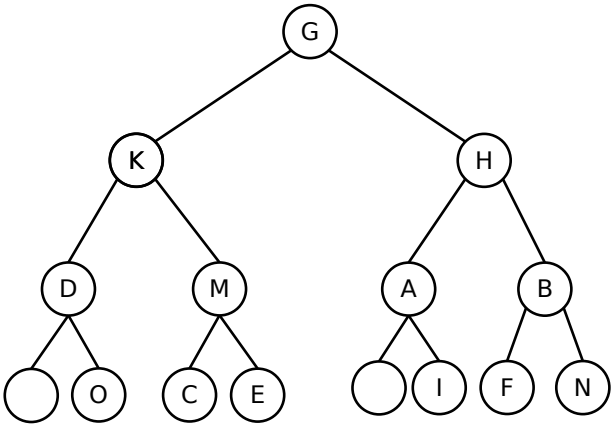

*Figura 8.21: Uma árvore binária **perfeita**.*

As árvores binárias perfeitas têm obviamente as mais rigorosas restrições de tamanho. Só é possível, de fato, ter árvores binárias perfeitas com $2^{h+1} - 1$ nós, se h for a altura da árvore. Portanto, existem árvores binárias perfeitas com 1, 3, 7, 15, 31, ... nós, mas nenhuma entre esses. Em cada uma dessas árvores, 2^h dos nós (quase exatamente a metade) são folhas.

Agora, como veremos, as árvores binárias podem possuir alguns poderes espantosos se os nós dentro delas estiverem organizados de certas formas. Especificamente, uma árvore de busca binária e um *heap* são dois tipos especiais de árvores binárias que obedecem a restrições específicas. Em ambos os casos, o que as torna tão poderosas é o ritmo em que uma árvore cresce à medida que lhe são adicionados nós.

Suponha que tenhamos uma árvore binária perfeita. Para torná-la concreta, digamos que tem altura 3, o que lhe daria 1+2+4+8=15 nós, 8 dos quais são folhas. Agora, o que acontece se aumentarmos a altura desta árvore para 4? Se ainda for uma árvore "perfeita", teremos acrescentado mais 16 nós (todos folhas). Assim, duplicamos o número de folhas, simplesmente adicionando mais um nível. Isto acontece à medida que se vão adicionando mais níveis. Uma árvore de altura 5 duplica novamente o número de folhas (para 32), e a altura 6 duplica novamente (para 64).

Se isto não lhe parece surpreendente, é provavelmente porque não compreende bem a rapidez com que este tipo de **crescimento exponencial** pode acumular-se. Suponha que você tinha uma árvore binária perfeita de altura 30 — certamente não parece uma figura imponente. Poder-se-ia imaginá-la cabendo num pedaço de papel... em altura, por exemplo. Mas verifique os números e descobrirá que tal árvore teria mais de meio bilhão de folhas, mais de duas para cada pessoa no Brasil. Aumente a altura

da árvore para apenas 34 — apenas 4 níveis adicionais — e de repente terá mais de 8 bilhões de folhas, facilmente acima da população do planeta Terra.

O poder de crescimento exponencial só é plenamente alcançado quando a árvore binária é perfeita, uma vez que uma árvore com alguns nós internos "faltando" não carrega a capacidade máxima de que é capaz. Tem alguns buracos. Ainda assim, desde que a árvore seja bastante frondosa (isto é, não é horrivelmente desequilibrada), o enorme crescimento previsto para árvores perfeitas continua a ser aproximadamente o caso.

A razão pela qual isto se chama crescimento "exponencial" é que a quantidade que estamos variando — a altura — aparece como um expoente no número de folhas, que é de 2^h. Cada vez que adicionamos apenas um nível, duplicamos o número de folhas.

Assim, o número de folhas (chamemos-lhe l) é de 2^h, se h for a altura da árvore. Invertendo isto, dizemos que $h = \lg(l)$. A função "lg" é um logaritmo, especificamente um logaritmo com base 2. É isto que os cientistas da computação utilizam frequentemente, em vez de uma base 10 (que se escreve "log") ou uma base e (que se escreve "ln"). Uma vez que 2^h cresce muito, muito rapidamente, segue-se que $\lg(l)$ cresce muito, muito lentamente. Depois da nossa árvore atingir alguns milhões de nós, podemos acrescentar cada vez mais nós sem que a altura da árvore cresça significativamente.

