ÁRVORES BINÁRIAS

ESTRUTURA DE DADOS I SI/DCET-1

PROFESSORA: MARIA INÉS RESTOVIC

ÁRVORES

ÁRVORES

- **Definição:**
- Relação de hierarquia ou de composição entre os dados (***nós***).
- Conjunto finito T de um ou mais nós, tais que:

  (a) existe um nó denominado ***raiz*** da árvore;

  (b) os demais nós formam m >= 1 conjuntos disjuntos S1,...,Sm,

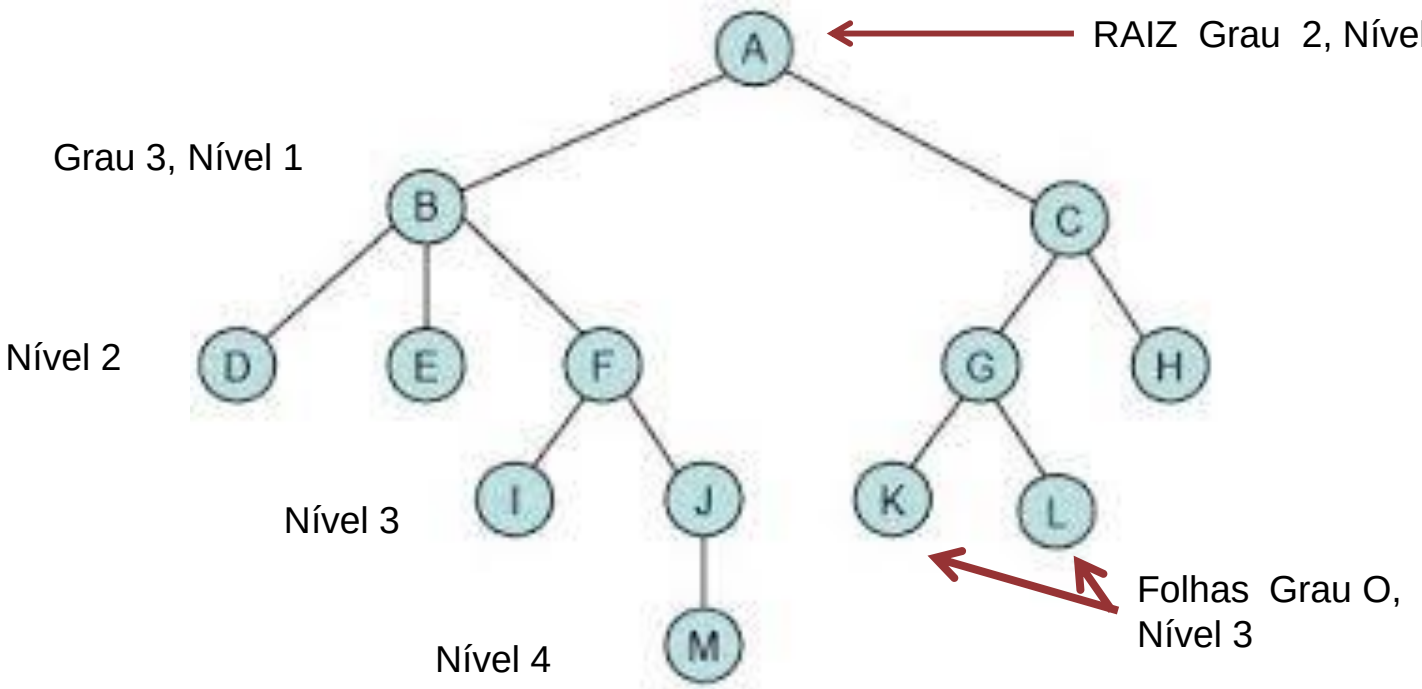
  onde cada um desses conjuntos é uma árvore.
- As árvores Si recebem a denominação de ***Sub-árvores.***

ÁRVORES

- **Terminologia:**
  - Cada nó da árvore é a raiz de uma **Sub-árvore**.
  - O número de **Sub-árvores** de um nó é o grau daquele nó.
  - Um nó de grau igual a zero é denominado folha ou nó terminal.
  - A raiz da árvore tem nível 0.
  - Os demais nós: nível = número de "linhas" que o liga à raiz.
  - Altura: nível mais alto da árvore.

