ED3. Árvores AVL

PROJECTO CODEBOARD DE SUPORTE A ESTE MÓDULO:
https://codeboard.io/projects/46752

Tipos Abstractos de Dados e Estruturas de Dados (revisão de conceitos)

Os tipos abstractos de dados (*Abstract Data Types*, ADTs) constituem um instrumento fundamental de abstracção, separando a **interface** de uma estrutura de dados (o conjunto de operações disponíveis sobre ela) da sua **implementação** concreta.

Os tipos de dados abstractos são implementados com base em **estruturas de dados concretos,** como sejam por exemplo as *sequências*, as *árvores*, ou os *grafos,*
a que está já associada uma forma ou organização interna particular, *linear*, *hierárquica*, ou *relacional*.

É comum definir-se estruturas de dados por *especialização* de outras. Por exemplo,

- Uma **árvore** é um caso particular de grafo (acíclico e com raíz).
- Uma **árvore binária** é um caso particular de árvore (cada nó tem no máximo dois descendentes).
- Uma árvore **binária de procura** (*Binary Search Tree*, BST) é um caso particular de árvore binária (com um invariante que estabelece uma relação de ordem *inorder*).

Cada estrutura de dados (lógica) pode ser implementada de diversas formas, a que correspondem diferentes **estruturas de dados físicas.** Por exemplo uma sequência de elementos de um mesmo tipo pode ter

- Implementação contígua: um *array* (estrutura indexada com acesso em tempo constante, podendo ser estático ou dinâmico);
- Implementação ligada: o acesso ao elemento seguinte é feito através de um campo "próximo".

Note-se que se um *array* é já uma estrutura física, a noção de sequência ligada é algo que se encontra ainda ao nível lógico, podendo ser implementada também ela sobre um *array*, ou então como uma *lista ligada*, alocada dinamicamente, com utilização de apontadores.

O ADT Dicionário / Array Associativo / ou Mapeamento

Armazena pares chave → valor, tendo a semântica de uma função finita. As operações básicas são

1. A **inserção** de um par chave → valor;
2. A **alteração** do valor associado a uma chave;
3. A **consulta** com base numa chave, podendo obter-se como resultado um valor ou a indicação de que a chave não ocorre no dicionário;
4. A **remoção** de um par, dada a respectiva chave.

Note-se que as operações 1 e 2 podem ser implementadas pela mesma operação.

Se as chaves forem de *um tipo que admita uma noção de ordem*, um dicionário pode ser implementado por uma árvore binária de procura.

Árvores Binárias de Procura

Uma *árvore binária de procura* ("binary search tree", BST) é uma estrutura de dados que pode ser utilizada para implementar *dicionários* ou simplesmente (multi-)conjuntos, e cujas operações se caracterizam por um comportamento largamente dependente da *forma* da árvore.

Assim, temos os dois seguintes casos extremos:

- Uma árvore *totalmente desequilibrada* assume a forma de uma lista de elementos (cada nó tem sempre um descendente vazio); a operação de procura pode executar no melhor caso em tempo constante, e no pior caso em tempo linear: $T(N) = \Omega(1), \mathcal{O}(N)$. O mesmo sucede com a operação de inserção.

- Já numa árvore *equilibrada*, os elementos inseridos ocupam um número de níveis próximo do mínimo possível, e o pior caso da procura passa para logarítmico: $T(N) = \Omega(1), \mathcal{O}(\log N)$. As inserções executam *todas* em tempo logarítmico, $T(N) = \Theta(\log N)$.

É claramente desejável trabalhar com árvores equilibradas. Num cenário em que ocorram muito mais operações de procura do que inserções, uma solução possível é reequilibrar a árvore periodicamente (por exemplo, após cada 100 inserções). Mas no caso geral será preferível manter a árvore permanentemente equilibrada.

Árvores AVL

Uma árvore AVL (Adelson-Velskii & E.M. Landis) é uma árvore binária de procura em que todos os nós satisfazem adicionalmente o seguinte invariante estrutural:

> *As alturas da sub-árvore da esquerda e da sub-árvore da direita diferem no máximo numa unidade:* $|h_e - h_d| \leq 1$.

```
      10


      10
        \
        20

```

```

      10
    /    \
   5     20
           \
            30

      10
    /    \
   5     20
 /      /   \
3      15    30
           /
          25
```

Note-se que este invariante admite árvores que não são completas, i.e. é possível que um nível da árvore contenha elementos sem que o nível anterior esteja completamente preenchido.