A lição final aqui é simplesmente que um número incrivelmente grande de nós pode ser acomodado numa árvore com uma altura muito modesta. Isto torna possível, entre outras coisas, pesquisar uma enorme quantidade de informação de forma espantosamente rápida… desde que o conteúdo da árvore esteja devidamente arranjado.

Árvores de busca binária (BST's)

Muito bem, então vamos falar sobre como organizar esses conteúdos. Uma árvore de busca binária (BST, do inglês *binary search tree*) é qualquer árvore binária que satisfaz uma propriedade adicional: cada nó é "maior que" todos os nós na sua subárvore esquerda, e "menor que (ou igual a)" todos os nós na sua subárvore direita. Vamos chamar isto de propriedade da BST. As frases "maior que" e "menor que" estão aqui entre aspas porque o seu significado é algo flexível, dependendo do que estamos armazenando na árvore. Se estivermos armazenando números, usaremos a ordem numérica. Se estivermos armazenando nomes, usaremos a ordem alfabética. O que quer que estejamos a armazenar, precisamos simplesmente de uma forma de comparar dois nós para determinar qual dos dois "vai antes" do outro.

Um exemplo de uma BST contendo pessoas é dado na Figura 8.22. Imagine que cada um destes nós contém uma boa quantidade de informação sobre uma determinada pessoa — um registo de funcionário, histórico médico, informação de conta bancária, o que você tiver. Os nós em si são indexados pelo nome da pessoa, e os nós são organizados de acordo com a regra BST. Michel vem depois de Benício/Jéssica/Josabete e antes de Rodolfo/Osvaldo/Monalisa/Xavier em ordem alfabética, e esta relação de ordenação entre pais e filhos repete-se até ao fim da árvore. (Verifique!)

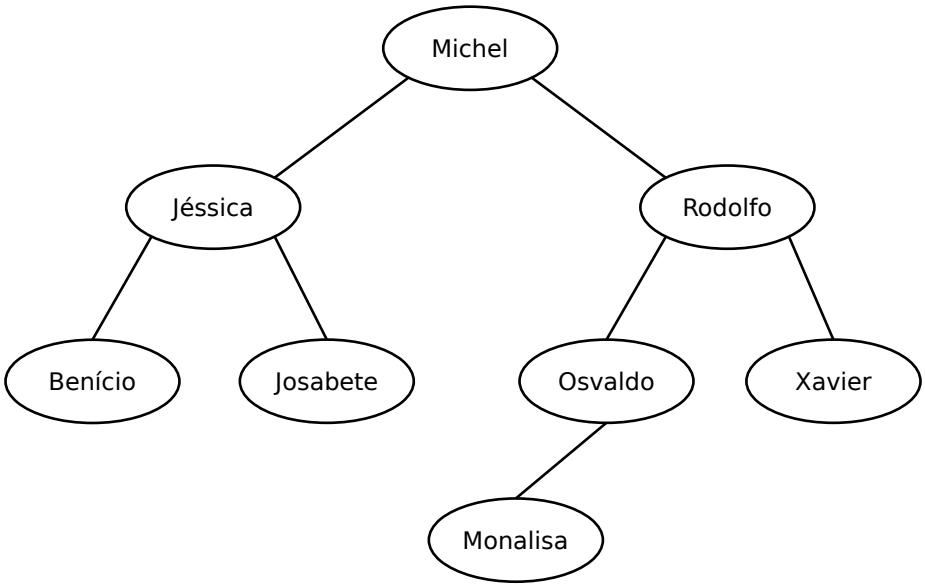

Figura 8.22: Uma árvore de busca binária.

