Capítulo 14 – Árvores de Busca

A busca, que foi discutida ao longo do texto, é uma operação muito comum e tem sido extensivamente estudada. Uma pesquisa linear de um array ou lista Python é muito lenta, mas pode ser melhorada com uma pesquisa binária. Mesmo com o tempo de pesquisa aprimorado, arrays e listas Python apresentam uma desvantagem quando se trata de inserção e exclusão de chaves de pesquisa. Lembre-se de que uma pesquisa binária só pode ser realizada em uma sequência ordenada. Quando chaves são adicionadas ou removidas de um array ou lista Python, a ordem deve ser mantida. Isto pode ser demorado, pois as chaves precisam ser deslocadas para liberar espaço ao adicionar uma nova chave ou para fechar a lacuna ao excluir uma chave existente. O uso de uma lista encadeada proporciona inserções e exclusões mais rápidas sem a necessidade de mudar as chaves existentes. Infelizmente, o único tipo de pesquisa que pode ser realizada em uma lista encadeada é uma pesquisa linear, mesmo que a lista esteja ordenada. Neste capítulo, exploramos algumas das muitas maneiras pelas quais a estrutura em árvore pode ser usada na realização de pesquisas eficientes.

A estrutura em árvore, apresentada no capítulo 13 (anterior), pode ser usada para organizar dados dinâmicos de forma hierárquica. As árvores vêm em vários formatos e tamanhos dependendo de sua aplicação e do relacionamento entre os nós. Quando usado para pesquisa, cada nó contém uma chave de pesquisa como parte de sua entrada de dados (às vezes chamada de **carga útil**, ou **payload**) e os nós são organizados com base no relacionamento entre as chaves. Existem muitos tipos diferentes de árvores de pesquisa, algumas das quais são simplesmente variações de outras e outras que podem ser usadas para pesquisar dados armazenados externamente. Mas o objetivo principal de todas as árvores de busca é fornecer uma operação de busca eficiente para localizar rapidamente um item específico contido na árvore.

As árvores de busca podem ser usadas para implementar muitos tipos diferentes de contêineres, alguns dos quais podem precisar apenas armazenar as chaves de pesquisa dentro de cada nó da árvore. Mais comumente, entretanto, os aplicativos associam dados ou uma carga útil a cada chave de pesquisa e usam a estrutura da mesma maneira que um TAD Map seria usado. O TAD Map foi apresentado no Capítulo 3, quando o implementamos usando uma estrutura de lista. Os exercícios em vários capítulos ofereceram a oportunidade de fornecer novas implementações utilizando diversas estruturas de dados.

No Capítulo 11, implementamos uma versão de tabela hash do TAD Map que melhorou os tempos de pesquisa. Mas sua eficiência depende do tipo de chaves armazenadas no mapa, já que a escolha da função hash pode impactar bastante na operação de busca. Ao longo do capítulo, exploramos diversas árvores de busca diferentes, cada uma das quais usaremos para implementar novas versões do TAD Map. Para ajudar a evitar confusão entre as diversas implementações, usamos um nome de classe diferente para cada implementação.

14.1 Árvore de Busca Binária

Uma **árvore de busca binária** (**ABB** ou **BST**, em inglês) é uma árvore binária na qual cada nó contém uma chave de pesquisa dentro de sua carga útil e a árvore é estruturada de tal forma que para cada nó interior V:

- Todas as chaves menores que a chave no nó V são armazenadas na subárvore esquerda de V.
- Todas as chaves maiores que a chave no nó V são armazenadas na subárvore direita de V.

Considere a árvore de busca binária da Figura 14.1, que contém chaves de busca inteiras. O nó raiz contém o valor-chave 60 e todas as chaves na subárvore esquerda da raiz são menores do que 60 e todas as chaves na subárvore direita são maiores do que 60. Se você examinar cada valor-chave nos nós, notará que o mesmo relacionamento de valor-chave se aplica para cada nó da árvore. Dado o relacionamento entre os nós, um percurso em-ordem visitará os nós em ordem crescente de chave de pesquisa. Para a ABB exemplo, a ordem seria (1, 4, 12, 23, 29, 37, 41, 60, 71, 84, 90, 100).

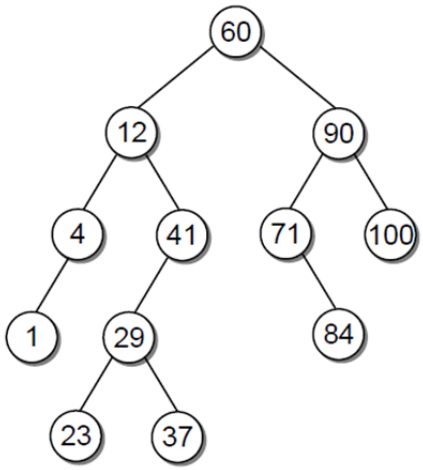


Figura 14.1: Uma árvore de busca binária armazenando chaves inteiras.

Nossa definição de ABB impede o armazenamento de chaves duplicadas na árvore, o que torna a implementação das diversas operações muito mais fácil. Também é apropriado para alguns aplicativos, mas a restrição pode ser alterada para permitir chaves duplicadas, se necessário. Além disso, para fins ilustrativos, mostramos apenas a chave dentro de cada nó de nossas árvores de busca. Você deve assumir que o valor dos dados correspondente também está armazenado nos nós.

Uma implementação parcial da versão da árvore de busca binária do TAD Map é mostrada na Listagem 14.1. O código restante será adicionado à medida que cada operação for discutida ao longo da seção. Como acontece com qualquer árvore binária, uma referência ao nó raiz também deve ser mantida para uma árvore binária de busca. O construtor define o campo **_root** para essa finalidade e também define o campo **_size** para controlar o número de entradas no mapa. O campo **_size** é necessário para o método **__len__**. A definição da classe privada de armazenamento usada para criar os nós da árvore é mostrada nas linhas 16-21.

Listagem 14.1 Implementação parcial do Map ADT usando uma árvore de busca binária.

```
1.    class BSTMap:
2.      # Creates an empty map instance.
3.      def __init__(self):
4.        self._root = None
5.        self._size = 0
6.
7.      # Returns the number of entries in the map.
8.      def __len__(self):
9.        return self._size
10.
11.     # Returns an iterator for traversing the keys in the map.
12.     def __iter__(self):
13.       return _BSTMapIterator(self._root)
14.
```

```
15.   # Storage class for the binary search tree nodes of the map.
16.   class _BSTMapNode:
17.     def __init__(self, key, value):
18.       self.key = key
19.       self.value = value
20.       self.left = None
21.       self.right = None
```

14.1.1 Pesquisando

Dada uma árvore de pesquisa binária, você eventualmente desejará pesquisar a árvore para determinar se ela contém um determinado valor-chave ou para localizar um item específico. No último capítulo, vimos que existe um único caminho da raiz para todos os outros nós de uma árvore. Se a árvore de busca binária contiver a chave de destino, haverá um caminho exclusivo da raiz até o nó que contém essa chave. A única questão é: como sabemos que caminho seguir?

Como o nó raiz fornece o único ponto de acesso a qualquer árvore binária, nossa busca deve começar nele. O valor alvo é comparado à chave no nó raiz conforme ilustrado na Figura 14.2. Se a raiz contiver o valor alvo, nossa pesquisa terminará com um resultado bem-sucedido. Mas se o alvo não estiver na raiz, devemos decidir qual dos dois caminhos seguir.

A partir da definição da árvore de busca binária, sabemos que a chave no nó raiz é maior que as chaves na sua subárvore esquerda e menor que as chaves na sua subárvore direita. Assim, se o alvo for menor que a chave da raiz, movemo-nos para a esquerda e movemos para a direita se for maior. Repetimos a comparação no nó raiz da subárvore e seguimos o caminho apropriado. Este processo é repetido até que o alvo seja localizado ou encontremos um link filho nulo.

