Estruturas de Informação

aula 19: As árvores AVL

1 Introdução

Hoje nós vamos ver uma outra solução elegante para o problema de construir e manter árvores binárias de busca balanceadas.

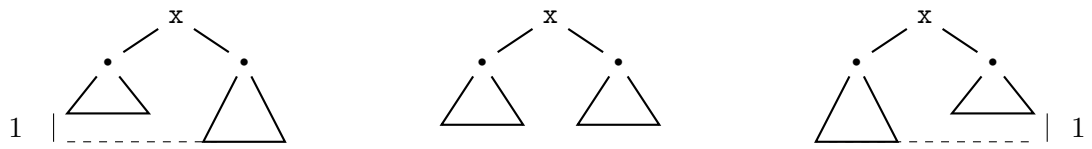
(. . .)

2 As árvores AVL

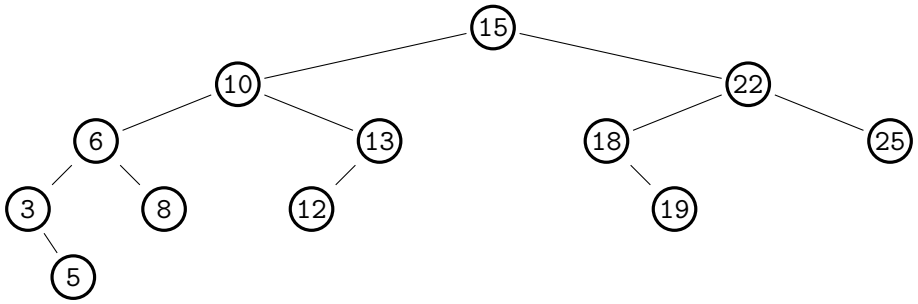
As árvores AVL controlam o balanceamento da árvore mantendo a seguinte propriedade

→ *Para cada elemento, a diferença entre as alturas das subárvores esquerda e direita nunca é maior que 1*

Abaixo nós temos uma ilustração dos 3 casos possíveis



E abaixo nós temos um exemplo de árvore AVL

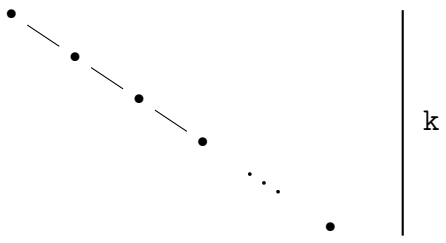


Daqui a pouco nós vamos ver como as operações de inserção e remoção são implementadas para garantir que a propriedade sempre se mantém.

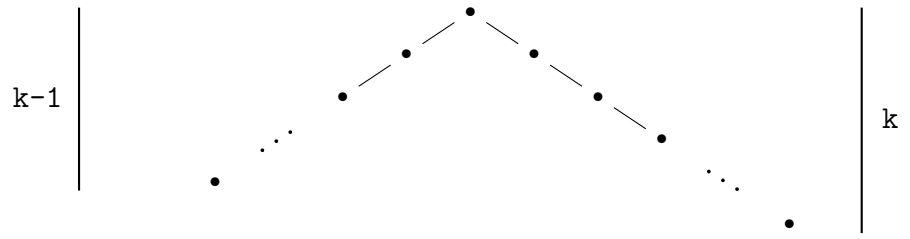
Mas antes disso, vejamos porque uma árvore com essa propriedade sempre tem altura $O(\log n)$.

A ideia é bem simples.

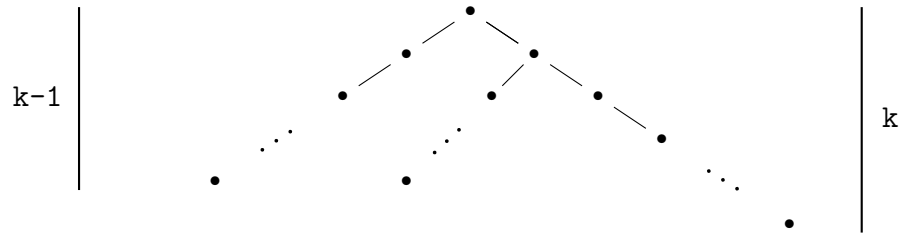
Imagine que o maior caminho da árvore tem tamanho `k` — (*de modo que* `k` *é a altura da árvore*)



