

**2ª Lista de Exercícios de Pesquisa e Ordenação**  
**Prof. Glauber Cintra – Entrega: 07/dez/2015**

DEZ

EXCERENTE!!

**Alunos:** Levi Moreira de Albuquerque

Marcus Reuber

Matheus Vasconcelos – Utilizamos a matrícula e os dados deste aluno

d <sub>1</sub>	d <sub>2</sub>	d <sub>3</sub>	d <sub>4</sub>	d <sub>5</sub>	d <sub>6</sub>
0	2	0	2	2	1

Número de matrícula

Ano de nascimento

a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>
1	9	9	0

$v_1=6d_1+7a_4$	$v_2=6d_2+7d_1$	$v_3=6d_3+7d_2$	$v_4=6d_4+7d_3$	$v_5=6d_5+7d_4$	$v_6=6d_6+7d_5$	$v_7=6a_1+7d_6$	$v_8=6a_2+7a_1$	$v_9=6a_3+7a_2$	$v_{10}=6a_4+7a_3$
00	12	14	12	26	20	13	61	117	63

- 1) Escreva uma versão não recursiva do algoritmo de **busca binária**.

**Algoritmo BB\_Iterativa**

**Entrada:** um vetor L em ordem crescente, um valor x e as posições início e fim

**Saída:** Sim, se x existe no vetor

Não, caso contrário

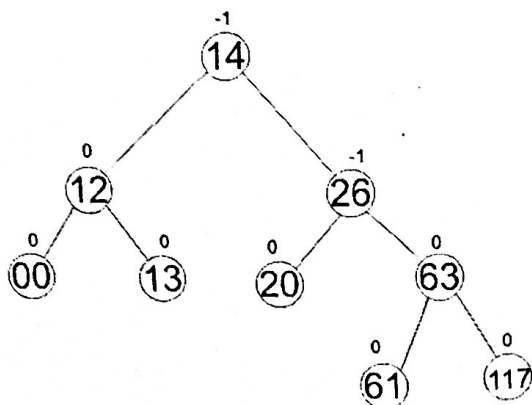
```

se inicio > fim
    devolva não e pare
enquanto inicio <= fim
    meio = (inicio + fim) / 2 //divisão inteira
    se x = L[meio]
        devolva sim e pare
    se x < L[meio]
        fim = meio - 1
    senão
        inicio = meio + 1
    devolva não e pare
    
```

✓ 0,9

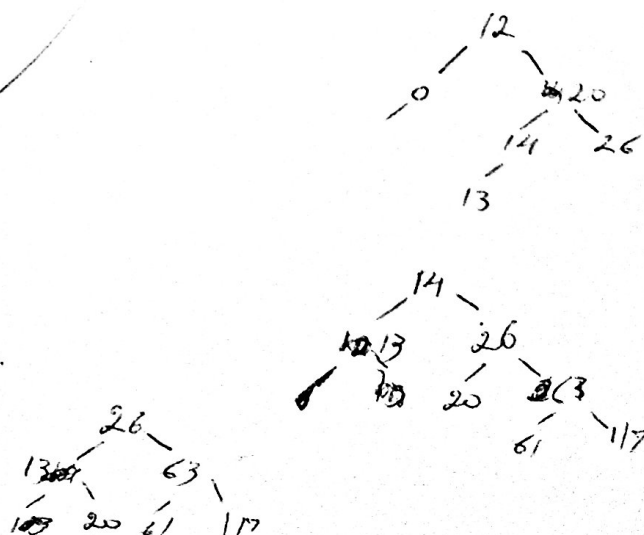
- 2) Insira as chaves  $v_1, v_2, \dots, v_{10}$ , nessa ordem, numa **árvore AVL**. Em seguida, remova  $v_1, v_2$ , e  $v_3$ , nessa ordem, da árvore. Desenhe como ficou a árvore, incluindo o **bal** de cada nó.

