

Estruturas de Dados 1

481440

Junho/2018

Mario Liziér
lazier@ufscar.br

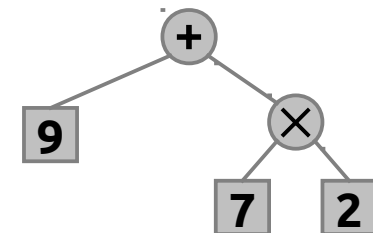
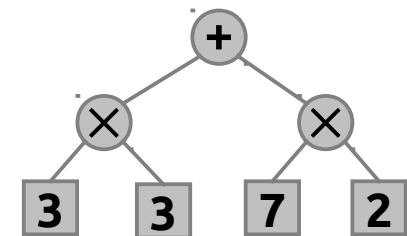
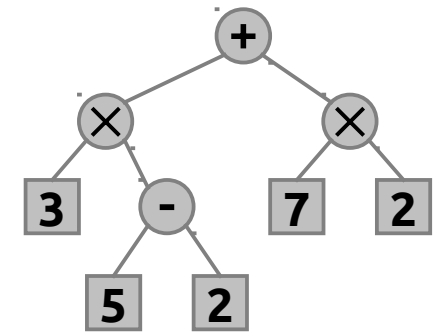
Percurso ou linearização

- Percorrer os elementos de uma árvore (linearmente)
 - Um por vez, sequencialmente
- Definição de uma ordem a partir de uma árvore
- Aplicações:
 - Indicar uma ordem de processamento sequencial dos elementos
 - Definir um percurso para um iterador

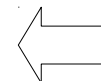
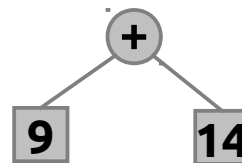
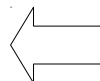
Percursos

- Como definir um percurso para resolvermos a expressão:

$$3*(5-2)+7*2$$

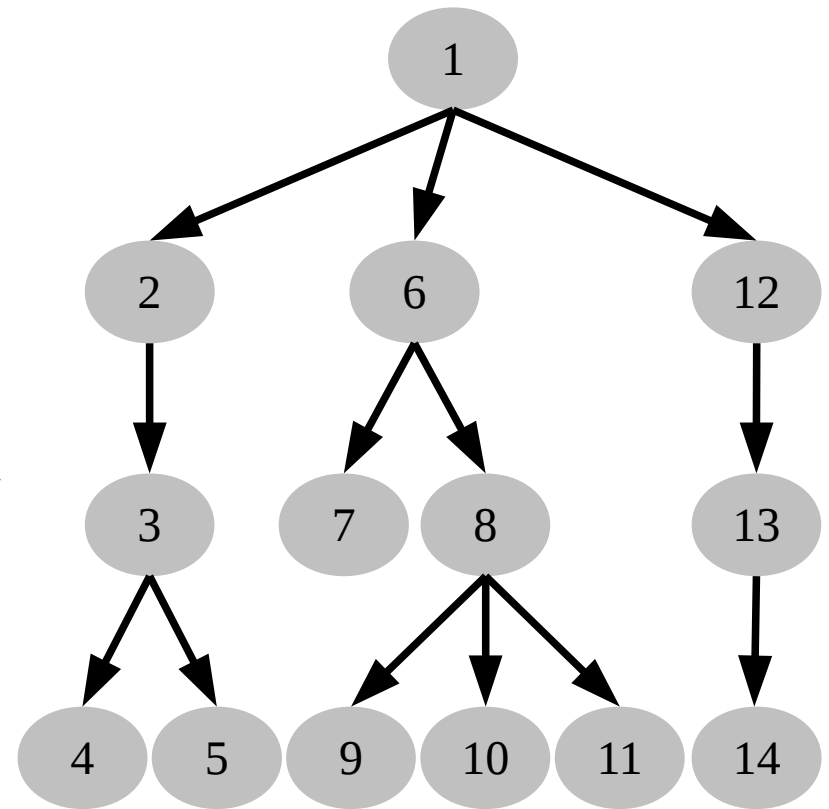


23



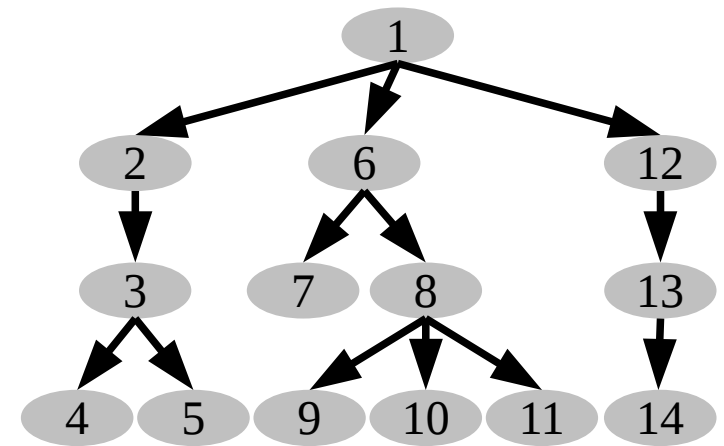
Percurso Pré-Ordem

- Processar o nó atual, antes de processar as suas sub-árvores
 - Podemos utilizar uma pilha (ou a pilha da recursão!)



