**ÁRVORE AVL**

**ÁRVORE BALANCEADA**

- Árvore AVL é um tipo de árvore binária balanceada. Uma árvore balanceada é aquela cuja diferença de altura entre a sub-árvore da direta e a sub-árvore da esquerda (em qualquer nó) é de, no máximo, uma unidade.

- Para manter o balanceamento, as operações de inserção e remoção levam em consideração o fator de balanceamento: altura da direita menos altura da esquerda – hd-he.

- O intuito é manter o baixo custo das operações, que numa árvore balanceada é de “O (*logN*)”.

**ÁVORE AVL**

- Árvore que admite rebalanceamento local, isto é, a parte afetada pelas operações de inserção ou remoção pode sofrer uma ação de rebalanceamento. O rebalanceamento é feio através de rotações simples ou duplas.

**ROTAÇÃO**

- Rotação é a operação básica para balanceamento da árvore.

- Dois tipos: simples e dupla.

- Na rotação simples, o nó desbalanceado e seu filho estão no mesmo sentido da inclinação.

- Na rotação dupla, o nó desbalanceado e seu filho estão no sentido inverso ao pai.

Rotação RR - Rotação simples à esquerda

- Nó inserido na sub-árvore da direita da sub-árvore da direita. É feita uma rotação à esquerda.

- Nó inserido: C. Nó A fica desbalanceado.

- O nó intermediário B torna-se a raiz da árvore resultante.

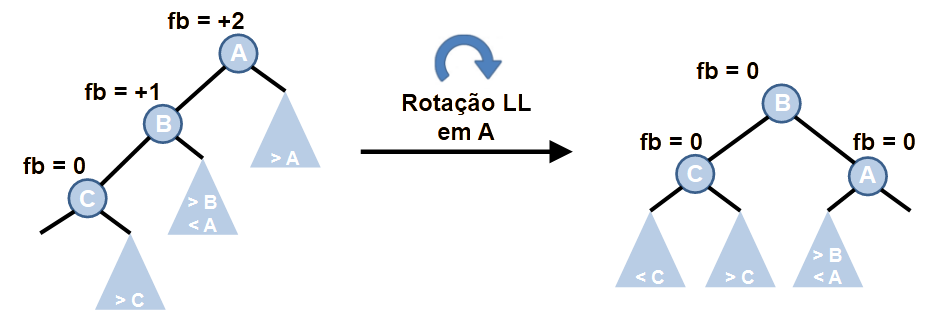


Rotação LL - Rotação simples à direita

- Nó inserido na sub-árvore da esquerda da sub-árvore da esquerda. É feita uma rotação à direita.

- Nó inserido: C. Nó A fica desbalanceado.

- O nó intermediário B torna-se a raiz da árvore resultante.



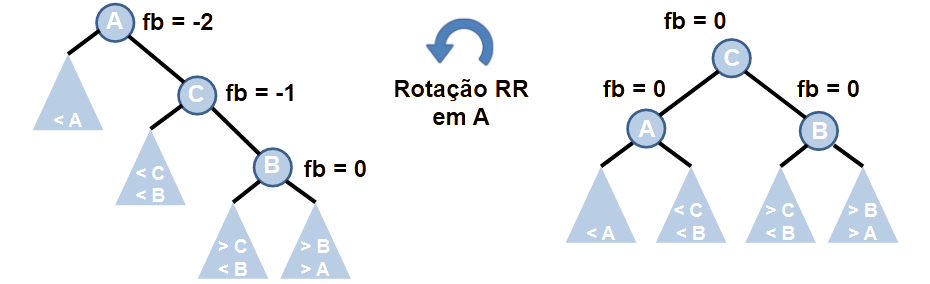
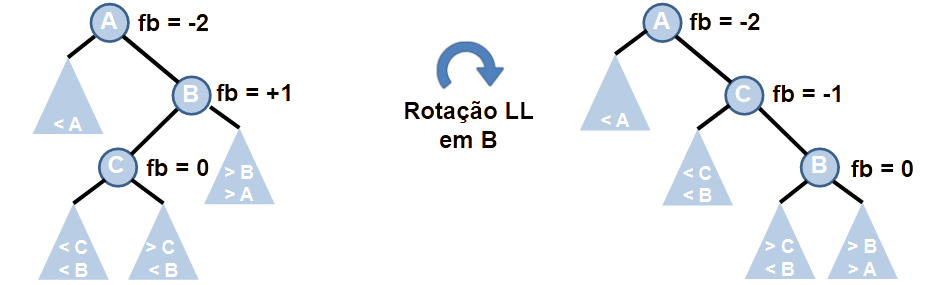
Rotação RL

- Nó inserido na sub-árvore esquerda da sub-árvore da direita.

