Apontamentos de Estruturas de Dados

Árvores Binárias de Pesquisa

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Novembro de 2022

Índice

13. ÁRVORES BINÁRIAS DE PESQUISA 1

13.1. INTRODUÇÃO 1

13.2. BINARYSEARCHTREE ADT 2

13.3. INTERFACE BINARYSEARCHTREE 3

13.4. IMPLEMENTAR ÁRVORES BINÁRIAS DE PESQUISA COM LISTAS LIGADAS 5

13.5. IMPLEMENTAR ÁRVORES BINÁRIAS DE PESQUISA COM ARRAYS 12

13.6. BALANCEAMENTO DE ÁRVORES BINÁRIAS DE PESQUISA 14

13.7. ROTAÇÕES 15

13.8. ÁRVORES AVL 18

13.9. NOTAS FINAIS 21

13.10. EXERCÍCIOS PROPOSTOS 21

13. Árvores Binárias de Pesquisa

13.1. Introdução

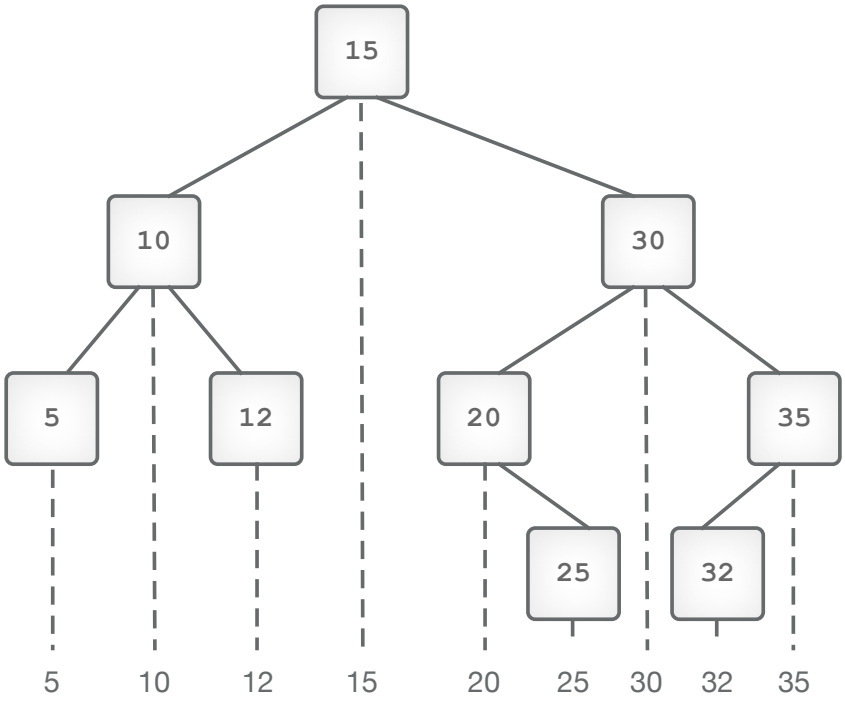
A característica essencial da árvore binária que vimos anteriormente é a relação entre os valores de um nó e os valores das suas subárvores esquerda e direita. Esta é a base para a definição de árvores binárias de pesquisa.

> **Árvore binária de pesquisa:** Uma árvore binária em que cada nó é tal que o valor em cada nó é maior que os valores em sua subárvore esquerda e menor que os valores em sua subárvore direita.

As árvores binárias de pesquisa são uma estrutura de dados importante que retém a propriedade que percorrendo os nós através de uma travessia em-ordem visita os valores nos nós de forma ordenada. No entanto, uma árvore binária de pesquisa pode não estar cheia, a sua altura pode ser maior que log n. De facto, uma árvore binária de pesquisa em que cada nó, exceto um, possui apenas um único filho, terá altura n-1.

As árvores binárias de pesquisa são interessantes porque são rápidas tanto ao inserir elementos quanto ao pesquisá-las (desde que relativamente equilibradas). Isso contrasta com a maioria das coleções, que geralmente são rápidas para inserções mas lentas para pesquisas ou vice-versa. Por exemplo, os elementos podem ser inseridos rapidamente numa `LinkedList` (não ordenada), mas pesquisar numa `LinkedList` é um processo lento, enquanto que num `ArrayList` (ordenado) pode ser pesquisado rapidamente através de uma pesquisa binária, mas inserir elementos para mantê-lo ordenado é lento.

Podemos então dizer de forma muito simplificada que uma árvore binária de pesquisa é uma árvore binária em que os elementos são ordenados de forma que:



Para qualquer nó numa árvore, todos os elementos nos nós da sub-árvore esquerda são menores que o elemento do nó e todos os elementos da sub-árvore direita são maiores ou iguais ao elemento do nó. **Esta será a propriedade das árvores binárias de pesquisa.**

As sub-árvores esquerda e direita são também árvores binárias de pesquisa. Dado este refinamento para a nossa definição anterior de uma árvore binária, agora podemos incluir operações adicionais.

13.2. BinarySearchTree ADT

O ADT de uma árvore binária de pesquisa de T tem como conjunto valores o conjunto de todas as árvores binárias de pesquisa cujos nós possuem um valor do tipo T. É, portanto, um subconjunto do conjunto de valores do ADT da árvore binária de T. As operações neste ADT incluem todas as operações do ADT de árvore binária mais as operações que manipulam a coleção (adicionar e remover) e que dependem do significado da árvore como por exemplo alguns métodos de pesquisa. Podemos então dizer que as operações essenciais para este tipo são:

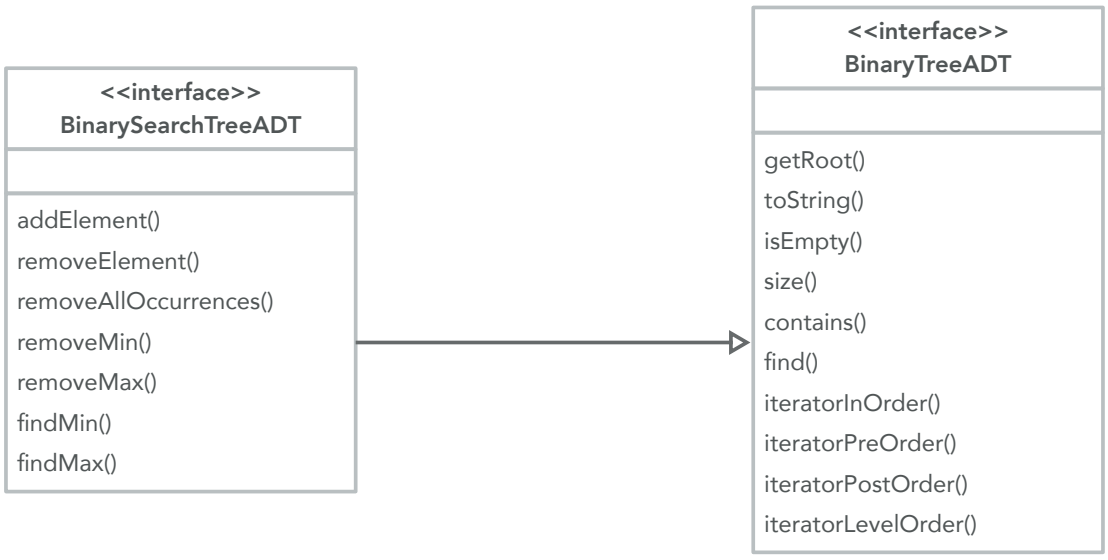
Operação	Descrição
`addElement`	Adiciona um elemento à árvore
`removeElement`	Remove um elemento da árvore
`removeAllOccurrences`	Remove todas as ocorrências do elemento da árvore
`removeMin`	Remove o menor elemento na árvore
`removeMax`	Remove o maior elemento na árvore
`findMin`	Retorna uma referência do menor elemento na árvore
`findMax`	Retorna uma referência do maior elemento na árvore

De realçar os métodos que acabam em *Min* e *Max*, todos estes métodos necessitam de informação acerca da forma como a árvore vai ser representada e usada. Até este momento não tinha sido discutido como é que a árvore se iria comportar tanto na adição de elementos como na remoção.

Este ADT é a base para uma classe `BinarySearchTree` concreta.

13.3. Interface BinarySearchTree

Uma árvore binária de pesquisa é apenas uma árvore binária com a propriedade de ordenação imposta a todos os nós da árvore. Assim, podemos definir a interface `BinarySearchTreeADT` como uma extensão da interface `BinaryTreeADT` tal como temos vindo a fazer para outras coleções.



```
1. /**
2.  * BinarySearchTreeADT defines the interface to a binary search tree.
3.  *
4.  */
5.
6. public interface BinarySearchTreeADT<T> extends BinaryTreeADT<T> {
7.
8.    /**
9.     * Adds the specified element to the proper location in this tree.
10.    *
11.    * @param element  the element to be added to this tree
12.    */
13.   public void addElement(T element);
14.   /**
15.    * Removes and returns the specified element from this tree.
16.    *
17.    * @param targetElement  the element to be removed from this tree
18.    * @return               the element removed from this tree
19.    */
20.   public T removeElement(T targetElement);
21.
22.   /**
23.    * Removes all occurences of the specified element from this tree.
24.    *
25.    * @param targetElement   the element that the list will
26.    *                        have all instances of it removed
27.    */
28.   public void removeAllOccurrences(T targetElement);
29.
30.   /**
31.    * Removes and returns the smallest element from this tree.
32.    *
33.    * @return  the smallest element from this tree.
34.    */
```

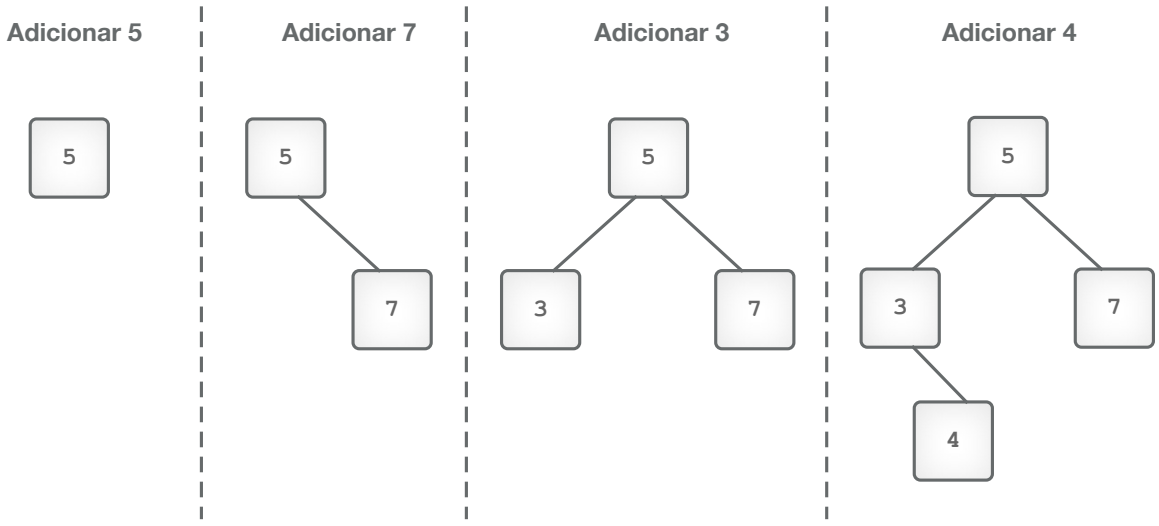
```
35.    public T removeMin();
36.
37.    /**
38.     * Removes and returns the largest element from this tree.
39.     *
40.     * @return  the largest element from this tree
41.     */
42.    public T removeMax();
43.
44.    /**
45.     * Returns a reference to the smallest element in this tree.
46.     *
47.     * @return  a reference to the smallest element in this tree
48.     */
49.    public T findMin();
50.
51.    /**
52.     * Returns a reference to the largest element in this tree.
53.     *
54.     * @return  a reference to the largest element in this tree
55.     */
56.    public T findMax();
57.}
```

