Pesquisa em Memória Primária

Ciência da Computação
Laboratório de Ordenação e Pesquisa
Prof. M.Sc. Elias Gonçalves

Pesquisa em Memória Primária

Objetivos:

→ Encontrar item cuja chave seja igual a do elemento dado na pesquisa.

→ Recuperar item com uma determinada chave.

→ Recuperar dados armazenados em uma base de dados.

Questões envolvidas na escolha do método de pesquisa:

→ Qual a quantidade de dados no conjunto?

→ Qual é a frequência com que operações de inserção e retirada de dados são realizadas?

→ Os dados estão estruturados?

→ Os dados estão ordenados?

→ Há valores repetidos?

Pesquisa em Memória Primária

Métodos de pesquisa:

→ Pesquisa sequencial;

→ Pesquisa binária;

→ Árvore de pesquisa;

→ Pesquisa digital;

Árvore AVL

- **Definição:**
  - Tem as mesmas características de uma árvore binária, porém com balanceamento.
  - O nome AVL vem de **A**delson-**V**elskii e **L**andis, autores do algoritmo de rotação para balanceamento das árvores binárias proposto em 1962.

Árvore AVL

- **Definição:**
  - Uma árvore AVL é uma árvore binária de busca balanceada, ou seja:
    - Para qualquer nó da árvore, as alturas de seus filhos (subárvore esquerda e subárvore direita) diferem em módulo de até uma unidade.
    - Existe um limite máximo para a diferença de altura dos filhos;
    - Existe um fator de balanceamento para indicar se um nó da árvore tem filhos que pendem para a direita, para a esquerda ou se está equilibrado.

Árvore AVL

- **Operações:**
  - Buscar nó com um chave específica;
  - Inserir novo nó;
  - Remover nó;
- A árvore binária de busca e a árvore AVL fazem as mesmas operações, no entanto, na AVL ao inserir e ao remover elemento é preciso calcular o fator de balanceamento e se necessário, fazer uma ou mais rotação para manter o equilíbrio.

  É preciso preservar as propriedades de AVL!

Árvore AVL

- **Tempo de busca:**
  - Com a árvore AVL o tempo máximo de busca é de log2(n);
  - **Ex:** ao dobrar o tamanho do conjunto de dados, a busca precisará de apenas mais um passo para encontrar uma chave.
  - Encontrar uma chave em uma árvore de 1024 elementos custará 10 passos no pior caso.
  - Encontrar uma chave em uma árvore de 2048 elementos custará 11 passos no pior caso...

Árvore AVL

- **Como encontrar o balanceamento?**
  - Olhar para a altura dos descendentes de um nó:
    - Quando a altura dos filhos de um nó é a mesma, esse nó (pai) tem altura 0;
    - Quando o filho da direita tem um nível a mais, a altura do nó (pai) é -1;
    - Quando o filho da esquerda tem um nível a mais, a altura do nó (pai) é +1;

Há autores que consideram o inverso em relação às duas últimas regras.

Árvore AVL

- **Fator de balanceamento:**
  - É definido como a diferença entre a altura da subárvore direita e a altura da subárvore esquerda, sendo que essa diferença é menor ou igual a >= -1 e <= 1.

    FB(n) = altura(n->esq) – altura(n->dir) | FB(n) >= -1 e <= 1

  - Quando a diferença entre alturas das subárvores for maior que 1 a árvore não é AVL.

