INF 1010
Estruturas de Dados Avançadas

Árvores AVL

Árvores Balanceadas

Balanceamento de Árvores Binárias de Busca

Motivação:

busca em O(log(n))

Estratégia:

diminuir a diferença de altura entre
a sub-árvore à esquerda e
a sub-árvore à direita
de cada nó

árvores AVL

Adelson-Velskii & Landis

1922

1921

Árvore binária de busca construída de tal modo que em todos seus nós, a altura de sua sub-árvore à direita difere da altura da sub-árvore à esquerda de no máximo 1 unidade

Fator de balanceamento

Fator de balanceamento de um nó

fb = hd – he

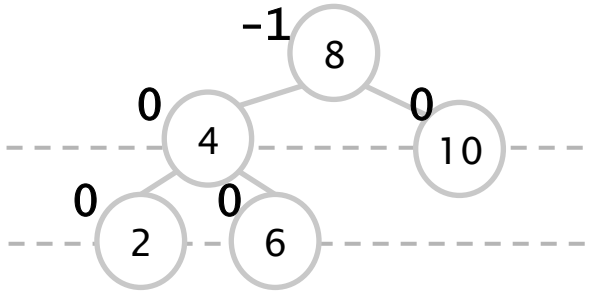
onde he = altura da sub-árvore à esquerda

hd = altura da sub-árvore à direita

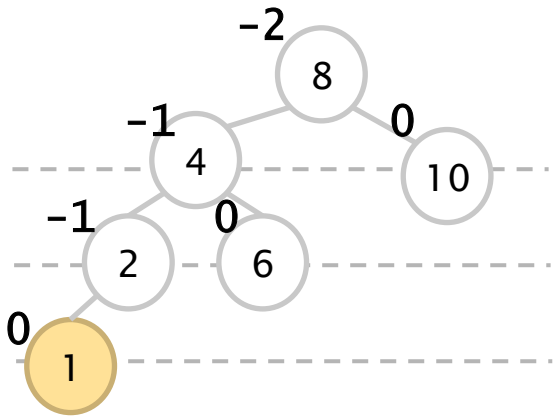
```
struct _avl {
    int chave;
    int fb; /*fator de balanceamento*/
    struct _avl *pai;
    struct _avl *esq;
    struct _avl *dir;
};
```

Árvore AVL – desequilíbrio (caso 1a)

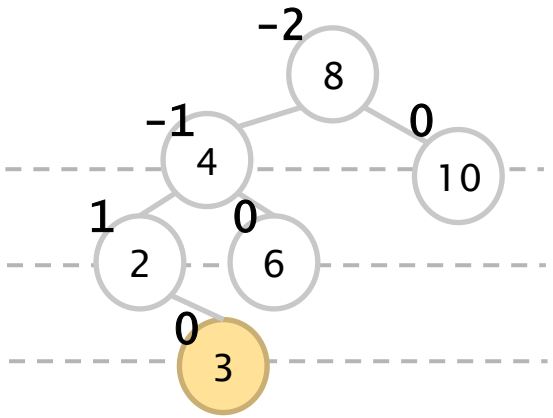
árvore balanceada (equilibrada)



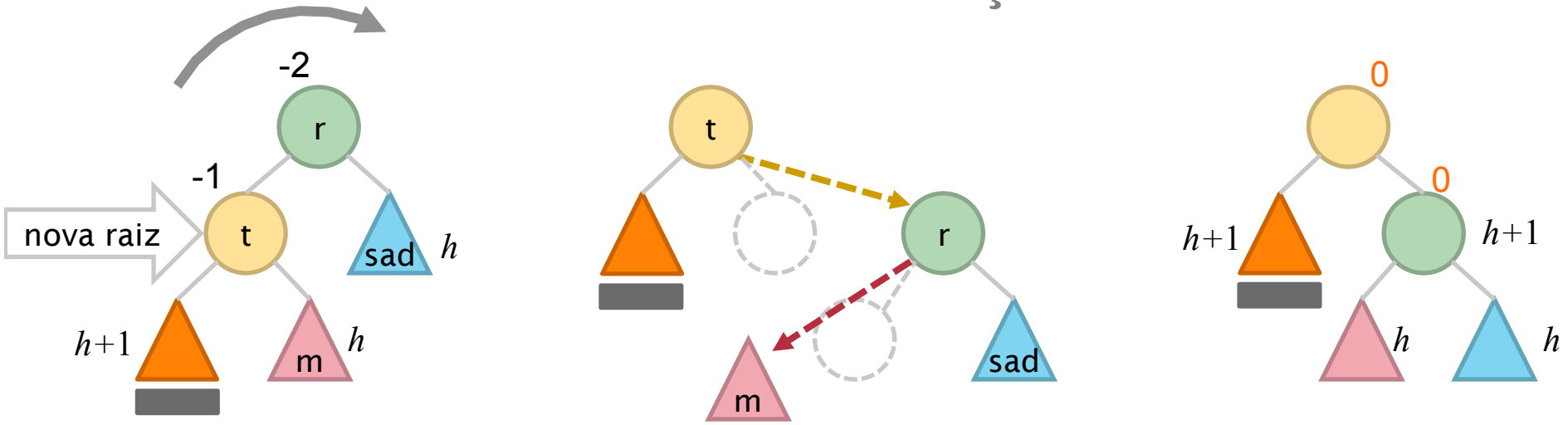
desequilíbrio após inserir 1



desequilíbrio após inserir 3



Árvore AVL – Caso 1a: rotação à direita



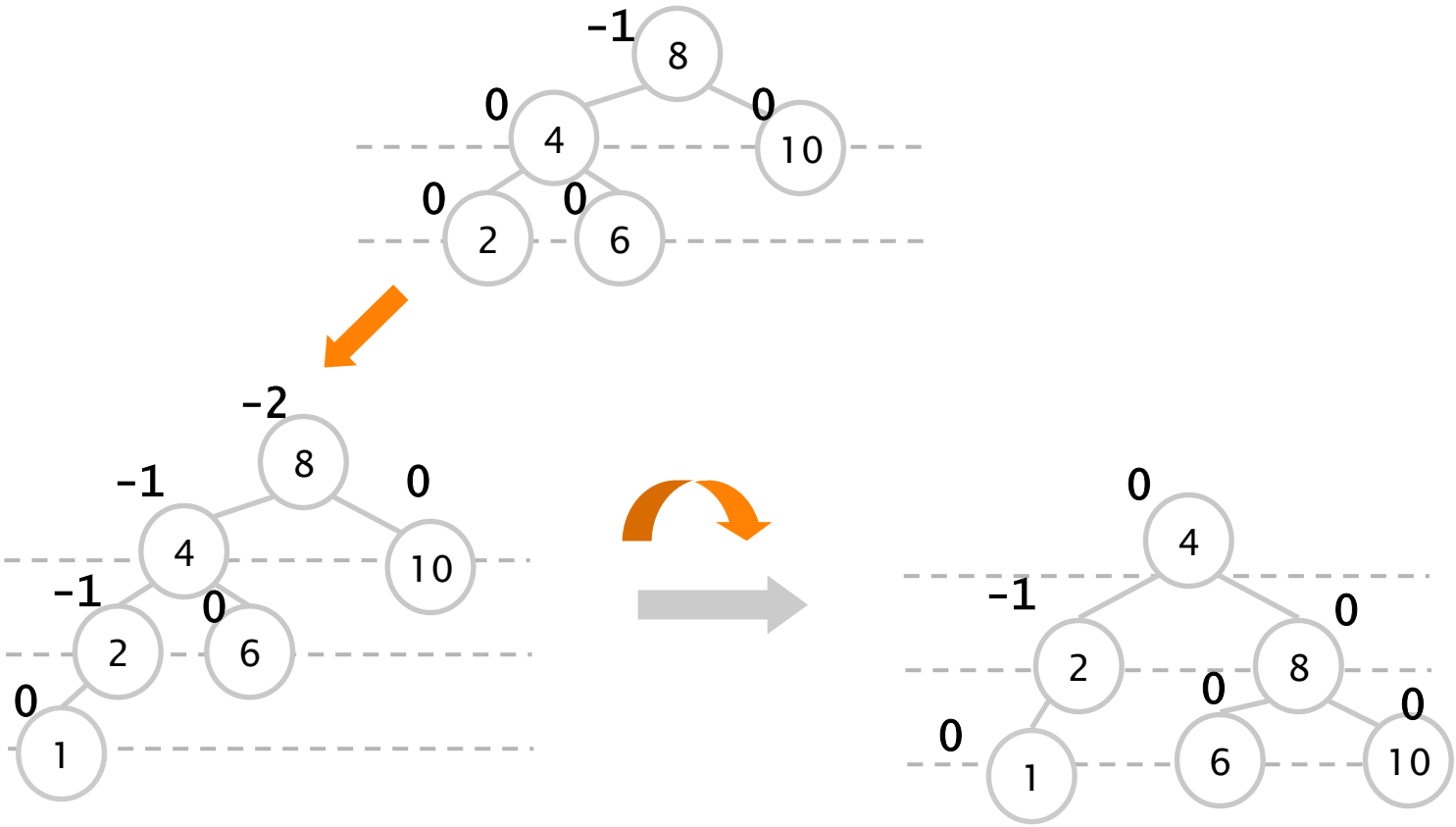

```
static Avl* rotacao_direita(Avl *r) {
    Avl *pai=r->pai, *t=r->esq, *m=t->dir;
    t->dir = r;  r->pai = t;
    r->esq = m;  if (m) m->pai = r;
    t->pai = pai;
    if (pai) {
        if (pai->dir == r) pai->dir = t;
        else pai->esq = t;
    }
    t->fb = r->fb = 0;
    return t;
}
```

Caso 1a - Exemplo de rotação à direita

Uma rotação à direita ☑

Caso 1a - Exemplo de rotação à direita

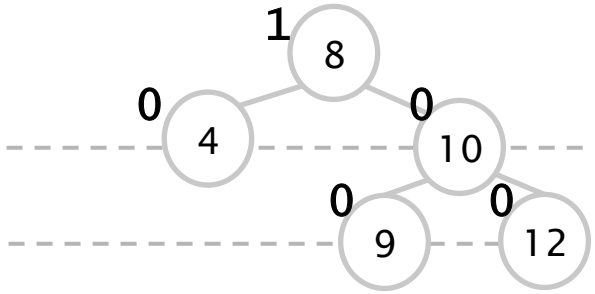
árvore balanceada (equilibrada)



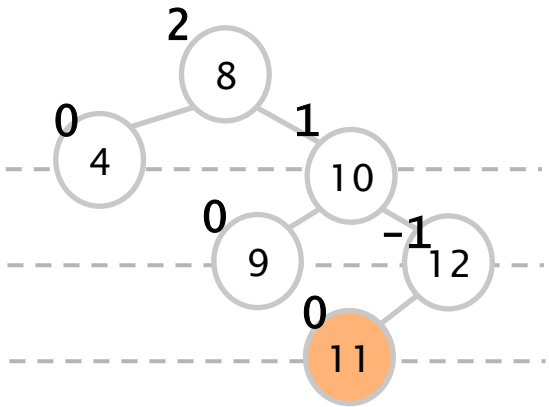
Uma rotação à direita ☑

Árvore AVL – desequilíbrio (caso 1b)

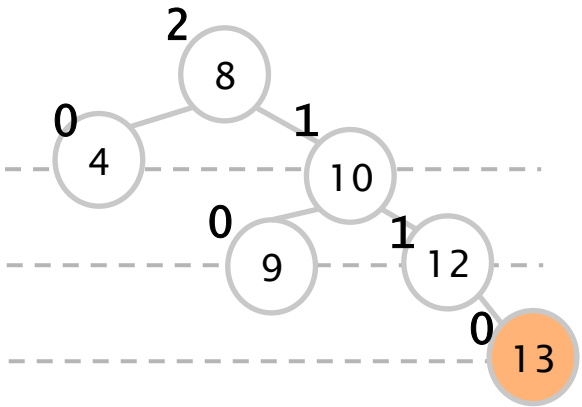
árvore balanceada (equilibrada)



