Grafos e Árvores

Alguns conceitos e representações

Prof. Leônidas de Oliveira Brandão
Departamento de Ciência da Computação - IME - USP
http://www.ime.usp.br/~leo

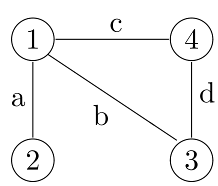
1.1 Grafos

$G = (\texttt{AG}, \texttt{VG})$ $\quad \texttt{AG}=$ conjunto de arestas
$\qquad\qquad\qquad\ \ \texttt{VG}=$ conjunto de vértices (ou nós)

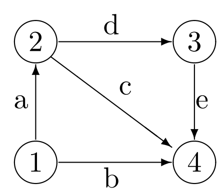
$(u, v) \in \texttt{AG}$ é uma aresta de G ligando u a v (ou v a u).

Se existir ordem em $\texttt{AG}$, isto é, $(u, v) \in \texttt{AG} \not\Rightarrow (v, u) \in \texttt{AG}$ o grafo é dito **orientado** ou **dirigido**.

- Representação gráfica



G_1 Grafo não orientado



G_2 Grafo orientado

$$\begin{array}{ll} \texttt{AG}_1 = (\text{a,b,c,d}) & \text{a} = (2, 1) \text{ ou} \\ \texttt{VG}_1 = (1, 2, 3, 4) & \text{a} = (1, 2) \end{array}$$

- Representação computacional
  - Matriz de adjacência ($\texttt{VG} \times \texttt{VG}$)

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Matriz Adj. para G_1

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Matriz Adj. para G_2

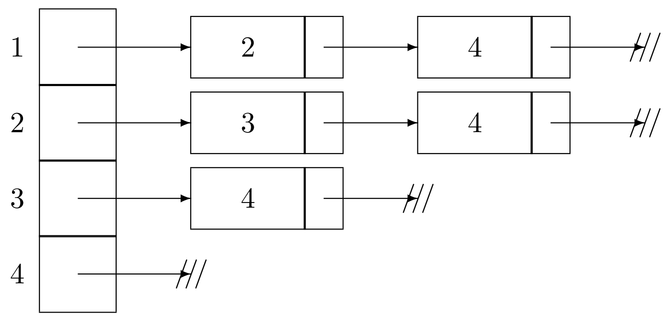
  - Matriz de incidência

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} -1 & -1 & 0 & 0 & 0 \\ 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

se (u,l)=-1 e (v,l)=1, então l=(u,v)

– Lista de adjacência



Lista de adj. para G_2
usando listas ligadas

1.1.1 Conceitos básicos

Subgrafo Um subgrafo G' de um grafo G ($G' \subseteq G$) é qualquer grafo tal que $VG' \subseteq VG$, $AG' \subseteq AG$ e $(v, w) \in AG' \Rightarrow (v, w) \in VG' \times VG'$ (ou seja, é qualquer grafo definido a partir de vértices e arestas de G)

Passeio É qualquer sequência $(s_0, s_1, \ldots, s_k)$, onde $(s_i, s_{i+1}) \in \mathtt{AG}, 0 \leq i < k$. Dizemos que k é o tamanho do passeio.

Caminho É qualquer passeio no qual nenhum vértices ocorre mais que uma vez.

Componente conexa dado $G' \subseteq G$, G' é componente conexa de G se

$$\forall (u, v) \in \mathtt{VG}' \times \mathtt{VG}'$$

existe algum caminho algum caminho v para u. Algumas vezes dizemos apenas *componente*.

Por exemplo, no grafo G_1, anterior, os sub-grafos formados pelos vértice $\{1, 3, 4\}$ e $\{1, 2\}$ (e as arestas entre eles em G) são componentes.

Da definição podemos concluir que cada vértice, isoladamente, é uma componete.

Grafo conexo G é conexo se para todos $u \in \mathtt{VG}$ e $v \in \mathtt{VG}$ existe caminho de u a v (ou de v a u).

Das duas últimas definições, segue que: se uma componente de um grafo G contém todos seus vértice, então G é conexo.

Para um grafo dirigido G, podemos falar em

Ordenação Topológica uma sequência de vértices $(s_1, s_2, \ldots, s_n)$ de G, $n = \mid \mathtt{VG} \mid$, é uma ordenação topológica de G se e só se $\forall s_i$ e s_{i+k} ($i \in \{1, 2, \ldots, n\}$ e $k \geq 0$) $\Longrightarrow (s_{i+k}, s_i) \notin \mathtt{AG}$.

Exemplo: grafo definido pela dependência entre arquivos de programas, $(s_i, s_j) \in \mathtt{AG} \Leftrightarrow$ programa s_j depende do s_i, para se compilar é necessário seguir uma ordem topológica.

Um primeiro resultado sobre grafos, segue sem demonstração

Teorema Um grafo dirigido G admite ordenação topológica se, e somente se, G não contiver componentes conexas não triviais (um só vértice).

Um grafo que não contém componentes não triviais é dito **acíclico**, em caso contrário, **cíclico**.

Existem duas técnicas de busca em grafos:

- Busca em profundidade ("Depth First Search")

```
DFS(w) /* busca em profundidade a partir do nó w */
     visitado(w) ← 1;
     para cada vertice v ∈ Adj(w)
           se  visitado(v)=0 entao BFS(v)
/* aqui existe uma pilha implicita devido a recursao */
```

- Busca em Largura ("Breadth First Search")

```
BFS(w) /* busca em largura a partir do nó w */
     visitado(w) ← 1;
     inicializa fila Q com w;
     enquanto verdade /* laco infinito */
                 w ← retire último da fila Q;
                 para cada vertice v ∈ Adj(w)
                       se  visitado(v)=0 entao Insere v em Q /* final da fila */
                                                visitado(v) ← 1;
                 se  fila Q vazia entao retorne;
```

Outro conceito muito importante à várias aplicações que podem ser modeladas via grafos é o seguinte

Grau de um nó é o número de arestas incidentes ao nó.

Exemplos Denotamos por $g_G(v)$ o grau do nó v no grafo G, em não havendo duvidas a respeito de qual grafo se tratar, denotamos apenas por $g(v)$.

Em $G_1 : g_{G_1}(1) = 3,\ g_{G_1}(2) = 1,\ g_{G_1}(3) = 2$ e $g_{G_1}(4) = 2$.