No entanto, tal como nas árvores completas, a altura de uma árvore AVL é assimptoticamente logarítmica, o que garante que a operação de procura executa em tempo $T(N) = \mathcal{O}(\log N)$. Mas o que é mais interessante ainda é o seguinte:

> *O tempo de execução das operações de inserção e remoção,* modificadas por forma a efectuarem o necessário ajuste das árvores para preservar o invariante AVL*, é também $T(N) = \mathcal{O}(\log N)$.*

Veremos em seguida como modificar o algoritmo tradicional de inserção numa árvore binária de procura por forma a lidar com árvores AVL.

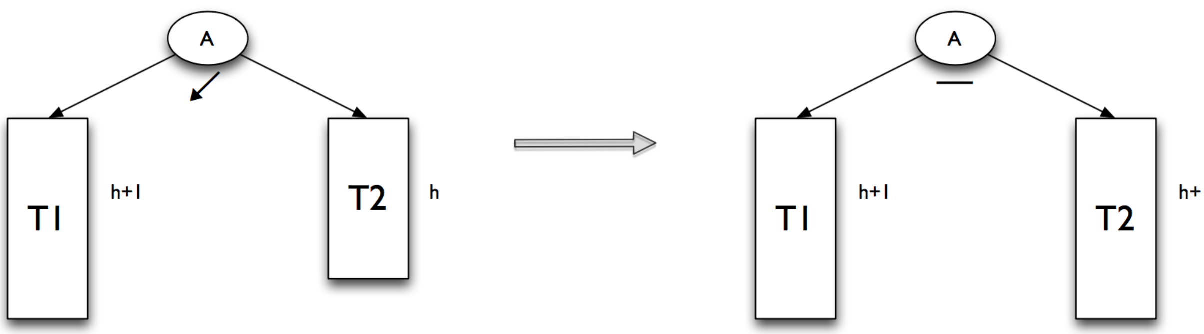
Algoritmo de inserção numa árvore AVL

Em algumas situações, a inserção de um novo elemento numa árvore AVL preservará o invariante estrutural ($|h_e - h_d| \leq 1$) em todos os nodos. Consideremos agora em detalhe o que poderá suceder quando se faz, recursivamente, uma inserção *à direita da raíz* (naturalmente, o outro caso é simétrico).

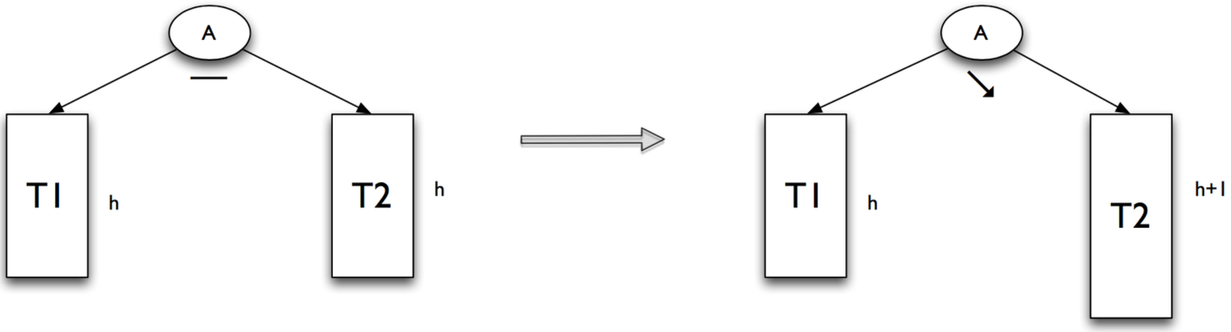
Caso 0: inserção à direita não provoca aumento da altura da sub-árvore da direita.
Neste caso não há nada a fazer, a relação entre h_e e h_d mantém-se.

Nos restantes casos haverá aumento da altura da árvore da direita.

Caso 1: $h_e = h_d + 1$
A sub-árvore da esquerda é mais pesada, e neste caso passaremos a ter, depois da inserção à direita, $h_e = h_d$.



