Apontamentos de Estruturas de Dados

Árvores

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Outubro de 2024

Índice

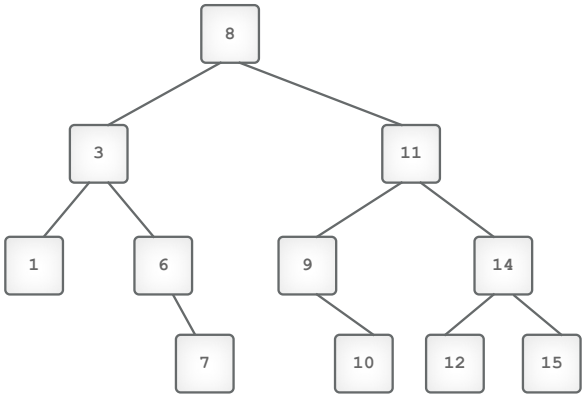
12. Árvores

12.1. Introdução

Na ciência da computação, uma árvore é uma estrutura de dados muito geral e poderosa que se assemelha a uma árvore real. Consiste num conjunto ordenado de nós ligados num grafo conectado, no qual cada nó possui no máximo um nó pai e zero ou mais nós filhos com uma ordem específica.

12.2. Especificação geral de árvores

Geralmente, podemos especificar uma árvore como consistindo de nós (também chamados de vértices) e arestas (por exemplo, para enfatizar o direcionamento, arcos). Geralmente é mais fácil representar as árvores pictoricamente, e faremos isso com frequência. Um exemplo simples é dado na figura seguinte.



Mais formalmente, uma árvore pode ser definida como a árvore vazia ou como um nó com uma lista de árvores sucessoras. Os nós geralmente, embora nem sempre, são rotulados com um item de dados (como um número ou chave de pesquisa). Vamos nos referir ao rótulo de um nó como o seu valor. Nos próximos exemplos iremos geralmente usar nós rotulados por números inteiros, mas pode-se facilmente escolher outra coisa como por exemplo cadeias de caracteres.

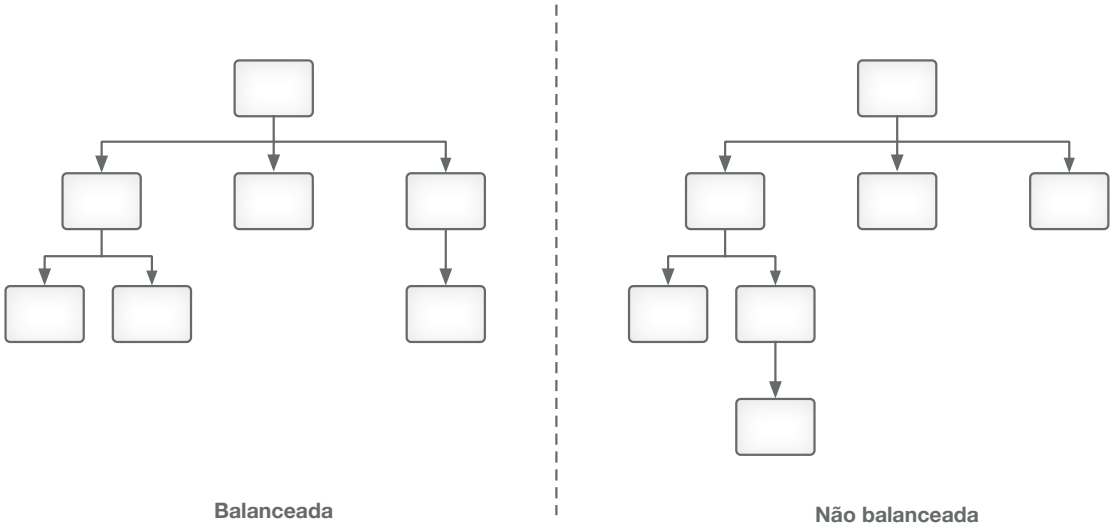
Para falar com rigor acerca de árvores, é conveniente ter alguma terminologia: tem sempre de haver um nó único de “nível superior” conhecido como raiz. Na figura anterior este nó é rotulado com um 8. É importante notar que, em ciências da computação, as árvores são normalmente exibidas de cabeça para baixo representando a hierarquia com a raiz formando o nível superior. Então, dado um nó, cada nó no próximo nível 'para baixo', que está conectado ao nó dado através de uma ramificação, é um filho desse nó. Os filhos do nó 8 são os nós 3 e 11. Inversamente, o nó (há no máximo um) conectado ao nó dado (através de uma aresta) no nível acima, é seu pai. Por exemplo, o nó 11 é o pai do nó 9 (e do nó 14 também). Os nós que têm o mesmo pai são conhecidos como irmãos – os irmãos estão, por definição, sempre no mesmo nível.