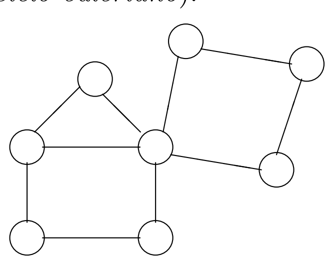
Em $G_2 : g_{G_2}(1) = 0,\ g_{G_2}(2) = 1,\ g_{G_2}(3) = 1$ e $g_{G_2}(4) = 3$.

NOTA: Para todo grafo G, vale a seguinte propriedade

$$\sum_{v \in \mathtt{VG}} g(v) = 2 \mid \mathtt{AG} \mid .$$

Problema do Circuito Hamiltoniano: dada uma representação gráfica de um grafo, determinar, se existir, um caminho que passe por todos os vértices, sem repetir vértice, começando e terminando no mesmo vértice (*ciclo euleriano*).

Problema do Circuito Euleriano: dada uma representação gráfica de um grafo, determinar, se existir, um caminho que passe por todas as arestas, sem repetir aresta, começando e terminando no mesmo vértice (*ciclo euleriano*).



Não existe ciclo euleriano para este grafo.

Problema da Árvore Espalhada Mínima: dado um grafo conexo G, com custos $c(.)$ nas arestas, encontrar uma **árvore espalhada** H ($VH = VG$, H árvore e H conexa) de custo mínimo (i.é., $\forall \overline{H}$ árvore espalhada $\Rightarrow c(\overline{H}) \geqq c(H)$).

Problema Código de Huffman: dado um conjunto de letras $\mathcal{L}$ e suas frequências $(f(l_i), l_i \in \mathcal{L})$, encontrar uma árvore de codificação para $\mathcal{L}$ de modo que

$$\sum_{l_i \in \mathcal{L}} f(l_i) alt(l_i)$$

seja mínimo.

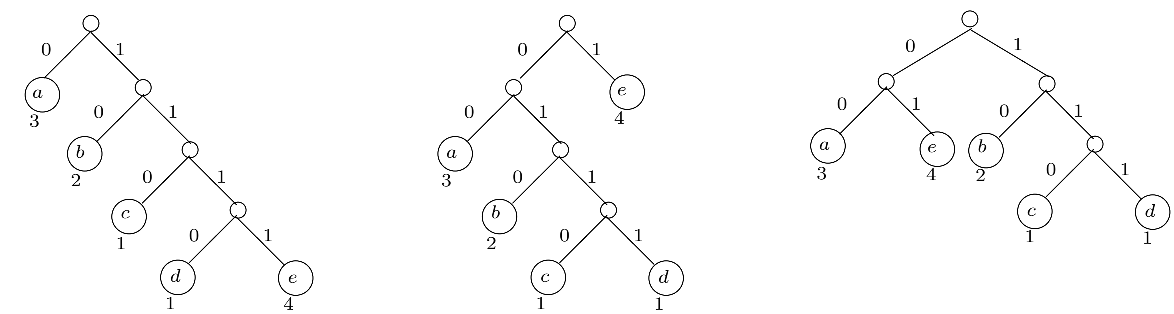
Uma árvore de Huffman é qualquer codificação de caracteres como acima, servindo portanto para codificarmos textos sem que precisemos olhar as palavras formadas.

Exemplo 1.1 *Se os caracteres* (a, b, c, d, e) *aparecem em um texto com frequências* $(3, 2, 1, 1, 4)$*, podemos codificá-las de mais de uma maneira. Por exemplo, podemos usar o código:*

(a) com o código $(0, 10, 110, 1110, 1111)$*, que precisa de* $1 \times 3 + 2 \times 2 + 3 \times 1 + 4 \times 1 + 4 \times 4 = 30$ *bits.*

(b) $(00, 010, 0110, 0111, 1)$*, que tem tamanho (custo)* $2 \times 3 + 3 \times 2 + 4 \times 1 + 4 \times 1 + 1 \times 4 = 24$ *bits; ou ainda*

(c) com o código $(00, 10, 110, 111, 01)$*, com* $2 \times 3 + 3 \times 2 + 4 \times 1 + 4 \times 1 + 1 \times 4 = 24$ *bits;*



(a) $(a, b, c, d, e) \equiv (0, 10, 110, 1110, 1111)$ (b) $(a, b, c, d, e) \equiv (00, 010, 0110, 0111, 1)$ (c) $(a, b, c, d, e) \equiv (00, 10, 110, 111, 01)$

Três possíveis árvores de Huffman para os caracteres (a, b, c, d, e), sendo (b) e (c) ótimas.

Sendo,

- **árvore de codificação para $\mathcal{L}$:** uma árvore binária, cujas folhas representam as letras $\mathcal{L}$ (p.e., se o caminho da raíz até a folha l_i for *esquerda*, *direita*, *direita*, *esquerda* e representarmos *esquerda* por 0 e *direita* por 1, l_i será codificada como 0110).
- $alt(l_i)$**:** é altura, desde a raíz da árvore de codificação, da folha contendo a letra l_i.

Algoritmo 1.1 *Constrói a árvore de codificação de menor custo (código de Huffman)*

Huffman(L, f) / L=($l_1, l_2, \ldots, l_n$), f(.) função de frequências */*
var C, n_conj;
inicio

```
1  C := monta_conjunto(L); // monta conjunto ordenando de modo crescente em relação às frequências
2  n_conj := #L;
3  enquanto ( n_conj > 0 )
4            u ← menorValor(C); // C ← C - u
5            v ← menorValor(C); // C ← C - v
6            w ← novo_no( f(u) + f(v) );
7            w.no_esq ← u;
8            w.no_dir ← v;
9            insereOrdenado(C,w); // insere w em C de modo ordenado - pode ser "heap"
10           n_conj ← n_conj -1;
final
```

Prova de que algoritmo de Huffman produz a árvore de codificação de menor custo (por contradição).

Vamos supor existir uma árvore $\overline{H}$ diferente da de Huffman H, donde extrairemos uma contradição.