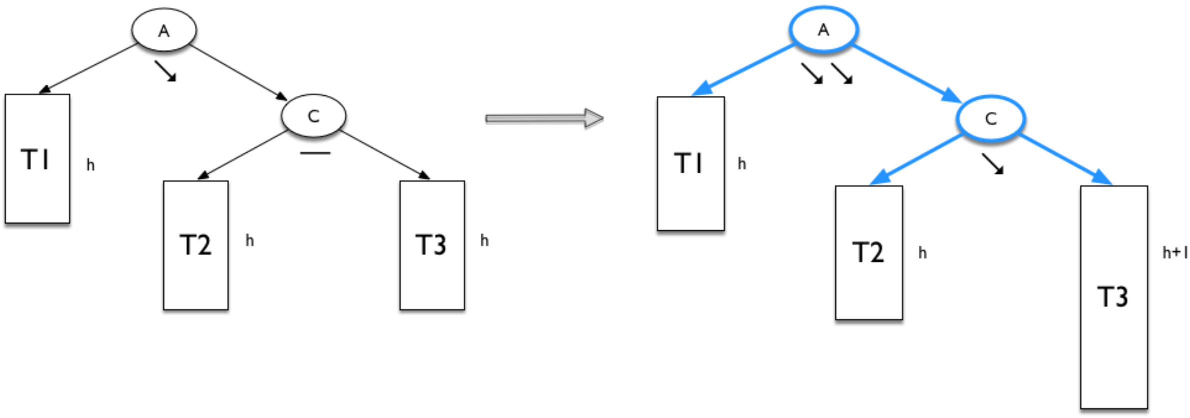
Caso 2: $h_e = h_d$
As duas sub-árvores têm à partida a mesma altura. Neste caso passaremos a ter, depois da inserção à direita, $h_d = h_e + 1$. A raíz (A) continua a satisfazer o invariante.



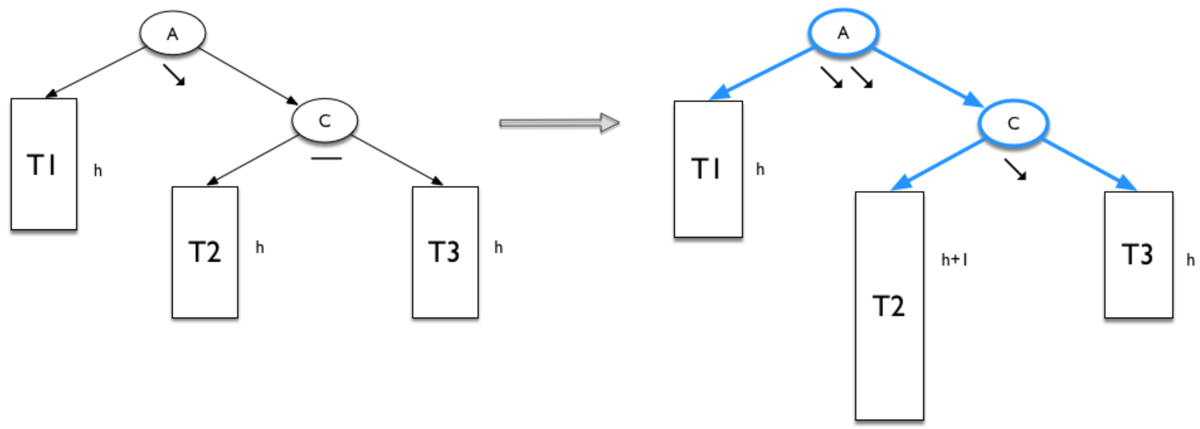
Já se adivinha que o caso problemático, que levará à necessidade de reajustar a árvore, ocorre quando a sub-árvore da direita já é à partida a mais pesada:

Caso 3: $h_d = h_e + 1$.
Note-se que neste caso a sub-árvore da direita tem necessariamente duas sub-árvores com a mesma altura h. Chamemos-lhes T2 e T3.
Então, podem agora surgir dois casos diferentes, consoante o aumento de altura ocorra em T2 ou em T3. Ambos os casos levam à violação do invariante na raíz (A), com $h_d = h_e + 2$.