Tenha o cuidado de observar que a regra de ordenação se aplica entre um nó e todo o conteúdo das suas subárvores, e não apenas aos seus filhos imediatos. Este é um erro de principiante que se quer evitar. A sua primeira impressão, ao olhar para a Figura 8.23, abaixo, é considerá-la uma BST. Não é, no entanto, uma árvore de busca binária! Jéssica está à esquerda de Michel, como deve ser, e Neide está à direita de Jéssica, como deve ser. Parece que se verifica. Mas o problema é que Neide é uma descendente da subárvore esquerda de Michel, enquanto que ela deve ser colocada corretamente em algum lugar na sua subárvore direita. E sim, isto importa. Portanto, não se esqueça de verificar as suas BST's de cima abaixo.

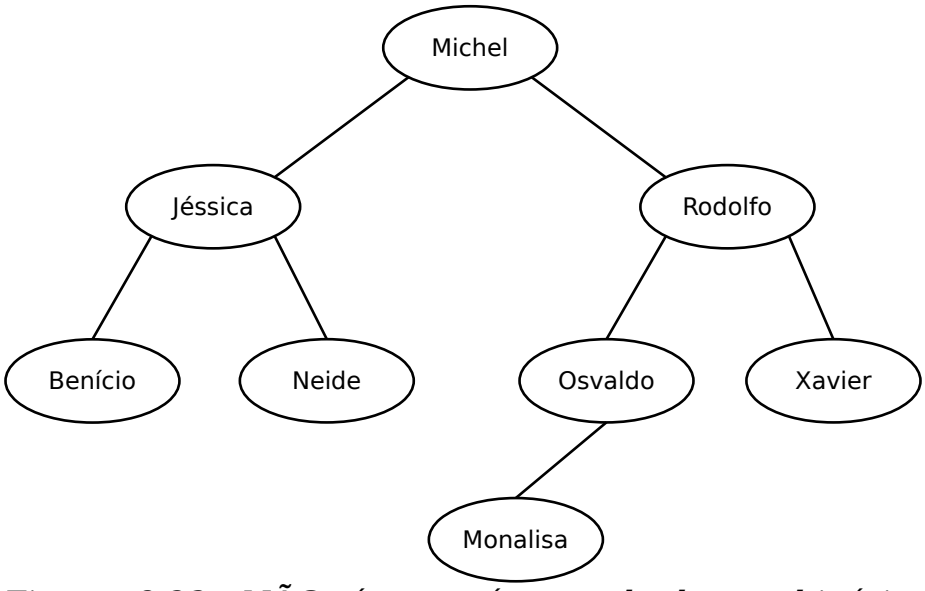

*Figura 8.23: **NÃO** é uma árvore de busca binária, embora se pareça com uma à primeira vista (repare Neide e Michel).*

O Poder das BST's

Muito bem, afinal, o que é que há de especial nas BST's? A principal ideia é perceber que se estiver procurando um nó, tudo o que tem de fazer é começar pela raiz e descer a altura da árvore fazendo uma comparação em cada nível. Digamos que estamos à procura de Monalisa na Figura 8.22. Olhando para Michel (a raiz), sabemos de imediato que Monalisa deve estar na subárvore direita, e não na esquerda, porque ela vem depois de Michel em ordem alfabética. Por isso, olhamos para Rodolfo. Desta vez, descobrimos que Monalisa vem antes de Rodolfo, por isso ela deve estar em algum lugar no ramo esquerdo de Rodolfo. Osvaldo envia-nos novamente para a esquerda, altura em que encontramos a Monalisa.

Com uma árvore deste tamanho, não parece ser assim tão espantoso. Mas suponha que a sua altura fosse 10. Isto significaria cerca de 2000 nós na árvore — clientes, utilizadores, amigos, o que quer que seja. Com uma BST, você só teria de examinar dez desses 2000 nós para encontrar o que procura, enquanto que se os nós estivessem simplesmente numa lista qualquer, você teria de comparar contra cerca de 1000 antes de esbarrar com o que procurava. E à medida que o tamanho da árvore cresce, esta discrepância torna-se (muito) maior. Se quisesse encontrar os registros de uma única pessoa em Nova Iorque, preferiria procurar 7 milhões de nomes, ou 24 nomes? Pois é desse nível de diferença que estamos falando.