Percurso Pré-Ordem

- Processar o nó atual, antes de processar as suas sub-árvores

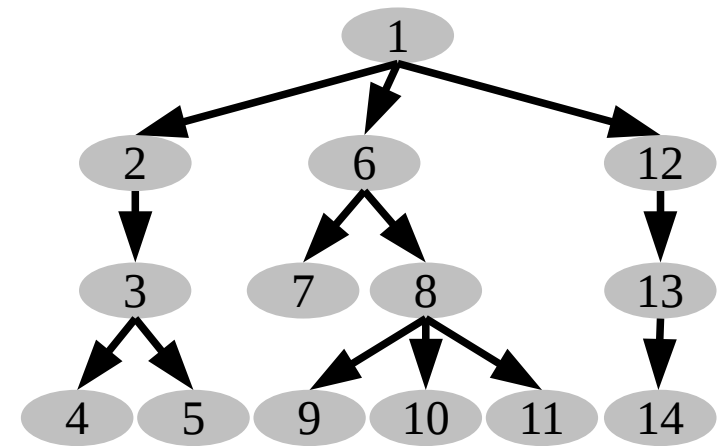


```
void pre_order( struct node *p ) {  
    process( p->data ); // printf("%d ", p->data);  
  
    for(Iterador it = primeiro(p->filhos); !acabou(it); proximo(it)) {  
        pre_order(elemento(it));  
    }  
  
    return;  
}
```

```
void pre_order( struct node *p ) {  
    process( p->data ); // printf("%d ", p->data);  
  
    for(int i = 0; i < p->nfilhos; i++) {  
        pre_order(p->filhos[i]);  
    }  
  
    return;  
}
```

Percurso Pré-Ordem

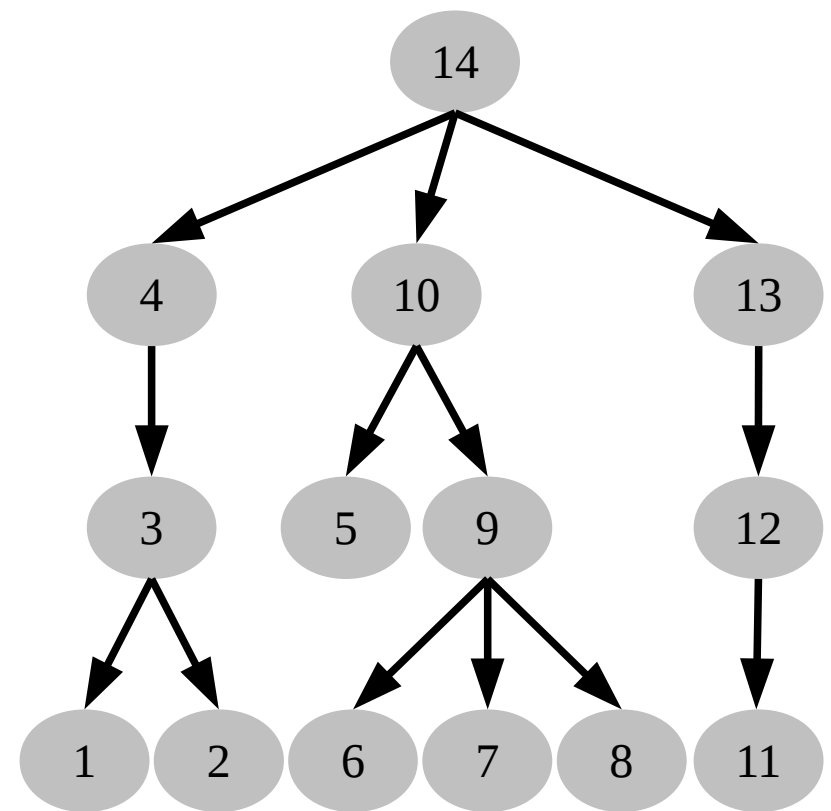
- Processar o nó atual, antes de processar as suas sub-árvores
 - E uma versão iterativa?



```
void pre_order( struct node *p ) {  
  
    return;  
}
```

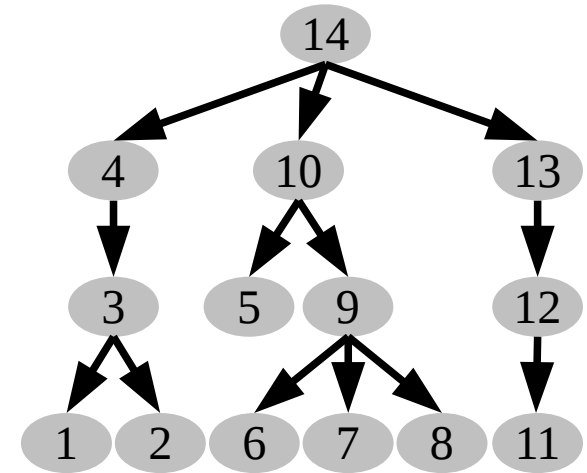
Percurso Pós-Ordem

- Processar o nó atual, depois de processar as suas sub-árvores
 - Também podemos utilizar uma pilha (ou a pilha da recursão!)



