Árvores

Alessandro

1 Introdução

As estruturas de dados estudadas até agora são caracterizadas como lineares, como vetores e listas. A importância dessas é inegável, mas elas não são adequadas para representar dados que devem ser dispostos de maneira hierárquica. Como por exemplo, os arquivos em um computador são armazenados dentro de uma estrutura hierárquica de diretórios. Existe um diretório base dentro do qual podemos armazenar diversos subdiretórios e arquivos. Por sua vez, dentro deles, pode-se armazenar outros subdiretórios, e assim por diante, recursivamente.

Por conta disso, as árvores são estruturas de dados adequadas para a representação de hierárquias. A forma mais natural de definir uma estrutura de árvore é usando a recursividade. Uma árvore pode ser representada graficamente como mostrado na Figura 1. As linhas que unem dois nós representam os relacionamentos lógicos e as dependências de subordinação existentes entre eles. Na figura pode-se observar que o nó **A** se relaciona somente com os nós **B**, **C**, e **D**, e não com os demais. Por sua vez, o nó **B** se relaciona com o **A** e também com o **E**. Intuitivamente, observa-se que os relacionamentos do nó **B** com **A** e **E** são diferentes − existe uma hierárquia que faz com que o relacionamento de **E** para **B** seja o mesmo de **B** para **A**. A hierárquia de subordinação mostra que um subconjunto de nós é subordinado a outro nó. Por exemplo, o subconjunto de nós **H**, **I** e **J** está subordinado ao nó **D**.

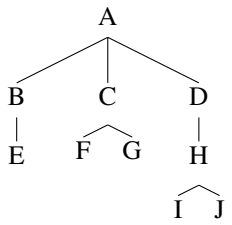

Figura 1: Representação gráfica de uma árvore.

Os relacionamentos de subordinação, formando hierarquias, podem representar diferentes significados, como:

1. hierarquias de especialização, representando classes e subclasses, conforme mostrado na Figura 2, na qual um **veículo** (classe) pode ser especializado em **aéreo**, **terrestre** ou **aquático** (subclasses). Cada uma dessas classes pode, por sua vez, ser especializada em outras categorias;
2. hierarquias de composição, conforme mostrado na Figura 3. Neste exemplo, o nó que representa um carro é composto por três partes: **chassi**, **motor** e **rodas**.

Uma árvore é composta por um conjunto de nós.

2 Definição

Definição Uma árvore é um tipo abstrato de dados que armazena elementos de forma hierárquica. Com exceção do elemento topo, cada elemento da árvore tem um **pai** e zero ou mais elementos **filhos**.

Normalmente, o elemento topo é chamado **raiz** da árvore, mas é desenhado como sendo o elemento mais alto, com todos os demais conectados abaixo (exatamente ao contrário de uma árvore real).

Formalmente define-se uma **árvore** T como um conjunto de **nós** que armazenam elementos em relacionamentos **pai-filho** com as seguintes propriedades:

1. Se T não é vazia, ela tem um nó especial chamado de **raiz** de T que não tem pai.
2. Cada nó v de T diferente da raiz tem um único nó **pai**, w; todo nó com pai w é **filho** de w.

