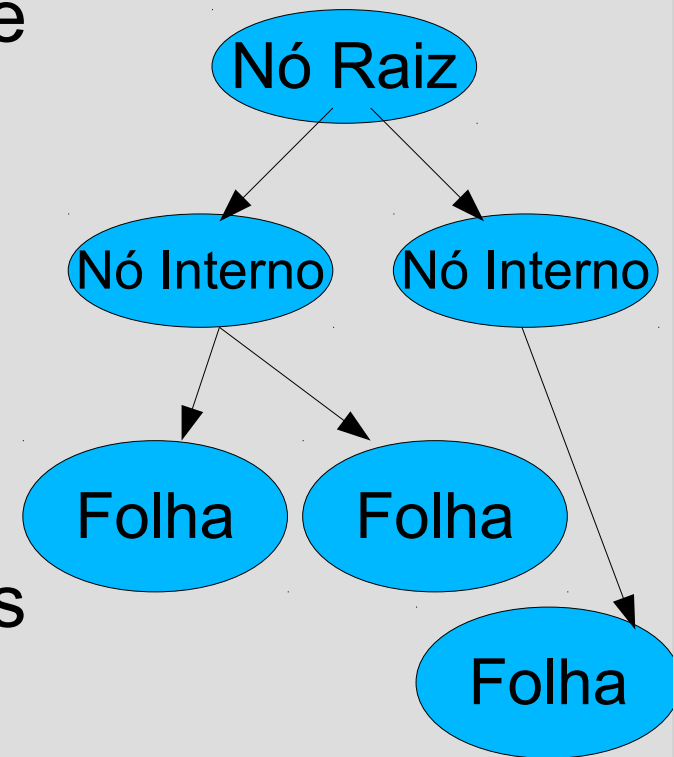


Estruturas de Dados

Árvore

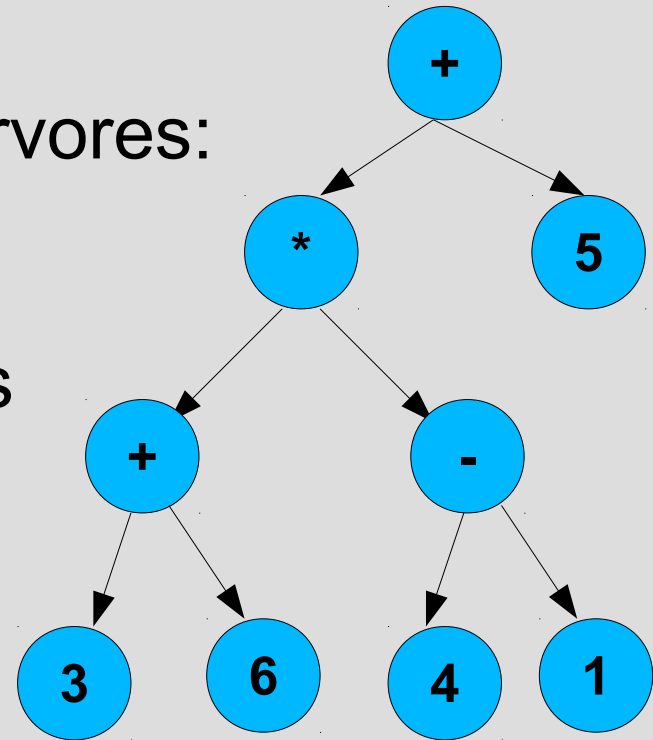
Árvore

- Em diversas aplicações, precisamos de estruturas não-lineares. Em especial, quando precisamos de estruturas hierárquicas;
- Uma Árvore é constituída por um **conjunto de nós** tal que:
 - Existe um nó r , denominado de **raiz**, com zero ou mais sub-árvores, cujas raízes estão ligadas a r . Os nós raízes destas sub-árvores são os **filhos** de r ;
 - Os **nós internos** são nós com filhos;
 - Os **nós externos ou folhas** são os nós sem filhos.



Árvore Binária

- Árvore em que cada nó tem no máximo 2 filhos
- Uma árvore binária é
 - Uma árvore vazia
 - Um nó Raiz com duas sub-árvores:
 - Sub-árvore da esquerda.
 - Sub-árvore da direita.
- Em geral, utilizam-se algoritmos recursivos
- Exemplo: $((3+6)*(4-1))+5$



Tipo Abstrato de Dado Árvore Binária

Podemos criar um TAD Árvore Binária de caracteres. Para tanto, devemos criar o arquivo `arvore.h` com o nome do tipo e os protótipos.

```
typedef struct arv Arv;
```

```
/*Função que cria uma Árvore Binária Vazia.*/
```

```
Arv* arv_cria_vazia(void);
```

```
/*Função que cria um nó em uma Árvore Binária.*/
```

```
Arv* arv_cria_no(char c, Arv *sae, Arv *sad);
```

```
/*Testa se uma Árvore Binária é vazia.*/
```

```
int arv_vazia(Arv *a);
```

```
/*Função que imprime os elementos de uma Árvore Binária.*/
```

```
void arv_imprime(Arv *a);
```

```
/*Função que determina se um caractere pertence à Árvore.*/
```

```
int arv_pertence(Arv *a, char c);
```

```
/*Libera o espaço alocado para uma Árvore Binária.*/
```