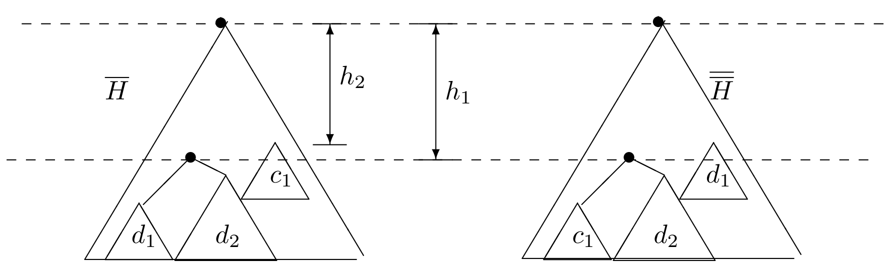
Assim, seja $\overline{H}$ uma árvore estritamente melhor que a de Huffman. Se considerarmos uma ordem de junção que escolhe sempre as sub-árvores que ficarão mais profundas na árvere final, então cada árvore de código admite uma única sequência de construção (junções entre nós para formá-la, das folhas para a raíz, como os passos 7, 8 e 9), do mesmo modo que a árvore de Huffman também é única.

Consideraremos a sequência do algoritmo acima para montar a árvore de Huffman e uma sequência correspondente, juntando-se sempre que possível as sub-árvores de menor frequência (as primeiras escolhidas ficarão em níveis maiores da árvore final).

Deste modo, sejam $(c_1, c2)$ e (d_1, d_2), respectivamente, as primeira junções de sub-árvores diferentes nas sequências de junções de Huffman e da $\overline{H}$. Por *diferente* devemos entender as sub-árvores com custo (ou frequências) diferentes, pois é isso que importa para definir custo final (i.é., se as letras l_i e l_j tiverem a mesma frequência, é indiferente ter uma árvores com estas letras em profundidades, respectivamente, h_i e h_j ou em profundidades h_j e h_i).

Vamos agora definir uma árvore $\overline{\overline{H}}$, construida a partir de $\overline{H}$, trocando-se d_1 por c_1 (e vice-versa).

Sejam h_1 e h_2, respectivamente, as alturas das sub-árvores (d_1, d_2) e c_1 em $\overline{H}$. Portanto, a altura de c_1 em $\overline{\overline{H}}$ será h_1 e a de (d_1, d_2) será h_2[1].



Árvore $\overline{H}$ com sua sub-árvore (d_1, d_2) de altura h_1 e a subárvore substitura (c_1, d_2) de altura h_1 em $\overline{\overline{H}}$.

[1]Como estamos supondo a sequência de construção de $\overline{H}$ o mais parecida possível com H, e uma vez que, uma mesma sub-árvore c esteja presente no passo k de construção de H e $\overline{H}$, então sempre existirá um nó que tenha como filho c (e claro, ambas as árvores H e $\overline{H}$ têm um nó cujo filho é a sub-árvore c).

Como o algoritmo de Huffman não escolheu as sub-árvores d_1 e d_2, escolhendo c_1 e c_2 segue que

$$\begin{array}{l} custo(c_1) < custo(d_1) \\ h_1 > h_2. \end{array}$$

As últimas desigualdades (estritas) seguem da hipótese de $\overline{H}$ ser estritamente melhor que H (e termos pego uma ordem de construção que une primeiro as sub-árvores que ficarão mais profundas na árvore final).

Daí podemos concluir que o custo da árvore $\overline{\overline{H}}$ é estritamente menor que o de $\overline{H}$, pois

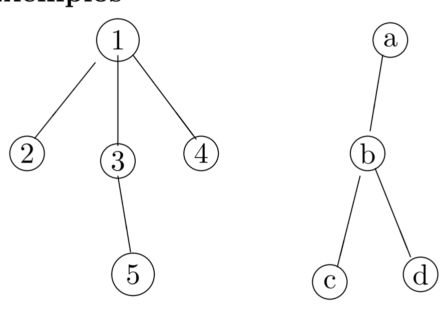
$$\begin{array}{rcl} custo(\overline{\overline{H}}) & = & custo(\overline{H}) \;+\; custo(c_1) + h_1 \sum_{i \in c_1} f_i + custo(d_1) + h_2 \sum_{i \in d_1} f_i \\ & & \qquad\quad -\; (custo(d_1) + h_1 \sum_{i \in d_1} f_i) - (custo(c_1) + h_2 \sum_{i \in c_1} f_i) \\ & = & custo(\overline{H}) \;+\; (h_1 - h_2)(\sum_{i \in c_1} f_i) - \sum_{i \in d_1} f_i)) \\ & < & custo(\overline{H}). \end{array}$$

Portanto não pode existir uma árvore $\overline{H}$ melhor que a H obtida pelo algoritmo de Huffman. ∎

1.2 Árvores

Todo grafo acíclico é uma **árvore**. Podemos "olhar" uma árvore como sendo uma estrutura de hereditariedade, isto é, existe um primeiro nó que denominamos **raiz** e a cada nó da árvore podem "descender" filhos.

Exemplos



H_1 $\qquad$ H_2

Raiz de H_1 é 1
de H_2 é a

$H_2 \begin{cases} \text{c e d são } \textit{filhos} \text{ de b} \\ \text{a é } \textit{pai} \text{ de b} \end{cases}$

Sobre as árvores, temos os seguintes conceitos principais

Nível as duas árvores H_1 e H_2 têm 3 níveis, em H_1 o primeiro nível é composto pelo nó $\{1\}$, o segundo por $\{2, 3, 4\}$ e o terceiro por $\{5\}$.

Profundidade de um nó u é o comprimento do caminho da raíz até u.

Se u é raiz a profundidade de u é 1 (ou 0),
caso contrário é a profundidade do pai de u + 1 (note a recursão!).

Folha é um nó sem filhos.

Altura de um nó é o comprimento máximo do nó à uma folha. Por exemplo, em H_1:

alt(1)=2 $\qquad$ alt(2)=0 $\qquad$ alt(3)=1.

Árvores binárias são árvores onde cada nó tem no máximo dois filhos. Quando existir filho será diferenciado por esquerdo ou direito.

Podemos formalizar algumas destas definições, e consequências, de modo mais rigoroso através de recorrências. Para isso, vamos adotar a convenção matemática de que *somatório indexado por conjunto vazio resulta* 0, isto é,

$$\sum_{\alpha \in \emptyset} \Phi(\alpha) = 0.$$

Assim, dada uma árvore H:

Número de nós de H: se $nn(v)$ é o número de nós a partir do nó v, então $nn(v) = 1 + \sum_{w \in Adj(v)} nn(w)$.

Número de folhas em H: se $nf(v)$ é o número de folhas a partir do nó v, então $nf(v) = \sum_{w \in Adj(v)} nf(w)$.