Então, do outro lado deve haver um caminho com ao menos $k-1$ elementos

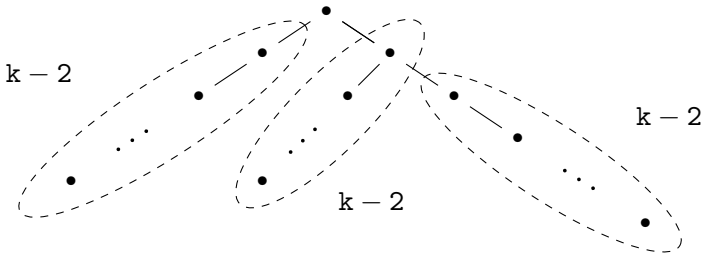


Agora, concentrando a atenção no segundo elemento do maior caminho, nós observamos que à sua esquerda deve haver um caminho com ao menos $\texttt{k}-2$ elementos



Quer dizer, nós acabamos de descobrir que

→ *A existência de um caminho de tamanho* $\texttt{k}$ *na árvore AVL*
implica a existência de 3 caminhos (disjuntos) de tamanho $\texttt{k}-2$



Ora, mas então

→ *A existência de 3 caminhos de tamanho* $\texttt{k}-2$
implica a existência de $3 \times 3 = 3^2$ *caminhos (disjuntos) de tamanho* $\texttt{k}-4$

e

→ *A existência de* 3^2 *caminhos de tamanho* $\texttt{k}-4$
implica a existência de 3^3 *caminhos (disjuntos) de tamanho* $\texttt{k}-6$

Daí, levando a repetição até o fim, nós concluímos que

→ *A existência de um caminho de tamanho* $\texttt{k}$ *na árvore AVL*
implica a existência de $3^{k/2}$ *caminhos (disjuntos) de tamanho* $\texttt{1}$

Ou, em outras palavras

→ *A existência de um caminho de tamanho* `k` *na árvore AVL*
implica que a árvore contém ao menos $3^{k/2}$ *elementos*

Denotando o número de elementos na árvore por n, nós podemos escrever

$$n \quad \geq \quad 3^{k/2}$$

E agora, tomando o $\log_3$ dos dois lados, nós chegamos ao seguinte resultado

$$\frac{k}{2} \quad \leq \quad \log_3 n$$

Daí que, toda árvore AVL tem altura $O(\log n)$.

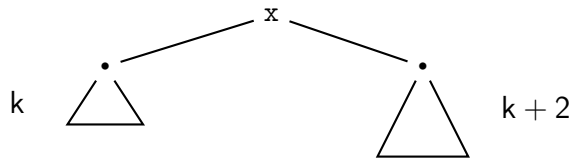
Legal, não é?

3 Inserção na árvore AVL

Agora está na hora de ver como as coisas funcionam na prática.

Quer dizer, após uma operação de inserção, a propriedade da árvore AVL pode ser violada.

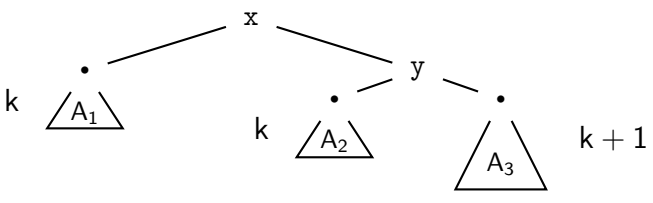
Por exemplo,



A ideia aqui é que o desbalanceamento foi provocado por uma inserção no lado direito.

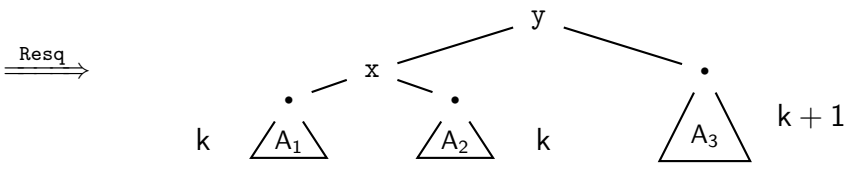
E para resolver o problema, é preciso examinar a estrutura da subárvore direita.

O caso mais simples é o seguinte

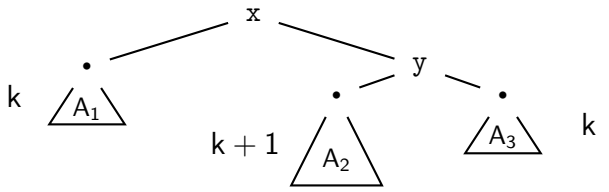


onde a inserção ocorreu à direita de `y`

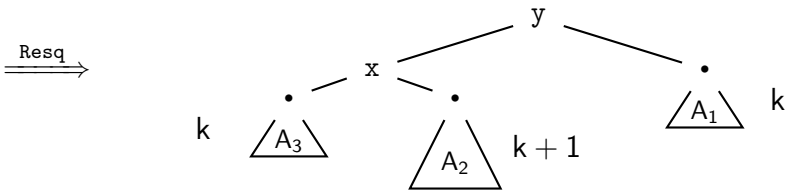
Esse caso é fácil porque basta uma rotação à esquerda para restaurar o balanceamento