Caso 3a: a inserção produziu um aumento da altura de T3

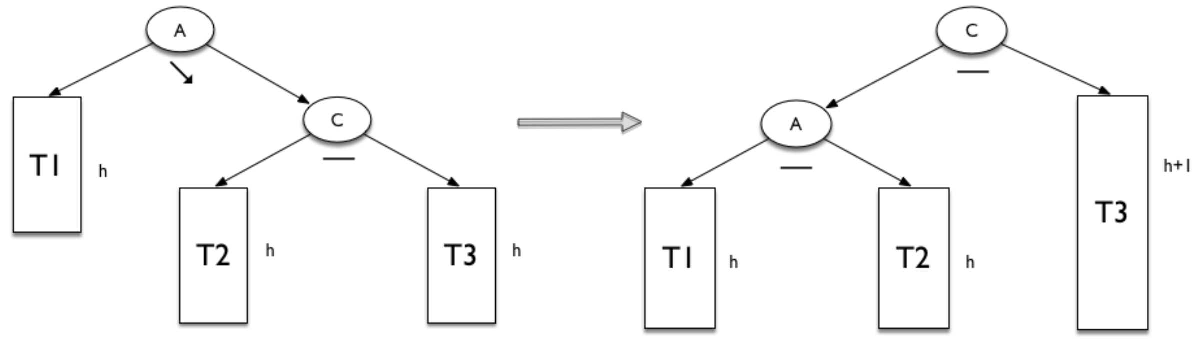


Caso 3b: a inserção produziu um aumento da altura de T2



Vejamos como será reposto o invariante em cada caso, ajustando-se a estrutura da árvore.

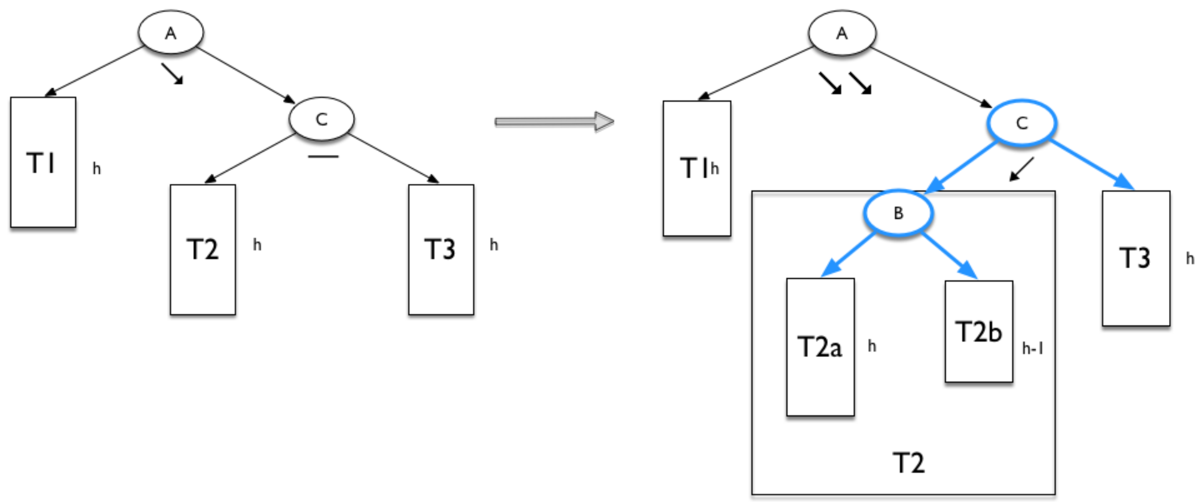
Comecemos pelo caso **Caso 3a.** Neste caso procede-se a uma operação de *rotação* à esquerda da árvore: o nó C tomará o papel da raíz, descendo o nó A para a esquerda de C. A árvore T2 passará a estar à direita de A.



