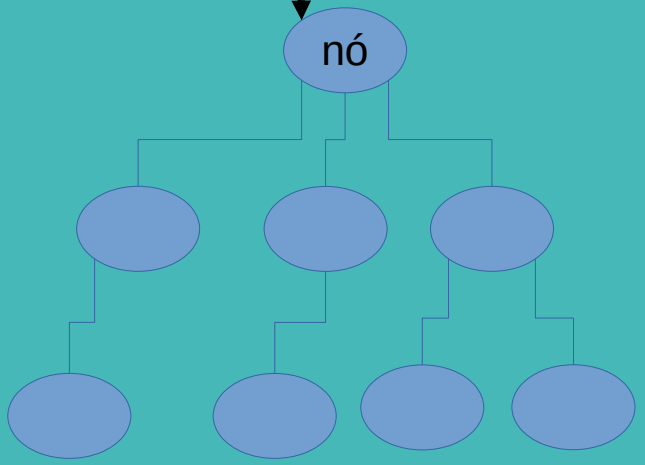
Árvores

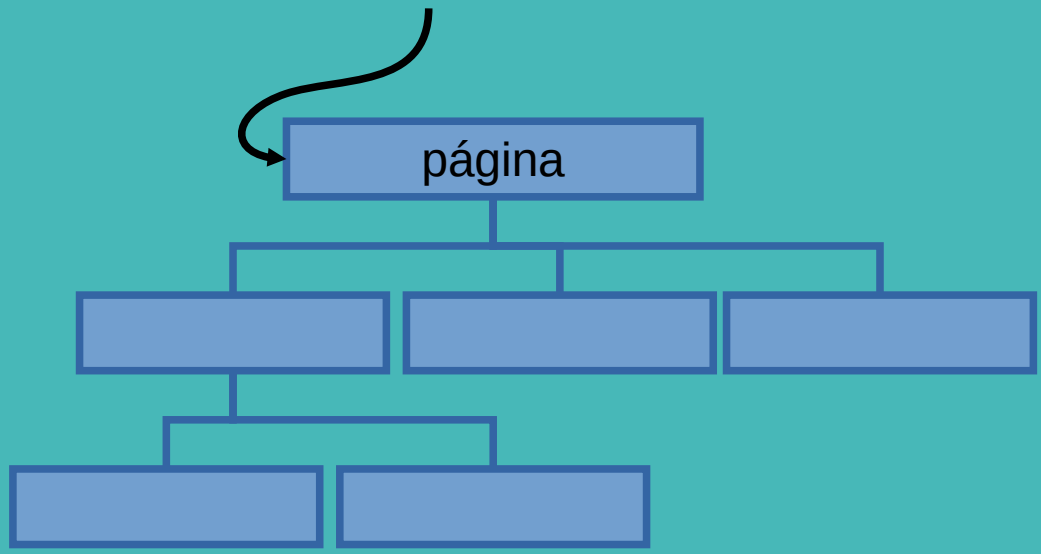
Alguns Conceitos Gerais

Inicialmente falaremos sobre um tipo de árvore que se estrutura como uma relação entre cada nó e seu(s) filho(s)”.



Mas há também as chamadas árvores multivias (ou multidirecionais), onde as ligações hierárquicas ocorrem entre **conjuntos** de nós chamados “páginas”.

A relação é entre cada página e sua(s) página(s) filha(s)



Conceitos Gerais sobre Árvores

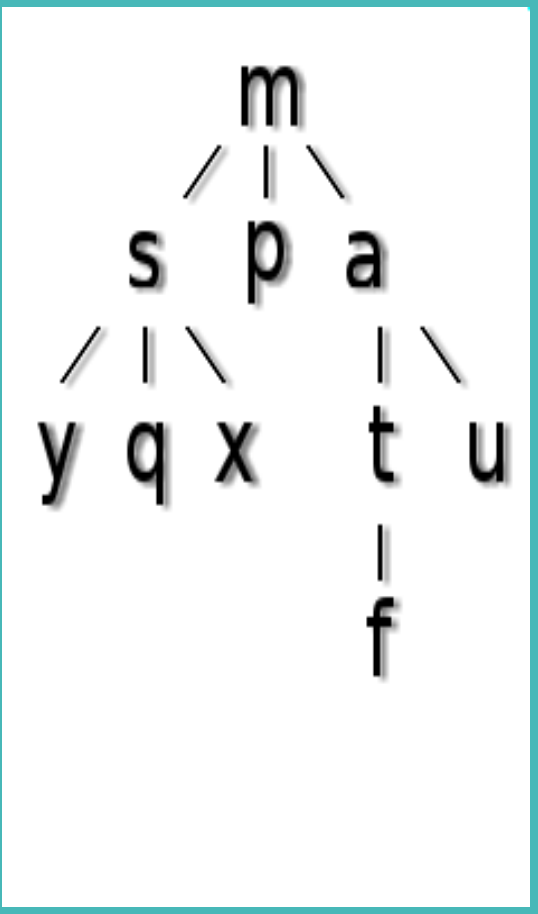
- Uma árvore é uma estrutura hierárquica dividida em níveis, que ou está vazia, ou contém elementos chamados *nós*;

- O nó-raiz é o ascendente comum de todos os nós da árvore, portanto encontra-se acima de todos os outros nós;

- Os nós-folhas são as extremidades finais da árvore. Nós-folhas não possuem descendentes;

- Diferentemente das pilhas, filas e listas, árvores se estendem em duas dimensões: na largura e na profundidade.

Conceitos Gerais sobre Árvores



Nó pai, nó filho, nós irmãos:

- o nó raiz (*m*) é pai de *s*, *p* e *a* (raízes de subárvores).
- *y*, *q* e *x* são exemplos de nós-irmãos, pois estão no mesmo nível e são filhos do mesmo pai.

Grau de um nó: quantidade de filhos que um determinado nó possui.

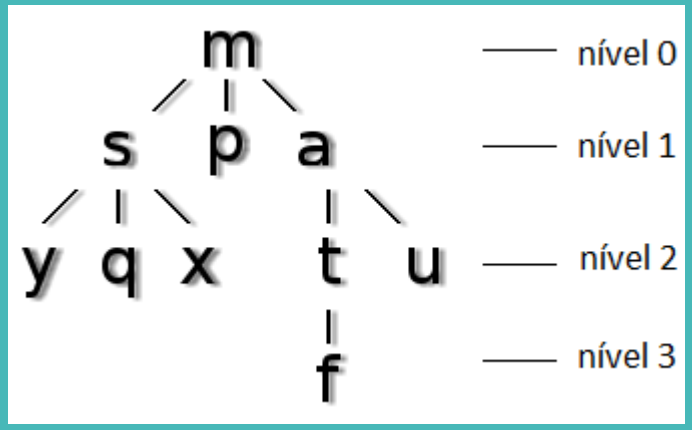
- As folhas têm grau zero.
- grau(m) = grau(s) = 3,
- grau(t) = 1,
- grau(a) = 2.

Grau de uma árvore: quantidade máxima de filhos que um nó desta árvore pode apresentar.

Conceitos Gerais sobre Árvores

Comprimento de caminho: contagem das arestas (há autores que contam os nós) atravessadas no percurso do caminho.
Ex: caminho(*m*,*u)* = {*m*,a,u} cujo comprimento é *L = 2*.

Nível de profundidade de um nó: corresponde ao comprimento de um caminho que começa obrigatoriamente na raiz e vai até o nó considerado. A profundidade de *u* é, portanto, igual a dois.

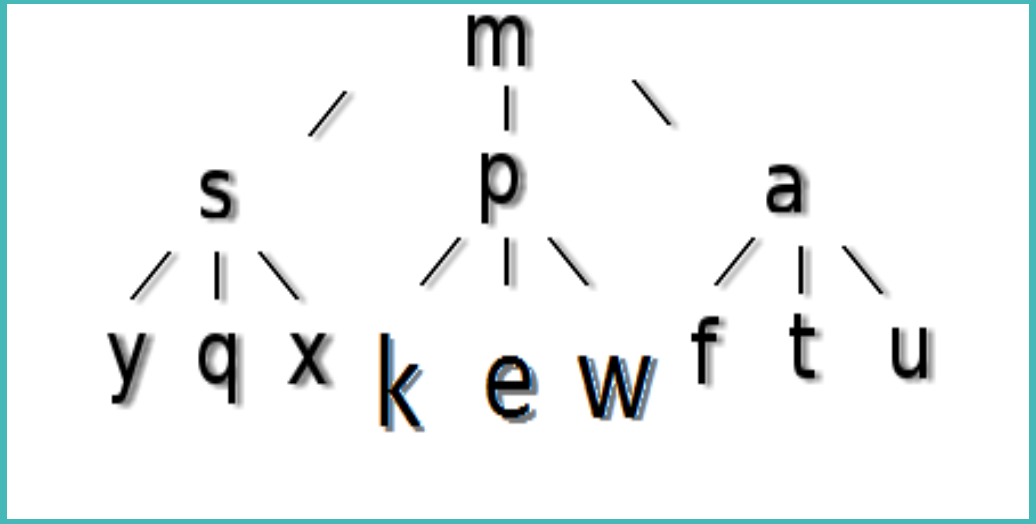


Altura de uma árvore: é o maior caminho entre a raiz e uma folha, em outras palavras, corresponde ao nível máximo de uma árvore.
Ex: A altura da árvore (na figura) é igual a três.

Conceitos Gerais sobre Árvores

Um árvore é dita completa quando todos os seus nós (exceto as folhas) apresentam o grau máximo *d*

Abaixo temos uma árvore completa de grau 3:



Conceitos Gerais sobre Árvores

Em geral teremos:

- A quantidade *m* de nós em uma árvore completa de altura *h* e grau *d* corresponde a soma dos *k* primeiros elementos de uma Progressão Geométrica:

$$(i)\sum_{i=0}^{h} d^i = d^0 + d^1 + \ldots + d^h \text{ corresponde à soma de uma PG: } (ii)S_{PG}^{k} = a_1\left(\frac{q^k - 1}{q - 1}\right)$$

De (i) e (ii) obtemos a expressão para o total de nós de uma árvore completa de ordem d e altura h:

$$(iii)\sum_{i=0}^{h} d^i = d^0\left(\frac{d^{(h+1)} - 1}{d - 1}\right) = \left(\frac{d^{(h+1)} - 1}{d - 1}\right)$$

www . brasilescola . com/ matematica/ soma– uma– pg – finita .htm

- A quantidade de nós folha dessa árvore completa será: W= d^h e a sua altura será igual a $h=log_d W$

  - É de onde se diz que a complexidade do pior caso de busca (por um elemento terminal em uma árvore completa) nesse tipo de árvore é de ordem logarítmica.