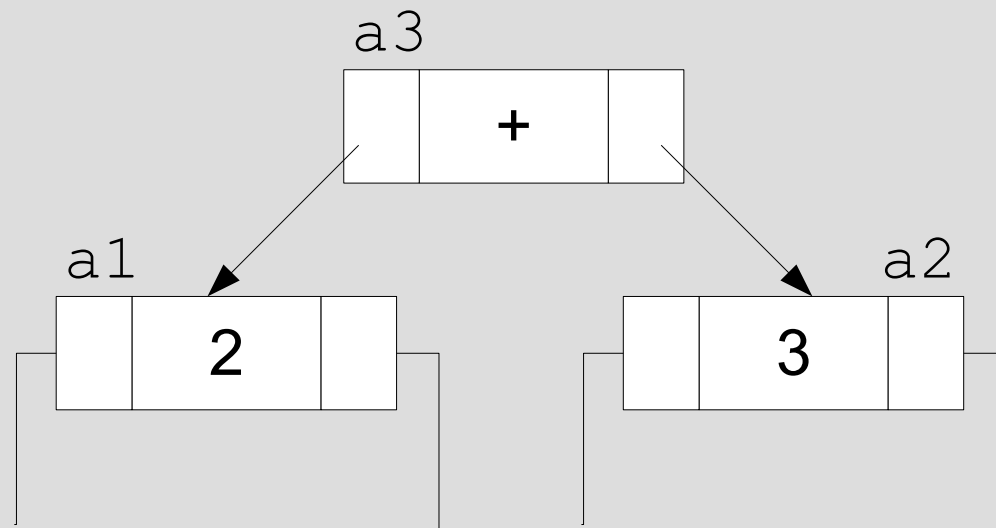
```
void arv_libera(Arv *a);
```

Estrutura de Árvore Binária

- A estrutura que representa o tipo árvore binária deve ser composta pela informação a ser armazenada e a sub-árvore da esquerda e da direita

```
struct arv{  
    char info;  
    Arv *esq;  
    Arv *dir;  
};
```

```
Arv a1; a1.info='2'; a1.esq=NULL; a1.dir = NULL;  
Arv a2; a2.info='3'; a2.esq=NULL; a2.dir = NULL;  
Arv a3; a3.info='+'; a3.esq=&a1; a3.dir = &a2;
```



Árvore Binária

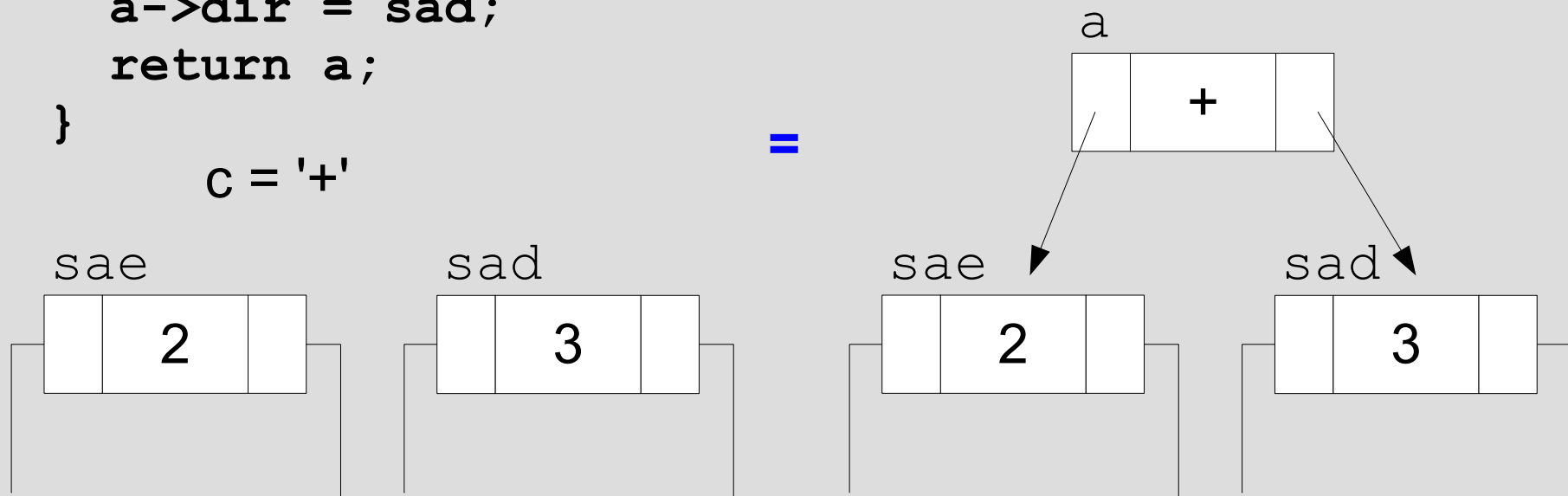
Funções Cria Árvore Vazia e Nó

```
Arv* arv_cria_vazia(void) {  
    return NULL;  
}
```

```
Arv* arv_cria_no(char c, Arv *sae, Arv *sad) {  
    Arv* a = (Arv*)malloc(sizeof(Arv));  
    a->info = c;  
    a->esq = sae;  
    a->dir = sad;  
    return a;  
}
```

c = '+'

