Árvores

> *Com organização e tempo, acha-se o segredo de fazer tudo e bem feito.*
>
> PITÁGORAS

Quando aumenta o volume de itens a serem manipulados, é fundamental que eles sejam organizados de forma que sua manipulação seja eficiente.

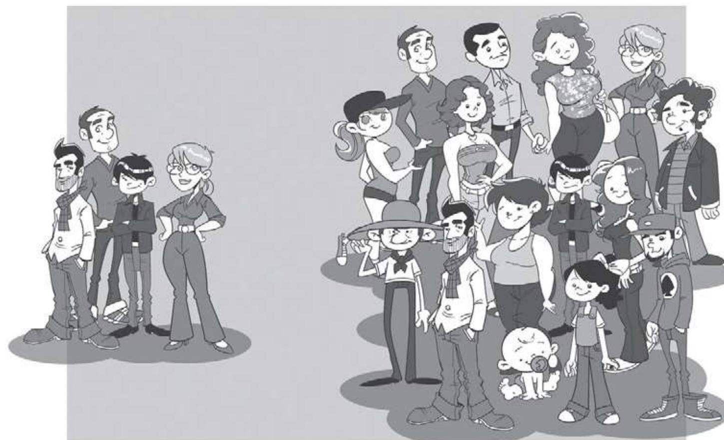
OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- entender o motivo da existência da estrutura de dados árvore;
- conhecer os conceitos, terminologias utilizadas e a estruturação de dados na forma de árvore;
- criar e manipular dados na forma de árvores binárias.

Para começar

Sua tarefa: localizar o José da Silva e Souza nas imagens apresentadas.



Observando a primeira imagem, percebemos que não seria difícil localizar determinada pessoa dentre as ali apresentadas. Essa tarefa seria realizada com facilidade e em curto período de tempo. Por outro lado, o mesmo não pode ser dito em relação à segunda imagem. Localizar uma pessoa no meio da multidão não é tarefa fácil e nem mesmo poderia ser atendida num curto período de tempo.

Precisaríamos, de alguma forma, estabelecer uma organização na multidão para que a tarefa proposta fosse viável num tempo aceitável. Sabemos que uma das formas de organização mais antigas é a hierárquica. Na organização hierárquica da multidão poderíamos estabelecer critérios que permitissem agrupar as pessoas de acordo com determinadas características. Isso reduziria sensivelmente o espaço de busca, uma vez que não seria necessária a busca nos agrupamentos cujos integrantes não tivessem as características da pessoa procurada.

Outro exemplo seria localizar uma fotografia digital tirada numa viagem de férias feita há muitos anos e armazenada como arquivo num computador de uso pessoal, ou mesmo num repositório virtual. Imagine se todos os arquivos que você armazenou não estivessem organizados em pastas — seria quase impossível localizar a fotografia. É por esse motivo que os sistemas de arquivos são organizados de forma hierárquica.

Foi nesse contexto que surgiu a estrutura de dados denominada árvore. Nela, as informações são hierarquizadas segundo critérios que se mostram adequados a cada situação. Com a organização em árvores, é possível reduzir

de forma sensível o espaço de busca, tornando mais fáceis tarefas como as citadas.

Atenção

Árvores são estruturas em que os dados são dispostos de forma hierárquica. Nelas, os dados são armazenados em nós. Existe um nó principal, conhecido como *raiz*, a partir do qual surgem as ramificações, conhecidas como *subárvores*.

Para compreender melhor esse novo conceito, procure criar uma estruturação hierárquica na forma de árvore de diretórios (pastas) para suas fotografias digitais, de forma que seja fácil localizar determinada foto.

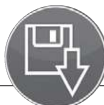
Dica

Para realizar essa tarefa, imagine como você procuraria uma fotografia. A partir disso, qual(is) critério(s) você consideraria para agrupar suas fotos em álbuns, pensando na facilidade futura para encontrá-la?

Conseguiu? Como ficou?

É provável que você tenha tido diferentes ideias, como, por exemplo, organizar as fotografias pelo ano e, dentro de cada ano, pelo mês, ou por tipo de relação social (família, amigos, colegas de trabalho, ou ainda por tipo de evento: férias, trabalho, lazer no cotidiano, festas etc.). Você pode ter misturado diferentes critérios, mas tudo bem. O mais importante é haver na hierarquia estabelecida uma forma de estruturação que facilite sua vida no futuro. Agora você já sabe o porquê da existência da estrutura de dados conhecida como árvore!!!

Neste capítulo, num primeiro momento, conheceremos o conceito, a terminologia e as formas de representação de árvores como estruturas de dados. Depois, estudaremos as árvores binárias, apresentando suas operações básicas. Vamos lá!



Conhecendo a teoria para programar

Como já descrito, a estrutura de dados árvore é uma forma de organização hierárquica em que os dados são armazenados em elementos conhecidos como *nós*. Portanto, um conjunto finito vazio (ou não) de nós compõe uma árvore.

Nessa estrutura existe um elemento principal, que é o *nó raiz***.** Pode haver outros nós associados a ele, os quais seriam seus filhos ou descendentes. Os *nós filhos* formam subárvores do *nó pai* ou antecessor. Essas subárvores apresentam a mesma estrutura de árvore definida, ou seja, os nós filhos do nó raiz poderão ter seus próprios descendentes, e assim por diante.

Para ilustrar esse conceito, a Figura 11.1 mostra a representação gráfica de uma árvore contendo nomes do nosso país, de alguns de seus estados e algumas cidades. Nessa figura temos como nó raiz aquele que contém o nome *Brasil*. Como nós filhos temos os *estados*, que, por sua vez, têm como filhos os nós contendo os *municípios*.

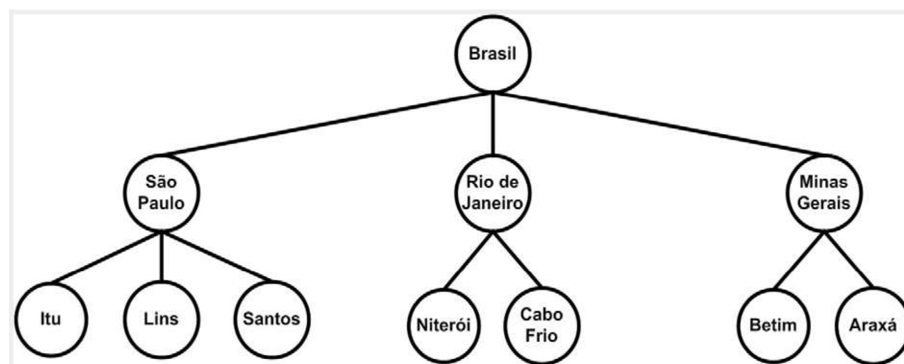


FIGURA 11.1: Árvore contendo país, estados e municípios.

Definição

Uma árvore *A* pode ser definida como:

1. estrutura vazia, A ={};
2. conjunto finito e não vazio de nós, no qual existe um nó raiz *R* e nós que fazem parte de subárvores de *A*, ou seja, A = {R,A1,A2,A3,...,An}.

Para ilustrar essa definição, vamos considerar a árvore *A* (Figura 11.2), em que temos a raiz B0 e as subárvores A1, A2 A3 e A4:

Conceitos

- **Nó raiz:** principal elemento da árvore, não apresenta ancestrais, e todos os nós da árvore são seus descendentes (diretos ou indiretos);
- **Nó pai**: elemento que apresenta descendentes na árvore, podendo ter um ou mais filhos;
- **Nó filho**: elemento que descende de algum outro nó da árvore, tendo apenas um nó pai;
- **Nó folha**: nó que não apresenta descendentes;

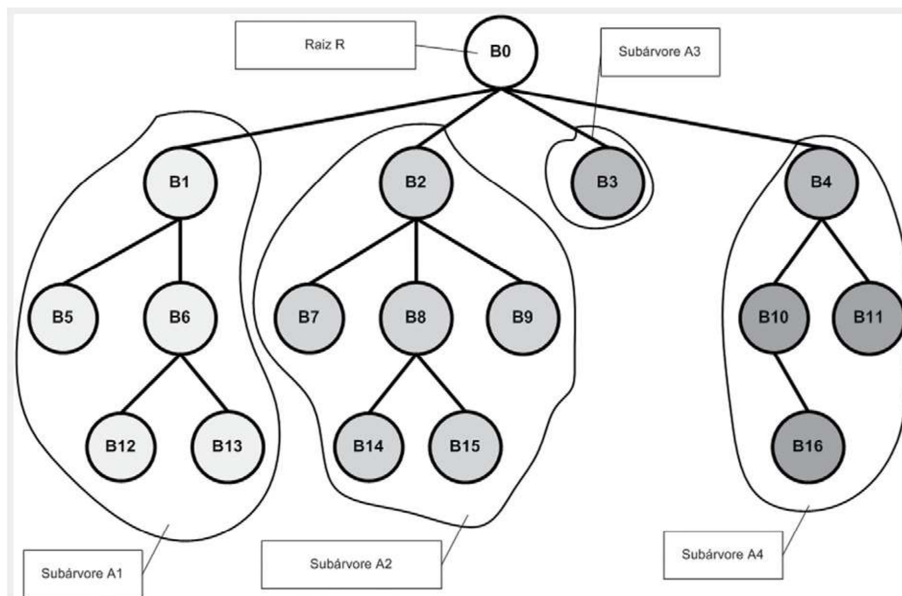


FIGURA 11.2: Árvore A.