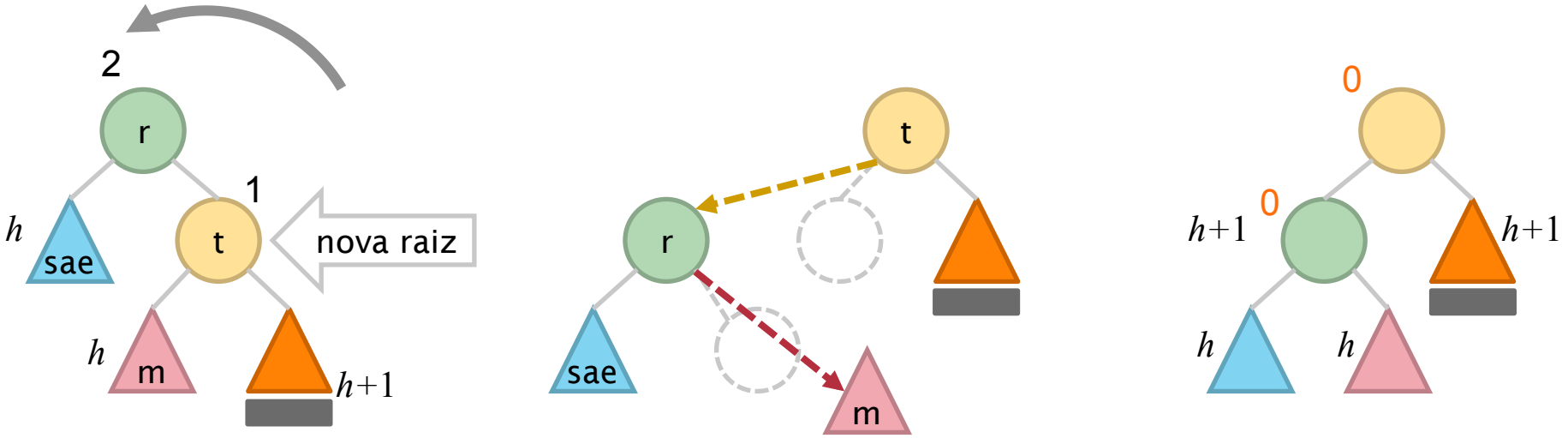
desequilíbrio após inserir 11



desequilíbrio após inserir 13

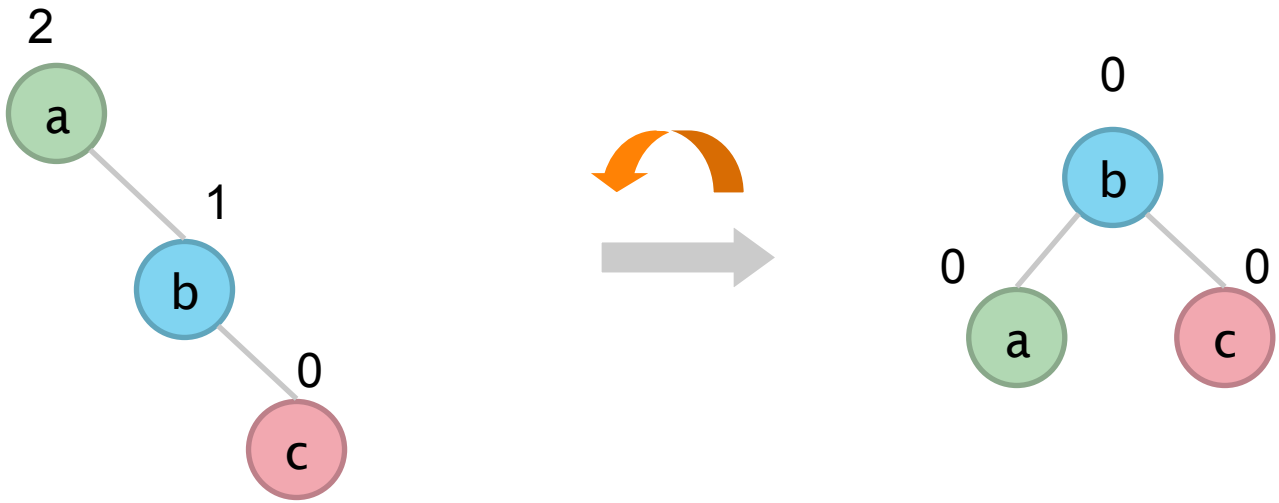


Árvore AVL – Caso 1b: rotação à esquerda



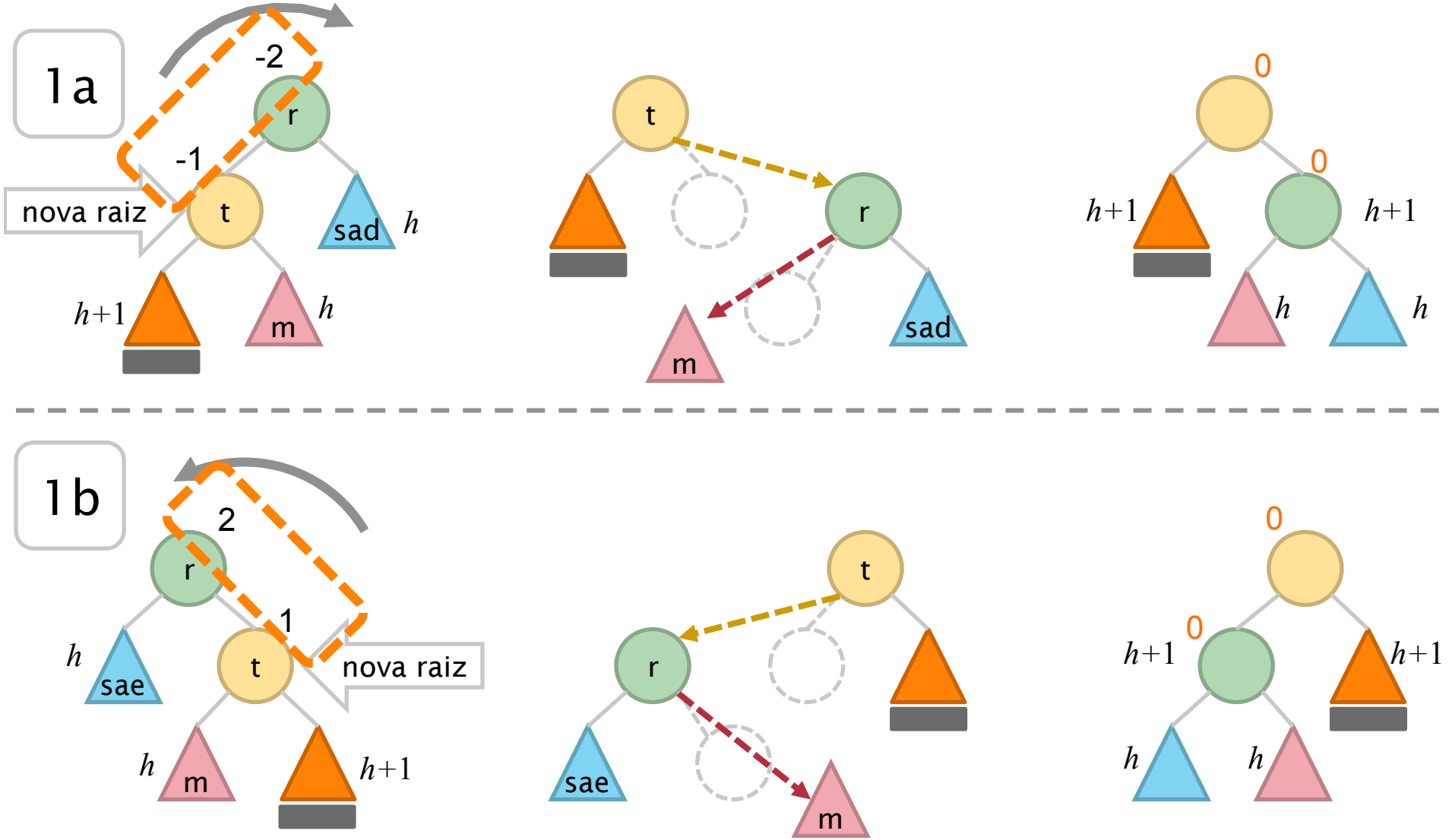

```
static Avl* rotacao_esquerda(Avl *r) {
    Avl *pai=r->pai, *t=r->dir, *m=t->esq;
    t->esq = r;   r->pai = t;
    r->dir = m;   if (m) m->pai = r;
    t->pai = pai; if (pai) {
        if (pai->dir == r) pai->dir = t;
        else               pai->esq = t;
    }
    t->fb = r->fb = 0;
    return t;
}
```

Caso 1b - Exemplo de rotação à esquerda

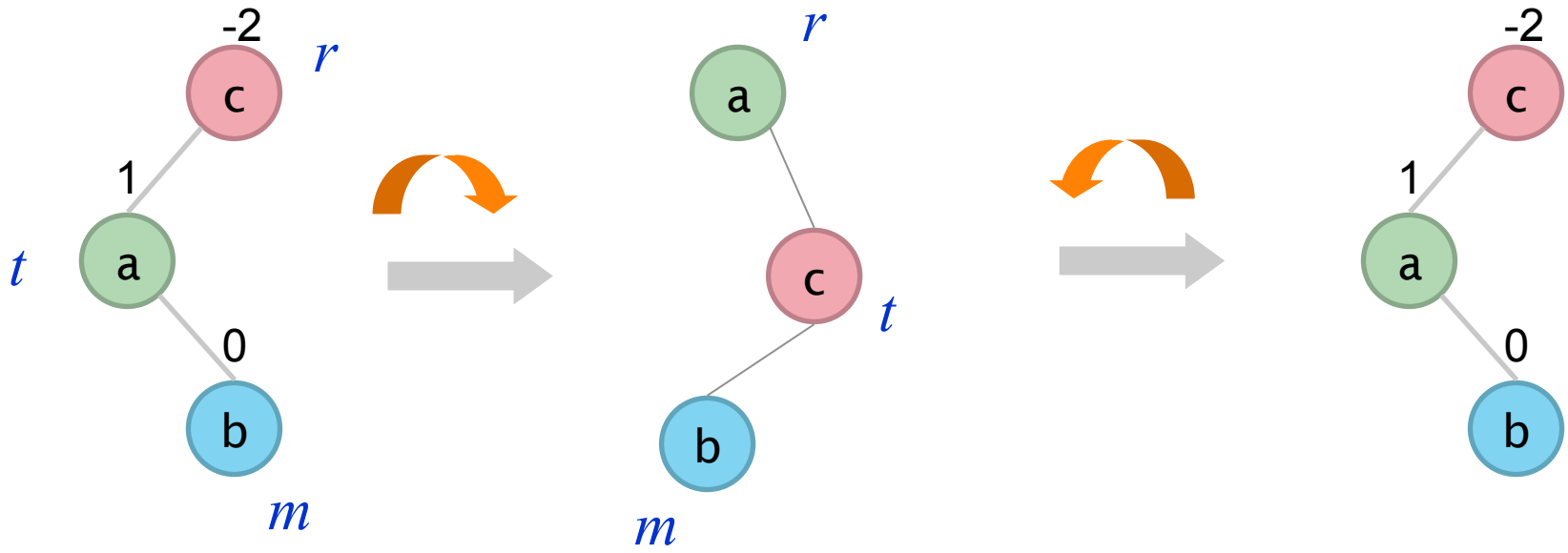


Uma rotação à esquerda ☑

Árvore AVL – Casos simples: fb com mesmo sinal



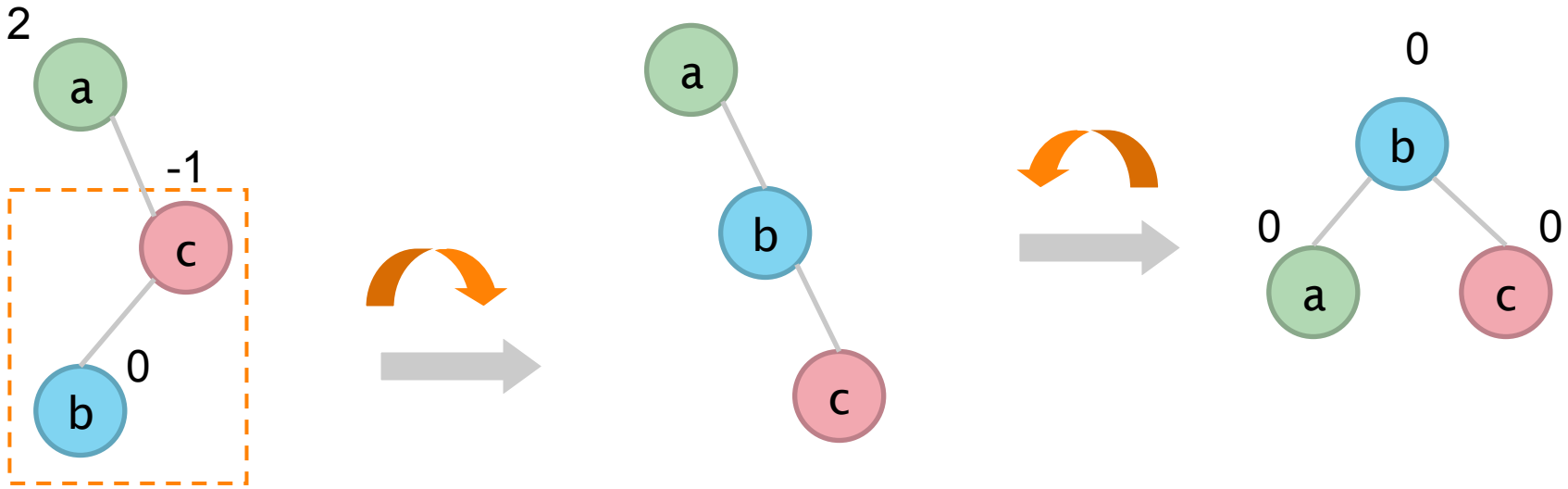
Exemplo: caso em que rotação simples não resolve



Não resolve!

