Árvore Binária de Busca – Modos de Travessia

Com base no que foi visto implemente, utilizando recursividade, as operações de percurso de uma árvore em pré-ordem, in-ordem e pós-ordem.

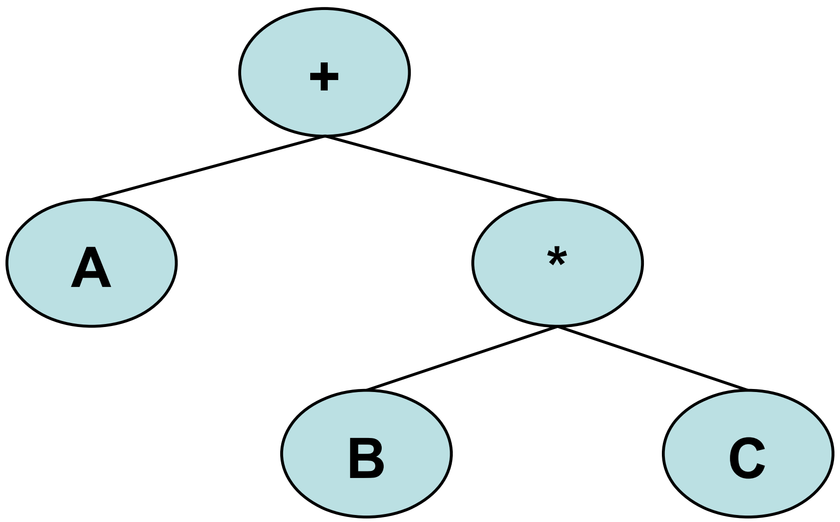
```c
void precursoPreOrdem(ARV_BIN_ENC arvore)
{
    if (arvore)
    {
        printf("%d ", info(arvore));      /*V*/
        precursoPreOrdem(left(arvore));   /*L*/
        precursoPreOrdem(right(arvore)); /*R*/
    }
}
```

```c
void precursoInOrdem(ARV_BIN_ENC arvore)
{
    if (arvore)
    {
        precursoInOrdem(left(arvore));   /*L*/
        printf("%d ", info(arvore));     /*V*/
        precursoInOrdem(right(arvore));  /*R*/
    }
}
```

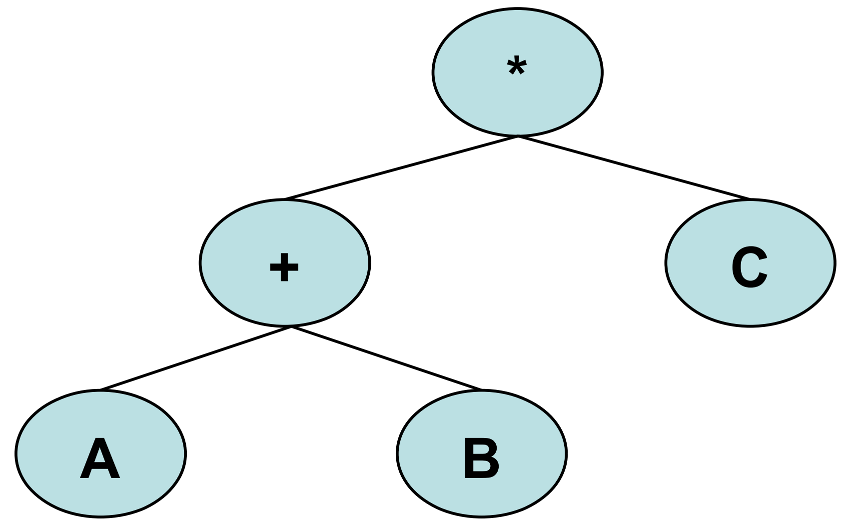
```c
void precursoPosOrdem(ARV_BIN_ENC arvore)
{
    if (arvore)
    {
        precursoPosOrdem(left(arvore));  /*L*/
        precursoPosOrdem(right(arvore)); /*R*/
        printf("%d ", info(arvore));     /*V*/
    }
}
```

Árvore Binária de Busca – Modos de Travessia

Para finalizarmos nossa análise sobre este tema verifique como ficaria o percurso das árvores abaixo em pré-ordem, in-ordem e pós-ordem?



Pré-ordem
+A*BC
In-ordem
A+B*C
Pós-ordem
ABC*+



Pré-ordem
***+ABC**
In-ordem
A+B*C
Pós-ordem
AB+C*

Árvore Binária de Busca

Vimos a seguinte definição para o TAD ARV_BIN_ENC

```
typedef struct node
{
    int info;
    struct node *left;
    struct node *right;
    struct node *father;
} NODE;
typedef NODE * ARV_BIN_ENC;
```

e implementados as operações: **maketree, setleft, setright, info, left, right, father, brother, isleft e isright.**

Árvore Binária de Busca

Considerando a mesma definição, do TAD ARV_BIN_ENC, para o TAD ARV_BIN_BUSCA, inclusive suas operações, implemente a operação de inserção de um elemento em uma árvore binária de busca. A qual terá o seguinte protótipo:

void ins_ele (ARV_BIN_BUSCA *arv, int v);

onde **arv** é uma referência para uma referência para a raiz de uma árvore binária de busca qualquer, que inclusive pode ser vazia, e **v** o valor a ser inserido.

Árvore Binária de Busca

A remoção de um nó em uma árvore binária de busca já não é tão trivial.

A que é proporcional a complexidade do algoritmo de remoção?

Ao número de filhos que o nó a ser removido possui.

Cite as possibilidades.

Nó sem filho – o ponteiro correspondente a seu ascendente é ajustado para nulo e a memória ocupada pelo nó é liberada.

Árvore Binária de Busca

Nó com um filho – o ponteiro correspondente a seu ascendente é ajustado para apontar para o filho do nó e a memória ocupada pelo nó é liberada.

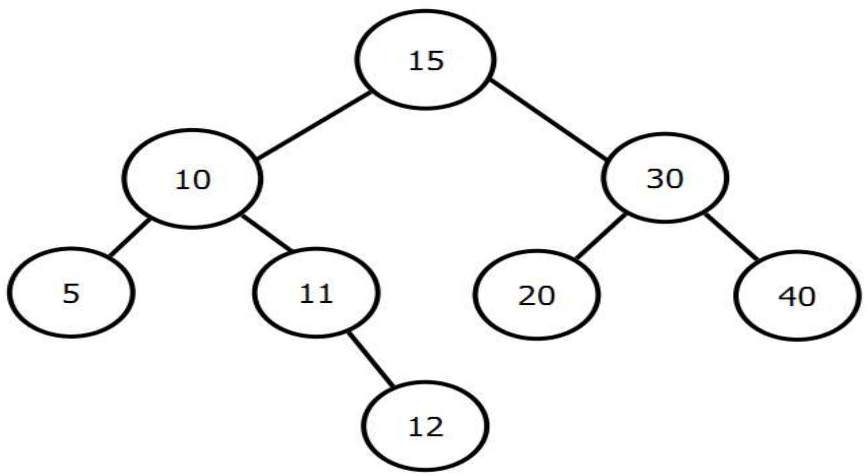
Nó com dois filhos – para este caso nenhuma operação de apenas uma etapa pode ser executada, discutiremos duas soluções para este caso.

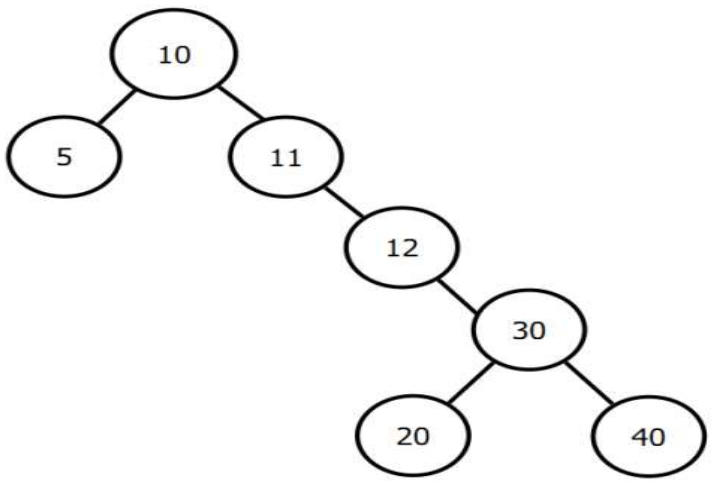
Remoção por fusão e remoção por cópia.