=

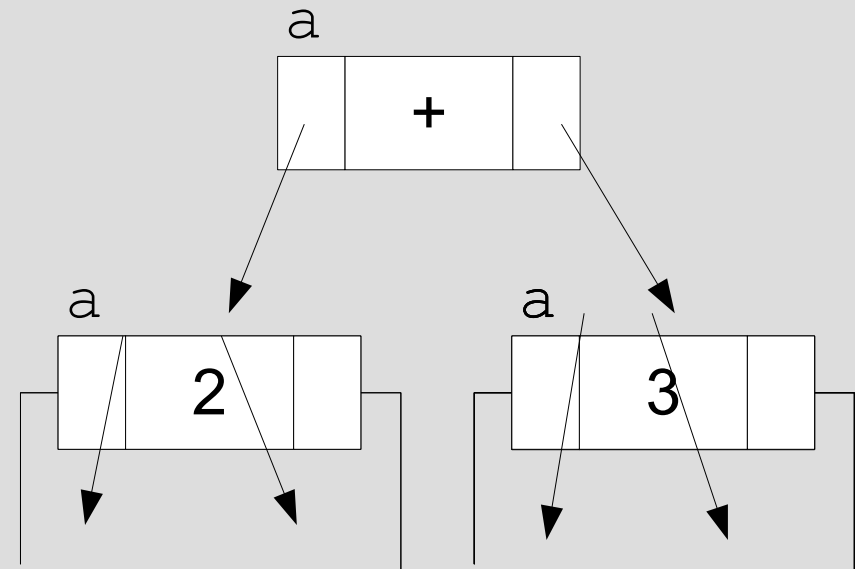


Árvore Binária

Funções Árvore Vazia e Imprime

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

```
void arv_imprime(Arv *a) {  
    if(!arv_vazia(a)) {  
        printf("%c ", a->info);  
        arv_imprime(a->esq);  
        arv_imprime(a->dir);  
    }  
}
```



+ 2 3

Árvore Binária

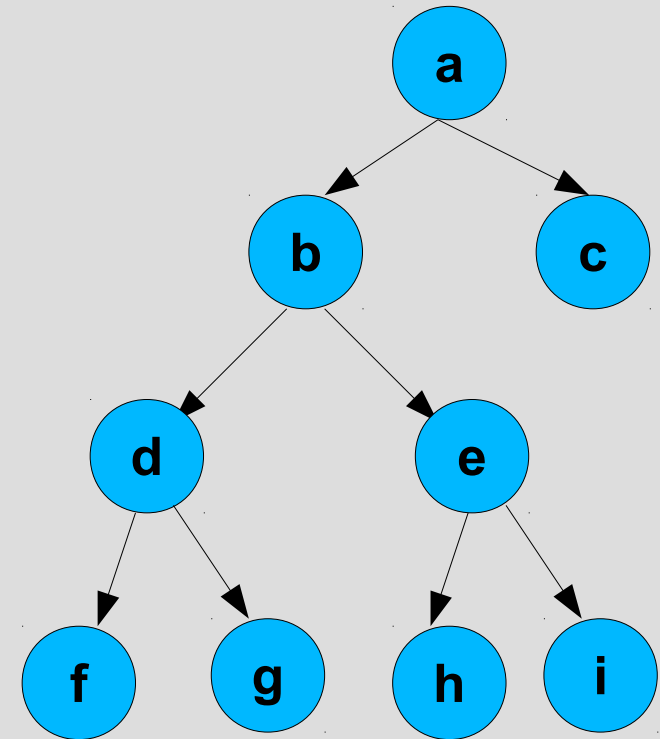
Funções Libera e Pertence

```
int arv_pertence(Arv *a, char c) {
    if(arv_vazia(a))
        return 0;
    else
        return a->info ==c || arv_pertence(a->esq,c)
                        || arv_pertence(a->dir,c) ;
}

void arv_libera(Arv *a) {
    if(!arv_vazia(a)) {
        arv_libera(a->esq) ;
        arv_libera(a->dir) ;
        free(a) ;
    }
}
```


Ordens de Percurso em Árvores Binárias

- Pré-Ordem:
 - Trata a raiz, percorre sae e percorre sad
 - Exemplo: Função Imprime
- Ordem Simétrica:
 - Percorre sae, trata a raiz, e percorre sad
 - Exemplo: Função de Busca
- Ordem Pós-Ordem:
 - Percorre sae, percorre sad e trata a raiz
 - Exemplo: Função Libera