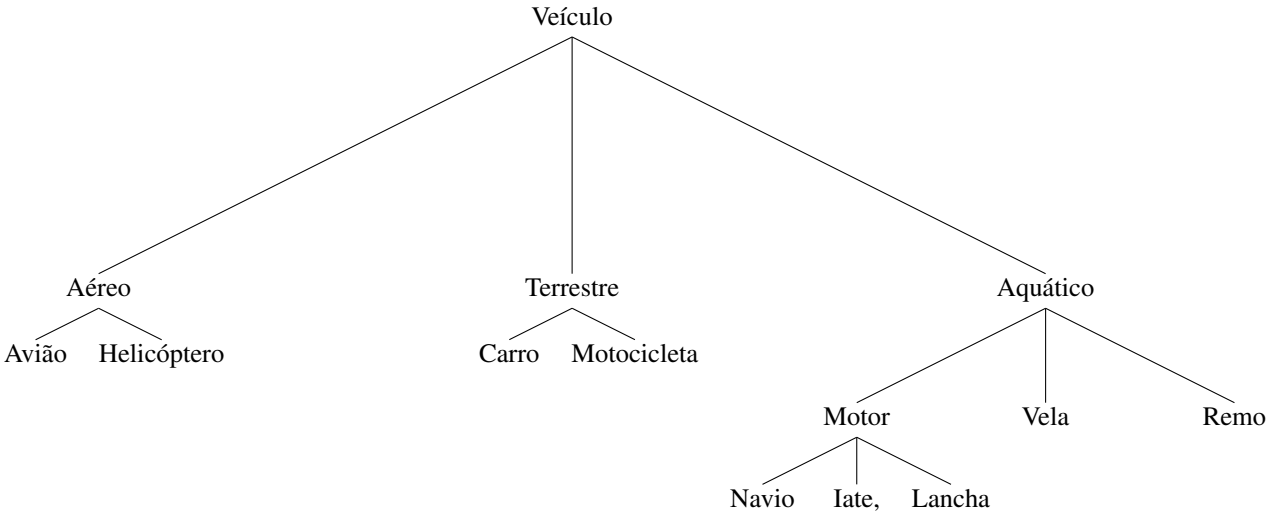

Figura 2: Hierárquia de especialização.

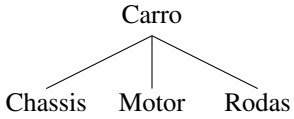

Figura 3: Hierárquia de composição.

Com base nessa definição, uma árvore pode ser vazia, o que significa que ela não tem nós. Isso permite definir uma árvore recursivamente, de maneira que uma árvore T ou está vazia ou consiste em um nó r, chamado raiz de T, e um conjunto de árvores cuja raízes são filhas de r.

2.1 Relacionamento entre os nós

Os relacionamentos de uma árvore caracterizam-se como: Dois nós que são filhos do mesmo pai são **irmãos**. Um nó v é **externo** se v não tem filhos. Um nó v é **interno** se tem um ou mais filhos. Nós externos também são conhecidos como **folhas**.

Um nó u é **ancestral** de um nó v, se $u = v$, ou u é ancestral o pai de v. Da mesma forma, diz-se que um nó v é **descendente** de um nó u se u é ancestral de v. Por exemplo, na Figura 1 C é ancestral de F e I é descendente de H. A **subárvore** de T **enraizada** no nó v é a árvore que consiste em todos os descendentes de v em T (incluindo o próprio v).

2.2 Arestas e Caminhos em Árvores

Uma aresta de uma árvore T é um par de nós (u, v) tal que u é pai de v ou vice-versa. Um **caminho** de T é uma sequência de nós tais que quaisquer dois nó consecutivos da sequência formam uma aresta. Por exemplo, a árvore da Figura 1 contém o caminho (A, D, H, J). Por exemplo, os relacionamento de herança entre classes em programas Java formam uma árvore. A raiz, *java.lang.Object*, é o ancestral de todas as outras classes. Cada classe C é descendente desta raiz e é a raiz de uma subárvore de classes que estendem C. Logo, existe um caminho de C para a raiz, *java.lang.Object*, nessa árvore de herança.

2.3 Árvores ordenadas

Uma árvore é **ordenada** se existe uma ordem linear definida para os filhos de cada nó, ou seja, se é possível identificar os filhos de um nó como sendo o primeiro, segundo, terceiro e assim por diante. Tal ordenação normalmente é desenhada organizando-se os irmãos da esquerda para direita, de acordo com a relação entre os mesmos. Árvores ordenadas normalmente indicam o relacionamento linear existente entre os irmãos, listando-os na ordem correta. Por

exemplo, os componentes de um documento estruturado, tal como um livro, é organizado hierarquicamente como uma árvore cujos nós internos são partes, capítulos e seções, e os nós externos são os parágrafos, tabelas, figuras e assim por diante. A raiz da árvore corresponde ao livro propriamente dito. Esta árvore é um exemplo de uma árvore ordenada porque existe uma ordem bem definida entre os filhos de cada nó.

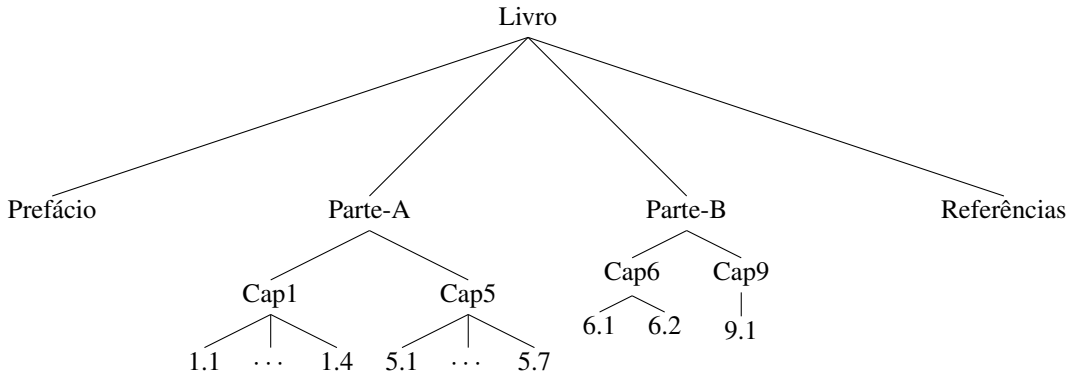

Figura 4: Árvore ordenada associada a um livro.