Na remoção por fusão, uma das duas subárvores do nó é extraída e anexada à outra subárvore.

Árvore Binária de Busca

Para uma melhor compreensão do processo analisaremos a árvore abaixo considerando a remoção do nó com valor 15.





Árvore Binária de Busca

Com base no que foi discutido codifique uma função, na linguagem C, que implemente a remoção por fusão.

Árvore Binária de Busca

Outra solução é a Remoção por cópia.

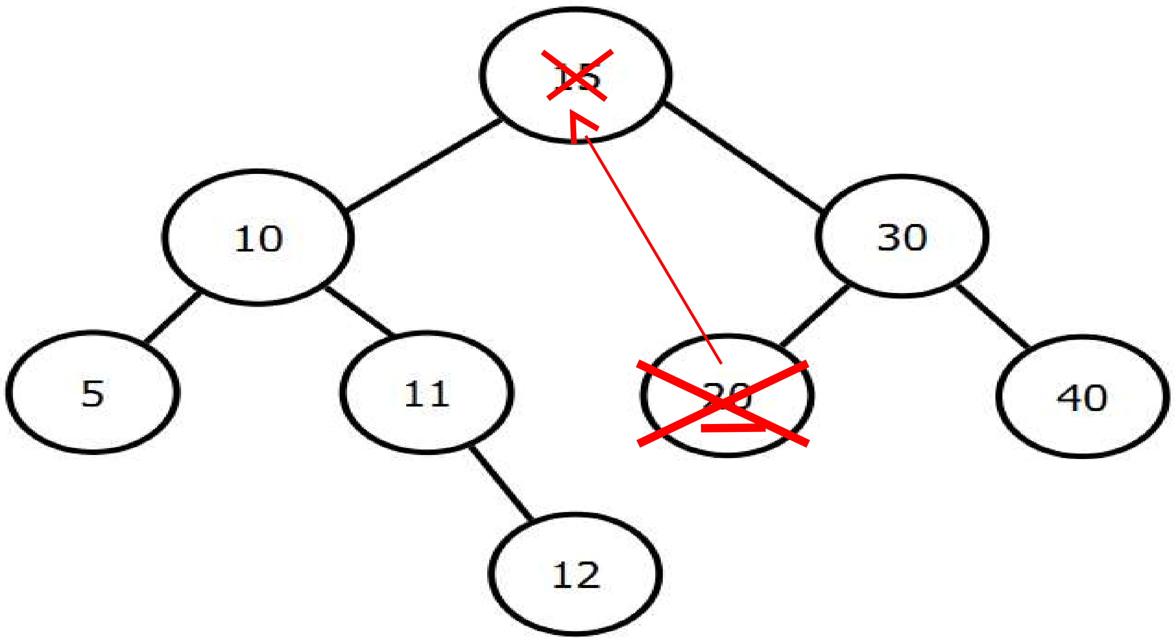
Proposta por Thomas Hibbard e Donald Knuth, propõe que um nó com dois filhos a ser removido pode ser reduzido a uma das duas situações básicas: nó com apenas um filho e nó sem nenhum filho.

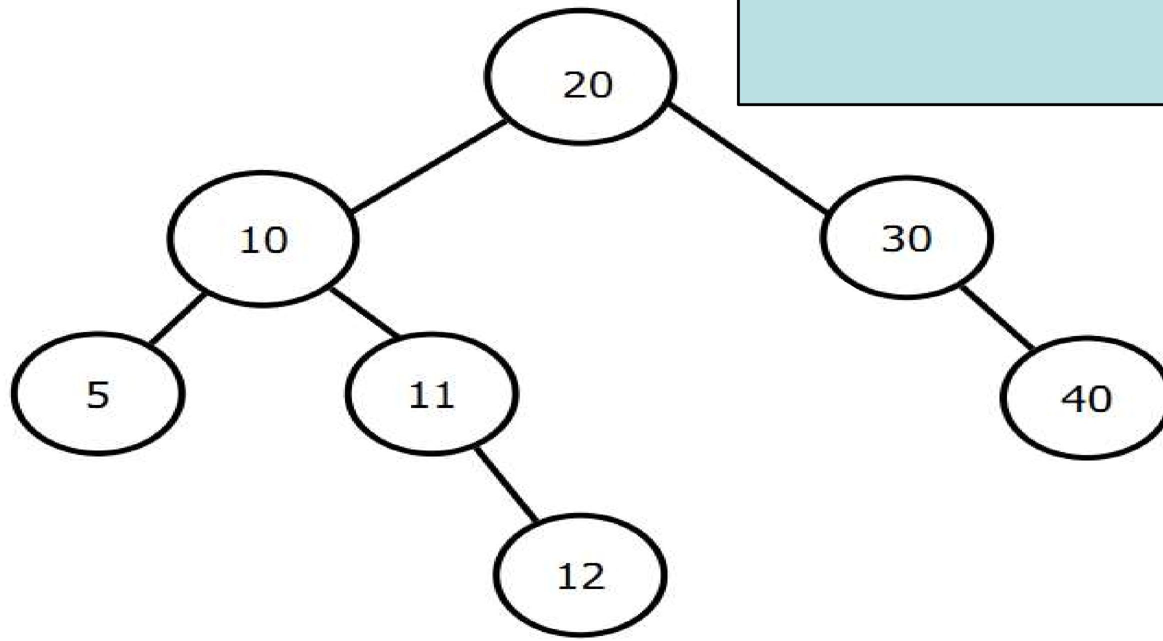
Isso é feito substituindo pela chave de seu sucessor imediato a chave que está sendo removida e em seguida removendo o nó que continha a chave do sucessor imediato.

Obs.: o sucessor imediato de um nó é o nó mais à esquerda em sua subárvore à direita.

Árvore Binária de Busca

Vejamos um exemplo:





Exercício:

Com base no que foi discutido codifique uma função, na linguagem C, que implemente a remoção por cópia.

Árvore Balanceada
Árvore AVL

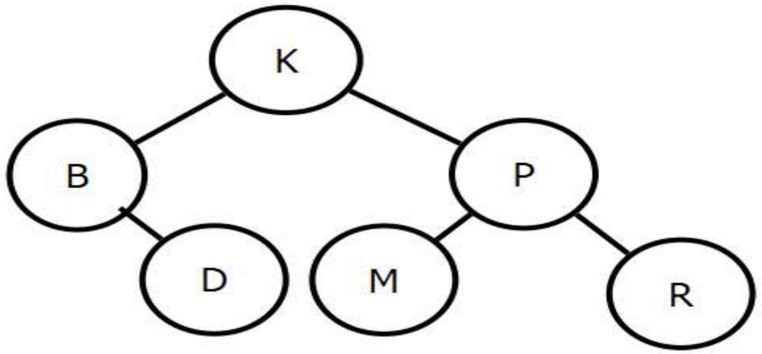
Árvore Balanceada

Além das árvores serem muito apropriadas para representar a estrutura hierárquica de um certo domínio, o processo de busca por um elemento em uma árvore tende a ser muito mais rápido do que em uma lista encadeada.

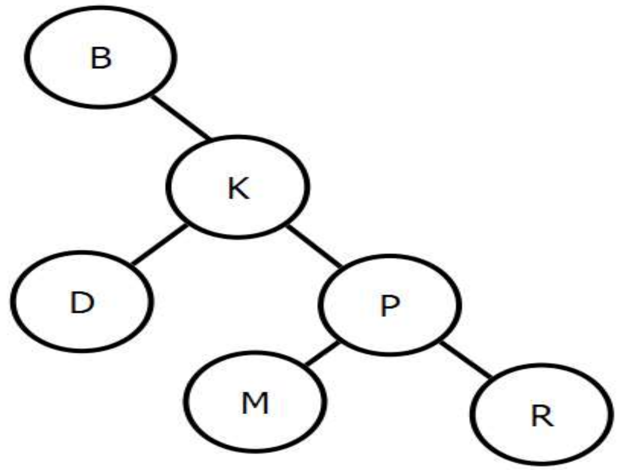
Contudo, para que efetivamente uma busca por um determinado elemento em uma árvore seja eficiente esta, além de ser uma árvore binária de busca, deve ter seus nós adequadamente distribuídos.