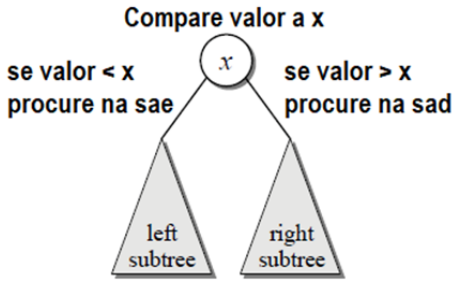


Figura 14.2: A estrutura de uma ABB é baseada nas chaves de busca.

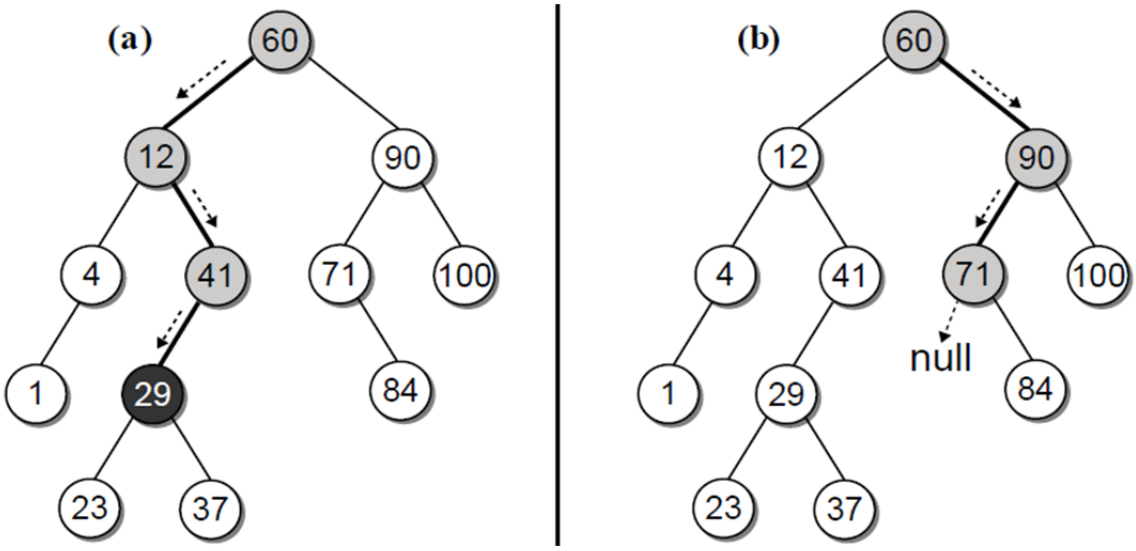


Figura 14.3: Pesquisando uma árvore de pesquisa binária: (a) pesquisa bem-sucedida por 29 e (b) pesquisa malsucedida por 68.

Suponha que queiramos pesquisar o valor-chave 29 na ABB da Figura 14.1. Começamos comparando o valor do alvo com a raiz 60. Como o alvo é menor que 60, movemos para a esquerda. O alvo é então comparado com 12. Desta vez movemo-nos para a direita, uma vez que o alvo é maior que 12. Em seguida, o alvo é comparado com 41, resultando em um movimento para a esquerda. Finalmente, quando examinamos o filho esquerdo do nó 41, encontramos o alvo e relatamos uma busca bem-sucedida. O caminho percorrido para encontrar a chave 29 na árvore da Figura 14.1 é ilustrado na Figura 14.3(a) pelas linhas tracejadas direcionadas.

E se o alvo não estiver na árvore? Por exemplo, suponha que queiramos procurar a chave 68. Repetiríamos o mesmo processo usado para encontrar a chave 29, conforme ilustrado na Figura 14.3(b). A diferença é o que acontece quando alcançamos o nó 71 e o comparamos com o alvo. Se 68 estivesse na árvore de busca binária, teria que estar na subárvore esquerda do nó 71. Mas você notará que o nó 71 não tem um filho esquerdo. Se continuarmos nessa direção, "cairemos" da árvore. Assim, atingir um link filho nulo durante a busca por uma chave de destino indica uma busca malsucedida.

As operações da árvore de busca binária podem ser implementadas iterativamente ou com o uso de recursão. Implementamos funções recursivas para cada operação e deixamos as versões iterativas como exercícios. O método auxiliar **`_bstSearch()`**, fornecido nas linhas 14-22 da Listagem 14.2, navega recursivamente em uma ABB para encontrar o nó que contém a chave alvo. O método tem dois casos básicos: o alvo está contido no nó atual ou um link filho nulo é encontrado. Quando um caso base é alcançado, o método retorna uma referência ao nó que contém a chave ou **`None`**, de volta a todas as chamadas recursivas. Este último indica que a chave não foi encontrada na árvore. A chamada recursiva é feita passando o link para a subárvore esquerda ou direita, dependendo do relacionamento entre o alvo e a chave no nó atual.

Listagem 14.2 Procurando por uma chave alvo em uma ABB.

```
1.    class BSTMap:
2.      # ...
3.      # Determines if the map contains the given key.
4.      def __contains__(self, key):
5.        return self._bstSearch(self._root, key) is not None
6.
7.      # Returns the value associated with the key.
8.      def valueOf(self, key):
9.        node = self._bstSearch(self._root, key)
10.       assert node is not None, "Invalid map key."
11.       return node.value
12.
13.     # Helper method that recursively searches the tree for a target key.
14.     def _bstSearch(self, subtree, target):
15.       if subtree is None:         # base case
16.         return None
17.       elif target < subtree.key:  # target is left of the subtree root.
18.         return self._bstSearch(subtree.left)
19.       elif target > subtree.key:  # target is right of the subtree root.
20.         return self._bstSearch(subtree.right)
21.       else:                       # base case
22.         return subtree
```

Você pode está se perguntando por que retornamos um link ao nó e não apenas um valor booleano indicando o sucesso ou o fracasso da busca. Isso nos permite usar o mesmo método auxiliar para implementar os métodos **`__contains__`** e **`valueOf()`** da classe **`Map`**. Ambos chamam o método auxiliar recursivo para localizar o nó que contém a chave alvo. Ao fazer isso, a referência do nó raiz deve ser passada ao auxiliar para iniciar a recursão. O valor retornado de **`_bstSearch()`** pode ser avaliado para determinar se a chave foi encontrada na árvore e a ação apropriada pode ser tomada para a operação do TAD Map correspondente.

Uma árvore de busca binária pode está vazia, conforme indicado por uma referência de raiz nula, portanto, devemos garantir que qualquer operação executada na árvore também funcione quando a árvore estiver vazia. No método **`_bstSearch()`**, isso é tratado pelo primeiro pelo caso base na primeira chamada ao método.

14.1.2 Valores Mínimos e Máximos

Outra operação semelhante a uma pesquisa que pode ser realizada em uma árvore de busca binária é encontrar os valores mínimos ou máximos entre as chaves da árvore. Dada a definição da árvore de busca binária, sabemos que o valor mínimo está na raiz ou em um nó à sua esquerda. Mas como sabemos se a raiz é o menor valor e não está em algum lugar da subárvore esquerda? Poderíamos comparar a raiz com seu filho esquerdo, mas se você pensar bem, não há necessidade de comparar as chaves individuais. O motivo tem a ver com o relacionamento entre as chaves. Se o nó raiz contiver chaves em sua subárvore esquerda, então a raiz não poderá conter o valor-chave mínimo, pois todas as chaves à esquerda da raiz são menores que a raiz. E se o nó raiz não tiver um filho esquerdo? Nesse caso, a raiz conteria o menor valor-chave, pois todas as chaves à direita são maiores que a raiz.