Altura de H: se $alt(v)$ é a altura do nó v, então $alt(v) = 1 + \max\left\{ \ alt(filho\_esq(v)), filho\_dir(v) \ \right\}$.

Exercício 1.1 *Implemente em alguma linguagem de alto nível os três algoritmos acima, supondo dada a árvore.*

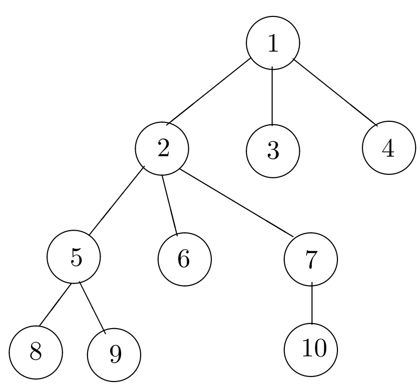
1.2.1 Representações de Árvores

■ Utilizando listas

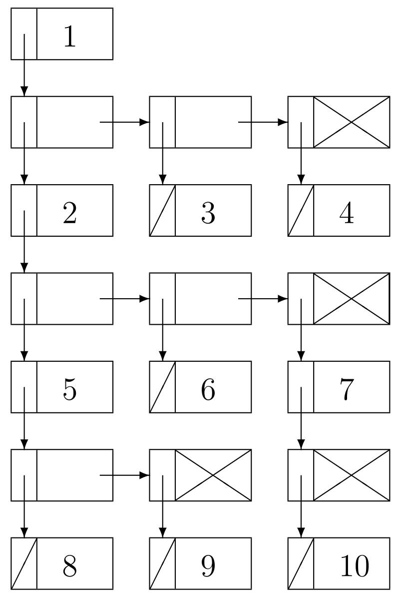
```
( 1 ( 2 ( 5 (8) (9) )
        ( 6 )
        ( 7 (10) )
    ) (3) (4)
)
```

Listas Generalizadas (LISP)

■ Utilizando diagrama hierárquico

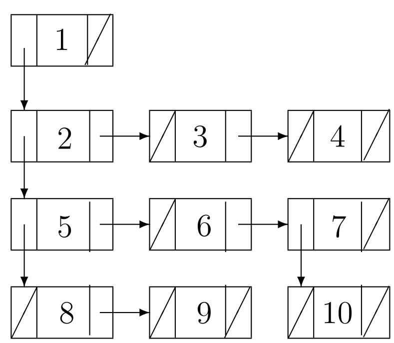


■ Utilizando listas ligadas



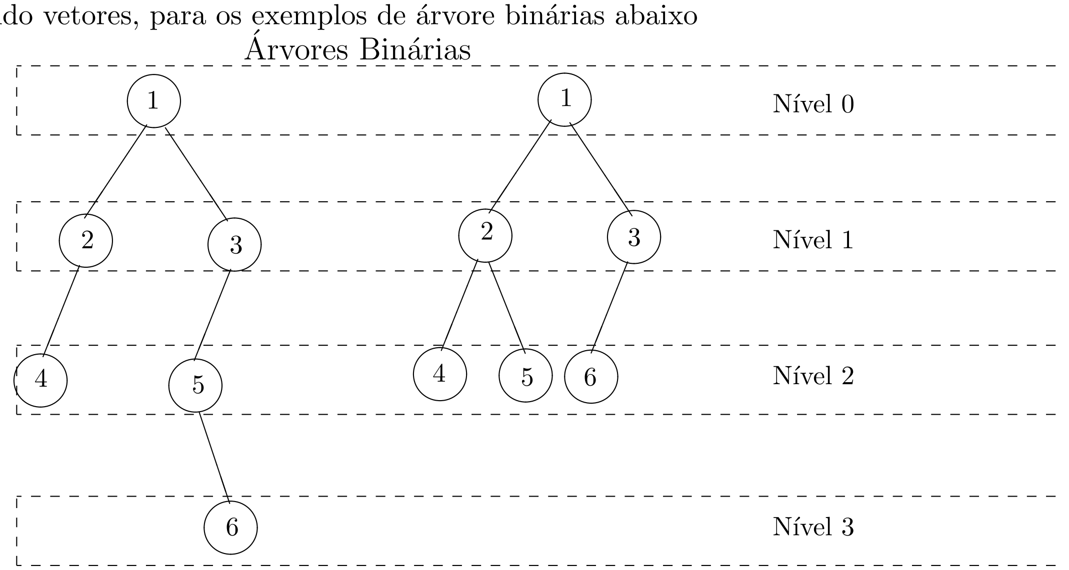
Representação interna de árvores: usando listas ligadas com nós especiais para filhos

■ Utilizando árvores binárias

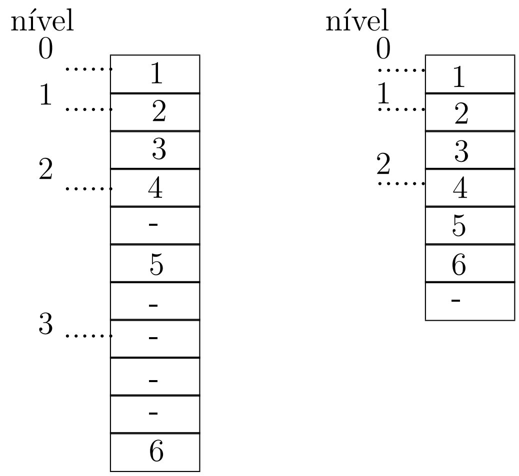


Representação interna de árvores: usando listas ligadas com campos direito e esquerdo

■ Utilizando vetores, para os exemplos de árvore binárias abaixo



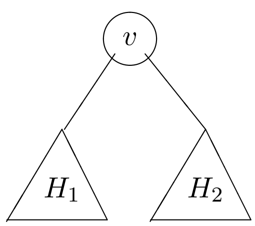
Também é possível adotar uma representação simples (e ineficiente se as árvore binária for esparsa), via vetores. Para isso usaremos o índice do vetor para indicar as relações de sucessão: a raíz tem o índice 0 e dado um nó de índice i, seus filhos terão índices $2*i+1$ e $2*i+2$, ou seja, o filho esquerdo do i está no índice $2*i+1$ e seu filho direito em $2*i+2$. Além disso será necessário reservar um símbolo para indicar nó vazio, como na figura abaixo. Esta é mesma estrutura adotada no algoritmo **Heap Sort**.