Árvore Balanceada

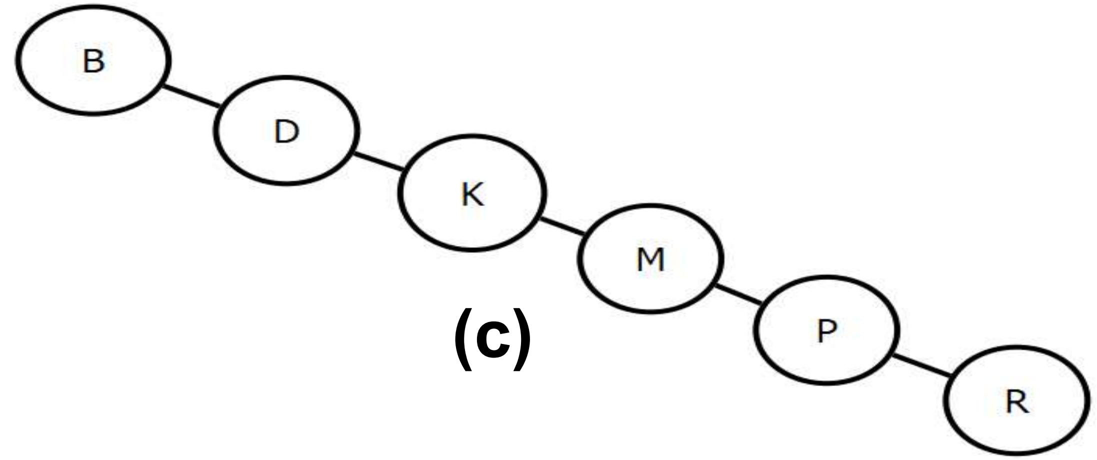
Observe as árvores a seguir:



(a)



(b)



(c)

Sendo assim, podemos estabelecer a seguinte definição: Uma árvore binária é ***balanceada em altura*** ou simplesmente ***balanceada*** se a diferença na altura de ambas as subárvores de qualquer nó na árvore é zero ou um.

Árvore Balanceada

Uma definição complementar é a de árvore ***perfeitamente balanceada***.

Uma árvore binária é ***perfeitamente balanceada*** quando, além de ser balanceada, todas as suas folhas encontram-se em um ou dois níveis.

Para facilitar a visualização, uma árvore que possui 10.000 nós pode ser configurada em uma árvore com altura igual a $log_2(10000) = ceil(13.289) = 14$. Ou seja, qualquer elemento é passível de ser localizado com no máximo 14 comparações se a árvore for perfeitamente balanceada.

Árvore Balanceada

Neste ponto cabe a seguinte pergunta:

Ao se analisar uma árvore pode-se determinar qual dentre os seus nós seria uma raiz adequada para torná-la perfeitamente balanceada?

Qual seria este nó?

O nó cuja sua chave (valor) representa a mediana das chaves presentes nos nós que compõem a árvore, para uma árvore com número impar de nós. Ou o nó cuja sua chave (valor) representa um dentre os dois valores mais próximos da mediana das chaves presentes nos nós que compõem a árvore, para uma árvore com número par de nós.

Árvore Balanceada

Uma estratégia baseada nesta observação que pode ser utilizada para balancear uma árvore é:

- Retire os dados da árvore e armazene-os em um vetor;

- Após todos os dados terem sido armazenados no vetor, ordene-o;

- Agora determine como raiz o elemento do meio do vetor;

- O vetor consistirá agora em dois subvetores. O filho esquerdo da raiz será o nó com valor no meio do subvetor constituído do início do vetor até o elemento escolhido como raiz;

Árvore Balanceada

- Um procedimento similar é adotado para a definição do filho direito da raiz;

- Este processo se repete até não existirem mais elementos a serem retirados do vetor.

Exercício:

Determine o código fonte de uma função, na linguagem C, que implementa este processo.

Árvore Balanceada

O algoritmo apresentado possui um sério inconveniente, pois, caso a árvore já exista, os seus elementos devem ser retirados e colocado em um vetor, para que a mesma seja recriada.

Caso a árvore ainda não exista, o inconveniente ainda persiste. Pois, todos os dados precisam ser colocados em um vetor antes da árvore ser criada.

Uma pequena melhoria pode ser feita. Pois, mesmo que a árvore binária de busca esteja desbalanceada, se for efetuado um percurso in-ordem elimina-se a necessidade de ordenar o vetor.

Árvore Balanceada

Existem formas mais eficientes de se balancear uma árvore.

Um exemplo é o algoritmo denominado DSW (proposto por Colin Day e posteriormente melhorado por Quentin F. Stout e Bette L. Warren). O qual baseia-se em percorrer uma árvore binária de busca tornando-a uma arvore degenerada (similar a uma lista encadeada) e posteriormente percorrê-la novamente tornando-a uma árvore perfeitamente balanceada.

Árvore AVL

Até o momento vimos algoritmos que balanceiam árvores globalmente.

Contudo, o rebalanceamento pode ocorrer localmente se alguma porção da árvore for desbalanceada por uma operação de inserção ou remoção de um elemento da árvore.

Um método clássico, que nomeia uma árvore modificada pelo mesmo como AVL. (Em função de seus idealizadores, Adel’son-Vel’skii e Landis)

Árvore AVL

Uma árvore AVL é uma árvore binária de busca onde a diferença em altura entre as subárvores esquerda e direita de cada nó é no máximo um (positivo ou negativo).

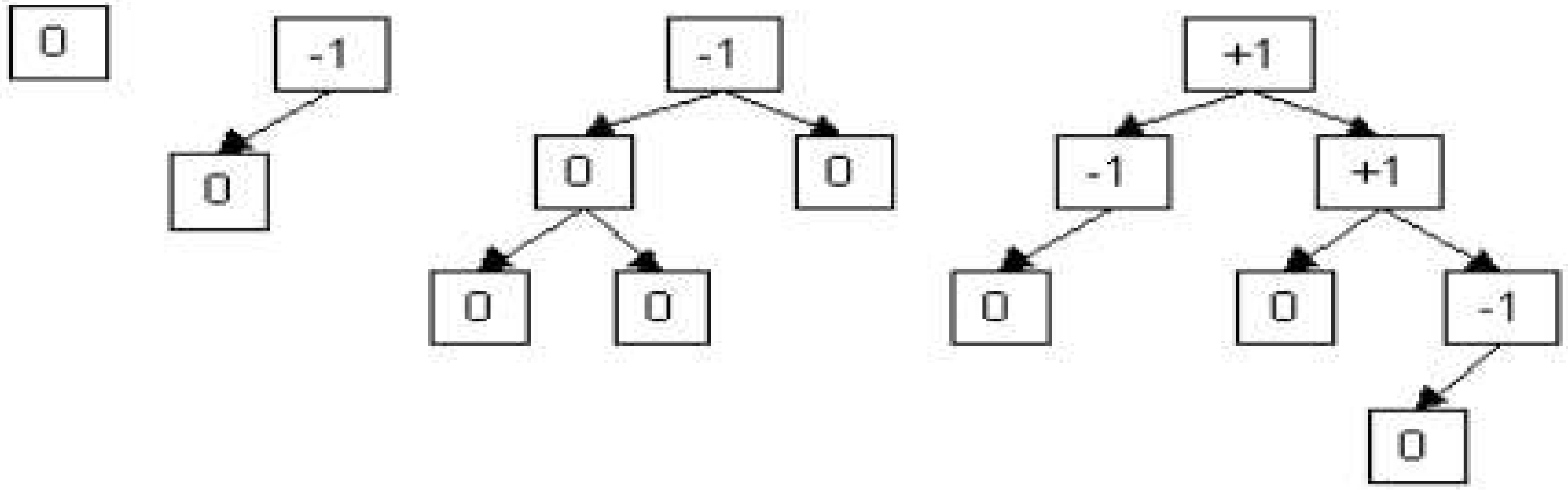
Esta diferença é chamada de fator de balanceamento (FB).