Se aplicarmos a mesma lógica ao filho esquerdo do nó raiz (assumindo que ele tenha um filho esquerdo) e depois ao filho esquerdo desse nó e assim por diante, eventualmente encontraremos o valor-chave mínimo. Esse valor será encontrado em um nó que seja uma folha ou um nó interior sem filho esquerdo. Ele pode ser localizado começando na raiz e seguindo os links filhos esquerdos até que um link nulo seja encontrado, conforme ilustrado na Figura 14.4. O valor máximo da chave pode ser encontrado de maneira semelhante.

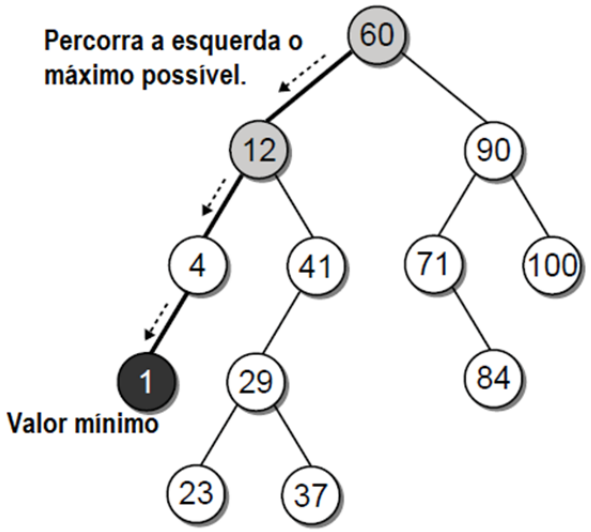


Figura 14.4: Encontrando a chave mínima ou máxima em uma árvore binária de busca.

A Listagem 14.3 fornece um método auxiliar recursivo para localizar o nó que contém o valor mínimo da chave na árvore de pesquisa binária. O método requer a raiz da árvore ou de uma subárvore como argumento. Ele retorna uma referência ao nó que contém o menor valor de chave ou None quando a árvore está vazia.

Listagem 14.3 Encontre o elemento com o valor de chave mínimo em uma árvore de busca binária.

```
1.    class BSTMap:
2.      # ...
3.      # Helper method for finding the node containing the minimum key.
4.      def _bstMinumum(self, subtree):
5.        if subtree is None:
6.          return None
7.        elif subtree.left is None:
8.          return subtree
9.        else:
10.         return self._bstMinimum(subtree.left)
```

14.1.3 Inserção

Quando uma árvore de pesquisa binária é construída, as chaves são adicionadas uma de cada vez. À medida que as chaves são inseridas, um novo nó é criado para cada chave e vinculado à sua posição adequada na árvore. Suponha que queiramos construir uma árvore de pesquisa binária a partir da lista de chaves [60, 25, 100, 35, 17, 80] inserindo as chaves na ordem em que estão listadas. A Figura 14.5 ilustra as etapas de construção da árvore, que você pode seguir à medida que descrevemos o processo.

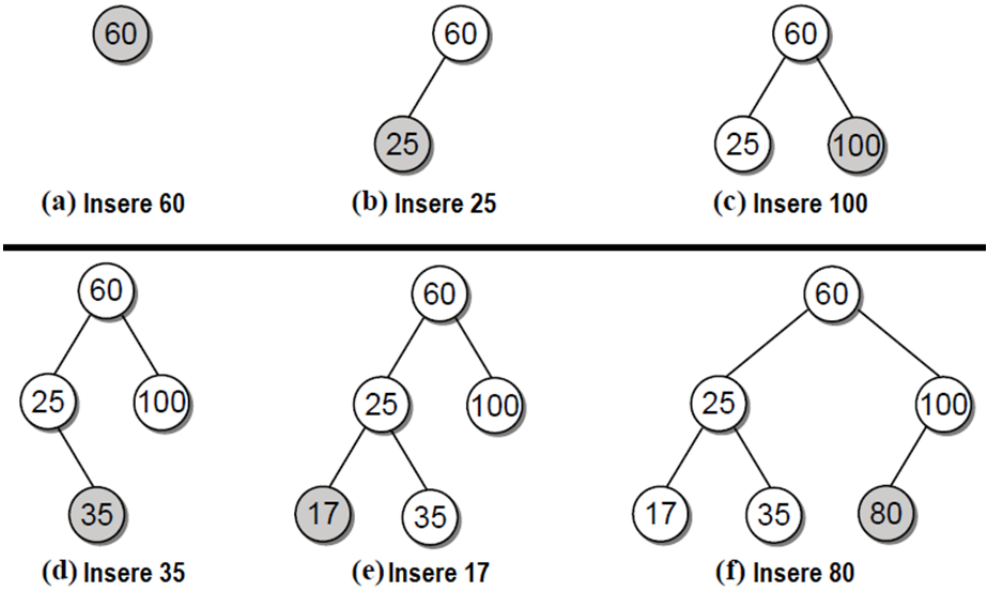


Figura 14.5: Construindo uma árvore binária inserindo as chaves [60, 25, 100, 35, 17, 80].

Começamos inserindo o valor 60. Um nó é criado e seu campo de dados definido com esse valor. Como a árvore está inicialmente vazia, este primeiro nó torna-se a raiz da árvore (a). A seguir inserimos o valor 25. Por ser menor do que 60, deve ser inserido à esquerda da raiz, o que significa que se torna o filho esquerdo da raiz (b). O valor 100 é então inserido em um nó vinculado como filho direito da raiz, pois é maior que 60 (c). O que acontece quando o valor 35 é inserido? A raiz já tem seus filhos esquerdo e direito. Quando novas chaves são inseridas, não modificamos os campos de dados dos nós existentes ou os links entre os nós existentes. Assim, existe apenas um local no qual o valor-chave 35 pode ser inserido em nossa árvore atual e ainda manter a propriedade da árvore de pesquisa. Deve ser inserido como filho direito

do nó 25 (d). Você deve ter notado o padrão que se forma à medida que novos nós são adicionados à árvore binária. Os novos nós são sempre inseridos como um nó folha em sua posição adequada, de modo que a propriedade da árvore de busca binária seja mantida. Concluímos este exemplo inserindo as duas últimas chaves, 35 e 80, na árvore (e) e (f).

Trabalhando neste exemplo manualmente, foi fácil ver onde cada novo nó deveria ser ligado à árvore. Mas como inserimos as novas chaves no código do programa? Suponha que queiramos inserir a chave 30 na árvore que construímos manualmente. O que acontece se usarmos o método **`_bstSearch()`** e procurarmos a chave 30? A busca nos levará ao nó 35 e então cairemos da árvore ao tentar seguir seu link de filho esquerdo, conforme ilustrado na Figura 14.6(a). Observe que este é o local exato onde a nova chave precisa ser inserida.

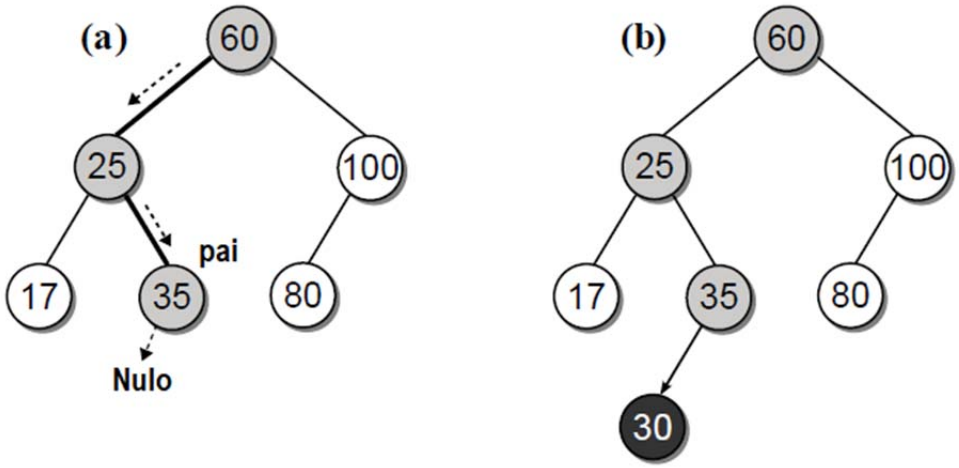


Figura 14.6: Inserindo um novo nó em uma árvore binária de busca: (a) procurando a localização do nó e (b) ligando o novo nó na árvore.