- **Nível de um nó**: o nível do nó raiz é 0 (N = 0), e o nível dos nós restantes é igual ao nível do seu nó pai acrescido de 1. Para exemplificar, na Figura 11.3, o nível de B0 é 0, B2 é 1, B5 é 2, e B16 é 3;

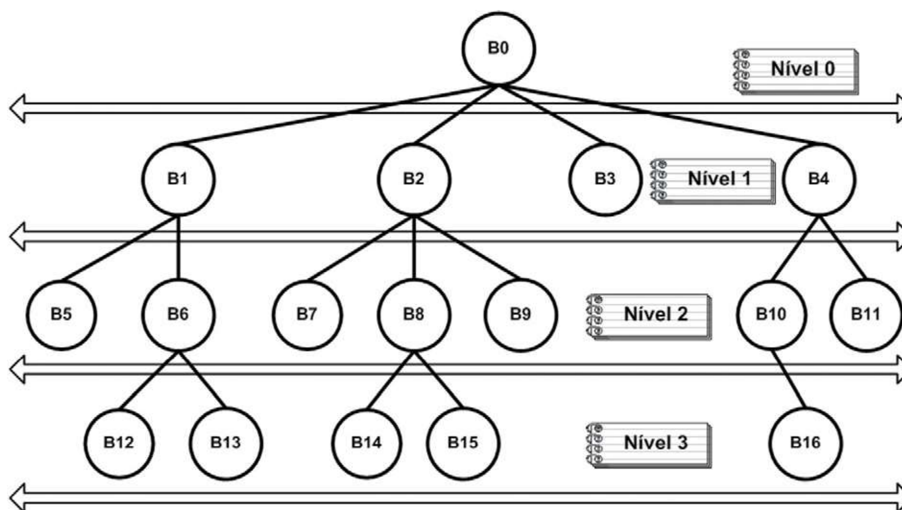


FIGURA 11.3: Nível de um nó da árvore.

- **Grau de um nó:** o grau de determinado nó é dado pelo número de seus filhos. Para exemplificar, na Figura 11.4, o grau do nó B0 é 4, e o do nó B6 é 2;

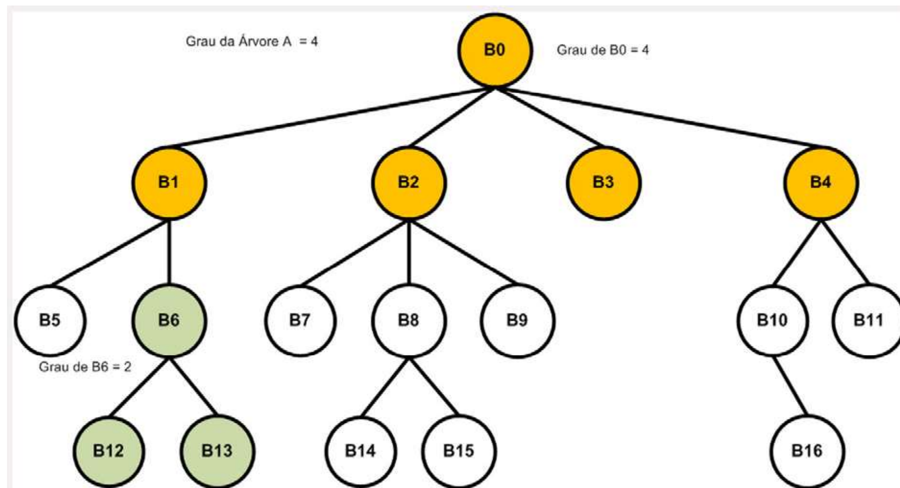


FIGURA 11.4: Grau de um nó e grau da árvore.

- **Grau da árvore:** uma árvore *A* tem grau igual ao grau máximo verificado para seus nós, ou seja, é igual ao do nó que apresenta mais filhos. Na árvore da Figura 11.4 temos o grau da árvore *A* igual a 4;
- **Altura de um nó:** a altura de um nó Bi é igual à maior distância entre Bi e um nó folha que seja seu descendente. Para exemplificar, na Figura 11.5 a altura de B0 é 3, a de B5 é 0, e a de B10 é 1;

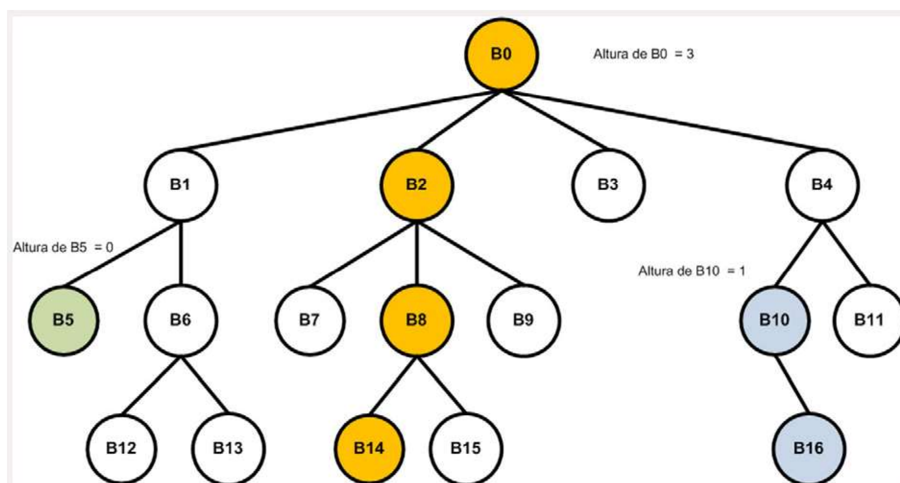


FIGURA 11.5: Altura de um nó.

- **Altura de uma árvore:** a altura de uma árvore corresponde à altura do nó raiz. Na árvore da Figura 11.6 temos a altura da árvore *A* igual a 3.

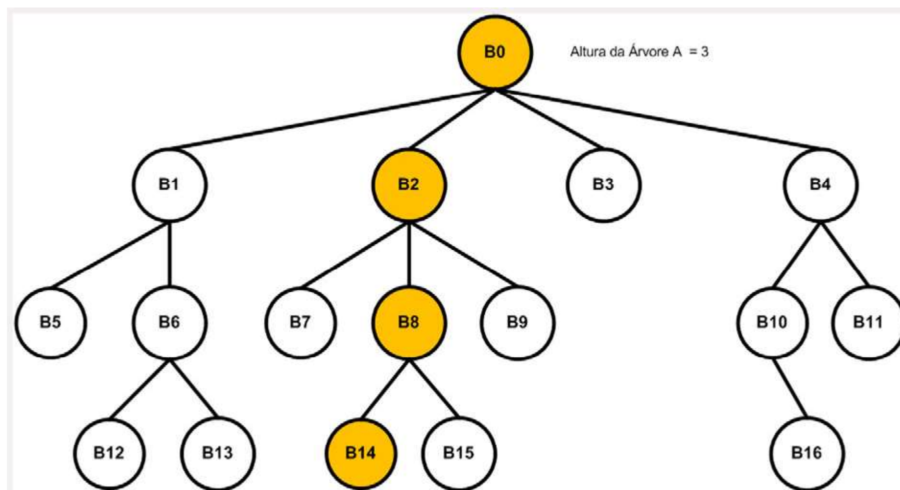


FIGURA 11.6: Altura da árvore.