O FB (ou informações que permitam sua obtenção) é acrescentado a cada nó da árvore AVL.

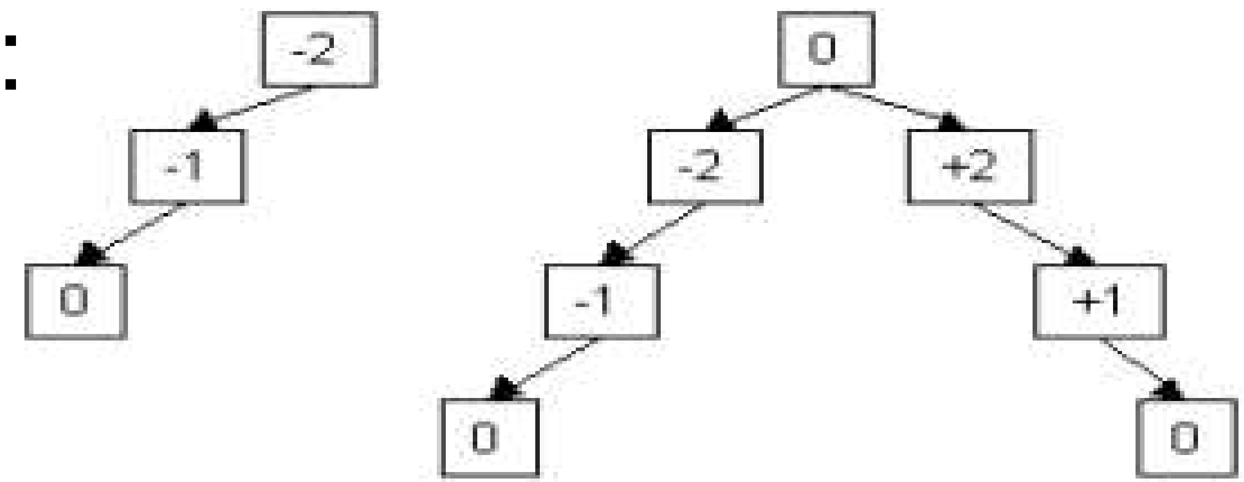
FB (nó *p*) = altura (subárvore direita de *p*) – altura (subárvore esquerda de *p*)

Árvore AVL

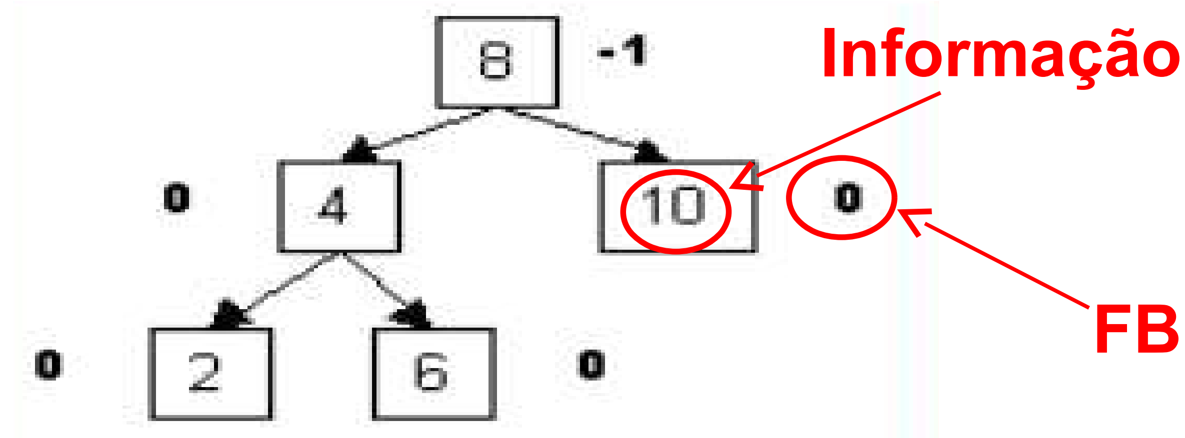
Árvores AVL:



Árvores não AVL:



Árvore AVL



Árvore AVL

Ao inserirmos um novo nó em uma árvore AVL, podemos ou não violar a propriedade de balanceamento.

Caso, ocorra uma violação devemos rebalancear a árvore através da execução de operações de rotação sobre nós da árvore.

Árvore AVL

Após uma inserção que gere um desbalanceamento na árvore AVL podemos nos deparar com duas classes de desbalanceamento.

Onde estas são identificadas com base na análise dos FB's.

Ao inserirmos um novo nó devemos ajustar os FB's, desde o nó inserido até a raiz ou até encontrarmos um fator de balanceamento inaceitável, ou seja, com valor 2 ou -2.

Árvore AVL

Quando o FB do nó filho com valor 1 (+ ou -) possuir o mesmo sinal do FB de seu pai (nó com FB 2 ou -2) trata-se da classe de desbalanceamento 1, que requer apenas uma rotação simples para a árvore ser rebalanceada.

Quando o FB do nó filho com valor 1 (+ ou -) possuir sinal oposto ao FB de seu pai (nó com FB 2 ou -2) trata-se da classe de desbalanceamento 2 que requer uma rotação dupla, ou em outras palavras, duas rotações para a árvore ser rebalanceada.

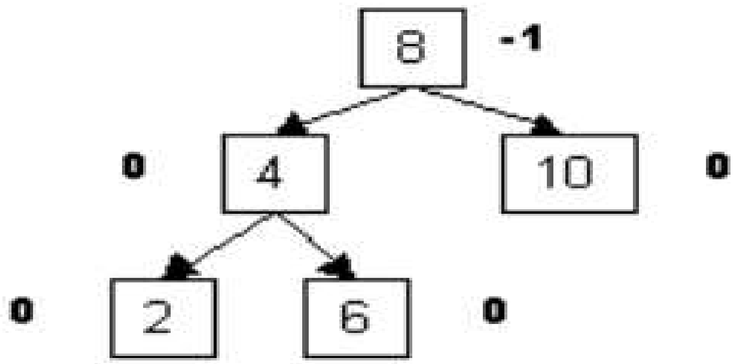
Árvore AVL

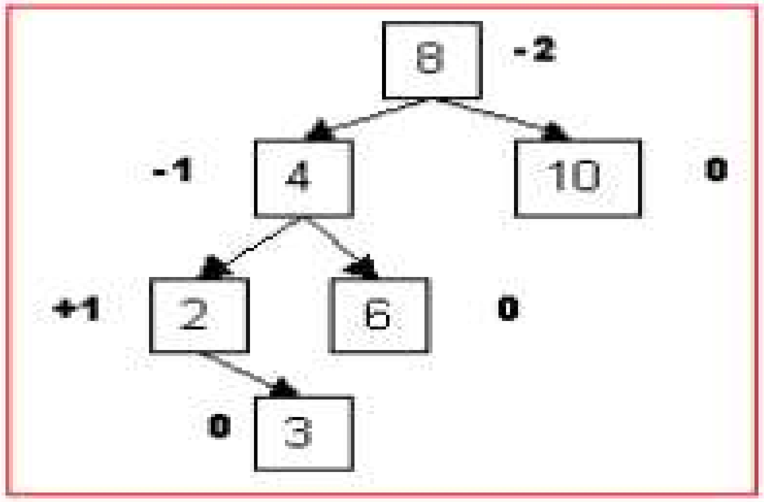
Se o sinal do FB do nó que caracteriza o desbalanceamento for positivo a rotação será para a esquerda.

Se o sinal do FB do nó que caracteriza o desbalanceamento for negativo a rotação será para a direita.

