

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PIAUÍ
Campus Teresina - Central

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E
TECNOLOGIA DO PIAUÍ

CAMPUS TERESINA-CENTRAL

DIRETORIA DE ENSINO

Estrutura de Dados II – Árvores

Parte I

Professora: Elanne Cristina O. dos Santos

elannecristina.santos@gmail.com

elannecristina.santos@ifpi.edu.br

Árvore

É uma coleção de $n \geq 0$ nós ORGANIZADOS DE FORMA HIERÁRQUICA. Se $n=0$ então a árvore está vazia.

As listas ligadas podem fornecer maior flexibilidade que as matrizes, mas são ESTRUTURAS LINEARES, e é difícil usá-las para organizar uma representação hierárquica de objetos. São limitadas a somente uma dimensão.

- São representadas de cima para baixo, com a raiz no topo e as folhas na base.

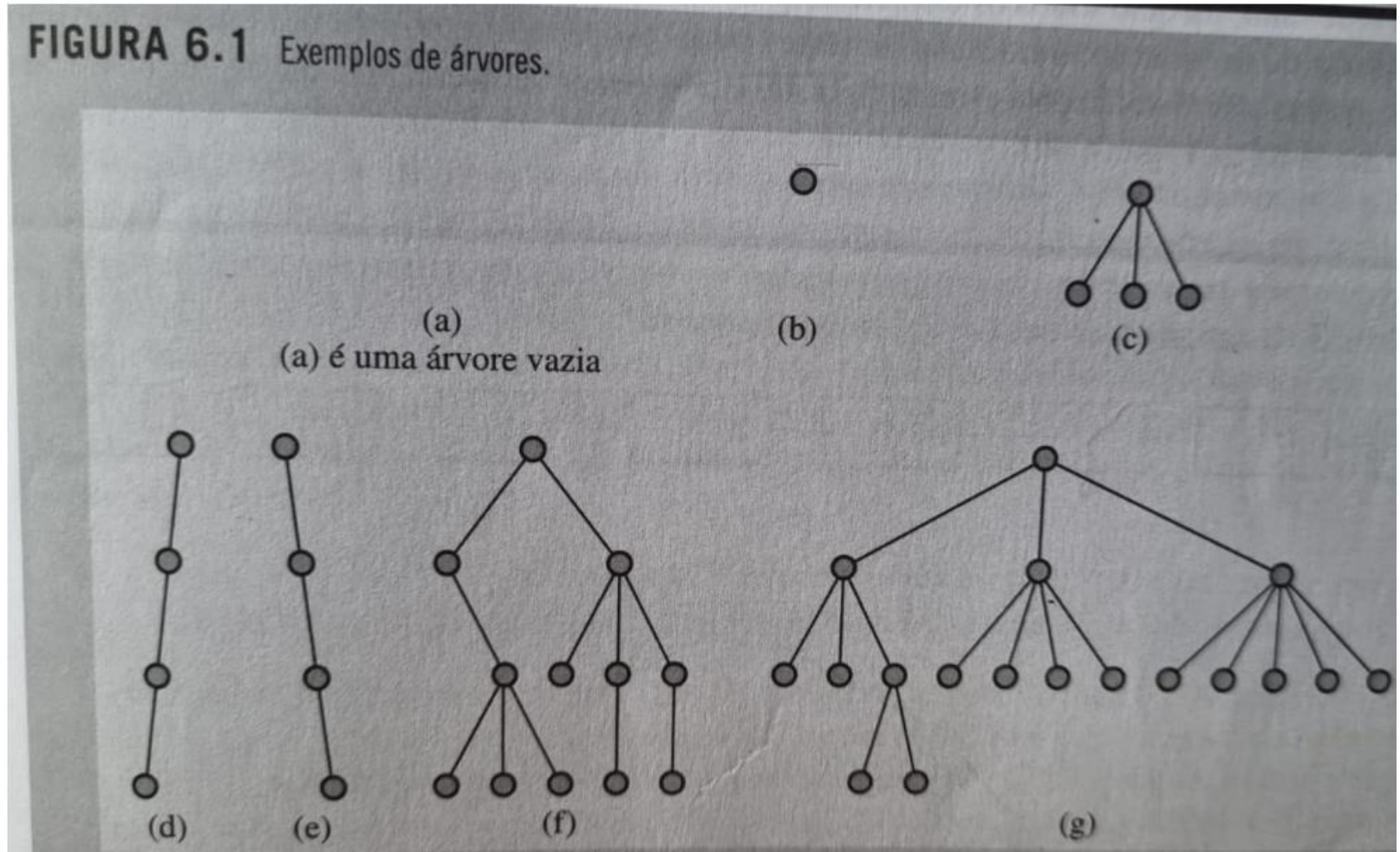


Fig 6.1 pag 187 – E.d.&Algoritmos Em C++

FIGURA 6.2 Estrutura hierárquica de uma universidade mostrada como uma árvore.

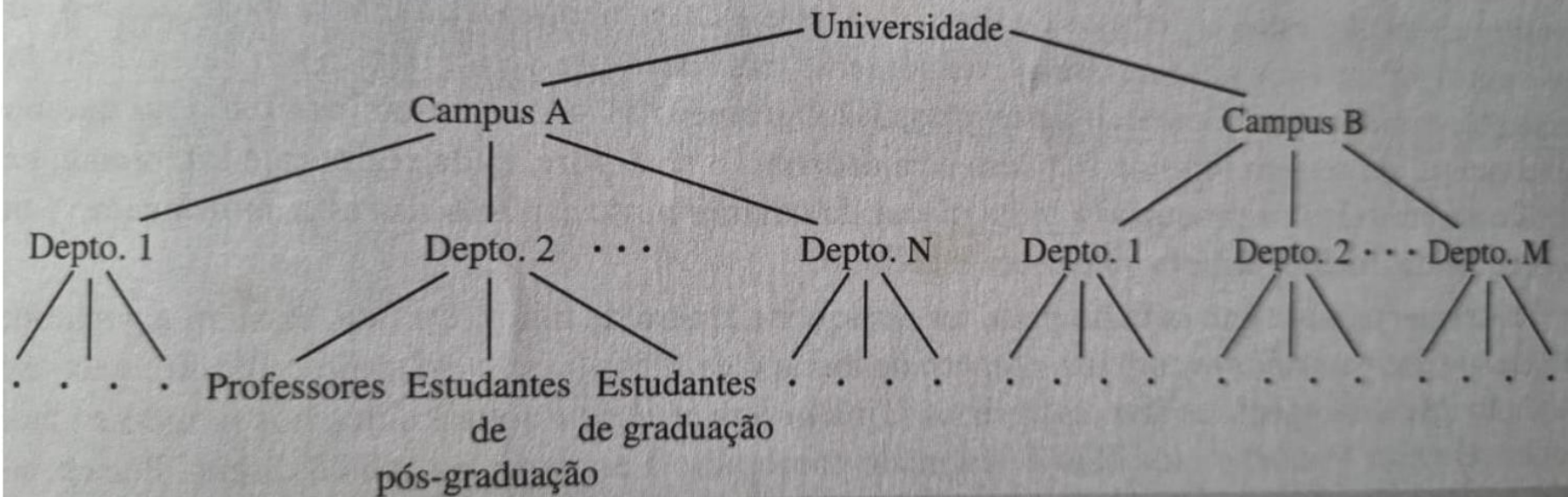
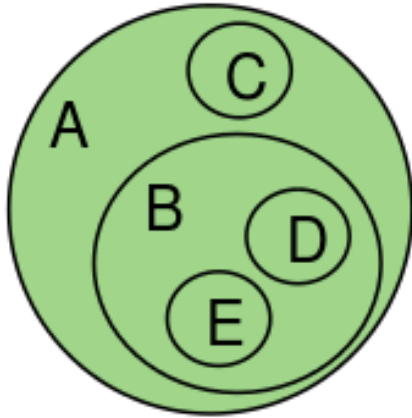


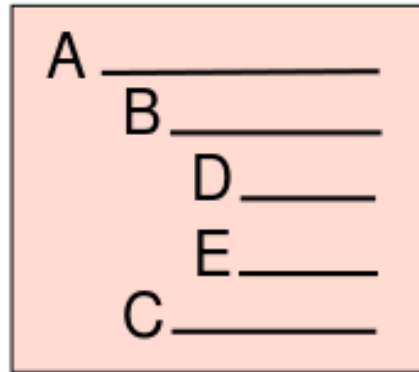
Fig 6.2 pag 187 – E.d.&Algoritmos Em C++

- O número de filhos de um nó pode variar de 0 a qualquer número inteiro.
- Na figura 6.2 a universidade tem 2 campus, mas cada um pode ter número diferentes de departamento.
- Árvores podem ser usadas em sistemas de gerenciamento de banco de dados, organizando o relacionamento entre os dados.
- Operações em árvores podem ACELERAR O PROCESSO DE PESQUISA.

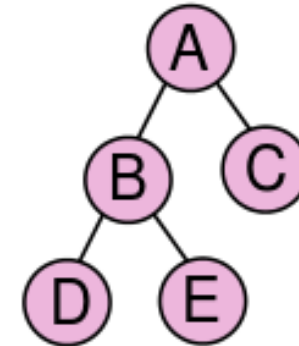
Árvores



c)



b)



a)

1A; 1.1B; 1.1.1D; 1.1.2E;1.2C ;

e)

(A(B(D)(E))(C))

d)

Diferentes representações de uma árvore: a) hierárquica, b) diagrama de barras, c) diagrama de inclusão, d) aninhamento e e) numeração por níveis



- Árvores são estruturas recursivas. Quando a raiz de uma árvore é removida, o que sobra é uma coleção de árvores. Por ex., na figura, quando *a* é removida, obtemos as árvores com raízes *b* e *c*.