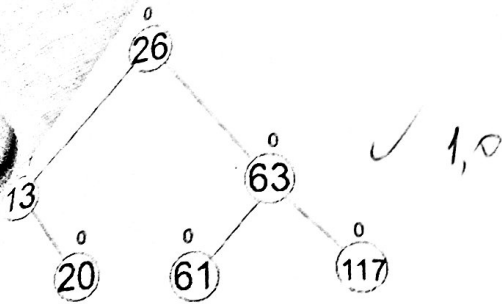
**Após inserções**



A chave  $v_4$  não foi inserida, pois já existia na árvore.

**Após Remoções**





3) Escreva uma função que receba um ponteiro para a raiz de uma **árvore AVL** e devolva a altura da árvore.

Podemos utilizar dois métodos para resolver este problema.

Por ser uma árvore binária de busca, a árvore AVL pode utilizar o mesmo procedimento que se usaria para calcular a altura de uma árvore de busca binária qualquer. Uma modificação foi necessária, visto que o algoritmo originalmente considera uma árvore que só contém a raiz da árvore com altura 0. Na árvore AVL a altura de uma árvore que só tem a raiz é 1.

#### Algoritmo alturaAVL

**Entrada:** r, um ponteiro para a raiz da árvore AVL

**Saída:** 0 se a árvore estiver vazia ou a altura da árvore caso contrário.

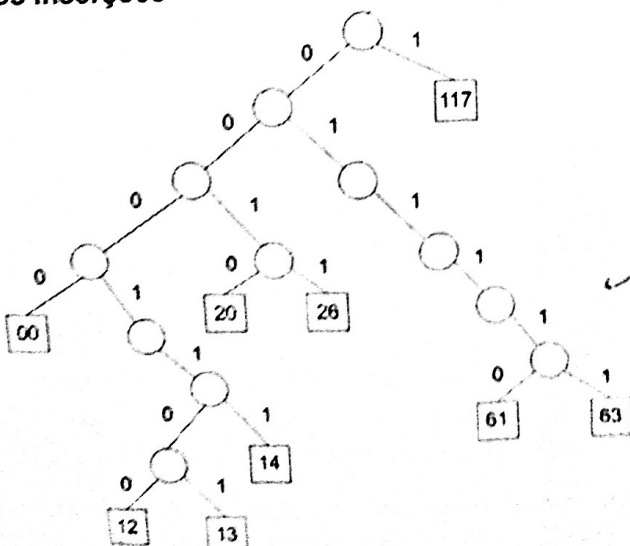
```

se r = nulo
  devolva 0
senão
  [
    he = alturaAVL(r.esq)
    hd = alturaAVL(r.dir)
    se he < hd
      [devolva hd+1]
    senão
      [devolva he+1]
  ]

```

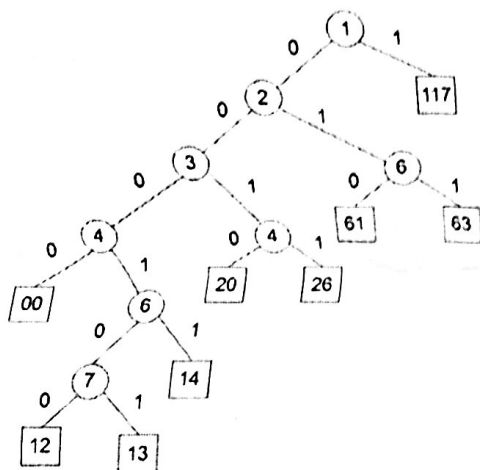
4) Mostre como ficaria uma **árvore trie de ordem 2** após a inserção da representação binária (com 7 bits) das chaves  $v_1, v_2, \dots, v_{10}$ . Em seguida, remova a representação binária de  $v_2, v_3$  e  $v_4$  e mostre como ficaria a árvore.