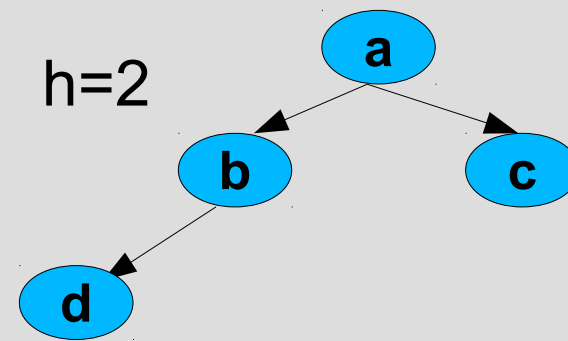
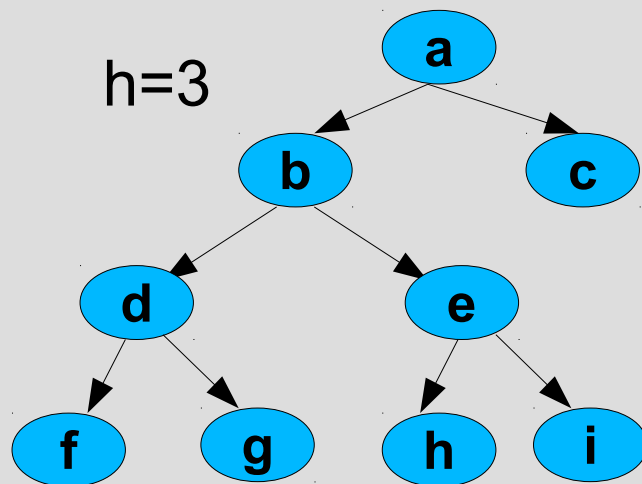
Pré-ordem: a b d f g e h i c

Simétrica: f d g b h e i a c

Pós-ordem: f g d h i e b c a

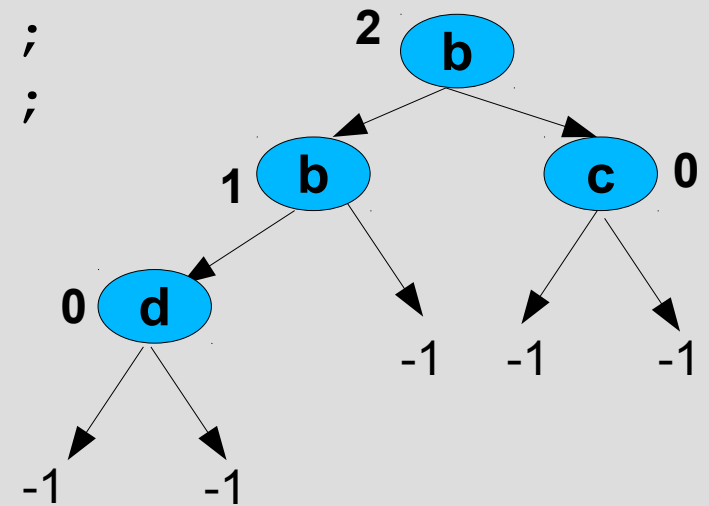
Árvores Binárias - Altura

- Propriedade Fundamental de Árvores:
 - Só existe um caminho da raiz para qualquer nó
- Altura de uma árvore
 - Comprimento do caminho mais longo da raiz até uma das folhas
 - A altura de uma árvore vazia é -1
 - A altura de uma árvore com um único nó raiz é 0