ÁRVORES – representação estrutural

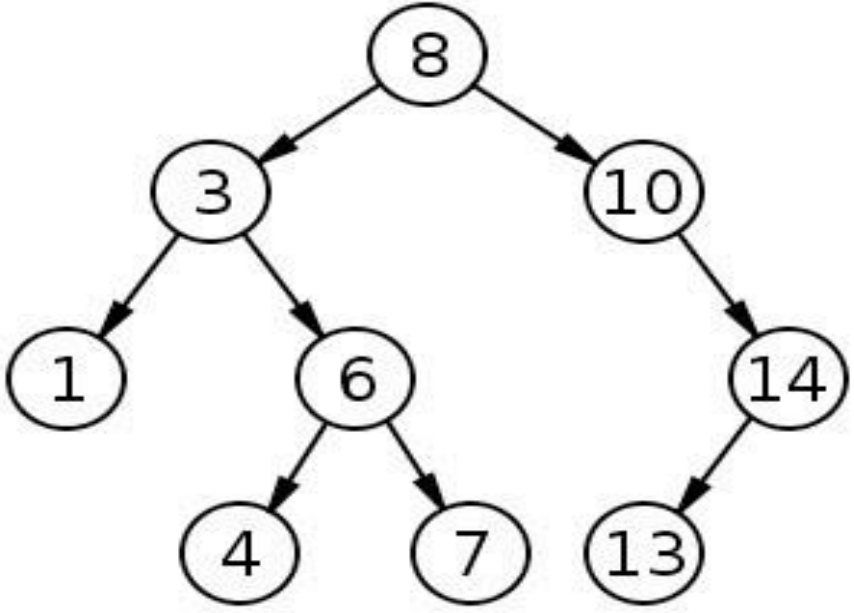
Árvore com altura igual a 4.



ÁRVORES BINÁRIAS

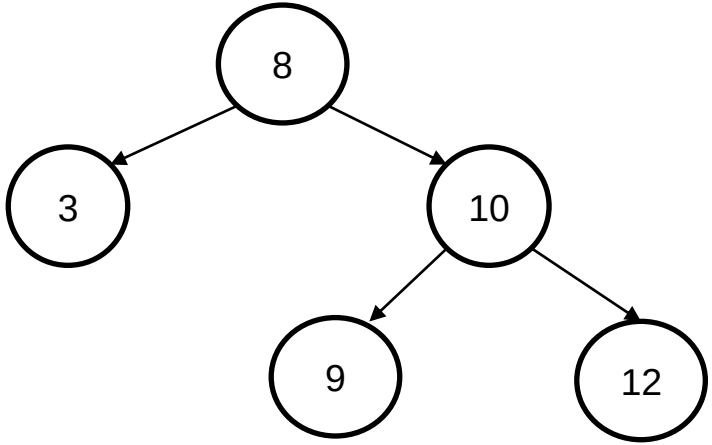
- **Definição:**
  - Uma árvore binária é uma estrutura de dados útil quando precisam ser tomadas decisões bidirecionais em cada ponto de um processo.
  - O Grau de cada nó é menor ou igual a 2 (Sub-árvores da esquerda e da direita).
  - Se grau = 1, deve ser especificado se a sua Sub-árvore é a da esquerda ou a da direita.

ÁRVORES BINÁRIAS – Representação Estrutural



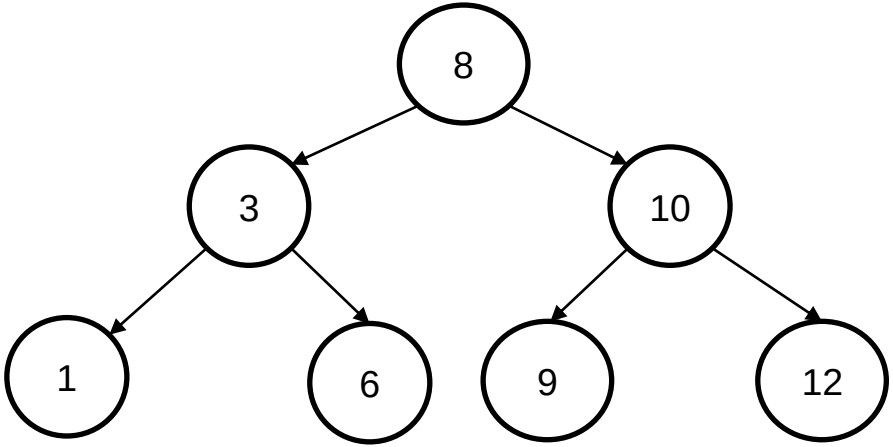
ÁRVORES BINÁRIAS

- **Árvore Estritamente Binária**: é a árvore onde todo o nó que não é folha possuí Sub-árvores a esquerda e a direita.



ÁRVORES BINÁRIAS

- Uma **árvore binária completa** é uma árvore estritamente binária sendo que todas as folhas devem estar no mesmo nível.



ÁRVORE BINÁRIA- Percurso

- **A natureza recursiva de uma árvore binária:**
  - Existem três métodos recursivos para que possamos percorrer uma árvore passando por todos os seus elementos:

ÁRVORE BINÁRIA- Percurso

- **Em Pré-ordem**

1º. visitamos a raiz
2º. Sub-árvore esq. em pré-ordem (Centro, Esquerda, Direita)
3º. Sub-árvore dir. em pré-ordem

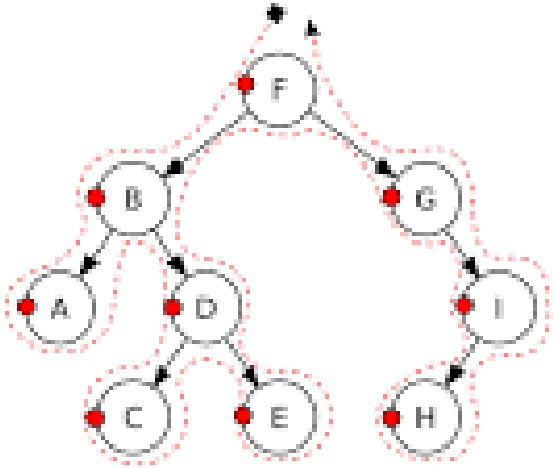
- **Em Ordem**

1º. Sub-árvore esq. em ordem
2º. visitamos a raiz (Esquerda, Centro, Direita)
3º. Sub-árvore dir. em ordem.