Se um nó é filho de um filho de um outro nó, dizemos que o primeiro nó é descendente do segundo nó. Por outro lado, o segundo nó é um ancestral do primeiro nó. Os nós que não têm filhos são conhecidos como folhas (por exemplo, os nós rotulados com 1, 7, 10, 12 e 15 na figura).

Um caminho é uma sequência de arestas conectadas de um nó a outro. As árvores têm a propriedade de que para cada nó existe um caminho único conectando-o à raiz. Na verdade, essa é outra definição possível de uma árvore. A profundidade ou nível de um nó é dado pelo comprimento deste caminho. Portanto, a raiz tem nível 0, seus filhos têm nível 1 e assim por diante. O comprimento máximo de um caminho numa árvore também é chamado de altura da árvore. Um caminho de comprimento máximo vai sempre da raiz até uma folha. O tamanho de uma árvore é dado pelo número de nós que ela contém. Normalmente, assumiremos que toda árvore é finita, embora geralmente esse não seja o caso. A árvore na figura anterior tem altura 3 e tamanho 11. Uma árvore que consiste em apenas um nó tem altura 0 e tamanho 1. A árvore vazia obviamente tem tamanho 0 e é definida (convenientemente, embora um tanto artificialmente) para ter altura −1.

Como a maioria das estruturas de dados, precisamos de um conjunto de operadores primitivos (construtores, seletores e condições) para construir e manipular as árvores. Os detalhes dependem do tipo e da finalidade da árvore.

Uma árvore é considerada balanceada ou equilibrada quando todas as folhas da árvore estão aproximadamente na mesma profundidade. Alguns algoritmos definem equilibrado quando todas as folhas estão no nível *h* ou *h-1*, onde *h* é a altura da árvore e onde $log_N n = h$ é uma árvore n-ária



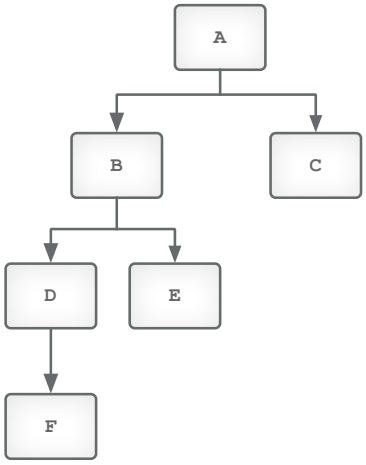
O conceito de uma árvore completa está relacionado com o equilíbrio de uma árvore. Uma árvore é considerada completa se estiver balanceada e se todas as folhas no nível *h* estiverem no lado esquerdo da árvore. Esta definição tem implicações na forma como a árvore é armazenada em certas implementações.

12.3. Implementar árvores gerais com listas ligadas

Embora não seja possível discutir os detalhes da implementação de uma árvore, sem definir o tipo de árvore e o seu uso, podemos olhar para as estratégias gerais de implementação de árvores. A implementação mais óbvia da árvore é uma estrutura ligada. Cada nó pode ser definido como uma classe `TreeNode`, como fizemos com a classe `LinearNode` para as listas ligadas. Cada nó deve conter uma referência para o elemento a ser armazenado no nó, assim como as referências para cada um dos possíveis filhos desse nó. Dependendo da aplicação, pode também ser útil armazenar uma referência em cada nó para o seu pai

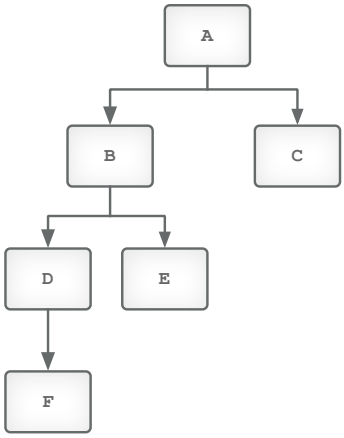
12.4. Implementar árvores gerais com *arrays*