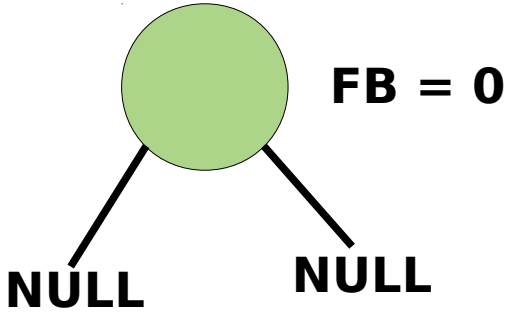
Árvore AVL

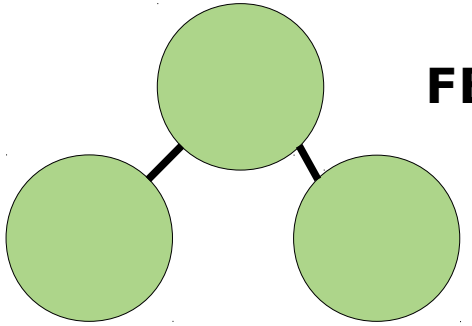
- **Fator de balanceamento:**

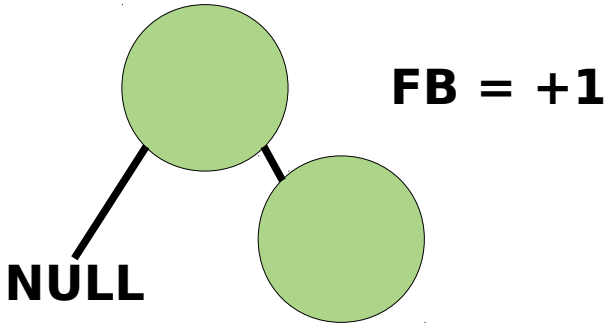
```
// Retorna a altura de um no da arvore
int obter_altura(NO *raiz) {
    if (raiz == NULL)
        return 0;
    return raiz->altura;
}

// Retorna a diferenca entre as alturas das subarvores do no
int obter_fb(NO *raiz){
    if (raiz == NULL)
        return 0;
    return obter_altura(raiz->esq) - obter_altura(raiz->dir);
}

// Retorna a maior altura de 2 subarvores
int obter_maximo(int x, int y){
    return( x>y ? x : y );
}
```