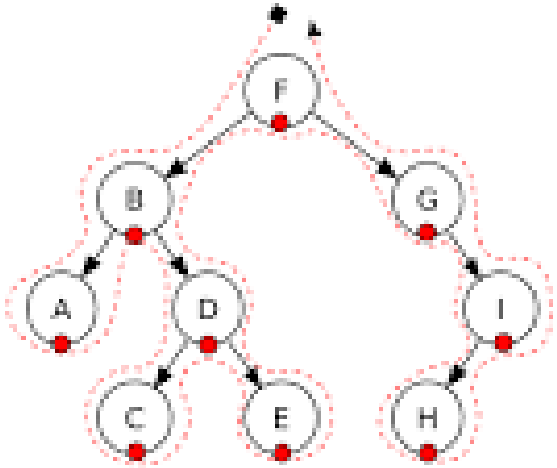
- **Em Pós- Ordem**

1º. Sub-árvore esq. em pós-ordem
2º. Sub-árvore dir. em pós-ordem (Esquerda, Direita, Centro)
3º. Visitamos a raiz

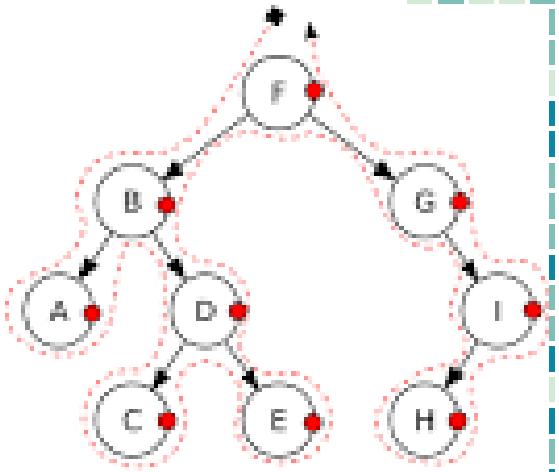
ÁRVORE BINÁRIA- Percurso



Pré-ordem: F, B, A, D, C, E, G, I, H

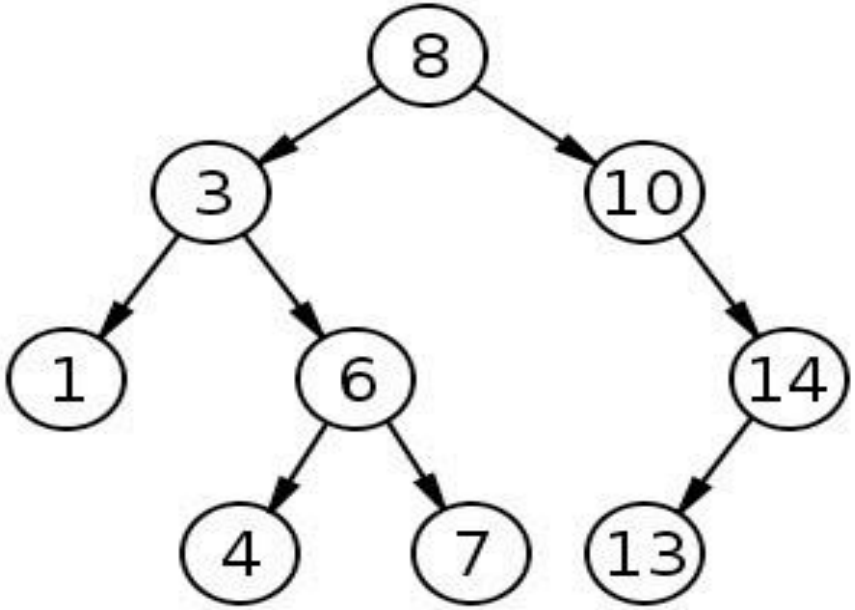


Ordem simétrica: A, B, C, D, E, F, G, H, I



Pós-ordem: A, C, E, D, B, H, I, G, F

ÁRVORE BINÁRIA- exemplo



Pré-Ordem
8, 3, 1, 6, 4, 7, 10, 14, 13

Em-Ordem
1, 3, 4, 6, 7, 8, 10, 13, 14

Pós-Ordem
1, 4, 7, 6, 3, 13, 14, 10, 8

ÁRVORE BINÁRIA de Pesquisa

- **Regra Geral de Inserção:**
  - Os valores menores devem ficar a esquerda da raiz e os maiores a direita.
  - Os valores repetidos não devem ser inseridos.
  - As inserções sempre são feitas nas folhas, dessa forma, deve se percorrer a árvore até encontrar a folha que será o pai do novo elemento a ser inserido.
  - O percurso é baseado no valor da informação que está sendo inserida. Se o novo elemento for menor que o nó comparado, deve andar para a esquerda, caso contrário deve andar para a direita.