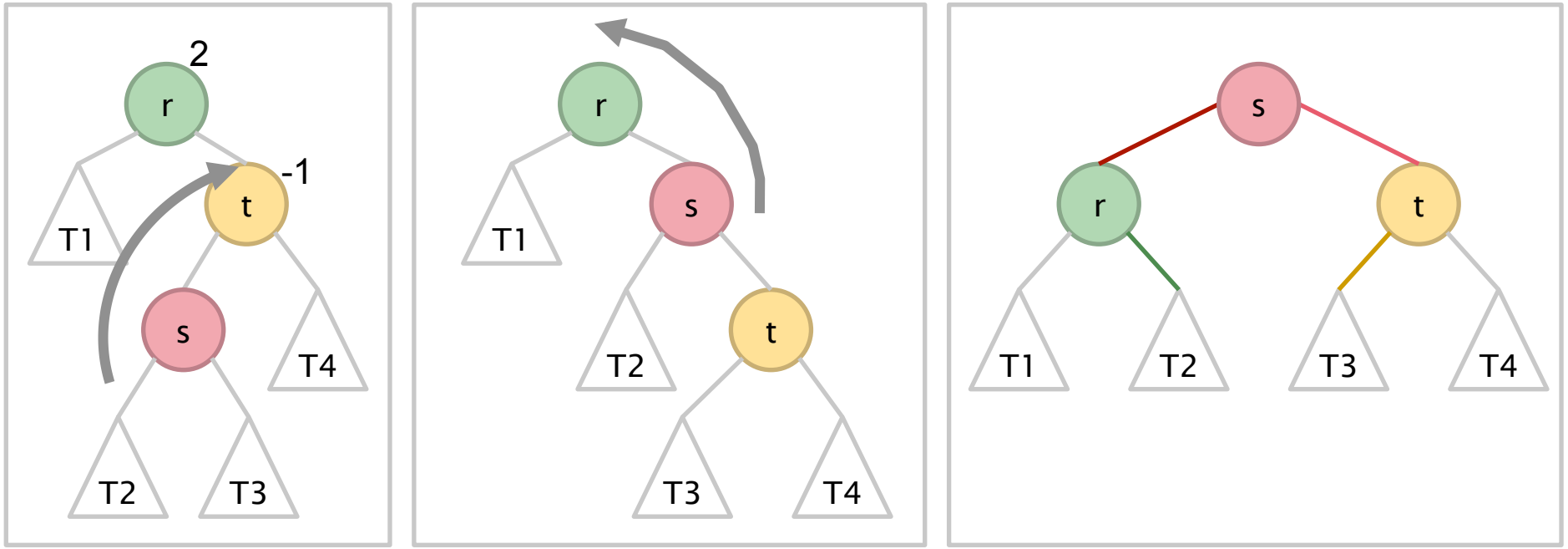
Caso 2a - Balanceamento com rotação dupla

Uma rotação à direita na sub-árvore à direita...



...seguida de uma rotação à esquerda

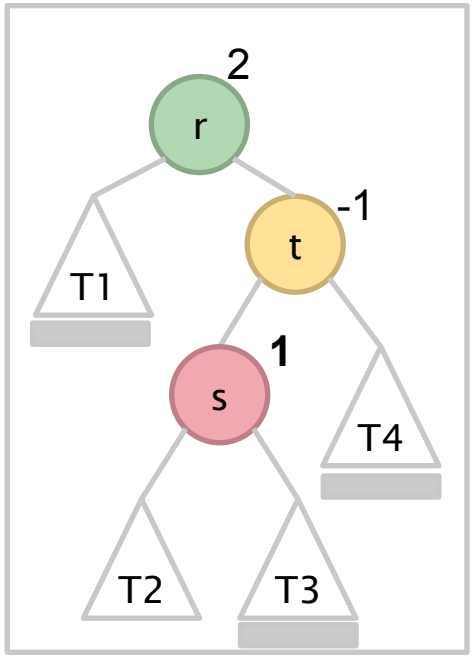
Caso 2a – Rotação direita-esquerda

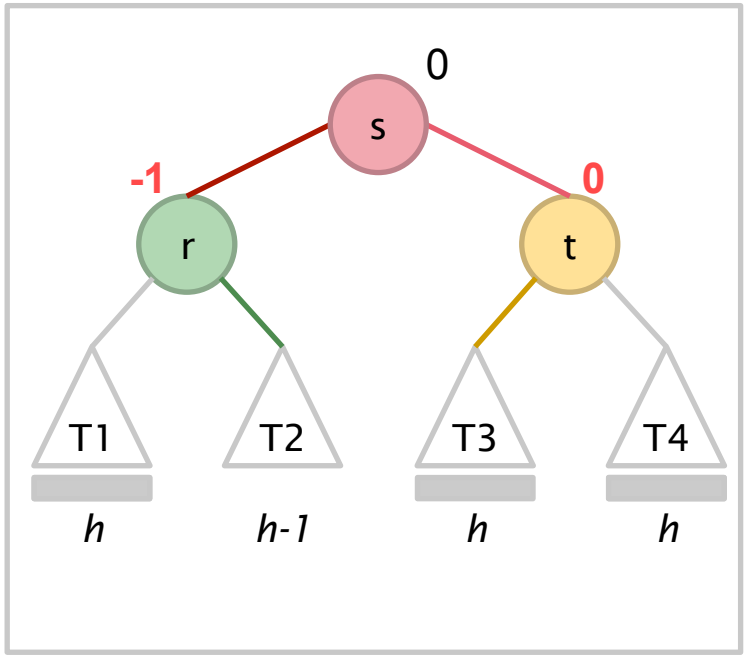



```
Avl* avl_rotacao_direita_esquerda(Avl *r) {
  Avl *t  = r->dir, *s  = t->esq;
  Avl *T2 = s->esq, *T3 = s->dir;
  s->esq = r;      s->dir = t;
  r->dir = T2;     t->esq = T3;
  r->fb = (s->fb==1)  ? -1 : 0;
  t->fb = (s->fb==-1) ?  1 : 0;
  s->fb = 0;
  return s;
}
```

(Explicação a seguir)

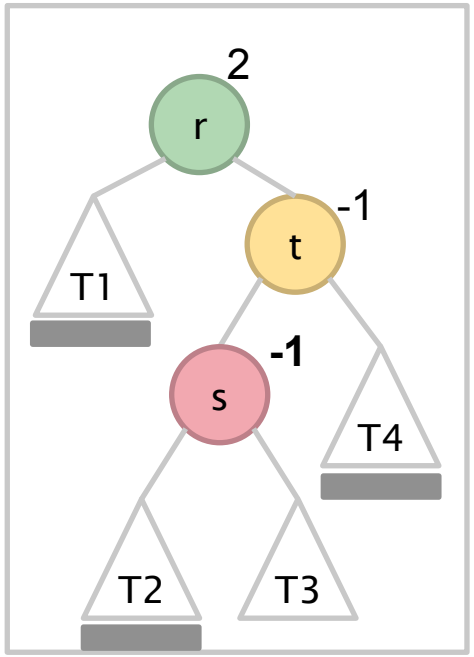
Caso 2a.1 – Rotação direita-esquerda

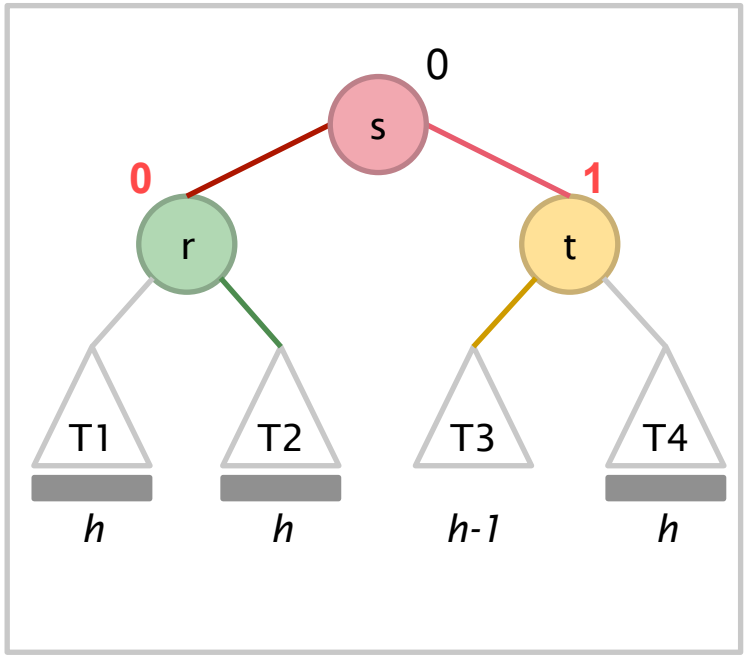





```
Avl* avl_rotacao_direita_esquerda(Avl *r) {
  Avl *t  = r->dir, *s  = t->esq;
  Avl *T2 = s->esq, *T3 = s->dir;
  s->esq = r;      s->dir = t;
  r->dir = T2;     t->esq = T3;
  r->fb = (s->fb==1)  ? -1 : 0;
  t->fb = (s->fb==-1) ?  1 : 0;
  s->fb = 0;
  return s;
}
```

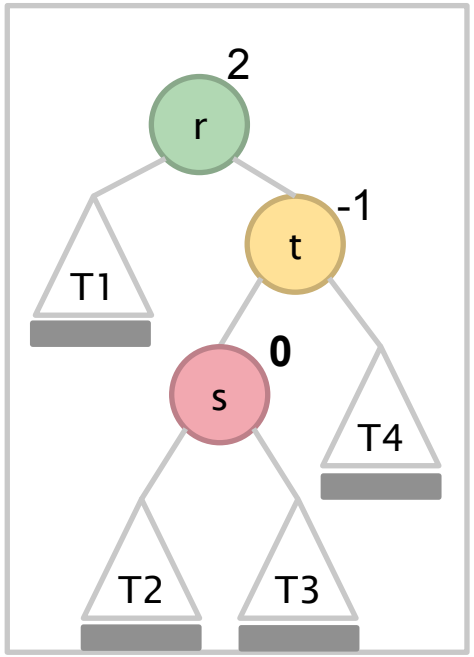
Caso 2a.2 – Rotação direita-esquerda

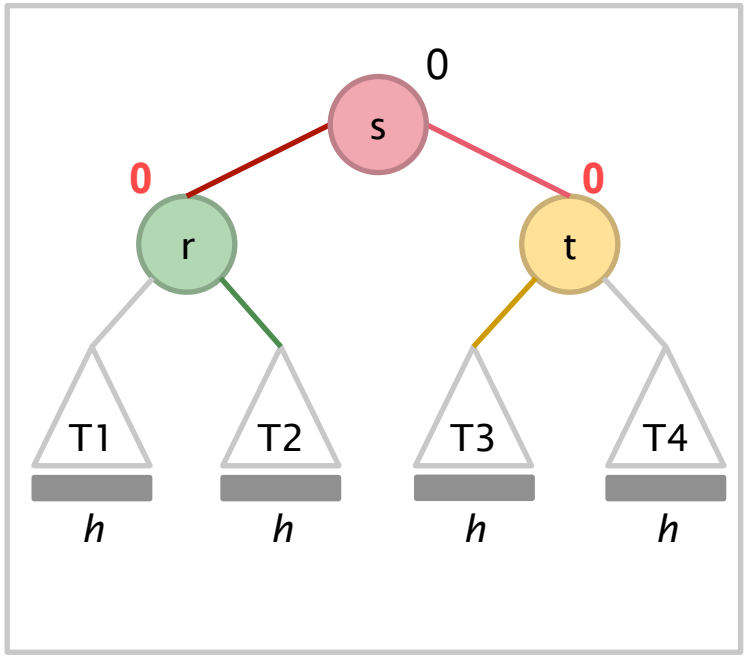





```
Avl* avl_rotacao_direita_esquerda(Avl *r) {
  Avl *t  = r->dir, *s  = t->esq;
  Avl *T2 = s->esq, *T3 = s->dir;
  s->esq = r;      s->dir = t;
  r->dir = T2;     t->esq = T3;
  r->fb = (s->fb==1)  ? -1 : 0;
  t->fb = (s->fb==-1) ?  1 : 0;
  s->fb = 0;
  return s;
}
```