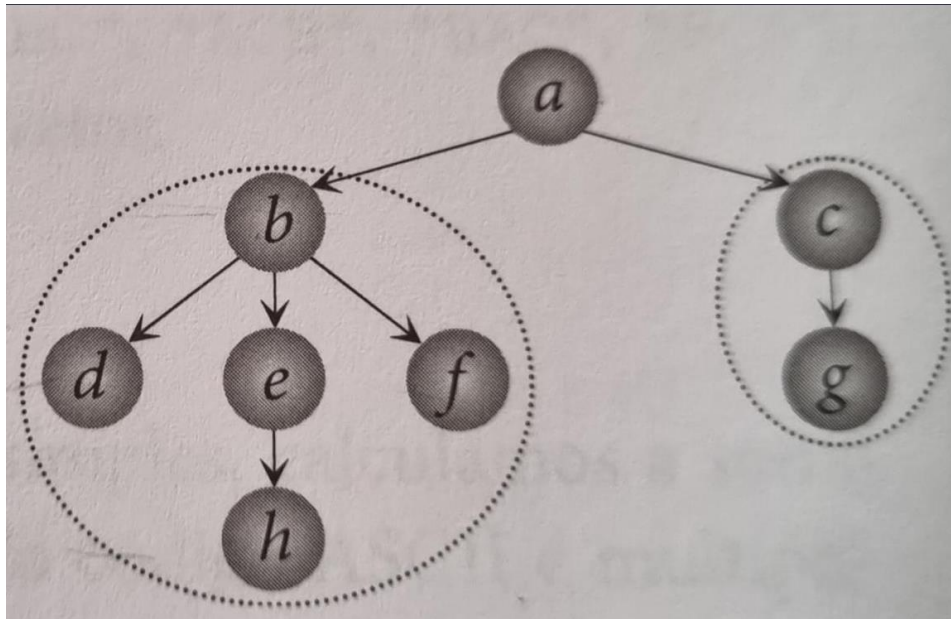


Fig. 13.1 – pag. 130 – E.D.em C

Árvores

- Pode-se designar um nó para ser a **raiz** da árvore, o que demonstra uma relação lógica entre os nós.
- Essas árvores são ditas hierárquicas e a distância entre cada nó e a raiz é denominada de **nível**.
- Em uma árvore hierárquica os nós podem ser rotulados de acordo com a denominação de uma árvore genealógica: **filhos, pais e ancestrais**.

Árvores

- O número de filhos permitido por nó e as informações armazenadas em cada nó diferenciam os diversos tipos de árvores existentes.
- **árvores binárias:** onde cada nó tem, no máximo, dois filhos.
- **árvores genéricas:** onde o número de filhos é indefinido.
 - **Estruturas recursivas** serão usadas como base para implementação das operações com árvores.

Árvores

- Uma árvore é um conjunto de nós composto de um **nó especial (chamado raiz)** e conjuntos disjuntos de **nós subordinados ao nó raiz** que são eles próprios **(sub)árvores**.
- Um nó r , denominado **raiz**, contém zero ou mais **sub-árvores**, cujas raízes são ligadas diretamente a r .
- Esses nós raízes das sub-árvores são **filhos do nó pai, r** .
- **Nós internos:** Nós com filhos.
- **Folhas ou Nós externos:**
- Nós que não têm filhos.

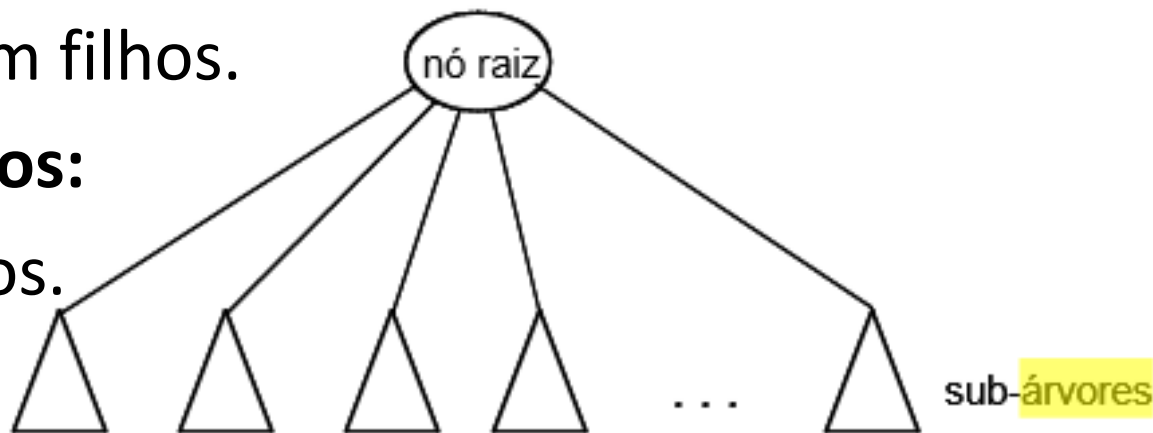
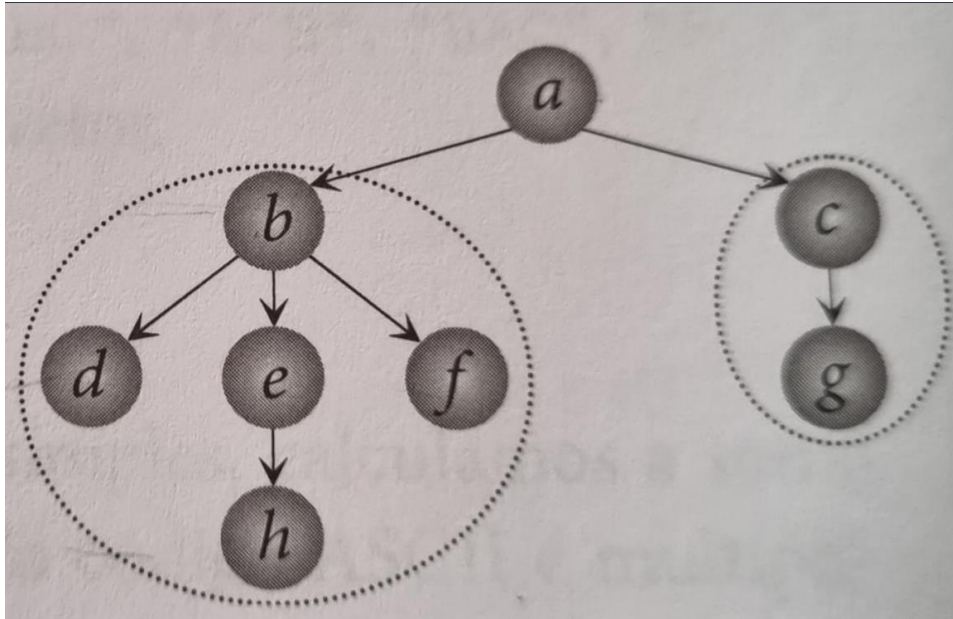


Figura 13.2: Estrutura de árvore.

Árvores

- O número de filhos de um nó é o **GRAU** do nó.



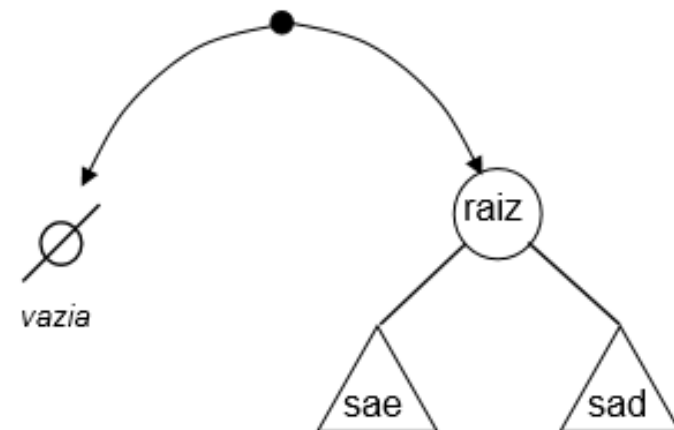
- Na Figura, o grau do nó “a” é 2, o grau de “b” é 3, o grau de “c” e “e” é 1. O grau dos demais nós é 0.
- O **grau de um árvore** é o máximo grau dos seus nós. Na figura, o grau da árvore é 3, pois nessa árvore cada nó tem no máximo 3 filhos.

Árvores

- O nível da raiz de uma árvore é 1. O nível dos filhos de um nó num nível h é $h+1$.
- A altura de uma árvore é o máximo nível dos seus nós (ou 0, se a árvore é vazia).
- Na figura anterior, a altura da árvore é 4. A altura da subárvore enraizada em b é 3 e a altura da subárvore enraizada em c é 2.

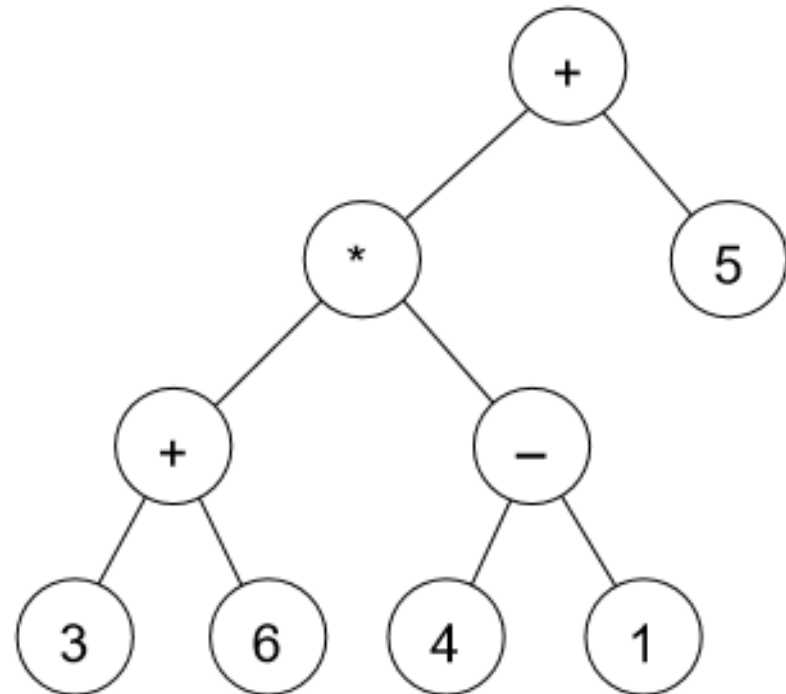
Árvores Binárias

- um árvore em que cada nó tem zero, um ou dois filhos
- uma árvore binária é:
 - uma árvore vazia; ou
 - um nó raiz com duas sub-árvores:
 - a sub-árvore da direita (sad)
 - a sub-árvore da esquerda (sae)