Formas de representação

A forma gráfica de representação para a estrutura de dados árvore verificada na Figura 11.2 é a mais utilizada e difundida. Porém, existem outras formas de representação que merecem destaque, como as mencionadas a seguir.

Diagrama de Venn (conjuntos aninhados)

O Diagrama de Venn foi criado em 1880, pelo lógico matemático inglês John Venn, para a representação de relações entre conjuntos. Consiste basicamente em criar círculos contendo todos os elementos do conjunto.

Ao considerarmos a estrutura de dados árvore um conjunto contendo o nó raiz e as subárvores formadas por seus descendentes, é fácil imaginar como seria a representação da árvore *A* (Figura 11.2) pelo Diagrama de Venn. Dentro do círculo que representaria a árvore *A*, teríamos o nó raiz B0 e círculos representando as subárvores formadas por seus nós filhos. Dentro dos círculos de cada subárvore ficam o nó raiz dessas subárvores e novos círculos que representam as subárvores compostas pelos filhos da raiz da subárvore, e assim por diante, até chegarmos aos nós folhas. Na Figura 11.7 temos a representação da árvore *A* por um Diagrama de Venn.

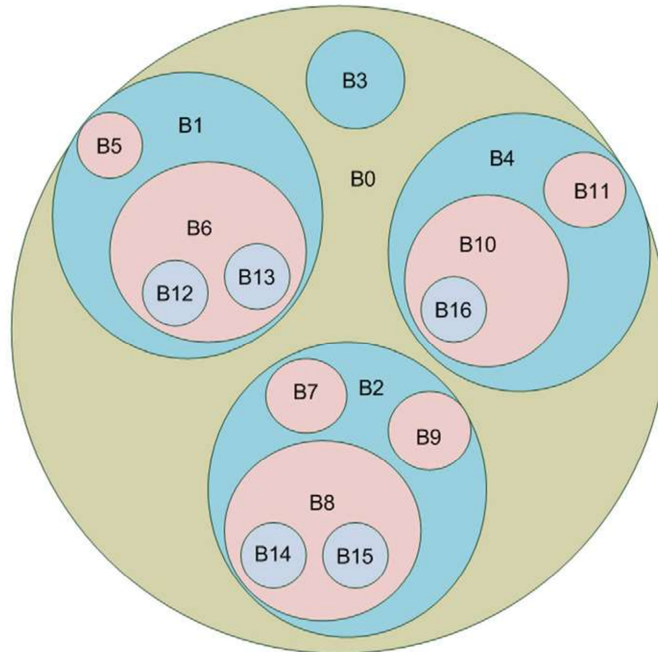


FIGURA 11.7: Árvore *A* representada por um Diagrama de Venn.

Parênteses aninhados

Essa forma de representação assemelha-se à observada no Diagrama de Venn. A diferença é que, nesse caso, utilizam-se parênteses em substituição aos círculos na delimitação do escopo dos conjuntos. Apesar de oferecer uma visualização inferior das relações entre os conjuntos, talvez uma vantagem dessa forma de representação esteja na possibilidade de representar a estrutura de dados árvore em uma única linha. A árvore *A*, representada por parênteses aninhados, ficaria:

```
(B0(B1(B5)(B6((B12)(B13)))(B2(B7)(B8(B14)(B15)))(B3)(B4(B10(B16))(B11)))
```

Atenção:

Na literatura, encontramos também os conceitos de *ordem de um nó* e *ordem de uma árvore*. O primeiro é usado por outros autores para representar o número (grau) mínimo de filhos de um nó, e alguns autores utilizam *ordem* como número máximo de filhos. Cuidado! O mesmo vale para *ordem de uma árvore*, que pode ser igual à *ordem do nó* com número mínimo ou máximo de filhos, dependendo do autor.

Árvores binárias

Numa estrutura de dados árvore, quando restringimos a dois o número máximo de filhos para um nó, temos uma *árvore binária*. Uma árvore binária

é caracterizada como um conjunto finito vazio (ou não) de nós. Esses nós são apresentados na forma de um nó raiz e seus descendentes, organizados em uma subárvore esquerda e uma subárvore direita.

Para melhor compreensão, observe a árvore binária da Figura 11.8.

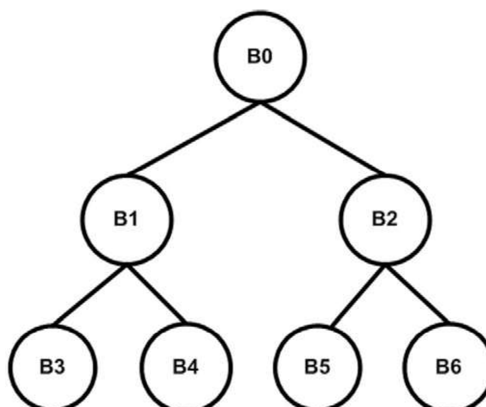


FIGURA 11.8: Árvore binária.

Nessa árvore temos B0 como nó raiz, B1 e B2 como nós filhos de B0, e assim por diante. Verificamos, também, que cada nó filho é um nó raiz da subárvore gerada a partir dele.

As árvores binárias herdam todos os conceitos existentes para a estrutura de dados árvore já vista, como: tipos de nó, nível de um nó ou árvore, grau de um nó ou árvore, e altura de um nó ou árvore. Da mesma forma, são aplicáveis as formas de representação estudadas.

Ao considerarmos que em determinado nível *N* de uma árvore binária temos *K* nós, teremos no máximo 2*K nós no nível imediatamente subsequente (**N + 1**). Dessa forma, como temos apenas 1 nó no nível 0, teremos no máximo 2 nós no nível 1, 4 no nível 2, e assim por diante.

Conceito

O número máximo de nós de uma árvore binária no nível *N* será 2^N. Por indução, é possível concluir que o número máximo de nós da árvore de nível *T* será $2^{T+1} - 1$.

Classificação para árvores binárias

Dependendo da distribuição dos nós em uma árvore binária, teremos as seguintes classificações: *árvore estritamente binária*, *árvore binária completa* e *árvore binária quase completa*.

Árvore estritamente binária

Nas situações em que, para todos os nós que não sejam nós folhas, não existem subárvores vazias, a árvore será denominada *árvore estritamente binária*. Um exemplo pode ser visto na Figura 11.9.

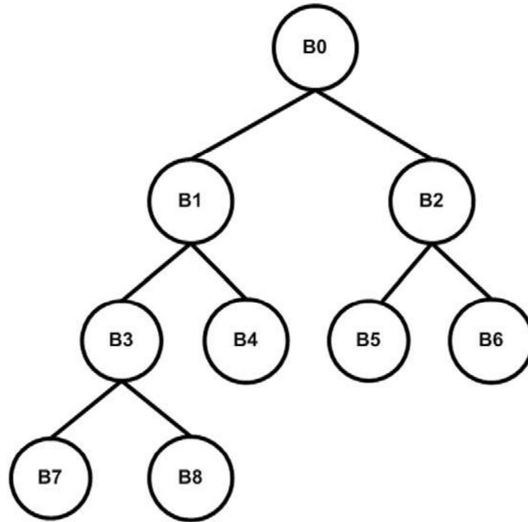


FIGURA 11.9: Árvore estritamente binária.

Árvore binária completa

Quando observamos a árvore da Figura 11.9, percebemos que nem todos os nós folhas apresentam o mesmo nível. De outro lado, é possível que tenhamos uma árvore estritamente binária em que todos os nós folhas estejam no mesmo nível. Nessa situação temos uma árvore binária completa; veja a Figura 11.10.