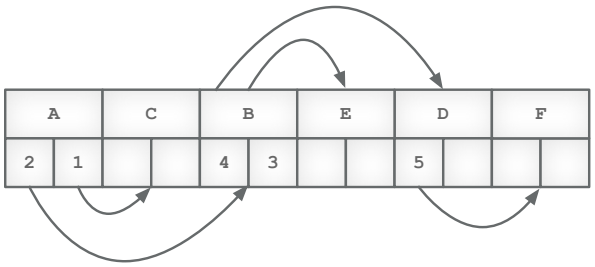
Para certos tipos de árvores, especialmente árvores binárias, pode ser usada uma estratégia computacional para armazenar uma árvore com recurso a um *array*. Para qualquer elemento armazenado na posição *n* do *array*, o filho esquerdo será armazenado na posição *((2 * n) + 1)* e o filho direito será armazenado na posição *(2 * (n +1))*. Esta estratégia pode ser gerida em termos de capacidade da mesma forma que fizemos para outras coleções baseadas em array. No entanto, apesar da elegância conceptual desta solução existem inconvenientes. Por exemplo, se a árvore em que estamos a armazenar não está completa ou relativamente completa, poderemos estar a desperdiçar uma grande quantidade de memória alocada no *array* para posições da árvore livres que não contêm dados.



elemento	A	B	C	D	E			F
posição	0	1	2	3	4	5	6	7

Uma segunda possibilidade para a implementação em *array* das árvores é modelada segundo a forma de gestão de memória dos Sistemas Operativos. Em vez de atribuir os elementos da árvore para a posição do *array* por localização na árvore, as posições do *array* são alocadas de forma contígua em que o primeiro a chegar, é primeiro a ser servido. Cada elemento do *array* será uma classe nó semelhante à classe `TreeNode` que analisámos anteriormente. No entanto, em vez de armazenarmos as variáveis da referência do objeto para os seus filhos (e talvez do pai), seria armazenado em cada nó o índice do *array* de cada filho (e talvez do pai). Esta abordagem permite que os elementos sejam armazenados de forma contígua no *array* de modo que o espaço não seja desperdiçado. No entanto, esta abordagem aumenta a sobrecarga (piora a performance) para excluir elementos na árvore uma vez que os restantes elementos terão de ser deslocados para manter a contiguidade ou então terá de ser gerida uma lista de posições livres





12.5. Árvores binárias

Árvores binárias são o tipo mais comum de árvore usado em ciências da computação. Uma árvore binária é uma árvore em que cada nó tem no máximo dois filhos, e pode ser definida "recursivamente" pelas seguintes regras:

> **Definição.** Uma árvore binária é
> *(Regra 1) a árvore vazia EmptyTree, ou*
> *(Regra 2) consiste num nó com duas árvores binárias, a subárvore esquerda e a subárvore direita.*

Novamente, a Regra 1 é o "caso base" e a Regra 2 é o "caso recursivo". Essa definição pode parecer circular, mas na verdade não é, porque as subárvores são sempre mais simples que a original e acabamos com uma árvore vazia.

Uma árvore binária completa é aquela em que todos os níveis estão completos, exceto possivelmente o último. Uma árvore binária cheia é uma árvore binária em que todos os nós internos têm dois filhos e todas as folhas estão no mesmo nível.

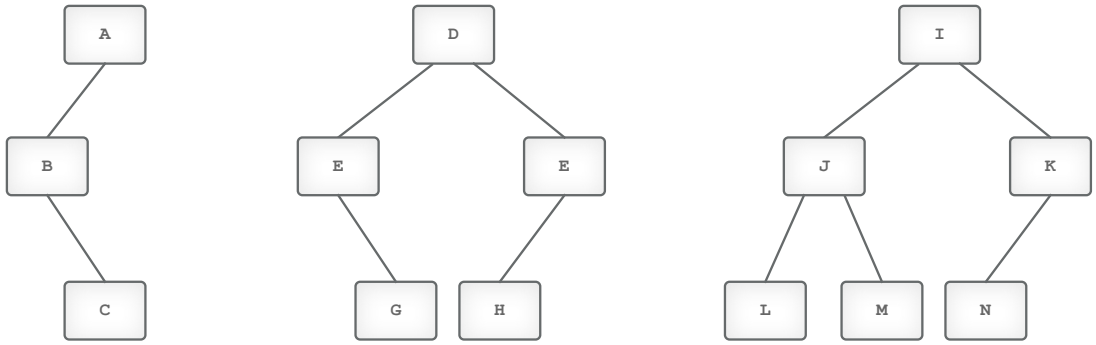
Características principais:

- **Árvore Binária Completa:**
    - Todos os níveis estão cheios, exceto possivelmente o último;
    - O último nível é preenchido da esquerda para a direita;

- Pode ter o último nível incompleto.

- **Árvore Binária Cheia:**
  - Todos os nós internos têm exatamente dois filhos;
  - Todas as folhas estão no mesmo nível;
  - São sempre "perfeitas" em sua estrutura.

Analise as seguintes árvores:



As árvores apresentadas anteriormente são árvores binárias. Na árvore à esquerda, o nó A tem um filho à esquerda, o nó B tem um filho à direita e o nó C não tem filhos. Esta árvore não é completa.

A árvore do meio não está cheia nem completa, pois seus nós E têm apenas um filho cada (uma árvore cheia requer que cada nó tenha 0 ou 2 filhos).

A árvore da direita está completa, pois todos os níveis exceto o último estão totalmente preenchidos, e o último nível está preenchido da esquerda para a direita (nós L, M e N) sem deixar espaços vazios entre os nós.

As árvores têm várias propriedades interessantes e importantes, entre elas:

- **Propriedade 1:** Uma árvore binária cheia de altura h tem $2^h - 1$ nós;
- **Propriedade 2:** A altura da árvore binária cheia com n nós é $log_2 n$. Este será também o tamanho do caminho mais longo de uma árvore binária cheia;

- **Propriedade 3:** A altura da árvore binária com *n* nós é no mínimo $log_2 n$ e no máximo *n* (isto ocorre quando nenhum dos nós tem mais de um filho). Será também o tamanho do maior caminho.

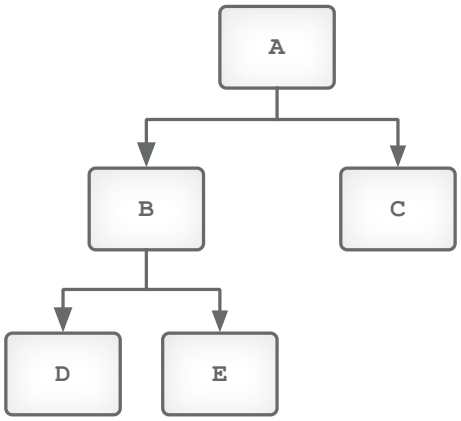
12.6. Travessias

Visitar ou enumerar os nós de uma árvore binária é percorrê-los ou iterar um de cada vez, processando os valores contidos em cada nó. Isso requer que os nós sejam percorridos em alguma ordem. Existem três ordens fundamentais para percorrer uma árvore binária. Todos são descritos mais naturalmente em termos recursivos.

- Pré-ordem: Quando os nós de uma árvore binária são visitados em pré-ordem, o nó raiz da árvore é visitado primeiro, então a sub-árvore esquerda (se houver) é visitada em pré-ordem, depois a sub-árvore direita (se houver) é visitada em pré-ordem.
- Em-ordem: Quando os nós de uma árvore binária são visitados em-ordem, a subárvore esquerda (se houver) é visitada em-ordem, depois o nó raiz é visitado, de seguida a subárvore direita é visitada em-ordem.
- Pós-ordem: Quando os nós de uma árvore binária são visitados em pós-ordem, a sub-árvore esquerda é visitada em pós-ordem, a sub-árvore direita é visitada em pós-ordem e, em seguida, o nó raiz é visitado.

Existe ainda a travessia nível-ordem que também será apresentado no entanto este não é recursivo e não terá o mesmo destaque que os outros no nosso estudo.

Para ilustrar as travessias, considere a árvore binária na figura abaixo.