13.4. Implementar árvores binárias de pesquisa com listas ligadas

Podemos simplesmente estender a nossa definição de uma `LinkedBinaryTree` para criar uma `LinkedBinarySearchTree`. Esta classe irá fornecer dois construtores, um para criar uma árvore vazia e o outro para criar uma de árvore binária de um elemento.

```
1. public class LinkedBinarySearchTree<T>  extends LinkedBinaryTree<T>
2.                                  implements BinarySearchTreeADT<T> {
3.    /**
4.     * Creates an empty binary search tree.
5.     */
6.    public LinkedBinarySearchTree() {
7.
8.       super();
9.    }
10.
11.   /**
12.    * Creates a binary search with the specified element as its root.
13.    *
14.    * @param element the element that will be the root of the new
15.    * binary search tree
16.    */
17.   public LinkedBinarySearchTree(T element) {
18.      super (element);
19.   }
20. (...)
```

Agora que sabemos mais sobre como esta árvore deverá ser usada (e estruturada), é possível definir um método para adicionar um elemento à árvore. O método `addElement` encontra o local apropriado para o elemento dado e adiciona-o (no local) como uma folha.

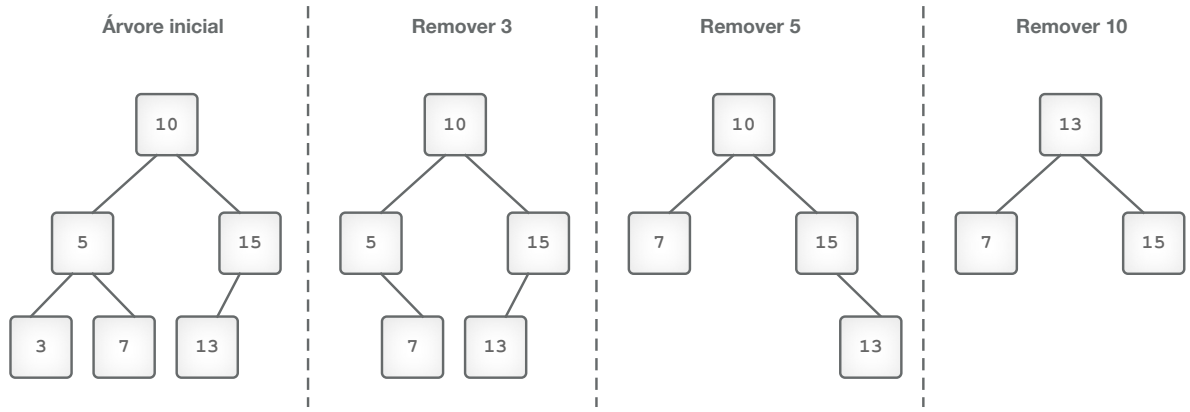


```
1.    /**
2.     * Adds the specified object to the binary search tree in the
3.     * appropriate position according to its key value.  Note that
4.     * equal elements are added to the right.
5.     *
6.     * @param element  the element to be added to the binary search
7.     * tree
8.     */
9.    public void addElement (T element) {
10.      BinaryTreeNode<T> temp = new BinaryTreeNode<T> (element);
11.      Comparable<T> comparableElement = (Comparable<T>)element;
12.
13.      if (isEmpty()) {
14.         root = temp;
15.      } else {
16.         BinaryTreeNode<T> current = root;
17.         boolean added = false;
18.         while (!added) {
19.            if (comparableElement.compareTo(current.element) < 0) {
20.               if (current.left == null) {
21.                  current.left = temp;
22.                  added = true;
23.               } else {
24.                  current = current.left;}
25.            } else {
26.               if (current.right == null) {
27.                  current.right = temp;
28.                  added = true;
29.               } else {
30.                  current = current.right;}
31.            }
32.         }
33.      }
34.      count++;
35.   }
```

Remover elementos de uma árvore de binária de pesquisa requer:

- *Encontrar o elemento a ser removido;*
- *Se esse elemento não é uma folha, de seguida substituí-lo com o seu sucessor inorder*;
- *Retornar o elemento removido.*

O método `removeElement` faz uso de um método de substituição privado (`replacement`) para encontrar o elemento adequado para substituir um elemento não-folha que é removido.




```
1.  /**
2.   * Removes the first element that matches the specified target
3.   * element from the binary search tree and returns a reference to
4.   * it.  Throws a ElementNotFoundException if the specified target
5.   * element is not found in the binary search tree.
6.   *
7.   * @param targetElement   the element being sought in the binary
8.   *                        search tree
9.   * @throws ElementNotFoundException  if an element not found
10.  *                                   exception occurs
11.  */
12.   public T removeElement (T targetElement)
13.                          throws ElementNotFoundException {
14.      T result = null;
15.      if (!isEmpty()) {
16.         if (((Comparable)targetElement).equals(root.element)) {
17.            result = root.element;
18.            root = replacement (root);
19.            count--;
```