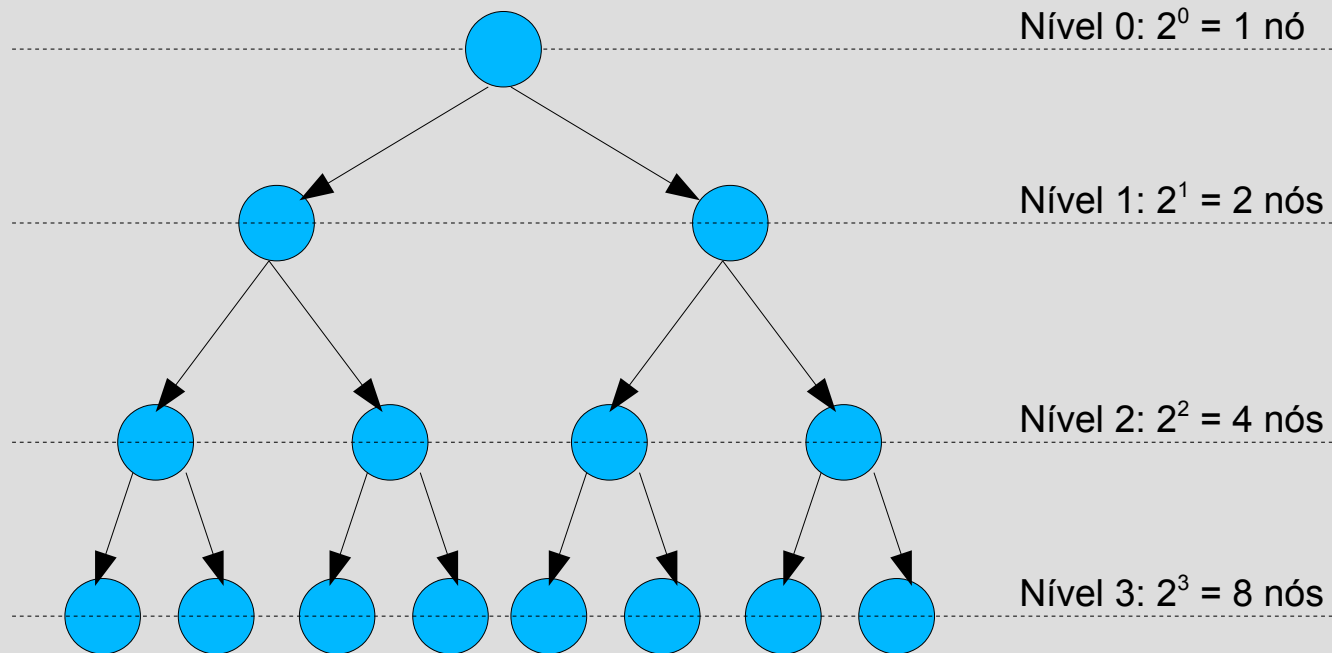
Árvore Binária - Altura

```
int arv_altura(Arv *a) {  
    if(arv_vazia(a))  
        return -1;  
    else{  
        int hSAE = arv_altura(a->esq);  
        int hSAD = arv_altura(a->dir);  
        if(hSAE > hSAD)  
            return 1+hSAE;  
        else  
            return 1+hSAD;  
    }  
}
```



Árvore Binária Completa

- Um Árvore Binária é dita **completa** (ou cheia) se todos os nós internos têm exatamente 2 filhos



Nível 0: $2^0 = 1$ nó

Nível 1: $2^1 = 2$ nós

Nível 2: $2^2 = 4$ nós

Nível 3: $2^3 = 8$ nós

O número n de nós de uma árvore de altura h é $n = 2^0 + 2^1 + 2^2 + \dots + 2^h = 2^{h+1} - 1$

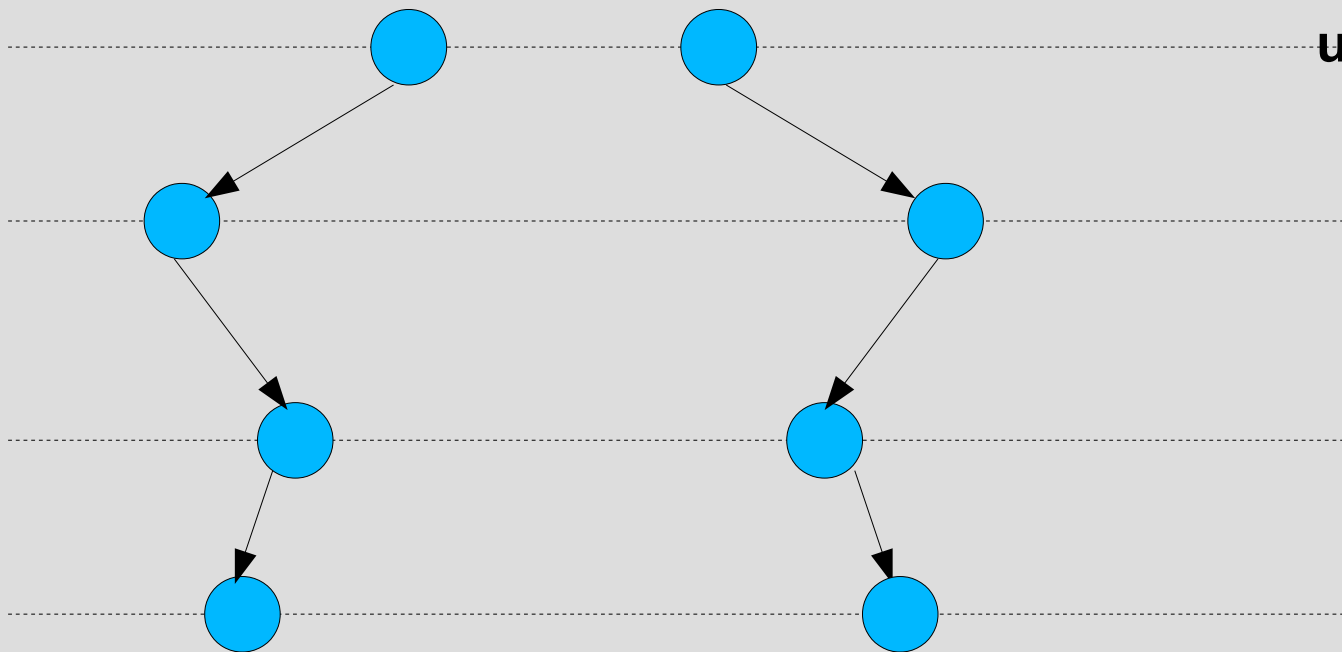
Seja h a altura de uma árvore binária completa. Dizemos que a árvore tem $O(2^h)$ nós e, para alcançar qualquer nó da árvore, temos complexidade de