ÁRVORE BINÁRIA- Inserção

```
typedef struct nodo{
        int info;
        struct nodo *esq,*dir;
        }Arvore;
Arvore* inserir( Arvore* raiz, int num) {
    if (raiz == NULL) {
    raiz =(Arvore *)malloc(sizeof(Arvore));
    if( raiz == NULL)exit(1);
    raiz->info =num;
    raiz->esq = NULL;
    raiz->dir = NUL
   return(raiz);}
```

```
else {
        if (num < raiz->info)
            raiz->esq = inserir(raiz->esq, num);
        else if ( num > raiz->info)
              raiz->dir = inserir(raiz->dir, num);
             else {
                  printf("\nERRO numero repetido\n");
                  system("pause");}
         return(raiz);  // retorna a raiz sem mudar
       }
}
```

ÁRVORE BINÁRIA- Percurso

```
void pre_ordem(
            Arvore *raiz){
  if (raiz == NULL) return;
  printf("%d ", raiz->info);
  pre_ordem(raiz->esq);
  pre_ordem(raiz->dir);

}
```

```
void em_ordem(
            Arvore *raiz){
  if (raiz == NULL) return;
  em_ordem(raiz->esq);
  printf("%d ", raiz->info);
  em_ordem(raiz->dir);

}
```

```
void post_ordem(
            Arvore *raiz){
  if (raiz == NULL) return;
  post_ordem(raiz->esq);
  post_ordem(raiz->dir);
  printf("%d ", raiz->info);
}
```

ÁRVORE BINÁRIA- Retirar

```
Arvore* retirar(Arvore* raiz, int num)
{ Arvore *ret;
 if (raiz != NULL)
   if (raiz->info < num)
     raiz->dir=retirar(raiz->dir, num);
   else
    if (raiz->info > num)
     raiz->esq=retirar(raiz->esq,num);
     else
if ((raiz->esq)==NULL)
      {   ret=raiz;
          raiz=raiz->dir;
          free(ret);
      }
      else
       if (raiz->dir==NULL)
       {   ret=raiz;
           raiz=raiz->esq;
           free(ret);
        }
        else
            raiz->dir=substitui(raiz, raiz->dir);
return(raiz); }
```

ÁRVORE BINÁRIA- Retirar

```
Arvore *substitui ( Arvore *raiz, Arvore *sucessor)
{ Arvore *ret;
  if ((sucessor->esq)==NULL)
  {    raiz->info=sucessor->info;
       ret=sucessor;
       sucessor=sucessor->dir;
       free(ret);
   }
  else
      sucessor->esq=substitui(raiz,  sucessor->esq);
return(sucessor);
}
```

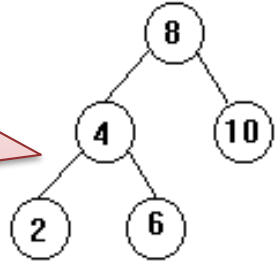
ÁRVORES AVL

Árvores AVL-Definição

O nome AVL corresponde ao dos seus criadores: Adelson, Velsky e Landis (1962)

- Uma árvore AVL é uma árvore binária de busca construída de tal modo que a altura de sua Sub-árvore direita difere da altura da Sub-árvore esquerda de no máximo 1.
- **O que pode acontecer quando um novo nó é inserido numa árvore balanceada ?**

Inserção dos nós 1, 3, 5 ou 7 requerem que a árvore seja rebalanceada!

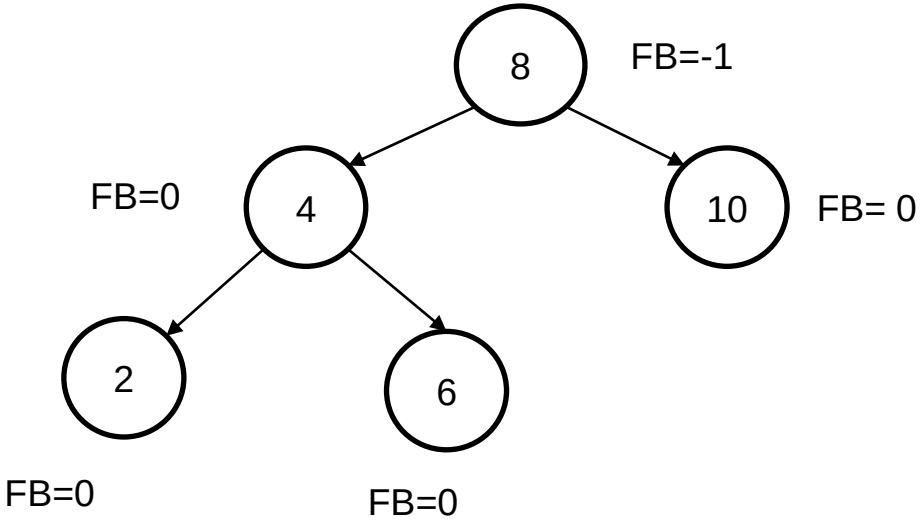


Nós 9 ou 11 podem ser inseridos sem balanceamento. Sub-árvore com raiz 10 passa a ter uma Sub-árvore e Sub-árvore com raiz 8 vai ficar melhor balanceada !