Árvores Binárias

- Exemplo
 - árvores binárias representando expressões aritméticas:
 - nós folhas representam operandos
 - nós internos operadores
 - exemplo: $(3+6)*(4-1)+5$



Árvore Binária

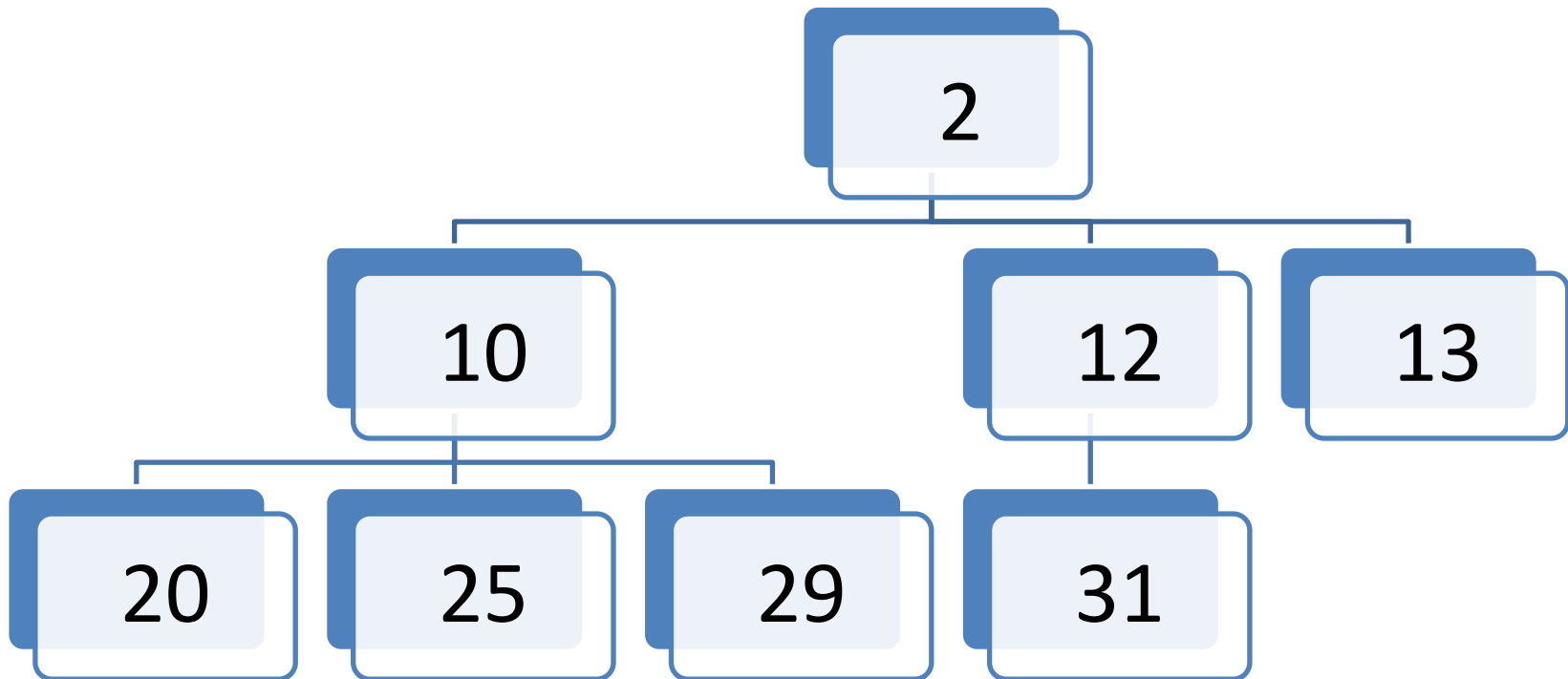
- É uma árvore de **grau 2**, ou seja todo nó tem no max. **2** filhos.
- O nível de um nó é o número de arcos visitados do raiz até o nó + 1.
 - Assim podemos considerar a raiz no nível 1. E seus filhos não vazios no nível 2 e assim por diante.
- Se TODOS os nós em TODOS os níveis tiverem 2 filhos, então haverá **1 (2 elevado 0)** nós no **nível 1**.
 - **2 (2 elevado 1)** nós no **nível 2**.
 - **4 (2 elevado 2)** nós no **nível 3**.
- **Em todas as árvores binárias existem no máximo 2 elevado i nós no nível i+1 =>> uma árvore que satisfaça esta condição é chamada de ÁRVORE BINÁRIA COMPLETA.**

Exemplo Árvore Genérica

- Exemplo: Representação da *lista01* em uma árvore.

• *lista01*:

2 ->	10 ->	12 ->	13->	20 ->	25 ->	29 ->	31 Null
------	-------	-------	------	-------	-------	-------	------------



Exemplo Árvore Genérica

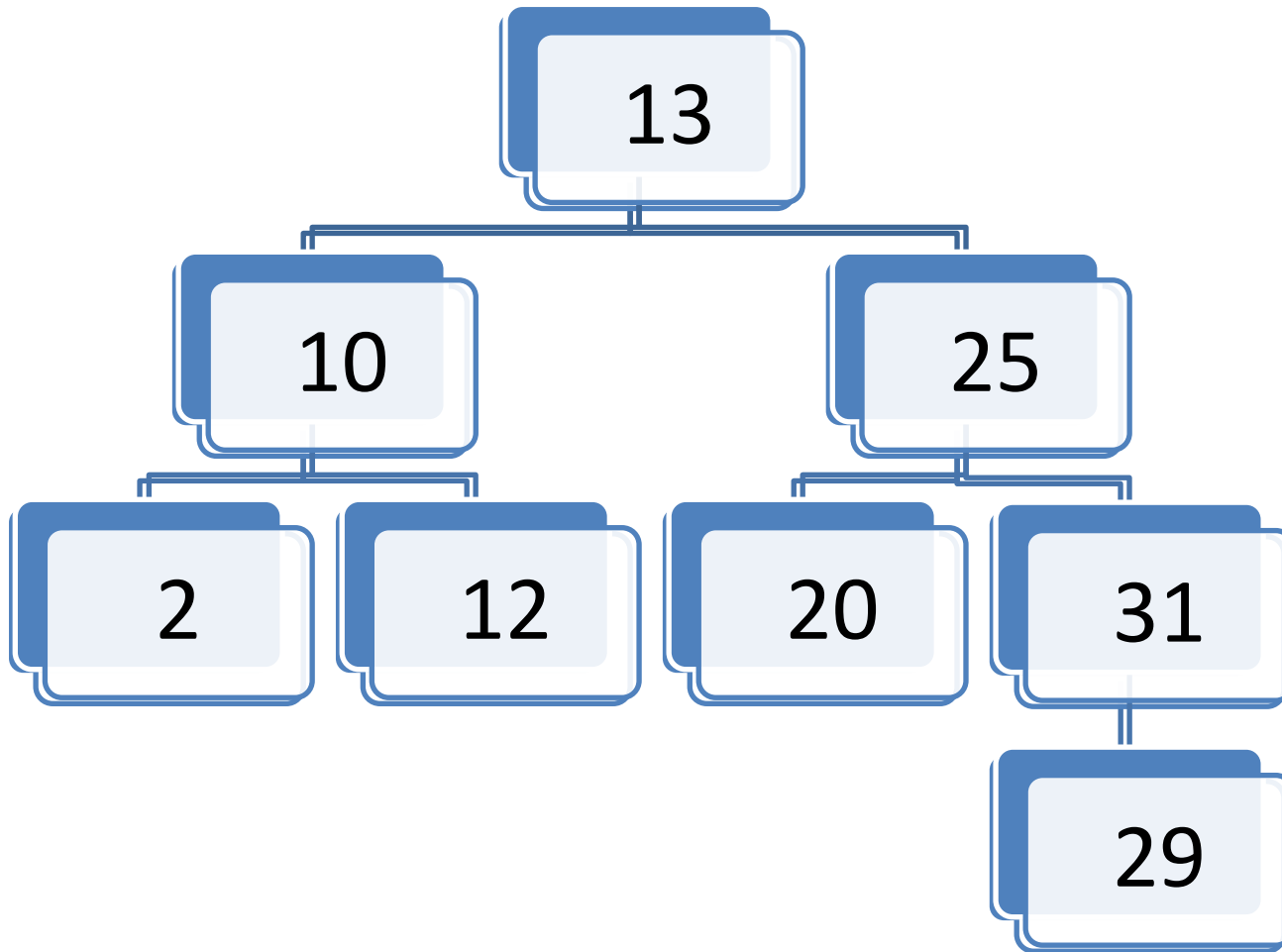
- Para localizar um elemento a pesquisa tem que começar do início da lista.
- Mesmo que esteja ordenada ele vai sempre começar do primeiro nó.

<<Se todos os elementos estão armazenados em uma árvore ordenada, segundo algum critério, o número de testes pode ser reduzido substancialmente.>>

Árvore Binária Ordenada ou Árvore Binária de Busca

- Para cada nó ***n*** da árvore, todos os valores armazenados em sua subárvore à esquerda (a árvore cuja raiz é o filho à esquerda) são menores que o valor ***v*** armazenado em ***n***
- E todos os valores armazenados na subárvore à direita são maiores que ***v***.
- Armazenar múltiplas cópias do mesmo valor é evitado.