Representação interna das árvores binárias da figura anterior usando vetores

1.2.2 Alguns resultados sobre árvore binárias

Dizemos que uma árvore é **completamente balanceada** se para todo nó v, $|\,\mathtt{VH}_e\,|=|\,\mathtt{VH}_d\,|$, sendo H_e é a sub-árvore esquerda de v e H_d a direita.



Teorema H_1: O número de nós em uma árvore completamente balanceada é $2^{h+1}-1$, se h for a altura da árvore.

Prova: Por indução na altura h de da árvore.

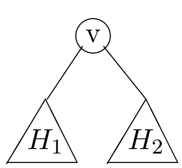
- Base da indução

h=1, então a árvore é trivial (um só nó) e portanto

$$1 = |\,\mathtt{VH}\,| = 2^{0+1}-1.$$

- Passo da indução

Vamos supor que o teorema seja válido para alturas menores que h+1. Podemos decompor a árvore H como segue,



portanto, o número de nós de H é 1 + a soma dos nós das sub-árvore esquerda e direita, nas quais podemos aplicar a hipótese de indução (pois, cada uma delas têm menos que $h+1$ nós), ou seja,

$$\mid \texttt{VH} \mid = \mid \texttt{VH}_e \mid + \mid \texttt{VH}_d \mid +1 \overset{h.i.}{=} 2^h - 1 + 2^h - 1 + 1 = 2^{h+1} - 1.$$

■

Teorema H_2 : Seja H uma árvore binária com n_i nós de grau i (claramente $i \in \{0,1,2\}$), então $n_0 = n_2+1$.

Prova: Cada nó, exceto a raiz, é "final" de uma (e só uma) aresta (seu "superior hierárquico"), logo

$$\mid \texttt{AH} \mid = \mid \texttt{VH} \mid -1 \text{ e } \mid \texttt{AH} \mid = n_1 + 2n_2.$$

Como $\mid \texttt{VH} \mid = n_0 + n_1 + n_2$, segue que

$$n_1 + 2n_2 = \mid \texttt{VH} \mid -1 = n_0 + n_1 + n_2 - 1 \Longleftrightarrow n_0 = n_2 + 1.$$

■

Análise de complexidade para um algoritmo recursivo

Algoritmo recursivo para determinar máximo e mínimo numa lista

Algoritmo 1.2
MaxMin(L,n) / L=($l_1, l_2, \ldots, l_n$) */*
var Max1, Max2, Min1, Min2, k;
inicio
    se $n = 1$ *entao retorne* (l_1, l_1)
    senao se $n = 2$ *entao*
        se $l_1 > l_2$ *entao retorne* (l_1, l_2)
            senao retorne (l_2, l_1)
      senao $k \leftarrow \lfloor n/2 \rfloor$;
        / neste pto usa-se indução: supõe-se alg. funcionar p/ até* $\lceil n/2 \rceil$ *elementos */*
        (Max1,Min1) $\leftarrow$ *MaxMin(*$l_1, \ldots, l_k, k$*);*
        (Max2,Min1) $\leftarrow$ *MaxMin(*$l_{k+1}, \ldots, l_n, n-k$*);*
        retorne (Maximo{Max1,Max2},Minimo{Min1,Min2});
final

Complexidade do algoritmo:

Seja $T(n)$ o número de comparações para listas com $\mid L \mid = n$. Para listas de tamanho 1 e 2 é trivial

$T(1) = 0$

$T(2) = 1$

Para $n > 2$, aplicamos um raciocício indutivo

$T(n) = T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + 2 = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2$

Para simplificarmos a contabilidade vamos admitir que $n = 2^k$,

$$\begin{aligned} T(n) &= 2T(n/2) + 2 = 2(2T(n/2^2) + 2 + 2 = \dots \\ &= 2^{k-1}T(2) + (2 + 2^2 + \dots + 2^{k-1}) = 2^{k-1} + 2^k - 2 = 2^{k-1}(3) - 2 \\ &= 3\frac{n}{2} - 2. \end{aligned}$$

1.3 Passeios em árvores binárias

Podemos percorrer os nós de uma árvore binária de três forma diferentes: visitando um nó depois seu ramo esquerdo e por último seu ramo direito; visitando o ramo esquerdo de um nó, depois o nó e por último seu ramo direito; visitando o ramo esquerdo de um nó, depois seu ramo direito e por último o nó.

1.3.1 Passeio "pré-ordem"

Seja T um árvore binária, o passeio "pré-ordem" de T pode ser feito da seguinte maneira:

1. inicia o passeio pela raíz da árvore, $no \leftarrow raiz(T)$;
2. para todo $no \in VT$, $visite(no)$, $pre\_ordem(no.esq)$ e por último $pre_ordem(no.dir)$.

ou seja,

Algoritmo 1.3 *Passeio "em pré ordem" (pré-ordem) em uma árvore de raíz no.*

```
Pre_ordem(No no)
inicio
    se      no vazio entao retorne ;
    senao   visite(no);
            Pre_ordem(no.esq);
            Pre_ordem(no.dir);
final
```

1.3.2 Passeio "in-ordem"

Seja T um árvore binária, o passeio "in-ordem" de T pode ser feito da seguinte maneira:

1. sendo no a raíz da árvore, comeęe com $in\_ordem(no)$;
2. para todo $no \in VT$, $in\_ordem(no.esq)$, $visite(no)$ e por último $in\_ordem(no.dir)$.

Refinando esta idéia podemos obter o seguinte algoritmo,

Algoritmo 1.4 *Passeio "em ordem" (in-ordem) em uma árvore de raíz no.*

Pre_ordem(No no)
<u>*inicio*</u>
 <u>*se*</u> *no vazio* <u>*entao*</u> <u>*retorne*</u> *;*
 <u>*senao*</u> *In_ordem(no.esq);*
  visite(no);
  In_ordem(no.dir);
<u>*final*</u>

1.3.3 Passeio "pós-ordem"

Seja T um árvore binária, o passeio "pós-ordem" de T pode ser feito da seguinte maneira:

1. sendo *no* a raíz da árvore, começe com *pos_ordem*(*no*);
2. para todo $no \in VT$, *pos_ordem*(*no.esq*), *pos_ordem*(*no.dir*) e por último *visite*(*no*),

ou seja,

Algoritmo 1.5 *Passeio "em pós ordem" (pós-ordem) em uma árvore de raíz no.*

Pre_ordem(No no)
<u>*inicio*</u>
 <u>*se*</u> *no vazio* <u>*entao*</u> <u>*retorne*</u> *;*
 <u>*senao*</u> *Pos_ordem(no.esq);*
  Pos_ordem(no.dir);
  visite(no);
<u>*final*</u>

