Módulo 9

Pesquisa em Memória Primária

Algoritmos e Estruturas de Dados II

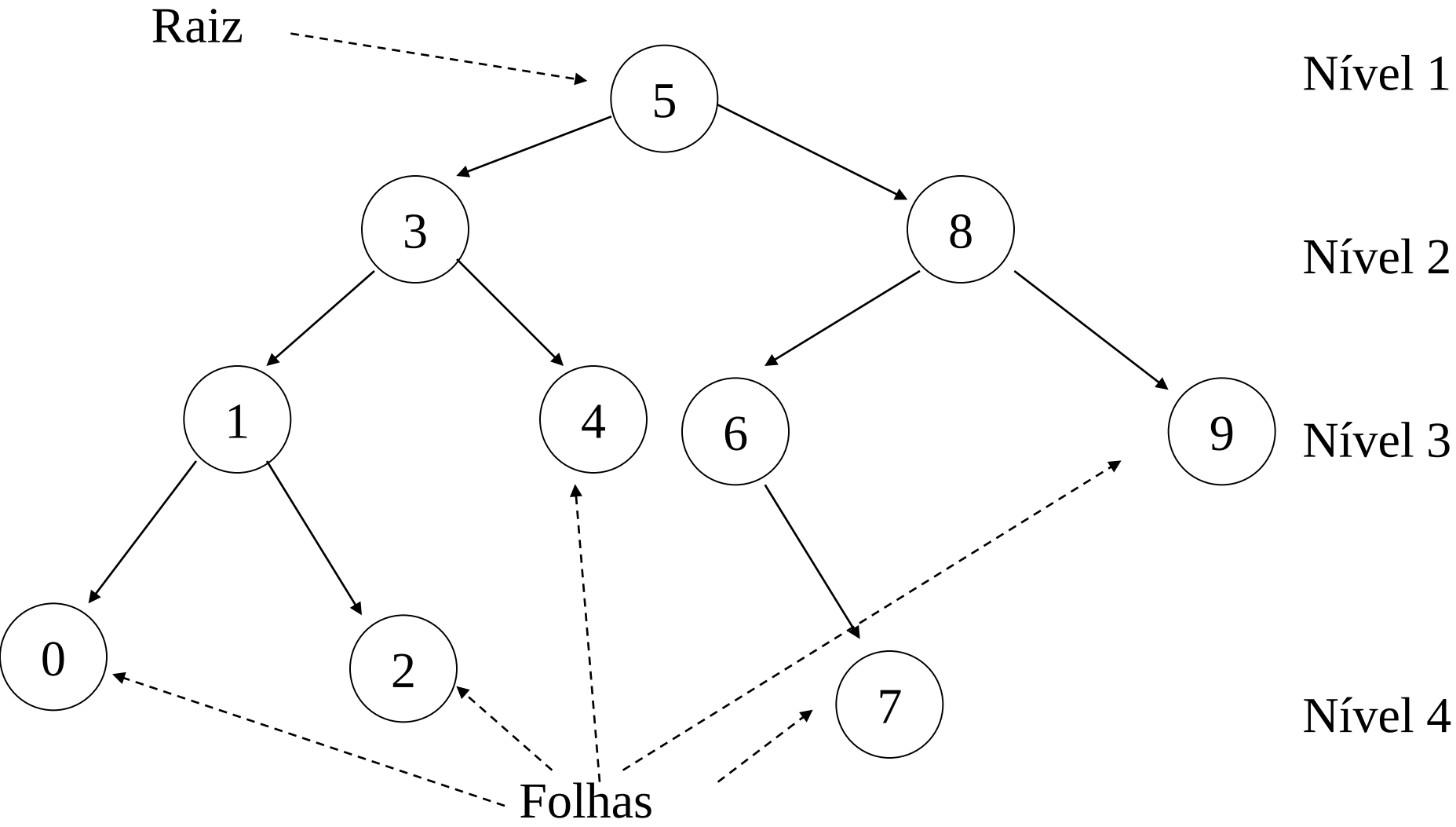
C++

(Rone Ilídio)