ÁRVORE BINÁRIA DE BUSCA

ABB

Árvores Binárias de Busca – ABB

ABB: árvore de segundo grau (nó pai terá no máximo dois filhos) projetada para otimizar operações de busca.

Tal otimização decorre da relação de ordem estabelecida entre os nós da ABB:

Considerando:

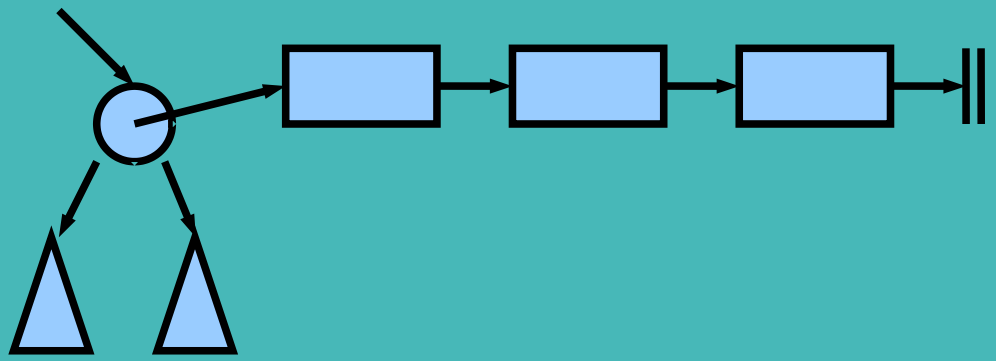
- *N* como um nó qualquer (não *folha*!);
- *chave(N)* retorna o valor do campo chave de ordenação do registro de dados associado ao nó *N*;
- N_{esq} e N_{dir} respectivamente: a subárvore esquerda e direita de *N*.

Teremos:

$$x \in N_{esq} \text{ e } y \in N_{dir} \Longrightarrow chave(x) < chave(N) < chave(y)$$

Árvores Binárias de Busca – ABB

Aqui se considera a chave como do tipo primária, no entanto, a inserção na ABB pode ser adaptada para lidar com uma chave do tipo secundária (identifica uma classe de instâncias de registros), nesse caso, cada nó da ABB poderia referenciar uma lista cujos itens apresentariam o mesmo valor de chave secundária.

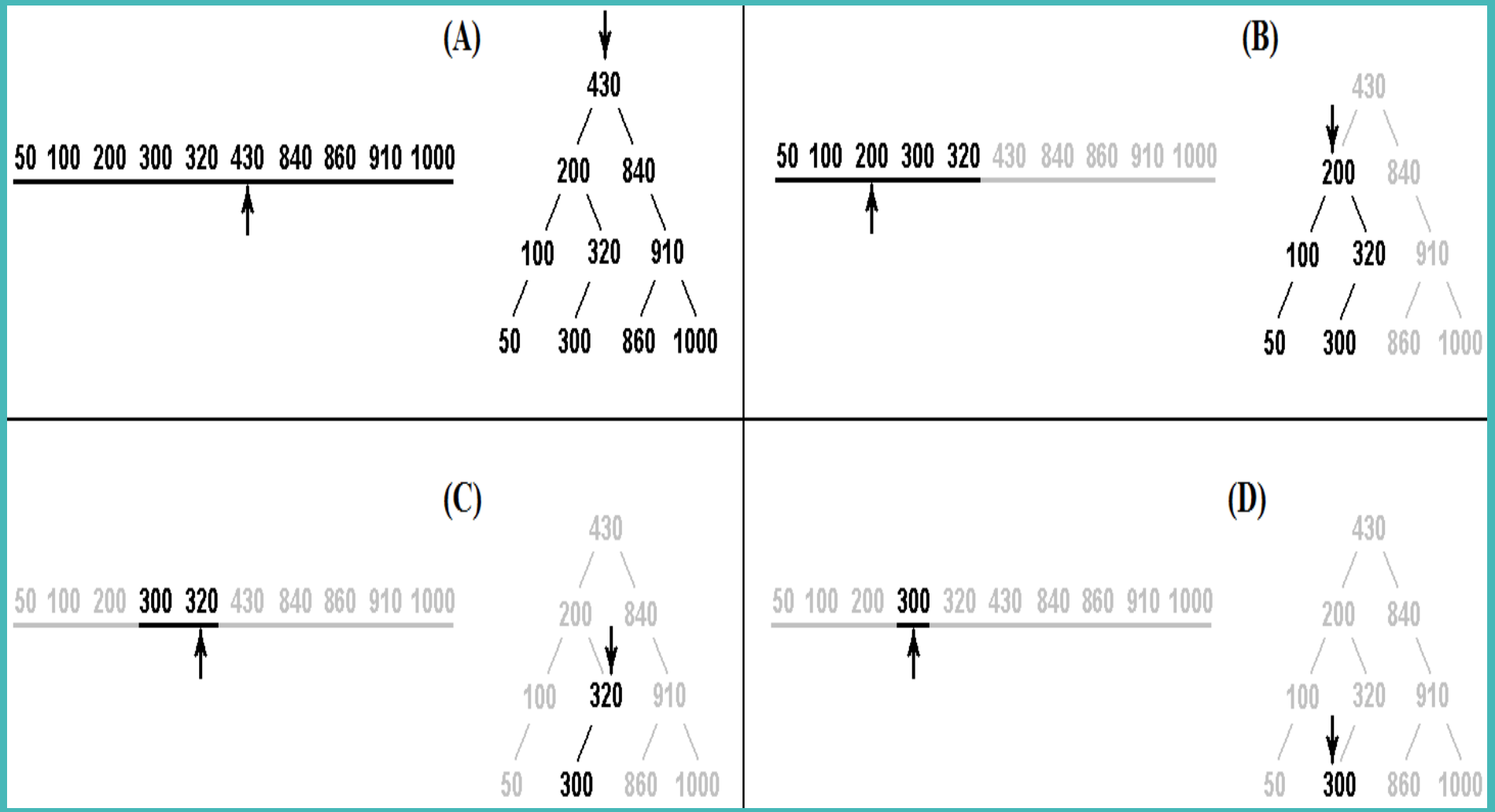


Árv. Bin. de Busca
X
Busca Binária

V é um vetor com os dados.
Inicio: indexa o início de V
Fim: indexa a última posição ocupada em V

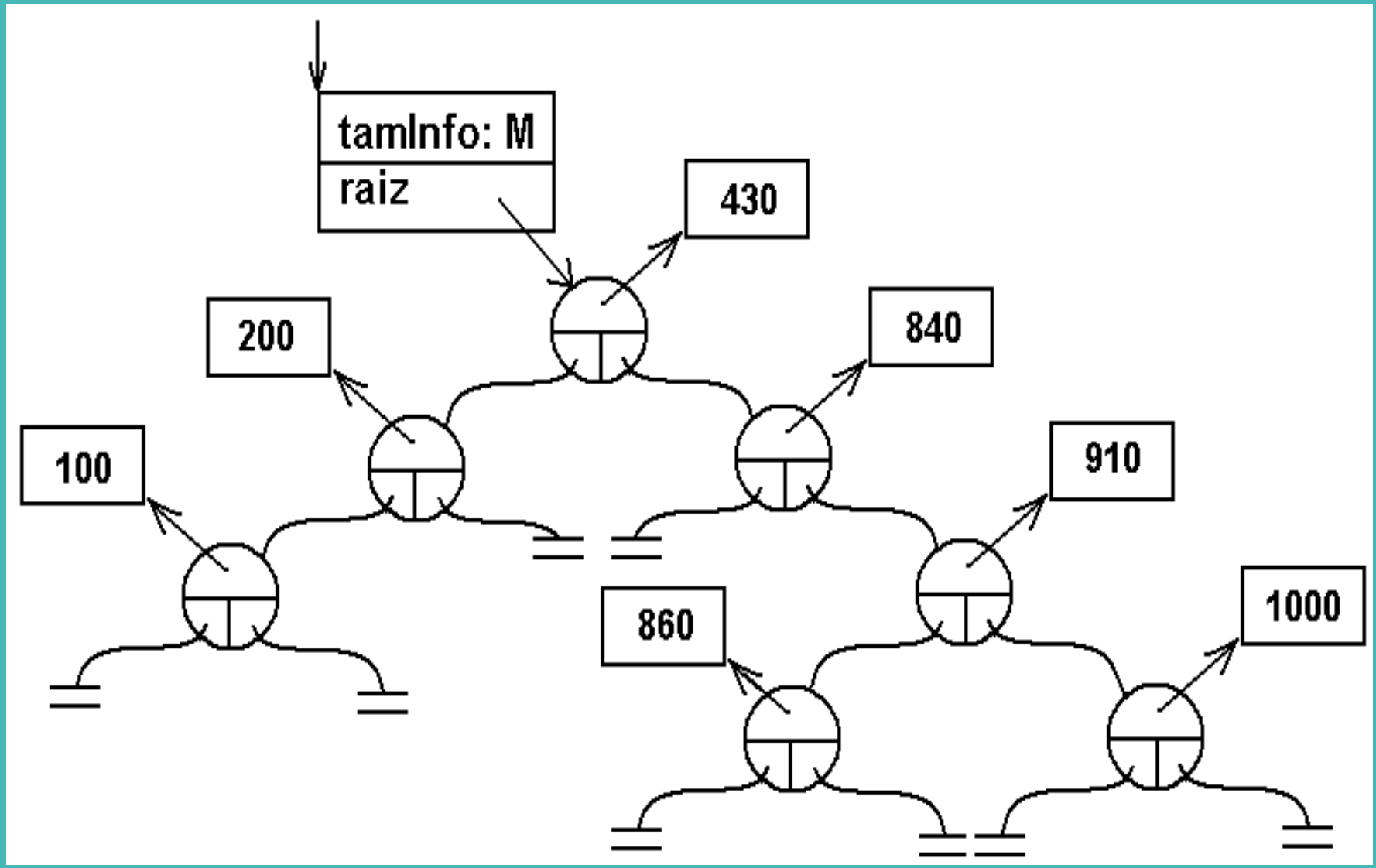
```
BUSCA-BINÁRIA (V[], início, fim, chaveDeBusca)
    i = (fim+inicio)/2   /* índice do meio entre início e fim */
    se (v[i] = chaveDeBusca) entao
            devolva o índice i   /* elemento e encontrado */
    senão (inicio = fim) entao
             não encontrou o elemento procurado
    senão
        se (V[i] vem antes de chaveDeBusca) então
             faça a BUSCA-BINÁRIA(V, i+1, fim, chaveDeBusca)
        senão faça a BUSCA-BINÁRIA(V, inicio, i-1, chaveDeBusca)
      fimse
    fimse
```

Árv. Bin. de Busca *Versus* a Busca Binária



Veremos o código da busca na ABB mais adiante!!!