1.3.4 Passeios em árvores e expressões aritméticas

Uma aplicação elementar de árvores binárias é na representação de expressões aritméticas. A forma mais usual de definirmos uma expressão aritmética é através de uma "regra de produção", por exemplo,

$$
\begin{array}{rcll}
EXPR & := & NUMERO; & (1)\\
 & & \text{-}EXPR; & (2)\\
 & & (EXPR); & (3)\\
 & & EXPR * EXPR; & (4)\\
 & & EXPR/EXPR; & (5)\\
 & & EXPR + EXPR; & (6)\\
 & & EXPR - EXPR; & (7)
\end{array}
$$

e esta representação é naturalmente representada por uma árvore binária. Para isso basta convencionar que os átomos (itens sintáticos $-$, $+$, ..., e os identificadores e constantes) sejam nós com filho esquerdo dado pelo termo à sua esquerda na definição e com filho direito dado pelo termos à sua direita (isso implica em apontador nula à esquerda do operador unário $-$).

Notem que as "produções" (4) a (7) (e as demais com operadores binários) podem produzir ambiguidades, pois o que deve ser calculado antes, a expressão à esquerda do operador ou àquela à sua direita ?

Escolher fazer primeiro a esquerda ou a direita, pode implicar resultados direntes (como em $2 + 3/4$ fazendo-se primeiro $2+3$ ou primeiro $3/4$). Em matemática eliminamos esta ambiguidade associando prioridades aos operadores ($prior(-\text{unário}) > prior(*, /) > prior(+, -)$) e, quando houver empate calculamos da esquerda para direita (os operadores com mesma prioridade são associativos: $a + (b - c) = (a + b) - c$ ou $a * (b/c) = (a * b)/c$).

Exemplo 1.2 *Utilizando as regras matemáticas de prioridades entre operadores, as expressões aritméticas* $A + (2 + 3 * 4)/10$ *e* $A + 2 + 3 * 4/10$ *são resentadas pelas árvores abaixo.*

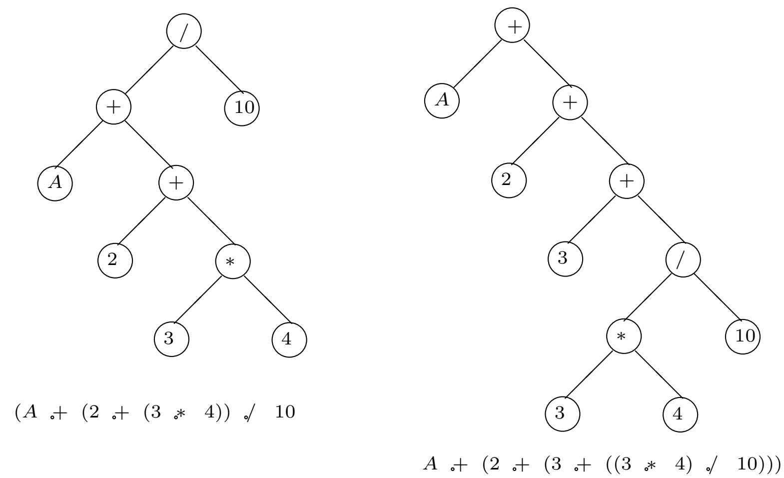


Figura 1.1: Árvores binárias de expressões ambiguas

Propriedade 1.1 *Sendo* $(i_1, i_2, \ldots, i_k)$ *e* $(p_1, p_2, \ldots, p_k)$ *permutações sobre os conjuntos* $(l_1, l_2, \ldots, l_k)$, *então existe uma e só uma árvore binária* T *de tal forma que:*

$$In\_ordem(T) = (i_1, i_2, \ldots, i_k) \quad e \; Pre\_ordem(T) = (p_1, p_2, \ldots, p_k).$$

Propriedade 1.2 *A propriedade 1.1 vale tomando-se qualquer combinação disjunta de passeios (in e pós, in e pré, pós e pré).*

Exemplo 1.3 *Sendo* I *e* P *os passeios, respectivamente, in-ordem e pré-ordem obtido de uma dada árvore, reconstruir a mesma:*

(1) $I = (2, 4, 1, 3, 5)$ *e* $P = (1, 5, 3, 2, 4)$;
(2) $I = (2, 4, 5, 3, 1)$ *e* $P = (1, 2, 3, 4, 5)$;
(3) $I = (1, 3, 5, 4, 2)$ *e* $P = (1, 5, 3, 2, 4)$.

1.3.5 Árvores costuradas

É possível implementar algoritmos não recursivo, e que não usem pilhas explícitas, para fazer passeios em árvores binárias. Nesta sub-seção consideraremos apenas o caso de passeio "em ordem".

Num passeio "em ordem" a visita a um nó ($visita(no)$) é feita após verificarmos toda sua sub-árvore esquerda e o último nó visitado (antes de no), de no, não tem filho direito. Assim, podemos aproveitar os apontadores vazios para filhos direitos para "costurá-los" aos seus sucessores "em ordem".

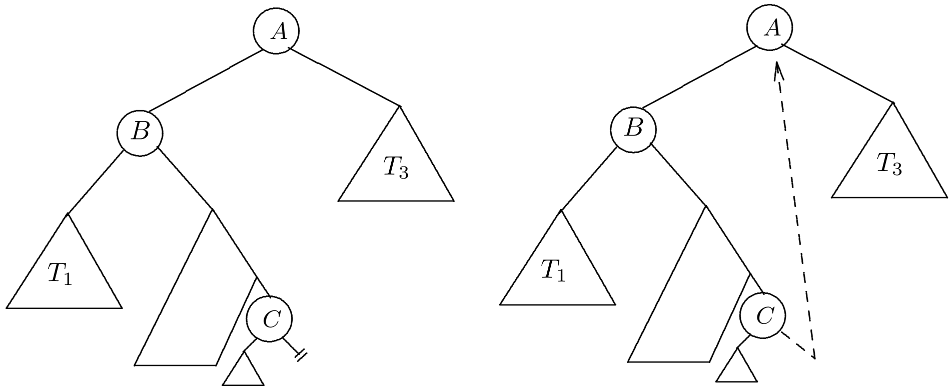


Figura 1.2: Árvores binária sem costura e com costura: nó A é sucessor "em ordem" de C

Usando este idéia é necessário diferenciar um "legítimo" filho direito de uma costura, e para isso basta uma variável booleana. Um algoritmo de passeio "em ordem" em uma árvore costurada é apresentado a seguir.