Percurso Pós-Ordem

- Processar o nó atual, depois de processar as suas sub-árvores

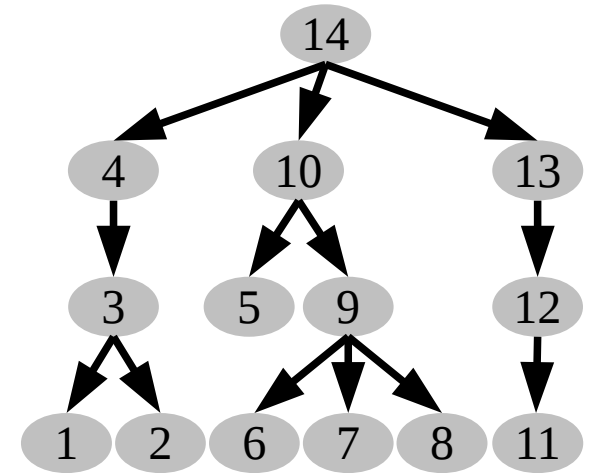


```
void pos_order( struct node *p ) {  
    for(Iterador it = primeiro(p->filhos); !acabou(it); proximo(it)) {  
        pos_order(elemento(it));  
    }  
  
    process( p->data ); // printf("%d ", p->data);  
  
    return;  
}
```

```
void pos_order( struct node *p ) {  
    for(int i = 0; i < p->nfilhos; i++) {  
        pos_order(p->filhos[i]);  
    }  
  
    process( p->data ); // printf("%d ", p->data);  
  
    return;  
}
```


Percurso Pós-Ordem

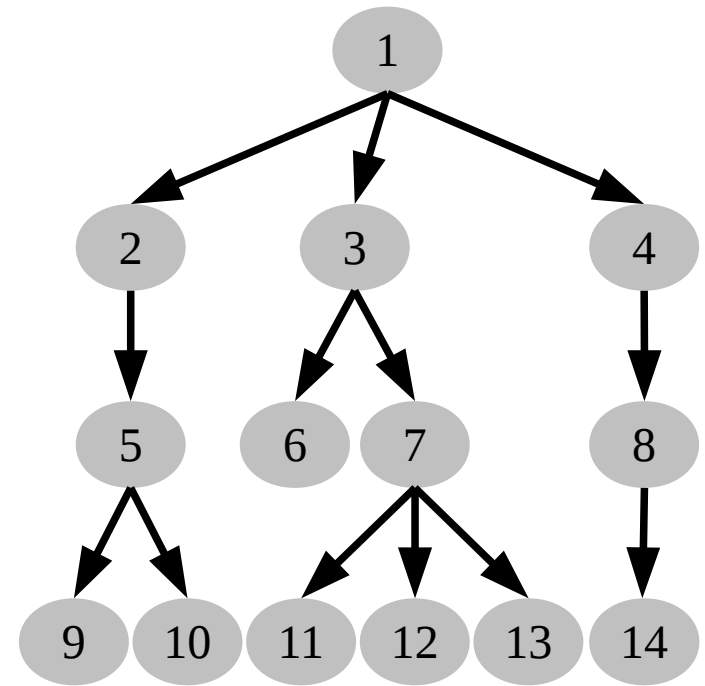
- Processar o nó atual, depois de processar as suas sub-árvores
 - E uma versão iterativa?



```
void pos_order( struct node *p ) {  
  
    return;  
}
```

Percurso Em-Nível

- Processar os nós de um nível de cada vez
 - Podemos utilizar uma fila!