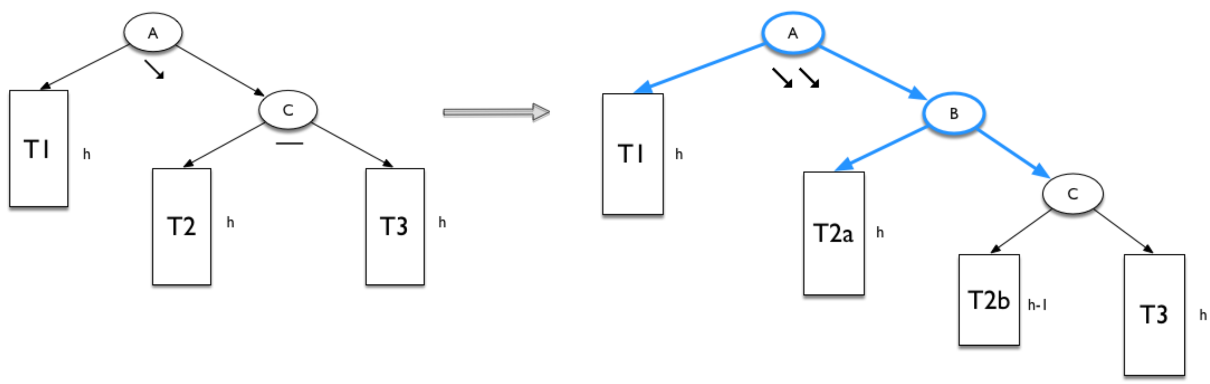
A figura acima mostra que as sub-árvores de A têm a mesma altura, e o mesmo acontece com as de C. Sendo assim, o invariante $|h_e - h_d| \leq 1$ é satisfeito em ambos os nós.

EXERCÍCIO: Mostre que o resultado da rotação continua a ser uma árvore binária de procura, i.e. que a ordem relativa dos elementos é preservada.

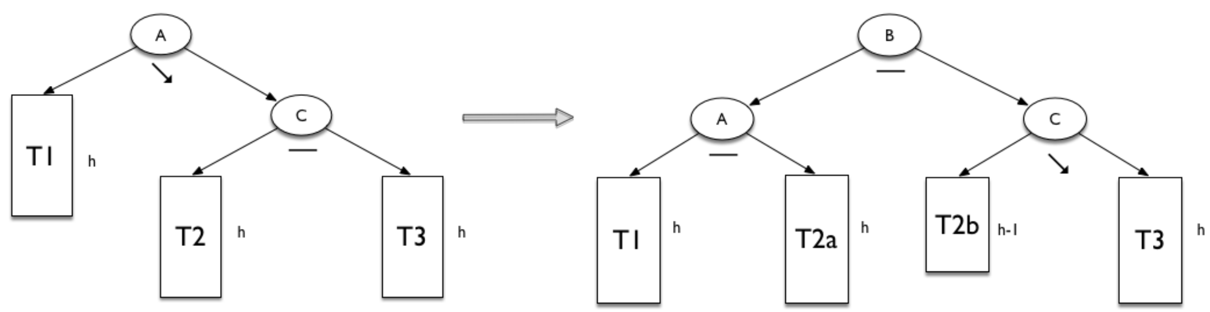
Caso 3b: a inserção produziu um aumento da altura de T2. Necessitamos neste caso de observar a estrutura de T2, com raíz B e sub-árvores T2a e T2b:



Uma simples rotação à esquerda não resolveria neste caso o problema. Antes disso é necessário efectuar uma *rotação à direita, com eixo no nó C. B sobe para o lugar de C, que desce para a sua direita:*



Em seguida faz-se a rotação à esquerda com eixo na raíz A, como no Caso 3A:



Mais uma vez as sub-árvores da nova raíz B têm a mesma altura, e o mesmo acontece com as de C. Note-se que este reajustamento repõe o invariante das árvores AVL, independentemente dos pesos relativos de T2a e T2b.

EXERCÍCIO: Represente graficamente a evolução de uma árvore AVL quando é efectuada a seguinte sequência de inserções: 10, 20, 30, 70, 40, 50. Não se esqueça de indicar os factores de balanceamento de cada nó.

```
     10,E



     10,R
         \
         20,E



     10,R
         \
         20,R
            \
           30,E



         20,E
      /         \
    10,E         30,E


```

```
         20,R
      /       \
    10,E        30,R
                    \
                     70,E


         20,R
      /       \
    10,E        30,R
                    \
                     70,L
                  /
               40,E


         20,R
      /       \
    10,E        30,
                    \
                      40,
                        \
                         70


         20,R
      /       \
    10,E        40,E
                 /    \
              30,E    70,E


```

```
        20,R
     /       \
   10,E        40,R
              /    \
            30,E    70,L
                   /
                50,E


             40,E
         /          \
        20,E          70,L
      /    \          /
    10,E   30,E     50,E


```

Implementação em C do Algoritmo de Inserção

Tipos de dados

Consideremos uma árvore binária de números inteiros. A estrutura habitual dos nós de uma árvore binária comum será aumentada com um campo correspondente ao *estado de balanceamento* do nó, que poderá ser um de três valores:

- LH (sub-árvore da esquerda tem maior altura),
- RH (sub-árvore da direitura tem maior altura), ou
- EH (alturas iguais).