Árvore Binária

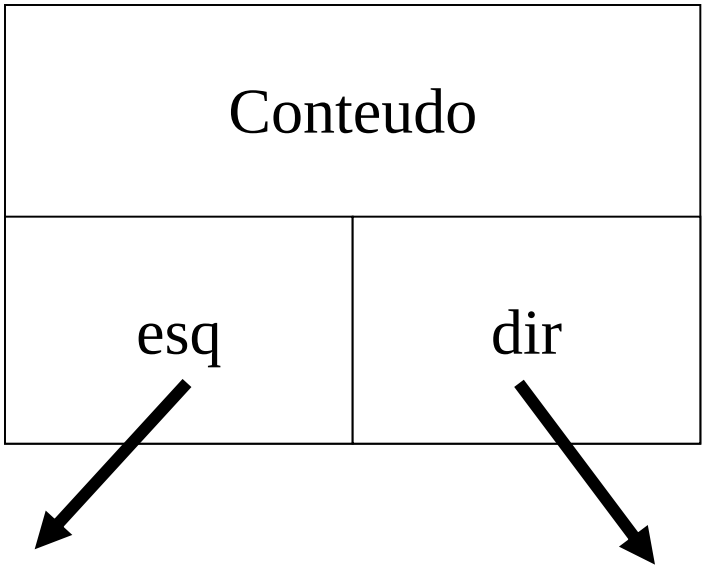
- Estrutura baseada em ponteiros
- Os elementos são chamados nós
- Cada nó é ligado a 0, 1 ou 2 elementos
- O número de elementos diz o nó da rede
- Não possui ciclos
- Árvore básica:
    - Nós menores a esquerda
    - Nós maiores a direita.

Árvore Binária



Árvore Binária

```
struct cel {
    int conteudo;
    struct cel *esq;
    struct cel *dir;
};
typedef struct cel no;
typedef no *arvore;
```



Árvore Binária

- Criação da árvore

```
int main(){
    arvore r;
    r=NULL;
   ...
}
```

Árvore Binária

- Cada célula é denominada nó
- O primeiro nó é denominado raiz
- Nós estão no mesmo nível se estão à mesma distância da raiz
- Pai: nó imediatamente superior a outro
- Filhos: nos apontados por esq ou dir
- Subárvore: subconjunto de uma árvore que mantém as mesmas características necessárias para uma árvore

Árvore Binária

- Caminho: sequência entre nós interligados
- Descendente: um nó A é decendente de B se existe um caminho entre eles e B está em um nível inferior a A
- Endereço da uma árvore: endereço da raiz

Exemplo de main

```
int main(){
    arvore r;
    r=NULL;
    inserir(r,5);  inserir(r,4);  inserir(r,3);  inserir(r,2);  inserir(r,1);
    cout << "\n\nPre-Ordem\n"; red(r);
    cout << "\n\nPos-Ordem\n"; edr(r);
    cout << "\n\nOrdem crescente\n"; erd(r);
    cout << "\n\nOrdem decrescente\n"; dre(r);
    cout << "\nAltura:" << altura(r);
    getch();
}
```

Inserir na Árvore

```
void inserir(arvore &r, int c){
    if(r==NULL){
        r = new no;
        r->conteudo = c;
        r->esq = NULL;
        r->dir = NULL;
    }
    else{
         if(c < r->conteudo)
             inserir(r->esq,c);
         else
             inserir(r->dir,c);
    }
}
```

Varredura

- Forma pela qual a árvore será percorrida
- Principais formas:
  - Pré-ordem: primeiro a raiz depois os filhos
  - Pós-ordem: primeiro os filhos depois a raiz
- Normalmente, o primeiro filho a ser pesquisado é o da esquerda

Varredura em Pré-ordem

```
//Raiz esquerda direita ou pré-ordem
void red(arvore r){
    if(r!=NULL){
        cout << "\n" <<r->conteudo;
        red(r->esq);
        red(r->dir);
    }
}
```

5-3-1-0-2-4-8-6-7-9

Varredura Pós-Ordem

```
//Esquerda direita raiz ou pos-ordem
void edr(arvore r){
    if(r!=NULL){
        edr(r->esq);
        edr(r->dir);
        cout << "\n" <<r->conteudo;
    }
}
```