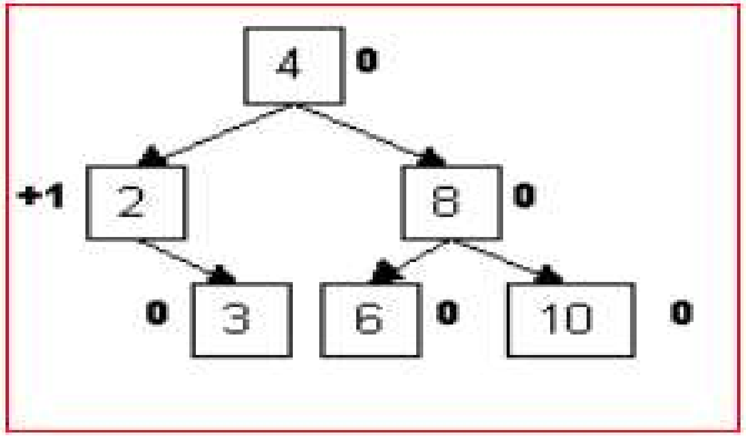
Analisaremos agora um exemplo de cada uma das classes citadas.

Ao inserirmos o valor 3 na árvore abaixo teremos:

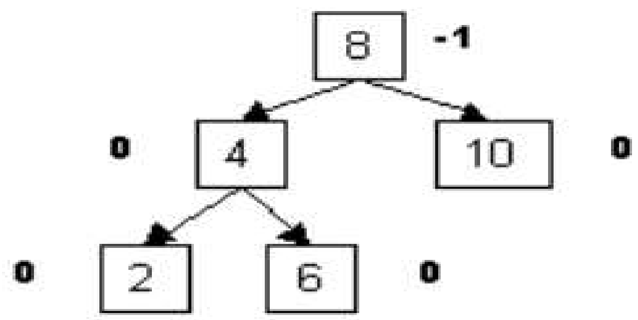


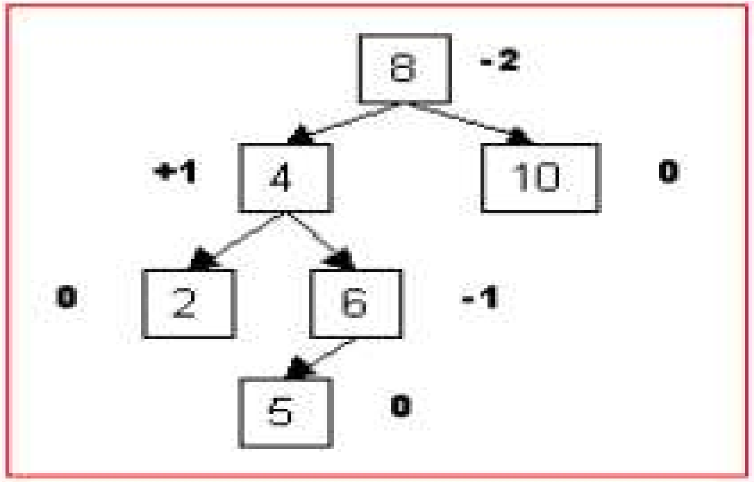


Identificamos a classe 1, que demanda uma rotação simples à direita. Após a rotação teremos:



Ao inserirmos o valor 5 na árvore abaixo teremos:



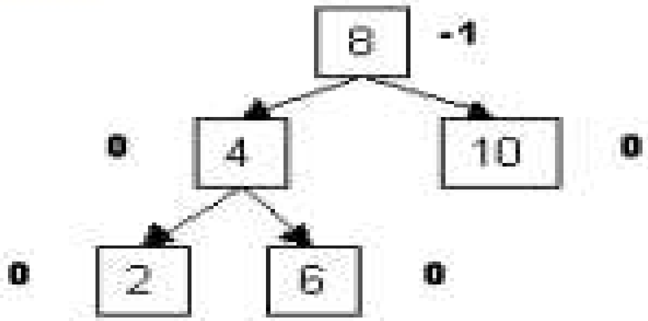


Identificamos a classe 2, que demanda uma rotação dupla à direita.

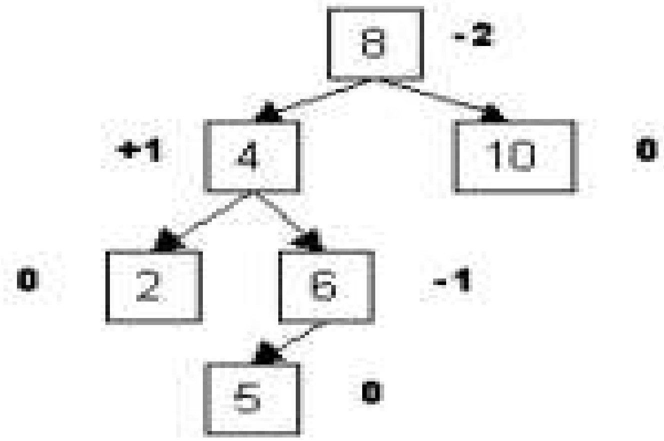
Iniciamos por uma rotação no nó filho com FB +1, no caso, com uma rotação à esquerda devido ao sinal +. Seguindo por uma rotação à direita no nó com FB -2.

O que gera a sequência de árvores a seguir.

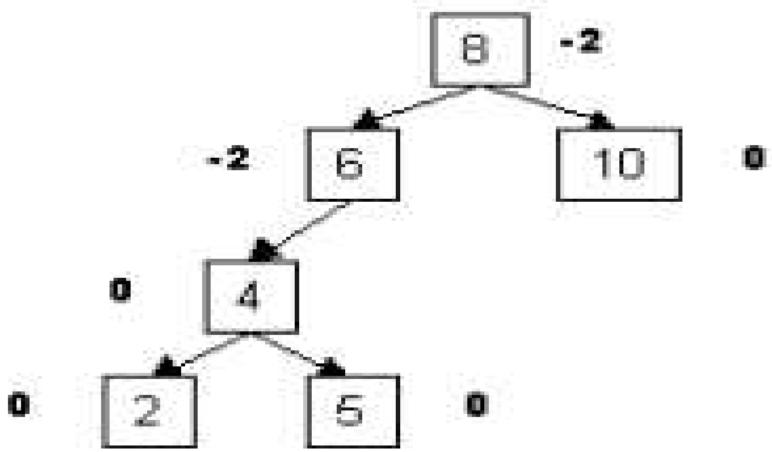
Árvore AVL



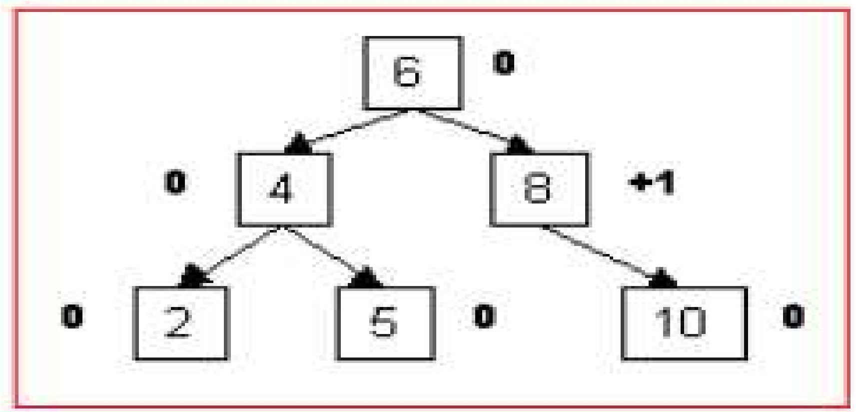
Árvore inicial



Árvore desbalanceada (após inserção do valor 5)



Árvore em processo de balanceamento (após a 1ª rotação)



Árvore balanceada (após a 2ª rotação)

Árvore AVL

Descreva uma sequência de passos para a construção de uma árvore AVL.