Algoritmo 1.6 *Passeio "em ordem" em uma árvore costurada de raíz no.*

```
In_ordem_costurado(No no)
inicio
    enquanto no ≠ vazio
        enquanto   no.esq ≠ vazio
                   no ← no.esq;
        visita(no);
        enquanto   no é costurado // percorre toda costura até um nó com legítimo filho direito
                   no ← no.dir; // pega o sucessor do nó atual
                   visita(no);     // visita o sucessor do último nó costurado
        no ← no.dir; // pega o filho direito do nó atual e retorne ao "passo inicial"
final
```

1.3.6 Árvores de Busca Binária

Uma árvore binária é de busca, **árvore de busca binária** (**ABB**), se cada nó tem uma **chave** (valor) associado e cada nó da mesma tem chave maior que a de todos os nós da sub-árvore esquerda e menor que a chave de todos os nós da sub-árvore direita.

$$T \text{ é ABB} \iff \forall no \in VT \Rightarrow \left\langle \begin{array}{l} chave(esq) < chave(no) < chave(dir), \\ \forall (esq, dir) \in subarvore(no.esq) \times subarvore(no.dir). \end{array} \right\rangle$$

Propriedade 1.3 *Se T é ABB e $In\_ordem(T) = (i_1, i_2, \ldots, i_k)$ é uma passeio "em ordem" de T, então $chave(i_1) < chave(i_2) < \ldots < chave(i_{k-1}) < chave(i_k)$.*

Exemplo 1.4 *Demonstre a propriedade acima.*

Algoritmo 1.7 *Inserção de um elemento com a informação* `info` *numa árvore de busca binária.*

```
InsereABB (No no, info) {
  se (no não vazio) {
      no = novo_no(info);
      retorne no;
      }
  atual := no;
  enquanto (atual não vazio) {
    enquanto (atual.esq não vazio) {
      ant := atual;
      se (info < info(atual)) {
         atual := atual.esq;
         eh_esq := verdadeiro;
         }
      senao
      se (info(atual) < info) entao
         atual := atual.dir;
         eh_esq := falso;
      senao
         retorne no; // info(atual)=info=>não insere
      }
    se (eh_esq) // veio de um apontador esquerdo
      ant.esq := novo_no(info);
    senao
      ant.dir := novo_no(info);
    retorne no;
    }
```

Versão iterativa

```
InsereABB_Rec (No no, info) {
    if (no é vazio) retorne novo_no(info);
    if (info(no) < info)
       no.dir = InsereABB_Rec (no.dir, info);
    else if ( info < info(no) )
       no.esq = insereABB_Rec(no.esq, info);
    retorne no;
    }
```

Versão recorrente

Algoritmo 1.8 *Remoção de um elemento com a informação* `info` *numa árvore de busca binária.*

```
RemoveABB (info) {
    atual := no;
    enquanto (atual != vazio) {
      se (info < info(atual)  // pode estar na sub-árvore esquerda
         atual := atual.esq;
      senão
      se (info(atual) < info) // pode estar na sub-árvore direita
         atual := atual.dir;
      senão quebre_laço;      // info(atual)=info => elimine "atual"
      }
    se (atual != vazio) {
       (ant_menor, eh_esq) := achaMenor( atual.dir ); // acha o ant. à menor chave da sub-árvore direita
       if (eh_esq) {
          atual.info( info(ant_menor.esq) );    // pegue a info do nó à esquerda de ant_menor
          ant_menor.esq := (ant_menor.esq).dir; // eliminamos o apontador
          }
       else // cai aqui quando "atual" tem um só nó à direita (sub-árvore direita com um só nó)
          atual.info( info(ant_menor.dir) );
       }
    retorne no;
```

```
}
```

Note que a linha `(ant_menor, eh_esq) := achaMenor( atual.dir );` poderia ser trocada pelo equivalente à esquerda, `(ant_maior, eh_dir) := achaMaior( atual.esq );`, procurando o nó com a maior chave na sub-árvore à esquerda (que só tem chaves menores que `info(atual)`).

Uma melhoria do algoritmo acima é manter na ABB as alturas de cada nó e escolhermos para substituir o nó a ser removido àquele pertencente a sub-árvore de maior altura (`achaMenor(atual.dir)` ou `achaMaior(atual.esq)`). Com isso, cada remoção melhora[2] o balanceamento da árvore resultante.

1.3.7 Árvores de Busca Binária AVL

Os dois algoritmos anteriores mantém a árvore com a propriedade ABB, entretando existe um terceiro algoritmo que muito provavelmente é utilizado um número de vezes maior que os anteriores, a **busca**. Durante um intervalo de tempo em que a árvore T não é alterada, se $\mathcal{C}$ é o conjunto de chaves buscadas no período, sendo $freq(c)$ e $alt(c)$ ($c \in \mathcal{C}$), respectivamente, o número de vezes que c foi procurado e a altura do nó c em T, então

$$\text{o tempo de busca é proporcional a } \sum_{c\in\mathcal{C}} alt(c) \times freq(c).$$

Se não dispomos de informações a priori das buscas, podemos supor que cada chave é equiprovável (de ser buscada). Portanto, para reduzir o tempo de cada busca devemos minimizar as alturas dos nós na ABB, tanto quanto possível. Uma situação em que isso ocorre é quando a árvore é perfeitamente **balanceada**, ou seja, quando todos os nós num mesmo nível tiverem a mesma altura.

Entretanto é difícil manter uma árvore balanceada[3], quer dizer, mantê-la de modo que todas as folhas estejam em no máximo dois diferentes níveis é computacionalmente caro (muda a complexidade dos algorimos anteriores). Assim, devemos tentar manter a árvore "quase-balanceda" sem alterar a complexidade dos algoritmos de inserção e remoção anteriormente vistos.

Uma propriedade que atende este objetivos é o de **árvores AVL**: para cada nó da árvore, a diferença entre as alturas das sub-árvores esquerda e direita é de no máximo uma unidade.

Se no algoritmo 1.8 utilizarmos a heurística de escolher a sub-árvore de maior altura (`achaMenor(atual.dir)` ou `achaMaior(atual.esq)`), conseguiremos manter a estrutura de AVL nas remoções e não aumentaremos a complexidade computacional da remoção.