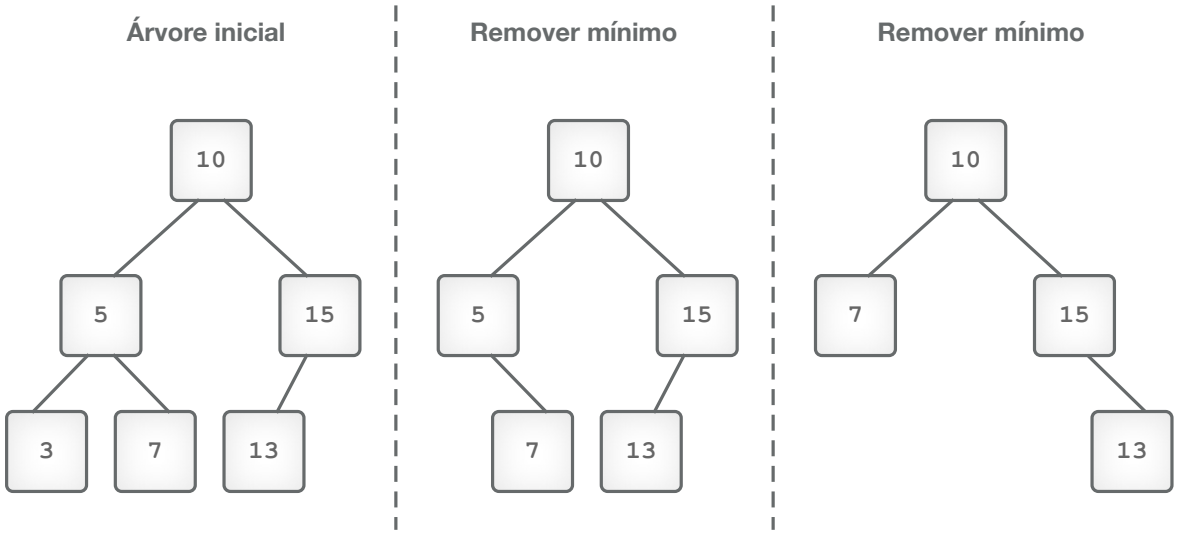
```
20.         } else {
21.            BinaryTreeNode<T> current, parent = root;
22.            boolean found = false;
23.
24.            if (((Comparable)targetElement).compareTo(root.element)<0)
25.               current = root.left;
26.            else
27.               current = root.right;
28.            while (current != null && !found) {
29.               if (targetElement.equals(current.element)) {
30.                  found = true;
31.                  count--;
32.                  result =  current.element;
33.                  if (current == parent.left) {
34.                     parent.left = replacement (current);
35.                  } else {
36.                     parent.right = replacement (current);
37.                  }
38.               } else {
39.                  parent = current;
40.          If (((Comparable)targetElement).compareTo(current.element)<0)
41.                     current = current.left;
42.                  else
43.                     current = current.right;
44.               }
45.            } //while
46.
47.            if (!found) {
48.               throw new
49.                   ElementNotFoundException("binary search tree");}
50.         }
51.      } // end outer if
52.      return result;
53.   }
54.
```

```
1.    /**
2.     * Returns a reference to a node that will replace the one
3.     * specified for removal.  In the case where the removed node has
4.     * two children, the inorder successor is used as its replacement.
5.     *
6.     * @param node  the node to be removeed
7.     * @return      a reference to the replacing node
8.     */
9.    protected BinaryTreeNode<T> replacement (BinaryTreeNode<T> node)
10.   {
11.      BinaryTreeNode<T> result = null;
12.
13.      if ((node.left == null)&&(node.right==null))
14.         result = null;
15.      else if ((node.left != null)&&(node.right==null))
16.         result = node.left;
17.      else if ((node.left == null)&&(node.right != null))
18.         result = node.right;
19.      Else {
20.         BinaryTreeNode<T> current = node.right;
21.         BinaryTreeNode<T> parent = node;
22.         while (current.left != null) {
23.            parent = current;
24.            current = current.left;
25.         }
26.         if (node.right == current)
27.            current.left = node.left;
28.         Else {
29.            parent.left = current.right;
30.            current.right = node.right;
31.            current.left = node.left;
32.         }
33.         result = current;
34.      }
35.      return result;
36.   }
```

O método para remover todas as ocorrências `removeAllOccurrences` remove todas as ocorrências de um elemento da árvore. Este método usa o método `removeElement`. Este método faz uma distinção entre a primeira chamada e as sucessivas chamadas ao método `removeElement`.

No caso do método para remover o elemento mínimo de uma árvore binária de pesquisa temos de ter em atenção aos três casos possíveis para a localização do mínimo elemento:

- Se a raiz não tem filho esquerdo, então a raiz é o elemento mínimo e o filho à direita da raiz torna-se a nova raiz;
- Se o nó mais à esquerda da árvore é uma folha, então colocamos a referência do filho esquerdo do pai a `null;`
- Se o nó mais à esquerda da árvore é um nó interno, então colocamos a referência do filho esquerdo do pai a apontar para o filho direito do nó a ser removido.



13.5. Implementar árvores binárias de pesquisa com arrays

Tal como fizemos para a `LinkedBinarySearchTree` podemos estender a nossa `ArrayBinaryTree` para criar uma `ArrayBinarySearchTree`.

```
1. /**
2.  * ArrayBinarySearchTree implements a binary search tree
3.  * using an array.
4.  *
5.  */
6.
7.  public class ArrayBinarySearchTree<T>  extends ArrayBinaryTree<T>
   implements BinarySearchTreeADT<T>
8. {
9.    protected int height;
10.   protected int maxIndex;
11.
12.   /**
13.    * Creates an empty binary search tree.
14.    */
15.   public ArrayBinarySearchTree() {
16.      super();
17.      height = 0;
18.      maxIndex = -1;
19.   }
20.
21.   /**
22.    * Creates a binary search with the specified element as its root
23.    *
24.    * @param element  the element that will become the root of
25.    * the new tree
26.    */
27.   public ArrayBinarySearchTree(T element) {
28.      super(element);
29.      height = 1;
30.      maxIndex = 0;
31.   }
```

```
32.
33.    /**
34.     * Adds the specified object to this binary search tree in the
35.     * appropriate position according to its key value.  Note that
36.     * equal elements are added to the right.  Also note that the
37.     * index of the left child of the current index can be found by
38.     * doubling the current index and adding 1.  Finding the index
39.     * of the right child can be calculated by doubling the current
40.     * index and adding 2.
41.     *
42.     * @param element  the element to be added to the search tree
43.     */
44.    public void addElement (T element) {
45.       if (tree.length < maxIndex*2+3) {
46.          expandCapacity();}
47.       Comparable<T> tempelement = (Comparable<T>)element;
48.
49.       if (isEmpty()) {
50.          tree[0] = element;
51.          maxIndex = 0;
52.       } else {
53.          boolean added = false;
54.          int currentIndex = 0;
55.          while (!added) {
56.             if (tempelement.compareTo((tree[currentIndex]) ) < 0) {
57.                /** go left */
58.                if (tree[currentIndex*2+1] == null) {
59.                   tree[currentIndex*2+1] = element;
60.                   added = true;
61.                   if (currentIndex*2+1 > maxIndex)
62.                      maxIndex = currentIndex*2+1;
63.                }
64.                else
65.                   currentIndex = currentIndex*2+1;
66.             } else {
67.                /** go right */
68.                if (tree[currentIndex*2+2] == null) {
69.                   tree[currentIndex*2+2] = element;
```

```
70.                     added = true;
71.                     if (currentIndex*2+2 > maxIndex)
72.                         maxIndex = currentIndex*2+2;
73.                 }
74.                 else
75.                     currentIndex = currentIndex*2+2;
76.             }
77.         }
78.     }
79.     height = (int)(Math.log(maxIndex + 1) / Math.log(2)) + 1;
80.     count++;
81. }
```