1. insira o novo nó normalmente, ou seja, da mesma maneira que insere-se um nó em uma árvore binária de busca;

2. iniciando com o nó pai do nó recém inserido, teste se a propriedade AVL foi violada, ou seja, atualize e teste se algum dos FB's passou a ser 2 ou -2. Temos duas possibilidades:

- **2.1. A condição AVL foi violada**
  - **2.1.1. Execute a operação de rotação conforme o caso (tipo 1 ou tipo 2);**
  - **2.1.2. Volte ao passo 1;**
- **2.2. A condição AVL não foi violada, volte ao passo 1;**

Árvore AVL

Com base no que foi visto, proponha uma estrutura para um nó de uma árvore AVL implementada com alocação dinâmica. Considere que a informação armazenada em cada nó da árvore resumi-se a um valor inteiro.

```
typedef struct nodo
{
    int num, altd, alte;
    struct nodo *dir, *esq;
}NODO;
```

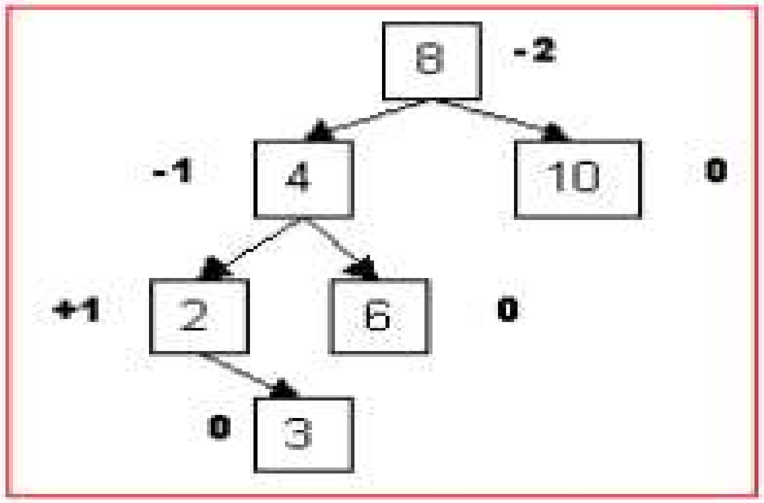
Como seria definido o tipo árvore AVL?

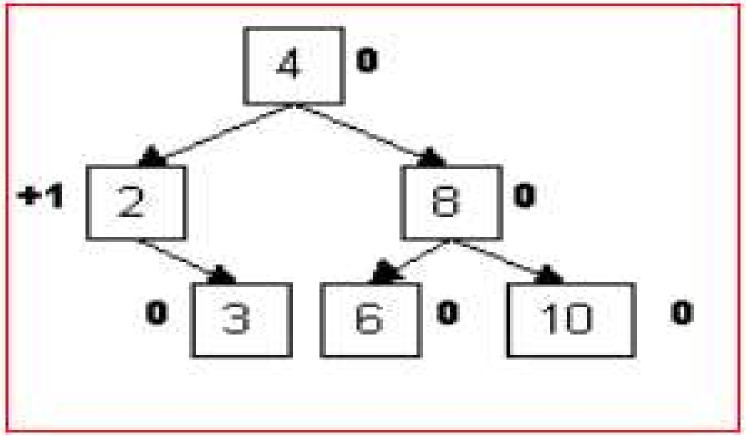
```
typedef NODO * ArvoreAVL;
```

Árvore AVL

Com base no que foi visto, implemente a operação de rotação à direita sobre o nó recebido como parâmetro. Considere o protótipo abaixo para a função que implementará a operação em questão.

void rotacao_direita(ArvoreAVL *arvore);

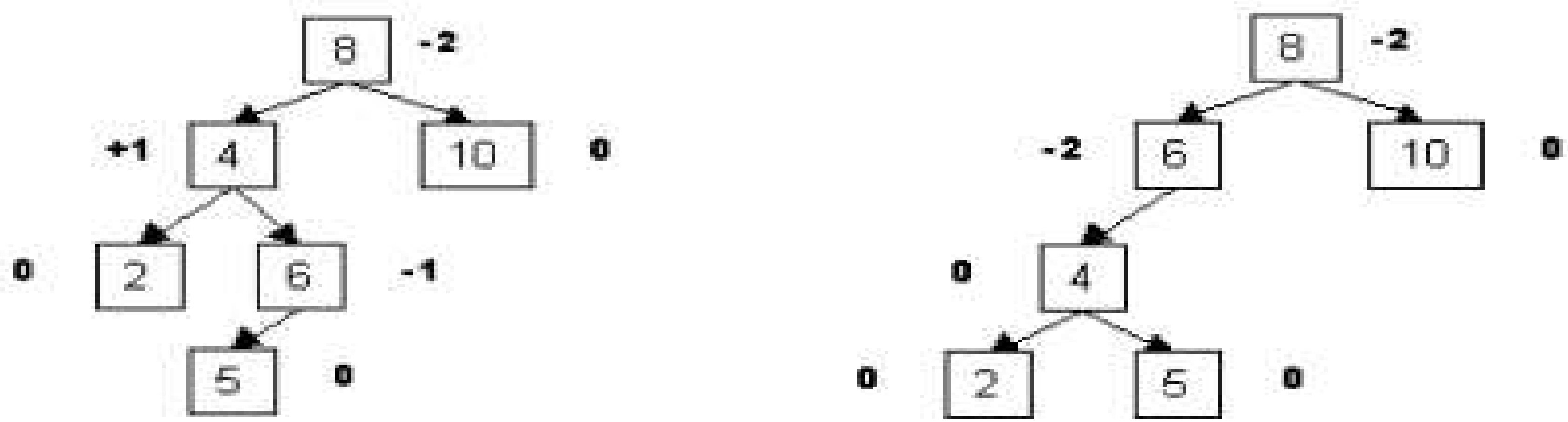




Árvore AVL

Com base no que foi visto, implemente a operação de rotação à esquerda sobre o nó recebido como parâmetro. Considere o protótipo abaixo para a função que implementará a operação em questão.

void rotacao_esquerda(ArvoreAVL *arvore);



Árvore AVL

Com base nas operações de rotação à esquerda e à direita, implemente a operação de balanceamento sobre o nó recebido como parâmetro. Considere o protótipo abaixo para a função que implementará a operação em questão.

void balanceamento (ArvoreAVL *arvore);

Árvores Multívias
Árvore 2-3-4

Árvore Multivias

Vimos inicialmente um conceito mais amplo de árvore e depois o restringimos fixando o número máximo de filhos que um nó pode ter em dois.

Se permitirmos a determinação de mais itens de dados e mais filhos por nó teremos como resultado árvores denominadas **Multivias** ou **M-vias**.

Uma estrutura multivia com algoritmo eficiente deve considerar:

- Tempo de acesso a cada nó;
- Balanceamento da árvore.

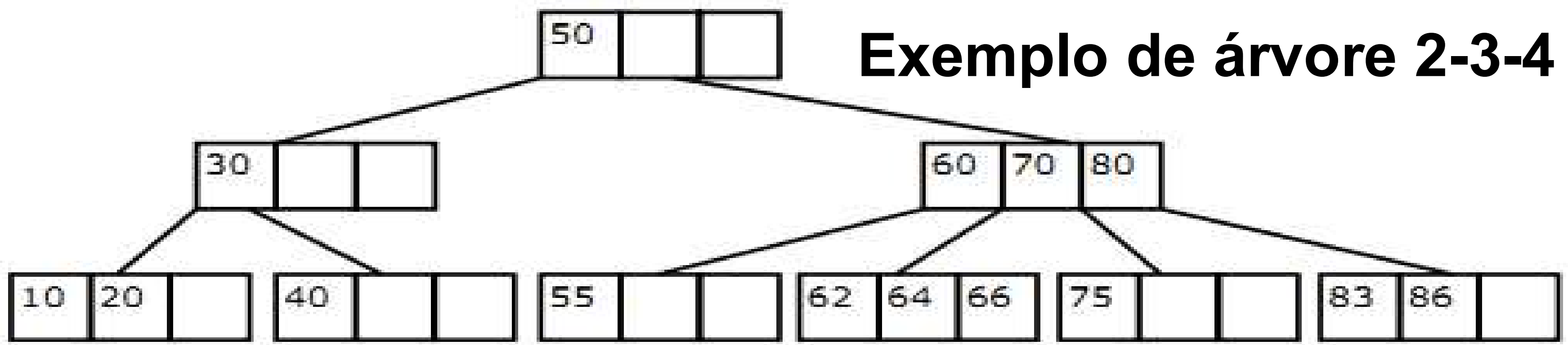
Árvore 2-3- 4

Um bom exemplo de uma árvore multivia é a árvore 2-3-4.

Uma árvore 2-3-4 pode ter até quatro filhos e três itens de dados por nó.