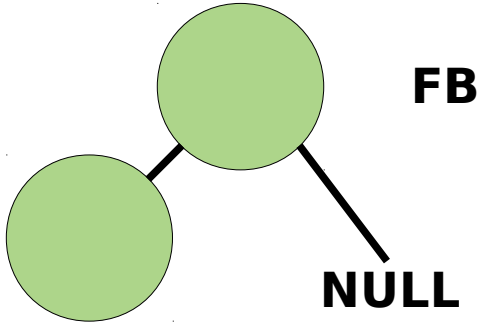
Árvore AVL

- **Exemplo Fator de Balanceamento**



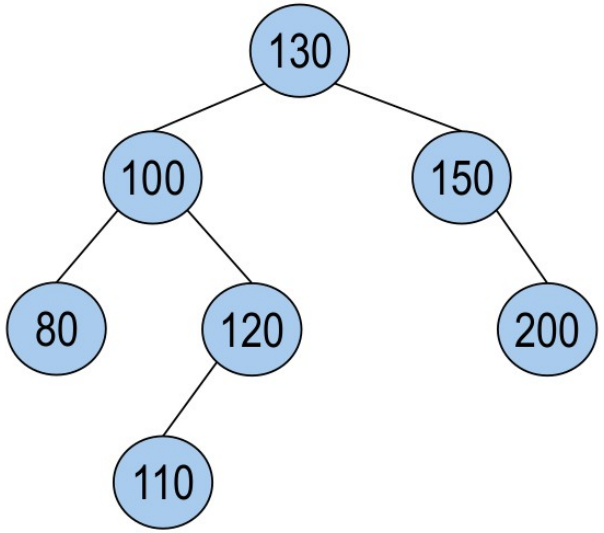


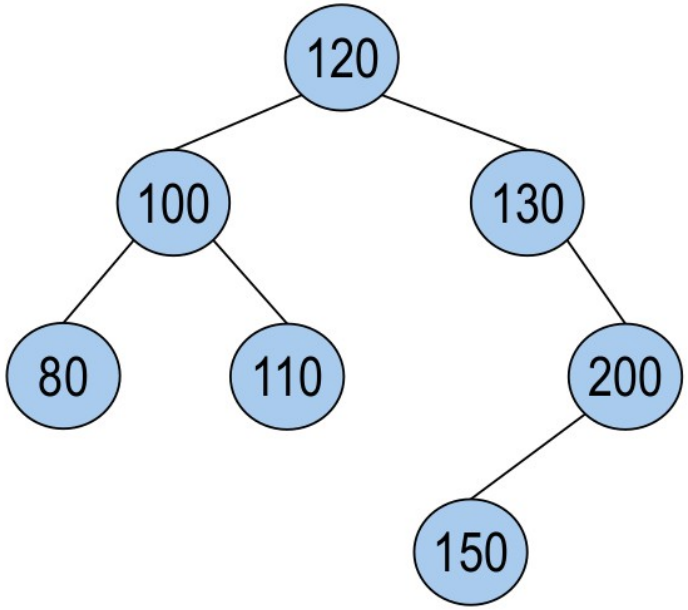




Árvore AVL

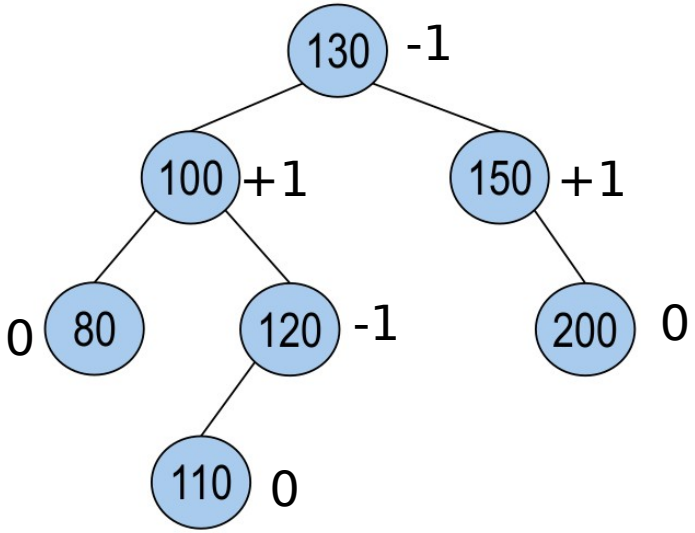
- **É AVL?**
  - Verifique se cada árvore é AVL. Determine o fator de balanceamento para cada nó.



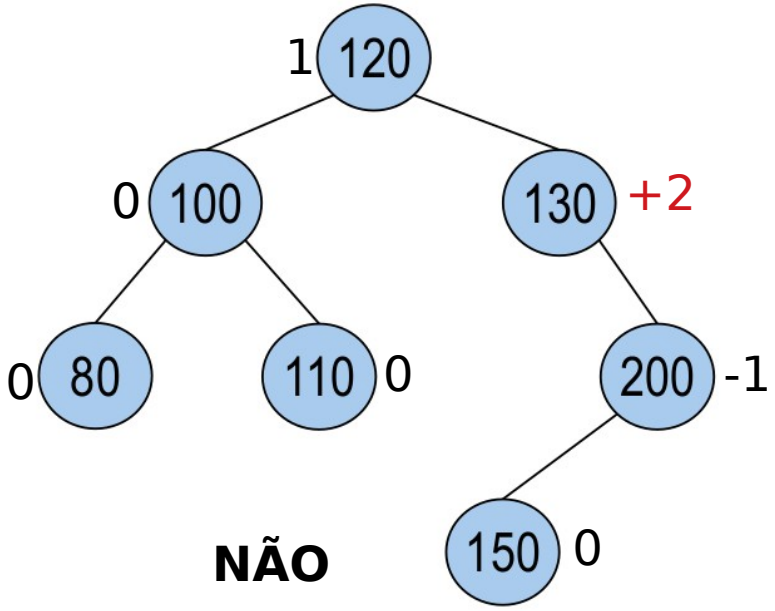


Árvore AVL

- **É AVL?**
  - Verifique se cada árvore é AVL. Determine o fator de balanceamento para cada nó.



SIM



NÃO

Árvore AVL

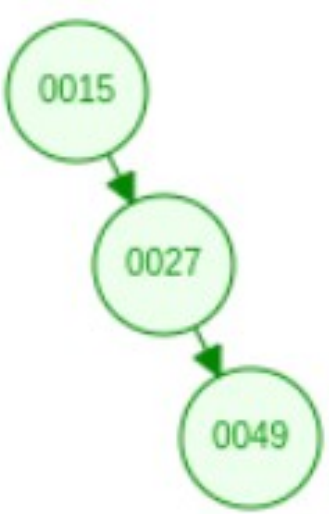
- **Rotação**
  - Se a inserção ou exclusão fizer com que a árvore perca as propriedades de árvore AVL, deve-se reestruturar a árvore com uma operação chamada Rotação.
  - A Rotação mantém a ordem das chaves, de modo que a árvore resultante continue sendo árvore binária de busca válida e árvore AVL válida.

