Árvores AVL

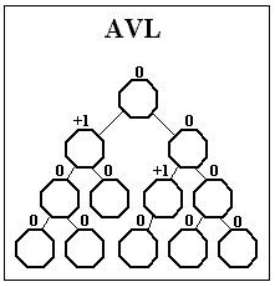
Adelson-Velskii e Landis em 1962 apresentaram uma árvore de busca binária que é balanceada com respeito a altura das sub-árvores. Uma característica importante deste tipo de árvore é que uma busca é realizada em ***O(lg(n))*** se a árvore possui n nós. Na verdade pode ser mostrado que:

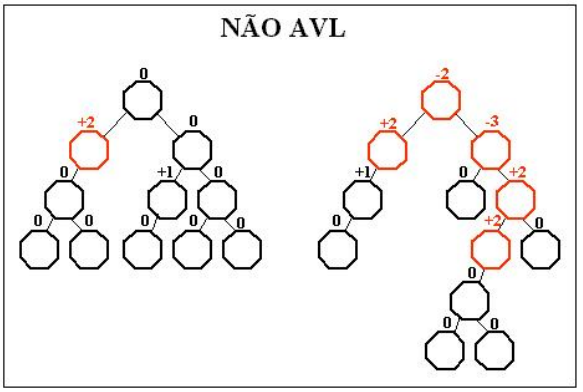
$$\log(n+1) \le h_b(n) \le 1.44\log(n+2) - 0.328$$ onde $h_b(n)$ é a altura da árvore AVL.

Definição: Uma árvore binária vazia é sempre balanceada por altura. Se T não é vazia e T_L e T_R são suas sub-árvores da esquerda e direita, então T é balanceada por altura se:

1. T_L e T_R são balanceadas por altura;
2. $|h_L \text{ e } h_R| \pm 1$, onde h_L e h_R são as alturas de T_L e T_R respectivamente. A definição de árvore binária balanceada por altura requer que toda sub-árvore seja balanceada por altura.

A definição de árvore binária balanceada por altura requer que toda sub-árvore seja balanceada por altura.





Definição: O fator de balanço de um nó X em uma árvore binária é definido como sendo $h_L - h_R$, onde e são as alturas das sub-árvores da esquerda e direita de X. Para qualquer nó X em uma árvore AVL o fator de balanço é -1, 0 ou 1.

Estrutura de um nó da Árvore

```
typedef struct no_AVL AVL;

struct no_AVL {
      int info;
      int fb; // fator de balanceamento
      AVL *pai;
      AVL *esq;
      AVL *dir;
};
```

Inserção em uma Árvore AVL

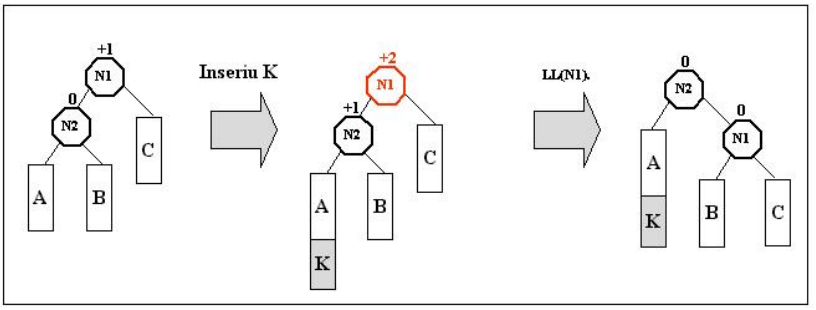
Suponha que uma árvore T é AVL e que um novo nó X seja inserido em T causando um desbalanceamento na árvore. A fim de mantermos a árvore T como AVL, precisamos de um rebalanceamento dos nós.Este rebalanceamento é realizado através de rotações no primeiro ancestral de X cujo fator de balanceamento torna-se ±2. Seja A o primeiro ancestral de X cujo fator de balanceamento torna-se ±2 após a inclusão de um novo elemento.

- **Rotação (LL):** O novo nó X é inserido na sub-árvore da esquerda do filho esquerdo de A;
- **Rotação (LR):** X é inserido na sub-árvore da direita do filho esquerdo de A;
- **Rotação (RR):** X é inserido na sub-árvore da direita do filho direito de A;
- **Rotação (RL):** X é inserido na sub-árvore da esquerda do filho direito de A.