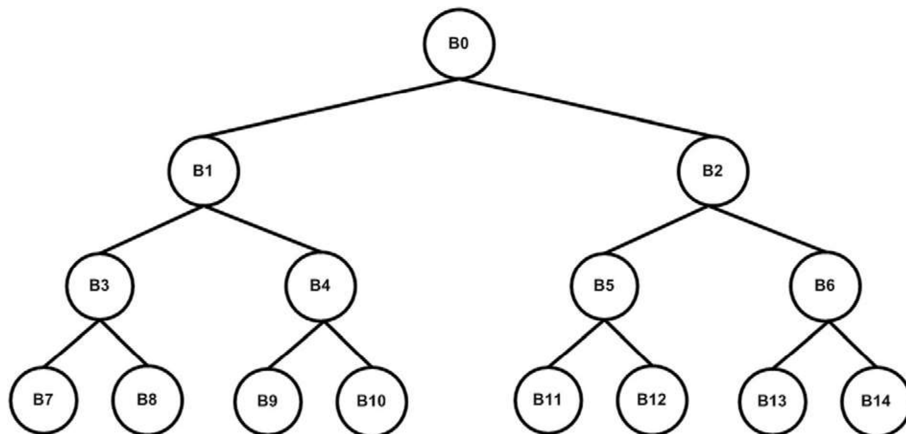


FIGURA 11.10: Árvore binária completa.

Árvore binária quase completa

Uma árvore binária que atender às condições a seguir será considerada árvore binária quase completa:

- todos os nós folhas estão no nível *N* ou N-1;
- para todo nó Bn que possuir um descendente direito no nível *N* (nível máximo da árvore), todo descendente esquerdo de Bn deverá ser nó folha no nível *N*.

Para compreender melhor, observe a Figura 11.11. Na árvore (a) temos a caracterização de uma árvore binária quase completa; já na árvore (b) isso não ocorre, ou seja, ela não é uma árvore binária quase completa, por não atender à condição 2.

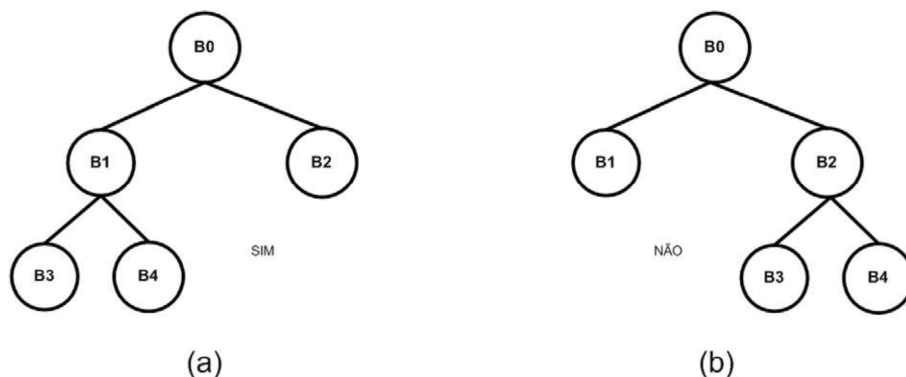


FIGURA 11.11: Caracterização de uma árvore binária quase completa.

Implementação de uma árvore binária

Para implementar uma árvore binária, existe a possibilidade de utilizar *alocação estática* ou *alocação dinâmica de memória*.

Implementação com alocação estática

O uso de alocação estática para a representação de uma árvore binária ocorre por meio da distribuição dos nós da árvore ao longo de um vetor (*array*). Essa distribuição ocorre da seguinte forma:

- o nó raiz fica na posição inicial do vetor (aqui considerada 0);
- para cada nó em determinada posição *i* do vetor, seu filho esquerdo ficará na posição **2*i + 1,** e seu filho direito ficará na posição **2*i + 2**.

Para ilustrar, veja na Figura 11.12 como ficaria a árvore da Figura 11.10 alocada num vetor.

Nos exemplos (I), (II) e (III) podemos notar que os filhos de B0 (posição 0) estão respectivamente nas posições 1 (= 2*0 + 1) e 2 (=2*0 + 2)), da

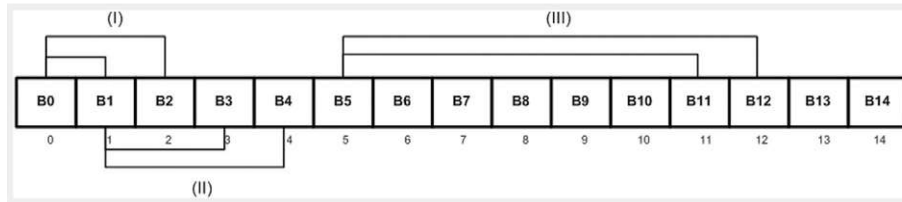


FIGURA 11.12: Árvore binária alocada num vetor.

mesma forma que os filhos de B1 estão nas posições 3 e 4 e os filhos de B5 nas posições 11 e 12.

Implementação com alocação dinâmica

O uso de alocação dinâmica para a representação de uma árvore binária ocorre através da definição de um registro contendo 3 campos básicos: um para armazenar o conteúdo do nó, uma para a ligação esquerda (nó filho à esquerda) e outro para a ligação direita (nó filho à direita), conforme a Figura 11.13.

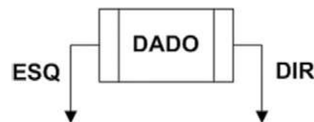


FIGURA 11.13: Representação de um nó alocado dinamicamente.

Para ilustrar, vejamos na Figura 11.14 como ficaria a árvore da Figura 11.10 alocada dinamicamente.

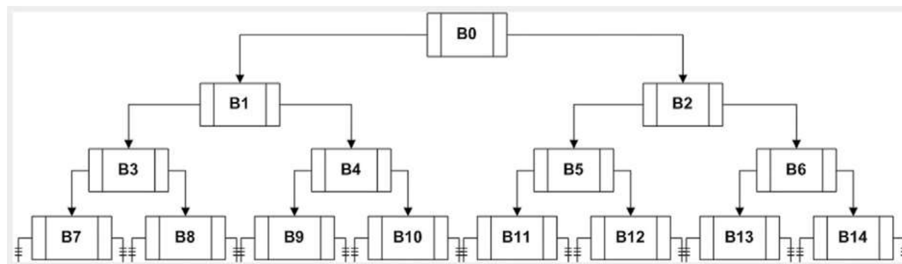


FIGURA 11.14: Representação de uma árvore binária alocada dinamicamente.

Dica

O uso de alocação estática tem como vantagem a simplicidade na manipulação de um vetor, mas, por outro lado, pode acarretar desperdício de espaço de armazenamento no caso de nós vazios. Já a alocação dinâmica se apresenta de forma oposta, com manipulação mais complexa da estrutura de armazenamento, mas eliminação do desperdício verificado na alocação estática.

Operações básicas numa árvore binária

A manipulação de uma árvore binária se dá através de diferentes operações possíveis. Neste livro, vamos focar as operações básicas de maior utilidade e difusão: *percurso* (também conhecido como *travessia* ou *varredura*), *inserção* e *remoção*.

Antes de começarmos a discutir e apresentar essas operações, necessitamos apresentar formas de declarar e criar uma árvore binária, que, como descrito, pode ser feita de forma estática ou de forma dinâmica. A forma estática consiste em apenas declarar um vetor e manipulá-lo como descrito no item "Implementação com Alocação Estática", porém, devido ao alto desperdício de memória provocado por essa forma, ela não é a mais utilizada. Resta-nos, portanto, fazer uso da forma dinâmica, que, deste ponto em diante, será a forma padrão adotada.

Declaração de uma árvore binária

Consiste na definição do nó por meio de um registro. Os campos que compõem esse registro são: *dado*, que armazena o(s) elemento(s) de dado(s) do nó; *esq*, que armazena a ligação com o nó filho à esquerda; e *dir*, que armazena a ligação com o nó filho à direita.

Código 11.1

```
typedef struct tipo_no no;
struct tipo_no
{
  tipo_dado dado;
  struct tipo_no *esq;
  struct tipo_no *dir;
};
```

Alguns autores utilizam também um campo adicional, denominado *pai*, que armazena a ligação com o nó pai. Esse campo pode ser descartado se a forma de percorrer a árvore for da raiz em direção às folhas. Por outro lado, se a forma adotada for das folhas para a raiz, os campos *esq* e *dir* é que serão descartados. Neste livro, adotaremos o percurso da raiz para as folhas, por isso a ausência do campo *pai* no registro que representa o nó.

Para facilitar a compreensão dos conceitos, nos exemplos sobre árvores vamos adotar que tipo_dado será um valor do tipo *inteiro*. Assim:

```
typedef int tipo_dado;
```

Percurso de árvore binária

Quando desejamos apresentar todos os elementos de uma árvore binária (independentemente de existir ou não alguma ordenação interna entre os elementos), é necessário estabelecermos um padrão para o percurso da árvore. No caso das árvores binárias, encontramos três formas básicas de percurso: *pré-ordem*, *em-ordem* e *pós-ordem*.