Travessia Pré-Ordem

Dito em pseudocódigo, o algoritmo para uma travessia pré-ordem de uma árvore binária é:

```
1. Visit node
2. Traverse(left child)
3. Traverse(right child)
```

A travessia pré-ordem é realizada, visitando cada nó, começando pela a raiz e de seguida pelos seus filhos. Dada a árvore binária completa da figura anterior, um percurso pré-ordem seria:

```
1. A B D E C
```

Travessia Em-Ordem

Dito em pseudocódigo, o algoritmo para uma travessia em-ordem de uma árvore binária é:

```
1. Traverse(left child)
2. Visit node
3. Traverse(right child)
```

A travessia em-ordem é realizada ao visitar o filho esquerdo do nó, o nó, então todos os nós do filho restantes a partir da raiz. Um percurso em-ordem da árvore anterior produz a ordem:

```
1. D  B  E  A  C
```

Travessia Pós-Ordem

Dito em pseudocódigo, o algoritmo para uma travessia pós-ordem de uma árvore binária é:

```
1. Traverse(left child)
2. Traverse(right child)
3. Visit node
```

A travessia pós-ordem é realizado ao visitar os filhos, e depois o nó a partir da raiz. Dada a mesma árvore, um percurso pós-ordem produz a seguinte ordem:

```
1. D E B C A
```

Travessia Nível-Ordem

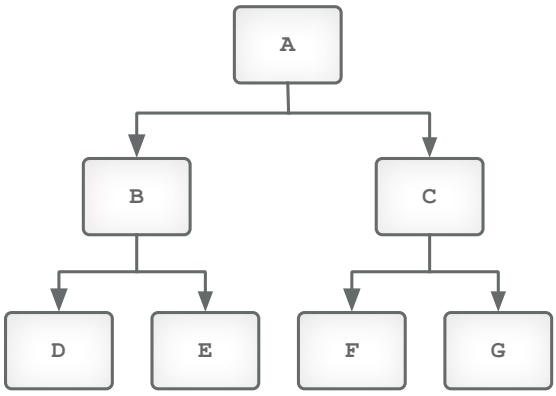
Dito em pseudocódigo, o algoritmo para uma travessia nível-ordem de uma árvore binária é:

```
1. Create a queue called nodes
2. Create an unordered list called results
3. Enqueue the root onto the nodes queue
4. While the nodes queue is not empty
5.   Dequeue the first element from the queue
6.   If that element is not null
7.           Add that element to the rear of the results list
8.           Enqueue the children of the element on the nodes queue
9.   Else
10.          Add null on the result list
11.Return an iterator for the result list
```

A travessia nível-ordem é realizada, visitando todos os nós em cada nível, um nível de cada vez, começando pela raiz. Dada a mesma árvore, uma passagem nível-ordem produz a ordem:

```
1. A B C D E
```

Para resumir todas as travessias, considere a seguinte árvore binária:



Para as várias travessias analisadas: a ordem e que os nós são visitados é a seguinte:

Ordem da Travessia	Ordem em que os nós são visitados
preorder(node) *if node is not null* *visit this node* *preorder(node's left child)* *preorder(node's right child)*	***preorder(root)*** visita os nós na seguinte ordem: A B D E C F G
inorder(node) *if node is not null* *inorder(node's left child)* *visit this node* *inorder(node's right child)*	***inorder(root)*** visita os nós na seguinte ordem: D B E A F C G
postorder(node) *if node is not null* *postorder(node's left child)* *postorder(node's right child)* *visit this node*	***postorder(root)*** visita os nós na seguinte ordem: D E B F G C A
levelorder(node) *if node is not null* *add node to a queue* *while the queue is not empty* *get the node at the head of the queue* *visit this node* *if the node has children* *put them in the queue in left to right order*	***levelorder(root)*** visita os nós na seguinte ordem: A B C D E F G