Caso 2a.3 – Rotação direita-esquerda

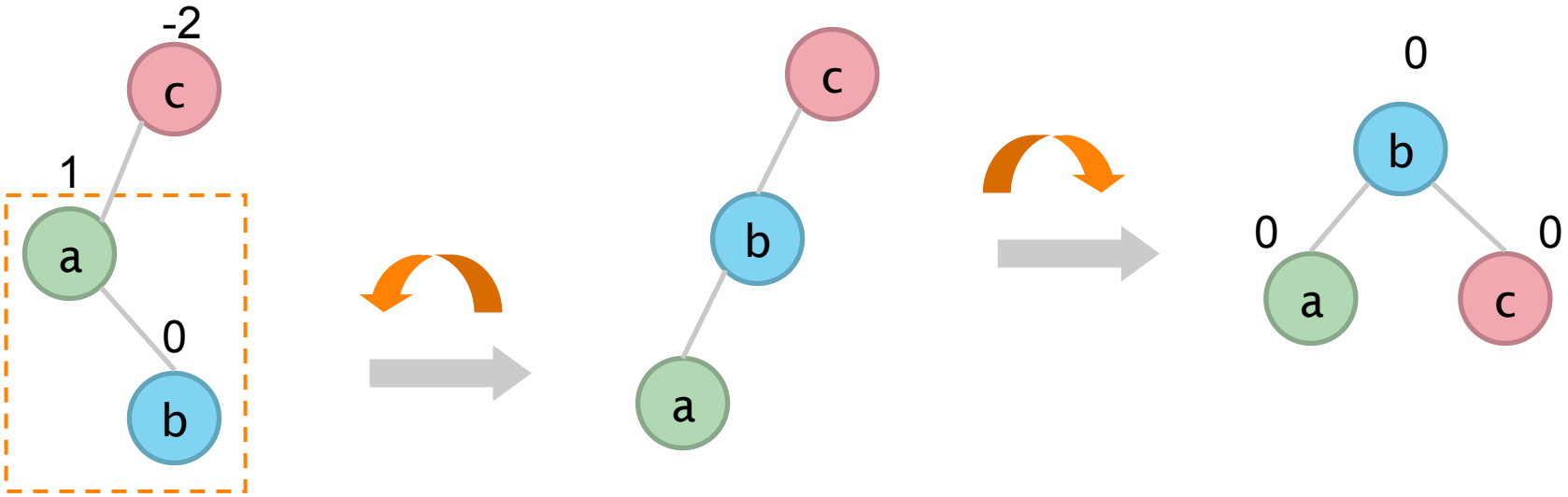





```
Avl* avl_rotacao_direita_esquerda(Avl *r) {
  Avl *t  = r->dir, *s  = t->esq;
  Avl *T2 = s->esq, *T3 = s->dir;
  s->esq = r;      s->dir = t;
  r->dir = T2;     t->esq = T3;
  r->fb = (s->fb==1)  ? -1 : 0;
  t->fb = (s->fb==-1) ?  1 : 0;
  s->fb = 0;
  return s;
}
```

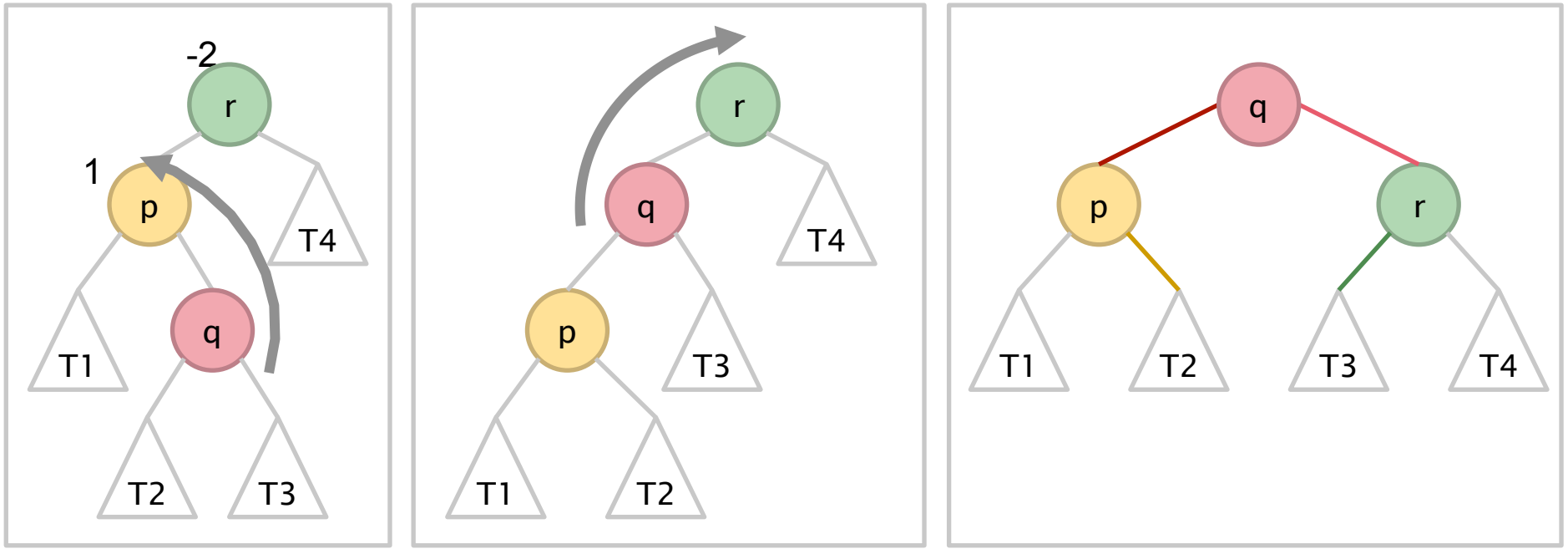
Caso 2b - Balanceamento com rotação dupla

Uma rotação à esquerda na sub-árvore à esquerda...



...seguida de uma rotação à direita

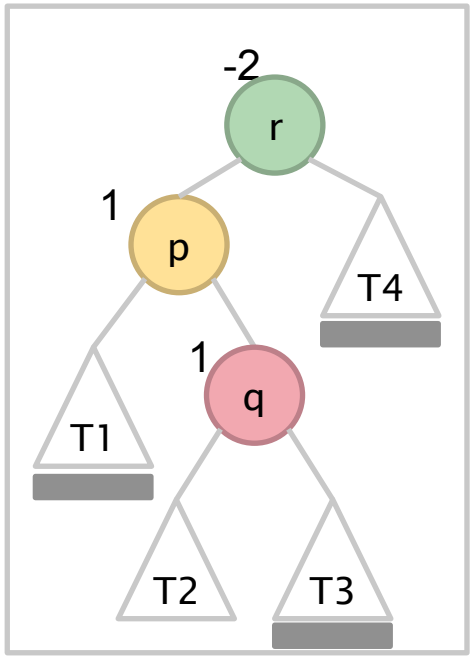
Caso 2b – Rotação esquerda-direita

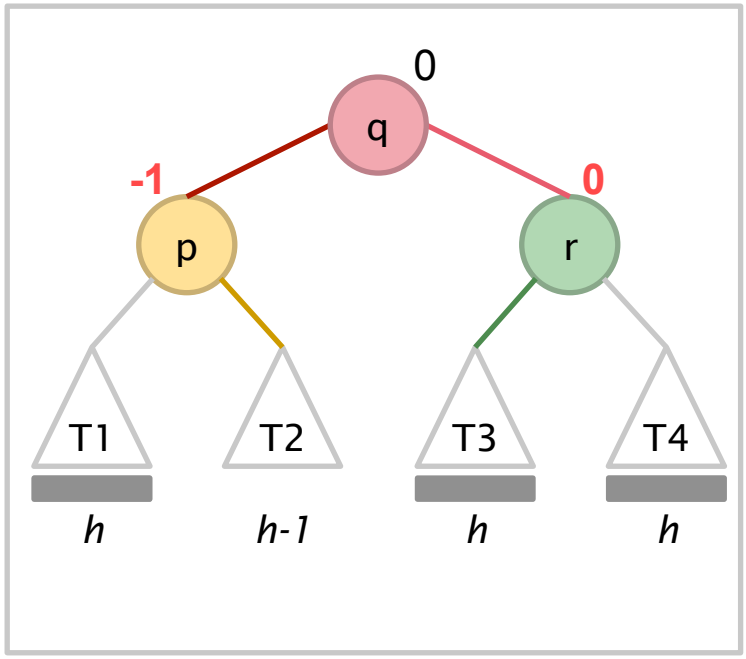



```
Avl* avl_rotacao_direita_esquerda(Avl *r) {
  Avl *p  = r->esq, *q  = p->dir;
  Avl *T2 = q->esq, *T3 = q->dir;
  q->esq = p;      q->dir = r;
  p->dir = T2;     r->esq = T3;
  p->fb = (q->fb==1)  ? -1 : 0;
  r->fb = (q->fb==-1) ?  1 : 0;
  q->fb = 0;
  return q;
}
```

(Explicação a seguir)

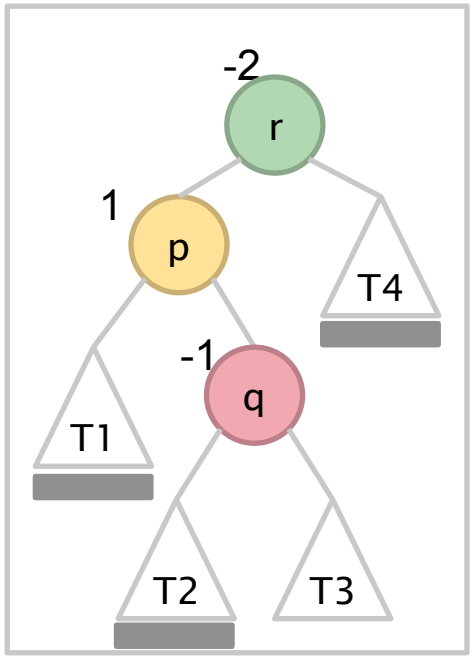
Caso 2b.1 – Rotação esquerda-direita

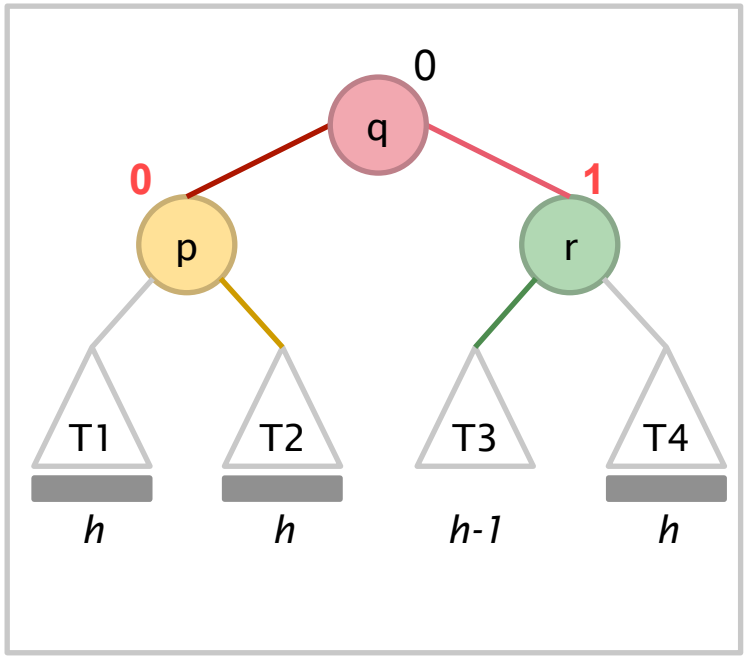





```
Avl* avl_rotacao_direita_esquerda(Avl *r) {
  Avl *p  = r->esq, *q  = p->dir;
  Avl *T2 = q->esq, *T3 = q->dir;
  q->esq = p;      q->dir = r;
  p->dir = T2;     r->esq = T3;
  p->fb = (q->fb==1)  ? -1 : 0;
  r->fb = (q->fb==-1) ?  1 : 0;
  q->fb = 0;
  return s;
}
```

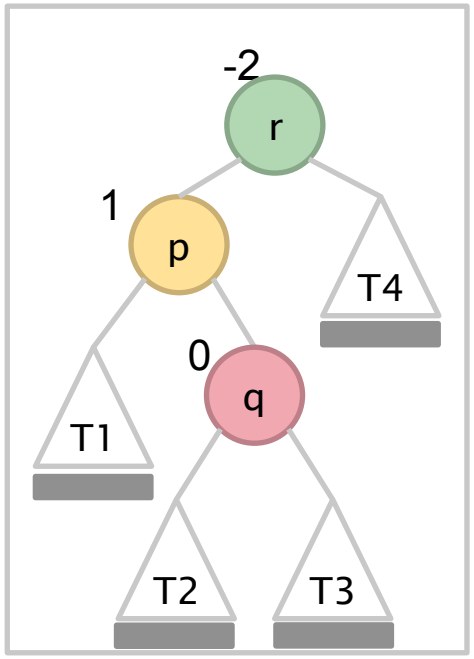
Caso 2b.2 – Rotação esquerda-direita

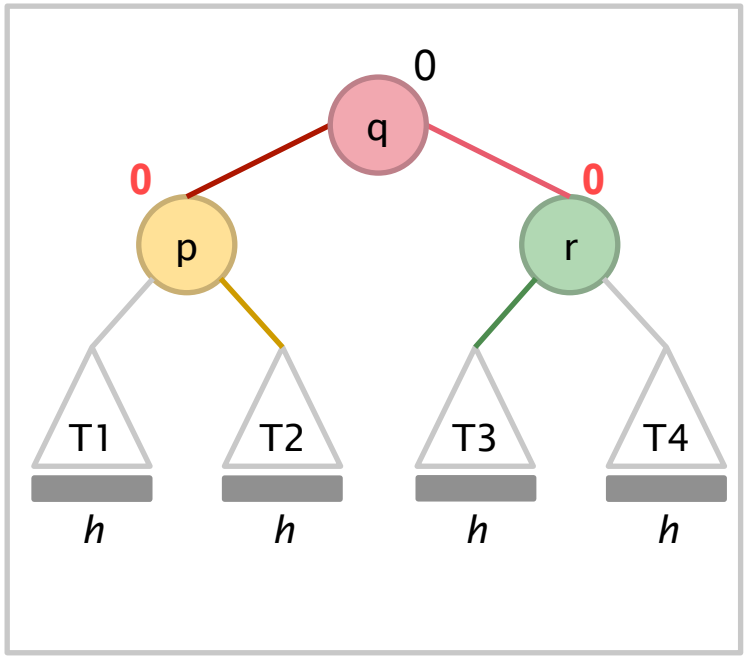





```
Avl* avl_rotacao_direita_esquerda(Avl *r) {
  Avl *p  = r->esq, *q  = p->dir;
  Avl *T2 = q->esq, *T3 = q->dir;
  q->esq = p;      q->dir = r;
  p->dir = T2;     r->esq = T3;
  p->fb = (q->fb==1)  ? -1 : 0;
  r->fb = (q->fb==-1) ?  1 : 0;
  q->fb = 0;
  return s;
}
```