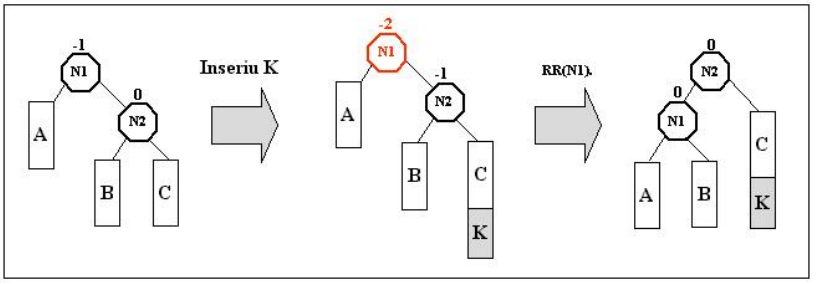
Para inserir um nó X em uma árvore AVL, basta seguirmos os seguintes passos:

1. Inserir X na árvore AVL usando o mesmo algoritmo de inserção de um nó em uma árvore de busca binária. Recursivamente, empilhar cada nó que é visitado a partir do nó raiz até X, exceto o próprio X;
2. Verificar se a pilha está vazia:
    - Se sim, o algoritmo termina.
    - Senão, vá para o passo (3).
3. Desempilhar um nó e verificar se a diferença de altura entre a sub-árvore da esquerda e da direita desse nó é maior que 1.
    - Se sim, vá para o passo (2).
    - Senão, você precisará rotacionar os nós. Depois de realizada a rotação, o algoritmo termina.

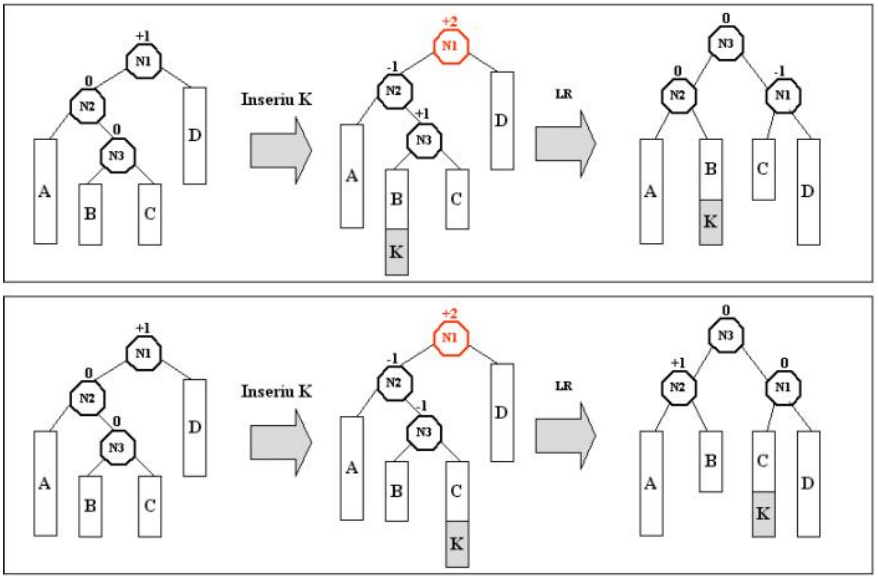
Rotação LL



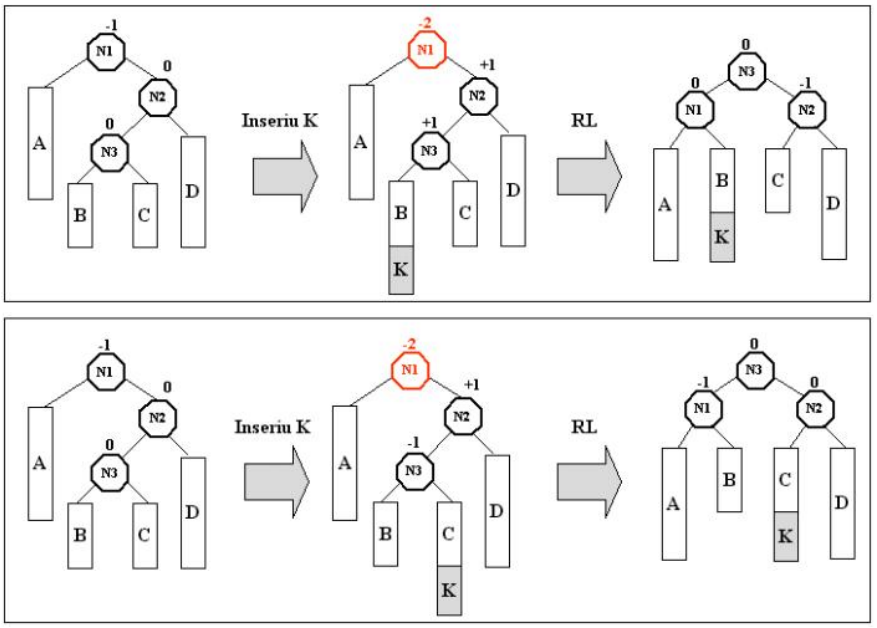
Rotação RR



Rotação em LR



Rotação RL



Note que a operação de inclusão pode ser realizada em tempo ***O(lg(n))***. Obs: Após as rotações, a árvore possui a mesma altura que antes da inclusão do novo elemento, logo os fatores de balanceamento dos elementos que não estão envolvidos nas rotações não mudam.