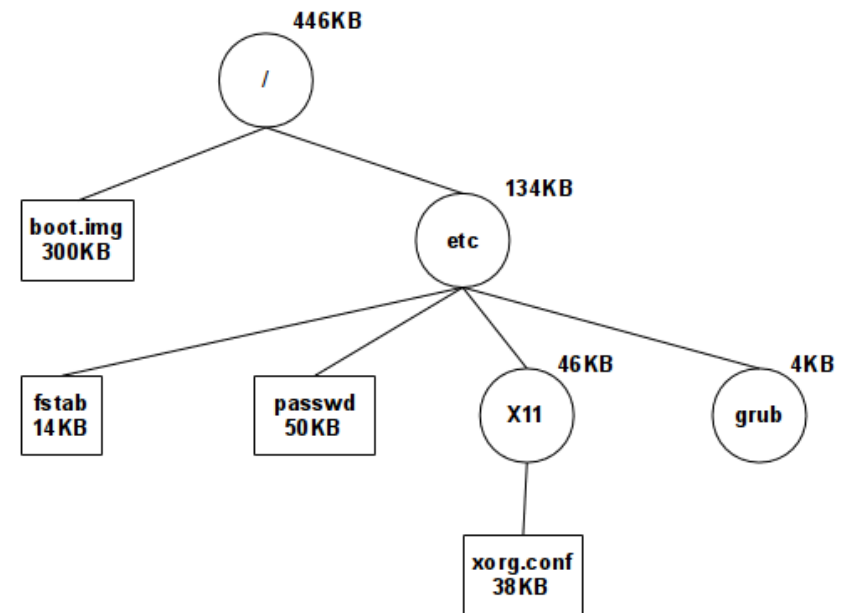


```
void em_nivel( struct node *p ) {  
  
  
  
  
  
  
  
  
  
    return;  
}
```

Exercício

Em geral, arquivos armazenados em disco são organizados hierarquicamente em estruturas chamadas “diretórios” e apresentadas ao usuário na forma de árvore. Considerando as afirmações abaixo, implemente um procedimento para calcular o tamanho ocupado por um diretório (incluindo os arquivos e diretórios que ele contém).

- Um diretório conter arquivos e/ou diretórios (ou estar vazio);
- A estrutura de um diretório ocupa 4KB;
- Cada referência para um arquivo ou diretório ocupa 4KB;
- Cada arquivo possui o seu tamanho armazenado no seu cabeçalho, sendo obtida pela função *size(arquivo)*;



Árvores Binárias

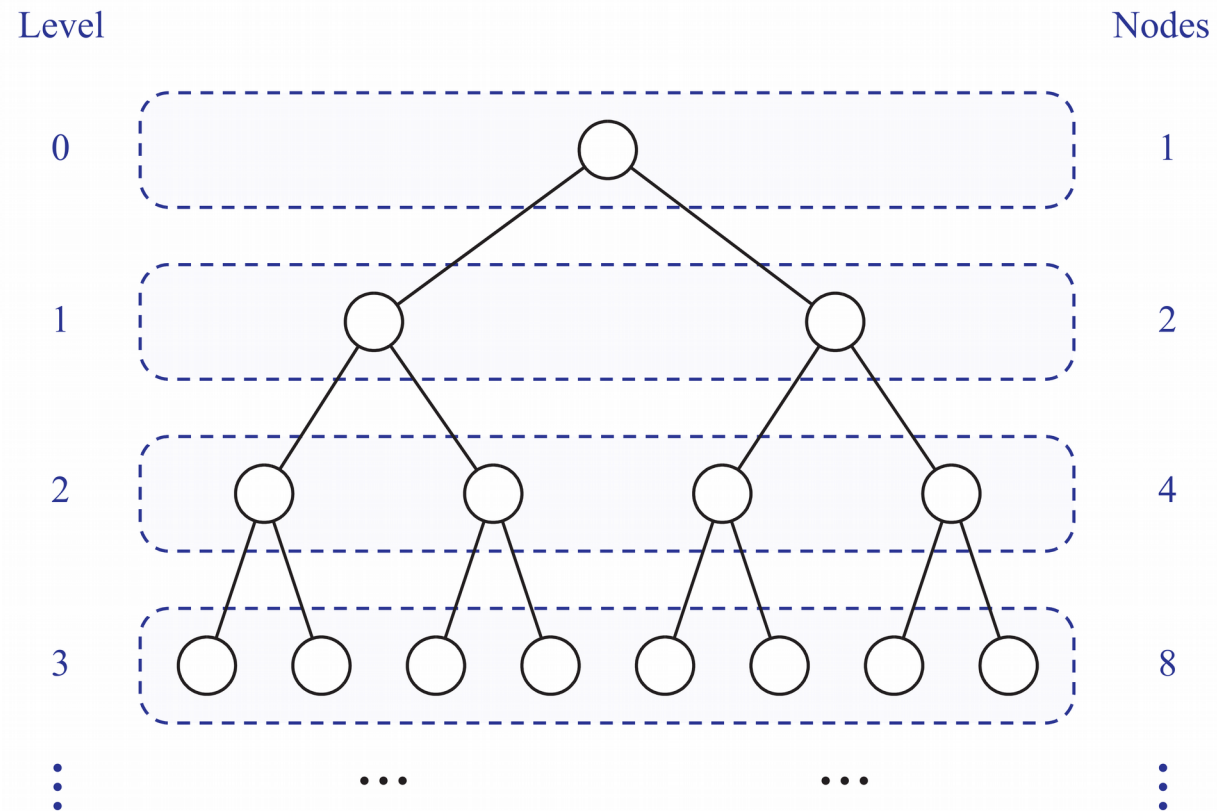
- No máximo 2 filhos (0, 1 ou 2 filhos)
- Árvore própria: apenas 0 ou 2 filhos
- Nomenclatura:
 - Filho da esquerda/direita
 - Subárvore da esquerda/direita

```
struct node {  
    T data;  
    struct node *esq, *dir;  
    struct node *pai; // opcional  
};
```