Razões para se estudar árvores 2-3-4:

- São árvores balanceadas;
- São fáceis de implementar;
- Servem como uma introdução para o estudo de árvores B.

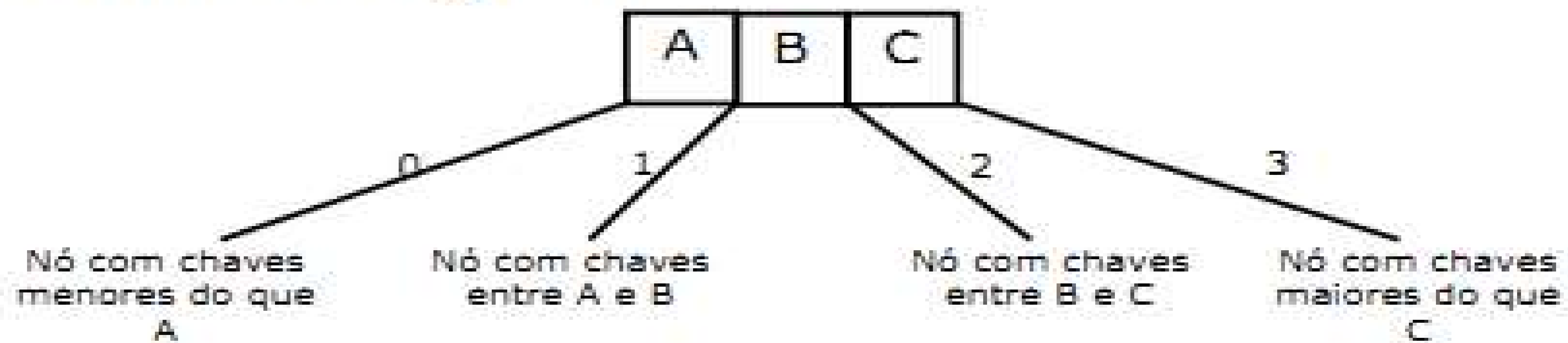


Exemplo de árvore 2-3-4

Árvore 2-3- 4

Em uma árvore binária, todos os filhos com chaves menores que a chave do nó estão “enraizados” no nó filho à esquerda, e todos os filhos com chaves maiores estão “enraizados” no nó filho à direita.

Na árvore 2-3-4 o princípio é o mesmo, com alguns detalhes a mais.

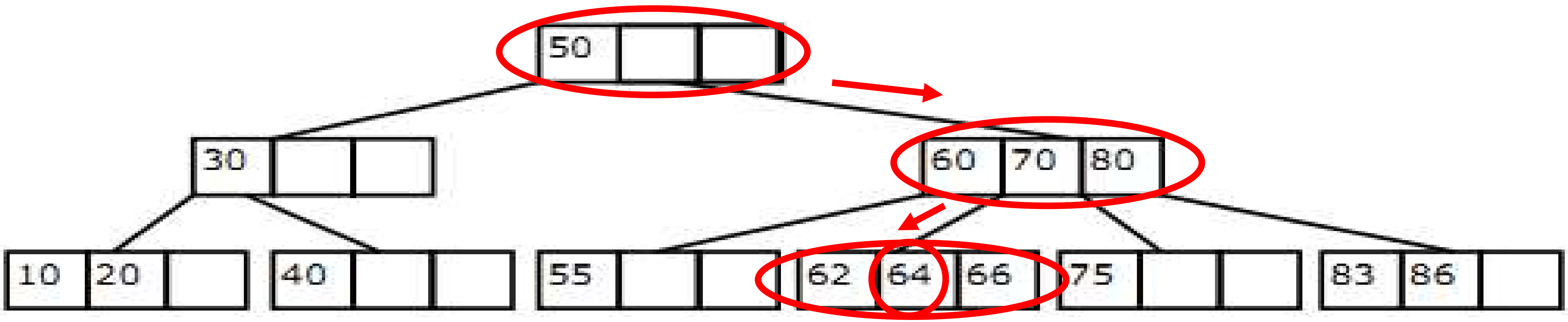


Árvore 2-3- 4

Valores duplicados geralmente não são permitidos, o que possibilita que não nos preocupemos com comparações de chaves iguais.

Veremos agora um exemplo de como ocorre a pesquisa por uma chave em uma árvore 2-3-4.

Simularemos a busca pela chave 64.



Árvore 2-3- 4

Trataremos agora do processo de inserção de uma nova chave em uma árvore 2-3-4.

Novos itens de dados são sempre inseridos nas folhas. Por quê?

Porque se os itens forem inseridos em um nó com filhos, então o número de filhos necessitará ser mudado para manter a estrutura da árvore, a qual estipula que a árvore deve ter um filho a mais do que os itens de dados em cada nó não folha.

O processo de inserção começa pela busca do nó folha apropriado.

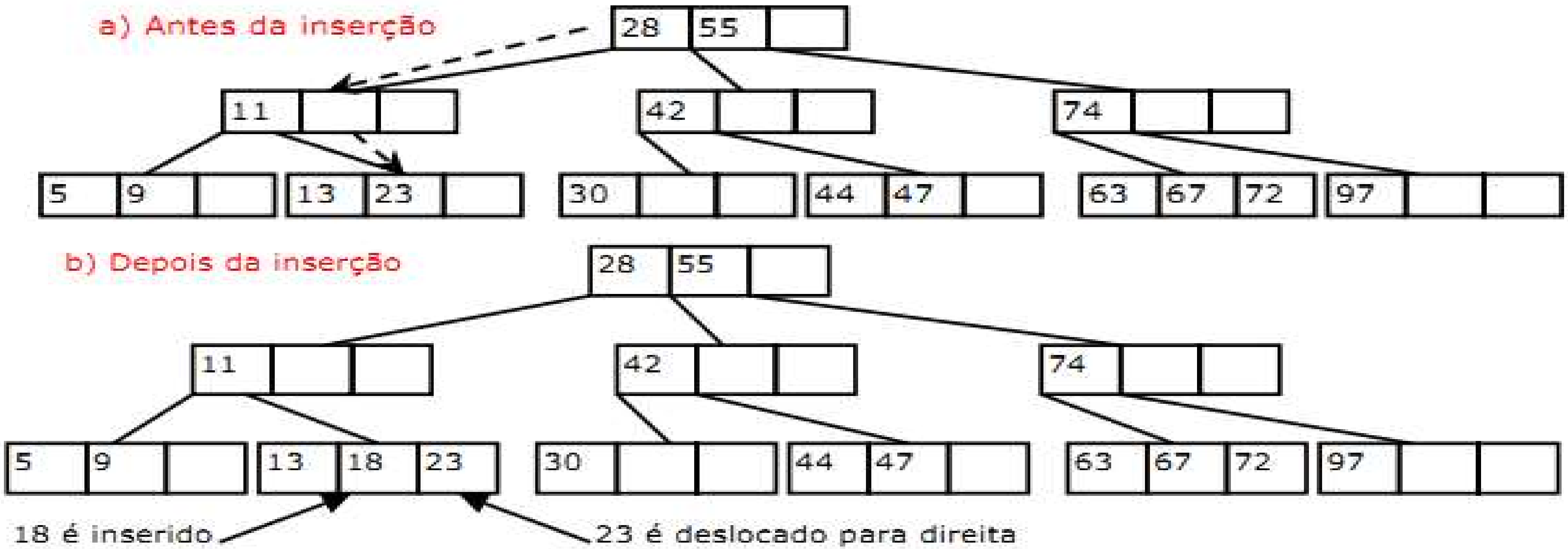
Se apenas nós que não estão cheios são encontrados durante a busca, a inserção é mais fácil.

Árvore 2-3-4

A inserção pode envolver a movimentação de um ou dois itens de dados em um nó.

As chaves deverão estar na ordem correta após o novo item ser inserido.

Exemplo da inserção item 18 na árvore abaixo:



Árvore 2-3- 4

As inserções tornam-se mais complexas se um nó cheio é encontrado no caminho abaixo do ponto de inserção.

Quando isso ocorre o nó precisa ser dividido.

É este processo de divisão que mantém a árvore balanceada.