0-2-1-4-3-7-6-9-8-5

Varredura em Ordem Crescente

```
//Varedura esqueda raiz direita -- em ordem
    crescente
void erd(arvore r){
     if(r!=NULL){
          erd(r->esq);
          cout << "\n" <<r->conteudo;
          erd(r->dir);
     }
}
```

0-1-2-3-4-5-6-7-8-9

Varredura em Ordem Decrescente

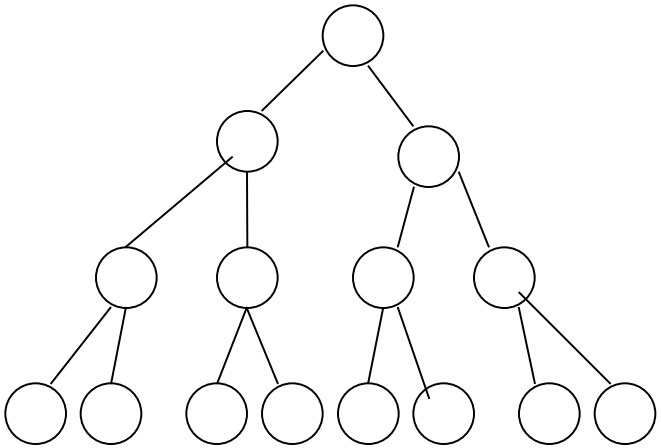
```
//Varedura esqueda raiz direita -- em ordem
    decrescente
void dre(arvore r){
     if(r!=NULL){
          dre(r->dir);
          cout << "\n" <<r->conteudo;
          dre(r->esq);
     }
}
```

9-8-7-6-5-4-3-2-1-0

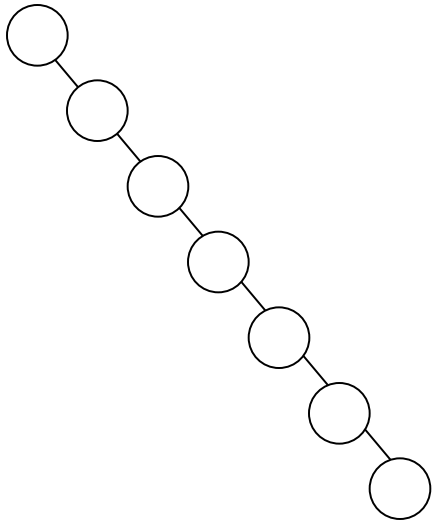
Altura

- Altura *h* de uma árvore é a maior distância entre a raiz e um nó folha
- $\lfloor \log_2 n \rfloor <= h < n$
- Se $h = \log_2 n$ → a árvore está cheia
- Se h = (n-1) →a árvore é um “tronco”

Altura



H = 3 n = 15 $\lfloor \log_2 15 \rfloor = 3$



H= n-1 = 6

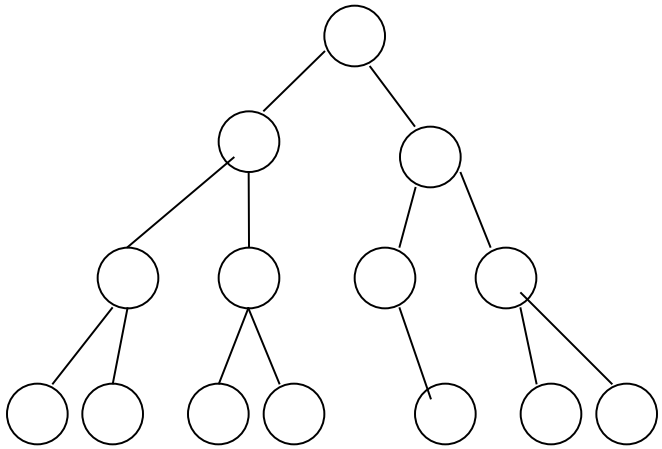
Altura

```
int altura(arvore r){
    if(r==NULL)
        return -1;
    else{
         int he = altura(r->esq);
         int hd = altura(r->dir);
         if (he > hd) return he+1;
         else return hd+1;
    }
}
```