2.4 Tipo Abstrato de Dados Árvore (TAD)

O Tipo Abstrato de Dados (TAD) árvore armazena elementos em posições como as de uma lista, que são definidas em relação às posições de seus vizinhos. As **posições** de uma árvore são **nós**, e o posicionamento pela vizinhança satisfaz as relações pai-filho, que definem uma árvore válida. Entretanto, os termos "posição" e "nó" são usados com o mesmo sentido no caso de árvore.

Na tabela 1 apresenta as operações válidas para um tipo árvore.

Operação	**Descrição**
element()	Retorna o objeto nessa posição
root()	Retorna a raiz da árvore.
parent(v)	Retorna o nó pai de v.
children(v)	Retorna um conjunto contendo os filhos do nó v.
isInternal(v)	Testa se um nó v é interno.
isExternal(v)	Testa se um nó v é externo.
isRoot(v)	Testa se um nó v é a raiz.
size()	Retorna o número de nós na árvore.
isEmpty()	Testa se uma árvore tem ou não algum nó.
positions()	Retorna um conjunto com todos os nós da árvore.
replace(v,e)	Retorna o elemento armazenado em v e o substitui por e.

Tabela 1: Operações válidas para o tipo árvore

Na Listagem 1, temos a definição de uma interface Java representando uma árvore (Tree).

```
/**
 * Interface para uma árvore onde os nós podem ter uma quantidade arbritária de
 * filhos.
 *
 * @param <E>
 */
public interface Tree<E> {

  /**Retorna a quantidade de nós da árvore.*/
   public int size();
```

```
11    /**Retorna <code>true</code> se a árvore está vazia.*/
      public boolean isEmpty();
13    /**Retorna um iterator sobre os elementos armazenados na árvore. */
      public Iterator<E> iterator();
15    /**Retorna uma coleção iterável dos nós*/
      public Iterable<No<E>> positions();
17    /**Substitui o elemento armazenado em um dado nodo.*/
      public E replace(No<E> v, E e);
19    /** Retorna a raiz da árvore*/
      public No<E> root();
21    /** Retorna o pai de um dado nó. */
      public No<E> parent(No<E> v);
23    /**Retorna uma coleção iterável dos filhos de um dado nó.*/
      public Iterable<No<E>> children(No<E> v);
25    /**Retorna se um dado nó é interno.*/
      public boolean isInternal(No<E> v);
27    /**Retorna se um dado nó é externo.*/
      public boolean isExternal(No<E> v);
29    /**Retorna se um dado nó é a raiz da árvore.*/
      public boolean isRoot(No<E> v);
31  }
```

Listing 1: Interface Java Tree representando o TAD árvore

3 Algoritmos de Percurso em Árvores

3.1 Profundidade

Seja v um nó de uma árvore T. A profundidade de v é o número de **ancestrais** de v excluindo o próprio v. Por exemplo, na árvore da Figura 5, o nó que armazena *Internacional* tem profundidade igual a 2. Observa-se que a profundidade da raiz de T é zero.

A profundidade de um nó v pode ser definida recursivamente como:

- Se v é a raiz, então a profundidade de v é 0.
- Em qualquer outro caso, a profundidade de v é um mais a profundidade do pai de v.