Fator de Balanceamento de um nó

- É a altura da Sub-árvore direita do nó menos a altura da Sub-árvore esquerda do nó .
- FB = altura direita – altura esquerda
- **Se todos os FB forem [-1, 0, 1] a árvore está balanceada.**

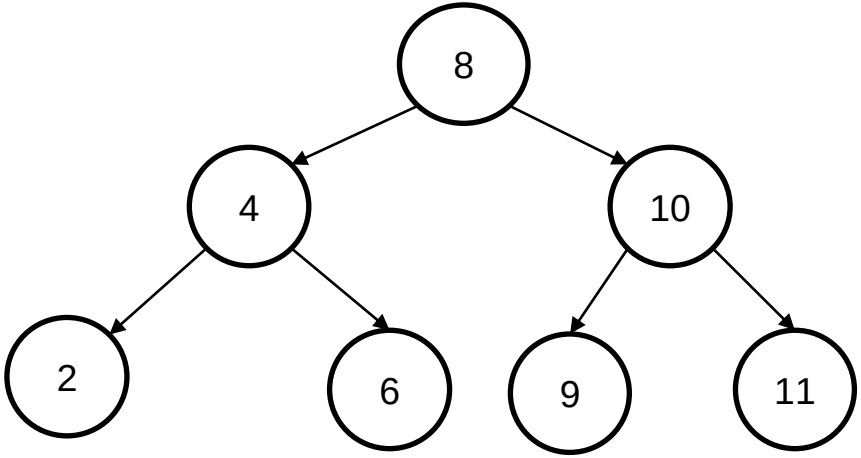
Exemplo: Fator de Balanceamento



Exemplo: Fator de Balanceamento

Nós **9** ou 11 podem ser inseridos sem balanceamento .

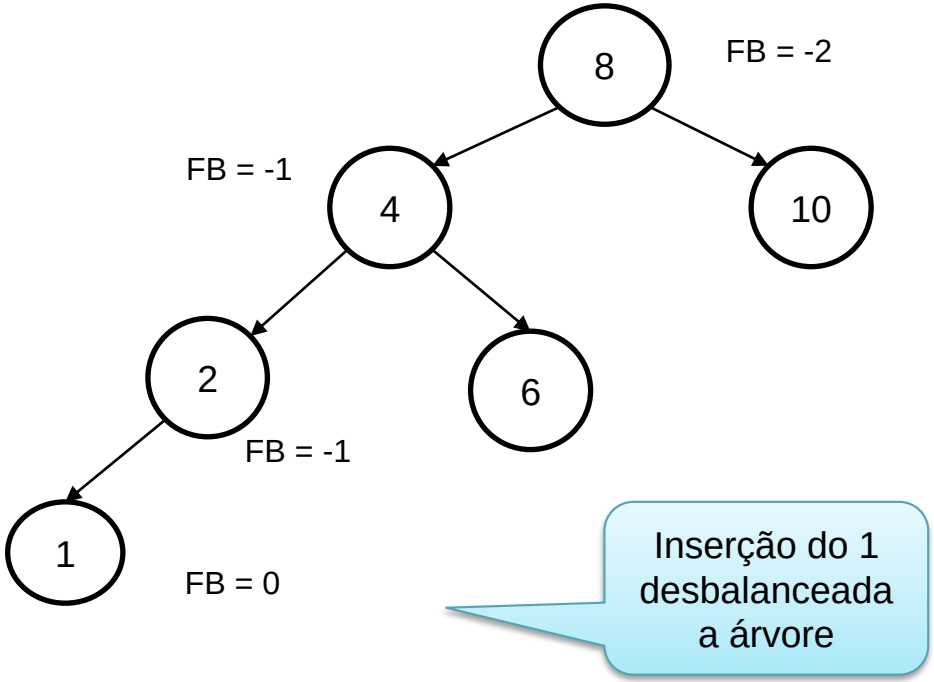
Sub-árvore com raiz **10** passa a ter uma Sub-árvore e Sub-árvore com raiz **8** vai ficar melhor balanceada !



Inserção dos nós 1, 3, 5 ou 7 requerem que a árvore seja rebalanceada!