Podemos usar uma versão modificada da operação de busca para inserir novas chaves em uma ABB, conforme mostrado na Listagem 14.4. Para descrever como funciona o método recursivo, suponha que queiramos inserir o valor-chave 30 na árvore que construímos manualmente na Figura 14.5. A Figura 14.7 ilustra a visualização da árvore pelo método em cada invocação e mostra as alterações na árvore à medida que as instruções específicas são executadas.

Listagem 14.4 Inserir uma chave em uma árvore binária.

```
1.    class BSTMap:
2.      # ... Adds a new entry to the map
3.      # or replaces the value of an existing key.
4.      def add(self, key, value):
5.        # Find the node containing the key, if it exists.
6.        node = self._bstSearch(key)
7.        # If the key is already in the tree, update its value.
8.        if node is not None:
9.          node.value = value
10.         return False
11.       # Otherwise, add a new entry.
12.       else:
13.         self._root = self._bstInsert(self._root, key, value)
14.         self._size += 1
15.         return True
16.
```

```
17.     # Helper method that inserts a new item, recursively.
18.     def _bstInsert(self, subtree, key, value):
19.       if subtree is None:
20.         subtree = _BSTMapNode(key, value)
21.       elif key < subtree.key:
22.         subtree.left = self._bstInsert(subtree.left, key, value)
23.       elif key > subtree.key:
24.         subtree.right = self._bstInsert(subtree.right, key, value)
25.       return subtree
```

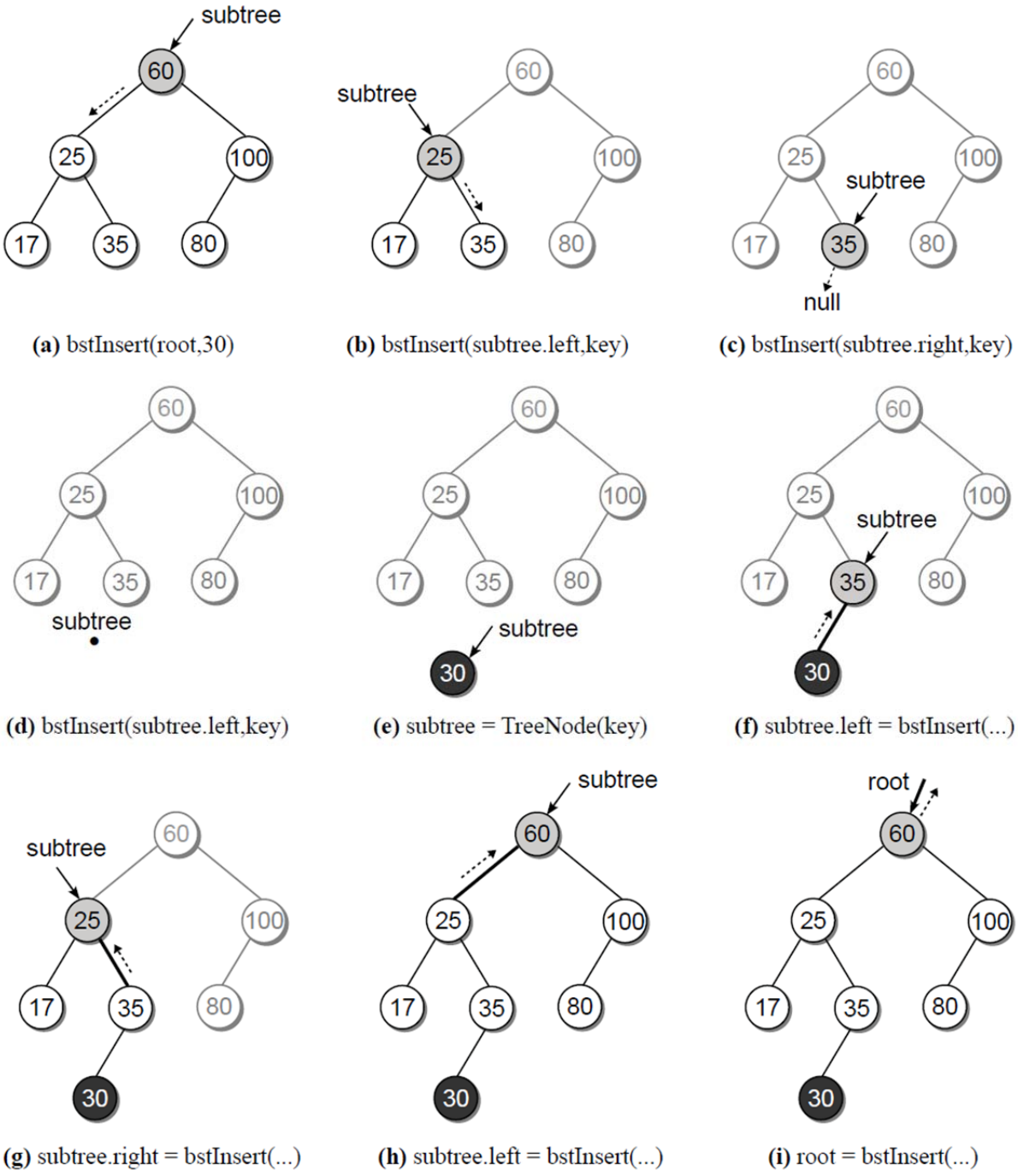


Figura 14.7: As etapas recursivas do método **`_bstInsert()`** ao inserir 30 na ABB. Cada árvore mostra os resultados após executar as instruções indicadas.

Lembre-se, dada a definição recursiva de uma árvore binária, cada nó é ele próprio a raiz de uma subárvore. À medida que o método recursivo **`_bstInsert()`** navega pela árvore, a raiz da subárvore atual será o nó de interesse. Para inserir o valor-chave 30, o método deve procurar sua localização final navegando recursivamente mais profundamente na árvore seguindo a subárvore esquerda ou direita em cada nó, conforme apropriado. Essas etapas recursivas são mostradas nas partes de (a) a (c) da Figura 14.7. Os nós cinza indicam a raiz da subárvore que está sendo processada pela chamada atual do método. As linhas tracejadas indicam a direção que devemos seguir para encontrar o caminho correto através da árvore e depois o caminho seguido durante o desenrolar da recursão.

O caso base é alcançado quando uma subárvore vazia é encontrada após obter o link do filho esquerdo do nó 35, conforme mostrado em (d) da Figura 14.7. Neste ponto, um novo nó é criado e seu campo de dados definido para a nova chave 30, conforme (e). Uma referência a este novo nó é então retornada e a recursão começa a retroceder. A primeira etapa do retrocesso nos leva de volta ao nó 35. A referência retornada pela chamada recursiva é atribuída ao campo **`left`** do nó **`subtree`**, resultando no novo nó sendo vinculado à árvore. À medida que a recursão continua retrocedendo, mostrada nas partes de (f) a (i), a referência de **`subtree`** atual é retornada e vinculada novamente ao seu pai. Isto não altera a estrutura da árvore, pois as mesmas referências estão simplesmente sendo reatribuídas. Isto é necessário dada a forma como ligamos o novo nó ao seu pai e para permitir que o método seja usado com uma árvore inicialmente vazia. Ao inserir a primeira chave na árvore, a referência da raiz será nula. Se chamarmos o método em uma árvore vazia com **`self._bstInsert(self._root, 30)`**, o novo nó será criado dentro do método, mas a referência ao novo nó não será atribuída ao campo **`_root`** e a árvore permanecerá vazia. Como os argumentos do método são passados por valor em Python, temos que retornar a referência e atribuí-la explicitamente ao campo **`_root`**, como é feito no método **`add()`**. Finalmente, após o novo item ser adicionado à árvore, o campo **`_size`** é incrementado em um para refletir essa alteração.