- Nó inserido: C. Nó A fica desbalanceado.

- Passos: rotação LL em B e rotação RR em A.

- Nó inserido C deve ser torna-se a raiz da árvore resultante.



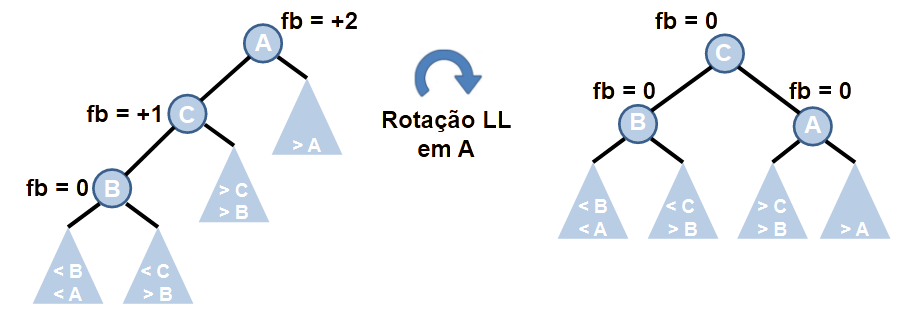
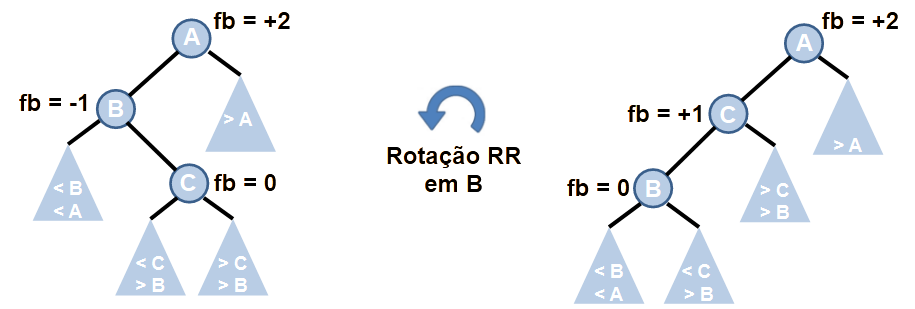
Rotação LR

- Nó inserido na sub-árvore direita da sub-árvore da esquerda.

- Nó inserido: C. Nó A fica desbalanceado.

- Passos: rotação RR em B e rotação LL em A.

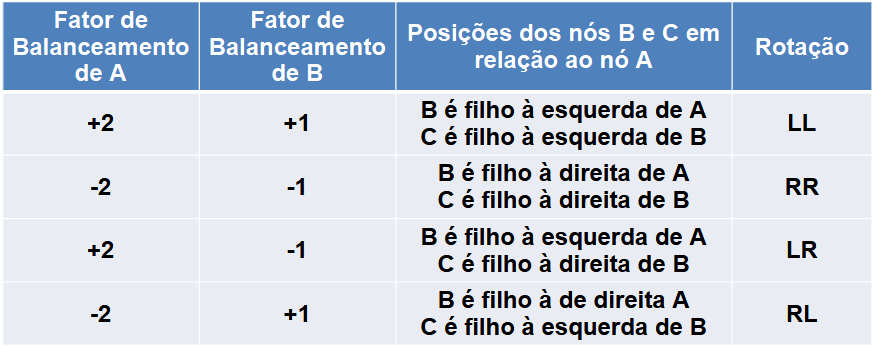
- Nó inserido C deve ser torna-se a raiz da árvore resultante.



- Quando usar cada rotação:

Considerando ESQUERDA - DIREITA

* Sinais iguais: rotação simples:
  + Sinal positivo: rotação à direita;
  + Sinal negativo: rotação à esquerda.
* Sinais diferentes: rotação dupla:
  + Sinal positivo: rotação dupla à direta;
  + Sinal negativo: rotação dupla à esquerda.



**OPERAÇÕES**

- Com exceção das operações de inserção e remoção, as operações sobre as árvores AVL são idênticas às árvores binárias de busca.

IMPLEMENTAÇÃO

- Definimos uma *struct* para o nó com os seguintes atributos:

* dado;
* altura;
* ponteiro para nó da esquerda;
* ponteiro para nó da direita.