ABB implementada dinamicamente

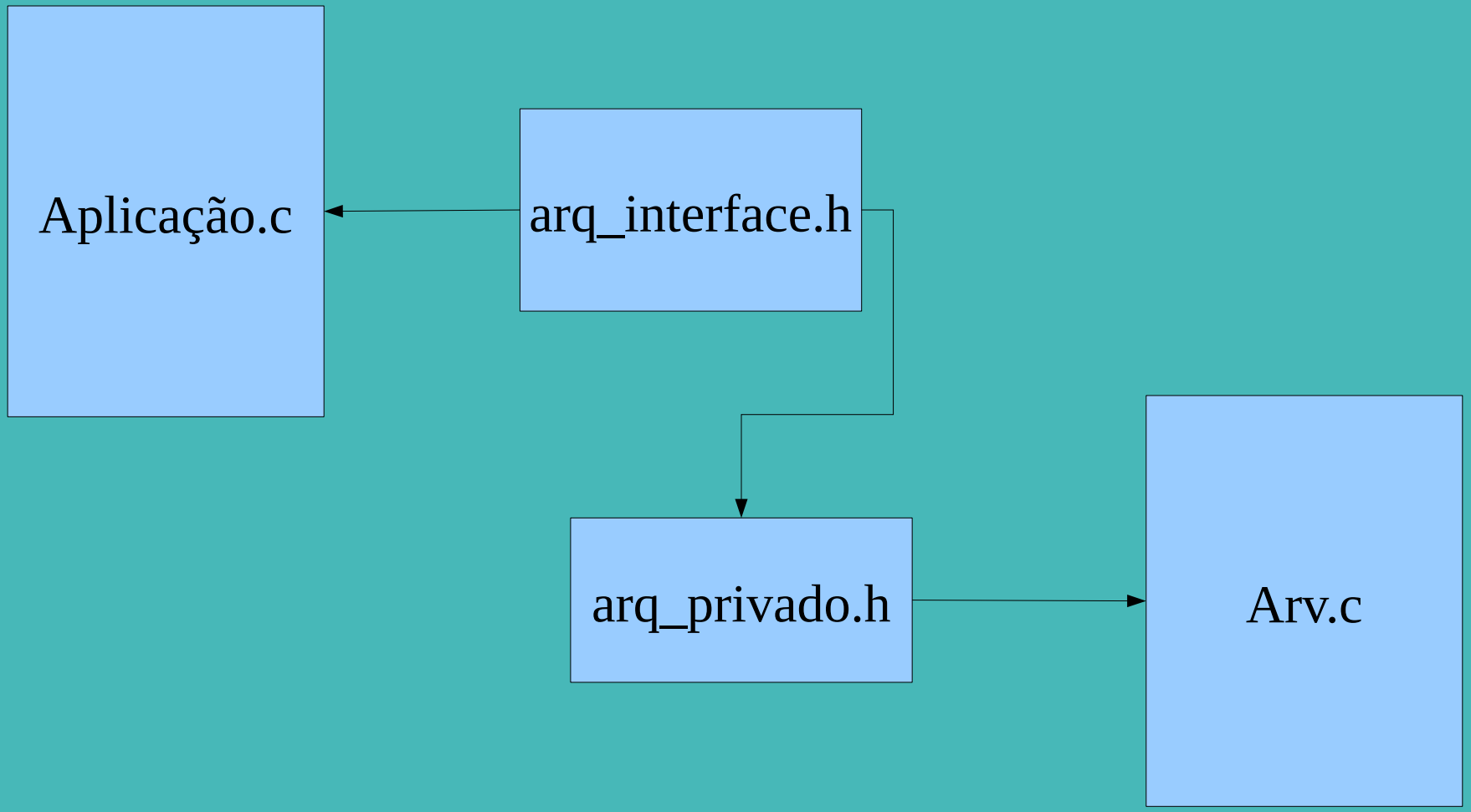


Utilização de Funções como Argumento - *Callback*

A ABB precisa comparar os valor da chave do registro para determinar a posição de um nó durante uma inserção ou a identificação de um nó em uma busca ou remoção.

Para a ABB será utilizada a mesma solução adotada para a Fila de Prioridade.

Implementação da ABB



Arquivo TDA_ABB.C

A Busca

Ao introduzir a ABB utilizou-se a operação de busca para descrever o comportamento desta árvore, nesse momento cabe apenas exibir a codificação dessa operação.

```
int buscaABB(ABB *pa, info *destino, int chave)
{  NoABB *aux;
   unsigned int ret= FRACASSO;
   aux = pa->raiz;
   while( aux != NULL && chave != chave(aux))
         aux = (chave < chave(aux) ? aux->esq : aux->dir);
   if (aux != NULL)
    {   memcpy(destino, &(aux->dados), pa->tamInfo);
        ret = SUCESSO;
    }
   return ret;
}
```

A inserção

A inserção bem sucedida em uma ABB (ordenada por chave primária) consiste basicamente em procurar pelo item a ser inserido:

1) Se o item for encontrado, implicará em falha na inserção.
2) Se o item não for encontrado, a posição de inserção terá sido determinada;
3) Para 1 e 2: adapte o algoritmo de busca e utilize dois ponteiros auxiliares: *auxPai* sempre o pai de *aux*. Na saída do laço de busca:

a) Se *aux* for diferente de null, então a inserção falhará;
b) Caso *aux* seja null, então *auxPai* estará referenciando o pai do novo nó, basta alocar memória e codificar a inserção;

Lembre-se: cada nova inserção deve manter a relação de ordem entre os nós:

- Para qualquer nó *N* que <u>não seja *folha*</u>, vale a expressão abaixo:

$$x \in N_{esq} \text{ e } y \in N_{dir} \Longrightarrow chave(x) < chave(N) < chave(y)$$

A remoção

A remoção também deve manter a relação de ordem que caracteriza a ABB.

Uma operação de remoção bem sucedida, consiste de duas etapas:

i) A busca bem sucedida pelo nó alvo da operação;

ii) Remoção propriamente dita do *nó alvo*.

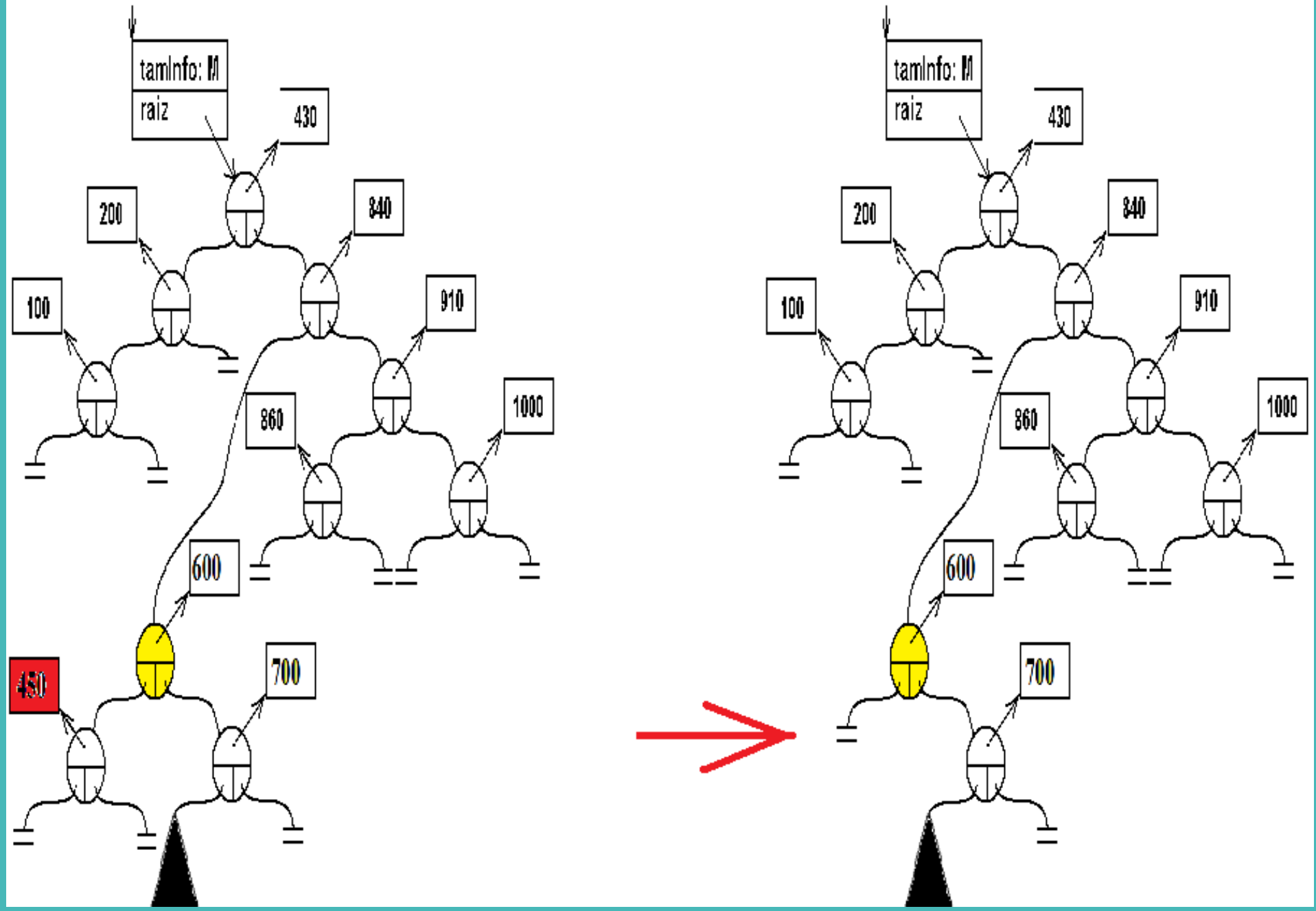
Na fase ii, o algoritmo de remoção deve tratar três casos possíveis:

a) Remoção de um nó folha;