Árvore AVL

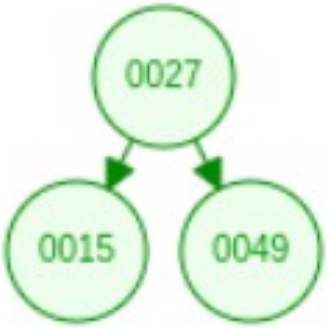
- **Rotação Simples (sinais do FB iguais)**
  - Direita - Quando inserir um novo nó à esquerda da subárvore esquerda provoca o desbalanceamento.
  - Esquerda – Quando inserir um novo nó à direita da subárvore direita provoca o desbalanceamento.
- **Rotação Dupla (sinais do FB diferentes)**
  - Direita, esquerda – Quando inserir um novo nó à direita da subárvore esquerda provoca o desbalanceamento.
  - Esquerda, direita - Quando inserir um novo nó à esquerda da subárvore direita provoca o desbalanceamento.

Árvore AVL

- **Ex de rotação simples esquerda**

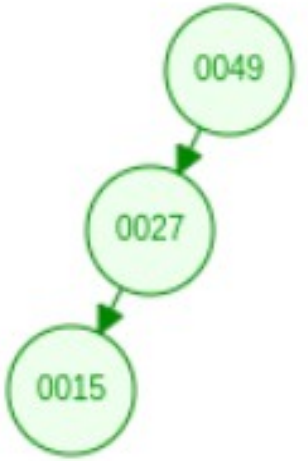


- FB(15) = 0 – 2 = -2
- FB(27) = 0 – 1 = -1
- FB(49) = 0 – 0 = 0
- Observe que o nó 15 está desbalanceado, pois não atende à condição **FB(15) >= -1 e <= 1**
- O nó desbalanceado e seu filho da direita possuem o **mesmo sinal no FB**, indicando rotação simples.
- O nó desbalanceado tem **FB com sinal de -** indicando olha o filho da direita (pesou na T→dir).
- Nesse caso aplica-se a rotação simples à esquerda: RSE(15,27);

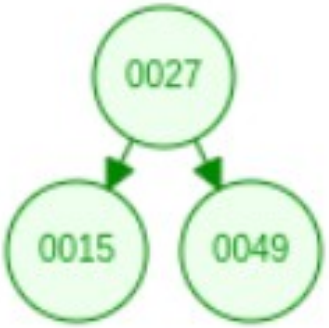


Árvore AVL

- **Ex de rotação simples direita**



- FB(15) = 0 – 0 = 0
- FB(27) = 0 – 1 = 1
- FB(49) = 0 – 0 = 2
- Observe que o nó 49 está desbalanceado, pois não atende à condição **FB(49) >= -1 e <= 1**
- O nó desbalanceado e seu filho da esquerda possuem o **mesmo sinal no FB**, indicando rotação simples.
- O nó desbalanceado tem **FB com sinal de +** indicando olha o filho da esquerda (pesou na T→esq).
- Nesse caso aplica-se a rotação simples à direita: RSD(49,27);



Árvore AVL

- **Rotação à direita**

```
// Faz a rotação a direita
NO *rotacao_direita(NO *y) {
    NO *x = y->esq;
    NO *T2 = x->dir;

    // Realiza a rotacao
    x->dir = y;
    y->esq = T2;

    // Atualiza alturas
    y->altura = obter_maximo(obter_altura(y->esq), obter_altura(y->dir))+1;
    x->altura = obter_maximo(obter_altura(x->esq), obter_altura(x->dir))+1;

    // Nova raiz eh retornada
    return x;
}
```