A distinção entre as formas de percurso se dá pela ordem em que os nós são visitados.

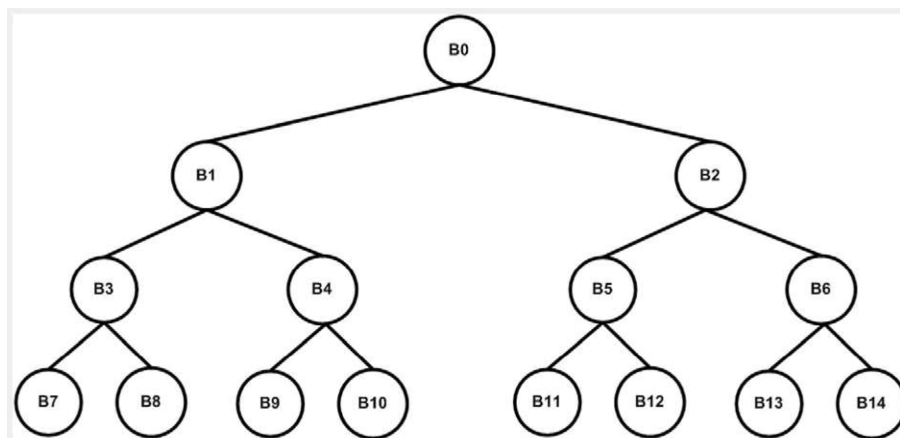
Percurso em pré-ordem

Nessa forma de percurso, os nós de uma árvore binária são visitados de forma recursiva na seguinte ordem:

- apresenta o elemento do nó visitado;
- passa para o elemento do nó filho à esquerda (subárvore à esquerda);
- passa para o elemento do nó filho à direita (subárvore à direita).

Para ilustrar, tomemos novamente o exemplo da árvore binária apresentada na Figura 11.10 e vejamos como ficaria a ordem de apresentação dos elementos dessa árvore num percurso pré-ordem.

PERCURSO PRÉ-ORDEM:



B0-B1-B3-B7-B8-B4-B9-B10-B2-B5-B11-B12-B6-B13-B14

A seguir, tem-se a implementação da função pre_ordem, que apresenta como saída os elementos da árvore binária, segundo os critérios já descritos.

Código 11.2

```
void pre_ordem(no *Raiz)
{
   if (Raiz != NULL)
  {
     printf("%d ", Raiz->dado);
     pre_ordem(Raiz->esq);
     pre_ordem(Raiz->dir);
   }
}
```

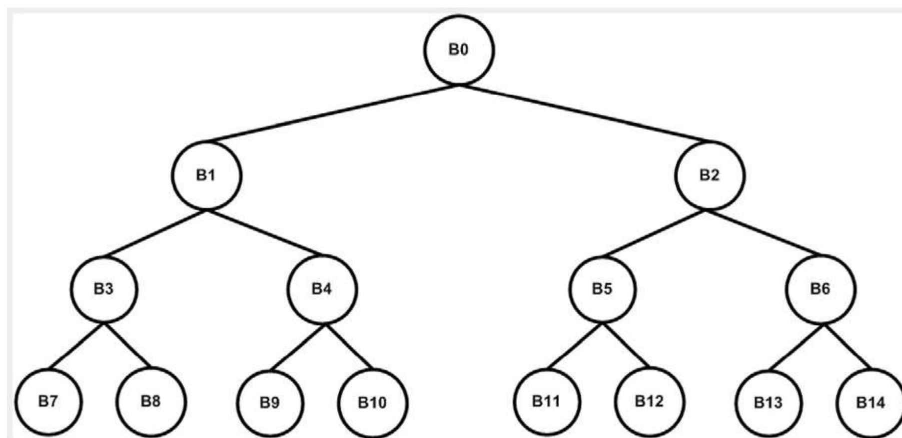
Percurso em em-ordem

Nessa forma de percurso, os nós de uma árvore binária são visitados de forma recursiva na seguinte ordem:

- passa para o elemento do nó filho à esquerda (subárvore à esquerda);
- apresenta o elemento do nó visitado;
- passa para o elemento do nó filho à direita (subárvore à direita).

Reproduzindo novamente a árvore da Figura 11.10, vejamos como ficaria a ordem de apresentação dos elementos dessa árvore num percurso em-ordem.

PERCURSO EM-ORDEM:



B7-B3-B8-B1-B9-B4-B10-B0-B11-B5-B12-B2-B13-B6-B14

A seguir, tem-se a implementação da função em_ordem, que apresenta como saída os elementos da árvore binária, segundo os critérios já descritos.

Código 11.3

```
void em_ordem(no *Raiz)
{
   if (Raiz != NULL)
  {
     em_ordem(Raiz->esq);
     printf("%d ", Raiz->dado);
     em_ordem(Raiz->dir);
   }
}
```

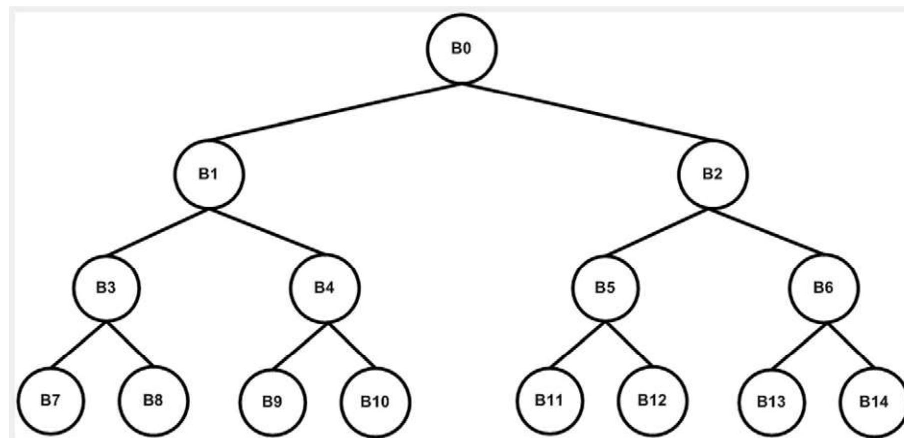
Percurso em pós-ordem

Nessa forma de percurso, os nós de uma árvore binária são visitados de forma recursiva na seguinte ordem:

- passa para o elemento do nó filho à esquerda (subárvore à esquerda);
- passa para o elemento do nó filho à direita (subárvore à direita);
- apresenta o elemento do nó visitado.

Usando mais uma vez a reprodução da árvore da Figura 11.10, vejamos como ficaria a ordem de apresentação dos elementos dessa árvore num percurso pós-ordem.

PERCURSO PÓS-ORDEM:



B7-B8-B3-B9-B10-B4-B1-B11-B12-B5-B13-B14-B6-B2-B0

A seguir, tem-se a implementação da função *pos_ordem*, que apresenta como saída os elementos da árvore binária, segundo os critérios já descritos.

Código 11.4

```
void pos_ordem(no *Raiz)
{
   if (Raiz != NULL)
  {
     pos_ordem(Raiz->esq);
     pos_ordem(Raiz->dir);
     printf("%d ", Raiz->dado);
   }
}
```

Inserção numa árvore binária

Quando abordamos a inserção numa árvore binária, é possível adotar diferentes formas de preenchimento. Um exemplo seria inserir novos elementos da esquerda para a direita, preenchendo a árvore por nível, ou seja, escolhendo sempre o primeiro nó livre (vazio) mais à esquerda no nível ainda não totalmente preenchido. A dificuldade, nessa forma de preenchimento, é que não existe qualquer organização dos dados (ordenação) que permita, posteriormente, uma busca mais eficiente por determinado elemento pertencente à árvore.

Assim, para se usar a árvore binária como estrutura que auxilie na busca (*árvore binária de busca*), adotou-se que, para cada elemento da árvore, existe uma *chave* associada ao mesmo, além dos *dados secundários* desse elemento. Essa *chave* será a responsável por promover a ordenação necessária entre os elementos da árvore.