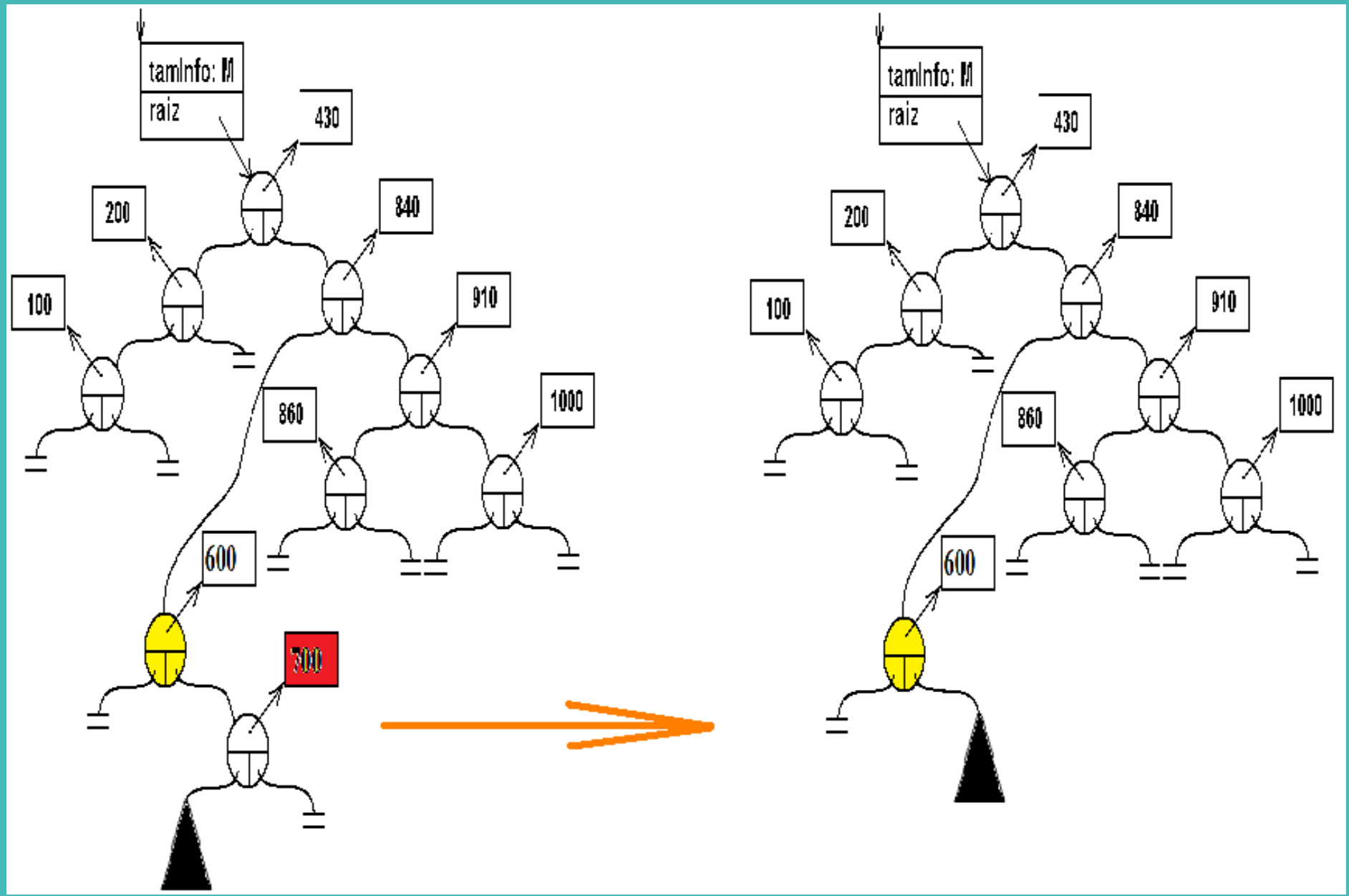
b) Remoção de um nó que possui apenas um filho e;

c) Remoção de um nó que possui seus dois filhos.

A remoção: a remoção de nó folha é trivial.



A remoção: a remoção de nó com um único filho também é trivial.



A remoção

Dos três casos acima citados, os itens *a* e *b* são triviais.

O item *c*, por sua vez, exige a localização de um nó que substitua o nó removido, mantendo a mesma relação de ordem entre os elementos restantes na ABB.

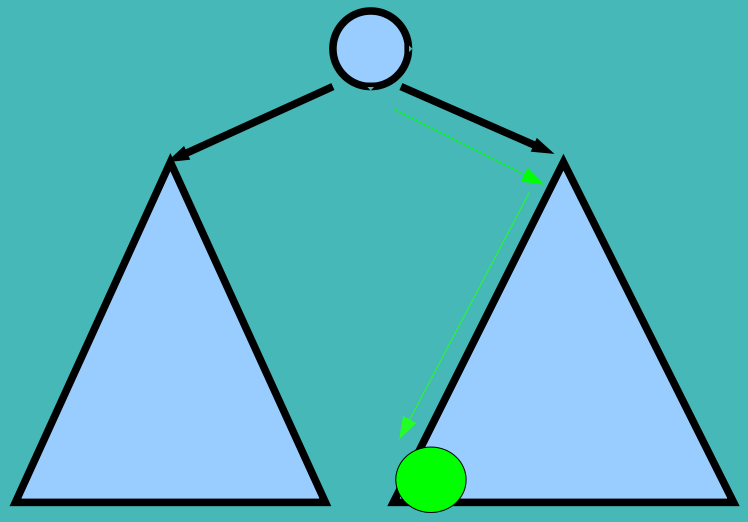
Para localizar este *substituto* deve-se buscar um nó cuja posição corresponda ao *sucessor* (ou antecessor) do *nó alvo* em um percurso *em ordem*.

O algoritmo de remoção implementa apenas uma das estratégias pelo nó sucessor ou pelo antecessor *em ordem*. Nunca ambas.

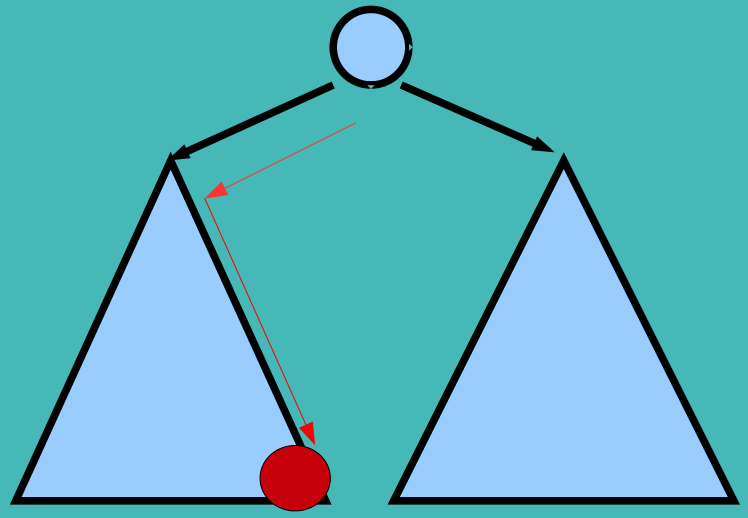
A remoção

Determinação do substituto:

A localização do nó *sucessor em ordem*: elemento sem filho esquerdo e localizado mais à esquerda na subárvore direita do nó removido.



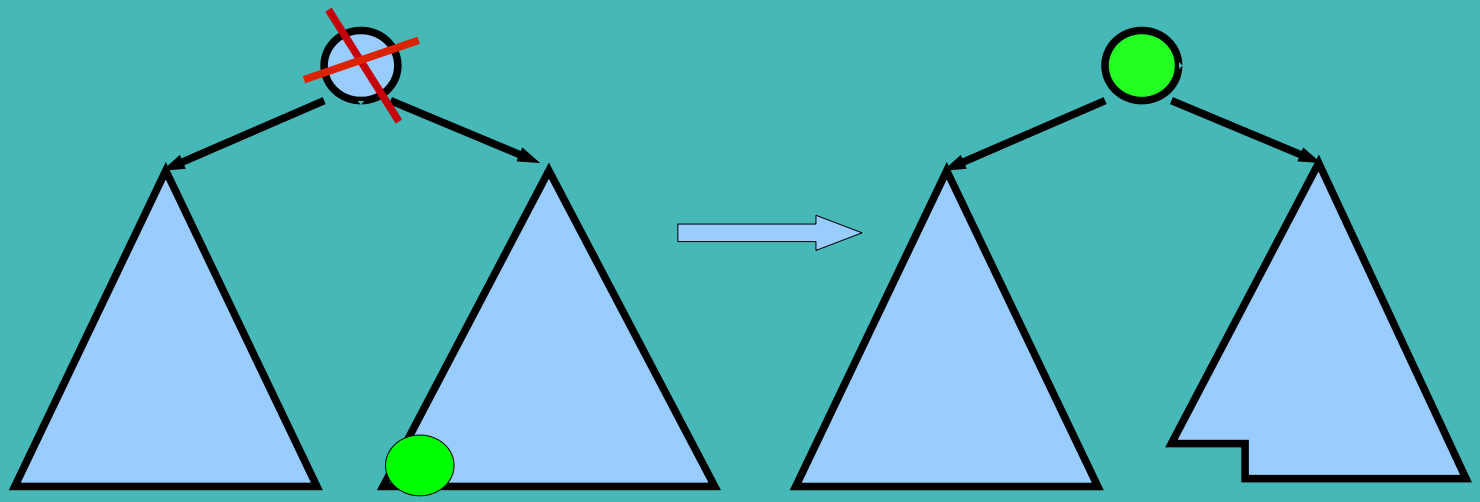
A localização do nó *antecessor em ordem*: elemento sem filho direito e localizado mais à direita na subárvore esquerda do nó removido.