Parece quase bom demais para ser verdade. Como é possível uma tal aceleração? O truque é perceber que com cada nó que se olha, se elimina efetivamente metade da árvore restante da nossa consideração. Por exemplo, se estivermos à procura de Monalisa, podemos ignorar toda a metade da árvore à esquerda de Michel sem sequer olhar para ela, e o mesmo acontece com toda a metade direita de Rodolfo. Se descartarmos metade de algo, então metade da metade restante, então metade novamente, não demorará muito até que tenhamos eliminado quase todas as pistas falsas.

Há uma maneira formal de descrever esta aceleração, chamada "Notação Big-O". Os detalhes são um pouco complexos, mas a ideia básica é esta. Quando dizemos que um algoritmo é "O(n)" (pronuncia-se "ó-de-n"), significa que o tempo necessário para executar o algoritmo é proporcional ao número de nós. Isto não implica qualquer número específico de milissegundos ou qualquer coisa — que seja altamente dependente do tipo de hardware de computador que se tem, da linguagem de programação, e de uma miríade de outras coisas. Mas o que podemos dizer sobre um algoritmo O(n) é que se duplicar o número de nós, vai duplicar aproximadamente o tempo de execução. Se quadruplicar o número de nós, vai quadruplicar o tempo de execução. Isto é o que se esperaria.

A busca por "Monalisa" numa simples lista de nomes não ordenados é uma expectativa O(n). Se houver mil nós na lista, em média encontrará Monalisa depois de percorrer 500 deles. (Pode ter sorte e encontrar Monalisa no início, mas depois, claro, pode ter muito azar e não a encontrar até o fim. Esta média é de cerca de metade do tamanho da lista no caso normal). No entanto, se houver um milhão de nós, serão necessários em média 500.000 passos antes de encontrar a Monalisa. Dez vezes mais nós significa dez vezes mais tempo para encontrar a Monalisa, e mil vezes mais significa mil vezes mais tempo. Que chatice.

Contudo, procurar Monalisa numa BST é um processo O(lg n). Lembre-se de que "lg" significa o logaritmo (base 2). Isto significa que duplicar o número de nós dá-lhe um aumento minúsculo no tempo de funcionamento. Suponha que havia mil nós na sua árvore, como acima. Não teria de procurar através de 500 para encontrar Monalisa: só teria de procurar através de dez (porque lg(1000) ≈ 10). Agora aumente-o para um milhão de nós. Não precisaria de procurar através de 500.000 para encontrar Monalisa: só teria de procurar através de vinte. Suponha que tivesse 6 bilhões de nós na sua árvore (aproximadamente a população da terra). Não teria de procurar através de 3 bilhões de nós: só teria de procurar através de trinta e três. Absolutamente extraordinário.

Adicionando nós a uma BST

Encontrar coisas numa BST é rápido como um relâmpago. Acontece que o mesmo se passa para adicionar-lhe coisas. Suponha que adquirimos uma nova cliente chamada Jacira, e precisamos adicioná-la à nossa BST para que possamos recuperar a informação da sua conta no futuro. Tudo o que fazemos é seguir o mesmo processo que seguiríamos se estivéssemos à procura da Jacira, mas assim que encontrarmos o local onde ela estaria, adicionamo-la lá. Neste caso, Jacira vem antes de Michel (ir para a esquerda), e antes de Jéssica (ir novamente para a esquerda), e depois de Benício (ir para a direita). Benício não tem filho direito, por isso colocamos Jacira na árvore precisamente nesse ponto. (Ver Figura 8.24.)