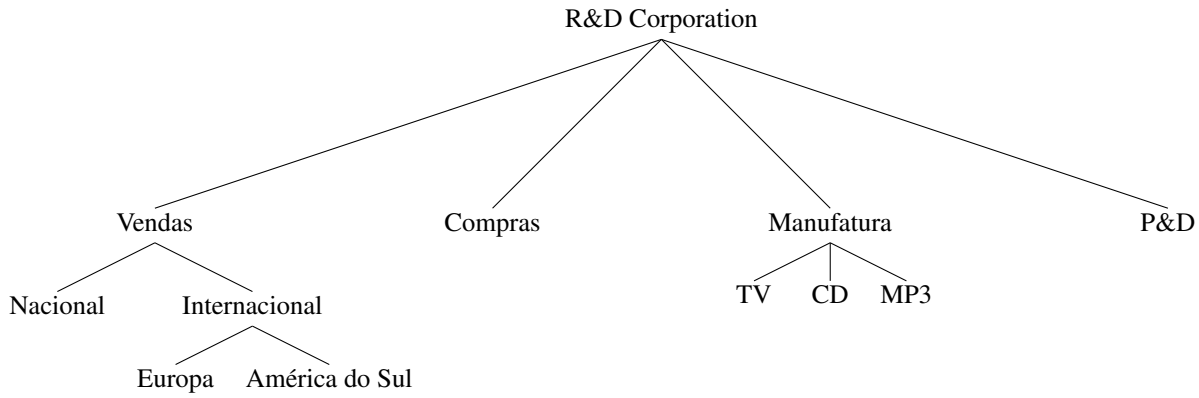

Figura 5: Árvore com 12 nós representando a estrutura organizacional de uma empresa fictícia. Os filhos da raiz armazenam P&D, Venda, Compras e Manufatura.

Baseado na definição, no algoritmo 1 é apresentado um algoritmo recursivo simples, **profundidade**, para calcular a profundidade de um nó T. Este algoritmo chama a si próprio recursivamente sobre o pai de v e acrescenta 1 ao valor retornado. Na listagem 2, temos uma implementação em Java deste algoritmo.

Algoritmo 1: profundidade(T, v)

```
1 se v é a raiz de T então
2     return 0
3 senão
4     return 1 + profundidade(T,w), onde w são os pais de v em T
```

```
1 public static <E> int profundidade(Tree<E> t, No<E> v){
    return (t.isRoot(v)) ? 0 :
3       1 + profundidade(t, t.parent(v));
  }
```

Listing 2: Algoritmo para calcular a profundidade de um nó v de uma árvore T escrito em Java

3.2 Altura

A **altura** de um nó v em uma árvore T é definida recursivamente como:

1. Se v é um nó externo, então a altura de v é 0.
2. Em qualquer outro caso, a altura de v é um mais a altura máxima dos filhos de v.

A **altura** de uma árvore não vazia T é a altura da raiz de T. Por exemplo, a árvore da figura 5 tem altura igual a 4.

A altura de uma árvore não vazia T é igual a profundidade máxima dos nós externos de T.

Algoritmo 2: Altura(T)

```
1 h ← 0
2 para cada vértice v em T faça
3     se v é um nó externo de T então
4         h ← Máximo(h, profundidade(T, v))
5     return h
```

```
  public static <E> int altura(Tree<E> t) {
2   int h = 0;
    for (No<E> v : t.positions()) {
4     h = Math.max(h, profundidade(t, v));
    }
6   return h;
  }
```

Listing 3: Algoritmo para calcular a altura de uma árvore não-vazia T

3.3 Caminhamento Prefixado (Pré-ordem)

O **caminhamento** de uma árvore T é uma forma sistemática de acessar ou "visitar" todos os nós de T.

Em um **caminhamento prefixado** de uma árvore T, a raiz de T é visitada primeira e, então as subárvores, cujas raízes são seus filhos, são percorridas recursivamente. Se a árvore está ordenada, então as subárvores são percorridas de acordo com a ordem dos filhos. A ação específica associada com a "visita" de um nó v depende da aplicação de caminhamento, e pode envolver qualquer coisa, desde incremento de um contador até um cálculo complexo para v.

O algoritmo 3 apresenta o pseudocódigo para o caminhamento prefixado de uma subárvore cuja raiz é o nó v.

Algoritmo 3: preorder(T, v)

```
1 executa a ação de visita para o nó v
2 para cada filho w de v em T faça
3     preorder(T, w)
```

A chamada inicia à esta rotina dar-se-a da seguinte forma:

```
preorder(t, t.root());
```

onde t é uma árvore não-vazia.

O algoritmo de caminhamento préfixado é útil para produzir uma ordenação linear dos nós de uma árvore, na qual os pais devem aparecer antes dos filhos na ordenação.

Exemplo: O caminahmento préfixado de uma árvore associada a um documento examina o documento inteiro, sequencialmente, do início ao fim. Se os nós externos são removidos antes do caminhamento, então o índice do documento é percorrido, conforme mostra a figura 6.