A remoção

Localizado o *substituto*, a operação de remoção propriamente dita consistirá na substituição do *nó alvo* pelo seu substituto.

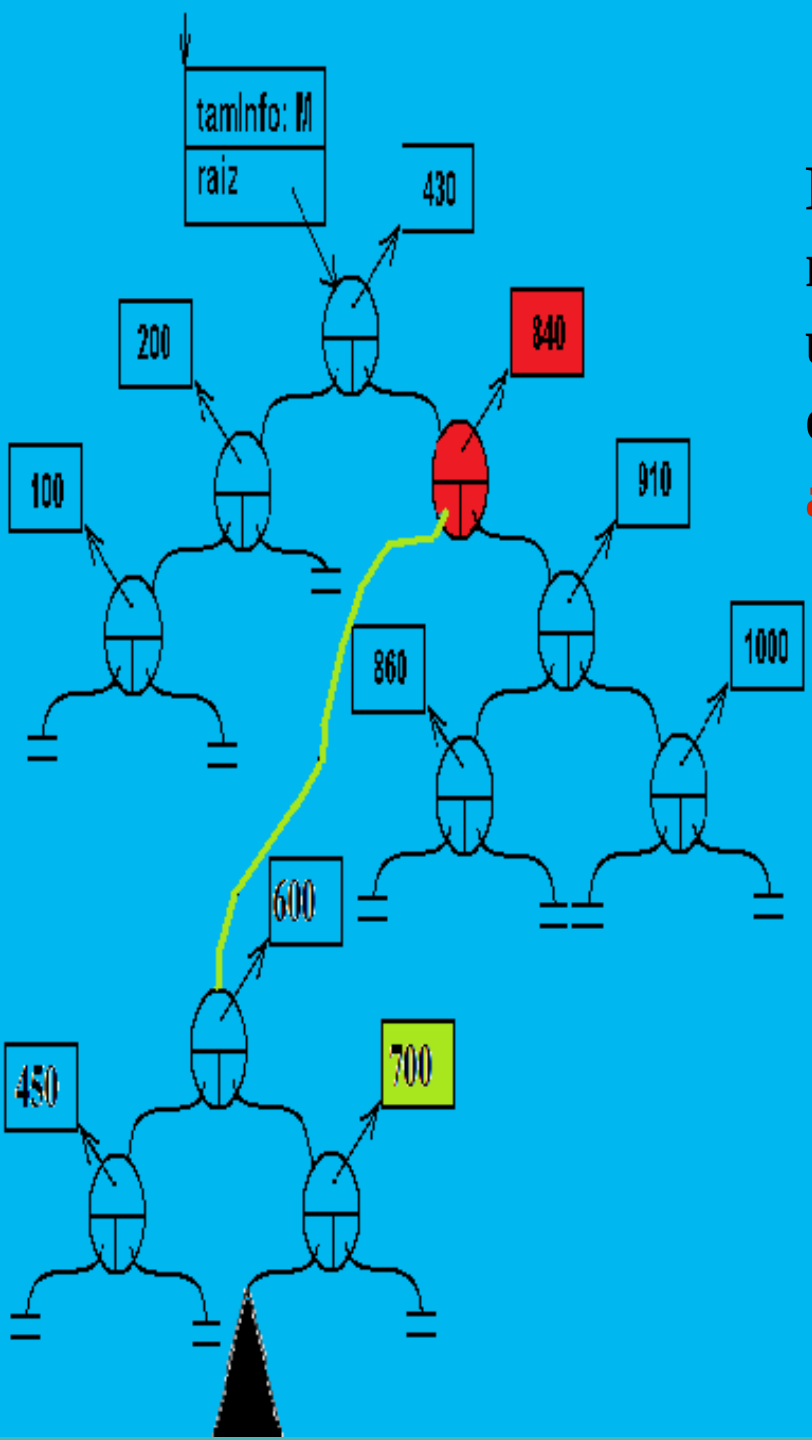
A substituição é realizada pela manipulação dos campos de ligação entre os nós. Abaixo, um exemplo considerando o substituto como o sucessor em ordem:

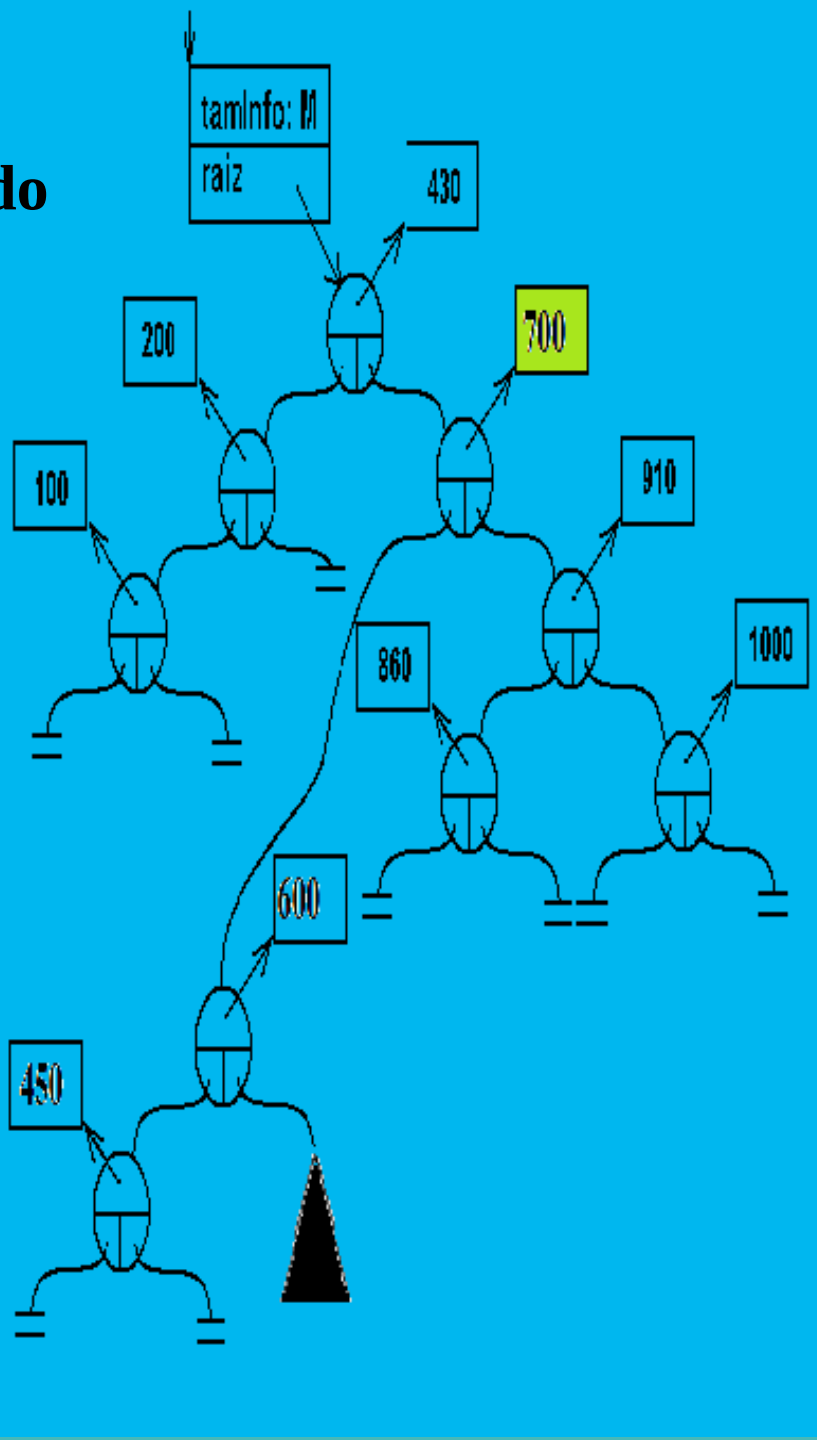


O algoritmo de remoção também trata dos casos onde o *nó alvo* é a própria raiz da árvore.


tamInfo: M
raiz
Nó alvo
430
200
840
100
910
1000
860
850
845
855
Substituto
Remoção
do nó 840
substituindo
pelo
Sucessor
em ordem
tamInfo: M
raiz
430
200
845
100
910
1000
860
850
855

Remoção do nó 840 utilizando o seu antecessor





```
int removeABB(ABB *pa, tipoChave chaveDeBusca, info *copia)
{   NoABB *subst, *paiSubst, *alvo, *paiDoAlvo, *avante;
    paiDoAlvo = NULL;
    alvo = pa->raiz;
    while (alvo != NULL && (chaveDeBusca != (alvo->dados.identificador))) {
        paiDoAlvo = alvo;
        if (chaveDeBusca < (alvo->dados.identificador))
                alvo = alvo->esq;
          else
                alvo= alvo->dir;
    }
    if (alvo==NULL)   /*alvo não encontrado */
        return FRACASSO;
    if (alvo->esq == NULL) {
        if (alvo->dir == NULL) // alvo é uma folha
            subst = NULL;
        else
            subst = alvo->dir; /*alvo possui apenas um filho à dir*/
    }
    else {
        if(alvo->dir == NULL)
            subst = alvo->esq; /*alvo possui apenas um filho à esq*/
        else  /* alvo possui os dois filhos*/
        {         /* determinando o nó-pai do nó-substituto e o nó-substituto (sucessor em ordem) */
        }
            if(paiSubst != alvo) {
                paiSubst->esq = subst->dir;
                subst->dir = alvo->dir;
            }
            subst->esq = alvo->esq; /*o pai do substituto eh o proprio alvo */
        }
    }
    if(pa->raiz == alvo) // ou seja se "paiDoAlvo = NULL"
        pa->raiz = subst; /*alvo era a raiz*/
    else
        alvo == paiDoAlvo->esq ? (paiDoAlvo->esq = subst) : (paiDoAlvo->dir = subst);
    memcpy(copia, &(alvo->dados), pa->tamInfo);
    free(alvo);
    return SUCESSO;
}
```

Remoção de um nó com os dois filhos utilizando o sucessor em ordem

Percursos em Árvores

Árvores podem ser percorridas de maneira que todos os seus nós sejam visitados e processados (por funções de *callback*) convenientemente.