Algoritmo de Inserção

```
void insere_no_avl(AVL *A,float x)
{
  int f=0;
  A->raiz = insere_avl(A->raiz,x,&f);
}

NO_AVL *insere_avl(NO_AVL *raiz,float x,int *flag)
{
  if (raiz)
  {
    if (raiz->elem > x)
    {
      raiz->fesq = insere_avl(raiz->fesq,x,flag);
      raiz->fesq->pai = raiz;
      if (*flag)
      {
        switch(raiz->bal){
           case -1: raiz->bal = 0;
                    *flag = 0;
                    break;
           case 0: raiz->bal = 1;
                   break;
           case 1: if (raiz->fesq->bal == 1)
                    {
                      raiz = rotacao_LL(raiz);
                      raiz->bal = 0;
                      raiz->fdir->bal = 0;
                    }
                    else
                    {
                      raiz = rotacao_LR(raiz);
                      if (raiz->bal == 1)
                      {
                        raiz->fesq->bal = 0;
                        raiz->fdir->bal = -1;
                      }
                      else
                      {
                        raiz->fesq->bal = 1;
                        raiz->fdir->bal = 0;
                      }
                      raiz->bal = 0;
                    }
                    *flag = 0;
                    break;
        }
      }
    }
    else
    {
      raiz->fdir = insere_avl(raiz->fdir,x,flag);
      raiz->fdir->pai = raiz;
      if (*flag)
      {
        switch(raiz->bal){
           case -1:
                    break;
           case 0: raiz->bal = -1;
                    break;
           case 1:
                    break;
         }
       }

    }
 }
 else
 {
    raiz = (NO *) malloc(sizeof(NO));
    raiz->fesq = raiz->fdir = raiz->pai = NULL;
    raiz->elem = x;
    raiz->bal = 0;
    *flag = 1;
 }
 return(raiz);
}
```

/* Exercicio: As rotações RR e RL nao foram implementadas, atualize o código a fim de incorporar tais rotações */

Implementação das Rotações

```
typedef struct _no NO_AVL;
typedef struct _avl AVL;

struct _no{
  float elem;
  NO_AVL *pai;     /* ponteiro para pai */
  NO_AVL *fesq;    /* ponteiro filho da esquerda */
  NO_AVL *fdir;    /* ponteiro filho da direita */
  int bal;         /* fator de balanceamento */
};

struct _abb{
  NO_AVL *raiz;
  int altura;
  int n_elem;
};
```

```
/* rotação tipo LL */
NO_AVL *rotacao_LL(NO_AVL *desb)
{
  NO_AVL *aux;

  aux = desb->fesq;
  if (desb->pai)            /* verifica se desb não é a raiz */
         if (desb->pai->fesq == desb)
                desb->pai->fesq = aux;
         else
                desb->pai->fdir = aux;

  aux->pai = desb->pai;
  desb->fesq = aux->fdir;
  if (desb->fesq)
         desb->fesq->pai = desb;
  aux->fdir = desb;
  desb->pai = aux;

  return(aux);
}
```

```
/* rotação tipo RR */
NO_AVL *rotacao_RR(NO_AVL *desb)
{
  NO_AVL *aux;

  aux = desb->fdir;
  if (desb->pai)            /* verifica se desb não é a raiz */
         if (desb->pai->fesq == desb)
                desb->pai->fesq = aux;
         else
                desb->pai->fdir = aux;

  aux->pai = desb->pai;
  desb->fdir = aux->fesq;
  if (desb->fdir)
         desb->fdir->pai = desb;
  aux->fesq = desb;
  desb->pai = aux;

  return(aux);
}
```