Transforma a Árvore Genérica em uma Árvore Binária Ordenada



Árvores Binárias

- Representação de uma árvore:
 - através de um ponteiro para o nó raiz
- Representação de um nó da árvore:
 - estrutura em C contendo
 - a informação propriamente dita (exemplo: um caractere)
 - dois ponteiros para as sub-árvores, à esquerda e à direita

```
struct noArv {  
    char info;  
    struct noArv*  esq;  
    struct noArv*  dir;  
};
```

Proposta Implementação - 1

```
typedef struct noArv NoArv;  
  
NoArv* arv_criavazia (void);  
NoArv* arv_cria (char c, NoArv* e, NoArv* d);  
NoArv* arv_libera (NoArv* a);  
int  arv_vazia (NoArv* a);  
int  arv_pertence (NoArv* a, char c);  
void arv_imprime (NoArv* a);
```

Proposta Implementação - 1

- função `arv_criavazia`
 - cria uma árvore vazia

```
NoArv* arv_criavazia (void)
{
    return NULL;
}
```

`NoArv *root = arv_criavazia();`

Ou simplesmente: `NoArv *root = NULL;`

Proposta Implementação - 1

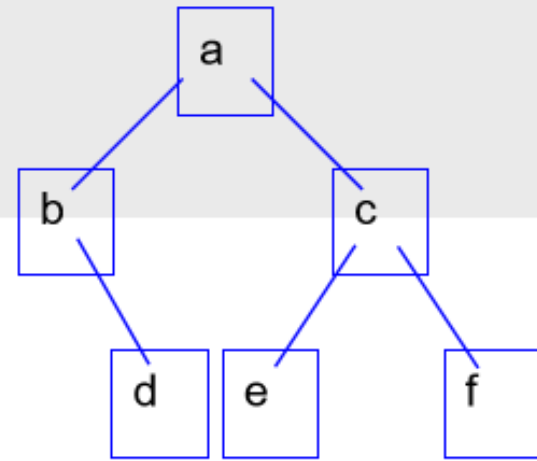
- função `arv_cria`
 - cria um nó raiz dadas a informação e as duas sub-árvores, a da esquerda e a da direita
 - retorna o endereço do nó raiz criado

```
NoArv* arv_cria (char c, NoArv* sae, NoArv* sad)
{
    NoArv* p=(NoArv*)malloc(sizeof(NoArv)) ;
    if(p==NULL) exit(1);
    p->info = c;
    p->esq = sae;
    p->dir = sad;
    return p;
}
```

Implementação 1 - Exemplo

- Exemplo: <a <b <> <d <><>> > <c <e <><> > <f <><> > > >

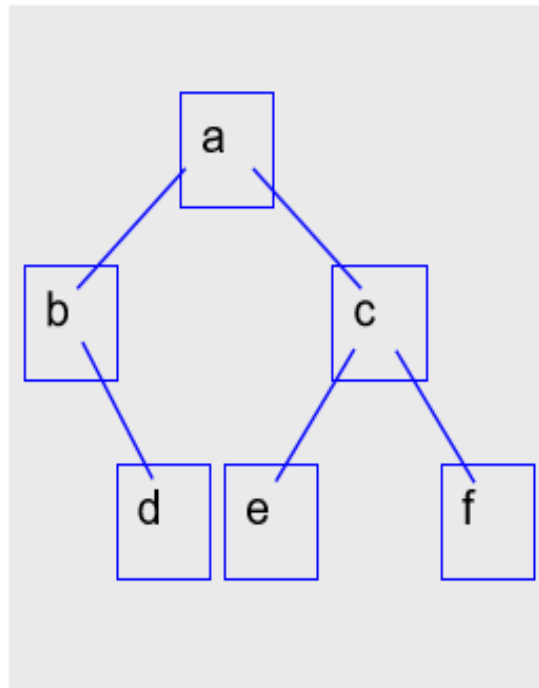
```
/* sub-árvore 'd' */
NoArv* a1= arv_cria('d',arv_criavazia(),arv_criavazia());
/* sub-árvore 'b' */
NoArv* a2= arv_cria('b',arv_criavazia(),a1);
/* sub-árvore 'e' */
NoArv* a3= arv_cria('e',arv_criavazia(),arv_criavazia());
/* sub-árvore 'f' */
NoArv* a4= arv_cria('f',arv_criavazia(),arv_criavazia());
/* sub-árvore 'c' */
NoArv* a5= arv_cria('c',a3,a4);
/* árvore 'a' */
NoArv* a = arv_cria('a',a2,a5 );
```



Implementação 1

- Exemplo: <a <b <> <d <><>> > <c <e <><>> <f <><>> > > >

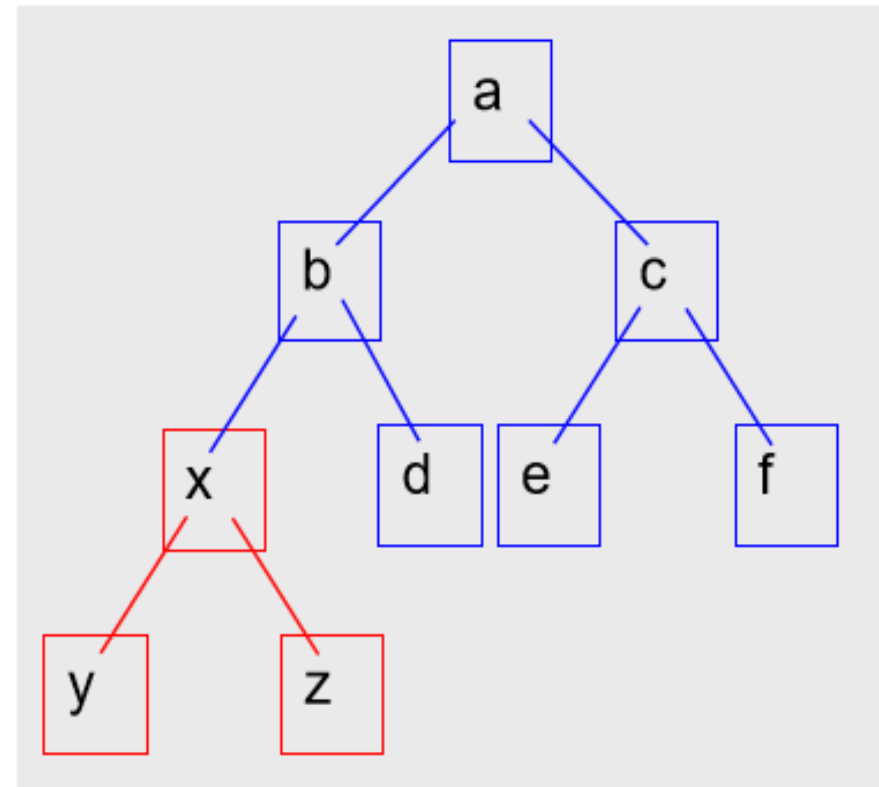
```
NoArv* a = arv_cria('a',  
    arv_cria('b',  
        arv_criavazia(),  
        arv_cria('d', arv_criavazia(), arv_criavazia())  
    ),  
    arv_cria('c',  
        arv_cria('e', arv_criavazia(), arv_criavazia()),  
        arv_cria('f', arv_criavazia(), arv_criavazia())  
    )  
);
```



Implementação 1

- Exemplo - acrescenta nós

```
a->esq->esq =  
  arv_cria('x',  
    arv_cria('y',  
      arv_criavazia(),  
      arv_criavazia()),  
    arv_cria('z',  
      arv_criavazia(),  
      arv_criavazia())  
  );
```



Implementação 1

- função `arv_imprime`
 - percorre recursivamente a árvore, visitando todos os nós e imprimindo sua informação

```
void arv_imprime (NoArv* a)
{
    if (!arv_vazia(a)) {
        printf("%c ", a->info);    /* mostra raiz */
        arv_imprime(a->esq);        /* mostra sae */
        arv_imprime(a->dir);        /* mostra sad */
    }
}
```

Implementação 2

```
class No{  
    public:  
        char nome;  
        No *left;  
        No *right;  
        No(char n){  
            nome=n;  
            left=NULL;  
            right=NULL;  
        }  
};
```

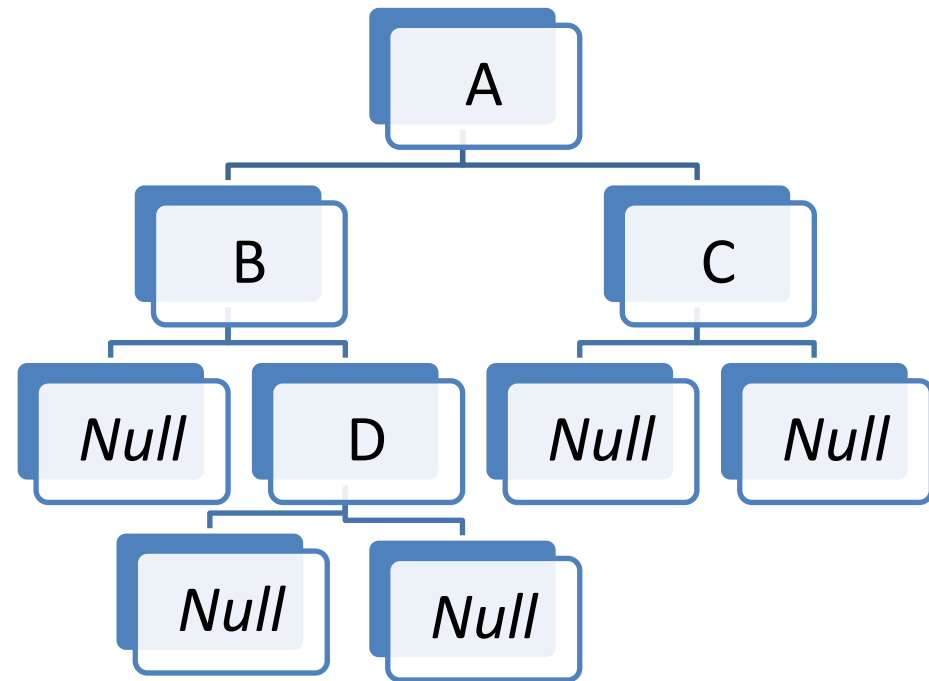
```
class Arvore {  
    public:  
        No *raiz;  
  
        Arvore(){  
            raiz= NULL;  
        }  
  
        int isEmpty(){  
            return (raiz==NULL);  
        }  
.....}
```

Atividade

- Implemente a criação de um novo nó, de maneira que, o método informe o pai do novo nó e de que lado ele deve ser inserido (se deve ser inserido a esquerda do pai ou a direita do pai).