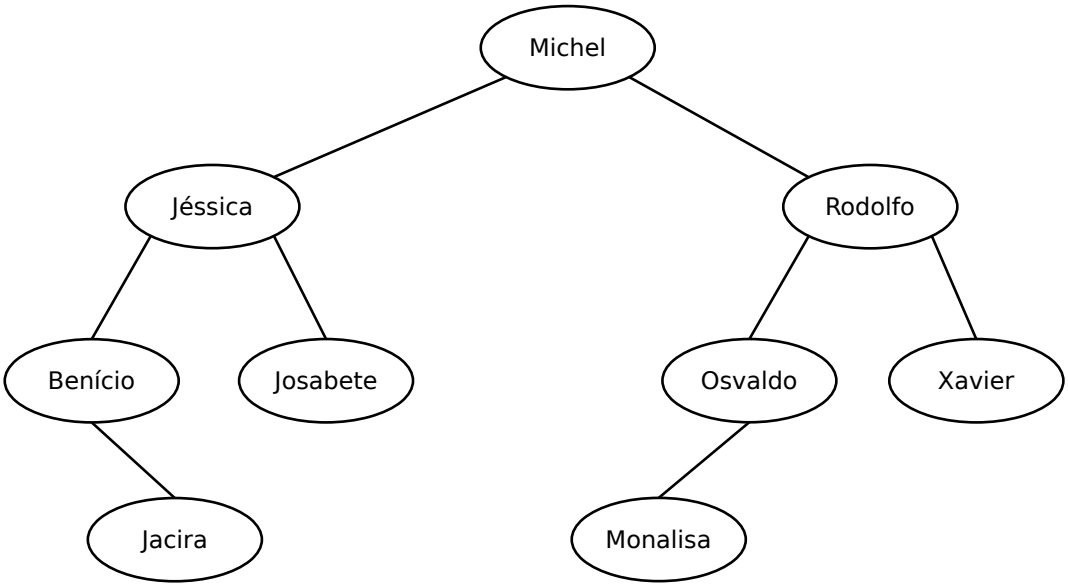

Figura 8.24: A BST depois de adicionar Jacira.

Este processo de adição é também um algoritmo O(lg n), uma vez que só precisamos olhar para um pequeno número de nós igual à altura da árvore.

Note que uma nova entrada torna-se sempre uma folha quando adicionada. De fato, isto permite-nos olhar para a árvore e reconstruir um pouco do que veio antes. Por exemplo, sabemos que Michel deve ter sido o primeiro nó originalmente inserido, e que Rodolfo foi inserido antes de Osvaldo, Xavier, ou Monalisa. Como exercício, acrescente o seu próprio nome a esta árvore (e alguns dos nomes dos seus amigos) para ter a certeza de que pegou o jeito. Quando terminar, a árvore deve evidentemente obedecer à propriedade da BST.

Remoção de nós de uma BST

A remoção de nós é um pouco mais complicada do que a sua adição. Como eliminar uma entrada sem estragar a estrutura da árvore? É fácil ver como eliminar Monalisa: uma vez que ela é apenas uma folha, basta removê-la e estamos terminados. Mas como eliminar Jéssica? Ou, por acaso, Michel?

A sua primeira inclinação pode ser eliminar um nó e promover um dos seus filhos para que suba para o seu lugar. Por exemplo, se eliminarmos Jéssica, poderíamos simplesmente elevar Benício até onde Jéssica estava, e depois mover Jacira para cima também sob Benício. Isto não funciona, no entanto. O resultado se pareceria com a Figura 8.25, com Jacira no lugar errado. Da próxima vez que procurarmos Jacira na árvore, procuraremos à direita de Benício (como devemos), perdendo-a completamente. Jacira foi efetivamente extraviada.