- Armazenar a altura do nó facilita os cálculos do fator de balanceamento.

struct No{

int dado;

int altura;

struct No \*esq;

struct No \*dir;

};

// ponteiro para a raiz da árvore

typedef struct No\* Arvore;

- No programa principal:

Arvore\* raiz = cria\_arvore();

**ROTAÇÃO**

- As rotações são aplicadas no ancestral mais próximo do nó inserido cujo fator de balanceamento passa a ser +2 ou -2.

- As rotações atualizam as novas alturas das sub-árvores.

Rotação LL

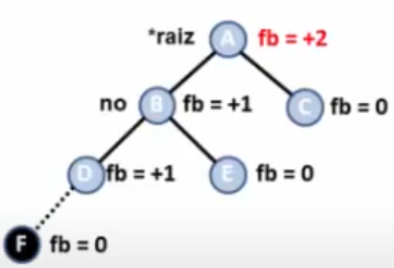
- Começando pela raiz (A) da sub-árvore em questão.

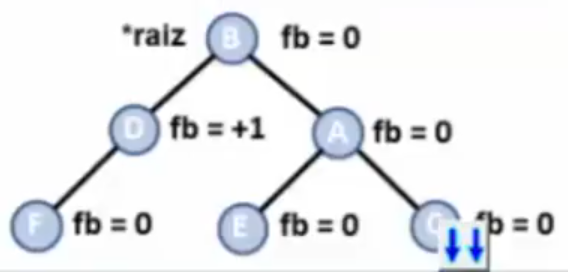
- 1) Troca B por E: direita do filho da esquerda se torna a sua esquerda.

- 2) Troca E por A: direita do filho da esquerda recebe a raiz.

- 3) Atualiza altura da raiz e do seu filho da esquerda (maior altura entre os filhos da direita e esquerda, +1).

- 4) B torna-se raiz: filho da esquerda se torna raiz.





void RotacaoLL(Arvore \*raiz){

struct No \*no;

no = (\*raiz)->esq;

(\*raiz)->esq = no->dir;

no->dir = \*raiz;

(\*raiz)->altura = maior(altura\_No((\*raiz)->esq),altura\_No((\*raiz)->dir)) + 1;

no->altura = maior(altura\_No(no->esq),(\*raiz)->altura) + 1;

\*raiz = no;

}

Rotação RR

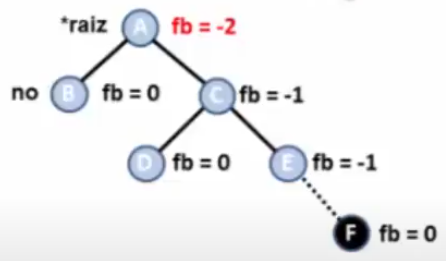
- Começando pela raiz (A) da sub-árvore em questão.

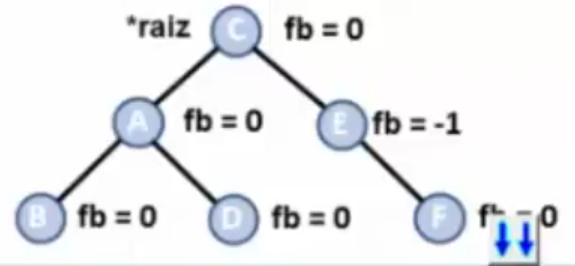
- 1) Troca C por D: esquerda do filho da direita se torna a sua direita.

- 2) Troca D por A: esquerda do filho da direita recebe a raiz.

- 3) Atualiza altura da raiz e do seu filho da direita (maior altura entre os filhos da direita e esquerda, +1).

- 4) C torna-se raiz: filho da direita se torna raiz.





void RotacaoRR(Arvore \*raiz){

struct No \*no;

no = (\*raiz)->dir;

(\*raiz)->dir = no->esq;

no->esq = (\*raiz);

(\*raiz)->altura = maior(altura\_No((\*raiz)->esq),altura\_No((\*raiz)->dir)) + 1;

no->altura = maior(altura\_No(no->dir),(\*raiz)->altura) + 1;

(\*raiz) = no;

}

Rotação LR

- Aplica-se:

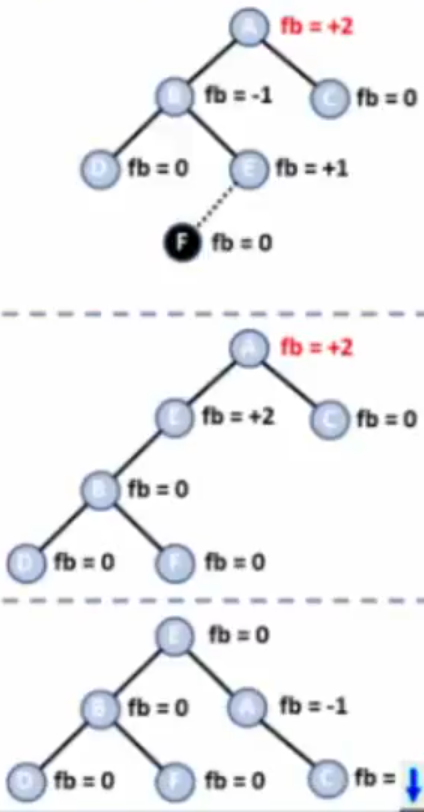
* Rotação à esquerda no filho da esquerda (B);
* Rotação à direta na raiz (A).

void RotacaoLR(Arvore \*raiz){

RotacaoRR(&(\*raiz)->esq);

RotacaoLL(raiz);

}



Rotação RL

- Aplica-se:

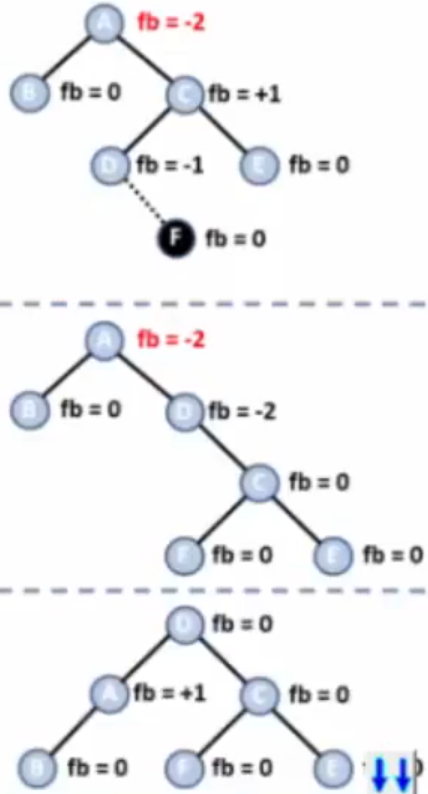
* Rotação à direita no filho da direita (C);
* Rotação à esquerda na raiz (A).

void RotacaoRL(Arvore \*raiz){

RotacaoLL(&(\*raiz)->dir);

RotacaoRR(raiz);

}



INSERÇÃO

- Para inserir um valor v:

* Verifica se raiz é *NULL*, isto é, inserção na raiz; se não;
* Se *v* é menor do que a raiz, vai para a sub-árvore da esquerda;
* Se *v* é maior do que a raiz, vai para a sub-árvore da direita.
* Aplique o método recursivamente.

- Ao voltar na recursão, recalcule as alturas de cada sub-árvore.

- Aplique a rotação necessária se o fator de balanceamento passa a ser +2 ou -2.

- A função abaixo primeiro verifica se já chegou a um nó folha.

- Caminha pela árvore indo para esquerda ou direita, dependendo se o valor é > ou < que o valor do nó visitado.

- Se a inserção ocorre, a variável res recebe 1, e a função parte para o cálculo do fator de balanceamento. Ao encontrar um fator de balanceamento maior que 2 (absoluto), aplica a devida rotação.

- Por fim, atualiza a altura no novo nó.

int insere\_Arvore(Arvore \*raiz, int valor){

int res; // variável para indicar o sucesso da inserção

// árvore vazia ou nó folha

if(\*raiz == NULL){

struct No \*novo;

novo = (struct No\*)malloc(sizeof(struct No));

if(novo == NULL)

return 0;

novo->dado = valor;

novo->altura = 0;

novo->esq = NULL;

novo->dir = NULL;

\*raiz = novo;

return 1;

}

// para esquerda

struct No \*atual = \*raiz;

if(valor < atual->dado){

if((res = insere\_Arvore(&(atual->esq), valor)) == 1){

if(fatorBalanceamento\_No(atual) >= 2){

if(valor < (\*raiz)->esq->dado ){ // esquerda

RotacaoLL(raiz);

}else{ // direita

RotacaoLR(raiz);

}

}

}

}else{

// para direita

if(valor > atual->dado){

if((res = insere\_Arvore(&(atual->dir), valor)) == 1){

if(fatorBalanceamento\_No(atual) >= 2){

if((\*raiz)->dir->dado < valor){ // direita

RotacaoRR(raiz);

}else{ // esquerda

RotacaoRL(raiz);

}

}

}

}else{

printf("Valor duplicado!!\n");

return 0;

}

}

// atualiza a altura do nó

atual->altura = maior(altura\_No(atual->esq),altura\_No(atual->dir)) + 1;

return res;

}

REMOÇÃO

- Três casos de remoção: nó folha, nó com 1 filho e nó com 2 filhos.