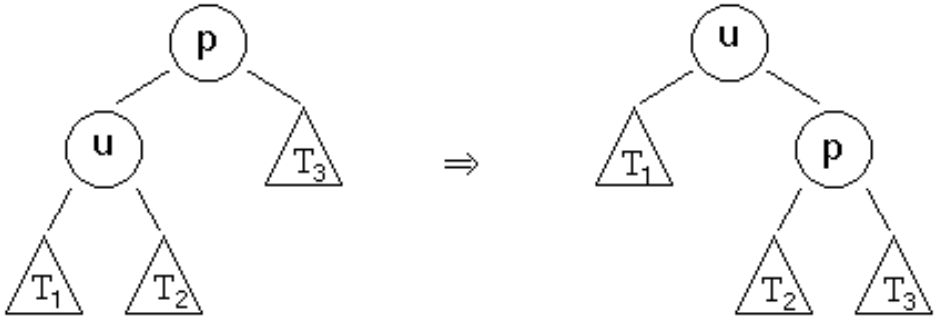
Exemplo: Fator de Balanceamento

Inserção dos nós **1, 3, 5 ou 7** requerem que a árvore seja desbalanceada!



Balanceamento

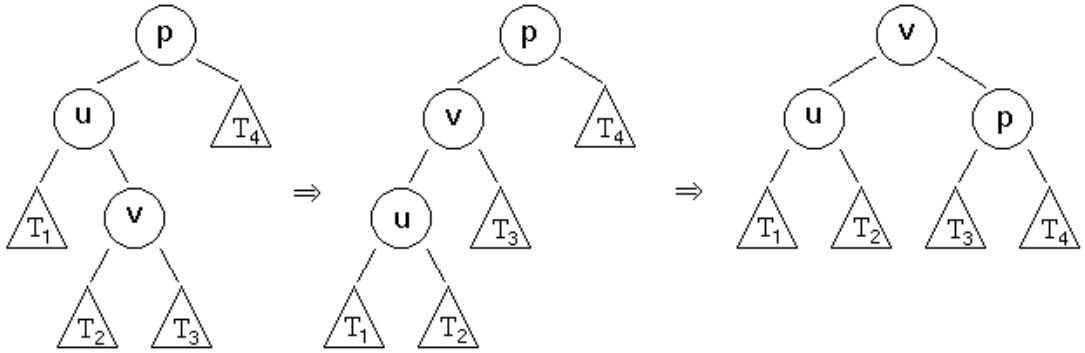
- Nos casos seguintes considere P como sendo o nó raiz de uma Sub-árvore desbalanceada e U como sendo o nó filho dessa raiz.
  - Caso 1: Altura Esquerda de P > Altura Direita de P
    - Caso 1.1 : Altura Esquerda de U > Altura Direita de U



Rotação a Direita

Balanceamento

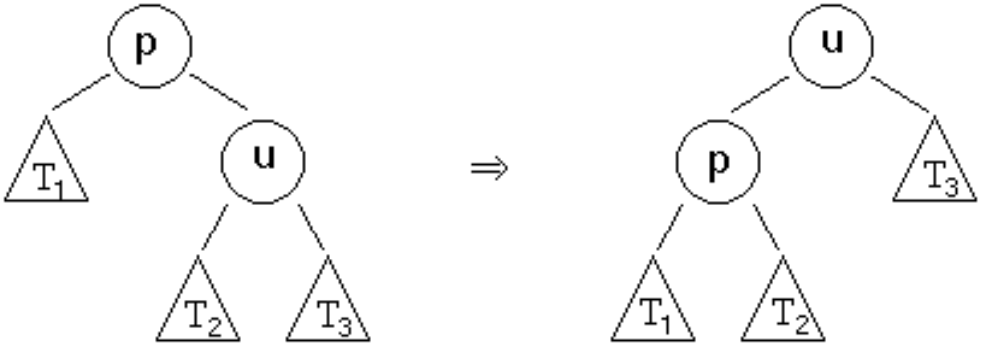
- Caso 1: Altura Esquerda de P > Altura Direita de P
- Caso 1.2 : Altura Esquerda de U < Altura Direita de U



Rotação para a esquerda e em seguida para a direita

Balanceamento

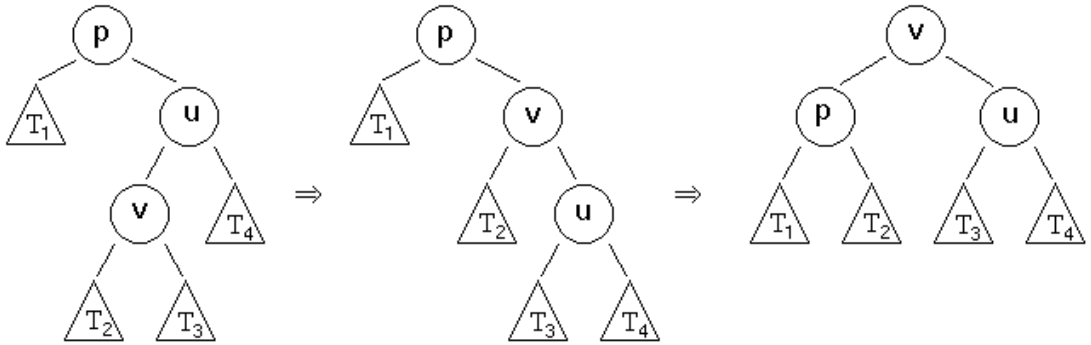
- Caso 2 : Altura Direita de P > Altura Esquerda de P
  - Caso 2.1: Altura Direita de U > Altura Esquerda de U