Árvore AVL

- **Rotação à esquerda**

```
// Faz a rotação a esquerda
NO *rotacao_esquerda(NO *x){
    struct no *y = x->dir;
    struct no *T2 = y->esq;

    // Realiza a rotacao
    y->esq = x;
    x->dir = T2;

    // Atualiza alturas
    x->altura = obter_maximo(obter_altura(x->esq), obter_altura(x->dir))+1;
    y->altura = obter_maximo(obter_altura(y->esq), obter_altura(y->dir))+1;

    // Nova raiz eh retornada
    return y;
}
```

Árvore AVL

- **Inserindo balanceado (início)**

```
// Inserir um no de modo a manter seu fator de balanceamento >= -1 e <= 1
NO *insere(NO *raiz, int chave ){

    // Insercao normal
    if (raiz == NULL)
        return(criaNO(chave));

    if ( chave < raiz->chave ) {
        printf( "%10d <-- esquerda de %d\n", chave, raiz->chave );
        raiz->esq = insere ( raiz->esq, chave );
    }
    else if ( chave > raiz->chave ) {
        printf( "%10d --> direita de %d\n", chave, raiz->chave );
        raiz->dir = insere ( raiz->dir, chave );
    }
    else {
        printf ( "A chave %d ja existe!\n", chave );
        return raiz;
    }
```

Árvore AVL

```
// Atualiza a altura do pai
if(obter_altura(raiz->esq) > obter_altura(raiz->dir) )
    raiz->altura = 1 + obter_altura(raiz->esq);
else
    raiz->altura = 1 + obter_altura(raiz->dir);

// Obtem o fator de balanceamento
int fb = obter_fb(raiz);

// Caso o no inserido esteja desbalanceado, ha 4 casos a se observar:
/* 1)
 * Rotacao simples DIREITA, pois:
 * inserir na esquerda da subarvore esquerda
 * causou o desbalanceamento.
*/
if (fb > 1 && chave < raiz->esq->chave)
    return rotacao_direita(raiz);
/* 2)
 * Rotacao simples ESQUERDA, pois:
 * inserir na direita da subarvore direita
 * causou o desbalanceamento.
*/
if (fb < -1 && chave > raiz->dir->chave)
    return rotacao_esquerda(raiz);
```

Árvore AVL

- **Inserindo balanceado (fim)**

```
    /* 3)
     * Rotacao DUPLA: esquerda, direita:
     * inserir na direita da subarvore esquerda
     * causou o desbalanceamento.
    */
    if (fb > 1 && chave > raiz->esq->chave){
        raiz->esq = rotacao_esquerda(raiz->esq);
        return rotacao_direita(raiz);
    }
    /* 4)
     * Rotacao DUPLA: direita, esquerda:
     * inserir na esquerda da subarvore direita
     * causou o desbalanceamento.
    */
    if (fb < -1 && chave < raiz->dir->chave){
        raiz->dir = rotacao_direita(raiz->dir);
        return rotacao_esquerda(raiz);
    }

    return raiz;
}
```

Árvore AVL

- **Imprimindo a árvore**

```c
// Imprime em pre-ordem: imprime raiz, visita a sub esq, visita a sub dir
void imprime_preordem(NO *raiz) {
    if(raiz != NULL){
        printf(" %d ", raiz->chave);
        imprime_preordem(raiz->esq);
        imprime_preordem(raiz->dir);
    }
}

// Imprime em pos-ordem: visita a sub esq, visita a sub dir, imprime raiz
void imprime_posordem(NO *raiz) {
    if(raiz != NULL){
        imprime_posordem(raiz->esq);
        imprime_posordem(raiz->dir);
        printf(" %d ", raiz->chave);
    }
}

// Imprime em ordem: visita a sub esq, imprime raiz, visita a sub dir
void imprime_emordem(NO *raiz) {
    if(raiz != NULL){
        imprime_emordem(raiz->esq);
        printf(" %d ", raiz->chave);
        imprime_emordem(raiz->dir);
    }
}
```

Árvore AVL

- **Exercício**

- Observe a função de excluir na árvore binária de busca e a função de inserir com balanceamento na árvore AVL e desenvolva uma função que exclui na árvore AVL mantendo o balanceamento da árvore.