**Após inserções**

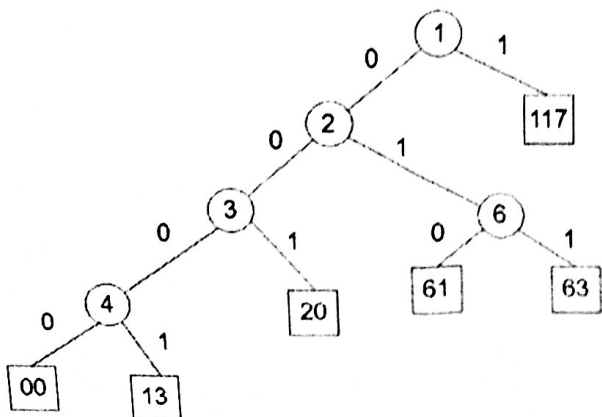


[illegible]

### Após as inserções



### Após as remoções



- Um nó em nossa árvore Patricia contém os seguintes campos:

- **Bit:** se esse valor for 0, esse nó é uma folha (uma chave válida), se for 1 é um nó interno.
- **Info:** Caso o campo bit seja 0, esse campo contém uma chave válida. Caso seja 1, esse campo contém a posição do bit que deverá ser olhado no momento da busca.

- **Esq:** um ponteiro para o filho esquerdo do nó, associado ao símbolo de bit 0
- **Dir:** um ponteiro para o filho direito do nó, associado ao símbolo de bit 1

#### Algoritmo BuscaP

**Entrada:** r, um ponteiro para a raiz de uma árvore Patricia. x um valor a ser pesquisado.

**Saída:** Sim, se x ocorre na árvore. Não, caso contrário

```

se r = nulo
    devolva não e pare
se r.bit = 0
    se x = r.info
        devolva sim e pare
    senão
        devolva não e pare
senão
    bit = extraibit(x, r.info)
    se bit = 0
        devolva buscaP (r.esq, x)
    senão
        devolva buscaP (r.dir, x)

```

4/09

A função extraibit :

#### Algoritmo extraibit

**Entrada:** Um número inteiro com M bits, e a posição K do bit que deve ser extraído.

**Saída:** O bit da posição K

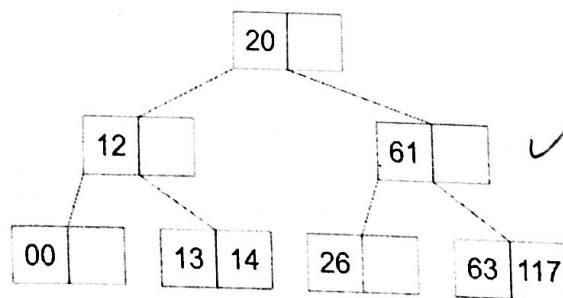
```

bit = x << (k-1)
Devolve bit >> (M-1)

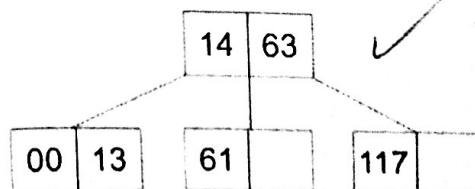
```

- 7) Mostre como ficaria uma **árvore B de ordem 1** após a inserção das chaves  $v_1, v_2, \dots, v_{10}$ , nesta ordem. Em seguida, remova  $v_4, v_5$ , e  $v_6$  e mostre como ficaria a árvore.