$O(\log n)$

O número de nós $n = 2^0 + 2^1 + 2^2 + 2^3 = 15$

Árvore Binária Degenerada

- Um Árvore Binária é dita **degenerada** se todos os nós internos têm uma única sub-árvore



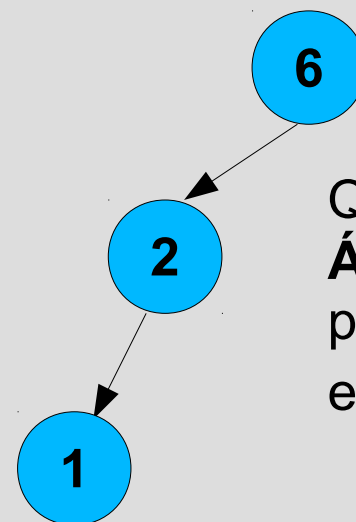
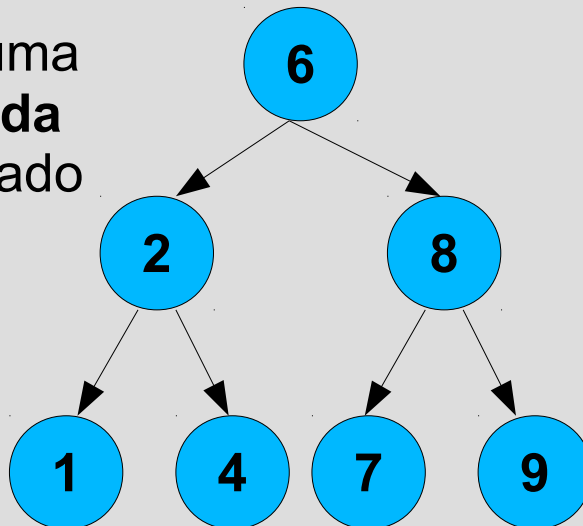
O número n de nós de uma árvore de altura h é $n = h+1$

Seja h a altura de uma árvore binária degenerada. Dizemos que a árvore tem $O(h)$ nós e, para alcançar qualquer nó da árvore, temos complexidade de **$O(n)$**

Árvore Binária de Busca

- Podemos utilizar uma árvore binária para realizarmos busca de forma eficiente desde que a árvore tenha a seguinte propriedade:
 - O valor associado a **raiz** é sempre **maior** do que qualquer valor da **sub-árvore da esquerda**;
 - O valor associado a **raiz** é sempre **menor** do que qualquer valor da **sub-árvore da direita**;

Qualquer nó em uma **Árvore Balanceada** pode ser encontrado em **$O(\log n)$**



Qualquer nó em uma **Árvore Degenerada** pode ser encontrado em **$O(n)$**

Tipo Abstrato de Dado Árvore Binária de Busca

Podemos criar um TAD Árvore Binária Busca de inteiros. Para tanto, devemos criar o arquivo arvb.h com o nome do tipo e os protótipos.

```
typedef struct arvb ArvB;
```

```
/*Função que cria uma Árvore Binária de Busca Vazia.*/
```

```
ArvB* arvb_cria_vazia(void);
```

```
/*Testa se uma Árvore Binária é vazia.*/
```

```
int arvb_vazia(ArvB *a);
```

```
/*Função que busca a sub-árvore que contém um inteiro.*/
```

```
ArvB* arvb_busca(ArvB *a, int c);
```

```
/*Função que imprime os elementos de uma Árvore.*/
```

```
void arvb_imprime(ArvB *a);
```

```
/*Função que insere um inteiro em uma Árvore.*/
```

```
ArvB* arvb_insere(ArvB *a, int c);
```

```
/*Função que remove um inteiro em uma Árvore.*/
```

```
ArvB* arvb_remove(ArvB *a, int c);
```

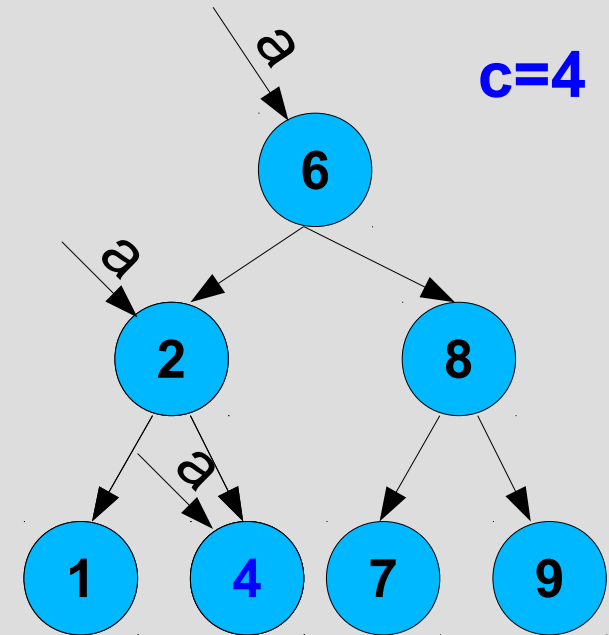
```
/*Libera o espaço alocado para uma Árvore.*/
```

```
void arvb_libera(ArvB *a);
```