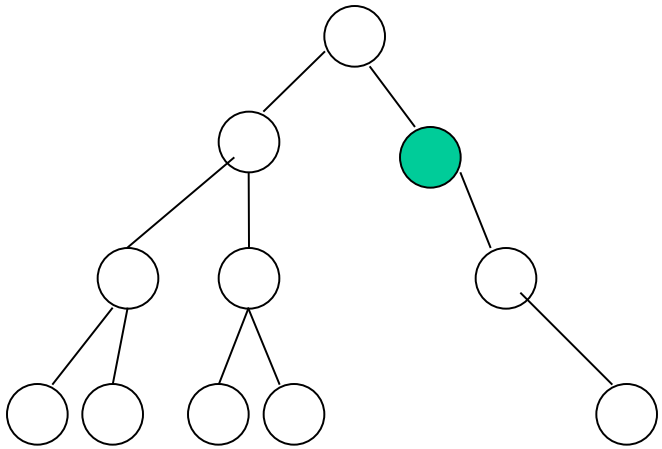
Árvore AVL ou Árvore Balanceada

Árvore AVL

É aquela que para todos os nós as sub-árvores direita e esquerda possuem alturas iguais, ou com difereça de apenas 1



Balanceada



Desbalanceada

Árvore AVL

- Fator de Balanceamento
  - Diferença entre as alturas das subarvore direita e esquerda de um determinado nó
  - Função

```
int fb(arvore r){
    int hd = altura(r->dir);
    int he = altura(r->esq)
    return hd - he;   //Utilizaremos esse padrão
}
```

Árvore AVL

- Análise de fb
  - Fb = 0 → subarvores com alturas iguais
  - Fb < = → esquerda maior
  - Fb > 0 → direita maior

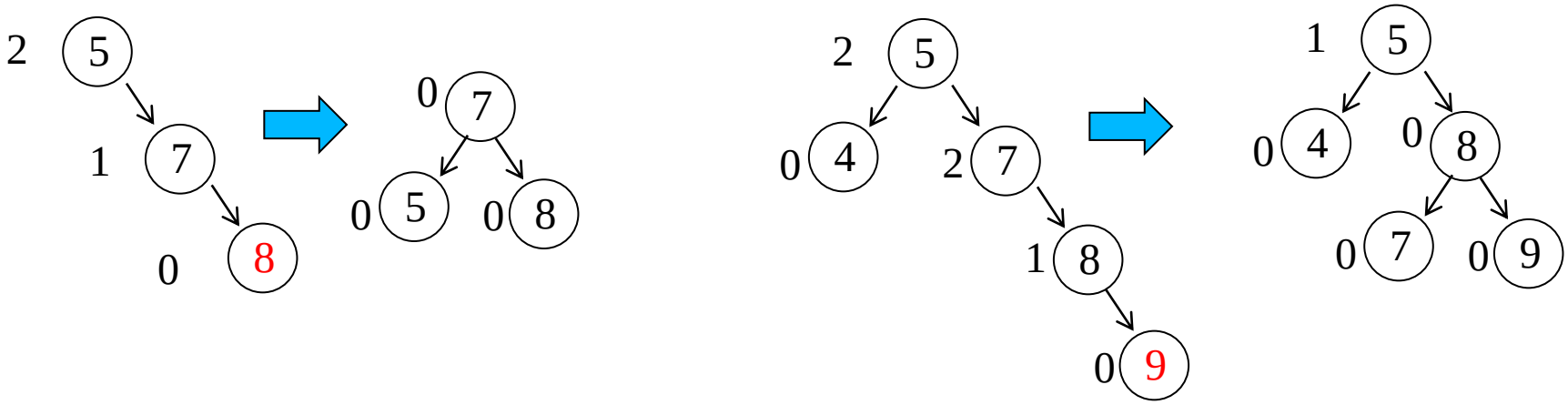
  - Obs: em uma árvore AVL -1<=fb<=1 para todos os nós
  - Se algum nó |fb| > 1 a árvore precisa ser balanceada

Árvore AVL

- Operações
  - Inserção: log n
  - Pesquisa: log n
  - Remoção: é necessário fazer uma pesquisa → log n
- Rotação: operação para balancear uma árvore
  - Rotação simples: esquerda ou direita
  - Rotação dupla: duas rotações em sentidos opostos