No contexto de árvores binárias de busca, adotou-se também que os elementos com chaves *menores* que a contida num elemento que está armazenado no nó *N* da árvore serão inseridos como descendentes à *esquerda* de *N* na árvore. Consequentemente, os elementos com chaves *maiores* serão inseridos como descendentes à *direita* de *N*.

Atenção

Lembramos aqui que, quando manipulamos uma árvore binária como uma estrutura de dados que ofereça uma forma de organização ordenada dos dados, na verdade estamos nos referindo a uma árvore binária de busca, que, por motivo de simplificação, neste livro estaremos denominando apenas árvore binária.

Para facilitar a exemplificação dos conceitos, neste livro optamos por utilizar apenas o campo *chave* para caracterizar o elemento que será armazenado na árvore, desconsiderando a presença de dados secundários do elemento. Como consequência, temos que no campo *dado* (do registro que caracteriza um nó da árvore) vamos armazenar um valor que será também a própria chave de ordenação.

Consideremos o exemplo da Figura 11.15, no qual desejamos inserir a chave (valor) 22 na árvore binária apresentada. Notemos que, na localização do ponto para inserção de 22, primeiro vemos que 22 é menor que 50 e, a seguir, que 22 é menor que 40 (subárvore à esquerda); por fim, vemos que 22 é maior que 15 (subárvore à direita). Como 15 não apresenta subárvore à direita, 22 é inserido como filho à direita de 15.

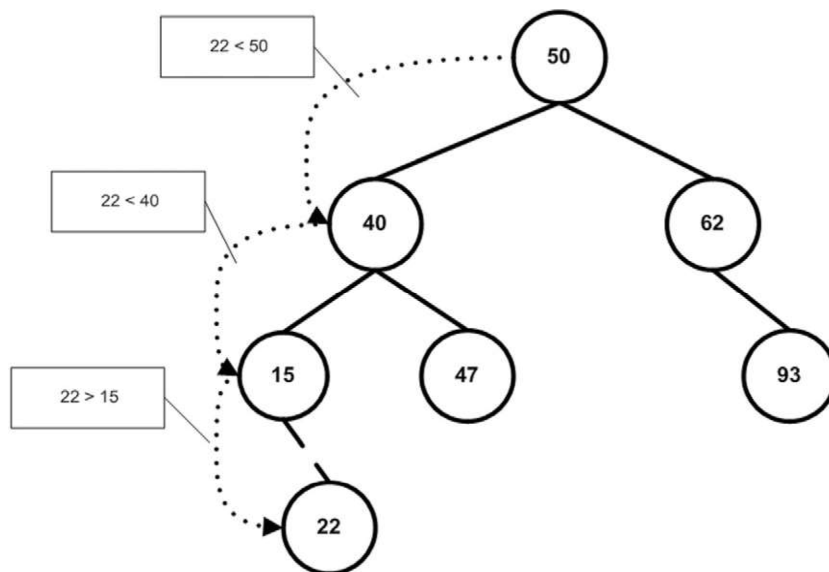


FIGURA 11.15: Inserção na árvore binária.

A seguir, tem-se a implementação da função *inserir_binaria*, que insere a chave *valor* na árvore binária.

Código 11.5

```
void inserir_binaria(no **Raiz, tipo_dado valor)
{
  no *E;

  if (*Raiz == NULL)                          // Cria nó para inserção
  {
     E = (no *)malloc(sizeof(no));            // Aloca espaço na memória correspondente ao nó E
     E->dado = valor;                         // insere o conteúdo (chave) do nó E
     E->esq = NULL;                           // inicializa a subárvore esquerda com vazio
     E->dir = NULL;                           // inicializa a subárvore direita com vazio
     *Raiz = E;
  }
  else if (valor < (*Raiz)->dado)
          inserir_binaria(&(*Raiz)->esq,valor);
      else if (valor > (*Raiz)->dado)
              inserir_binaria(&(*Raiz)->dir,valor);
           else
              printf("Elemento ja existente\n");
}
```

Ao observarmos a implementação da função *inserir_binaria*, podemos notar que se trata de um *procedimento recursivo*, em que a posição de inserção é procurada baseando-se na organização já descrita, e em que as chaves menores ficam sempre à esquerda e as maiores à direita de determinado nó da árvore (subárvore). Dessa forma, se a chave procurada for menor que a existente no nó investigado, é feita uma chamada recursiva de *inserir_binaria*, em que, como raiz, é passado o nó raiz da subárvore à esquerda. De forma análoga, se a chave for maior, a chamada ocorre para a subárvore à direita.

Remoção numa árvore binária

Para apresentar a operação de remoção numa árvore binária, adotaremos o mesmo padrão verificado na inserção, ou seja, nós filhos à esquerda contêm chaves menores do que a encontrada no nó pai, consequentemente, nós filhos à direita contêm chaves maiores que a existente no nó pai.

Diferentemente da inserção, a remoção traz dificuldades adicionais para sua operacionalização. Quando removemos um elemento, para que a organização prévia seja mantida, torna-se necessária a reorganização da árvore binária.

Dependendo da posição do elemento removido, diferentes ações podem ser necessárias. De forma geral, há três situações possíveis quanto ao nó em que se encontra o elemento a ser removido:

- **nó não apresenta subárvore à esquerda**: basta fazer com que o nó filho à direita passe a ser o nó pai;
- **nó não apresenta subárvore à direita**: basta fazer com que o nó filho à esquerda passe a ser o nó pai;
- **nó apresenta subárvores à esquerda e à direita**: deve ser deslocado, para a posição em que se encontra o nó Bi a ser removido, o nó com o elemento de maior chave da subárvore à esquerda de Bi.*

Para exemplificar, observemos os exemplos da Figura 11.16. Na parte (a) temos a remoção da chave (valor) 60 correspondendo à situação (1), descrita anteriormente. Na parte (b), a remoção da chave 40 corresponde à situação (2) e, por fim, na parte (c), a remoção da chave 38 corresponde à situação (3).

A seguir são apresentadas as funções que permitem a remoção de um elemento da árvore binária. A função *remover_binaria* é a responsável por localizar e remover o elemento desejado. Porém, na situação descrita, quando o elemento a ser removido está num nó com subárvores à direita e à esquerda, torna-se necessário o uso da função *retornar_maior*, que tem como finalidade devolver o elemento de maior chave da subárvore esquerda, que ficará no lugar do elemento removido.

*Alguns autores adotam para a situação 3 o deslocamento, para a posição em que se encontra o nó Bi a ser removido, do nó com o elemento de menor chave da subárvore à direita de Bi.