Você deve ter notado que o método de inserção não possui uma cláusula **`else`** no final na instrução condicional, que trataria o caso em que a nova chave fosse igual a uma chave existente. Se uma chave duplicada for encontrada durante a fase de pesquisa, simplesmente retorne a referência da subárvore para interromper a recursão e permitir um retrocesso adequado.

14.1.4 Remoção

Remover um elemento de uma árvore de busca binária é um pouco mais complicado do que procurar um elemento ou inserir um novo elemento na árvore. Uma remoção envolve procurar o nó que contém o valor-chave alvo e, em seguida, desvincular o nó para removê-lo da árvore. Quando um nó é removido, os nós restantes devem preservar a propriedade da árvore de busca. Existem três casos a serem considerados depois que o nó for localizado:

1. O nó é uma folha.
2. O nó possui um único filho.
3. O nó possui dois filhos.

A primeira etapa para remover um elemento é encontrar o nó que contém a chave. Isso pode ser feito de maneira semelhante à usada na busca pelo local para inserir um novo item. Uma vez localizado o nó, ele deve ser desvinculado para removê-lo da árvore. Consideramos os três casos separadamente e, em seguida, fornecemos uma listagem completa do método recursivo **`_bstRemove()`** e seu uso na implementação do método **`remove()`**.

Removendo um Nó Folha

Remover um nó folha é o mais fácil entre os três casos. Suponha que queiramos excluir o valor-chave 23 da árvore de pesquisa binária na Figura 14.1. Após encontrar o nó, ele deve ser desvinculado, o que pode ser feito definindo o campo de filho esquerdo de seu pai (nó 29), como **`None`**, conforme mostrado na Figura 14.8(a).

Remover um nó folha em nosso método recursivo é tão simples quanto retornar uma referência nula. O método **`_bstRemove()`** usa a mesma técnica de retornar uma referência de cada chamada recursiva que a operação de inserção. Ao retornar **`None`** de volta ao nó pai, uma referência nula será atribuída ao campo de link apropriado no pai, desvinculando-o assim da árvore, como mostrado na Figura 14.8(b).

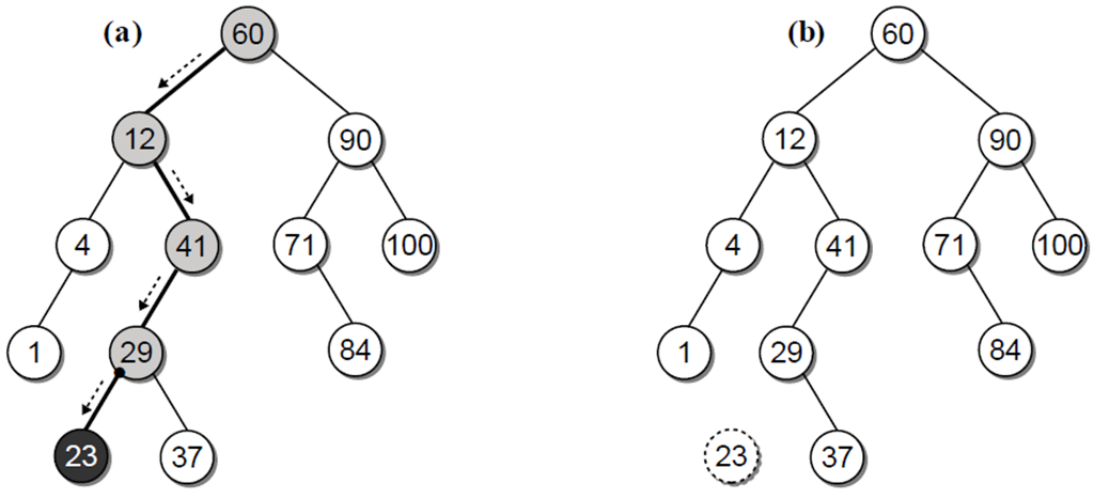


Figura 14.8: Removendo um nó folha de uma árvore binária de busca: (a) encontrar o nó e desvinculá-lo de seu pai; e (b) a árvore após a remoção de 23.

Removendo um Nó Interno com Um Filho

Se o nó a ser removido tiver um único filho, ele poderá ser o filho esquerdo ou direito. Suponha que queiramos excluir o valor-chave 41 da ABB na Figura 14.1. O nó que contém 41 possui uma subárvore vinculada como filho esquerdo. Se simplesmente retornarmos **`None`** ao pai (12), como fizemos para um nó folha, não apenas o nó 41 seria removido, mas também perderíamos todos os seus descendentes, conforme ilustrado na Figura 14.9.

Para remover o nó 41, teremos que fazer algo com seus descendentes. Mas não se preocupe, não precisamos desvincular cada descendente e adicioná-los de volta à árvore. Como o nó 41 contém um único filho, todos os seus descendentes terão chaves menores que 41 ou todos serão maiores. Além disso, dado que o nó 41 é o filho direito do nó 12, todos os descendentes do nó 41 também devem ser maiores que 12. Assim, podemos definir o link do

campo filho direito do nó 12 para referenciar o nó 29, como ilustrado na Figura 14.10. O nó 29 agora se torna o filho direito do nó 12 e todos os descendentes do nó 41 serão vinculados corretamente sem perder nenhum nó.

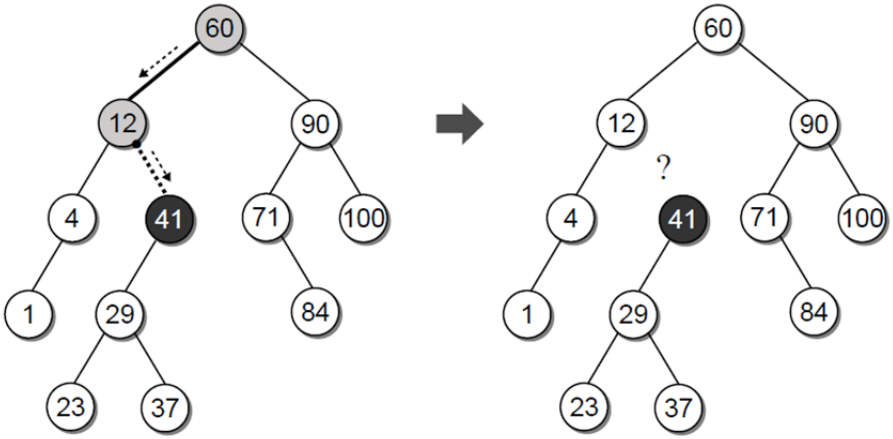


Figura 14.9: Desvinculando incorretamente o nó interior.

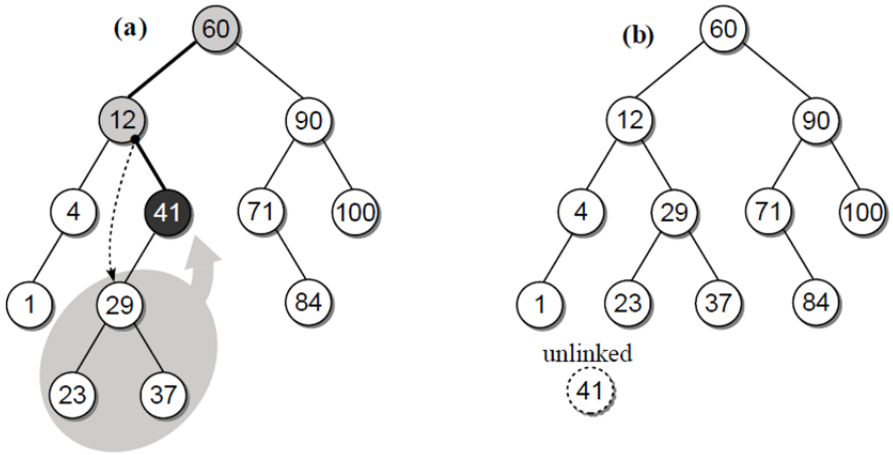


Figura 14.10: Removendo um nó interior (41) com um filho: (a) redirecionando o link do nó pai para sua subárvore filha; e (b) a árvore após a remoção de 41.