12.7. BinaryTree ADT

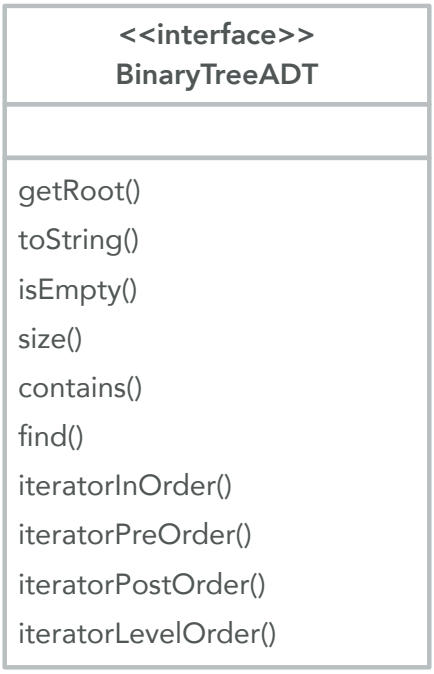
Como exemplo de possíveis implementações das árvores, vamos explorar a implementação de uma árvore binária. Tendo especificado que estamos a implementar uma árvore binária, podemos identificar um conjunto de operações possíveis que seriam comuns para todas as árvores binárias. Note porém, que para além de construtores, nenhuma destas operações adicionam elementos à árvore.

Não é possível definir uma operação para adicionar um elemento à árvore até sabermos mais sobre como a árvore irá ser usada. As operações de uma árvore binária são as seguintes:

Operação	Descrição
`getRoot`	Retorna uma referência à raiz da árvore binária
`isEmpty`	Determina se a árvore está vazia
`size`	Retorna o número de elementos na árvore
`contains`	Determina se o alvo específico está na árvore
`find`	Retorna uma referência ao elemento de destino especificado se for encontrado
`toString`	Retorna uma representação de string da árvore
`iteratorInOrder`	Retorna um iterador para uma travessia do inroder da árvore
`iteratorPreOrder`	Retorna um iterador para uma travessia pré-roder da árvore
`iteratorPostOrder`	Retorna um iterador para uma travessia postroder da árvore
`iteratorLevelOrder`	Retorna um iterador para uma travessia de levelroder da árvore

12.8. Interface BinaryTree

A implementação da Interface `BinaryTreeADT` segue a mesma abordagem como até aqui no entanto não nos podemos esquecer que esta é apenas uma implementação intermédia já que todos os métodos de manipulação da árvore binária são inexistentes da lista de operações. Apesar de ser uma abordagem curiosa acaba por ser a generalização de qualquer árvore binária que poderá ser implementada no futuro. Nesta fase, como não sabemos o seu comportamento, como irá manter os elementos – isto é, a sua organização, não podemos implementar qualquer operação que altera o significado da árvore. Essas operações só poderão ser implementadas na altura em que for tomada essa decisão.




```
1. public interface BinaryTreeADT<T> {
2.     /**
3.      * Returns a reference to the root element
4.      *
5.      * @return a reference to the root
6.      */
7.     public T getRoot ();
8.
9.     /**
10.     * Returns true if this binary tree is empty and false otherwise.
11.     *
12.     * @return  true if this binary tree is empty
13.     */
14.    public boolean isEmpty();
15.
16.    /**
17.     * Returns the number of elements in this binary tree.
18.     *
19.     * @return  the integer number of elements in this tree
20.    */
21.
22.    public int size();
```

```
23.    /**
24.     * Returns true if the binary tree contains an element that
25.     * matches the specified element and false otherwise.
26.     *
27.     * @param targetElement  the element being sought in the tree
28.     * @return true if the tree contains the target element
29.     */
30.    public boolean contains (T targetElement);
31.
32.    /**
33.     * Returns a reference to the specified element if it is found in
34.     * this binary tree.  Throws an exception if the specified element
35.     * is not found.
36.     *
37.     * @param targetElement  the element being sought in the tree
38.     * @return a reference to the specified element
39.     */
40.    public T find (T targetElement);
41.
42.    /**
43.     * Returns the string representation of the binary tree.
44.     *
45.     * @return  a string representation of the binary tree
46.     */
47.    public String toString();
48.
49.    /**
50.     * Performs an inorder traversal on this binary tree by calling an
51.     * overloaded, recursive inorder method that starts with the root.
52.     *
53.     * @return  an iterator over the elements of this binary tree
54.     */
55.    public Iterator<T> iteratorInOrder();
56.
57.    /**
58.     * Performs a preorder traversal on this binary tree by calling an
59.     * overloaded, recursive preorder method that starts
60.     * with the root.
```

```
61.     *
62.     * @return  an iterator over the elements of this binary tree
63.     */
64.    public Iterator<T> iteratorPreOrder();
65.
66.     /**
67.     * Performs a postorder traversal on this binary tree by
68.     * calling an overloaded, recursive postorder
69.     * method that starts with the root.
70.     *
71.     * @return  an iterator over the elements of this binary tree
72.     */
73.    public Iterator<T> iteratorPostOrder();
74.
75.    /**
76.     * Performs a levelorder traversal on the binary tree,
77.     * using a queue.
78.     *
79.     * @return  an iterator over the elements of this binary tree
80.     */
81.    public Iterator<T> iteratorLevelOrder();
82.}
```