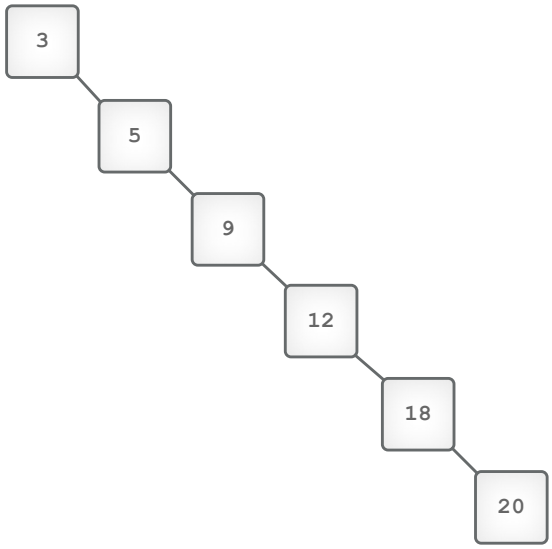
13.6. Balanceamento de árvores binárias de pesquisa

Se os itens são adicionados numa árvore binária de pesquisa de forma aleatória, a árvore tende a ser bem equilibrada com altura não muito maior que $\log_2 n$. No entanto, há muitas situações em que os itens adicionados não estão em ordem aleatória, como ao adicionar novos IDs de alunos (através do número). No caso extremo de os novos itens serem adicionados em ordem crescente, a árvore será composta por um longo caminho à direita, com altura $n \gg \log_2 n$.

Se todos os itens a serem inseridos numa árvore binária de pesquisa já estiverem ordenados é simples construir uma árvore binária perfeitamente balanceada a partir deles. Basta construir recursivamente uma árvore binária com o item do meio (ou seja, mediano) como raiz, a subárvore esquerda composta pelos itens menores e a subárvore direita composta pelos itens maiores. Essa ideia pode ser usada para reequilibrar qualquer árvore binária de pesquisa existente, porque a árvore existente pode ser facilmente gerada a partir de um *array* ordenado. Podemos ver uma exemplificação do problema ao introduzir a seguinte lista de elementos:

3 5 9 12 12 18 20

O resultado seria:



A árvore resultante é chamada de árvore binária degenerada. As árvores binárias de pesquisa degeneradas são muito menos eficientes do que as árvores binárias de pesquisa balanceadas (O(n) na pesquisa em oposição a O(log n)).

Existem muitas abordagens para balancear árvores binárias, um método é a força bruta:

- Escrever uma travessia em-ordem para um ficheiro;
- Usar uma pesquisa recursiva binária do ficheiro para reconstruir a árvore.

Soluções melhores envolvem algoritmos, tais como árvores vermelhas e pretas e as árvores AVL que persistentemente mantêm o equilíbrio da árvore. A maioria de todos estes algoritmos fazem uso de rotações para balancear a árvore.

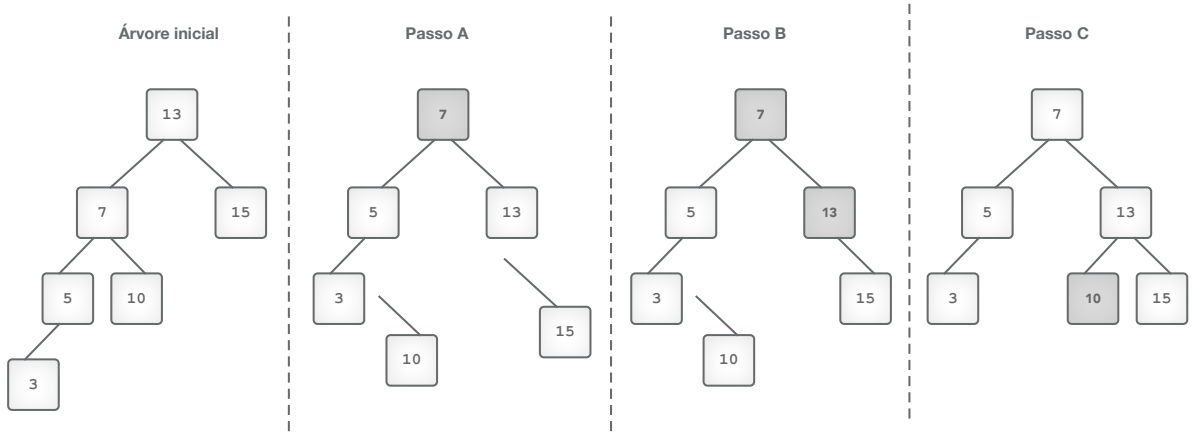
13.7. Rotações

Rotação à Direita

A rotação à direita vai resolver um não balanceamento se for causado por um longo caminho na sub-árvore esquerda do filho esquerdo da raiz e envolve:

- Fazer do elemento filho esquerdo da raiz o novo elemento raiz;