Para realizar essa remoção no método recursivo, precisamos apenas alterar o link do campo filho apropriado do pai para fazer referência ao filho do nó que está sendo excluído. A seleção do campo filho do pai a ser alterado é tratada automaticamente pela atribuição realizada no retorno da chamada recursiva. Tudo o que precisamos fazer é retornar o link do filho apropriado no nó que está sendo excluído.

Removendo um Nó Interno com Dois Filhos

O caso mais difícil é quando o nó a ser excluído possui dois filhos. Por exemplo, suponha que queiramos remover o nó 12 da ABB na Figura 14.1. O nó 12 tem dois filhos, ambos raízes de suas próprias subárvores. Se aplicássemos a mesma abordagem usada para remover um nó interior contendo um filho, qual filho escolheríamos para substituir o pai e o que aconteceria com o outro filho e sua subárvore? A Figura 14.11 ilustra o resultado da substituição do nó 12 pelo seu filho direito. Isso deixa a subárvore esquerda desvinculada e,

portanto, removida da árvore. Seria possível vincular o filho esquerdo e sua subárvore como o filho esquerdo do nó 23. Mas isso aumentará a altura da árvore, o que veremos mais tarde, fazendo com que as operações da árvore sejam menos eficientes.

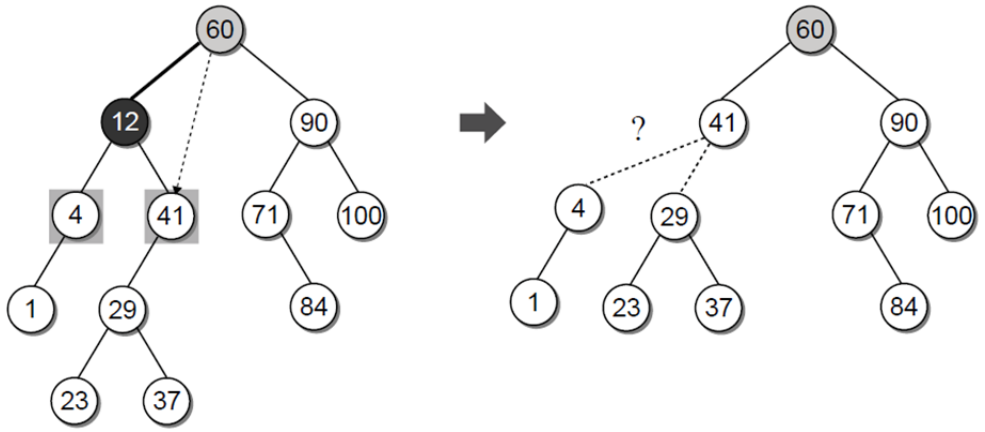


Figura 14.11: Tentativa de remover um nó interno com dois filhos substituindo o nó por um de seus filhos.

As chaves em uma árvore de busca binária são organizadas de forma que uma passagem em-ordem produza uma sequência de chaves ordenada. Assim, cada nó possui um predecessor e um sucessor lógicos. Para o nó 12, seu antecessor é o nó 4 e seu sucessor é o nó 23, conforme ilustrado na Figura 14.12. Em vez de tentar substituir o nó por um de seus dois filhos, podemos substituí-lo por um desses nós, sendo que ambos serão uma folha ou um nó interior com um filho. Como já sabemos como remover uma folha e um nó interior com filho único, aquele selecionado para substituir o nó 12 pode então ser facilmente removido da árvore. A remoção de um nó interno com dois filhos requer três etapas:

1. Encontrar o sucessor lógico, S, do nó a ser excluído, N.
2. Copiar a chave do nó S para o nó N.
3. Remover o nó S da árvore.

As duas últimas etapas são diretas. Depois de encontrarmos o sucessor, podemos simplesmente copiar os dados de um nó para outro. Além disso, como já sabemos como remover um nó folha ou um nó interior com um filho, podemos aplicar o mesmo método para remover o nó original que contém o sucessor.

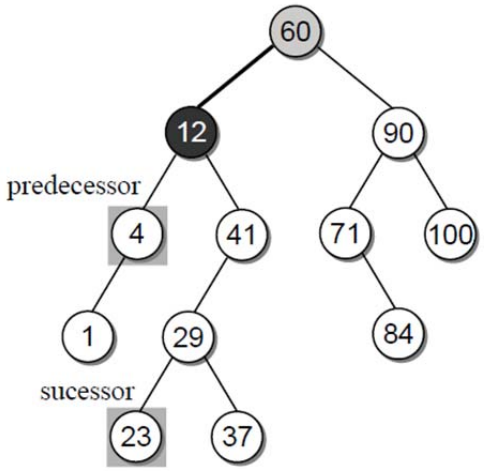


Figura 14.12: O sucessor lógico e o antecessor do nó 12.

Mas como encontramos o sucessor de um nó e onde ele pode estar localizado na árvore? Sabemos que o sucessor é o menor valor de chave dentre aqueles maiores que o nó fornecido. Em nossa árvore de exemplo, seria o nó 23. Com base na definição da árvore de pesquisa binária, a menor chave com valor imadiatamente maior do que um determinado nó é seu pai ou algum nó em sua subárvore direita. Como o nó 12 possui dois filhos, o sucessor estará na sua subárvore direita, o que reduz o conjunto de nós a serem pesquisados. A Figura 14.13 ilustra as etapas quando aplicadas ao nosso exemplo de árvore de pesquisa binária.

Como já sabemos como encontrar a chave mínima em uma árvore de pesquisa binária conforme implementado em **`_bstMinimum()`**, podemos usar este método, mas aplicá-lo à subárvore direita do nó que está sendo excluído. Esta etapa é ilustrada na Figura 14.13(a). Após encontrar o item que contém a chave sucessora, o copiamos para o nó que contém o item que está sendo removido, conforme mostrado na Figura 14.13(b).