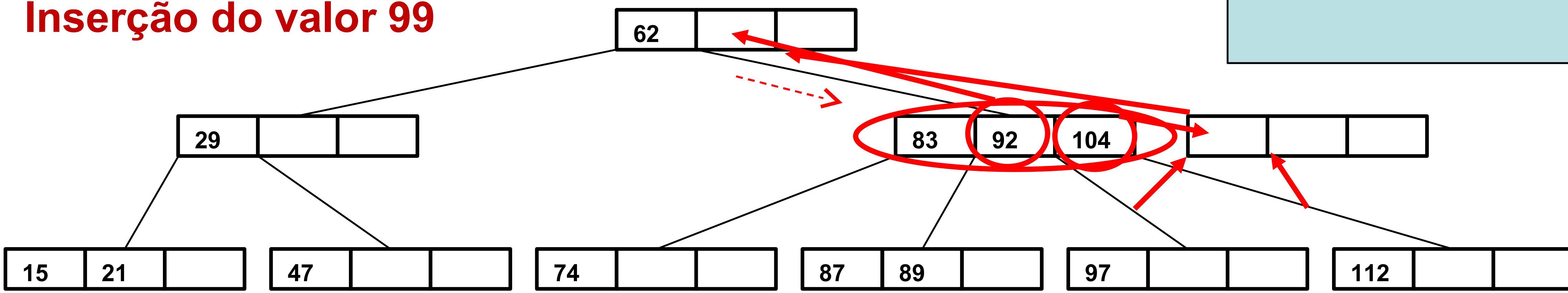
O tipo de árvore 2-3-4 que estamos estudando é frequentemente chamado de árvore 2-3-4 top-down, porque os nós são divididos “para baixo” do ponto de inserção.

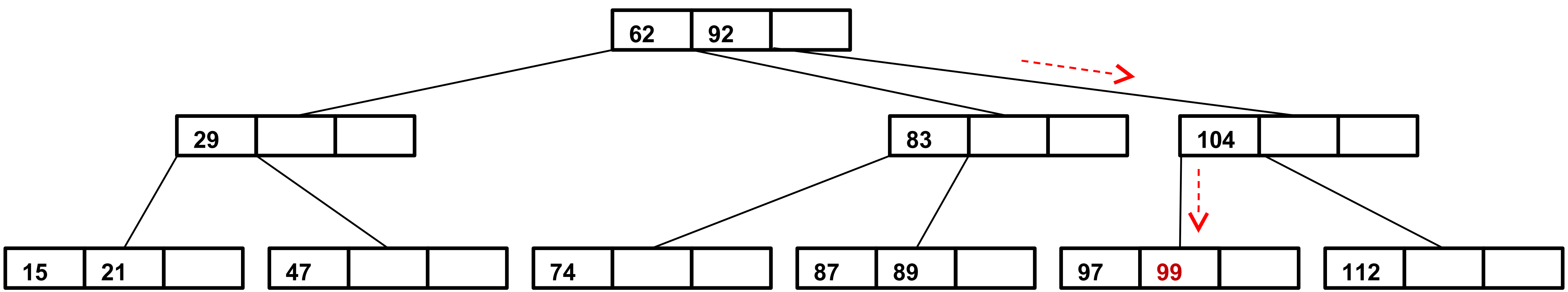
Chamaremos os itens de dados a serem divididos de A, B e C.

Assumiremos que o nó a ser dividido não é a raiz, examinaremos a divisão do nó raiz depois.

Árvore 2-3-4

Inserção do valor 99





Árvore 2-3-4

Quando uma raiz cheia é encontrada no início da busca para encontrar o ponto de inserção, o processo de inserção é ligeiramente mais complicado:

Um novo nó é criado, tornando-se a nova raiz, e a antiga raiz é dividida criando um novo nó irmão;

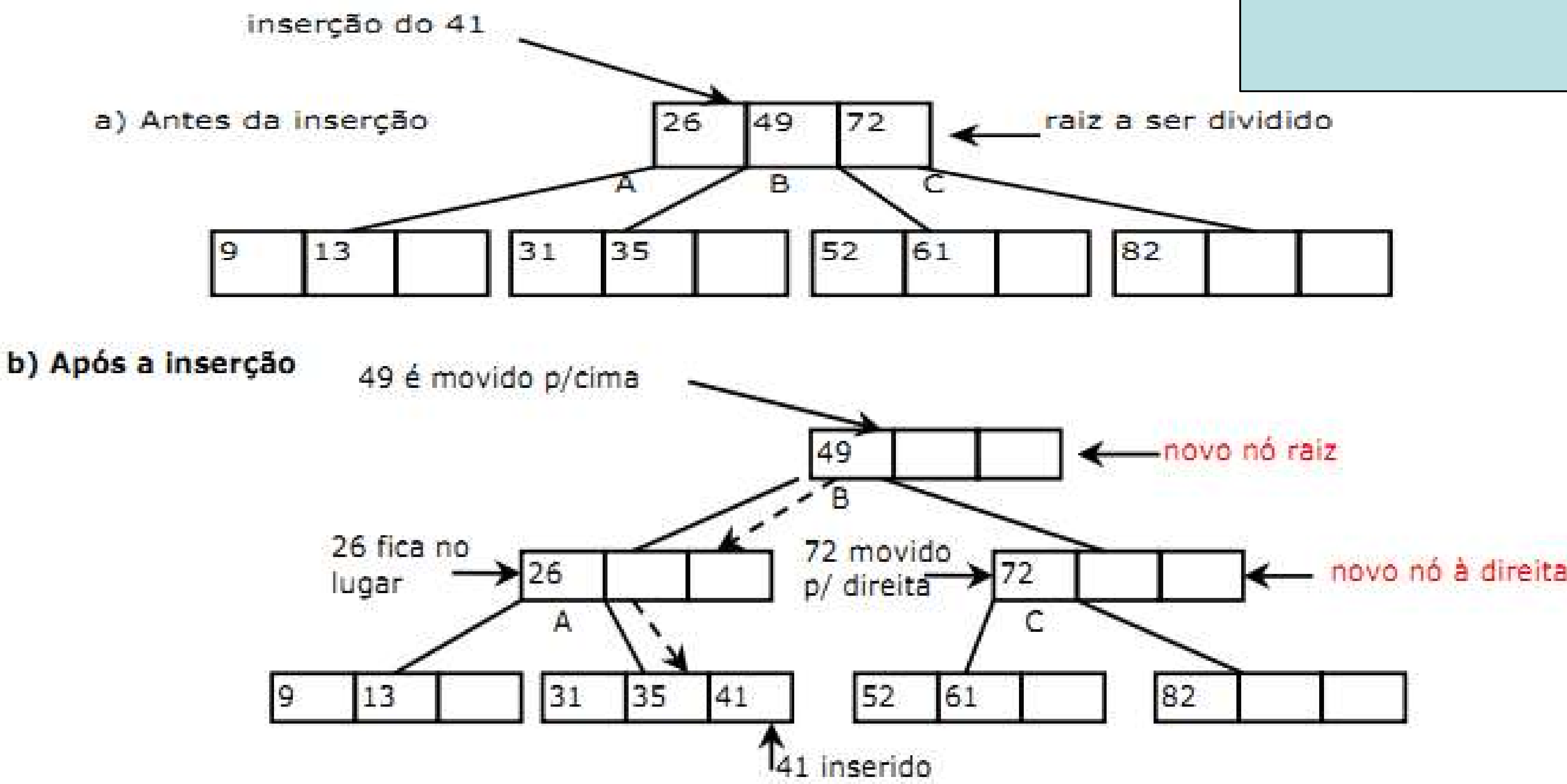
O item de dado C é movido para o novo nó irmão da antiga raiz;

O item de dado B é movido para a nova raiz;

O item de dados A é deixado onde está;

Os dois filhos mais a direita do nó que está sendo dividido são desconectados dele e conectados no novo nó do lado direito.

Árvore 2-3- 4


inserção do 41
a) Antes da inserção
26
49
72
raiz a ser dividido
A
B
C
9
13
31
35
52
61
82
b) Após a inserção
49 é movido p/cima
49
novo nó raiz
B
26 fica no lugar
26
72 movido p/ direita
72
novo nó à direita
A
C
9
13
31
35
41
52
61
82
41 inserido

FIM...