Caso 2b.3 – Rotação esquerda-direita

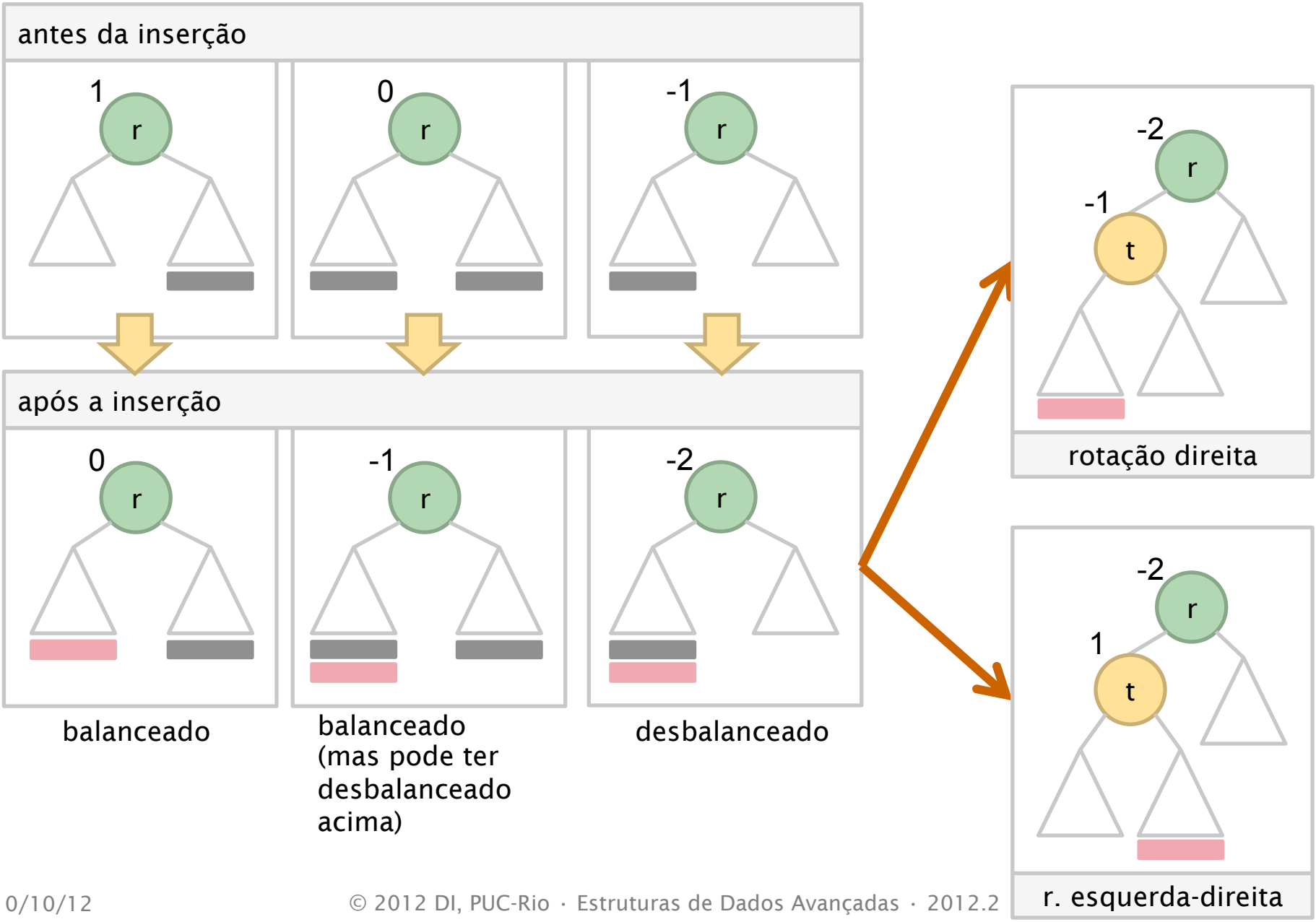





```
Avl* avl_rotacao_direita_esquerda(Avl *r) {
  Avl *p  = r->esq, *q  = p->dir;
  Avl *T2 = q->esq, *T3 = q->dir;
  q->esq = p;      q->dir = r;
  p->dir = T2;     r->esq = T3;
  p->fb = (q->fb==1)  ? -1 : 0;
  r->fb = (q->fb==-1) ?  1 : 0;
  q->fb = 0;
  return s;
}
```

Inserção

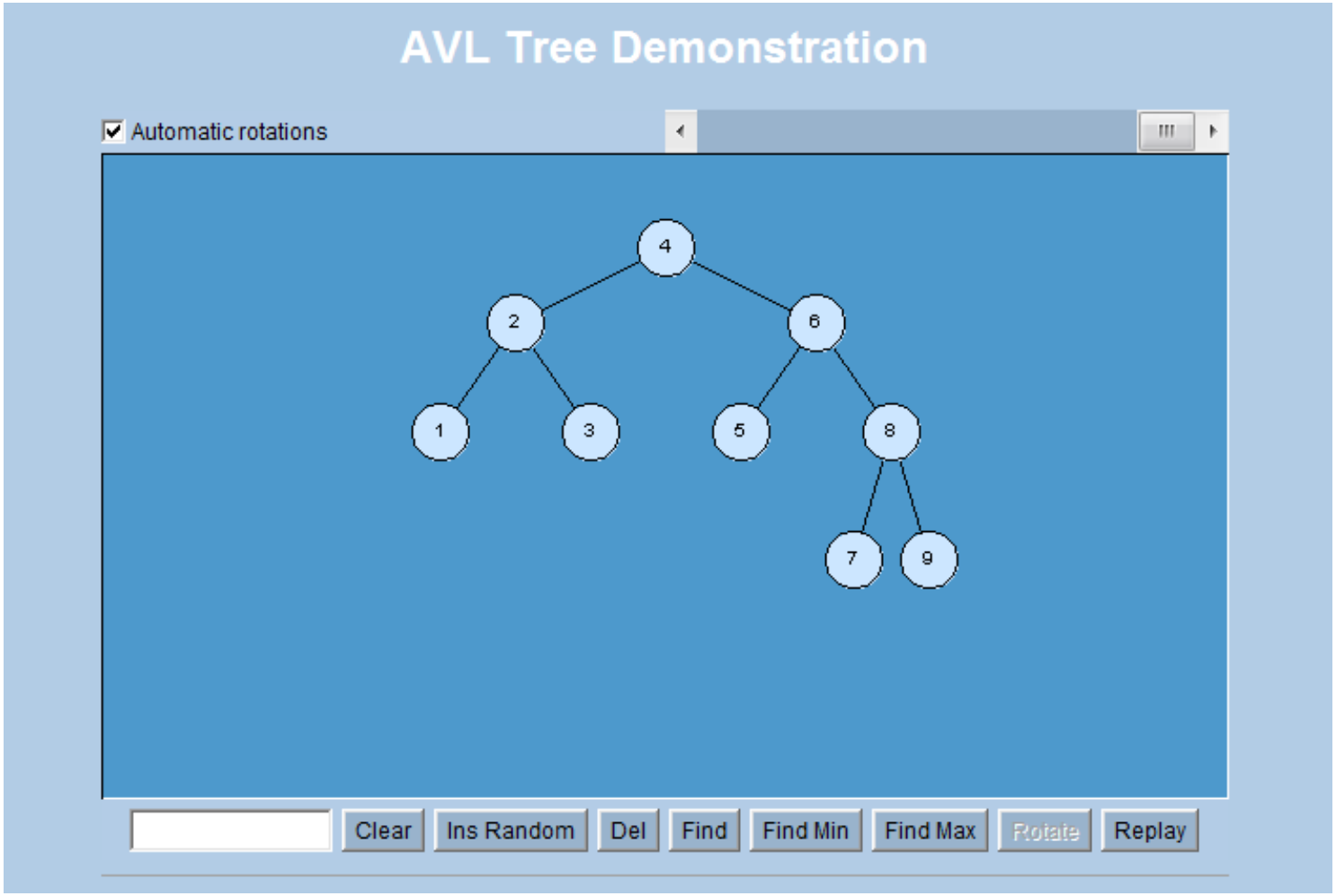
Inserção à esquerda



```
static Avl* avl_insere2(Avl* r,int chave, int* flag) {
  if (r==NULL) {
     r = (Avl*) malloc(sizeof(Avl));
     r->esq = r->dir = NULL;
     r->chave = chave;
     r->fb = 0;
     *flag = 1;
  }
  else if (r->chave > chave) {
     r->esq = avl_insere2 (r->esq, chave, flag);
     if (*flag) {       /* r cresceu à esquerda (ou seja, he aumentou)*/
       switch(r->fb) { /* análise do fator de balanceamento de r */
         case  1: /* antes: hd>he xxx depois: hd=he pois he aumentou */
           r->fb =  0; *flag = 0; break;
         case  0: /* antes: hd=he xxx depois: hd<he pois he aumentou */
           r->fb = -1; break;
         case -1: /* antes: hd<he xxx depois: hd-he=-2 pois he aumentou */
           if (r->esq->fb == -1) r = avl_rotacao_direita(r);
                 else r = avl_rotacao_esquerda_direita(r);
           *flag = 0; break;
       }
     }
  }
  else {
     r->dir = avl_insere2 (r->dir, chave, flag);
     if (*flag) {       /* r cresceu à direita (ou seja, he aumentou)*/
       switch(r->fb) { /* análise do fator de balanceamento de r */
         case -1: /* antes: hd<he xxx depois: hd=he pois he aumentou */
           r->fb =  0; *flag = 0; break;
         case  0: /* antes: hd=he xxx depois: hd>he pois he aumentou */
           r->fb = 1; break;
         case  1: /* antes: hd>he xxx depois: hd-he=2 pois he aumentou */
           if (r->dir->fb == 1) r = avl_rotacao_esquerda(r);
                 else r = avl_rotacao_direita_esquerda(r);
           *flag = 0; break;
       }
     }
  }
  return r;
}
Avl*
    i
    r
}
```

Animação de Árvores AVL

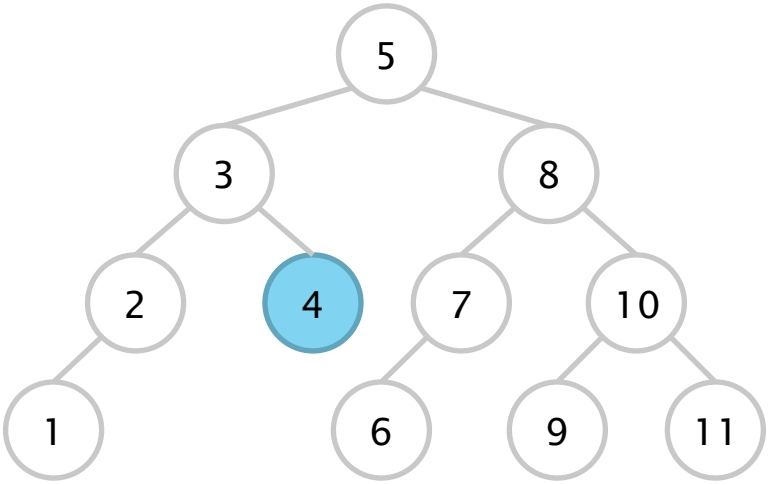
http://www.strille.net/works/media_technology_projects/avl-tree_2001/