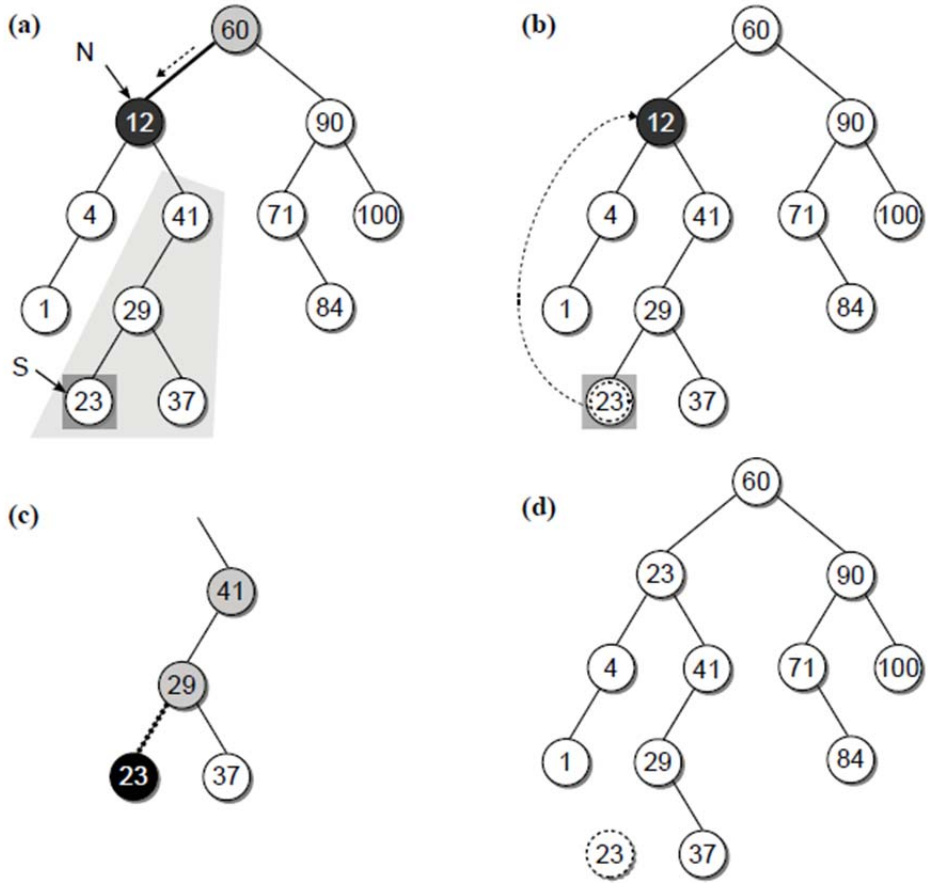


Figura 14.13: Os passos para remover uma chave de uma ABB: (a) encontrar o nó, N, e seu sucessor, S; (b) copiar a chave sucessora do nó N para S; (c) remover a chave sucessora da subárvore direita de N; e (d) a árvore após a remoção do item 12.

Após copiar o item que contém a chave do sucessor, o nó que originalmente contém o sucessor deve ser removido da subárvore direita, conforme mostrado na Figura 14.13(c). Isso pode ser feito chamando o método **`_bstRemove()`** e passando-o para a raiz da subárvore. O resultado da remoção do nó sucessor é ilustrado na Figura 14.13(d).

O método **`_bstRemove()`** é mostrado na Listagem 14.5 junto com a operação **`remove()`** do mapa, que usa o método auxiliar recursivo para remover uma entrada da árvore. O campo **`_size`** é decrementado para refletir a alteração.

Listagem 14.5 Exclua uma chave da árvore binária de busca.

```
1.    class BSTMap:
2.      # ...
3.      # Removes the map entry associated with the given key.
4.      def remove(self, key):
5.        assert key in self, "Invalid map key."
6.        self._root = self._bstRemove(self._root, key)
7.        self._size -= 1
8.
```

```
9.      # Helper method that removes an existing item recursively.
10.     def _bstRemove(self, subtree, target):
11.       # Search for the item in the tree.
12.       if subtree is None:
13.         return subtree
14.       elif target < subtree.key:
15.         subtree.left = self._bstRemove(subtree.left, target)
16.         return subtree
17.       elif target > subtree.key:
18.         subtree.right = self._bstRemove(subtree.right, target)
19.         return subtree
20.       # We found the node containing the item.
21.       else:
22.         if subtree.left is None and subtree.right is None:
23.           return None
24.         elif subtree.left is None or subtree.right is None:
25.           if subtree.left is not None:
26.             return subtree.left
27.           else:
28.             return subtree.right
29.         else
30.           successor = self._bstMinimum(subtree.right)
31.           subtree.key = successor.key
32.           subtree.value = successor.value
33.           subtree.right = self._bstRemove(subtree.right, successor.key)
34.           return subtree
```

14.1.5 Eficiência de Árvores de Busca Binária

As complexidades de tempo para as operações da árvore de busca binária estão listadas na Tabela 14.1. Esta avaliação assume que a árvore contém n nós. Começamos com o método **`_bstSearch()`**. Ao procurar um valor-chave alvo, a função começa no nó raiz e desce na árvore até que a chave seja localizada ou um link nulo seja encontrado. O pior momento para a operação de busca depende do número de nós que devem ser examinados.

No capítulo anterior, vimos que o tempo de pior caso no percurso de uma árvore era linear, uma vez que visitava todos os nós da árvore. Quando um link filho nulo é encontrado, o percurso da árvore retrocede para seguir os outros ramos. Durante uma pesquisa, entretanto, a função nunca retrocede; ele apenas desce na árvore, seguindo uma aresta ou outra em cada nó. Observe que a função recursiva retrocede para retornar ao local do programa onde foi invocada pela primeira vez, mas durante o retrocesso outras ramificações não são examinadas. A pesquisa segue um único caminho da raiz até o nó de destino ou até o nó no qual a pesquisa sai da árvore.

O pior caso ocorre quando o caminho mais longo da árvore é seguido em busca do alvo e o caminho mais longo é baseado na altura da árvore. As árvores binárias vêm em vários formatos e tamanhos e suas alturas podem variar. Mas, como vimos no capítulo anterior, uma árvore de tamanho n pode ter uma altura mínima de aproximadamente log n quando a árvore estiver completa e uma altura máxima de n quando houver um nó por nível. Se não temos conhecimento sobre a forma da árvore e sua altura, temos que assumir o pior e neste caso

seria uma árvore de altura n. Assim, o tempo necessário para encontrar uma chave em uma árvore de busca binária é O(n) no pior caso.

Procurar a chave mínima em uma árvore de busca binária também é uma operação de tempo linear. Mesmo que não compare chaves, ele precisa navegar pela árvore sempre pegando a aresta esquerda começando pela raiz. Na pior das hipóteses, haverá um nó por nível com cada nó ligado ao seu pai através do link filho esquerdo.

O método **`_bstInsert()`**, que implementa o algoritmo para inserir um novo item na árvore, realiza uma pesquisa para descobrir onde a nova chave pertence na árvore. Sabemos pela nossa análise anterior que a operação de busca requer tempo linear no pior caso. Quanto trabalho é realizado após localizar o pai do nó que conterá o novo item? O único trabalho realizado é a criação de um novo nó e o retorno de seu link ao pai, o que pode ser feito em tempo constante. Assim, a operação de inserção requer tempo O(n) no pior caso. O método **`_bstRemove()`** também requer tempo O(n), cuja análise fica como exercício.

Operação	Pior Caso
`_bstSearch(root, k)`	O(n)
`_bstMinimum(root)`	O(n)
`_bstInsert(root, k)`	O(n)
`_bstDelete(root, k)`	O(n)
`traversal`	O(n)

Tabela 14.1: Complexidades de tempo para as operações da árvore de busca binária.

14.2 Iteradores de Árvore de Busca

A definição do TAD SearchTree especifica um iterador que pode ser usado para percorrer as chaves contidas na árvore. A implementação dos iteradores para uso com as estruturas de lista linear foi bastante simples. Para os tipos de sequência, conseguimos inicializar uma variável de índice para acessar os elementos que foram incrementados após cada iteração do loop **`for`**. Com uma lista ligada, o iterador pode definir e usar uma referência externa que é inicializada no nó principal e depois avançada pela lista a cada iteração do loop.

Os percursos podem ser realizados em uma árvore de busca binária, mas isso requer uma solução recursiva. Não podemos avançar facilmente para a próxima chave sem descer na árvore e depois retroceder cada vez que uma folha é encontrada. Uma solução é fazer com que o iterador construa um array de itens percorrendo recursivamente a árvore, que podemos percorrer à medida que o iterador avança, assim como fizemos com as estruturas lineares. Um iterador usando essa abordagem é fornecido na Listagem 14.6.