Árvores Binárias

- Seja T uma árvore binária não vazia, n , n_E , n_I e h , respectivamente, o número de elementos (nós), número de nós externos (folhas), número de nós internos e altura. Seguem as seguintes propriedades:

1. $h + 1 \leq n \leq 2^{h+1} - 1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n + 1) - 1 \leq h \leq n - 1$



Árvore Binária de Busca (ABB ou BST)

- Árvore binária
- Critério de ordenação:
 - A chave de cada nó é:
 - Maior que todas as chaves da sub-árvore à esquerda;
 - Menor que todas as chaves da sub-árvore à direita.

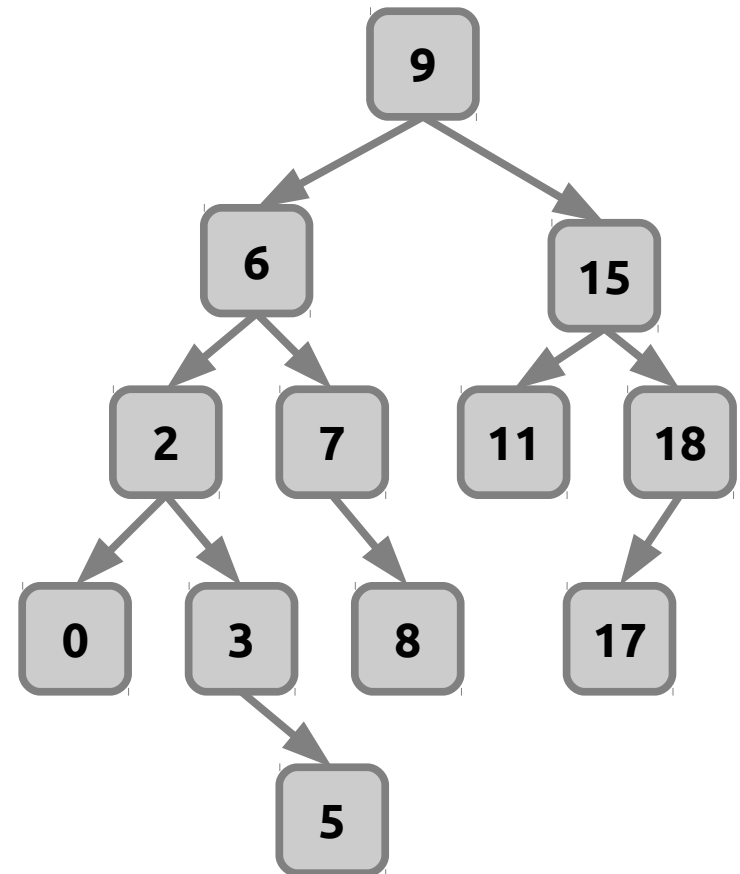
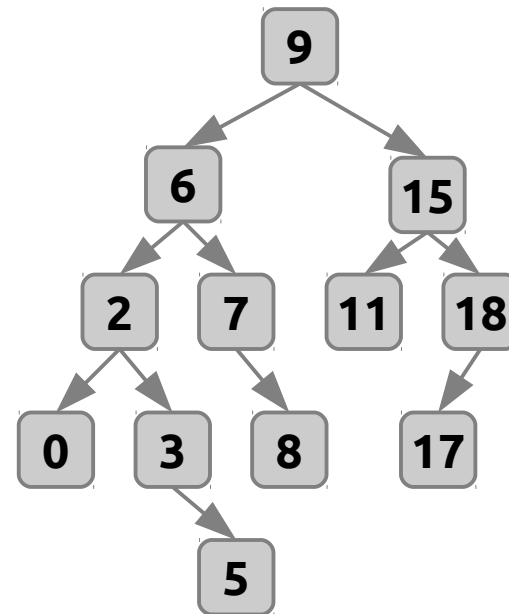