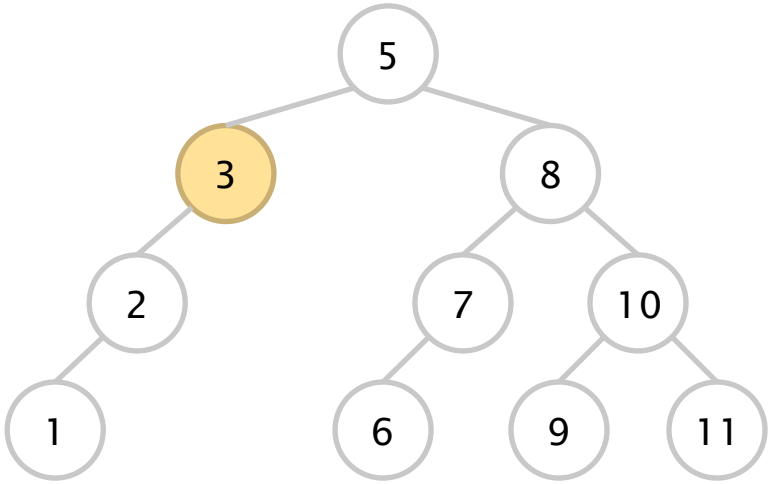


Remoções

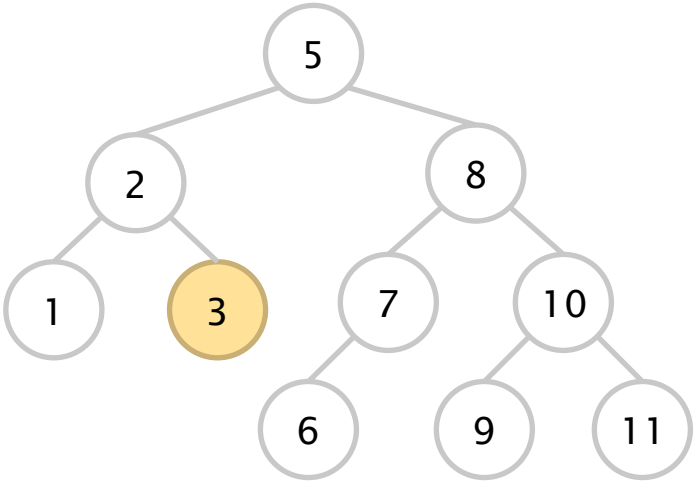
- Problemas
  - semelhantes aos das inserções, ou seja, pode ocorrer desbalanceamento
- Casos:
  - Caso 1: o nó removido é uma folha ou tem apenas 1 descendente
  - Caso 2: o nó removido possui as duas sub-árvores

Exemplo: Remove 4

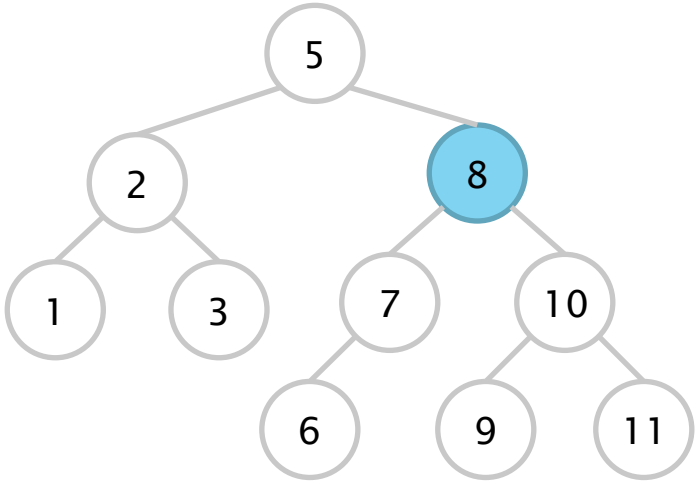


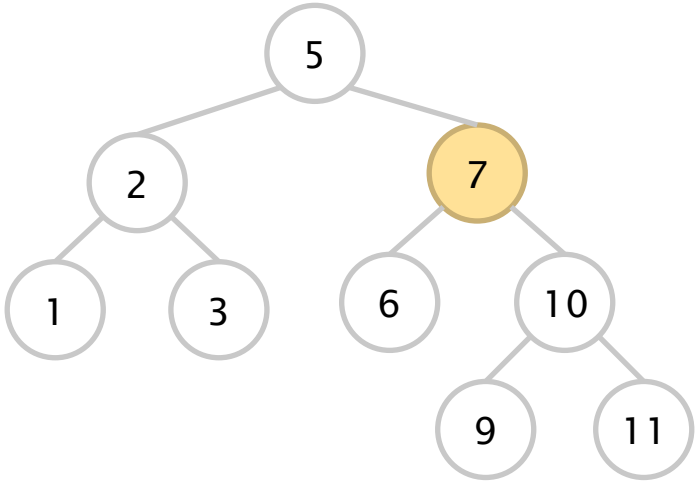


rotação à direita

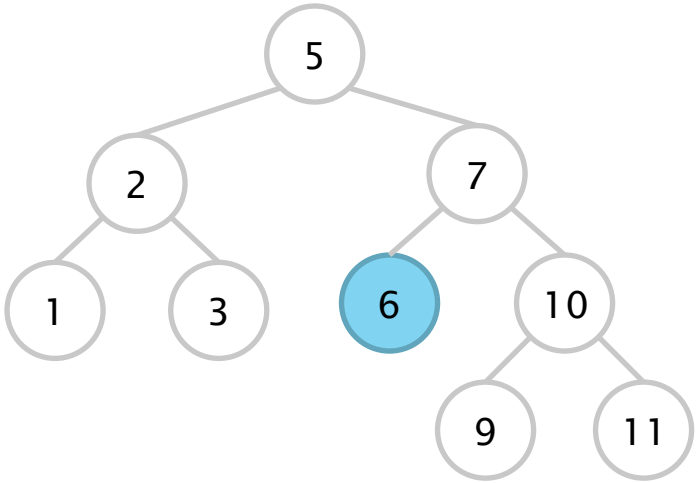


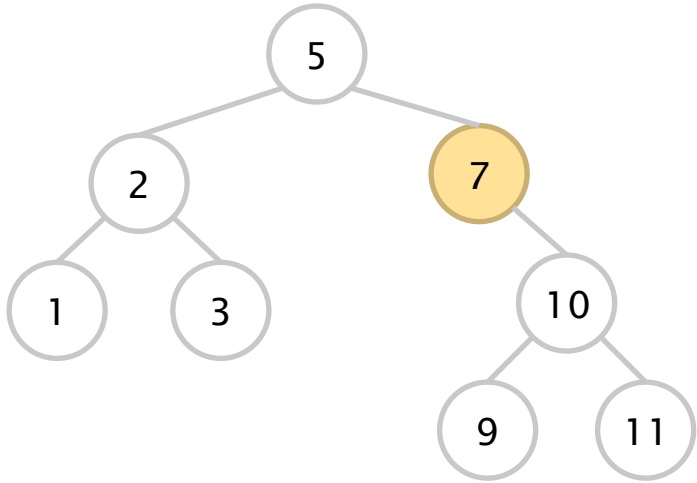
Exemplo: Remove 8



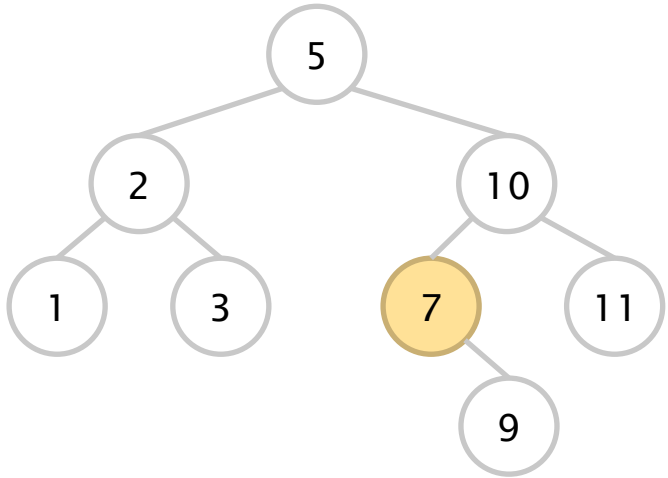


Exemplo: Remove 6





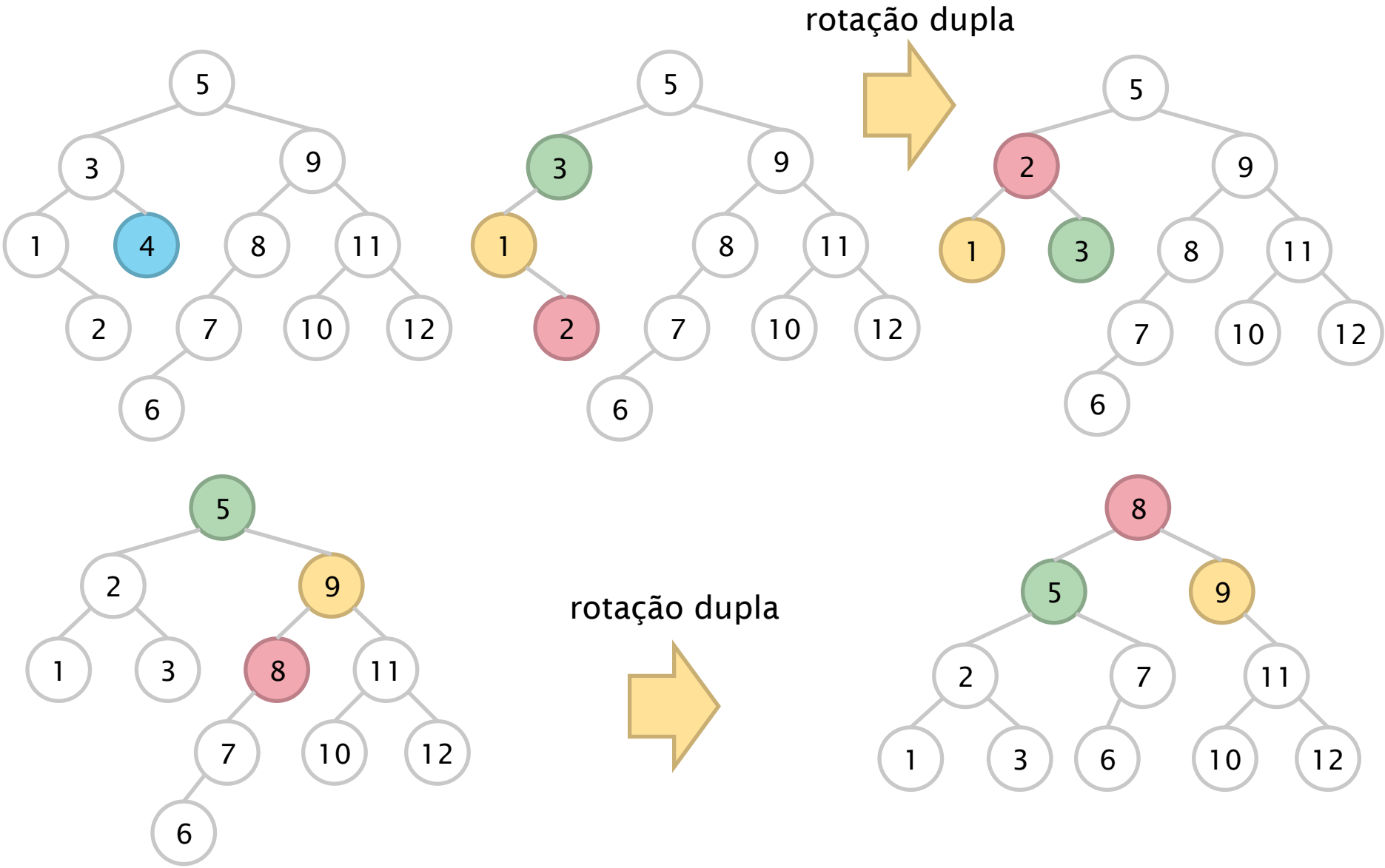
rotação à esquerda



Problemas com as Árvores AVL

- Problema:
  - Nem sempre uma ÚNICA rotação simples (ou dupla) resolve o problema de desbalanceamento dos nós
  - Há casos em que O(log n) rotações sào necessárias para tornar a árvore balanceada
- Exemplo (a seguir):
  - exclusão do nó d provoca diversas rotações

Múltiplas Rotações na Árvore AVL



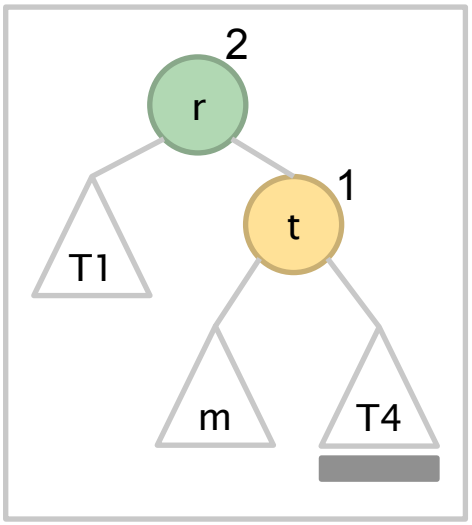
Esboço do algoritmo de remoção

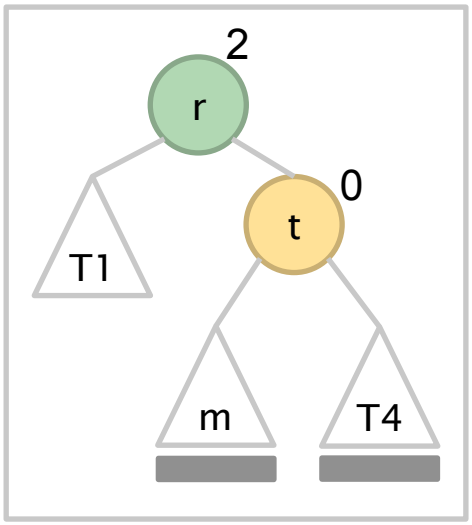
- Pesquise o nó que contém a chave procurada, aplicando recursivamente o algoritmo de remoção
- Quando achar o nó
  - faça a remoção como na árvore de busca binária
  - analise o balanceamento
  - Se o nó estiver desbalanceado (|fb| >1), faça as devidas rotações
- Recursivamente, reporte uma mudança de altura de um nó ao seu pai para que ele faça as devidas correções