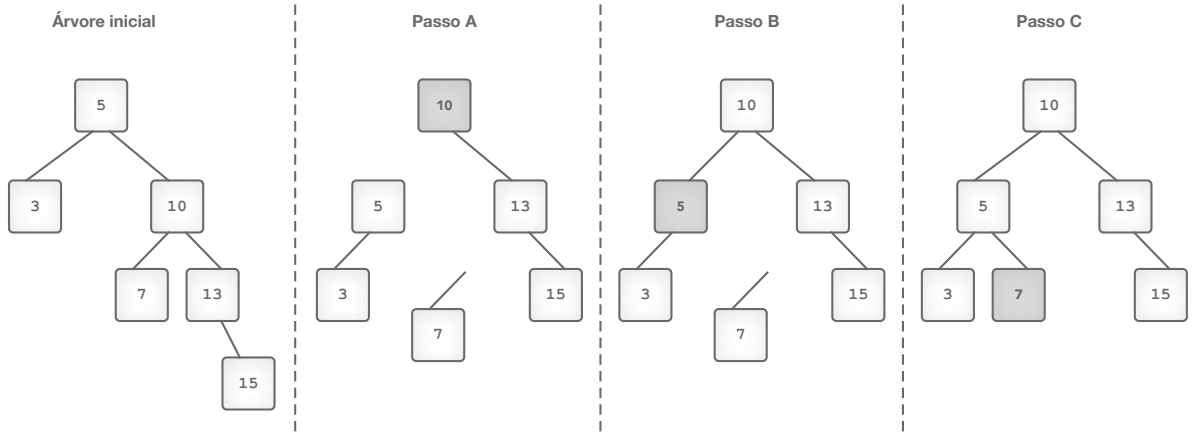
- Fazer do antigo elemento raiz, filho à direita da nova raiz;
- Fazer do antigo filho direito da nova raiz, filho esquerdo da antiga raiz



Rotação à Esquerda

A rotação à esquerda vai resolver um desequilíbrio que é causada por um longo caminho na sub-árvore direita do filho direito da raiz e envolve:

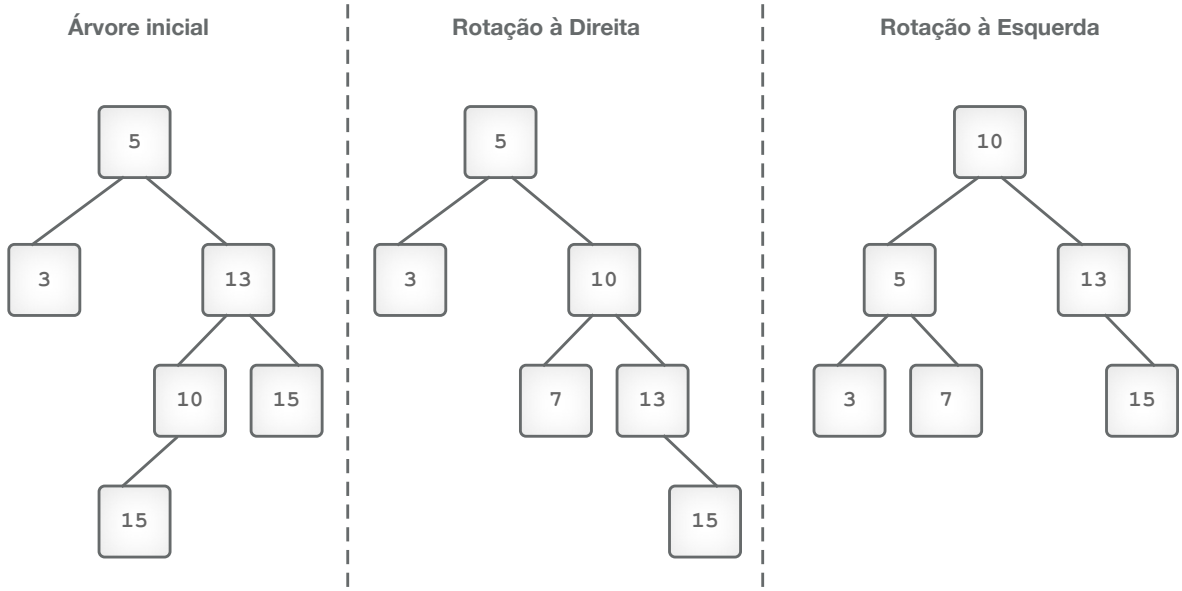
- Fazer do elemento filho direito da raiz o novo elemento raiz;
- Fazer do antigo elemento raiz, filho à esquerda da nova raiz;
- Fazer do antigo filho esquerdo da nova raiz, filho direito da antiga raiz.



Rotação Direita-Esquerda

A rotação Direita-Esquerda vai resolver um não balanceamento se for causado por um longo caminho na sub-árvore esquerda do filho à direita da raiz e envolve:

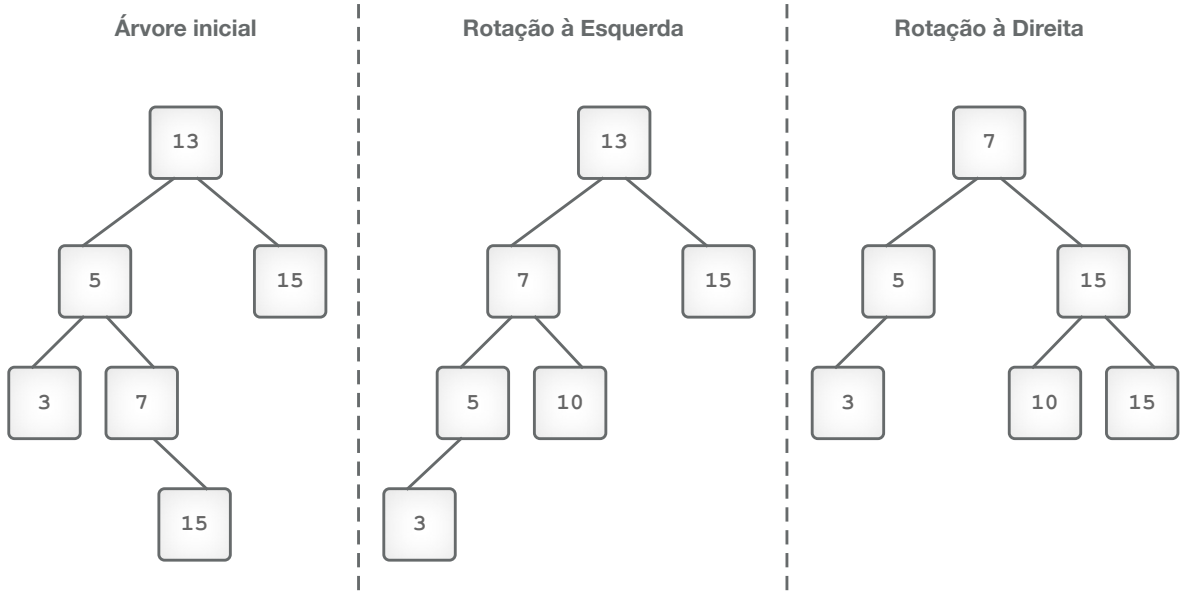
- Executar uma rotação à direita do filho esquerdo do filho direito da raiz em torno do filho à direita da raiz e, em seguida realizar uma rotação à esquerda do filho direito resultante da raiz em torno da raiz.