ABB - Busca

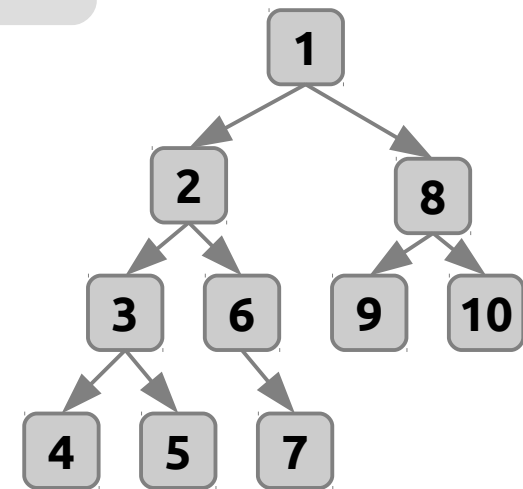
- Para buscar uma chave *key*, percorremos um caminho partindo da raiz para as folhas;
- O próximo nó a ser visitado depende do resultado da comparação
- Se alcançamos uma folha e esta não é a chave, significa que não encontramos

```
struct node* search( struct node *p, K key) {  
    if( n == NULL )  
        return NULL;  
    else if( key < p->data->key )  
        return search( p->esq, key);  
    else if( key > p->data->key )  
        return search( p->dir, key);  
    else  
        return p;  
}
```



Percurso: Pré-Ordem

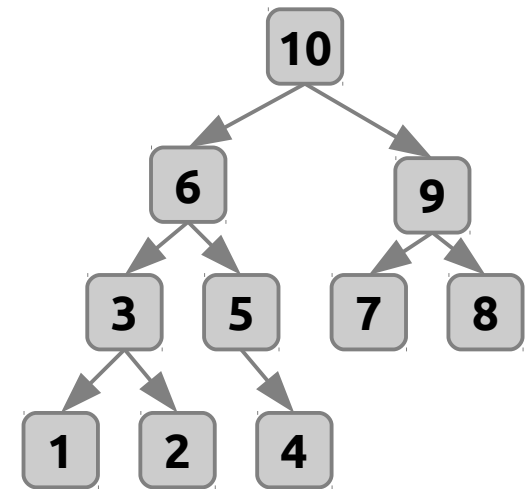
```
void pre_order( struct node *p ) {  
    process( p->data ); // printf("%d ", p->data);  
  
    if( p->esq )  
        pre_order(p->esq);  
  
    if( p->dir )  
        pre_order(p->dir);  
  
    return;  
}
```



* Ordem do percurso

Percurso: Pós-Ordem

```
void pos_order( struct node *p ) {  
    if( p->esq )  
        pre_order(p->esq);  
  
    if( p->dir )  
        pre_order(p->dir);  
  
    process( p->data ); // printf("%d ", p->data);  
  
    return;  
}
```

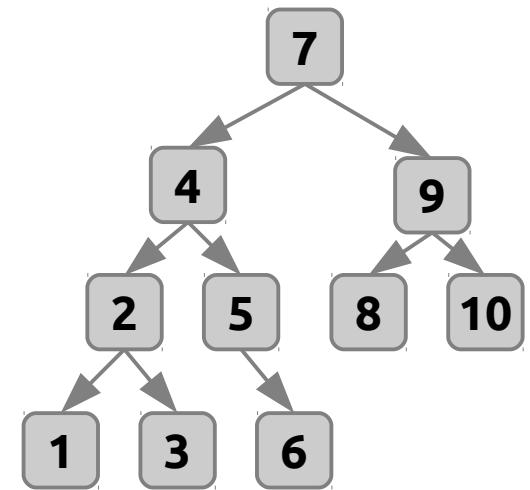


* Ordem do percurso

Percurso: Em-Ordem

```
void em_order( struct node *p ) {  
    if( p->esq )  
        pre_order(p->esq);  
    process( p->data ); // printf("%d ", p->data);  
    if( p->dir )  
        pre_order(p->dir);  
    return;  
}
```

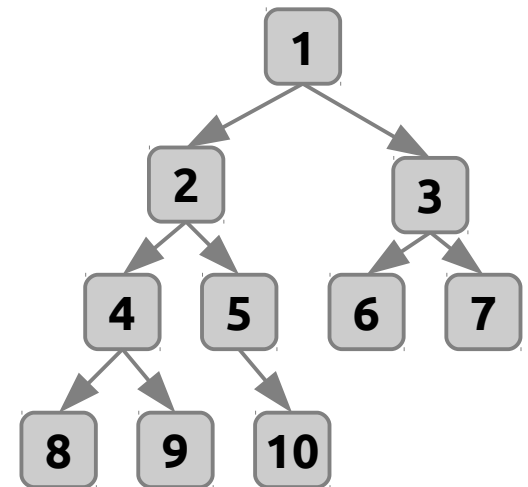
Veja como fica em uma ABB!
Em ordem!