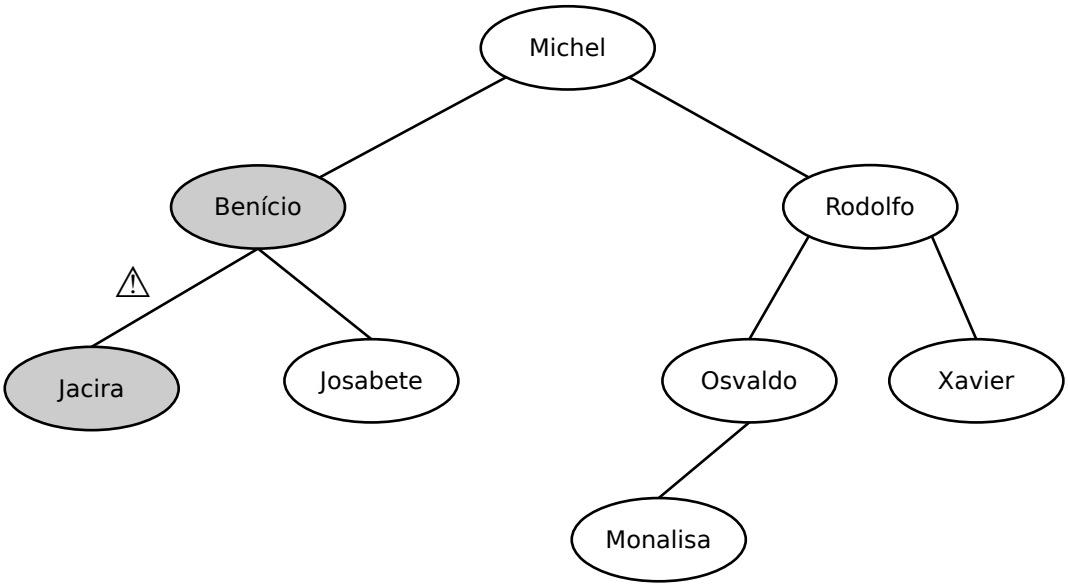

*Figura 8.25: Uma candidata a BST **ERRADA**, após remover Jéssica incorretamente.*

Uma forma correta (existem outras) de fazer uma remoção de nó é substituir o nó pelo descendente mais à esquerda da sua subárvore direita. (Ou, de forma equivalente, o descendente mais à direita da sua subárvore esquerda). A Figura 8.26 mostra o resultado após a remoção de Jéssica. Substituímo-la por Josabete, não porque seja correto promover cegamente o filho direito, mas porque Josabete não tinha descendentes esquerdos. Se tivesse, promovê-la teria sido tão errado como promover Benício. Em vez disso, teríamos promovido o descendente de Josabete mais à esquerda.

Como mais um exemplo, vamos com tudo e removamos o nó raiz, Michel. O resultado é o mostrado na Figura 8.27. Foi o golpe de sorte de Monalisa: ela foi promovida de uma folha até o topo. Por que a Monalisa? Porque ela era a descendente mais à esquerda da subárvore direita de Michel.

Para ver porque é que isto funciona, basta considerar que Monalisa estava imediatamente após Michel, na ordem alfabética. O fato de ele ser um rei e ela uma camponesa era enganador. Os dois eram na realidade muito próximos: consecutivos, de fato,

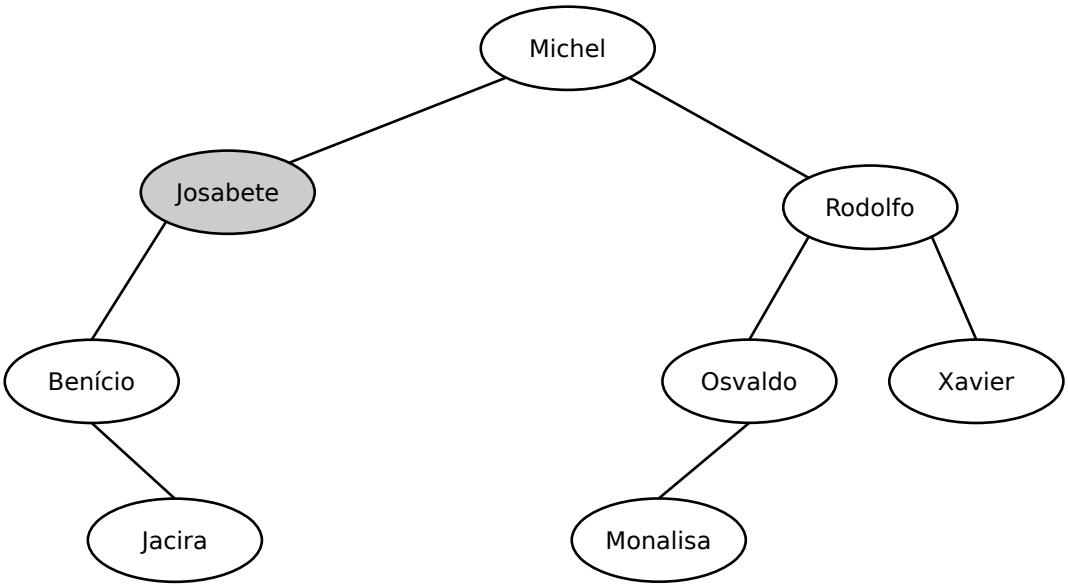

Figura 8.26: A BST depois de remover Jéssica corretamente.

com um percurso em ordem. Assim, substituir Michel por Monalisa evita embaralhar qualquer pessoa para fora da ordem alfabética, e preserva a importantíssima propriedade da BST.