- Ao remover, é necessário também verificar o fator de balanceamento. Se a remoção foi em um nó à esquerda, o desbalanceamento pode ocorrer na sub-árvore, e vice-versa. Nesses casos, deve-se aplicar a devida rotação.

- Duas funções são utilizadas:

* *remove\_Arvore*: busca o nó a ser removido e trata os diferentes casos;
* *procuraMenor*: função auxiliar para o caso de remoção de nó com 2 filhos.

- No caso em que o nó removido tem 2 filhos, a função *procuraMenor* procura o elemento mais à esquerda (menor valor) da sub-árvore da direita, para substituir o nó removido.

Funções

- Se o valor buscado for menor que o da raiz, procede a busca na sub-árvore da esquerda, passando ela para a função: *remove\_Arvore(&(\*raiz)->esq,valor))*.

- Se essa função retorna 1, quer dizer que a remoção ali aconteceu, e passamos para testar o fator de balanceamento na sub-árvore da direita. Caso haja desbalanceamento, aplica a rotação adequada.

- Se o valor buscado for maior que o da raiz, procede a busca na sub-arvore da direita, passando ela para a função: *remove\_Arvore(&(\*raiz)->dir, valor)*. A mesma lógica se aplica, testando o balanceamento na sub-árvore da esquerda.

- Quando o valor é achado, testa se o nó tem 1 ou nenhum filho, ou se tem 2 filhos: *if(((\*raiz)->esq == NULL || (\*raiz)->dir == NULL))*.

* No caso de 1, o filho substitui o nó removido.
* No caso de 2, procuraMenor procura o substituto para o nó removido.

- Por fim, recalcula a altura da raiz e a atualiza.

struct No\* procuraMenor(struct No\* atual){

struct No \*no1 = atual;

struct No \*no2 = atual->esq;

while(no2 != NULL){

no1 = no2;

no2 = no2->esq;

}

return no1;

}

int remove\_Arvore(Arvore \*raiz, int valor){

if(\*raiz == NULL){

printf("Valor não existe!!\n");

return 0;

}

int res;

if(valor < (\*raiz)->dado){

if((res = remove\_Arvore(&(\*raiz)->esq,valor)) == 1){

if(fatorBalanceamento\_No(\*raiz) >= 2){

if(altura\_No((\*raiz)->dir->esq) <= altura\_No((\*raiz)->dir->dir))

RotacaoRR(raiz);

else

RotacaoRL(raiz);

}

}

}

if((\*raiz)->dado < valor){

if((res = remove\_Arvore(&(\*raiz)->dir, valor)) == 1){

if(fatorBalanceamento\_No(\*raiz) >= 2){

if(altura\_No((\*raiz)->esq->dir) <= altura\_No((\*raiz)->esq->esq) )

RotacaoLL(raiz);

else

RotacaoLR(raiz);

}

}

}

if((\*raiz)->dado == valor){

// nó tem 1 filho ou nenhum

if(((\*raiz)->esq == NULL || (\*raiz)->dir == NULL)){

struct No \*oldNode = (\*raiz);

if((\*raiz)->esq != NULL)

\*raiz = (\*raiz)->esq;

else

\*raiz = (\*raiz)->dir;

free(oldNode);

// nó tem 2 filhos

}else {

struct No\* temp = procuraMenor((\*raiz)->dir);

(\*raiz)->dado = temp->dado;

remove\_Arvore(&(\*raiz)->dir, (\*raiz)->dado);

if(fatorBalanceamento\_No(\*raiz) >= 2){

if(altura\_No((\*raiz)->esq->dir) <= altura\_No((\*raiz)->esq->esq))

RotacaoLL(raiz);

else

RotacaoLR(raiz);

}

}

if (\*raiz != NULL)

(\*raiz)->altura = maior(altura\_No((\*raiz)->esq),altura\_No((\*raiz)->dir)) + 1;

return 1;

}

(\*raiz)->altura = maior(altura\_No((\*raiz)->esq),altura\_No((\*raiz)->dir)) + 1;

return res;

}