Rotação Esquerda-Direita

A rotação Esquerda-Direita vai resolver um não balanceamento, se for causado por um longo caminho na sub-árvore direita do filho esquerdo da raiz e envolve:

- Realizar uma rotação à esquerda do filho direito do filho esquerdo da raiz em torno do filho esquerdo da raiz, e, em seguida, executar uma rotação à direita do filho esquerdo resultante da raiz em torno da raiz.



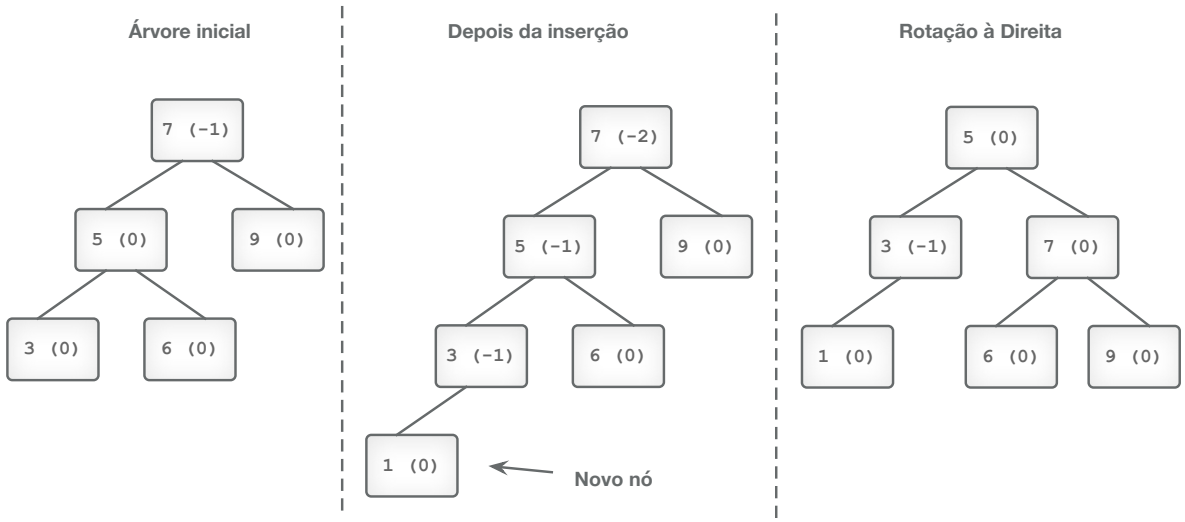
13.8. Árvores AVL

Árvores binárias de pesquisa auto-balanceadas evitam o problema das árvores degeneradas rebalanceando automaticamente a árvore durante todo o processo de inserção para manter a altura próxima a $\log_2 n$ em cada fase. Obviamente, haverá um custo envolvido com o rebalanceamento e haverá um *trade-off* entre o tempo envolvido no rebalanceamento e o tempo economizado pela redução da altura da árvore mas geralmente vale a pena.