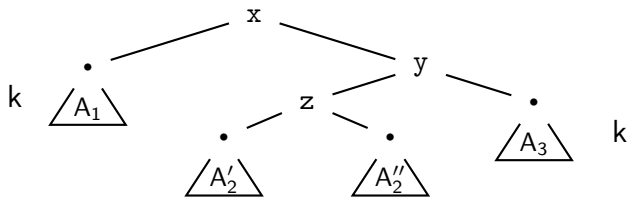
Mas, pode ser que a inserção tenha ocorrido à esquerda de `y`, produzindo a seguinte situação



E nesse caso, a rotação à esquerda apenas leva o problema para o outro lado



Daí que agora é preciso examinar a estrutura da subárvore esquerda de `y`

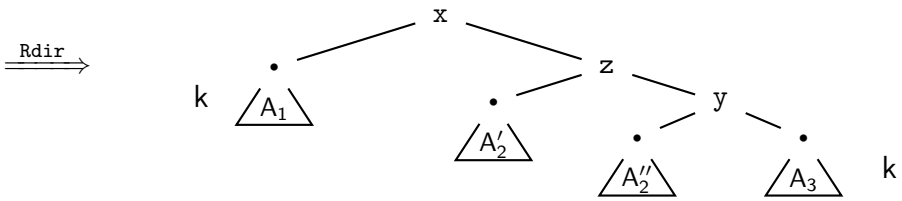


onde A_2' tem altura `k` e A_2'' tem altura `k + 1` — (*ou vice-versa*)

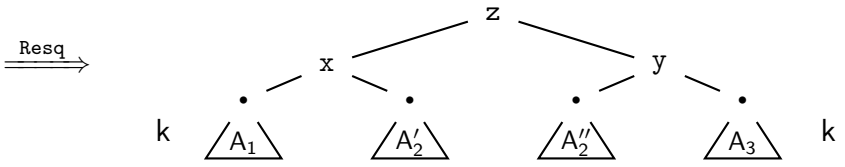
Em qualquer caso, uma operação de *rotação dupla* restaura o balanceamento de `x`.

Quer dizer,

- primeiro nós aplicamos uma rotação à direita sobre `y`



- e depois nós aplicamos uma rotação à esquerda sobre `x`



E daí, como

- A_1 e A_3 possuem altura `k`

- A'_2 tem altura `k` e A''_2 tem altura `k + 1` — (*ou vice-versa*)

após a rotação dupla a árvore se encontra balanceada.

Quando o desbalanceamento ocorre para o outro lado, a situação é basicamente análoga.

E agora nós estamos prontos para escrever o algoritmo de inserção.

3.1 O algoritmo de inserção

Para implementar as árvores AVL, nós acrescentamos o campo `.alt` ao registro `RegInt`

```
registro RegInt
 {
    int       val;
    int       alt;
    RegInt   *esq, *dir, *pai;
 }
```

e mantemos as alturas de todas as subárvores pré-calculadas.

Daí, quando um novo elemento é inserido na árvore, nós recalculamos o campo `.alt` de todos os elementos no caminho da raiz até o novo elemento — (*percorrendo o caminho de baixo para cima*)

E sempre que nós encontramos um desbalanceamento nós aplicamos as rotações adequadas para consertar a situação.

As funções que implementam as rotações serão apresentadas no apêndice dessa nota — (*porque elas são semelhantes às que vimos na aula passada*)

Abaixo nós temos o algoritmo de inserção

```
Algoritmo  Inserção-AVL (k,p)
 {
     aux  <--  Novo (RegInt)
     aux->val,alt  <--  k, 1
 aux->esq,dir,pai  <--  Nulo, Nulo, Nulo

     Se ( p = Nulo )    Retorna (aux)

     // 1a ETAPA: descida na árvore e inserção

     q1  <--  p;    q2  <--  Nulo

     Enquanto ( q1 != Nulo )
      {
          q2  <--  q1
          Se ( k < q1->val )   q1  <--  q1->esq
          Senão                q1  <--  q1->dir
      }
     Se ( k < q2->val )   q2->esq  <--  aux
     Senão                q2->dir  <--  aux

     // 2a ETAPA: subida pelo caminho e ajustes

     x  <--  q2
```

```
    Repita ( p/ sempre )
     {
         Se ( x->esq->alt  >  x->dir->alt + 1 )
          {
              y  <--  x->esq

              Se ( y->esq->alt  >  y->dir->alt )
                 x  <--  Rdir(x)
              Senão
               {
                 y  <--  Resq(y);   x  <--  Rdir(x)
               }
          }
         Senão Se ( x->dir->alt  >  x->esq->alt + 1 )
          {
              ( ... análogo ao caso anterior ... )
          }
         Senão
            x->alt  <--  Max { x->esq->alt , x->dir->alt }  +  1

         Se ( x->pai = Nulo)   Retorna (x)
         Senão                 x  <--  x->pai
 }    }
```