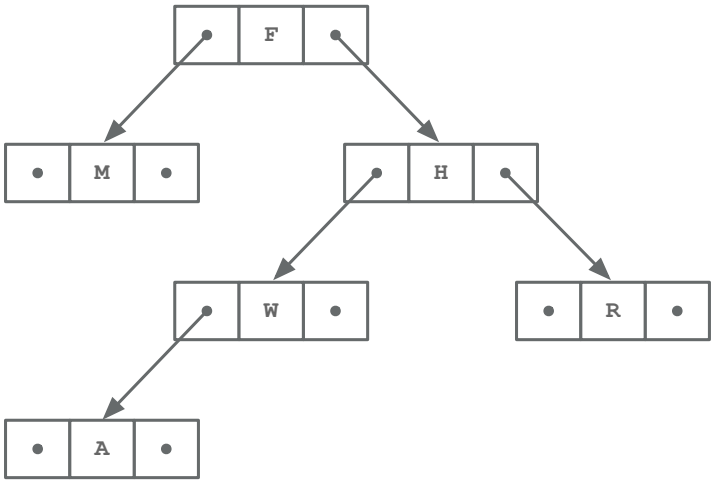
12.9. Implementar uma BinaryTree ADT com recurso a um *array*

Já consideramos como implementar árvores gerais através de um *array*. A implementação contígua é excelente para árvores binárias completas porque não desperdiça espaço em referências e fornece uma maneira rápida e fácil de navegar na árvore. Infelizmente, na maioria das aplicações as árvores binárias estão longe de serem completas, existem muitas posições num *array* que nunca serão usados o que desperdiça muito espaço. Mesmo que as nossas árvores binárias estivessem sempre cheias ainda há o problema de ter que prever o tamanho da árvore com antecedência para que um *array* alocado possa ser grande o suficiente para conter todos os nós da árvore. O *array* pode ser realocada se a árvore ficar muito grande mas essa é também uma operação que tem um custo.

É por esse motivo que não é particularmente útil ter uma implementação contígua de árvores binárias, ou mesmo construir uma como um mecanismo de implementação usado para implementar outras coleções.

12.10. Implementar uma BinaryTree ADT com recurso a uma lista ligada

Uma implementação ligada de árvores binárias assemelha-se à implementações de outros ADTs através de estruturas ligadas que já estudámos. Os nós da árvore binária são representados por objetos de uma classe `BinaryTreeNode` que possui atributos para os dados mantidos no nó e referência às subárvores esquerda e direita. Cada objeto `BinaryTree` contém uma referência do tipo `BinaryTreeNode` ao nó raiz da árvore. As árvores vazias são representadas como referências nulas. A figura seguinte ilustra essa abordagem.



Árvores são estruturas inerentemente recursivas, então é natural escrever muitas operações da classe `BinaryTree` de forma recursiva. Por exemplo, para implementar a operação `size`, a `BinaryTree` pode invocar uma operação `size(r:BinaryTreeNode)` interna no nó raiz. Essa operação retorna zero se o parâmetro for nulo e um mais a soma das chamadas recursivas nas subárvores esquerda e direita do nó passado por parâmetro. Muitas outras operações inclusivamente as travessias podem ser implementadas de forma recursiva com bastante facilidade.

Implementar iteradores é mais desafiador, no entanto o problema é que os iteradores não podem ser implementados recursivamente porque precisam parar sempre um novo nó é visitado de forma a devolver o valor do nó ao cliente. Existem duas maneiras de resolver este problema:

- Implementar uma operação recursiva para copiar os valores do nó para uma estrutura de dados (como uma fila) na ordem correta e, de seguida, extrair itens da estrutura de dados, um de cada vez, conforme o cliente os solicita.
- Não usar recursividade para implementar iteradores: usar uma pilha em vez disso.