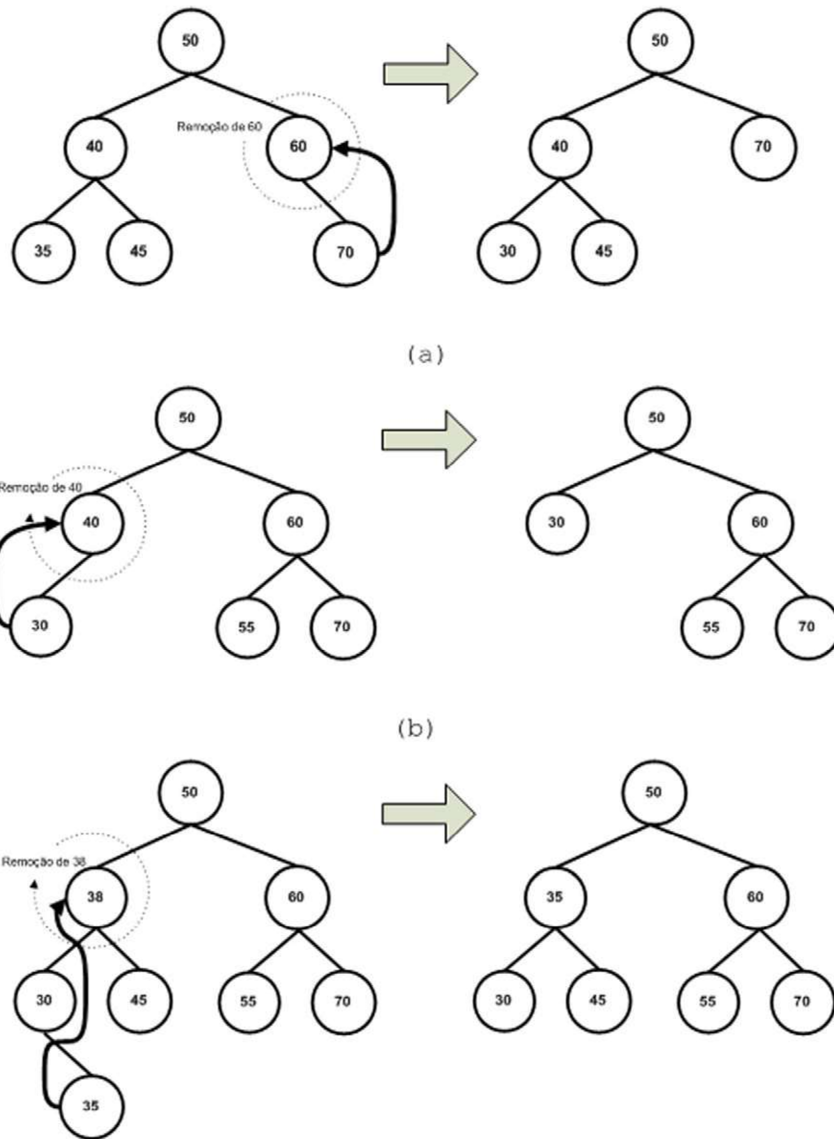


FIGURA 11.16: Remoções de elementos em árvores binárias.

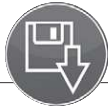
Código 11.6

```
tipo_dado retornar_maior(no **X)
{
    tipo_dado valor;
    no *Aux;

    if ((*X)->dir != NULL)
        return(retornar_maior(&(*X)->dir)); // Continua procurando pelo maior elemento (chave) da subárvore
    else {
         Aux = *X;
         valor = (*X)->dado; // Retorna o maior elemento (chave) da subárvore analisada
         *X = (*X)->esq;      // Filho à esquerda passa a ser o pai, uma vez que o seu pai deixou de existir por ter sido
                              // deslocado para o lugar do nó que foi removido
         free(Aux);           // Libera o espaço alocado na memória para o elemento deslocado
         return(valor);
        }
}

void remover_binaria(no **Raiz, tipo_dado valor)
{
  no *Aux;
  if (*Raiz != NULL)
  {
     if ((*Raiz)->dado == valor)
     {
         Aux = *Raiz;
         if ((*Raiz)->esq == NULL)    // Se a subárvore esquerda é vazia
            {
              *Raiz =(*Raiz)->dir;    // Transforma o filho à direita em nó pai
              free(Aux);              // Libera o espaço alocado na memória para o elemento removido
            }
         else if ((*Raiz)->dir == NULL)          // Se a subárvore direita é vazia
                    {
                      *Raiz=((*Raiz)->esq); // Transforma o filho à esquerda em nó pai
                      free(Aux);            // Libera o espaço alocado na memória para o elemento removido
                    }
// Caso o nó a ser removido tenha subárvores à direita e esquerda, então coloca ali o maior elemento (chave) da subárvore à esquerda
              else (*Raiz)->dado = retornar_maior(&(*Raiz)->esq);
      }
      else if ((*Raiz)->dado < valor)                 // elemento (chave) procurado está na subárvore à direita
               remover_binaria(&(*Raiz)->dir, valor);
           else if ((*Raiz)->dado > valor)             // elemento (chave) procurado está na subárvore à esquerda
                    remover_binaria(&(*Raiz)->esq, valor);
  }
  else printf ("Elemento inexistente!\n");          //elemento (chave) procurado não faz parte da Árvore Binária
}
```

Eficiência na busca numa árvore binária

As operações básicas numa árvore binária (de busca) tem tempo proporcional à sua altura. Como a altura da árvore dependerá da quantidade *N* de chaves e de sua ordem de inserção na árvore, o tempo de resposta das operações básicas dependerá da quantidade e da distribuição das chaves pelas subárvores (subárvores com diferentes alturas). No pior caso em termos de altura da árvore binária, teríamos as inserções das chaves de forma que a altura da árvore fosse N-1, ou seja, um tempo de execução das operações básicas O(**N**).*

Numa situação ideal de distribuição de *N* chaves pelas subárvores (árvore binária completa), porém, as operações básicas, no pior caso, seriam executadas num tempo $\mathbf{O(\log_2 N)}$. Para aprofundamento sobre a eficiência de uma árvore binária, recomendamos a leitura do material mencionado na seção “Para Saber Mais” deste capítulo.



Vamos programar

Para ampliar a abrangência do conteúdo apresentado, teremos, nas seções seguintes, implementações nas linguagens de programação Java e Phyton.

Java

Código 11.1

```
public class ArvoreBinaria
{

    private No raiz;
    private ArvoreBinaria arvoreEsquerda;
    private ArvoreBinaria arvoreDireita;

    public ArvoreBinaria() { }

    public ArvoreBinaria getArvoreDireita() {
        return arvoreDireita;
    }

    public void setArvoreDireita(ArvoreBinaria arvoreDireita) {
        this.arvoreDireita = arvoreDireita;
    }

    public ArvoreBinaria getArvoreEsquerda() {
        return arvoreEsquerda;
    }

    public void setArvoreEsquerda(ArvoreBinaria arvoreEsquerda) {
        this.arvoreEsquerda = arvoreEsquerda;
    }

    public No getRaiz() {
        return raiz;
    }

    public void setRaiz(No raiz) {
        this.raiz = raiz;
    }
    public class No {
        private Dado dado;

        public No(Dado dado) {
            this.dado = dado;
        }

        public Dado getDado() {
            return dado;
        }

        public void setDado(Dado dado) {
            this.dado = dado;
        }
    }

    public class Dado {
        private int valor;

        public Dado(int valor) {
            this.valor = valor;
        }

        public int getValor() {
            return valor;
        }

        public void setValor(int valor) {
            this.valor = valor;
        }
    }
```

*Em termos de eficiência em algoritmos, a notação O(..) representa a ordem associada ao que se deseja mensurar. Assim, O(N) representa uma ordem linear e O(log) uma ordem logarítmica.

Código 11.2

```
public void pre_ordem() {
    if (this.raiz == null) {
       return;
    }

    System.out.println("Valor: " + this.raiz.getDado().getValor());

    if (this.arvoreEsquerda != null) {
        this.arvoreEsquerda.pre_ordem();
    }

    if (this.arvoreDireita != null) {
        this.arvoreDireita. pre_ordem();
    }
}
```

Código 11.3

```
public void em_ordem() {
    if (this.raiz == null) {
       return;
    }

    if (this.arvoreEsquerda != null) {
        this.arvoreEsquerda. em_ordem();
    }

    System.out.println("Valor: " + this.raiz.getDado().getValor());

    if (this.arvoreDireita != null) {
        this.arvoreDireita. em_ordem();
    }
}

}
```

Código 11.4

```
public void pos_ordem() {
    if (this.raiz == null) {
       return;
    }

    if (this.arvoreEsquerda != null) {
        this.arvoreEsquerda. pos_ordem();
    }

    if (this.arvoreDireita != null) {
        this.arvoreDireita. pos_ordem();
    }
    System.out.println("Valor: " + this.raiz.getDado().getValor());
}
```

Código 11.5

```
public void inserir_binaria(int valor) {
    Dado dado = new Dado(valor);
    No no = new No(dado);
    inserir(no);

public void inserir(No no) {
    if (this.raiz == null) {
        this.raiz = no;
    } else {
            if (no.getDado().getValor() < this.raiz.getDado().getValor()) {
             if (this.arvoreEsquerda == null) {
                this.arvoreEsquerda = new ArvoreBinaria();
             }
                this.arvoreEsquerda.inserir(no);
            }
            else if (no.getDado().getValor() > this.raiz.getDado().getValor()) {
                  if (this.arvoreDireita == null) {
                     this.arvoreDireita = new ArvoreBinaria();
                  }
                     this.arvoreDireita.inserir(no);
                 } else System.out.println("Elemento ja existente");

      }
}
```