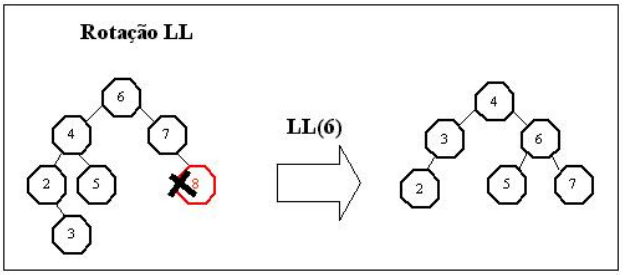
```
/* rotação tipo LR */
NO_AVL *rotacao_LR(NO_AVL *desb)
{
        rotacao_RR(desb->fesq);
        return(rotacao_LL(desb));
}
```

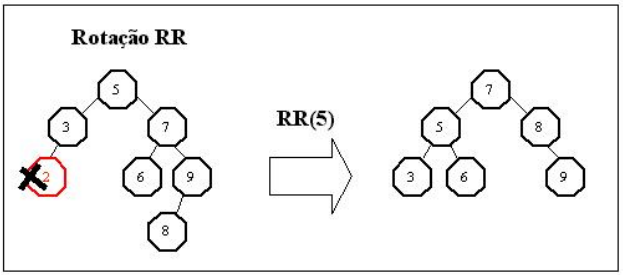
```
/* rotação tipo RL */
NO_AVL *rotacao_RL(NO_AVL *desb)
{
        rotacao_LL(desb->fdir);
        return(rotacao_RR(desb));
}
```

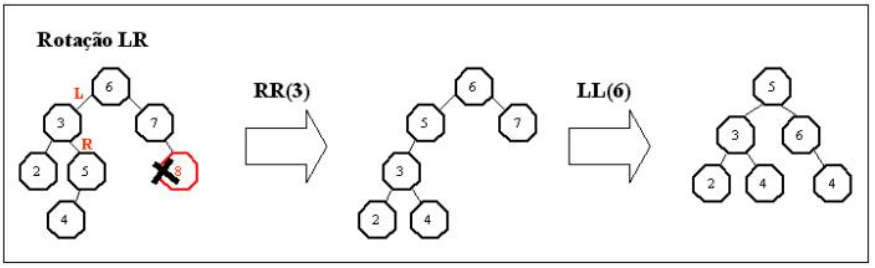
Remoção em uma Árvore AVL

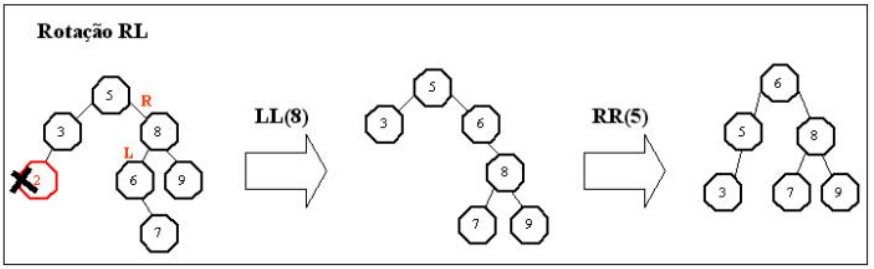
Para remover um nó s de uma árvore AVL, basta seguirmos os seguintes passos:

1. Remover X da árvore AVL usando o mesmo algoritmo de remoção de um nó em uma árvore de busca binária. Recursivamente, empilhar cada nó que é visitado a partir do nó raiz até o nó X, incluindo-o;
2. Verificar se a pilha está vazia
    - Se sim, o algoritmo termina.
    - Senão, vá para o passo (3).
3. Desempilhar um nó e verificar se a diferença de altura entre a sub-árvore da esquerda e da direita desse nó é maior que 1.
    - Se sim, você precisará rotacionar os nós. Dependendo do tipo de rotação realizada, o algoritmo pode não terminar aqui. Se ele não terminar, vá para o passo (2).
    - Senão, vá para o passo (2).

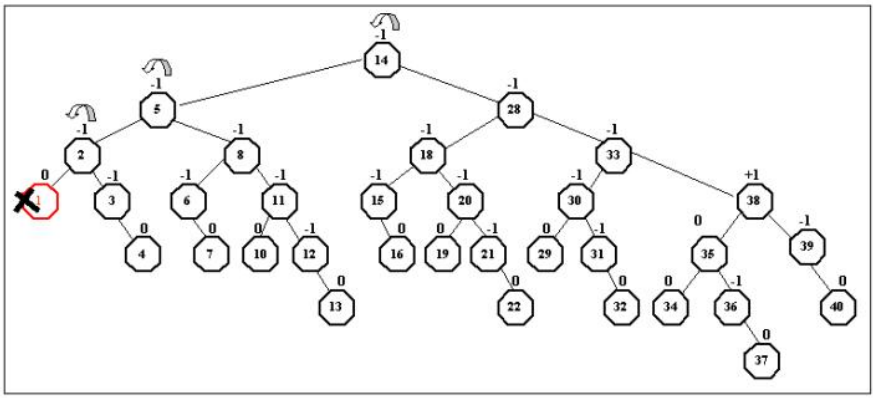








Note que a operação de remoção pode ser realizada em tempo ***O(lg(n))***. Na remoção de um elemento em uma árvore AVL, pode haver a necessidade de realizar mais de duas rotações (o que não acontece na inserção), podendo se estender para uma rotação em cada nível (***O(log(n))***) no pior caso. A figura a seguir mostra um exemplo deste caso:



Implementação da Remoção

```
/*
*  Atualização do FB e balanceamento para a raíz esquerda
*/
AVL *balanceamento_esquerdo(AVL *no, bool *h) {

    AVL *f_dir;
    int fb_dir;

    switch (no->fb) {
        case 1:
            no->fb = 0;
            break;
        case 0:
            no->fb = -1;
            *h = false;
            break;
        case -1:
                f_dir = no->dir;
                fb_dir = f_dir->fb;
                if (fb_dir <= 0) {
                    f_dir = rotacaoRR(no);
                    if (fb_dir == 0) {
                       no->fb = -1;
                       f_dir->fb = 1;
                       *h = false;
                    }
                    else {
                        no->fb = 0;
                        f_dir->fb = 0;
                    }
                    no = f_dir;
                }
                else {
                        no = rotacaoRL(no);
                        no->fb = 0;
                    }
     }
     return(no);
}

/*
*  Atualização do FB e balanceamento para a raiz direita
*/
AVL *balanceamento_direito(AVL *no, bool *h) {

    AVL *f_esq;
    int fb_esq;

    switch (no->fb) {
        case -1:
            no->fb = 0;
            break;
        case 0:
            no->fb = 1;
            *h = false;
            break;
        case 1:
            f_esq = no->esq;
            fb_esq = f_esq->fb;
            if (fb_esq >= 0) {
                f_esq = rotacaoLL(no);
                if (fb_esq == 0) {
                    no->fb = 1;
                    f_esq->fb = -1;
                    *h = false;
                }
                else {
                    no->fb = 0;
                    f_esq->fb = 0;
                }
                no = f_esq;
            }
            else {
                no = rotacaoLR(no);
                no->fb = 0;
            }
    }
    return(no);
}

/*
*  Busca nó substituto e realizada a remoção (busca o mais à direita do nó esquerdo
*/
AVL *busca_remove(AVL *no, AVL *no_chave, bool *h) {

    AVL *no_removido;
    if (no->dir != NULL) {
        no->dir = busca_remove(no->dir, no_chave, h);
        if (*h)
            no = balanceamento_direito(no, h);
    }
    else {
        no_chave->info = no->info;
        no_removido = no;
        no = no->esq;
        if (no != NULL)
            no->pai = no_removido->pai;
        *h = true;     //Deve propagar a atualização dos FB
        free(no_removido);
```

```
    }
    return(no);
}

/*
*  Remoção da Árvore AVL
*/
AVL *remove(AVL *raiz, int info, bool *h) {

    if (raiz == NULL) {
        printf("Chave não localizada !");
        *h = false;
    }
    else {
        if (raiz->info > info) {
            raiz->esq = remove(raiz->esq, info, h);
            if (*h)
                raiz = balanceamento_esquerdo(raiz, h);
        }
        else
            if (raiz->info < info) {
                raiz->dir = remove(raiz->dir, info, h);
                if (*h)
                    raiz = balanceamento_direito(raiz,h);
            }
            else { //Encontrou o elemento a ser removido
                if (raiz->dir == NULL) {
                    if (raiz->esq != NULL) //Escolhe o nó à esquerda como substituto
                        raiz->esq->pai = raiz->pai;
                    raiz = raiz->esq;
                    *h = true;
                }
                else
                    if (raiz->esq == NULL) {
                        if (raiz->dir != NULL) //Escolhe o nó à direita como substituto
                            raiz->dir->pai = raiz->pai;
                        raiz = raiz->dir;
                        *h = true;
                    }
                    else { // Busca o elemento mais à direita do nó esquerdo
                        raiz->esq = busca_remove(raiz->esq, raiz, h);
                        //Se necessário efetua balanceamento (Esquerdo pois a função
                        //busca_remove foi para o nó esquerdo)
                        if (*h)
                            raiz = balanceamento_esquerdo(raiz, h);
                    }
            }
    }
    return(raiz);
}
```