Já a inserção é mais complicada para ser implementada: após uma inserção devemos determinar o primeiro nó que teve a propriedade AVL estragada e trabalhar para recuperar a propriedade. Nesta tarefa de recuperação da estrutura AVL, podemos utilizar a propriedade de "associativadade" entres árvores de busca, como nas duas próximas propriedade.

Definição 1 *Se T_1 e T_2, são árvores binárias representaremos por $(T_1 \; A \; T_2)$ a árvore binária com raíz A, tendo T_1 como sub-árvore esquerda de A e T_2 como sub-árvore direita.*

Por analogia, denominaremos esta representação por "expressão aritmética da árvore".

[2] Mais precisamente, "não piora".

[3] Manter perfeitamente balanceada seria impossível, pois só poderíamos ter árvores com $2^{n+1} - 1 = \sum_{i=0}^{n} 2^i$ nós, o que significa que as inserções e remoções precisariam ser sempre de 2^n nós.

Propriedade 1.4 *Sendo T_1 e T_2 árvores de busca binárias e A uma chave tal que $chave(e) < A < chave(d)$, $\forall(e, d) \in T_1 \times T_2$, segue que $(T_1 \; A \; T_2)$ é árvore de busca binária.*

Propriedade 1.5 *Sendo T_1, T_2 e T_3 árvores de busca binárias e chaves A e B tais que $chave(t_1) < A < chave(t_2) < B < chave(t_3)$, $\forall(t_1, t_2, t_3) \in T_1 \times T_2 \times T_3$, segue que*

$$(T_1 \; A \; T_2) \; B \; T_3 \quad e \quad T_1 \; A \; (T_2 \; B \; T_3) \quad \textit{são árvores de busca binárias.}$$

Na figura a seguir representamos estas duas árvores.

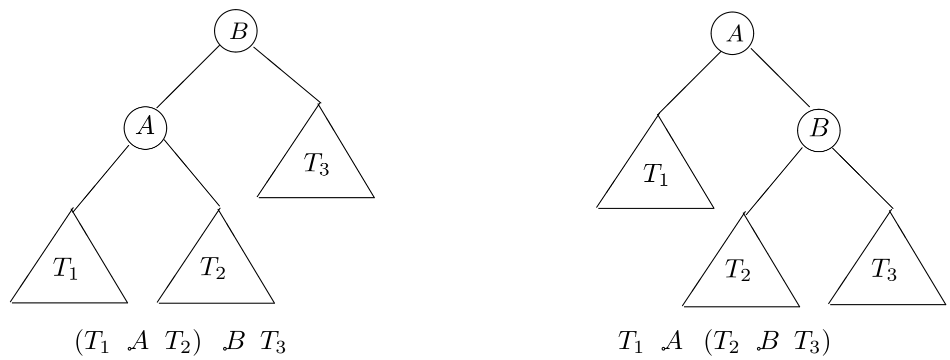


Figura 1.3: Representação das árvores AVL: rotação simples

Da representação acima podemos notar que a operação de mudança associativa pode produzir árvores de busca binárias com diferentes alturas, sendo que o número de parênteses na "expressão aritmética" da árvore define sua altura. Esta observação é essencial para operarmos eficientemente uma árvore de busca que deixou de ser AVL.

Existem dois tipo de mudanças associativas (na expressão da árvore) ou "rotações" (na representação gráfica) para recuperar a propriedade AVL:

1. Associação ou rotação simples: $(T_1 \; A \; T_2) \; B T_3$ é AVL, mas a inserção em T_1 destrói a propriedade.

   Recuperação: $T_1 \to \overline{T}_1$

   $$\left(\overline{T}_1 \; A \; T_2\right) \; B \; T_3 \to \overline{T}_1 \; A \; (T_2 \; B \; T_3)$$

   Vide figura anterior.

2. Associação ou rotação dupla: $(T_1 \; A \; T) \; B \; T_4$ é AVL, mas a inserção em T destrói a propriedade.

   Recuperação: Se tentar re-escrever a expressão da árvore com a representação nada conseguiremos, assim é necessário expandir a sub-árvore $T = (T_2 \; A \; T_3)$. Agora é indiferente que a inserção ocorra em T_2 ou em T_3, por isso re-escreveremos T_2, T_3 por $\overline{T}_2, \overline{T}_3$, entendendo que apenas um deles de fato foi alterado.

   $$\left(T_1 \; A \; \left(\overline{T}_2 \; C \; \overline{T}_3\right)\right) \; B \; T_4 \to \left(\left(T_1 \; A \; \overline{T}_2\right) \; C \; \overline{T}_3\right) \; B \; T_4 \to \left(T_1 \; A \; \overline{T}_2\right) \; C \; \left(\overline{T}_3 \; B \; T_4\right)$$

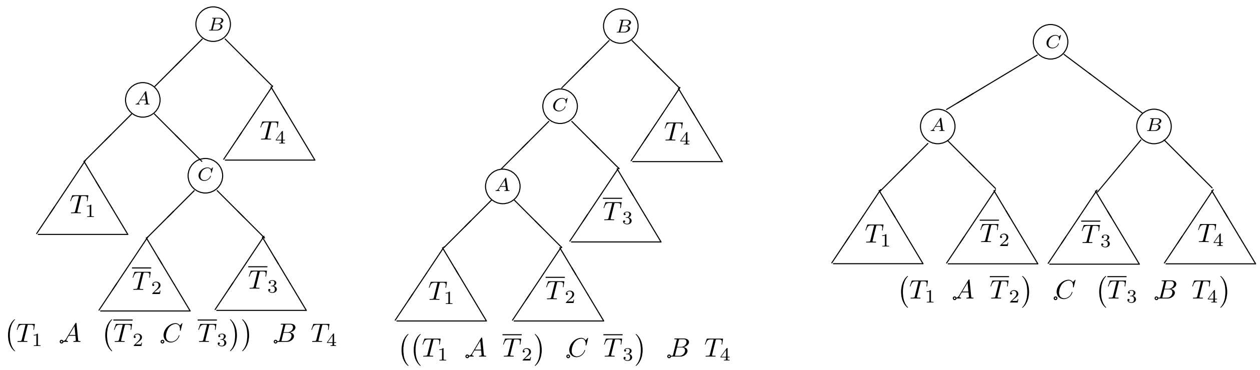


Figura 1.4: Representação das árvores AVL: dupla rotação