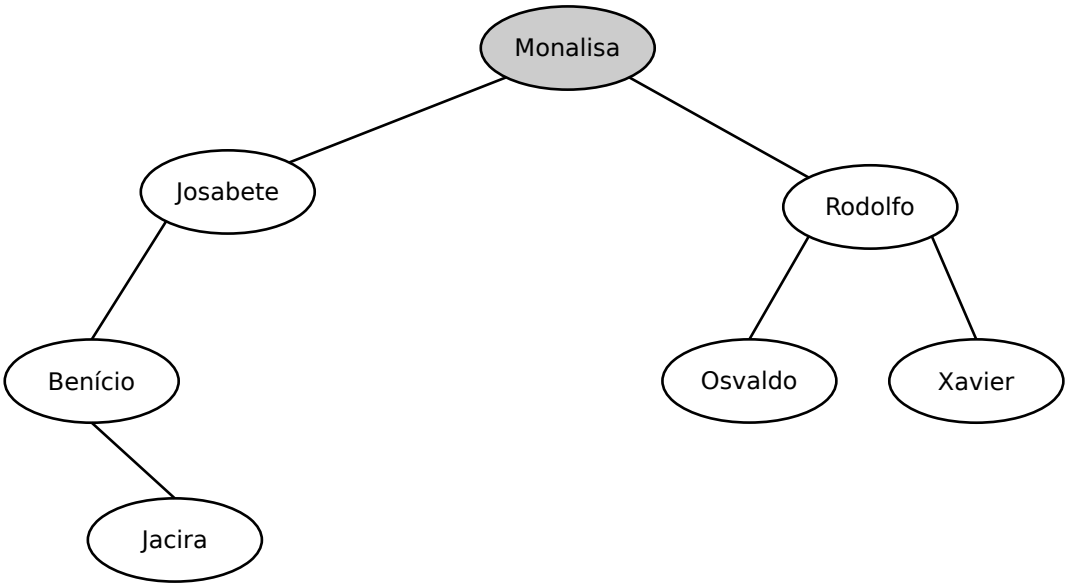

Figura 8.27: A BST depois da remoção de Michel.

Balanceamento

Finalmente, lembrem-se de que esta busca incrivelmente rápida é extremamente dependente de a árvore ser "frondosa", ou seja balanceada. Caso contrário, a aproximação que $h = \lg(l)$ se desfaz. Como um exemplo ridiculamente extremo, considere a Figura 8.28 que contém os mesmos nós que vimos utilizando. Esta é uma árvore de busca binária legítima! (Verifique!) No entanto, procurar um nó nesta monstruosidade não vai ser obviamente mais rápido do que procurar numa velha lista. Estamos de volta ao desempenho $O(n)$.