Embora esta abordagem funcione, ela requer a alocação de espaço de armazenamento adicional, o que pode ser significativo se a árvore contiver um grande número de itens. Como alternativa, podemos realizar um percurso recursivo com o uso de uma pilha. Lembre-se de que a recursão simula o uso de uma pilha sem a necessidade de executar diretamente as operações de empilhar e desempilhar. Qualquer função ou método recursivo pode ser implementado usando uma pilha de software. Para opercurso na árvore, as referências do nó são colocadas na pilha à medida que ele desce na árvore e as referências são exibidas à

medida que o processo retrocede. A Listagem 14.7 mostra a implementação do iterador usando uma pilha de software.

Listagem 14.6 Um iterador para a árvore de busca binária usando um array.

```
class _BSTMapIterator:
  def __init__(self, root, size):
    # Creates the array and fills it with the keys.
    self._theKeys = Array(size)
    self._curItem = 0 # Keep track of the next location in the array.
    self._bstTraversal(root)
    self._curItem = 0 # Reset the current item index.

  def __iter__(self):
    return self

  # Returns the next key from the array of keys
  def __next__(self):
    if self._curItem < len(self._theKeys):
      key = self._theKeys[self._curItem]
      self._curItem += 1
      return key
    else:
      raise StopIteration

  # Performs an inorder traversal used to build the array of keys.
  def _bstTraversal(self, subtree):
    if subtree is not None:
      self._bstTraversal(subtree.left)
      self._theKeys[self._curItem] = subtree.key
      self._curItem += 1
      self._bstTraversal(subtree.right)
```

Listagem 14.7 Um iterador para a árvore de busca binária usando uma pilha de software.

```
class _BSTMapIterator:
  def __init__(self, root):
    # Create a stack for use in traversing the tree.
    self._theStack = Stack()
    # We must traverse down to the node containing the smallest key
    # during which each node along the path is pushed onto the stack.
    self._traverseToMinNode(root)

  def __iter__(self):
    return self

  # Returns the next item from the BST in key order.
  def __next__(self):
    # If the stack is empty, we are done.
    if self._theStack.isEmpty():
      raise StopIteration
    else:
      # The top node on the stack contains the next key.
      node = self._theStack.pop()
      key = node.key
      # If this node has a subtree rooted as the right child, we must
```

```
22.          # find the node in that subtree that contains the smallest key.
23.          # Again, the nodes along the path are pushed onto the stack.
24.          if node.right is not None:
25.            self._traverseToMinNode(node.right)
26.
27.      # Traverses down the subtree to find the node containing the smallest
28.      # key during which the nodes along that path are pushed onto the stack.
29.      def _traverseToMinNode(self, subtree):
30.        if subtree is not None:
31.          self._theStack.push(subtree)
32.          self._traverseToMinNode(subtree.left)
```

EXERCÍCIOS

14.1 Prove ou explique por que o método **`_bstRemove()`** requer tempo O(n) no pior caso.

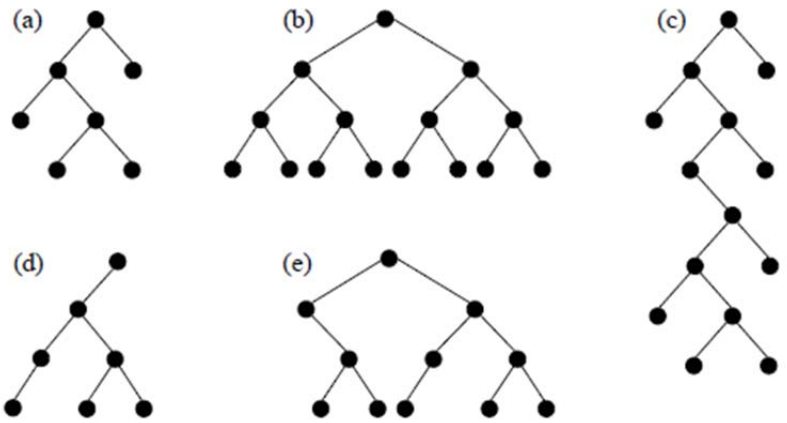
14.2 Por que novas chaves não podem ser inseridas nos nós internos de uma árvore 2-3?

14.3 Considere o seguinte conjunto de valores (30, 63, 2, 89, 16, 24, 19, 52, 27, 9, 4, 45). Use-o para construir o tipo de árvore indicado, adicionando um valor por vez na ordem listada.

a) árvore de busca binária b) árvore AVL c) árvore 2-3

14.4 Repita o Exercício 14.3, mas para as chaves (T, I, P, A, F, W, Q, X, E, N, S, B, Z) nesta ordem.

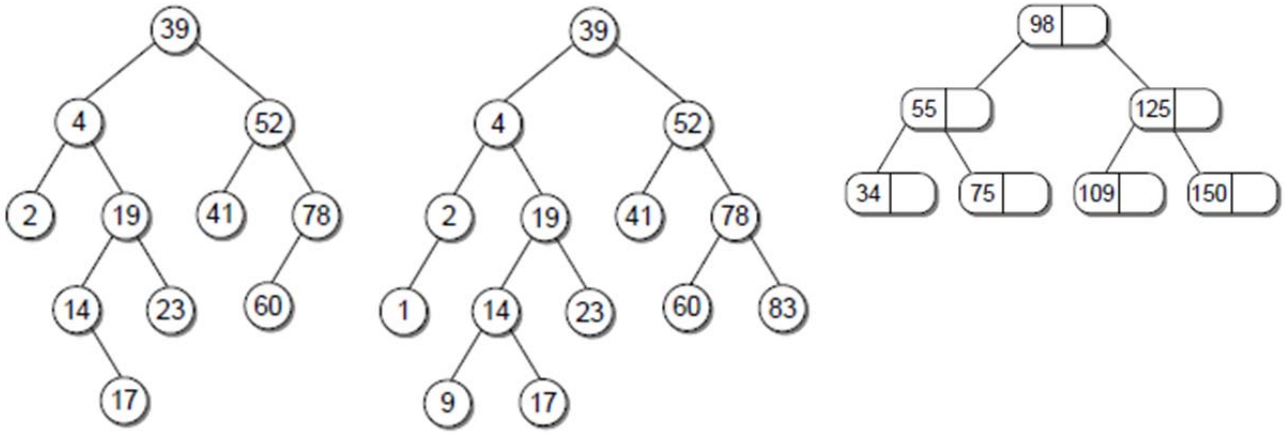
14.5 Dadas as seguintes árvores binárias ao lado, indique quais árvores estão balanceadas em altura.



14.6 Considere a árvore de busca binária em (a) abaixo, mostre a árvore resultante após excluir cada uma das seguintes chaves: 14, 52 e 39.

14.7 Considere a árvore AVL em (b) abaixo, mostre a árvore resultante após excluir os valores-chave 1, 78 e 41.

14.8 Dada a árvore 2-3 em (c) abaixo, mostre a árvore resultante após inserir os valores-chave 112, 80, 90, 41 e 20.



(a) **(b)** **(c)**

PROJETOS DE PROGRAMAÇÃO

14.1 As operações da árvore de busca binária também podem ser implementadas iterativamente. Projete e implemente uma solução iterativa para as operações de pesquisar um nó, encontrar o nó mínimo, inserir um novo nó e excluir um nó.

14.2 Projete e implemente a função **`_bstMaximum()`**, que encontra e retorna o valor máximo da chave em uma árvore de busca binária.

14.3 Implemente a operação de exclusão para AVL e 2-3 árvores.

14.4 Implemente o TAD Set usando uma árvore de busca AVL e avalie a complexidade temporal de cada operação.

14.5 Implemente uma nova versão do TAD ColorHistogram (do Capítulo 11) para usar uma árvore de busca binária para as cadeias em vez de uma lista encadeada.

14.6 Projete e implemente a função **`bstBuild()`**, que pega uma sequência de chaves e constrói uma nova árvore de busca a partir dessas chaves. Por exemplo, a função poderia ser usada para construir a árvore de busca binária da Figura 14.5.

```
keyList = [60, 25, 100, 35, 17, 80]
buildBST(keyList)
```