Rotação a esquerda

Balanceamento

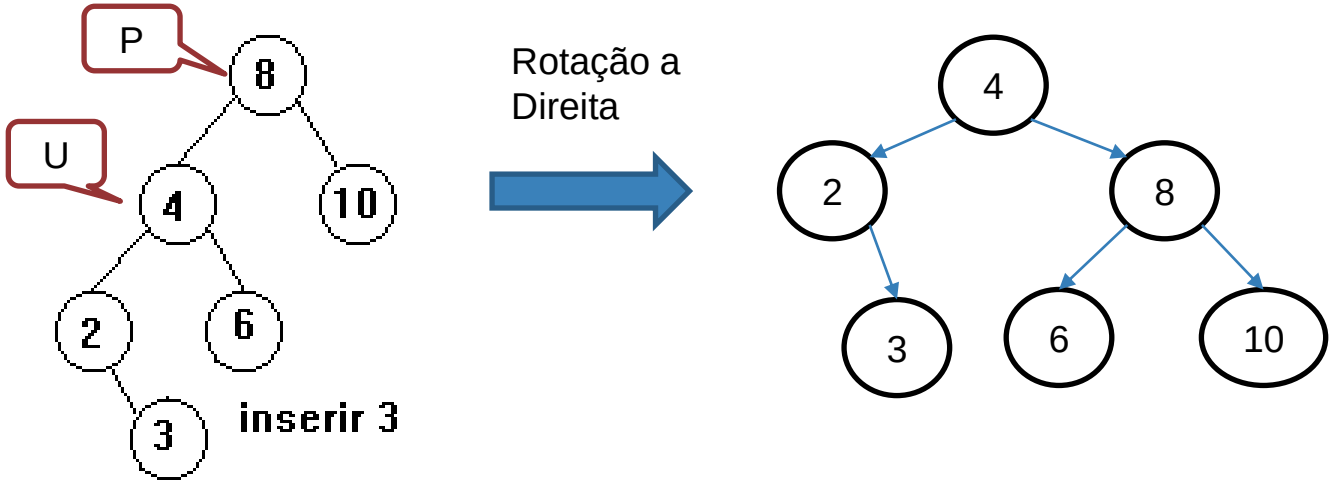
- Caso 2 : Altura Direita de P > Altura Esquerda de P
  - Caso 2.2: Altura Direita de U < Altura Esquerda de U



Rotação para a direita e em seguida para a esquerda

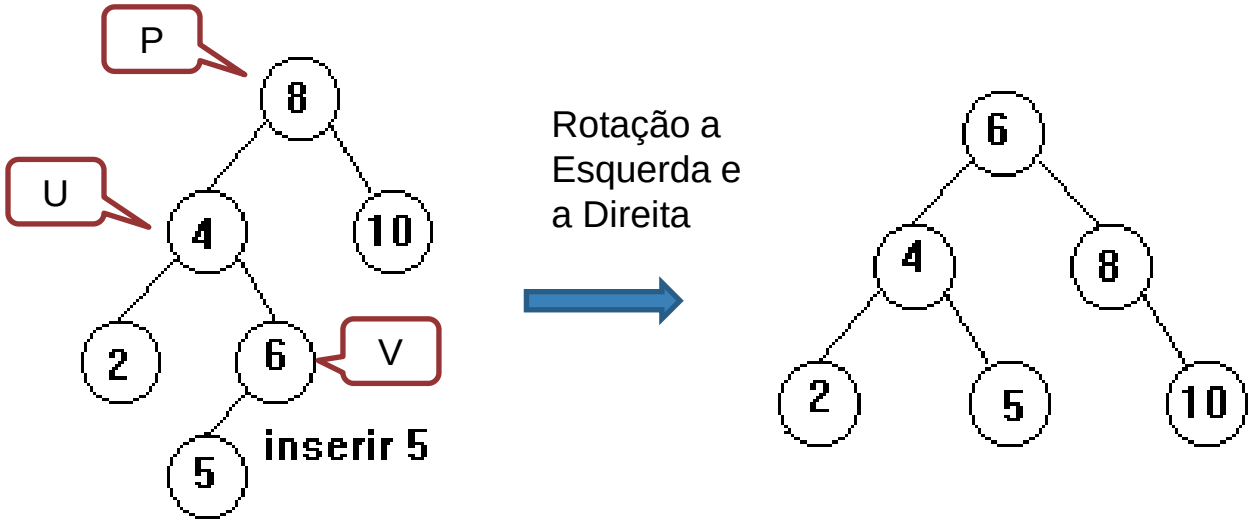
Exemplo

- Rotação simples a direita



Exemplo

- **Rotação dupla a direita**



AVL algoritmo

```
#include <stdio.h>
#include <stdlib.h>

#define true 0
#define false 1

typedef struct nodo
{   int          info;
    int          bal;
    struct nodo *esq;
    struct nodo *dir;
}AVL;

AVL *raiz;
int h;
```