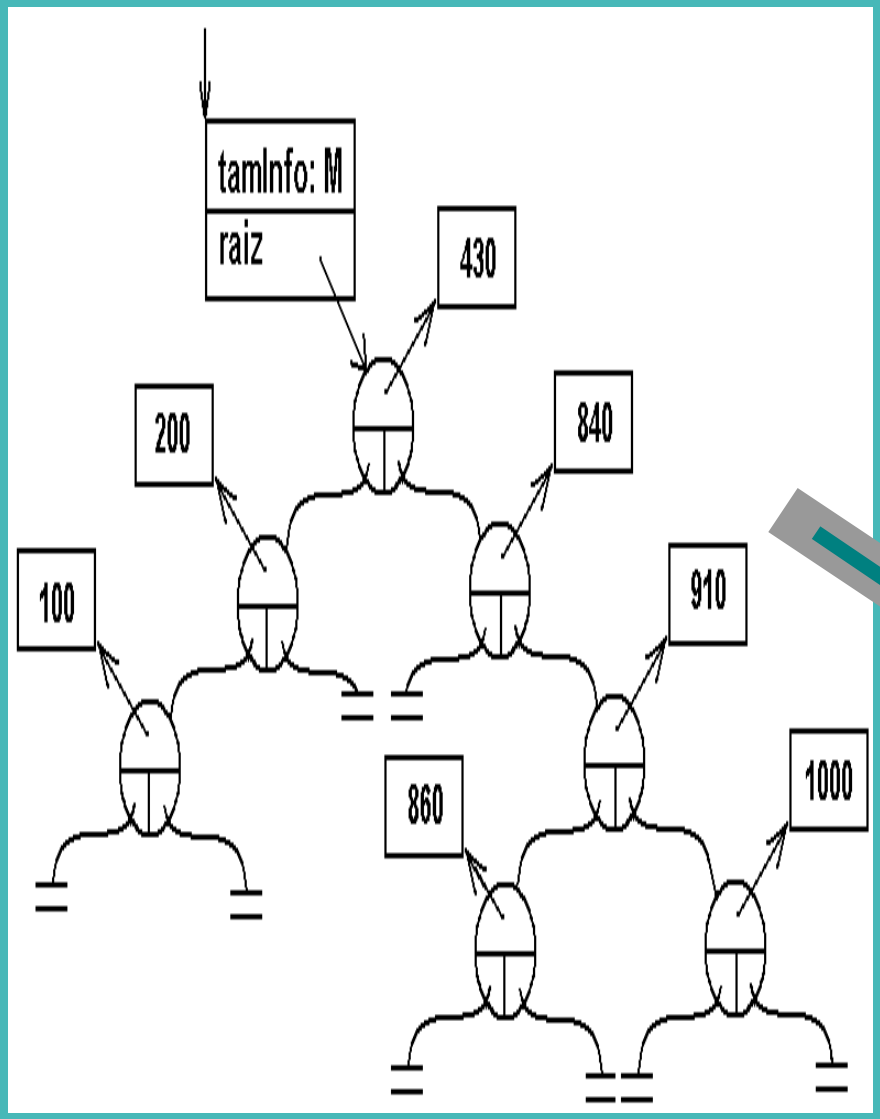
Os percursos podem ser em profundidade: “movimentos verticais”

OU...

Podem ser em largura: “movimentos horizontais”

Inicialmente veremos os percursos em profundidade...

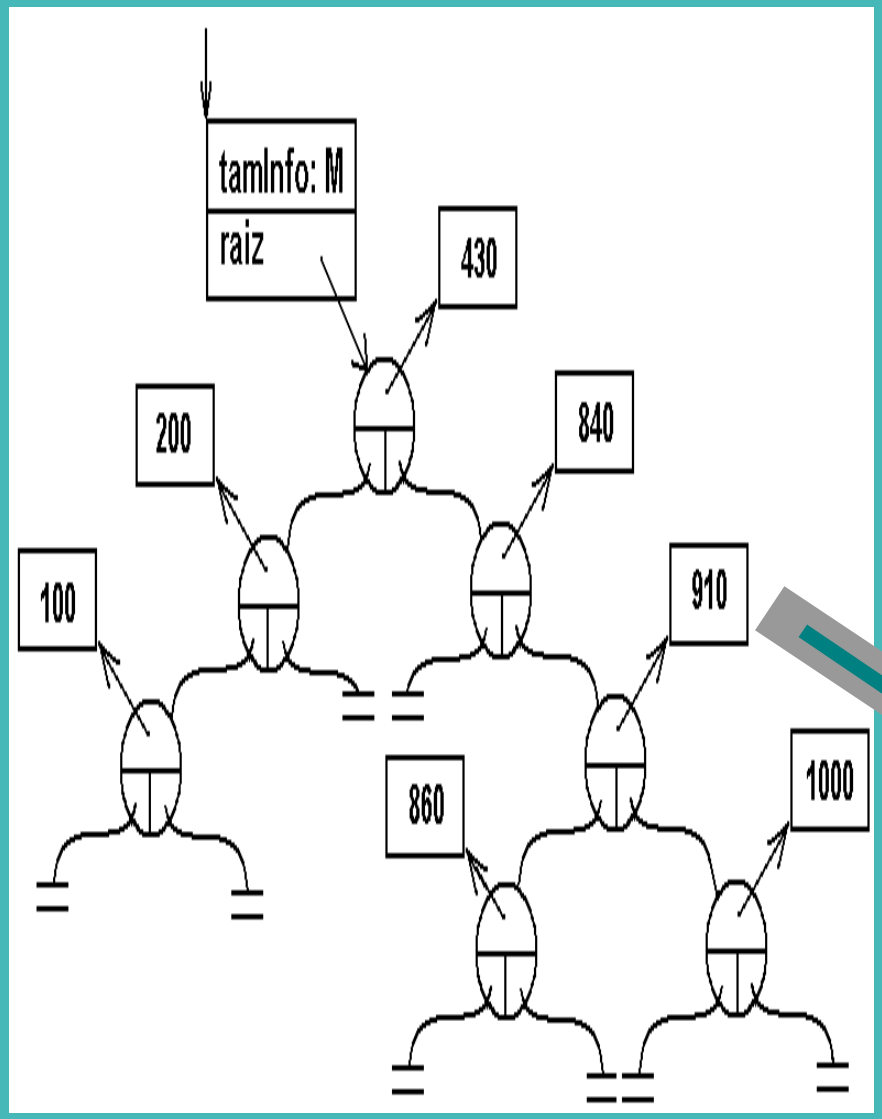
1) Profundidade: percurso em ordem




```
em_ordem(r, processa)
        se (r != Null)
                em_ordem(r->esq, processa);
                processa(r);
                em_ordem(r->dir, processa);
        senão
                return;
```

Exemplo considerando que a função *PROCESSA( )* exibe a chave do elemento armazenado, teremos a seqüência: 100,200,430,840,860,910,1000

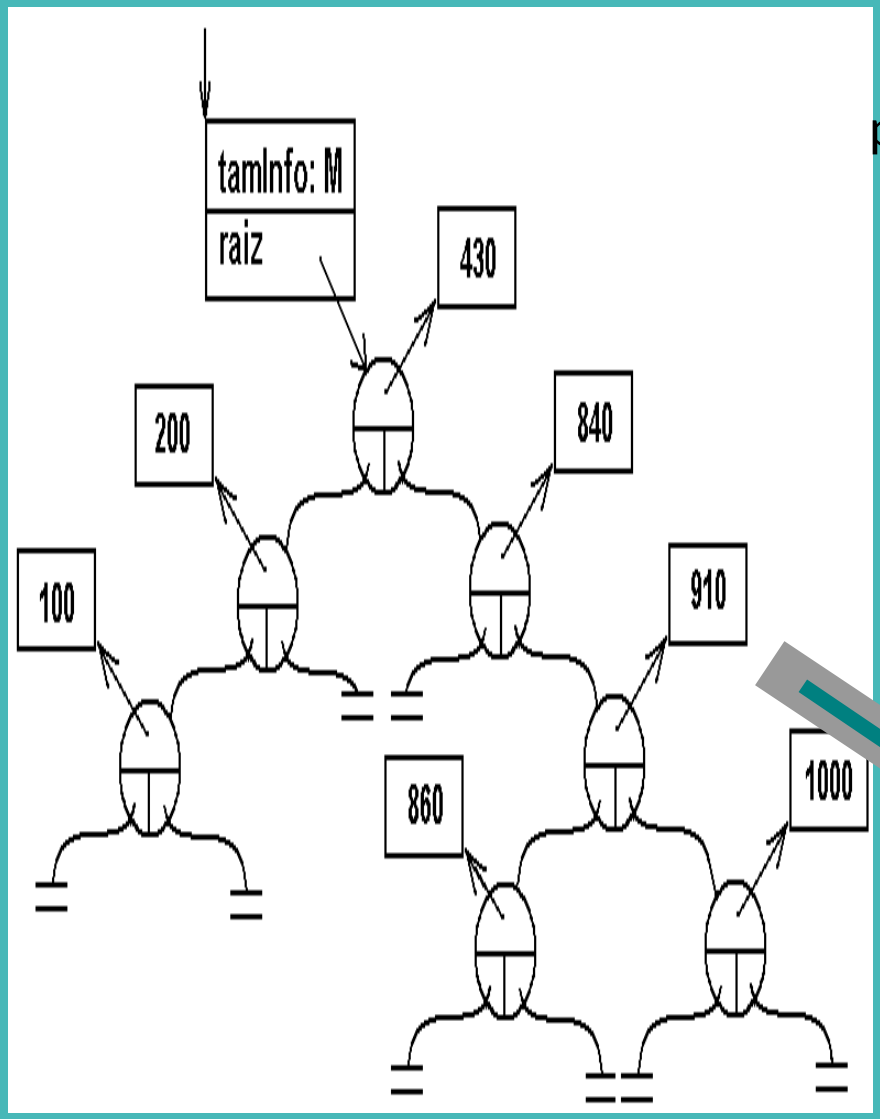
2) Profundidade: pós-ordem




```
pos_ordem(r, processa)
        se (r != Null)
                pos_ordem(r->esq, processa);
                pos_ordem(r->dir, processa);
                processa(r);
        senão
                return;
```

100,200,860,1000,910,840,430

3) Profundidade: pré-ordem




```
pre_ordem(r, processa)
        se (r != Null)
                processa(r);
                pre_ordem(r->esq, processa);
                pre_ordem(r->dir, processa);
        senão
                return;
```