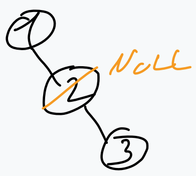
```c
typedef int TreeEntry;
typedef enum balancefactor { LH , EH , RH } BalanceFactor;

struct treenode {
    BalanceFactor bf;
    TreeEntry entry;
    struct treeNode *left;
    struct treeNode *right;
};

typedef struct treenode *Tree;
```

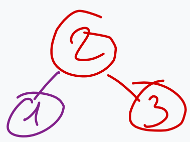
Funções auxiliares

Vimos acima que o ajuste da estrutura das árvores tem por base operações simples de *rotação.*

A seguinte função realiza uma rotação simples à esquerda da árvore com raíz apontada por t, devolvendo o endereço da nova raíz. Note que a função não pode ser executada se a árvore da direita for vazia!

```c
// requires:
// (t != NULL) && (t->right != NULL)
//
Tree rotateLeft(Tree t)
{
    Tree aux = t->right;
    t->right = aux->left;
    aux->left = t;
    t = aux;
```






```c
Tree rotateRight (Tree t)
{
    Tree aux = t->left;
    t->left = aux->right;
    aux->right = t;
    t = aux;
    return t;
}
```

```
    return t;
}
```

EXERCÍCIO: Implemente a função de rotação à direita.

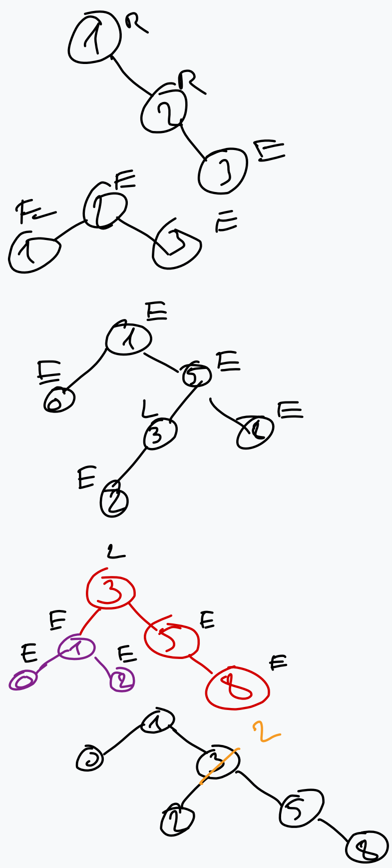
Escreveremos em seguida uma função que, recebendo uma árvore *cuja raíz deixou de satisfazer o invariante AVL* e que está *desequilibrada para a direita* (i.e. $h_d = h_e + 2$), corrige a estrutura da árvore.

Para isso, a função:

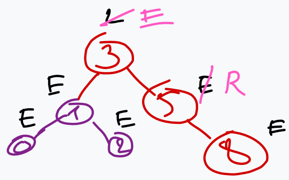
- começa por determinar se o caso exige uma rotação simples ou dupla, examinando o indicador de balanceamento do nó à direita;
- no segundo caso (rotação dupla), e tendo em conta o que foi dito anteriormente, é necessário ajustar os indicadores de balanceamento dos nós à esquerda (A na figura) e à direita (C) da nova raíz da árvore (B), de acordo com o estado actual do indicador de balanceamento de B;
- finalmente ajusta-se o indicador da raíz.

Tal como a anterior, esta função não pode ser executada se a árvore da direita for vazia.

```
// requires:
// (t != NULL) && (t->right != NULL)
//
Tree balanceRight(Tree t)
{
    if (t->right->bf==RH) {
        // Rotacao simples, caso 3a
        t = rotateLeft(t);
        t->bf = EH;
        t->left->bf = EH;
    }
    else {
        // Dupla rotação, caso 3b
        t->right = rotateRight(t->right);
        t=rotateLeft(t);
```



```
        switch (t->bf) {
            case EH:
                t->left->bf = EH;
                t->right->bf = EH;
                break;
            case LH:
                t->left->bf = EH;
                t->right->bf = RH;
                break;
            case RH:
                t->left->bf = LH;
                t->right->bf = EH;
        }
        t->bf = EH;
    }
    return t;
}
```



EXERCÍCIO: Implemente a função `balanceLeft`.