DICA

```
main(){  
    //lado 1=esq e 2=dir  
    Arvore *arv = new Arvore();  
    arv->cria_No('A');  
    arv->cria_No('B',1,'A');  
    arv->cria_No('D',2,'B');  
    arv->cria_No('T',2,'B');  
    arv->cria_No('C',2,'A');  
    arv->imprime(arv->raiz);  
}
```



DICA

```
void cria_No(char nov){  
    No *novo=new No(nov,NULL,NULL);  
    raiz=novo;  
  
}  
void cria_No(char nov, int lado, char pai){  
    No *novo=new No(nov,NULL,NULL);  
    insere(raiz,novo,lado,pai);  
}
```

<<Dica: Faça o método insere() recursivo.>>

```
// 1- ESQ 2 - DIR
```

```
void insere(No *arv, No *novo, int lado, char pai){  
    if (arv != NULL){  
        if (arv->info == pai){  
            if (lado == 1)  
                if (arv->esq == NULL)  
                    arv->esq = novo;  
                else  
                    cout << "\n ERRO - ja existe um no nessa posicao!!!";  
            if (lado == 2)  
                if (arv->dir == NULL)  
                    arv->dir = novo;  
                else  
                    cout << "\n ERRO - ja existe um no nessa posicao!!!";  
        }  
        else{  
            insere(arv->dir, novo, lado, pai);  
            insere(arv->esq, novo, lado, pai);  
        }  
    }  
}
```

Dica:

DICA:

```
void imprime(No *n){  
    if (raiz==NULL){  
        cout<<"\n <VAZIO>";  
    }  
    else{  
        if (n!=NULL){  
            cout<<"<"<<n->info;  
            imprime(n->esq);  
            imprime(n->dir);  
            cout<<">";  
        }  
        else  
            cout<<"<>";  
    }  
}
```

Verificando se existe o Nó

- função `arv_vazia`
 - indica se uma árvore é ou não vazia

```
int arv_vazia (NoArv* a)
{
    return a==NULL;
}
```

Retirando um nó da árvore

- função `arv_libera`
 - libera memória alocada pela estrutura da árvore
 - as sub-árvores devem ser liberadas antes de se liberar o nó raiz
 - retorna uma árvore vazia, representada por `NULL`

```
NoArv* arv_libera (NoArv* a) {  
    if (!arv_vazia(a)) {  
        arv_libera(a->esq);      /* libera sae */  
        arv_libera(a->dir);      /* libera sad */  
        free(a);                 /* libera raiz */  
    }  
    return NULL;  
}
```

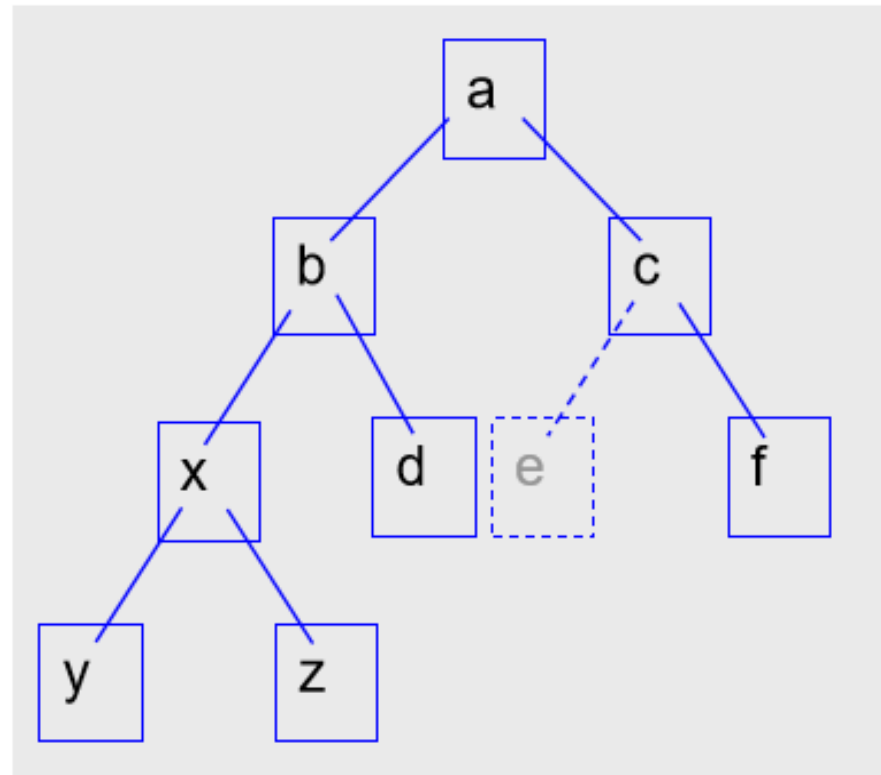
Exemplo

- Exemplo - libera nós

```
a->dir->esq = libera(a->dir->esq);
```

OBSERVE que ele já passa o **ENDEREÇO DO PAI** como **parâmetro!**

Se você NÃO tem o endereço do pai, É NECESSÁRIO INICIALMENTE ACHAR O PAI DO NÓ QUE DEVE SER RETIRADO!!!



- Primeiro encontre o nó que deve ser retirado.
- Para isso, passe como parâmetro para o método o nó raiz e o procurado.

```
void liberaFilho(char procurado, No *n){  
    No *tmp=NULL;  
    if (raiz->info==procurado){  
        tmp=raiz;  
        libera(tmp);  
        raiz=NULL;  
    }  
    else{  
        if (n->dir!=NULL){  
            if (n->dir->info==procurado){  
                No *prox=n->dir;  
                n->dir=NULL;  
                libera(prox);  
            }  
            else  
                liberaFilho(procurado, n->dir);  
        }  
        if (n->esq!=NULL){
```

DICA

Fazer o filho
apontar
para NULL

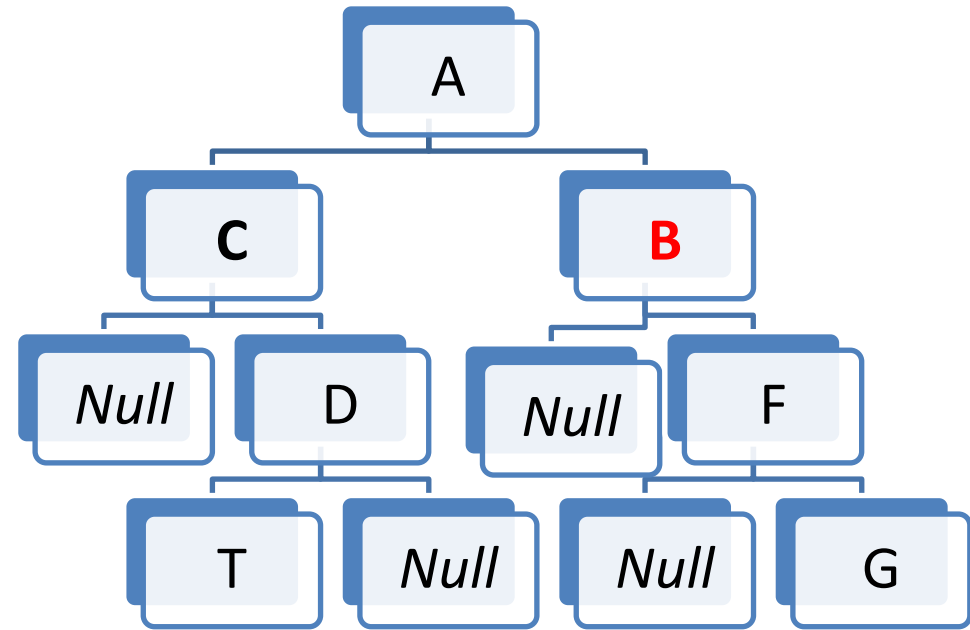
DICA

- Depois de fazer o ponteiro ***pai->esq*** ou ***pai->dir*** apontar para ***NULL***, libere o filho da memória (aplicar ***free(filho)***):

```
void libera(No *pai){  
    No *tmp;  
  
    if (pai!=NULL){  
        if (pai->esq!=NULL){  
            libera(pai->esq);  
        }  
        if (pai->dir!=NULL){  
            libera(pai->dir);  
        }  
  
        free(pai);  
    }  
}
```

Exemplo

- Primeiro encontra o B:
`arv->liberaFilho('B',arv->raiz);`



- Fazer *pai->dir* ou *pai->esq* igual a NULL, dependendo de onde está o filho. Depois liberar a sub-arvore:
if (*pai->dir*!=NULL)
 if (*pai->dir->nome*==procurado){
 No *prox=*pai->dir*;
 pai->dir=NULL;
 libera(prox); }

Exemplo

- **Se for folha:** verifica se os lados do pai são iguais a nulo, se for (é folha) pode liberar a memória: *free(a)*
- **Se não for folha:**
 - Se lado esquerdo é diferente de nulo, percorre filhos:
libera(a->esq);
 - Se lado direito é diferente de nulo, percorre filhos:
libera(a->dir);
 - Depois de remover os filhos, remova o pai: *free(a)*
- No exemplo: libera da memória da seguinte ordem: G, F, B

Verificando se o nó pertence a árvore

- função `arv_pertence`
 - verifica a ocorrência de um caractere `c` em um de nós
 - retorna um valor booleano (1 ou 0) indicando a ocorrência ou não do caractere na árvore

```
int arv_pertence (NoArv* a, char c)
{
    if (arv_vazia(a))
        return 0; /* árvore vazia: não encontrou */
    else
        return a->info==c ||
               arv_pertence(a->esq,c) ||
               arv_pertence(a->dir,c) ;
}
```