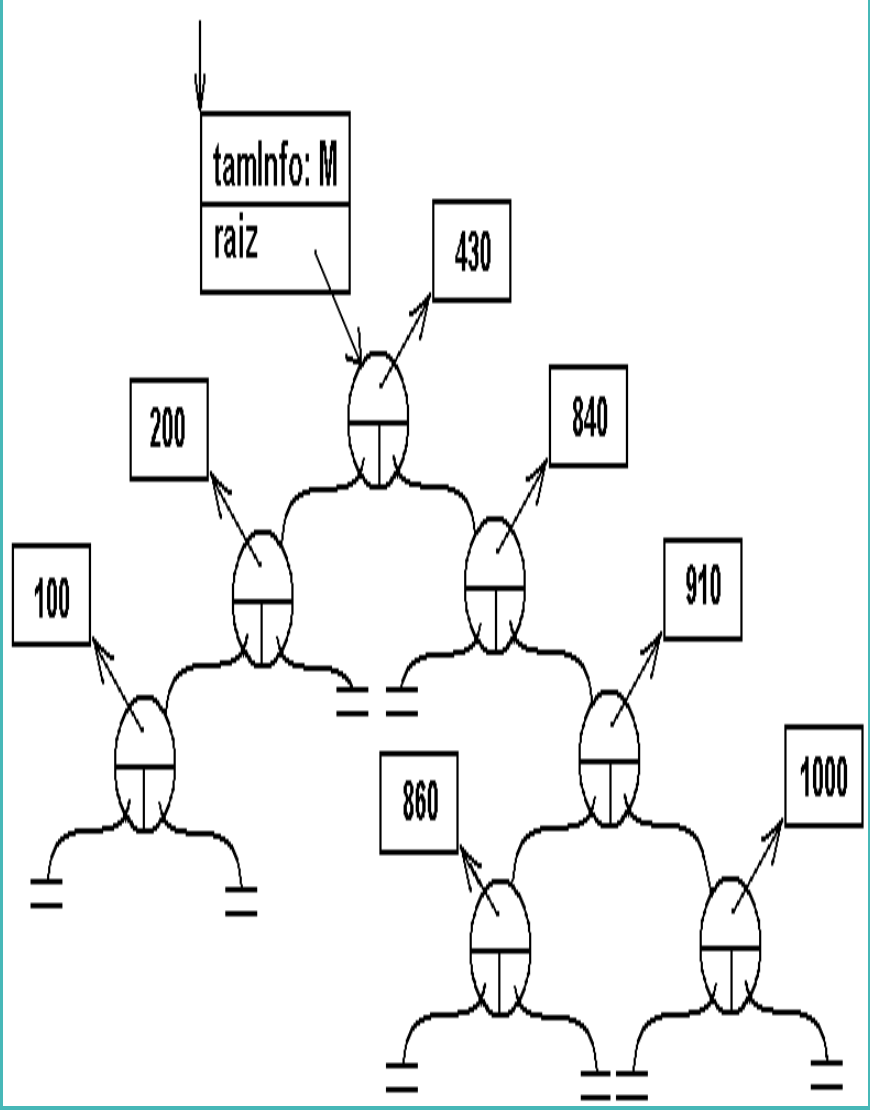
430,200,100,840,910,860,1000

Percurso em largura

Diferentemente dos percursos em profundidade, agora a visitação segue os níveis da árvore.

```
PercursoEmLargura(arvore, processa)
    fila = Criafila(...);
    SE (arvore não está vazia E fila foi criada com sucesso)
        enfileira(raiz);
        ENQUANTO(fila não vazia)
            desenfileira(noh);
            processa(noh);
            enfileira(filhoEsq(noh));
            enfileira(filhoDir(noh))
    destroiFila(...);
```

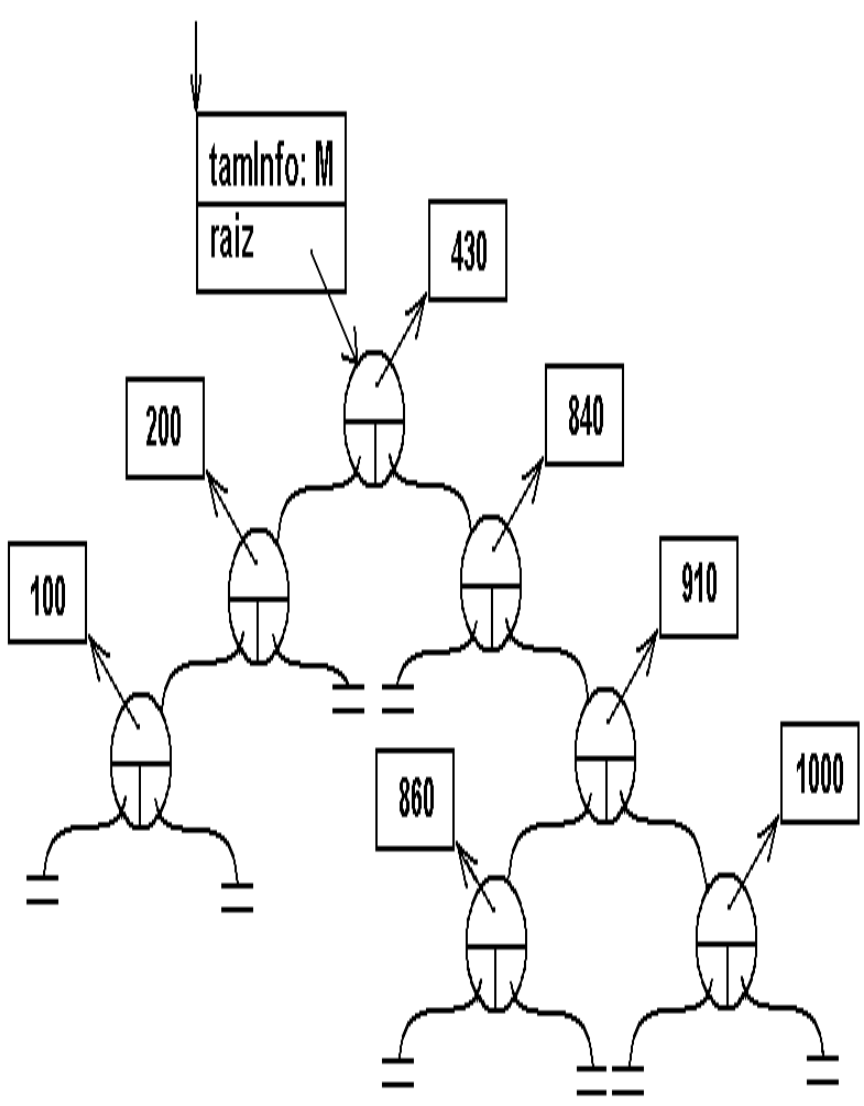
Percurso em largura



A busca em largura é orientada pelos níveis da árvore e acessa os nós na sua ordem de entrada (para uma ABB simples sem operações de balanceamento).

430,200,840,100,910,860,1000

Percursos



Porque não é interessante utilizar percursos para buscar dados na árvore?

Qual dos percursos é interessante para a operação de destruição da árvore?

Construa um TDA-ABB básico:

- Criação
- Remoção
- Inserção
- Busca
- Destruição
- Reinicialização
- Percursos
- Contagem de folhas
- Contagem de semi-folhas
- Contagem de nós completos
- Determinação da altura da árvore
- Exibição estrutural (gráfica) da árvore
- etc...

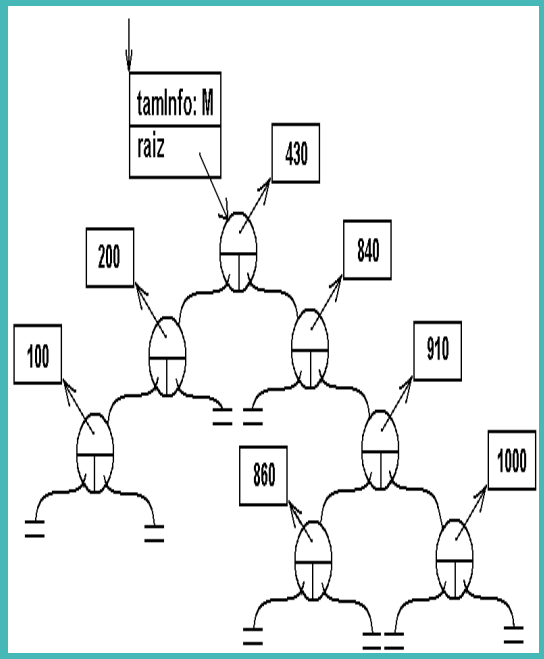


ABB implementada de maneira estática

Raiz: p->Vet[0]

Filho esquerdo de p->Vet[N] : p->Vet[2*N+1]

Filho direito de p->Vet[N] : p->Vet[2*N+2]