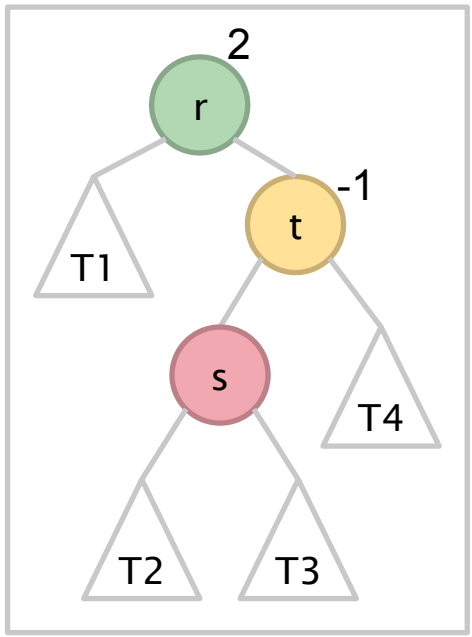
Esboço do algoritmo de remoção

- Pontos adicionais da remoção
  (que não ocorrem na inserção)
  - É necessário tratar todos os casos de alturas possíveis nas rotações
  - É necessário analisar quando
    uma sub-árvore muda de altura
    para reportar esta informação ao pai

Casos de rotação à esquerda (a s.a.e. perdeu altura)



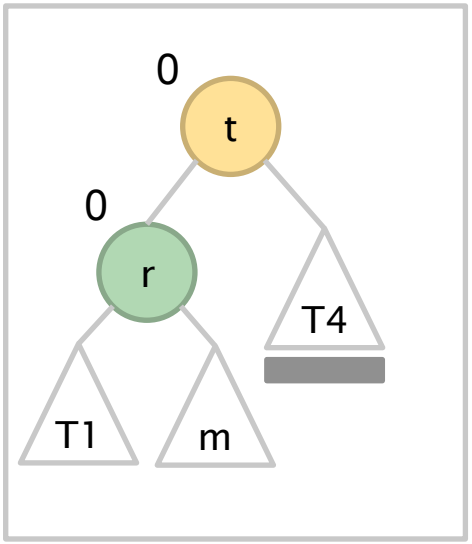


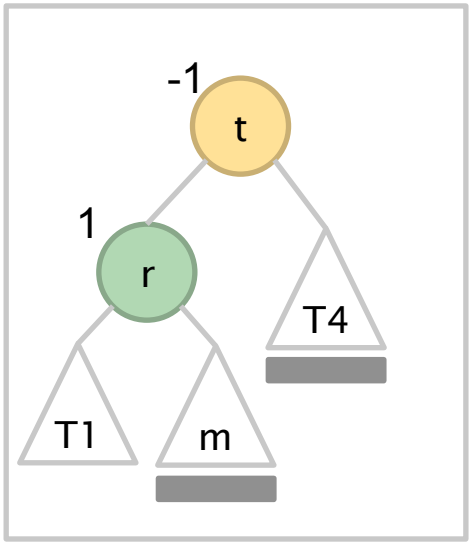


rotação dupla

rotação à esquerda

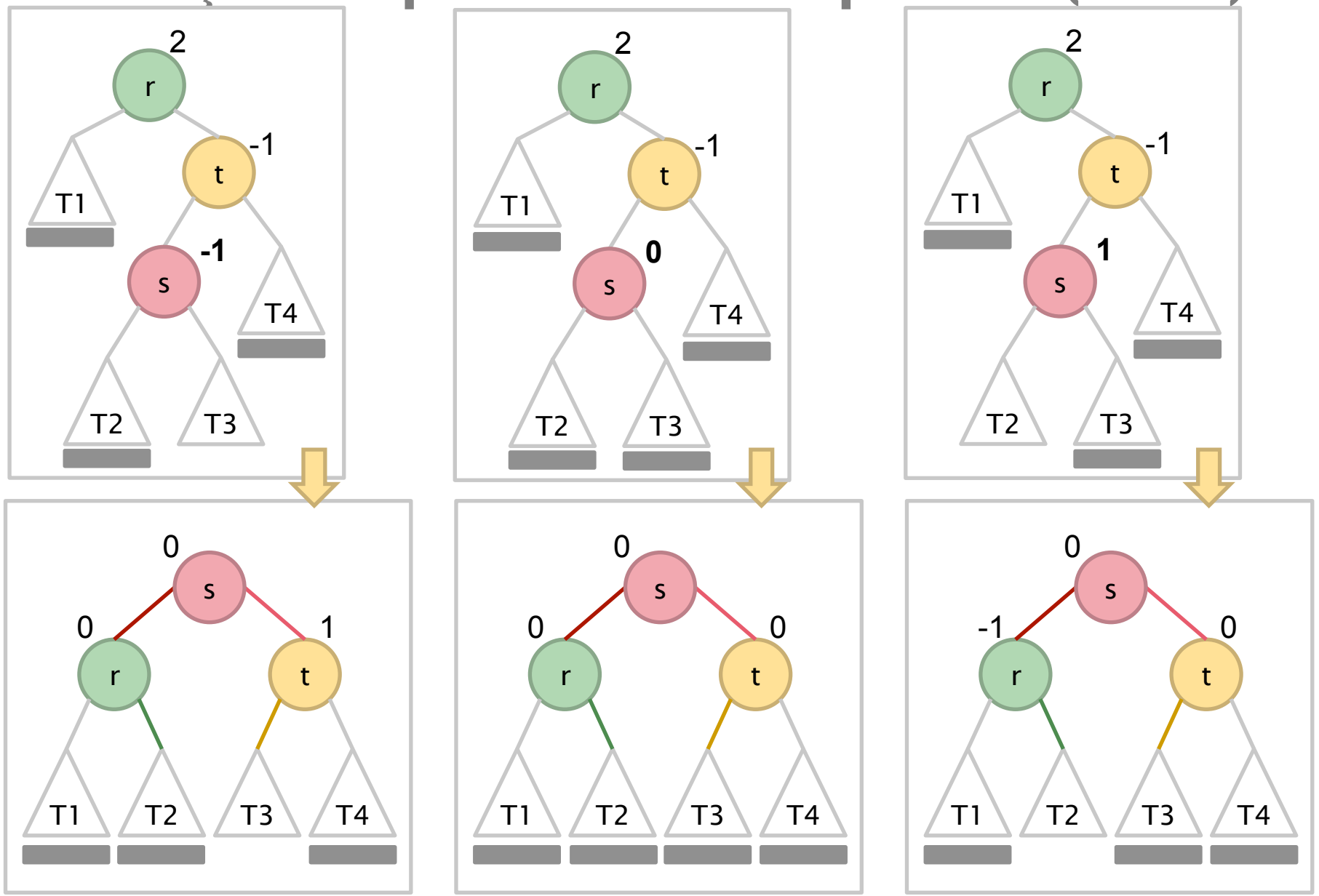
rotação à esquerda





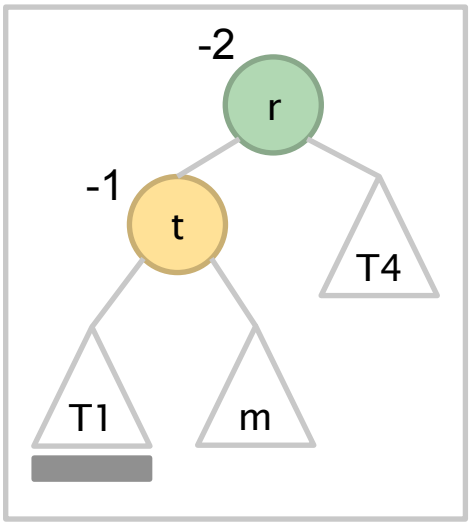

```
static Avl* rot_esquerda(Avl *r) {
  Avl *t=r->dir, *m=t->esq;
  t->esq = r;
  r->dir = m;
  if (t->fb==1) {
    t->fb =  0;  r->fb = 0; }
  else          {
    t->fb = -1;  r->fb= 1; }
  return t;
}
```

Rotação dupla direita-esquerda (fb=2)

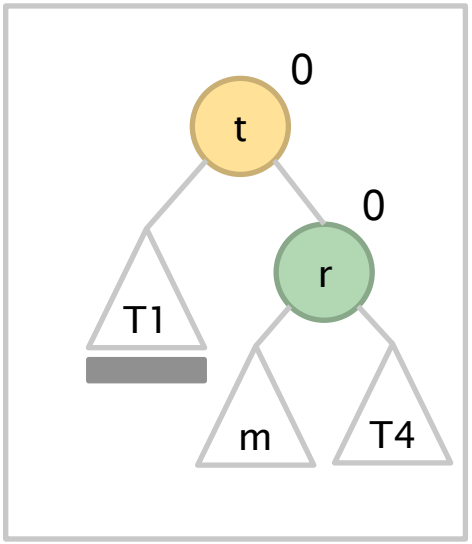


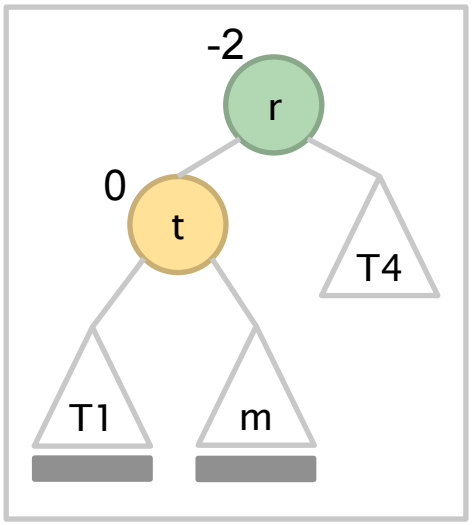
```
static Avl* rotacao_direita_esquerda(Avl *r) {
   Avl *t=r->dir,*s=t->esq,*T2=s->esq,*T3=s->dir;
   s->esq = r;
   s->dir = t;
   r->dir = T2;
   t->esq = T3;
   switch (s->fb){
      case -1:
         r->fb = s->fb = 0;
         t->fb = 1; break;
      case  0:
         r->fb = s->fb = t->fb = 0;  break;
      case  1:
         r->fb = -1;
         s->fb = t->fb = 0; break;
    }
    return s;
}
```

Casos de rotação à direita (a s.a.d. perdeu altura)