Código 11.6

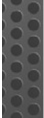
```
    public static ArvoreBinaria remover_binaria(ArvoreBinaria aux, int num) {
        ArvoreBinaria p, p2;
        if (aux.raiz.getDado().getValor() == num) {
            if (aux.arvoreEsquerda == aux.arvoreDireita) {
                return null;
            } else if (aux.arvoreEsquerda == null) {
                return aux.arvoreDireita;
            } else if (aux.arvoreDireita == null) {
                return aux.arvoreEsquerda;
            } else {
                p2 = aux.arvoreDireita;
                p = aux.arvoreDireita;
                while (p.arvoreEsquerda != null) {
                    p = p.arvoreEsquerda;
                }
                p.arvoreEsquerda = aux.arvoreEsquerda;
                return p2;
            }
        } else if (aux.raiz.getDado().getValor() < num) {
            aux.arvoreDireita = remover_binaria(aux.arvoreDireita, num);
        } else {
            aux.arvoreEsquerda = remover_binaria(aux.arvoreEsquerda, num);
        }
        return aux;
    }
```

```
}
```

Phyton

Código 11.1

```
class arvoreBinaria(object):
    def __init__(self, value=None, esq = None, dir = None):
        self.value = value
        self.esq = esq
        self.dir = dir
```



Código 11.2

```
def pre_ordem(self):
    if self.value <> None:
        print self.value
        if self.esq <>None:
           self.esq.pre_ordem()
        if self.dir <>None:
           self.dir.pre_ordem()
```

Código 11.3

```
def em_ordem(self):
    if self.value <> None:
        if self.esq <>None:
           self.esq.em_ordem()
        print self.value
        if self.dir <>None:
           self.dir.em_ordem()
```

Código 11.4

```
def pos_ordem(self):
      if self.value <> None:
          if self.esq <>None:
             self.esq.pos_ordem()
          if self.dir <>None:
             self.dir.pos_ordem()
          print self.value
```

Código 11.5

```
def inserir_binaria(self, item):
    if self.value == item:
       return
    else:
        if item < self.value:
            if self.esq != None:
                self.esq.inserir_binaria(item)
            else:
                self.esq = arvoreBinaria(item)
        else:
            if self.dir != None:
                self.dir.inserir_binaria(item)
            else:
                self.dir = arvoreBinaria(item)
```

Código 11.6

```
def remover_binaria(self, aux, num):
  p = arvoreBinaria()
  p2 = arvoreBinaria()
  if aux.value==num:
    if aux.esq==aux.dir:
      return None
    elif aux.esq==None:
            return aux.dir
    elif aux.dir==None:
       return aux.esq
    else:
      p2=aux.dir
      p=aux.dir
      while p.esq<>None:
         p=p.esq
  elif aux.value < num:
     aux.dir = remover_binaria(aux.dir,num)
  else:
    aux.esq = remover_binaria(aux.esq,num)
  return aux
```

Para fixar!

1. Considerando o que você estudou, pense numa situação do seu cotidiano em que a organização hierárquica em árvore possa ser usada. Procure estruturar os elementos que compõem a situação.
2. Usando as funções definidas neste capítulo, crie uma árvore binária contendo os valores (chaves) 50, 40, 15, 62, 93, 47, 35, 68, 10, 37 e 22. Apresente os percursos pré-ordem, em-ordem e pós-ordem para essa árvore. Tente remover o valor 100 (inexistente). Depois, remova os valores 93 e 40. Repita a apresentação dos percursos pré-ordem, em-ordem e pós-ordem para a nova árvore (após a remoção).
3. Procure na internet simuladores de árvores binárias e utilize-os com os conjuntos de dados que criou para o exercício 1. Depois, compare os resultados encontrados com os que você obteve usando as funções aqui apresentadas.

Para saber mais...

Dentro do propósito geral deste livro, que é apresentar de forma clara e numa linguagem acessível uma visão geral básica das principais estruturas de dados existentes, orientamos aqueles que desejam se aprofundar nos estudos sobre árvores binárias que leiam a Parte III (Capítulos 12 e 13) do livro *Algoritmos: teoria e prática* (Wirth, 1989).

Navegar é preciso

Existem diversas ferramentas disponíveis na internet que podem contribuir com o seu aprendizado sobre árvores binárias. Dentre elas citamos:

- ASTRAL (http://www.ic.unicamp.br/~rezende/Astral.htm);

Relaciona um conjunto de ferramentas executáveis gratuitas que permitem investigar, entre outras coisas, o funcionamento das árvores binárias.

- TBC_AED (http://algol.dcc.ufla.br/~heitor/Projetos/TBC_AED_GRAFOS_WEB/TBC_AED_GRAFOS_WEB.html);

Oferece ferramentas WEB para o aprendizado de estrutura de dados, dentre as quais uma para árvore binária.

- SIMULED (http://simuledufg.sourceforge.net/);

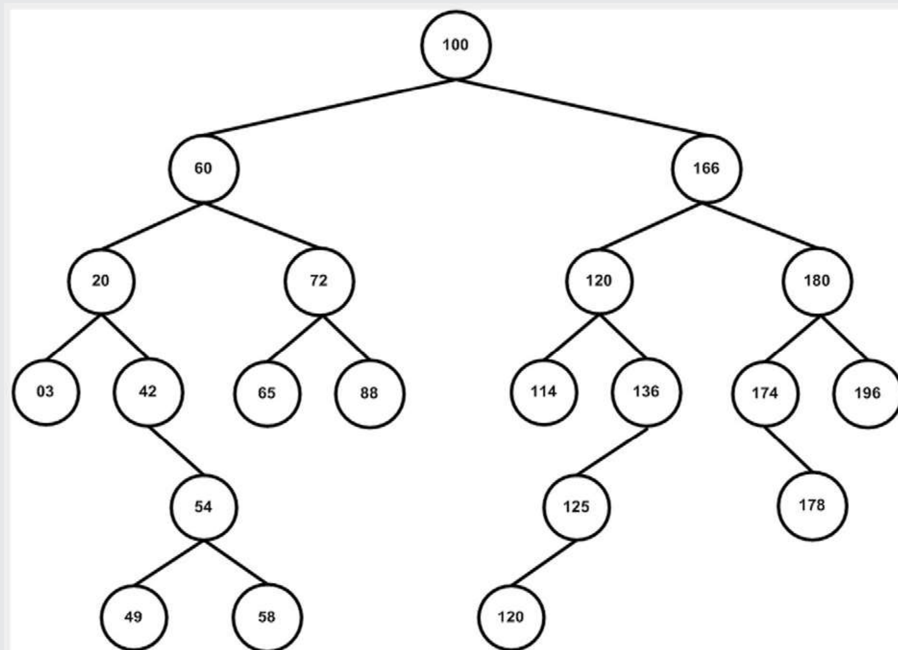
Ferramenta com código em Java (GNU GPL) que permite a execução das operações mais comuns com estruturas de dados.

- Outras ferramentas em forma de Applets
  (http://www.cs.umd.edu/~egolub/Java/BinaryTree.html);
  (http://www.cosc.canterbury.ac.nz/mukundan/dsal/BSTNew.html);
  (http://www.cs.jhu.edu/~goodrich/dsa/trees/btree.html);
  (http://nova.umuc.edu/~jarc/idsv/lesson4.html).

Exercícios

1. Crie uma função de percurso em que os elementos da árvore da Figura 11.10 sejam apresentados por nível, ou seja, em que a ordem de apresentação seja B0-B1-B2-B3-B4-B5-B6-B7-B8-B9-B10-B11-B12-B13-B14.

2. Para a árvore binária a seguir, responda:



a. A árvore foi construída corretamente?
b. Que nós pertencem ao nível 3 da árvore?
c. Qual é a altura do nó que contém o valor 120?
d. Como ficaria a árvore com a inserção dos valores 59, 61 e 128?
e. Como ficaria a árvore com a remoção dos valores 42, 166 e 174?

3. Crie uma função que devolva o nível de determinado nó que contenha o valor passado para a função.
4. Crie uma função que devolva a soma dos valores presentes nos *nós folhas* de uma árvore binária.

Conseguiu? Parabéns, você já domina os conceitos fundamentais da estrutura de dados árvore!!!

Referências bibliográficas

CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.

TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.

WIRTH, N. *Algoritmos e estruturas de dados*. Rio de Janeiro: Prentice-Hall, 1989.

O que vem depois

Agora que você já conhece os conceitos fundamentais de árvores e, principalmente, de árvores binárias, vamos generalizar essas ideias considerando que cada nó poderia ter um número de filhos maior que os dois vistos nas árvores binárias. Será que teremos mudanças significativas? É o que você descobrirá no Capítulo 12!