A segunda alternativa, embora mais difícil de fazer, é claramente melhor porque ocupa muito menos espaço.

Para a implementação da árvore binária o elemento mais importante é criar a classe que irá representar o nó. Tal como fizemos anteriormente com as listas neste caso em particular temos de criar uma classe `BinaryTreeNode`.

```
1. /**
2.  * BinaryTreeNode represents a node in a binary tree with a left and
3.  * right child.
4.  */
5. public class BinaryTreeNode<T> {
6.
7.    protected T element;
8.    protected BinaryTreeNode<T> left, right;
9.
10.   /**
11.    * Creates a new tree node with the specified data.
12.    *
13.    * @param obj  the element that will become a part of
14.    * the new tree node
15.   */
16.   BinaryTreeNode (T obj) {
17.      element = obj;
18.      left = null;
19.      right = null;
20.   }
```

```
21.
22.    /**
23.     * Returns the number of non-null children of this node.
24.     * This method may be able to be written more efficiently.
25.     *
26.     * @return  the integer number of non-null children of this node
27.     */
28.    public int numChildren() {
29.       int children = 0;
30.       if (left != null) {
31.          children = 1 + left.numChildren();}
32.       if (right != null) {
33.          children = children + 1 + right.numChildren();}
34.       return children;
35.    }
36.}
```

12.11. Notas Finais

A `BinaryTreeADT` descreve operações básicas para implementar e examinar árvores binárias cujos nós contêm valores do tipo T. Uma classe `BinaryTree` podem possuir várias operações que não estão no ADT, em particular, operações para percorrer os nós da árvore e aplicar alguma função aos dados armazenados em cada nó. De realçar que os Iteradores também são disponibilizados por esta classe.

As implementações contíguas da `BinaryTreeADT` são possíveis e úteis em algumas circunstâncias especiais, mas a principal técnica de implementação da `BinaryTreeADT` usa uma representação em estrutura ligada. A recursividade é uma técnica muito útil para implementar a maioria das operações da `BinaryTree`, mas não pode ser usada tão facilmente para implementar iteradores. A classe `BinaryTree` implementada será uma classe muito importante para futuras implementações que ainda iremos fazer.

12.12. Exercícios Propostos

Exercício 1

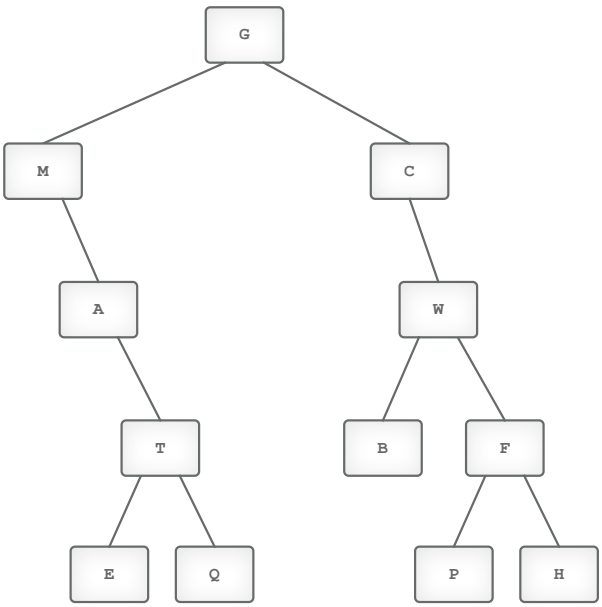
Porquê que a implementação contígua de árvores binárias não é muito útil?

Exercício 2

Qual é a relação entre as classes `LinkedBinaryTree` e `BinaryTreeNode`?

Exercício 3

Escreva os valores dos nós na árvore a seguir na ordem em que são visitados quando a árvore é percorrida em-ordem, em pré-ordem e em pós-ordem.



Exercício 4

Porquê que temos vários iteradores nas implementações de árvore?

Exercício 5

A BinaryTreeADT não possui operações de manipulação a árvore binária, porquê?

Exercício 6

Porquê que o iterador da árvore binária não é implementado normalmente de forma recursiva?

Exercício 7

Comente a implementação da travessia nível-ordem