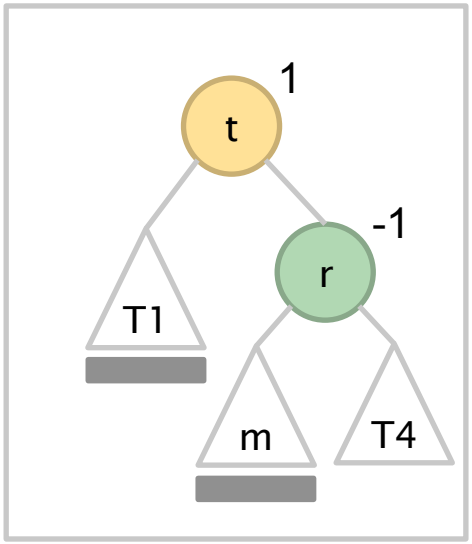


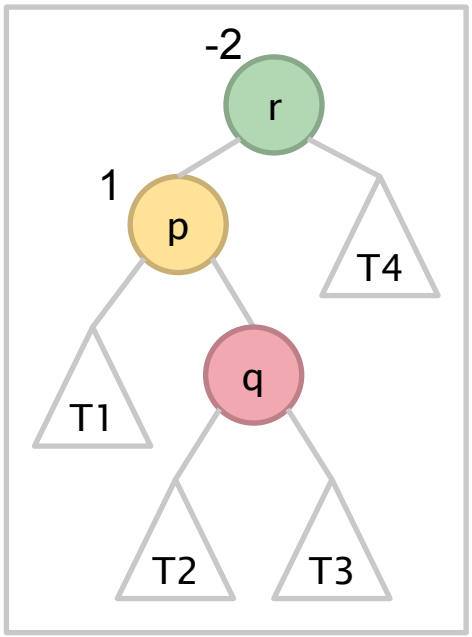
rotação à direita





rotação à direita

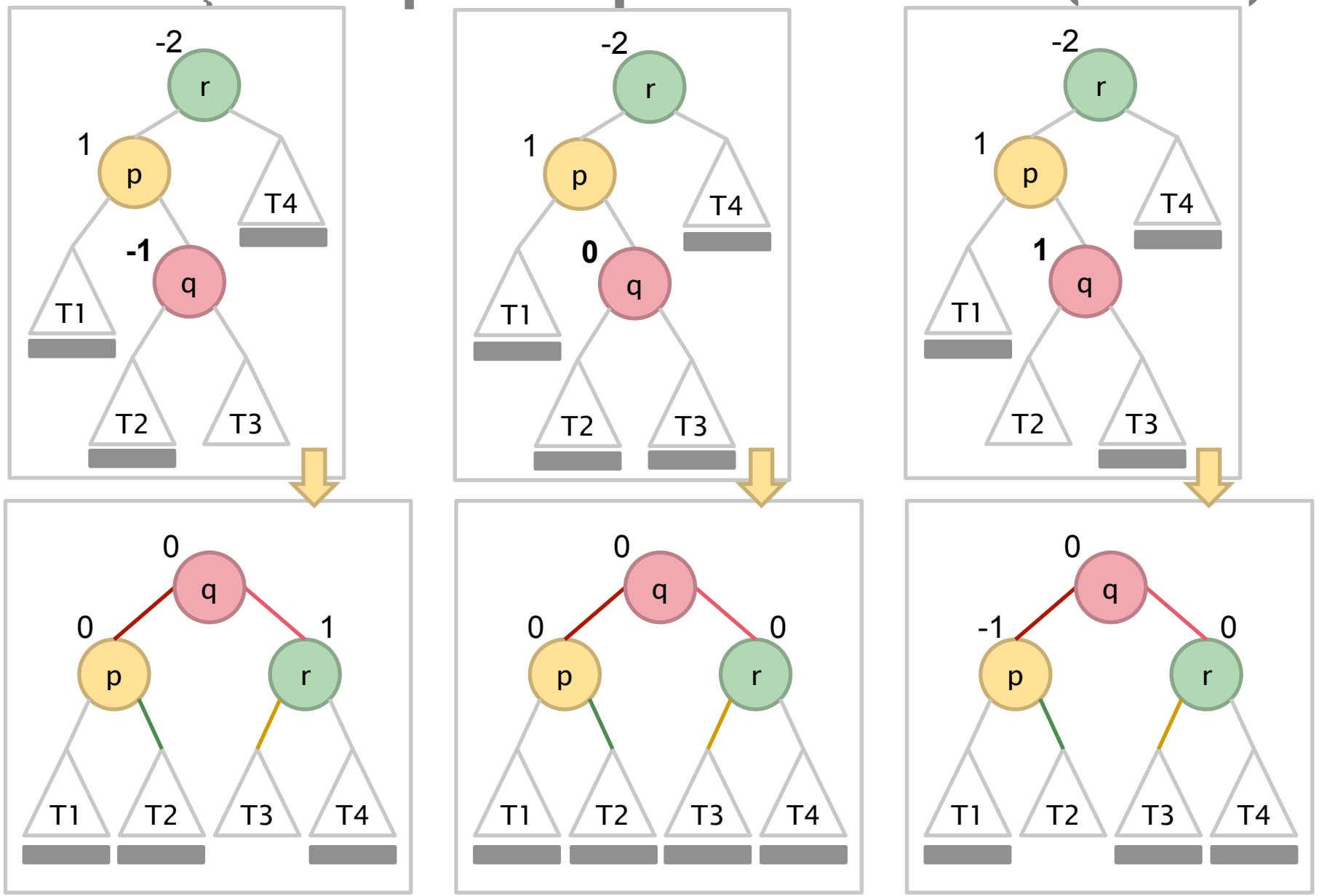




rotação dupla

```
static Avl* rot_direita(Avl *r) {
  Avl *t=r->esq, *m=t->dir;
  t->dir = r;
  r->esq = m;
  if (t->fb == -1) {
    t->fb = 0;  r->fb =  0; }
  else          {
    t->fb = 1;  r->fb = -1; }
  return t;
}
```

Rotação dupla esquerda-direita (fb=-2)



```
static Avl* rotacao_esquerda_direita(Avl *r) {
   Avl *p=r->esq,*q=p->dir,*T2=q->esq,*T3=q->dir;
   q->esq = p;
   q->dir = r;
   p->dir = T2;
   r->esq = T3;
   switch (q->fb){
      case -1:
         r->fb = 1;
         q->fb = p->fb = 0; break;
      case  0:
         r->fb = q->fb = p->fb = 0;  break;
      case  1:
         r->fb = q->fb = 0;
         p->fb = -1; break;
   }
   return q;
}
```

```
static Avl* avl_remove2(Avl *r, int chave, int *delta_h) {
  if (!r) return NULL;
  else if (chave < r->chave) {
    r->esq = avl_remove2(r->esq, chave, delta_h);
    r->fb -= *delta_h;
    if (r->fb == 2) {
      if (r->dir->fb == 1)      { r=rotacao_esquerda(r); *delta_h =-1; }
      else if (r->dir->fb == 0) { r=rotacao_esquerda(r); *delta_h = 0; }
      else if (r->dir->fb == -1)
        { r=rotacao_direita_esquerda(r); *delta_h=-1; }
    }
    else *delta_h=((r->fb==1) ? 0 : -1); /*a sad mantém a altura do nó*/
  }
  else if (chave > r->chave) { /* caso simétrico */ }
  else {  /* achou o nó para remover - remoção semelhante à abb */
    if (r->esq == NULL && r->dir == NULL) /* nó folha */
      { free(r); *delta_h = -1; r = NULL; }
    else if (r->dir == NULL)     /* só um filho, à esquerda */
       …
  }
  return r;
}

Avl* avl_remove(Avl *r, int chave) {
    int delta_h = 0;
    return avl_remove2(r, chave, &delta_h);
}
```

Relação entre altura e número de nós numa AVL

Seja N(h) o número mínimo de nós de uma AVL de altura h.

Então, no mínimo, temos uma das sub-árvores de altura h-1; a outra teria que ter, no mínimo, h-2.

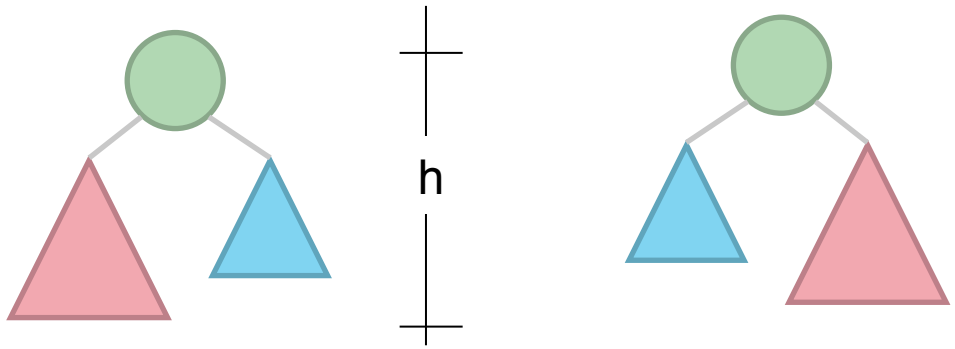
$$N(h) = N(h-1) + N(h-2) + 1$$

$$[N(h)+1] = [N(h-1)+1] + [N(h-2)+1]$$

$[N(h)+1] =$ *são números de Fibonacci*

$$N(h)+1 \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3}$$

$$h \approx 1.44\log(n))$$



TAD avl.h

```
typedef struct _avl Avl;

Avl* avl_create(int (*pcomp)(const void*,const void*), void (*pfree)(void*));

Avl* avl_free(Avl* tree);

void avl_insert(Avl* tree, void* pinfo);

void avl_remove (Avl* tree, void* pinfo);

void* avl_first (Avl* tree);

void* avl_next(Avl* tree);
```

TAD avl.c (1)

```
typedef struct _avl_node AvlNode;
struct _avl_node {
   void*    info;
   int      fb;           /* balance factor = hr - hl */
   AvlNode* parent;
   AvlNode* left;
   AvlNode* right;
};


struct _avl {
    AvlNode* root;
    AvlNode* current;
    int (*pcompare)(const void*,const void*);
    void (*pfree)(void*);
};
```

TAD avl.c (2)

```
static AvlNode* rotate_right(AvlNode *r) {
    AvlNode *parent=r->parent, *t=r->left, *m=t->right;
    t->right = r;  r->parent = t;
    r->left = m;  if (m) m->parent = r;
    t->parent = parent; if (parent) {
        if (parent->right == r) parent->right = t;
        else                    parent->left  = t;
    }
    if (t->fb==-1) { t->fb = 0; r->fb = 0; }
    else           { t->fb = 1; r->fb =-1; }
    return t;
}
```

TAD avl.c (3)

```
static AvlNode* rotate_right_left(AvlNode *r) {
   AvlNode *parent=r->parent,*t=r->right,*s=t->left,*T2=s->left,*T3=s->right;
   s->left = r;  r->parent = s;
   s->right = t;  t->parent = s;
   r->right = T2;  if (T2) T2->parent = r;
   t->left = T3;  if (T3) T3->parent = t;
   s->parent = parent; if (parent) {
      if (parent->right == r) parent->right = s;
      else                    parent->left = s;
   }
   switch (s->fb){
      case -1: r->fb=s->fb=0; t->fb=1; break;
      case  0: r->fb=s->fb=t->fb=0;  break;
      case  1: r->fb=-1; s->fb=t->fb=0; break;
   }
   return s;
}
```

TAD avl.c (4)

```
static AvlNode* insert(Avl* tree, AvlNode* r, void* info, int* delta_h) {
  if (r==NULL) {
    r = (AvlNode*) malloc(sizeof(AvlNode));
    r->left = r->right = r->parent = NULL;
    r->info = info;   r->fb = 0;  *delta_h = 1;
  }
  else if (tree->pcompare(info,r->info)<0) {
      r->left = insert(tree,r->left,info,delta_h);
      if (r->left) r->left->parent = r;
      if (*delta_h) {
        switch(r->fb){
          case  1: r->fb = 0; *delta_h = 0; break;
          case  0: r->fb = -1;   break;
          case -1: if (r->left->fb == -1)
                      r = rotate_right(r);
                   else
                      r = rotate_left_right(r);
                   *delta_h = 0;
                   break;
                 }
       }
    }
    else if (tree->pcompare(info,r->info)>0) {
.....
    return r;
}
```

TAD avl.c (5)

```
static AvlNode* remove_node(Avl* tree, AvlNode *r, void* info, int *delta_h) {
    if (!r) return NULL;
    else if (tree->pcompare(info,r->info)<0) {
        r->left = remove_node(tree,r->left, info, delta_h);
        if (r->left) r->left->parent=r;
        if (*delta_h==-1){
            r->fb++;
            if (r->fb==2){
                switch (r->right->fb){
                   case  1: { r=rotate_left(r);       *delta_h=-1; break; }
                   case  0: { r=rotate_left(r);       *delta_h= 0; break; }
                   case -1: { r=rotate_right_left(r); *delta_h=-1; break; }
                }
            }
            else
                *delta_h=(r->fb==1)?0:-1;    /* right sub-tree can sustain the node height */
        }
    }
    else if (tree->pcompare(info,r->info)>0) {

     .........
```

TAD avl.c (6)

```
    .........
}
   else {  /* found node to be removed */
      if (r->left==NULL && r->right==NULL) /* leaf */
        { if (tree->pfree) tree->pfree(r->info); free(r); *delta_h=-1; r = NULL; }
      else if (r->right == NULL)  /* only left child */
         { AvlNode* t = r; r = r->left; if (tree->pfree) tree->pfree(t->info); free(t);
           *delta_h=-1; }
      else if (r->left == NULL) /* only right child */
         { AvlNode* t = r; r = r->right; if (tree->pfree) tree->pfree(t->info); free(t);
           *delta_h=-1; }
      else { /* two children */
         AvlNode *r0=r, *successor = r->right;
         void* info2;
         while (successor->left !=NULL) successor = successor->left;
         info2 = successor->info;
         r = remove_node(tree,r,info2,delta_h);
         r0->info = info2;
      }
   }
   return r;
}
```

TAD avl.c (7)

```
void* avl_next(Avl *tree) {
        if (tree==NULL) return NULL;
        else {
                AvlNode* node = tree->current;
                if (node==NULL) return NULL;
                else if (node->right!=NULL) {  /* returns min of right sub-tree */
                        node=node->right;
                        while(node->left!=NULL) node=node->left;
                        tree->current=node;
                        return node->info;
                }  else {  /* returns the nearest ancestor that is larger than its child */
                        AvlNode* p = node->parent;
                        while ( p!=NULL && node==p->right ) {
                                node = p;
                                p = p->parent;
                        }
                        tree->current=p;
                        return (p!=NULL)?p->info:NULL;
          }
        }
}
```