Árvore Binária de Busca

Função Busca

```
ArvB* arvb_busca(ArvB *a, int c){  
    if(arvb_vazia(a))  
        return NULL;  
    else if(a->info < c)  
        return arvb_busca(a->dir,c);  
    else if(a->info > c)  
        return arvb_busca(a->esq,c);  
    else //(a->info == c)  
        return a;  
}
```

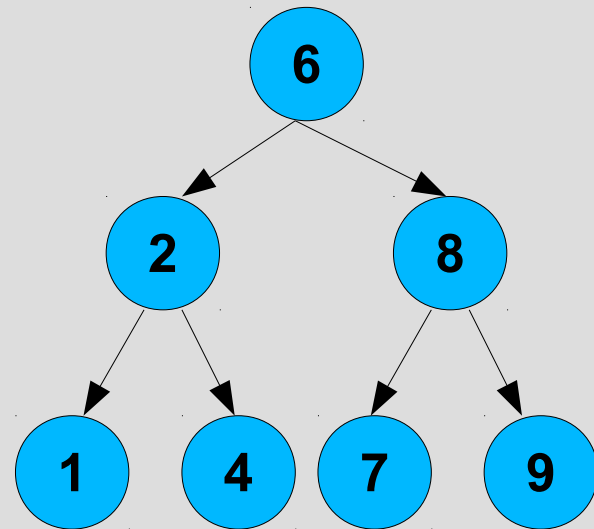


Árvore Binária de Busca

Função Imprime

- Basta utilizar a impressão na ordem simétrica (sae, raiz e sad)

```
void arvb_imprime(ArvB *a){  
    if(!arvb_vazia(a)){  
        arvb_imprime(a->esq);  
        printf("%d ",a->info);  
        arvb_imprime(a->dir);  
    }  
}
```