**Após inserções**



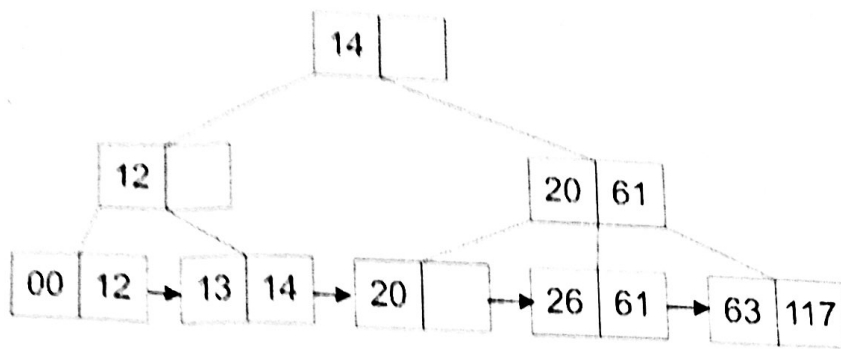
**Após remoções**



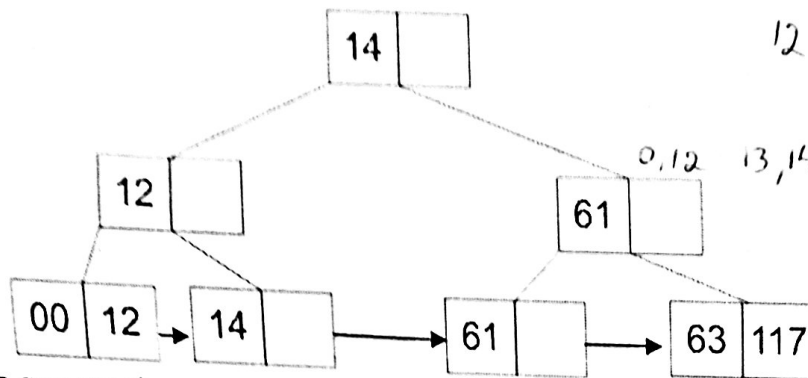
20  
63  
14  
117

- 8) Mostre como ficaria uma **árvore B+ de ordem 1** após a inserção das chaves  $v_1, v_2, \dots, v_{10}$ , nesta ordem. Em seguida, remova  $v_5, v_6$  e  $v_7$  e mostre como ficaria a árvore.

**Após inserções**



Após remoções



- 9) Escreva uma função que receba um ponteiro para a raiz de uma **árvore B de ordem m** e devolva a quantidade de chaves contidas na árvore.

#### Algoritmo ContaC

**Entrada:** O ponteiro para a raiz de uma árvore B, C uma variável passada por referência inicialmente igual a 0

**Saída:** A quantidade de chaves nessa árvore armazenada em C

se  $r == \text{nulo}$

devolva 0 e pare //árvore vazia

senão

$c += r.\text{numchaves}$

para  $i = 0$  até  $r.\text{numchaves}$

contac( $r \rightarrow p[i]$ , c)

- 10) Mostre como ficaria uma tabela de **hashing fechado** com 11 posições, após a inserção das chaves  $v_1, v_2, \dots, v_{10}$ , nesta ordem (nessa e na próxima questão, os valores associados às chaves devem ser ignorados). Utilize a seguinte função de **hashing**:  $h(x) = x \bmod 11$ . Em seguida, remova  $v_1, v_2$  e  $v_3$  (nesta ordem) e mostre como ficaria a tabela.

Após inserções

0	00
1	12
2	13
3	14
4	26
5	
6	61
7	117
8	63
9	20
10	

Após remoções

0	
1	
2	13
3	
4	26
5	
6	61
7	117
8	63
9	20
10	

- 11) Mostre como ficaria uma tabela de **hashing aberto** com 7 posições, após a inserção das chaves  $v_1, v_2, \dots, v_{10}$ , nesta ordem. Utilize a seguinte função de **hashing**:  $h(x) = x \bmod 7$ . Em seguida, remova  $v_7, v_8$  e  $v_9$  (nesta ordem) e mostre como ficaria a tabela.

exemplos

0	14	→	63	
1				
2				
3	12	→	26	→ 61 → 117
4	20	→	13	

Após remoções

0	00	→	14	→	63
1					
2					
3					
4					
5	12	→	26		
6	20				

12) Explique o que é a carga de uma tabela de hashing e diga quando ela é considerada *baixa*. Explique também o que é uma boa função de hashing.

A carga de uma tabela de hashing é a razão entre a quantidade de chaves na tabela e a quantidade de posições. Numa tabela que implementa o Hashing Fechado uma carga é considerada baixa se ela é inferior a 50%, já em uma tabela de Hashing Aberto a carga é baixa se limitada por uma constante.

Uma boa função de hashing é aquela que pode ser computada em tempo constante e que faz um bom espalhamento das chaves na tabela.

✓ 1,0