DICA

- Se o nó NÃO foi encontrado, verifique os filhos do nó (esquerda e direita).

```
void buscar(No *ele, char n){  
    if (ele->info==n)  
        cout<<"\n Elemento "<<ele->info<<" encontrado!!!";  
    else{  
        if (ele->dir!=NULL)  
            buscar(ele->dir,n);  
  
        if (ele->esq!=NULL)  
            buscar(ele->esq,n);  
  
    }  
}
```

Ordens de Percurso Em Profundidade

- Ordens de percurso:

- *pré-ordem*:

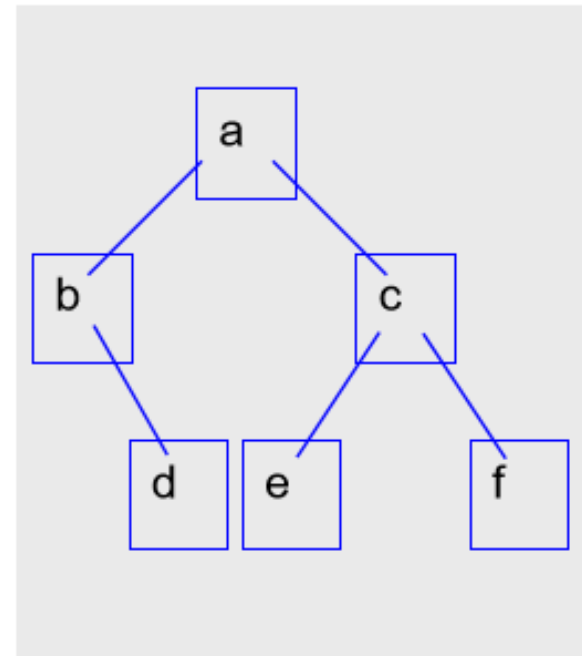
- trata *raiz*, percorre *sae*, percorre *sad*
 - exemplo: a b d c e f

- *ordem simétrica*:

- percorre *sae*, trata *raiz*, percorre *sad*
 - exemplo: b d a e c f

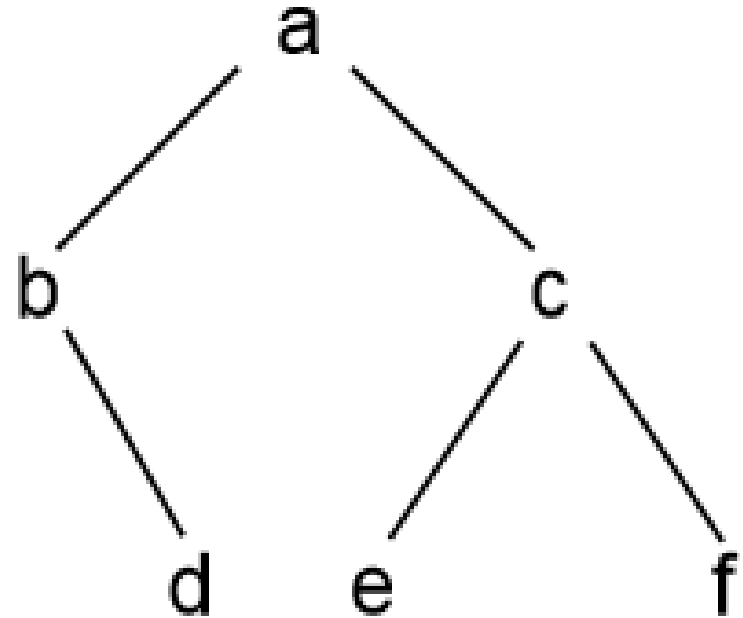
- *pós-ordem*:

- percorre *sae*, percorre *sad*, trata *raiz*
 - exemplo: d b e f c a



Atividade

1. Imprima os nós da árvore, de forma que a saída impressa reflita, além do conteúdo de cada nó, a estrutura da árvore. Assim, a saída da função seria: `<a<b<><d<><>>><c<e<><>><f<><>>>>` para o exemplo ao lado.
2. Qual tipo de percurso utilizado no exemplo acima?

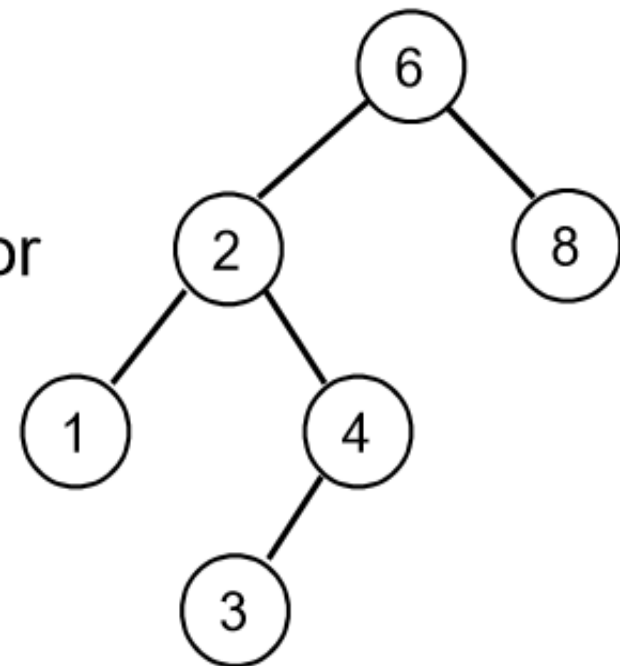


Atividade

- Faça método de impressão usando ordem simétrica (cruzamento de árvore in-ordem).
- Faça método de impressão usando pós-ordem.
- Procure um nó na árvore. Imprima o nó encontrado, se existir, ou informe que o nó não existe.
- Procure o pai de um nó na árvore. Imprima o nome do pai, se existir, ou informe que o pai não existe.
- Libere um nó na árvore.

Árvore Binária de Busca (ABB)

- o valor associado à raiz é sempre maior que o valor associado a qualquer nó da sub-árvore à esquerda (*sae*) e
- o valor associado à raiz é sempre menor ou igual (para permitir repetições) que o valor associado a qualquer nó da sub-árvore à direita (*sad*)
- quando a árvore é percorrida em ordem simétrica (*sae - raiz - sad*), os valores são encontrados em ordem não decrescente




Adicionando um nó em uma ABB

- para adicionar v na posição correta, faça:
 - se a (sub-)árvore for vazia
 - crie uma árvore cuja raiz contém v
 - se a (sub-)árvore não for vazia
 - compare v com o valor na raiz
 - insira v na sae ou na sad, conforme o resultado da comparação

Inserindo novo Nó na classe Árvore

```
void criaNo(char n){  
    No *novo= new No(n);  
    insere(raiz,novo);  
}
```



Inserindo nó - sem uso de recursão

```
void insere(No *arv, No *n){
    if (isEmpty()==1) //raiz nula, arvore vazia
        raiz=n;
    else{
        No *percorre=raiz;

        while (percorre!=NULL){
            if (percorre->nome<n->nome)
                if (percorre->right==NULL){
                    percorre->right=n;
                    break;
                }
            else
                percorre=percorre->right;
        }
    }
}
```

ATIVIDADE:

Continue a Implementação acima!

Inserindo nó – usando de recursão

```
No *insere2(No *arv, No *n){  
    if (isEmpty()==1) //raiz nula, arvore vazia  
        raiz=n;  
    else{  
        if (arv!=NULL){  
            |         if (arv->nome<n->nome){  
                        if (arv->right==NULL){  
                            arv->right=n;  
                        }  
                    }  
        }  
    }  
}
```

ATIVIDADE:

Continue a Implementação acima!

Pesquisa em ABB

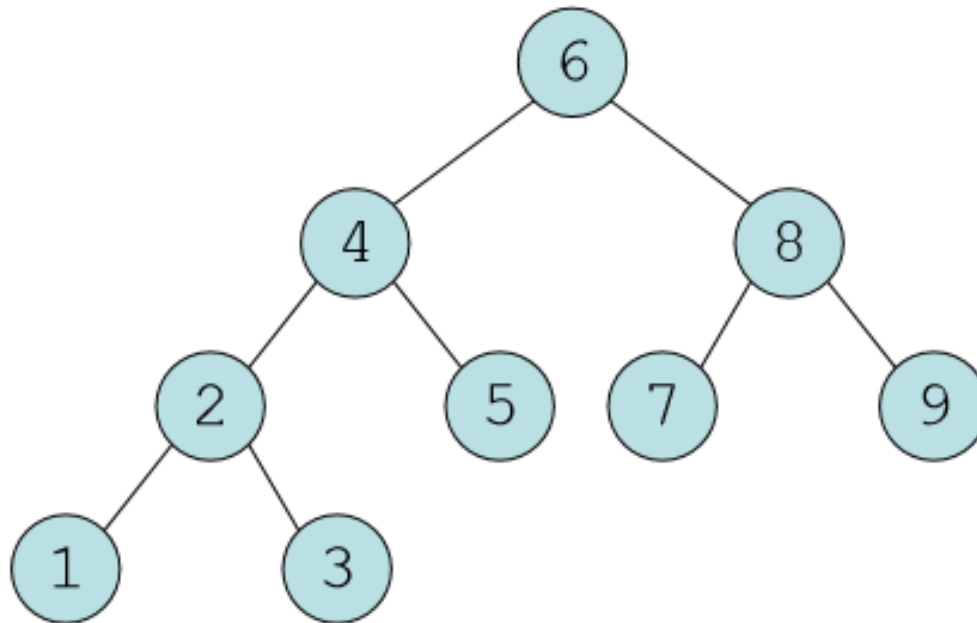
- compare o valor dado com o valor associado à raiz
- se for igual, o valor foi encontrado
- se for menor, a busca continua na sae
- se for maior, a busca continua na sad

DICA

```
No *buscarOrd(No *ele, char n){  
    if (ele->info==n)  
        return ele;  
    else{  
        if (ele->info<n){  
            if (ele->dir!=NULL)  
                return buscarOrd(ele->dir,n);  
            else  
                return NULL;  
        }  
        if (ele->info>n){  
            if (ele->esq!=NULL)  
                return buscarOrd(ele->esq,n);  
            else  
                return NULL;  
        }  
    }  
}
```