Árvore AVL

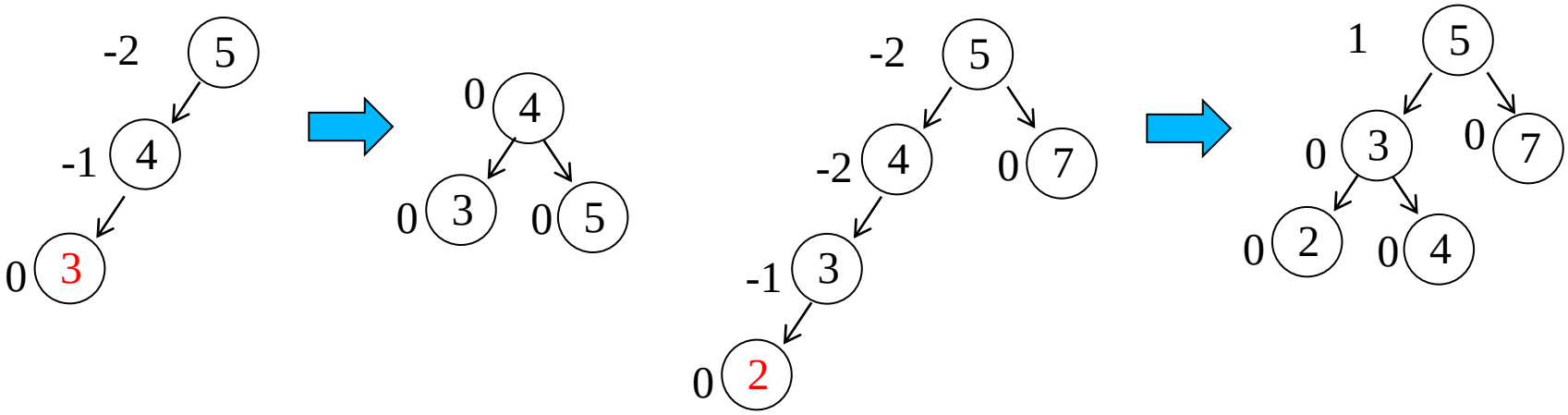
- Rotação simples para a esquerda



Se fb = 2 e fb do filho direito é 1 → Rotação simples para a esquerda

Árvore AVL

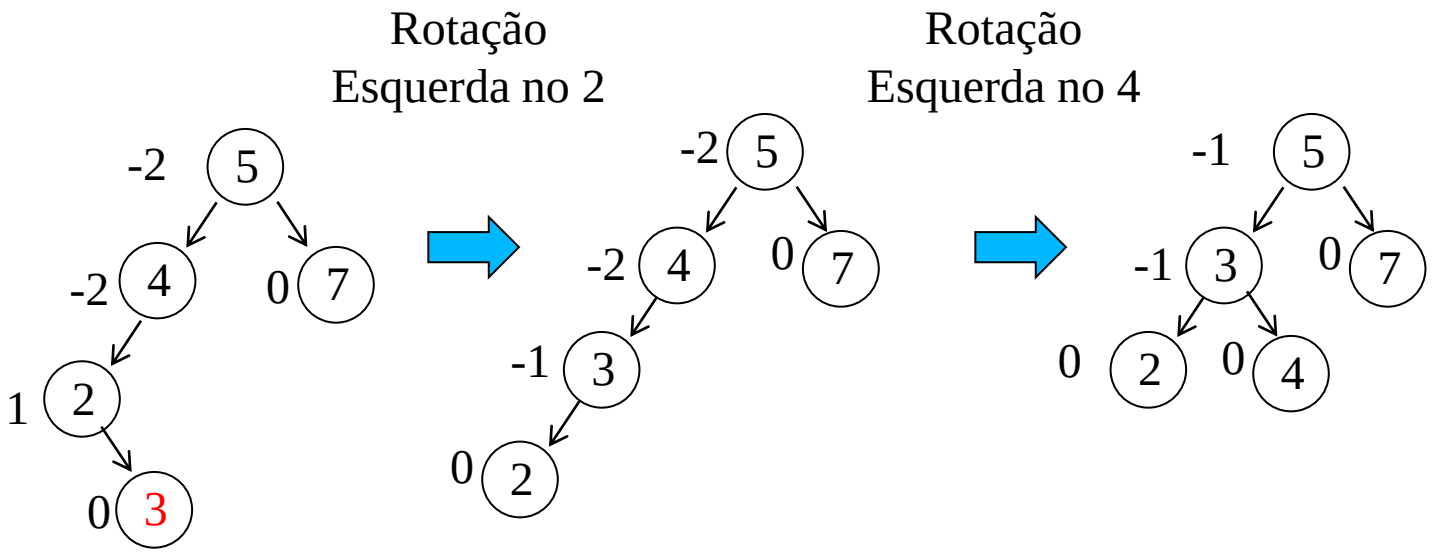
- Rotação simples para a direita



Se fb = -2 e fb do filho esquerdo é -1 → Rotação simples para a direita

Árvore AVL

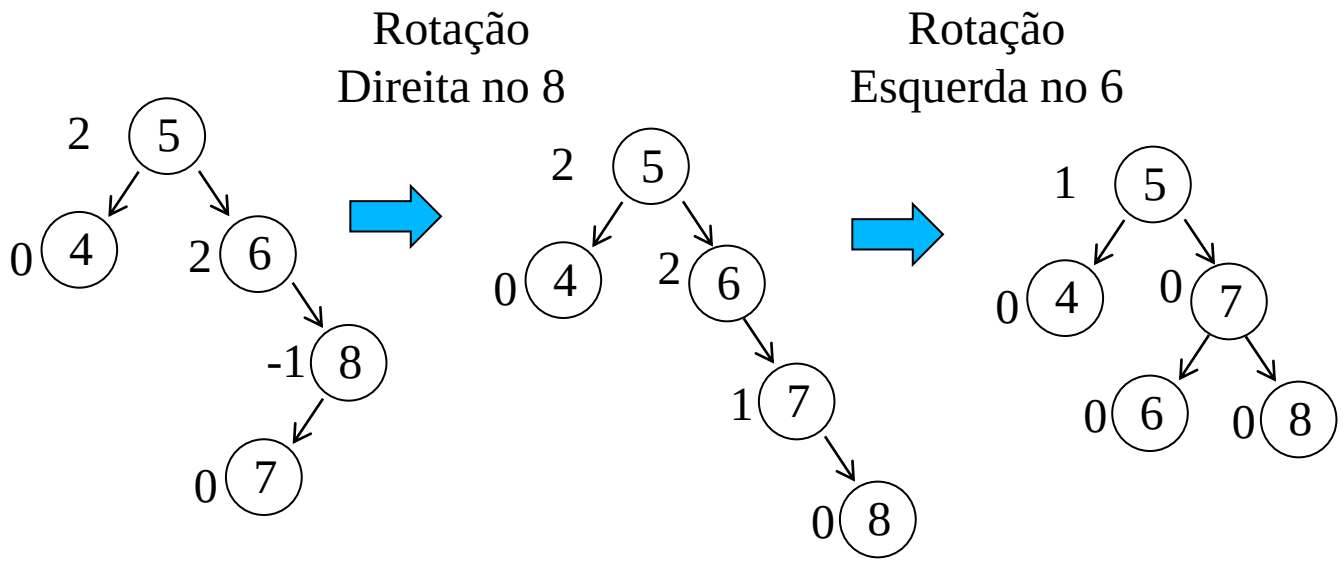
- Rotação dupla esquerda direita



Se fb = -2 e fb do filho esquerdo é 1 → Rotação simples para a **esquerda** no filho e rotação simples para a **direita** no nó desbalanceado

Árvore AVL

- Rotação dupla direita esquerda



Se fb = 2 e fb do filho direito é -1 → Rotação simples para a **direita** no filho e rotação simples para a **esquerda** no nó desbalanceado

Árvore AVL

- Rotação direita

```
void rotacaodireita(arvore &r){
    arvore aux = r;
    r = aux->esq;
    aux->esq = r->dir;
    r->dir = aux;
}
```

Árvore AVL

- Rotação esquerda

```
void rotacaoesquerda(arvore &r){
    arvore aux = r;
    r = aux->dir;
    aux->dir = r->esq;
    r->esq = aux;
}
```

Árvore AVL

- Rotação dupla direita esquerda

```
void rotacaoesquerdadireita(arvore &r){
    rotacaoesquerda(r->esq);
    rotacaodireita(r);
}
```