auxiliar para propagar verificação de Fator de Balanceamento

```
void mostra_dados(AVL *);
AVL *cria_nodo(int);
AVL *insere_AVL(int, AVL *);
AVL *caso1(AVL *);
AVL *caso2(AVL *);
AVL *rotacao_direita(AVL *);
AVL *rotacao_esq_dir(AVL *);
AVL *rotacao_esquerda(AVL *);
AVL *rotacao_dir_esq(AVL *);

void main()
{  int numero=0;
   raiz=NULL;
   do {  printf("Entre com a informação : ");
         scanf("%i",&numero);
         if (numero != -1)
         {  raiz=insere_AVL(numero,raiz);
            mostra_dados(raiz);
         }
   } while (numero != -1);
}
```

```
AVL *insere_AVL(int x, AVL *pt)
{  if (pt == NULL)
   {  pt = cria_nodo(x);
      h=true;
   }
   else
   {  if (x < pt->info)
      {     pt->esq=insere_AVL(x,pt->esq);
            if (h == true)
            {     switch (pt->bal)
                  { case 1 :  pt->bal = 0;
                              h=false;
                              break;
                    case 0 : pt->bal = -1;
                              break;
                    case -1: pt=caso1(pt);
                              h=false;
                              break;
                  }
            }
```

Inserção dos Elementos

Recursão Esquerda

Verificar Balanceamento

Era mais alto a direita, equilibrou

Ficou com a esquerda maior

Interrompe Balanceamento

Constata caso1

```
else
{  if (x > pt->info)
   {  pt->dir = insere_AVL(x,pt->dir);
      if (h == true)
      {  switch (pt->bal)
         {  case -1:   pt->bal=0;
                       h=false;
                       break;
            case 0 :   pt->bal=1;
                       break;
            case 1 :   pt=caso2(pt);
                       h=false;
                       break;
         }
      }
    }
     else
        printf("informação já  existente");
   }
}
return pt;
}
```

Recursão Direita

Verificar Balanceamento

Era mais alto a esquerda, equilibrou

Ficou com a direita maior

Constata caso2

```
AVL *caso1(AVL *pt)
{  AVL *ptu;
    ptu=pt->esq;
    if (ptu->bal == -1)
       pt=rotacao_direita(pt);
   else
       pt=rotacao_esq_dir(pt);
   pt->bal=0;
   return pt;
}
```

Caso1.1- sinais iguais e negativos

Caso1.2- sinais diferentes

```
AVL *caso2(AVL *pt)
{  AVL *ptu;
   ptu=pt->dir;
   if (ptu->bal == 1)
      pt=rotacao_esquerda(pt);
   else
      pt=rotacao_dir_esq(pt);
   pt->bal=0;
   return pt;
}
```

Caso2.1- sinais iguais e positivos

Caso2.2- sinais diferentes

```
AVL *rotacao_direita(AVL *pt)
{  AVL *ptu;
   ptu=pt->esq;
   pt->esq=ptu->dir;
   ptu->dir=pt;
   pt->bal=0;
   return ptu;
}
```

```
AVL *rotacao_esquerda(AVL *pt)
{  AVL *ptu;
   ptu=pt->dir;
   pt->dir=ptu->esq;
   ptu->esq=pt;
   pt->bal=0;
   return ptu;
}
```

Árvores AVL

```
AVL *rotacao_esq_dir(AVL *pt)
{  AVL *ptu, *ptv;
   ptu=pt->esq;
   ptv=ptu->dir;
   ptu->dir=ptv->esq;
   ptv->esq=ptu;
   pt->esq=ptv->dir;
   ptv->dir=pt;
   if (ptv->bal == -1)
      pt->bal=1;
   else
      pt->bal=0;
   if (ptv->bal == 1)
      ptu->bal=-1;
   else
      ptu->bal=0;
   return ptv;
}
```

```
AVL *rotacao_dir_esq(AVL *pt)
{  AVL *ptu, *ptv;
   ptu=pt->dir;
   ptv=ptu->esq;
   ptu->esq=ptv->dir;
   ptv->dir=ptu;
   pt->dir=ptv->esq;
   ptv->esq=pt;
   if (ptv->bal == 1)
      pt->bal=-1;
   else
      pt->bal=0;
   if (ptu->bal == -1)
      ptu->bal=1;
   else
      ptu->bal=0;
   return ptv;
}
```

```
AVL *cria_nodo(int n)
{  AVL *novo;
   novo=(AVL *) malloc(sizeof(struct nodo));
   if (novo == NULL)
   {  printf("Memória insuficiente");
      exit(1);
   }
   novo->info=n;
   novo->bal=0;
   novo->esq=NULL;
   novo->dir=NULL;
return(novo);
}
```