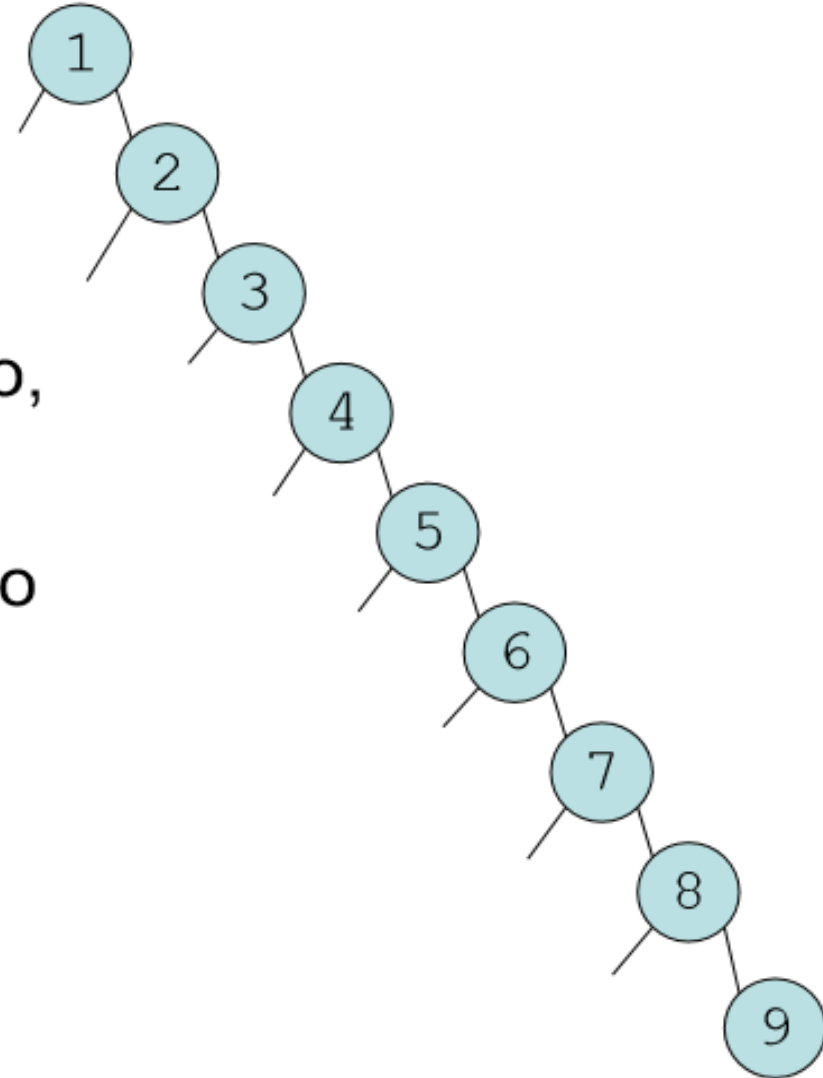
Árvore Binária de Busca Balanceada

- os nós internos têm todos, ou quase todos, 2 filhos
- qualquer nó pode ser alcançado a partir da raiz em $O(\log n)$ passos



Árvore Binária de Busca Degenerada ou Assimétrica

- todos os nós têm apenas 1 filho, com exceção da (única) folha
- qualquer nó pode ser alcançado a partir da raiz em $O(n)$ passos



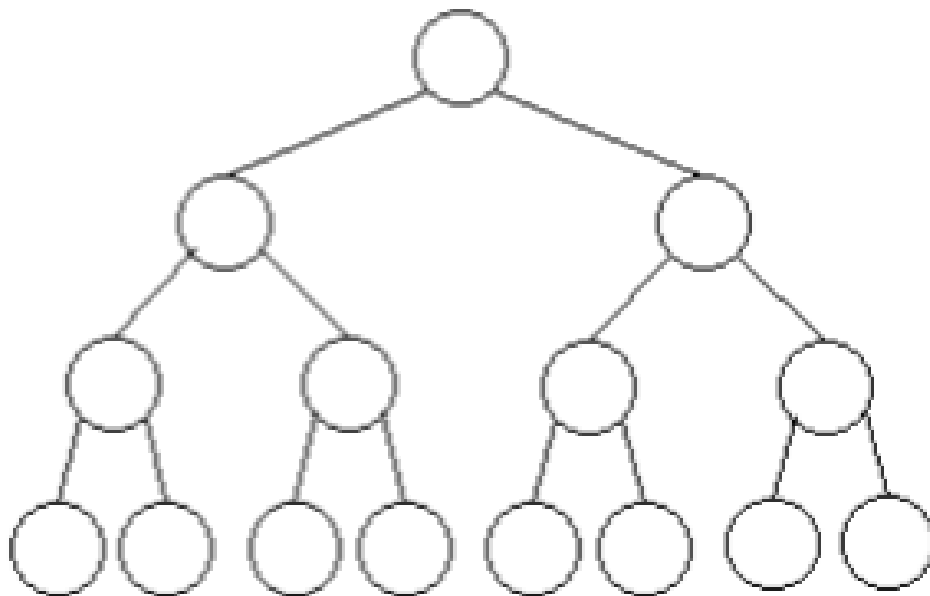
Busca na Árvore Binária

- explora a propriedade de ordenação da árvore
- possui desempenho computacional proporcional à altura ($O(\log n)$ para o caso de árvore balanceada)

```
NoArv* abb_busca (NoArv* r, int v)
{
    if (r == NULL)
        return NULL;
    else if (r->info > v)
        return abb_busca (r->esq, v);
    else if (r->info < v)
        return abb_busca (r->dir, v);
    else return r;
}
```

Propriedades

- O número máximo de nós possíveis no nível i é 2^{i-1} :