Função de inserção numa árvore binária de procura comum

O tipo esperado para uma função de inserção não deverá ser novidade: recebe um apontador para a raíz da árvore e o elemento a inserir, e devolve o endereço da raíz (que poderá ter sido alterado). O algoritmo recursivo é bem conhecido:

```
Tree insertTree(Tree t, TreeEntry e) {
    if (t==NULL){
        t = (Tree)malloc(sizeof(struct treenode));
        t->entry = e;
        t->right = t->left = NULL;
```

```
    }
    else if (e > t->entry)
        t->right = insertTree(t->right, e);
    else
        t->left = insertTree(t->left, e);
    return t;
}
```
+

Função de inserção numa árvore AVL

A função de inserção numa árvore AVL deve devolver, além do endereço da nova raíz da árvore, também *informação sobre se a altura da árvore cresceu ou não após esta inserção*. Para isso incluiremos no protótipo um parâmetro de tipo int passado por referência:

```
Tree insertTree(Tree t, TreeEntry e, int *cresceu);
```

A ideia é que depois da chamada `insertTree(t, e, &c)` com c de tipo `int`, c terá o valor 1 se a altura de t cresceu, e 0 em caso contrário.

O algoritmo de inserção numa árvore AVL segue a estrutura do anterior, com as seguintes diferenças:

- detecta os casos em que o invariante AVL é violado, e chama a função `balanceRight` ou `balanceLeft` para corrigir a estrutura da árvore
- reajusta os indicadores de balanceamento dos nós afectados pela inserção
- determina se a altura da árvore aumentou ou não, e atribui o valor adequado 0 ou 1 a `*cresceu`

EXERCÍCIO: Complete a definição da seguinte função de inserção:

```
Tree insertTree(Tree t, TreeEntry e, int *cresceu)
{
    if (t==NULL){
        t = (Tree)malloc(sizeof(struct treenode));
```

```
        t->entry = e;
        t->right = t->left = NULL;
        t->bf = EH;
        *cresceu = 1;
    }
    else if (e > t->entry) {
        t->right = insertTree(t->right, e, cresceu);
        if (*cresceu ==1) {
            switch (t->bf) {
                case LH: RH:
                    t->bf = EH;
                    *cresceu = 0;
                    break;
                case EH:
                    t->bf = RH; LH;
                    *cresceu = 1;
                    break;
                case RH: LH:
                    t = balanceRight left(t);
                    *cresceu = 0;
                    break;
            }
        }
    }
    else {
        t->left = insertTree(t->left,e,cresceu);
        if (*cresceu) {
            ...
        }
    }
    return t;
}
```

Outros Exercícios

1. A seguinte função calcula a altura de uma qualquer árvore binária, em tempo $\Theta(N)$:

```
int nonAVL_treeHeight(Tree t) {
    int l, r;
    if (t==NULL) return 0;
    l = treeHeight(t->left);
    r = treeHeight(t->right);
    if (l>r) return l+1;
    else return r+1;
}
```

Redefina a função por forma a calcular a altura de uma árvore AVL em tempo $\Theta(\log N)$.

2. É possível testar se uma árvore binária é ou não AVL da seguinte forma:

```
int isAVL (Tree t) {
    int l, r;
    if (t == NULL) return 1;

    l = treeHeight (t->left);
    r = treeHeight (t->right);

    return (abs (l-r) <= 1 &&
            isAVL(t->left) &&
            isAVL(t->right));
}
```

   a. Analise o tempo de execução no pior caso desta função.

   b. É possível optimizar esta função alterando-a por forma a calcular simultaneamente a altura da árvore, dispensando assim a utilização da função treeHeight. Complete a seguinte definição e analise o seu tempo de execução no pior caso:

```c
// altura da árvore será colocada em *p
int isAVL_aux (Tree t, int *p) {
    ...
}

int isAVL_opt (Tree a) {
    int p;
    return (isAVL_aux (a, &p));
}
```