Pai(N) = (N-1) div 2 "div" determina o quociente da divisão inteira

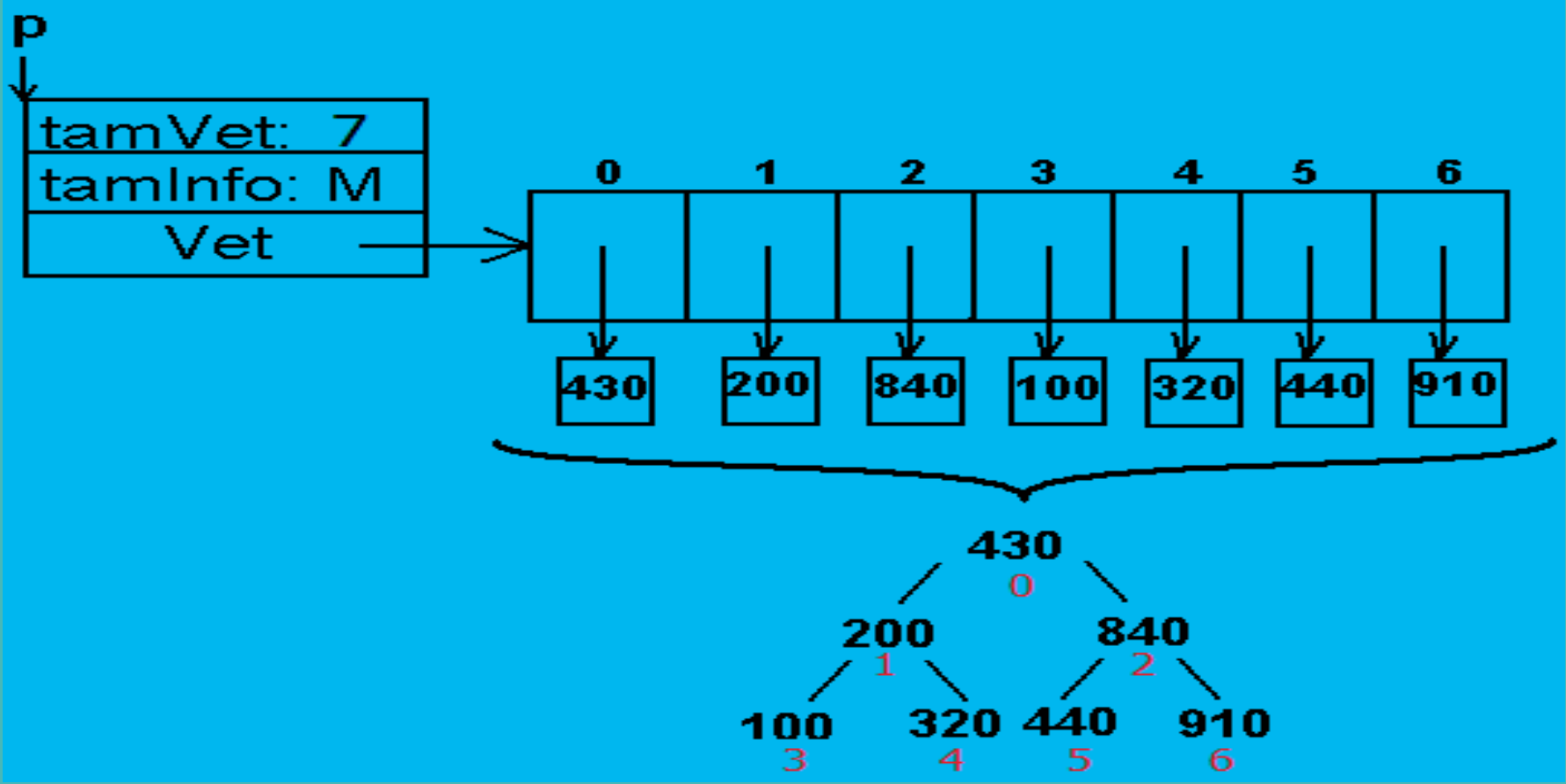
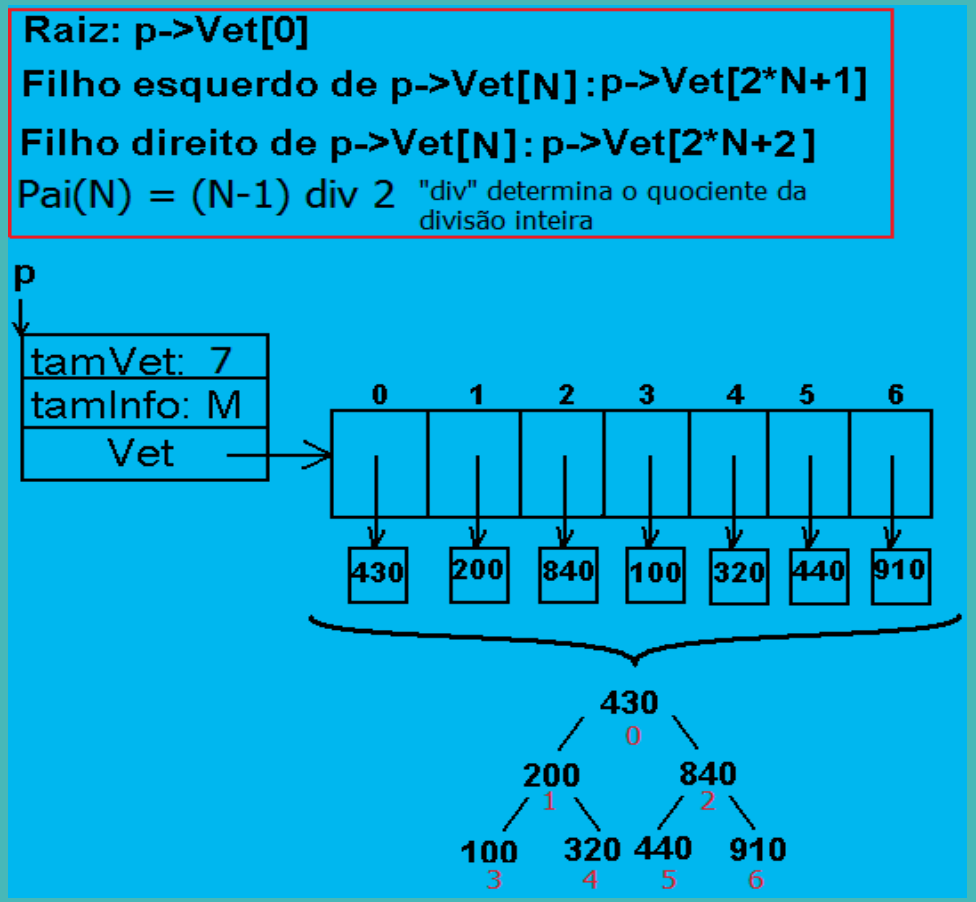


ABB implementada de maneira estática

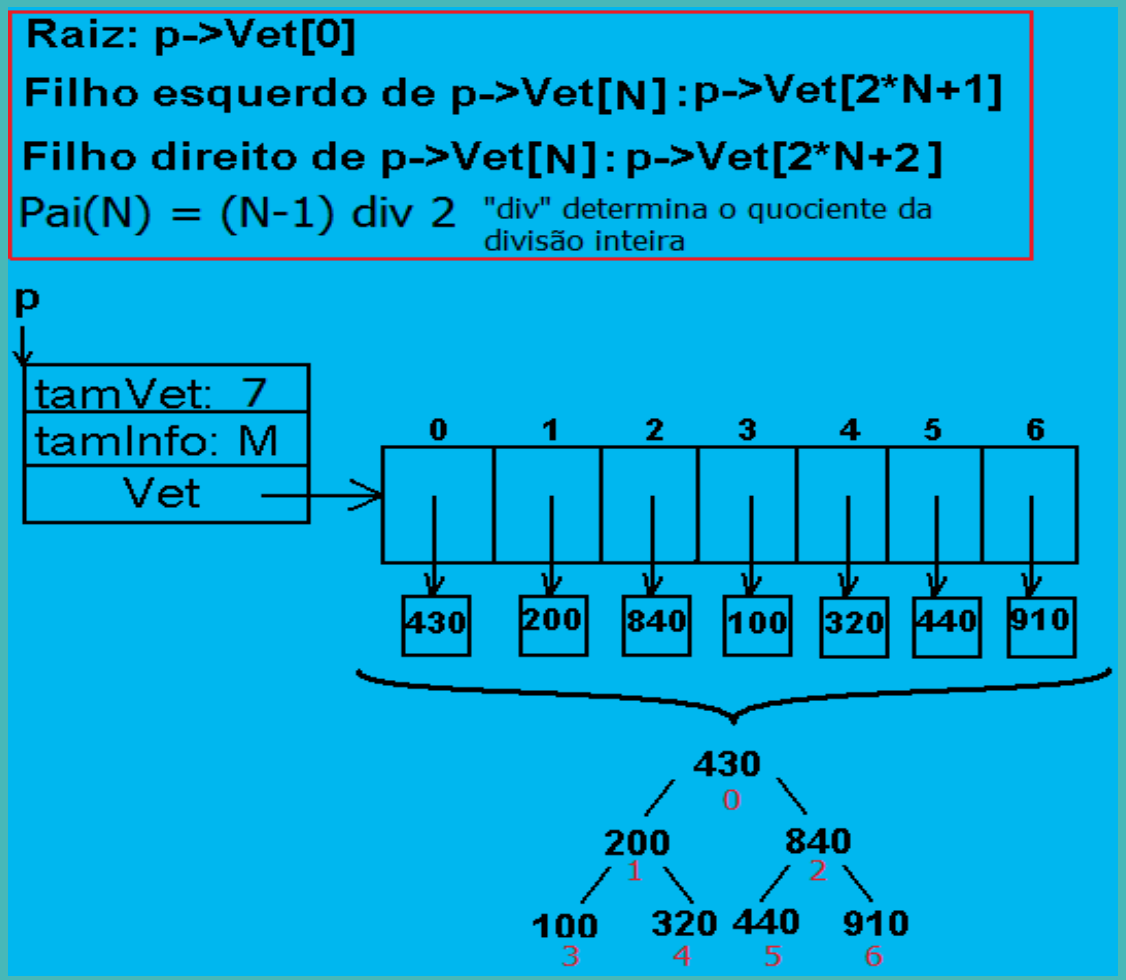
Para uma árvore binária completa, o tamanho do vetor pode ser calculado a partir do seu número de folhas:



TDA-ABB implementada de maneira estática

Construa um ABB estaticamente alocado:

1)Criação;
2)Remoção;
3)Inserção;
4)Busca;
5)Destruição;
6)Reinicialização;
7)Percursos
8)Altura;
9)Número de nós em determinado nível;
etc...



Arquivo arq.h

```
typedef struct{ tipoChave identificador;
                int idade;
                char nome[30];
        }info;

typedef struct noABB {
   info dados;
   struct noABB *esq;
   struct noABB *dir;
} NoABB;
typedef struct ABB {
   NoABB *raiz;
   int tamInfo;
} ABB;
//=====================ABB====================
void apagaNoABB(NoABB *p);
int calcNumFolhas(NoABB *p);
ABB *  criaABB(int tamInfo);
ABB *  destroiABB(ABB *p);
void reiniciaABB(ABB *p);
int buscaABB(ABB *pa, info *destino, int chaveDeBusca);
int insereABB(ABB *p, info *novoReg);
int removeABB(ABB *p, int chaveDeBusca, info *copia);
int testaVaziaABB(ABB *p);
int numFolhas(ABB *p);
int emOrdem(ABB *pa);
int posOrdem(ABB *pa);
int preOrdem(ABB *pa);
```