* Ordem do percurso

Percurso: Em-Nível

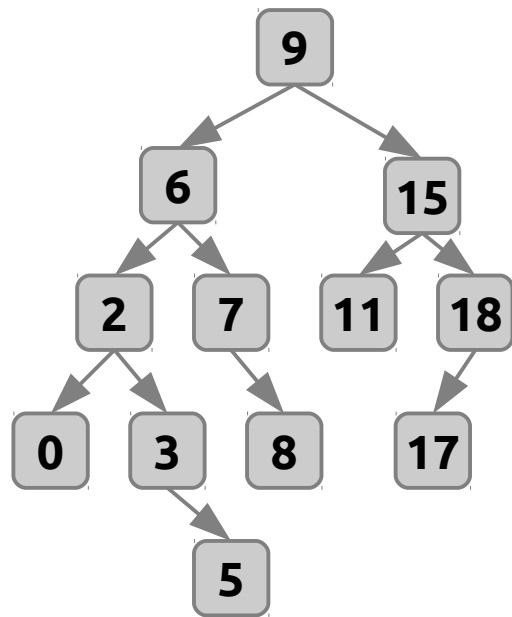
```
void em_nivel( struct node *p ) {  
  
    return;  
}
```



* Ordem do percurso

ABB - Inserção

- Procuramos pela posição adequada e inserimos assim que a encontrarmos
- E os elementos repetidos?



insert(4)

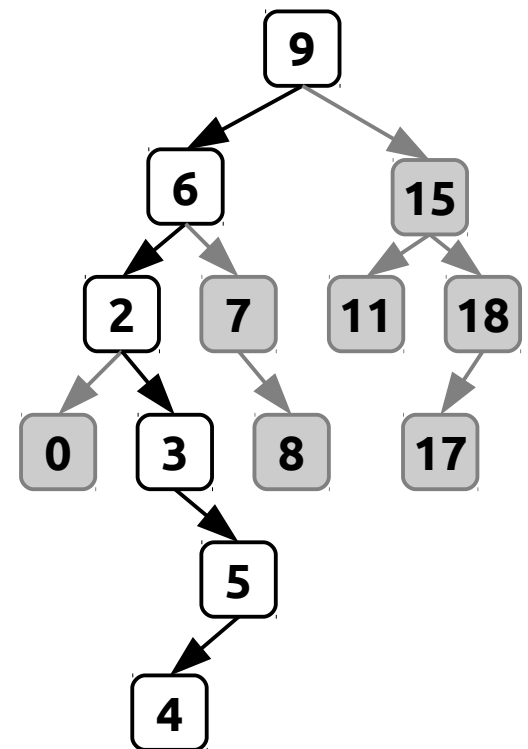


ABB - Inserção

```
typedef struct {
    struct node* raiz;
} ABB;

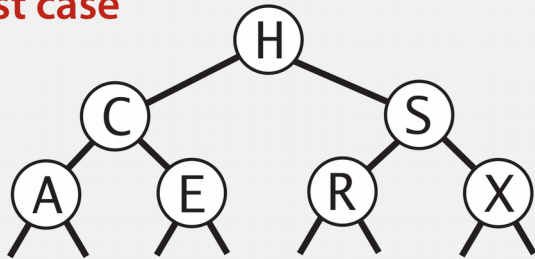
void inserir( ABB* arvore, T dados ) {
    arvore->raiz = inserir_privado( arvore->raiz, dados );
}

struct node* inserir_privado( struct node* p, T dados ) {
    if( !p ) {
        struct node* novo = (struct node*)malloc(sizeof(struct node));
        novo->data = dados;
        novo->esq = novo->dir = 0;
        return novo;
    } else if (dados->key < p->data->key ) {
        p->esq = inserir_privado( p->esq, dados);
    } else if (dados->key > p->data->key ) {
        p->dir = inserir_privado( p->dir, dados);
    } else { //repetido
        p->data = dados; // atualizando outros campos
    }
    return p;
}
```

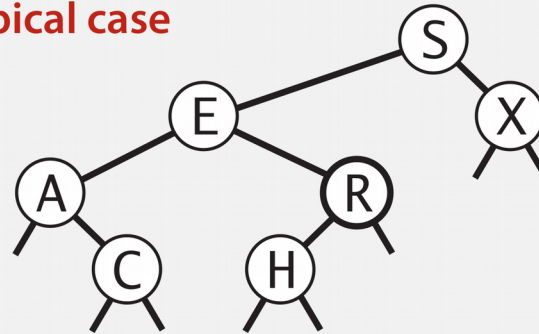
Exercício

- Vamos inserir cada conjunto de elementos, na ordem indicada, em uma ABB vazia:
 - Conjunto 1:
 - 6, 4, 9, 3, 7, 10, 5 e 8
 - Conjunto 2:
 - 3, 4, 5, 6, 7, 8, 9 e 10