- Rotação dupla esquerda direita

```
void rotacaodireitaesquerda(arvore &r){
    rotacaodireita(r->dir);
    rotacaoesquerda(r);
}
```

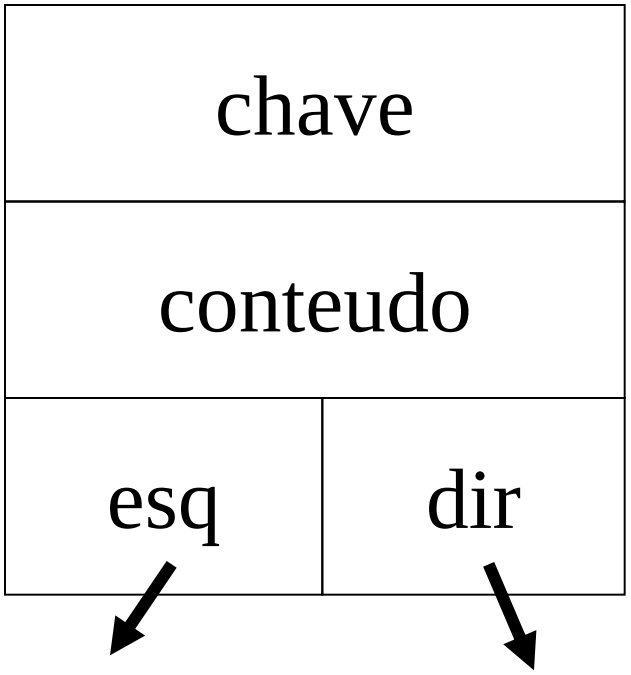
```
void balancear(arvore &r){
    if(r == NULL) return;
    if(fb(r) == 2){                          //Direita maior que a esquerda
       if(fb(r->dir)==-1)                    //O filho direito tem a esquerda maior que a direita
            rotacaodireitaesquerda(r);  //Sinal trocado == duas rotações
       if(fb(r->dir)==1)                   //O filho direito tambem tem a direita maior
            rotacaoesquerda(r);           //Mesmo sinal somente uma rotação
    }
    if(fb(r) == -2){                         //Esquerda maior que direita
       if(fb(r->esq)==1)                     //O filho de esquerdo tem a direita maior
            rotacaoesquerdadireita(r);   //Sinal trocado =  duas rotações
        if(fb(r->esq)==-1)                   //O filho esquerdo tem a esquerda maior
            rotacaodireita(r);               //Mesmo sinal somente uma rotação
    }
    //Balancear os filhos de r
    balancear(r->dir);
    balancear(r->esq);
}
```

Árvore Binária de Busca

Árvore Binária de Busca

- Tipo especial de árvore binária
- Estrutura

```
struct cel {
    int chave;
    char nome[30];
    cel *esq;
    cel *dir;
};
typedef cel no;
typedef no* arvore;
```



- Ordenação pelas chaves

Árvore Binária de Busca

```
void inserir(arvore &r, int ch, char n[]){
    if(r==NULL){
        r = new no;
        r->chave = ch;
        strcpy(r->nome,n);
        r->esq = NULL;
        r->dir = NULL;
    }
    else{
         if(ch < r->chave)
             inserir(r->esq,ch,n);
         else
             inserir(r->dir,ch,n);
    }
}
```

Buscar

```
no *busca(arvore r, int k){
    if(r==NULL || r->chave == k)
        return r;
    if(r->chave > k)
        return busca(r->esq, k);
    else
        return busca(r->dir, k);
}
```

Exemplo de main

```
int main(){
   arvore a = NULL;
   int chave;
   char nome[30];
   while(1){
      cout<<"Digite o nome (0 sair):";
      cin>>nome;
      if(strcmp(nome,"0")==0) break;
      cout<<"Digite a chave:";
      cin>>chave;
      inserir(a,chave,nome);
   }
...
```

```
...
while(1){
      cout << "\nDigite uma chave(0 sair):";
      cin >> chave;
      if(chave==0) break;
      arvore r = busca(a,chave);
      if (r==NULL) cout << "\nNao
     encontrado!";
      else {
         cout << "\nChave:" << r->chave;
         cout << "\nNome:" << r->nome;
      }
   }
}
```