nível 1 = $2^{1-1} = 1$ nó

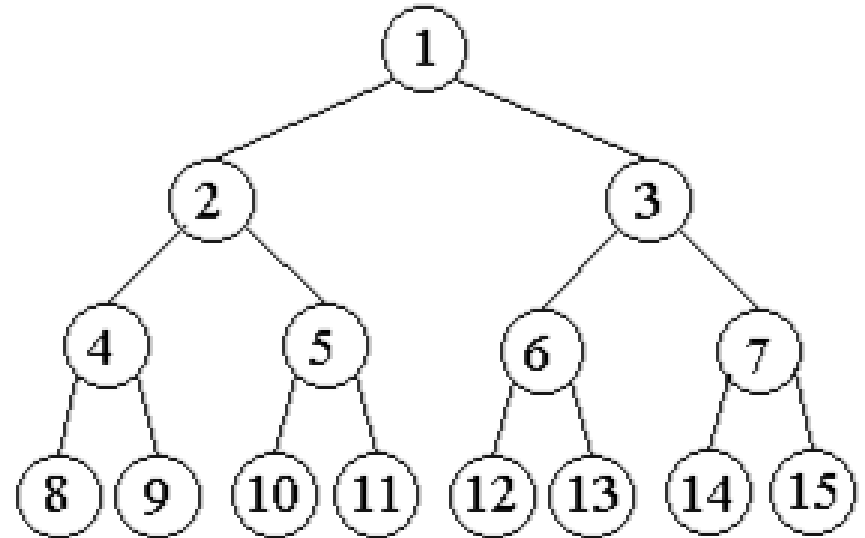
nível 2 = $2^{2-1} = 2$ nós

nível 3 = $2^{3-1} = 4$ nós

nível 4 = $2^{4-1} = 8$ nós

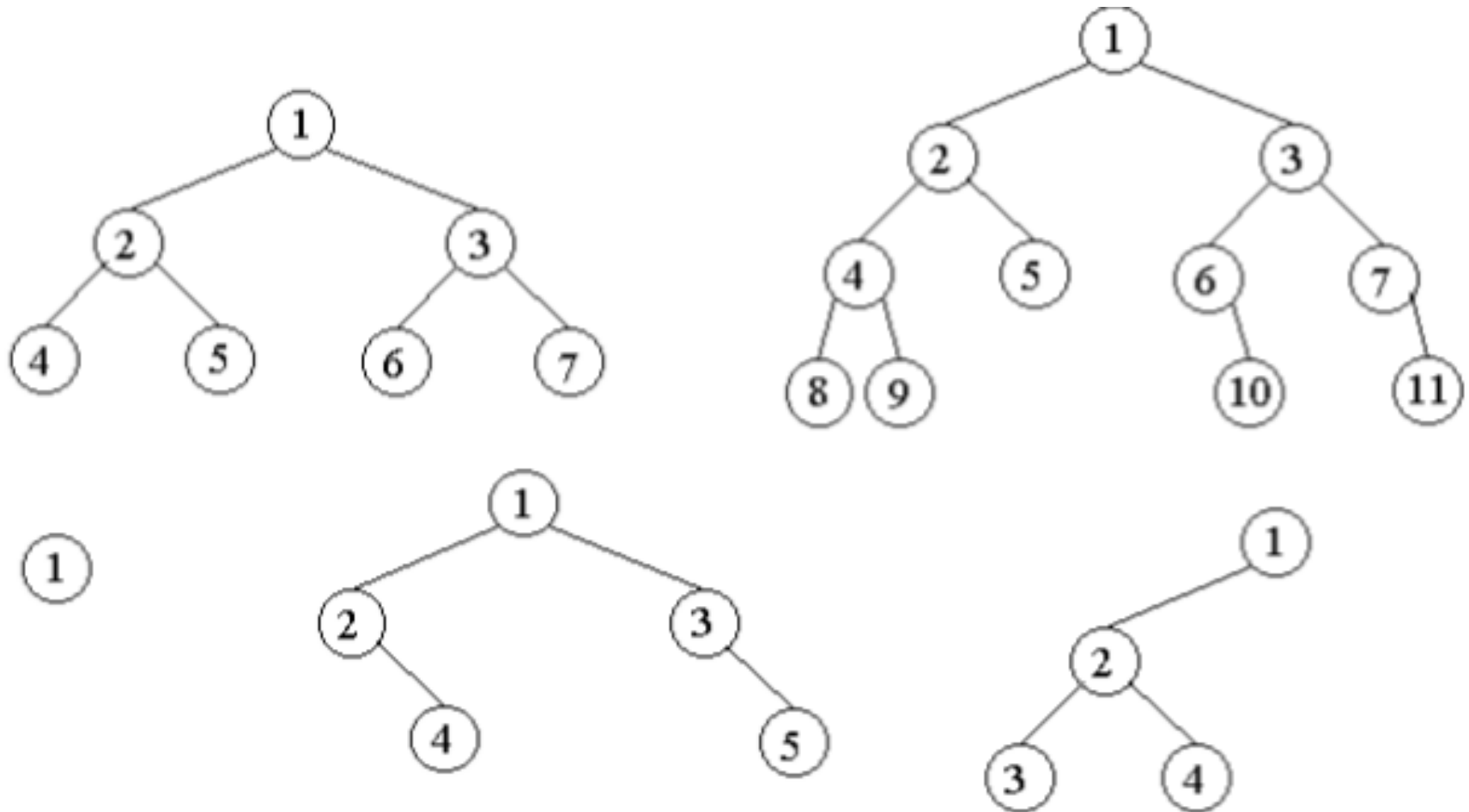
Propriedades

- Uma árvore binária de altura h tem no máximo $2^h - 1$ nós.
 - Ou considerando h começando de zero:
 - Uma árvore binária de altura h tem no máximo $2^{h+1} - 1$ nós.
 - E o mínimo de $h+1$ nós.



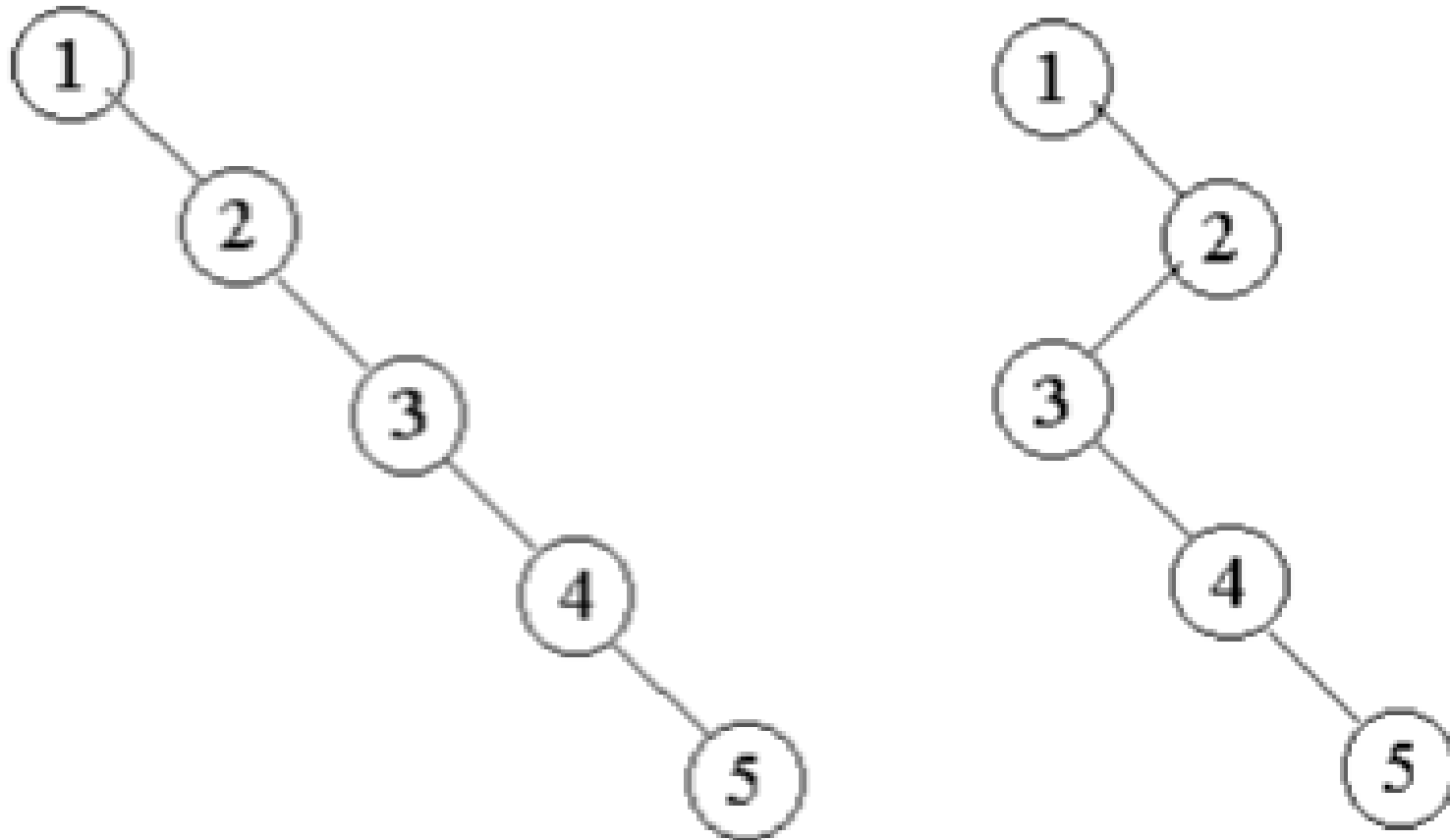
- A **altura mínima** de uma para $h = 4$, $n_{\text{máx}} = 2^4 - 1 = 15$
- árvore binária com $n > 0$ nós é $1 + \text{chão}(\log n)$, com
1º $h=1$, e a $h=\text{chão}(\log n)$, considerando 1º. $h=0$

Uma árvore binária de altura mínima é dita **completa**.



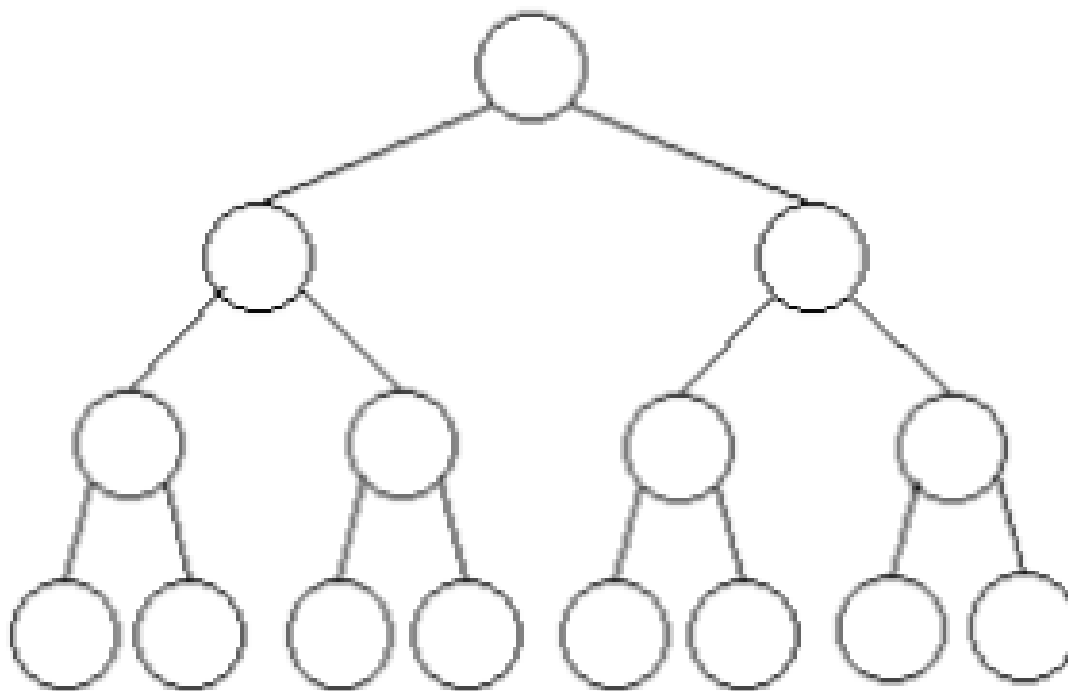
Exemplos de **árvores** completas. Seus nós não podem ser redistribuídos formando uma árvore de altura menor do que estas.

Uma árvore binária de altura máxima para uma quantidade de n nós é dita **assimétrica**. Neste caso, a altura é $h = n$ e seus nós interiores possuem exatamente uma subárvore vazia cada.



Exemplos de **árvores** assimétricas

Uma árvore binária de altura h é **cheia** se possui exatamente $2^h - 1$ nós.



Exemplo de árvore cheia – possuindo o número máximo de nós para a sua altura.

Atividade – observações

- Inserção em árvore binária não ordenada passando como informação o lado e o nó pai.
- Remoção na árvore indicando o elemento.
 - Obs: árvore de busca ordenada: escolher o lado
 - Obs: árvore NÃO ORDENADA: PERCORRER OS DOIS LADOS (recursivo e não recursivo)
- Fazer as três ordens de percurso em profundidade usando recursão
- **Fazer as três ordens de percurso em profundidade sem usar recursão (use a pilha (stack) do std)

Atividade

- Para um determinada árvore binária calcule se ela é uma árvore completa ou não.
- Para isso :
 - calcule a quantidade de nós, verifique a altura mínima para essa quantidade de nós.
 - Calcule a altura da árvore.
 - Compare a altura mínima com a altura da árvore, se forem iguais, é uma árvore completa, se forem diferentes é uma árvore incompleta.

Nó de uma árvore usando template

```
template<class T>
class ArvoreNo {
public:
    T el;
    ArvoreNo<T> *left,*right;
    ArvoreNo(){
        left=right=0;
    }
    ArvoreNo(T e,ArvoreNo<T> *l=0,ArvoreNo<T> *r=0){
        el=e;
        left=l;
        right=r;
    }
};
```

Complete criando a classe Arvore e os métodos de inserção e percorrer em profundidade(in-order, simétrica, pós-order)