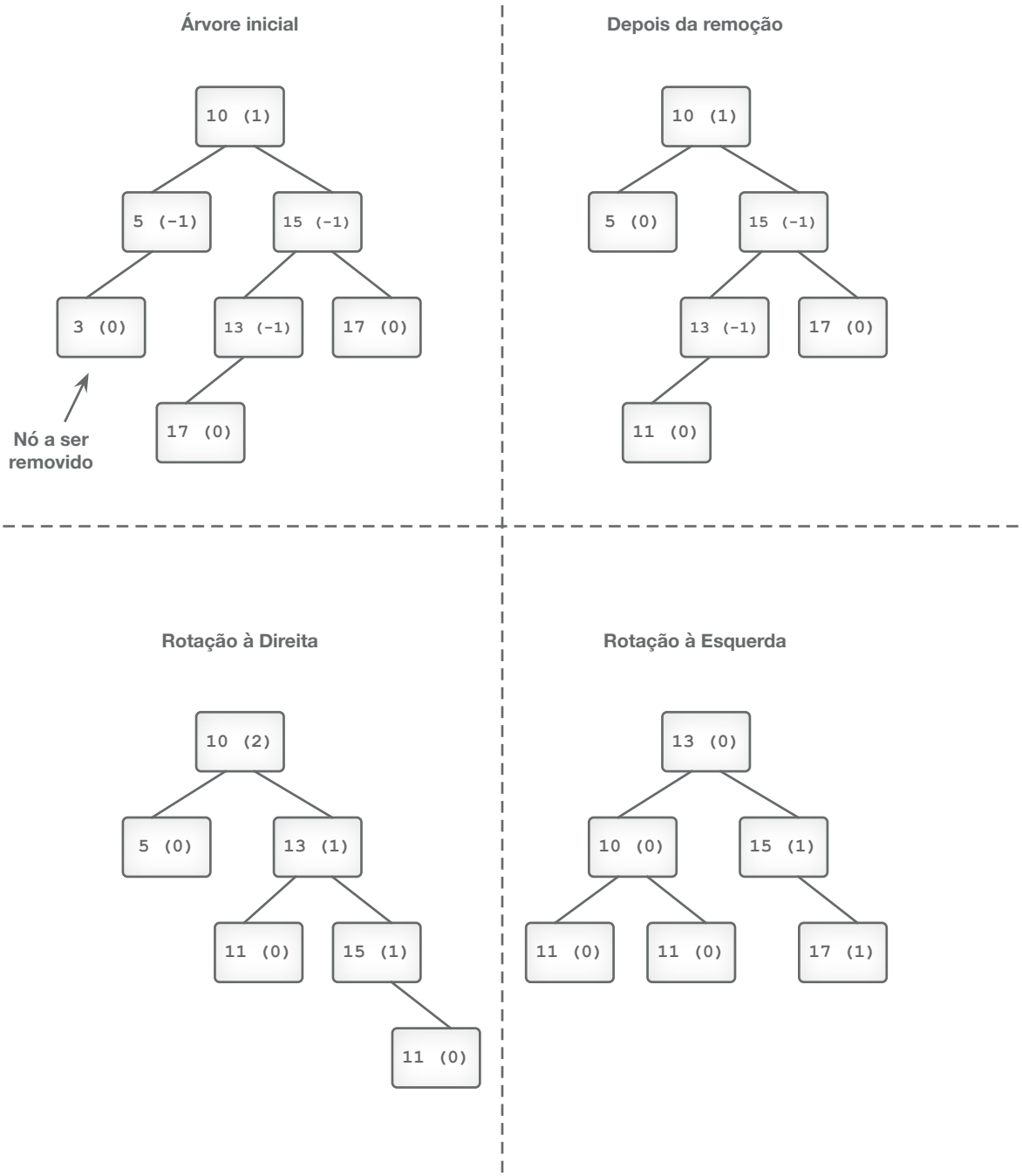
O primeiro tipo de árvore binária de pesquisa auto-equilibrada foi a árvore AVL (em homenagem a seus autores G.M. Adelson-Velskii e E.M. Landis). Este tipo de árvore mantêm a diferença nas alturas das duas sub-árvores de todos os nós para ser no máximo 1. Isso requer que a árvore seja periodicamente rebalanceada executando uma ou mais rotações de árvore conforme discutido acima, mas a complexidade de inserção, remoção e busca permanece em $O(\log_2 n)$.

A ideia geral é acompanhar o fator de equilíbrio para cada nó que é a altura da subárvore esquerda menos a altura da subárvore direita. Por definição, todos os nós numa árvore AVL têm uma variável que mantém o fator de equilíbrio no intervalo inteiro [−1, 1]. No entanto, a inserção ou remoção de um nó pode deixá-lo num intervalo mais amplo [−2, 2] exigindo uma rotação de árvore para trazê-lo de volta à forma AVL. Se o factor de balanceamento de

qualquer nó é menor que -1 ou maior que 1, então, a sub-árvore precisa ser balanceada. O factor de balanceamento de qualquer nó só pode ser alterado, quer através de inserção ou remoção de nós na árvore. Se o factor de balanceamento de um nó é -2, significa que a sub-árvore esquerda tem um caminho que é longo demais. Se o factor de balanceamento do filho esquerdo é -1, significa que o caminho longo é o da sub-árvore esquerda do filho esquerdo. Neste caso, uma rotação simples à direita do filho esquerdo em torno do nó original irá resolver o balanceamento.



Se o factor de balanceamento de um nó é +2, significa que a sub-árvore direita tem um caminho que é longo demais. Então, se o factor de balanceamento do filho direito é +1, significa que o caminho longo é da sub-árvore direita do filho direito. Neste caso, uma rotação simples à esquerda do filho direito em torno do nó original vai resolver o balanceamento. Se o factor de balanceamento de um nó é +2, significa que a sub-árvore direita tem um caminho que é longo demais. Então, se o factor de balanceamento do filho direito é -1, significa que o caminho longo é da sub-árvore esquerda do filho direito. Neste caso, uma rotação Direita-Esquerda irá resolver o balanceamento.



Se o factor de balanceamento de um nó é -2, significa a sub-árvore esquerda tem um caminho que é longo demais. Então, se o factor de balanceamento do filho esquerdo é +1, significa que o caminho longo é da sub-árvore direita do filho esquerdo. Neste caso, uma rotação Esquerda-Direita irá resolver o balanceamento

13.9. Notas Finais

A pesquisa binária é um algoritmo muito eficiente para pesquisar listas ordenadas com complexidade média e de pior caso em O(log n). Podemos representar o funcionamento da pesquisa binária numa árvore binária para produzir uma árvore binária de pesquisa completa. As árvores binárias de pesquisa possuem várias propriedades interessantes e fornecem um tipo de coleção que apresenta excelente desempenho para adição, remoção e pesquisa, exceto no pior caso. Também podemos percorrer árvores binárias de pesquisa para aceder aos elementos da coleção de forma ordenada.

13.10. Exercícios Propostos

Exercício 1

Que vantagem tem árvore binária de pesquisa sobre coleções como `ArrayList` e `LinkedList`?

Exercício 2

Uma pré-condição da pesquisa binária é que o *array* a ser pesquisado esteja ordenado. Qual é a complexidade de um algoritmo para verificar essa pré-condição?

Exercício 3

Desenhe todas as árvores binárias de pesquisa que podem ser formadas usando os valores a, b e c . Quantas são árvores binárias completas?

Exercício 4

Inserir os seguintes valores numa árvore binária de pesquisa: 40, 30, 50, 90, 45, 80, 95, 20, 25.

Exercício 5

Excluir os seguintes valores da árvore binária de pesquisa anterior: 20, 50, 40.

Exercício 6

Que vantagem tem uma árvore AVL sobre um árvore binária de pesquisa tradicional?

Exercício 7

Como implementaríamos outro tipo de árvore binária de pesquisa através da nossa estrutura de ADTs? Justifique a sua resposta.