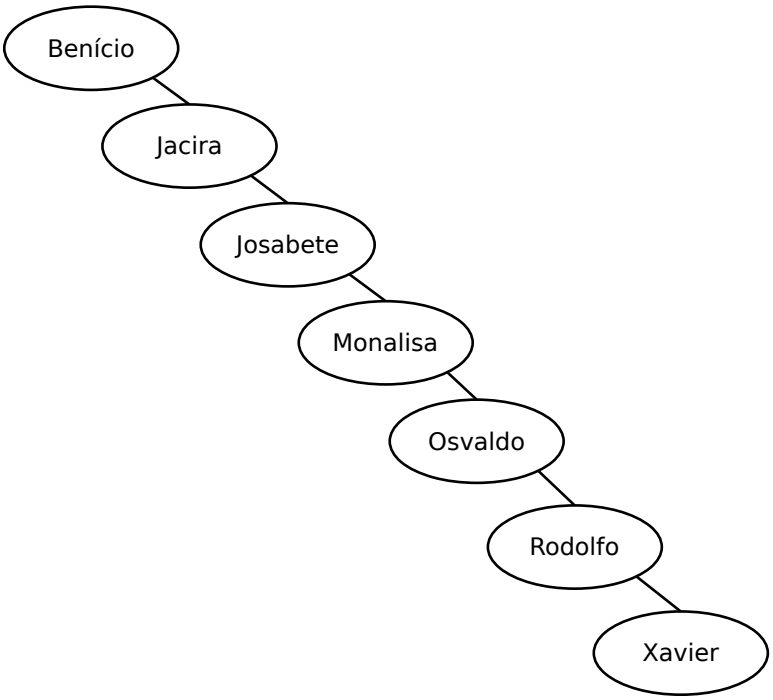

Figura 8.28: Uma BST incrivelmente ruim, mas tecnicamente legítima.

Na prática, há três maneiras de lidar com isto. Uma abordagem é simplesmente não se preocupar com isso. Afinal de contas, desde que estejamos inserindo e removendo nós aleatoriamente, sem padrão discernível, as chances de obter uma árvore tão desequilibrada como a Figura 8.28 são astronomicamente pequenas. É tão provável como atirar um baralho de cartas ao ar e vê-lo cair numa pilha arrumada. A lei da entropia diz-nos que vamos obter uma mistura de ramos curtos e ramos longos, e que numa árvore grande, o desequilíbrio será mínimo.

Uma segunda abordagem é a de reequilibrar periodicamente a árvore. Se o nosso website for desligado para manutenção de vez em quando, de qualquer forma, poderemos reconstruir a nossa árvore a partir do zero, inserindo os nós numa árvore nova, numa ordem benéfica. Em que ordem devemos inseri-los? Bem, lembrem-se de que o nó que for inserido primeiro será a raiz. Isto sugere que queremos inserir primeiro o nó médio na nossa árvore, de modo a que Monalisa se torne a nova raiz. Isto deixa metade dos nós para a sua subárvore esquerda e metade para a sua direita. Se seguir este processo logicamente (e recursivamente), perceberá que a seguir queríamos inserir os nós médios de cada metade. Isto equivaleria a Jacira e Rodolfo (em qualquer ordem). Isto devolve-nos uma árvore perfeitamente equilibrada com intervalos regulares, tornando quaisquer grandes desequilíbrios ainda mais improváveis (e de curta duração).

Em terceiro lugar, existem estruturas de dados especializadas sobre as quais poderá aprender em cursos futuros, tais como árvores AVL e árvores rubro-negras, que são árvores de busca binária que acrescentam regras extras para evitar desequilíbrios. Basicamente, a ideia é que quando um nó é inserido (ou removido), certas métricas são verificadas para garantir que a alteração não causou um desequilíbrio demasiadamente grande. Se o fez, a árvore é ajustada de modo a minimizar o desequilíbrio. Isto tem um ligeiro custo adicional sempre que a árvore é mudada, mas evita qualquer

possibilidade de uma árvore desequilibrada que causaria uma busca lenta a longo prazo.

8.3. Palavra final

Ufa, isso foi bastante informação sobre estruturas. Antes de continuarmos a nossa caminhada no próximo capítulo com um tema completamente diferente, deixo-vos com este pensamento sintético. Seja BST o conjunto de Árvores de Busca Binária, e seja BT o conjunto de Árvores Binárias. Que RT seja o conjunto de árvores enraizadas, e T seja o conjunto de árvores (livres ou enraizadas). Finalmente, que CG seja o conjunto de grafos conexos, e G o conjunto de todos os grafos. Então temos:

$$\text{BST} \subset \text{BT} \subset \text{RT} \subset \text{T} \subset \text{CG} \subset \text{G}.$$

É uma coisa linda.

8.4. Créditos

Todas as seções, foram adaptadas (traduzidas e modificadas) de [1], que está disponível sob a licença Creative Commons Attribution-ShareAlike 4.0 International.

8.5. Referências

1. Davies, Stephen. *A Cool Brisk Walk Through Discrete Mathematics*. Disponível em http://www.allthemath.org/vol-i/

8.6. Licença