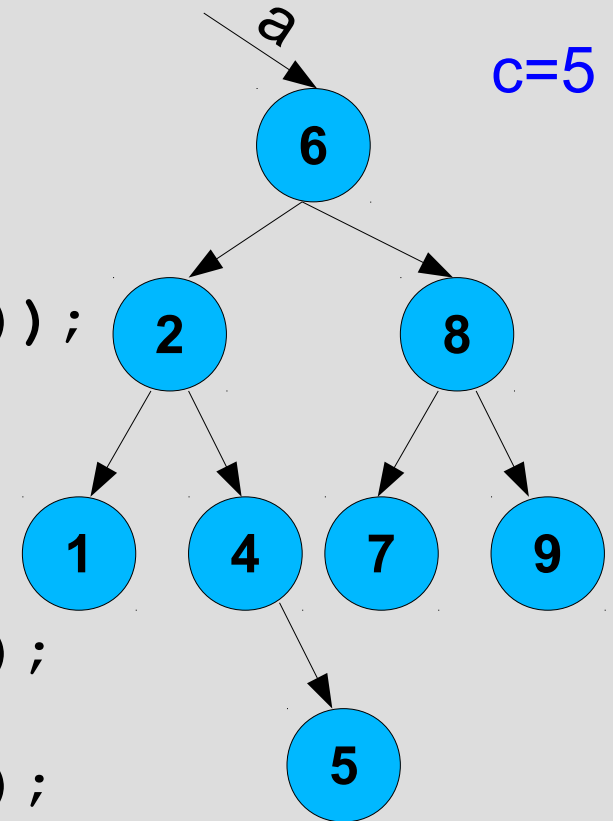


1 2 4 6 7 8 9

Árvore Binária de Busca

Função Insere

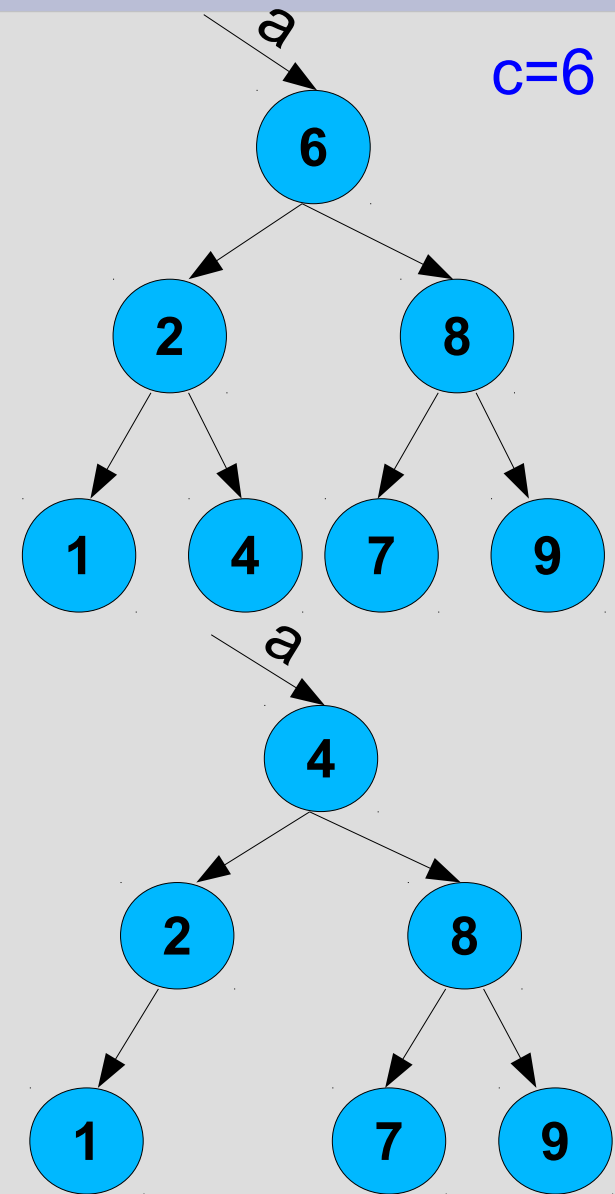
```
ArvB* arvb_insere(ArvB *a, int c){  
    if(arvb_vazia(a)){  
        a = (ArvB*)malloc(sizeof(ArvB));  
        a->info = c;  
        a->esq = NULL;  
        a->dir = NULL;  
    }else if(a->info > c)  
        a->esq = arvb_insere(a->esq,c);  
    else if (a->info < c)  
        a->dir = arvb_insere(a->dir,c);  
    else  
        printf("\nElemento Ja Pertence a Arvore");  
    return a;  
}
```



Árvore Binária de Busca

Função Remove

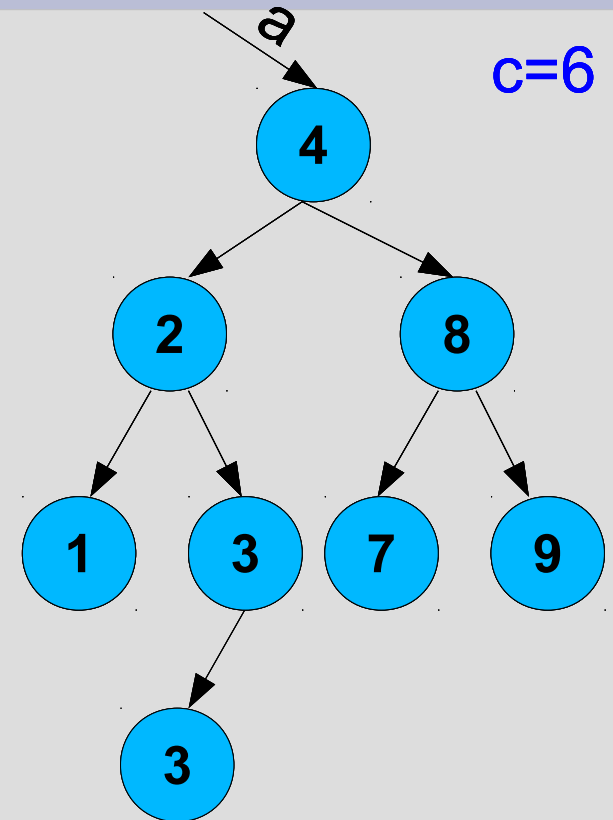
```
ArvB* arvb_remove(ArvB *a, int c){
    if(!arvb_vazia(a)){
        if(a->info > c)
            a->esq = arvb_remove(a->esq,c);
        else if (a->info < c)
            a->dir = arvb_remove(a->dir,c);
        else{
            ArvB* t;
            if (a->esq == NULL){
                t = a; a = a->dir;
                free(t);
            }else if (a->dir == NULL){
                t = a; a = a->esq;
                free(t);
            }else{
                t = a->esq;
                while(t->dir!=NULL)
                    t = t->dir;
                a->info = t->info; t->info = c;
                a->esq = arvb_remove(a->esq,c);
            }
        }
    }
    return a;
}
```



Árvore Binária de Busca

Função Remove

```
ArvB* arvb_remove(ArvB *a, int c){
    if(!arvb_vazia(a)){
        if(a->info > c)
            a->esq = arvb_remove(a->esq,c);
        else if (a->info < c)
            a->dir = arvb_remove(a->dir,c);
        else{
            ArvB* t;
            if (a->esq == NULL){
                t = a; a = a->dir;
                free(t);
            }else if (a->dir == NULL){
                t = a; a = a->esq;
                free(t);
            }else{
                t = a->esq;
                while(t->dir!=NULL)
                    t = t->dir;
                a->info = t->info; t->info = c;
                a->esq = arvb_remove(a->esq,c);
            }
        }
    }
    return a;
}
```



Slides baseados no livro **Introdução a Estruturas de Dados**, Waldemar Celes, Renato Cerqueira e José Lucas Rangel, Editora Campus, 2004.