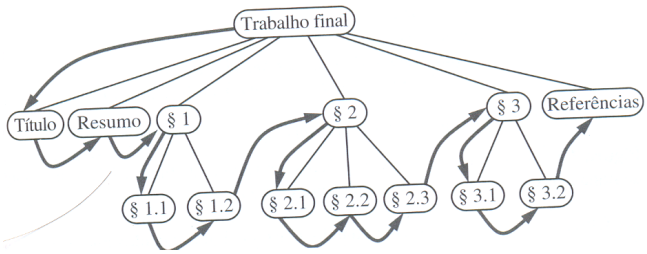

Figura 6: Caminhamento prefixado sobre uma árvore ordenada onde os filhos de cada nó estão ordenados da esquerda para a direita

3.4 Caminhamento Pós-Fixado (Pós-ordem)

Outro tipo importante de caminhamento em árvores é o caminhamento pós-fixado. Este algoritmo pode ser entendido como o oposto do caminhamento prefixado, porque primeiro percorre recursivamente as subárvores enraizadas nos filhos da raizl e depois visita a raiz. É similar ao caminhamento préfixado, entretanto, na medida que usando o mesmo para resolver um determinado problema, especializa-se a ação associada com a "visitação" de um nó v. Ainda, da mesma forma que o caminhamento prefixado, se a árvore for ordenada, as chamadas recursivas nos filhos de um nó v são feitas de acordo com sua ordem específica.

O algoritmo 4 apresenta o pseudocódigo para o caminhamento pós-fixado de uma árvore T cuja raiz é o nó v.

Algoritmo 4: Pós-Fixado(T, v)

```
1 para cada filho w de v em T faça
2     posorder(T, w)
3 executa a ação de visita para o nó v
```

```
public static <E> String imprimirPosfixado(Tree<E> t, No<E> v){
  StringBuilder s = new StringBuilder();
  for(No<E> w : t.children(v)){
    s.append(imprimirPosfixado(t, w) + " ");
  }
  s.append(v.element());
  return s.toString();
}
```

O nome do caminhamento pós-fixado vem do fato de que o caminhamento visitará o nó v depois de ter visitado todos os outros nós da subárvore com raiz em v, conforme nos mostra a Figura 7.

O método de caminhamento pós-fixado é útil para resolver problemas em que se deseja calcular alguma propriedade para cada nó v da árvore, mas o cálculo desta propriedade para v implica que se tenha calculado anteriormente a mesma propriedade para seus filhos. Um exemplo de tal aplicação é ilustrado a seguir.

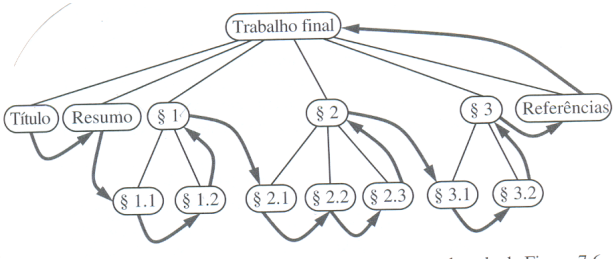

Figura 7: Caminhamento pós-fixado sobre uma árvore ordenada.

Exemplo: Considere-se a árvore T de um sistema de arquivos, cujos nós externos representam arquivos e os nós internos representam diretórios. Supondo-se que se deseja calcular o espaço em disco usado por um diretório, o que é recursivamente definido pela soma do:

1. tamanho do diretório propriamente dito.
2. tamanhos dos arquivos armazenados no diretório.
3. espaço usado pelos diretórios filhos.

Este cálculo pode ser feito com um caminhamento pós-fixado sobre a árvore T. Depois que as subárvores de um nó interno v forem percorridas, calcula-se o espaço usado por v, somando o tamanho do diretório v propriamente dito e o tamanho dos arquivos armazenados no próprio diretório v com o espaço usado por cada filho interno de v, que é calculado pelo caminhamento pós-fixado recursivo dos filhos de v.

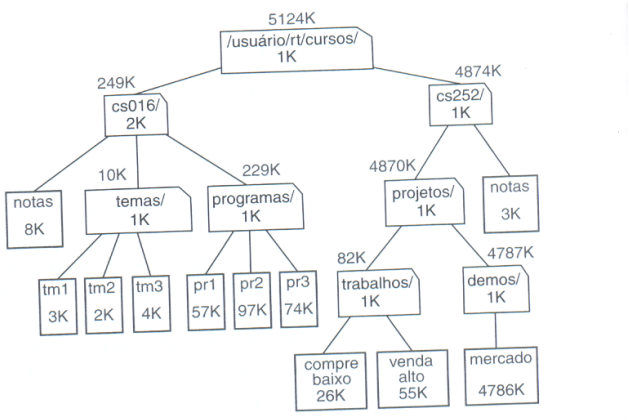

Figura 8: Árvore representando um sistema de arquivos, mostrando o nome o tamanho dos arquivos/diretórios associados a cada nó e o espaço em disco usado para os diretórios associados a cada nó interno

.