4 Remoção na árvore AVL

Como usual, nós começamos raciocinando sobre a remoção do menor elemento.

E como nós já sabemos, o menor elemento é encontrado fazendo a descida pela esquerda

```
q  <--  p;
Enquanto ( q->esq != Nulo )    q  <--  q->esq
```

Daí, a remoção é feita considerando dois casos fáceis

1. Se `q` é uma folha, então basta removê-lo dá árvore
2. Senão, basta colocar o seu filho direito em seu lugar

Certo.

Mas a remoção do menor elemento pode desbalancear a árvore.

Daí que, após a sua remoção, é preciso fazer a subida na árvore fazendo os ajustes necessários.

A coisa fica assim

```
função  Remove-menor-AVL (p)
 {
     Se ( p = Nulo )   Retorna (Nulo,Nulo)

     // 1a ETAPA: descida na árvore e remoção do menor

     q  <--  p
     Enquanto ( q->esq != Nulo )    q  <--  q->esq
```

```
    Se ( q->dir = Nulo )
     {
         Se ( q->pai = Nulo )   Retorna (q,Nulo)
         Senão                  q->pai->esq  <--  Nulo
     }
    Senão
     {
         Se ( q->pai = Nulo )   {  q->dir->pai  <--  Nulo
                                   Retorna (q,q->dir)
                                }
         Senão                  q->pai->esq  <--  Nulo
     }

    // 2a ETAPA: subida pelo caminho e ajustes

     ( ... semelhante à 2a etapa da inserção ... )

    Retorna (q,p)
 }
```

4.1 Algoritmo de remoção

Agora que nós já temos a função que remove (e retorna) o menor elemento, nós podemos fazer o caso geral da remoção da seguinte maneira

1. primeiro nós fazemos a descida na árvore para localizar o elemento `k`
2. se esse elemento possui algum filho `Nulo`, então a remoção é fácil
3. senão, nós removemos o menor elemento da subárvore direita, e o colocamos no lugar de `k`
4. depois, basta fazer a subida pelo caminho realizando os ajustes necessários

A coisa fica assim

```
Algoritmo  Remoção-AVL (k,p)
 {
    Se ( p = Nulo )  Retorna (Nulo)

    // 1a ETAPA: descida na árvore

    q  <--  p
    Enquanto ( q != Nulo )
     {
         Se ( k = q->val )         break
         Senão Se ( k < q->val )   q  <--  q->esq
         Senão                     q  <--  q->dir
     }
    Se ( q = Nulo )   Retorna ( p )

    // 2a ETAPA: casos fáceis

    Se ( q->esq  e  q->dir  =  Nulo )
     {
         Se ( q->pai = Nulo )     Retorna (Nulo)
         Senão
            Se ( q = q->pai->esq )   q->pai->esq   <--  Nulo
```

```
            Senão                    q->pai->dir   <--  Nulo
        x  <-- q->pai
     }
    Senão Se ( q->esq =  Nulo )
     {
        Se ( q->pai = Nulo )   {   q->dir->pai  <--  Nulo
                                   Retorna (q->dir)
                               }
        Senão Se ( q = q->pai->esq )
         {
              q->pai->esq  <--  q->dir
              q->dir->pai  <--  q->pai
         }
        Senão
         {
              q->pai->dir  <--  q->dir
              q->dir->pai  <--  q->pai
         }
        x  <-- q->pai
     }
    Senão Se ( q->dir =  Nulo )
     {
         ( ... análogo ao caso anterior ... )
     }




    // 3a ETAPA: caso geral

    Senão
     {
        q->dir->pai  <--  Nulo       // desconectando a subárvore direita

        (m,qd)  <--  Remove-menor-AVL ( q->dir )

        q->dir  <--  qd              // reconectando a subárvore direita
        Se ( qd != Nulo )
           qd->pai  <--  q

        q->val  <--  m->val          // substituindo k pelo menor

        x  <--  q
     }

    // 4a ETAPA: subida pelo caminho e ajustes

     ( ... semelhante à 2a etapa da inserção ... )
}
```