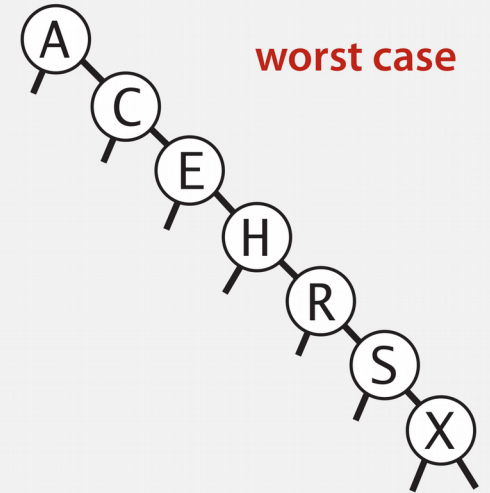
best case



typical case



worst case

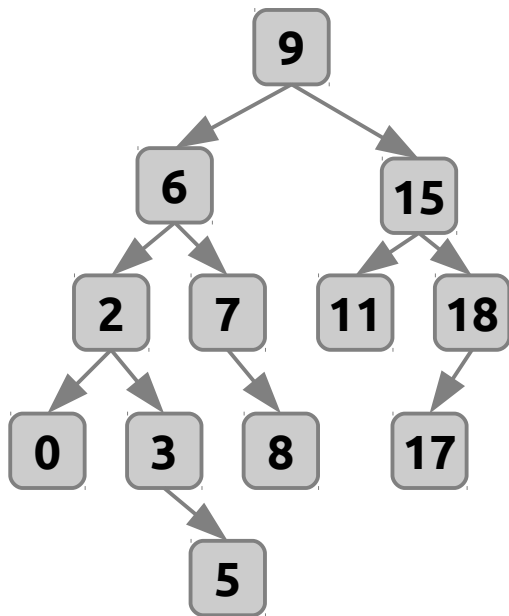


Remark. Tree shape depends on order of insertion.

Custos!

- Busca:
 - Melhor caso: $\log N$
 - Pior caso: N \leftarrow Isso é ruim!!!
- Inserção e remoção:
 - Custos idênticos a busca
 - Precisamos melhorar isso!
 - Este algoritmo de inserção pode deixar a árvore desbalanceada, dependendo da ordem de inserção !

ABB – Busca não-recursiva



```
int search( ABB *arvore, K key) {  
    return search( arvore->raiz, key) != NULL;  
}  
  
struct node* search( struct node *p, K key) {  
    while( p ) {  
        if( key < p->data->key )  
            p = p->esquerda;  
        else if( key > p->data->key )  
            p = p->direita;  
        else // key == p->data->key  
            return p;  
    }  
    return p;  
}
```

ABB - Remoção

- Vimos como inserir um elemento em uma árvore binária de busca (algoritmo simples, sem qualquer estratégia de balanceamento)
- Como removemos um elemento?
 - Não vamos nos preocupar agora com o balanceamento
 - Exemplo:
 - Remova o número 5
 - Remova o número 7
 - Remova o número 15

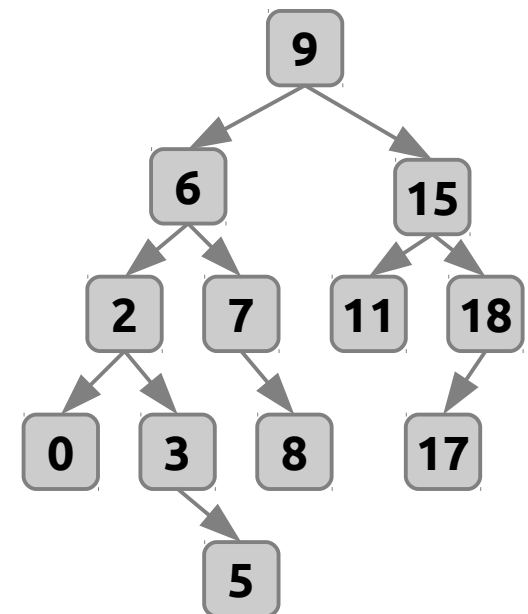
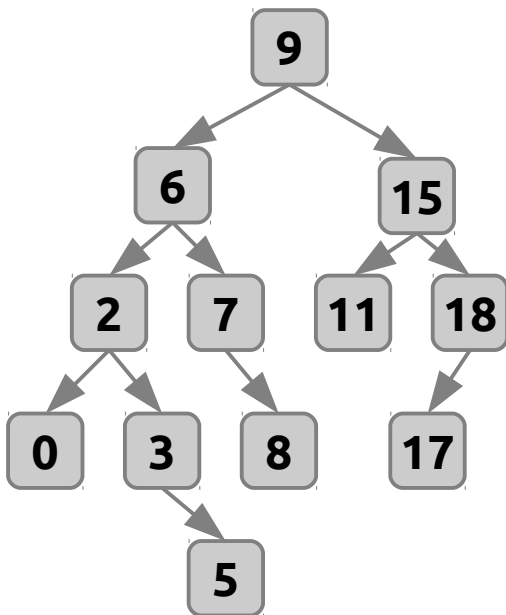


ABB - Remoção

- Caso 1: nó sem filhos



remove(raiz, 11)

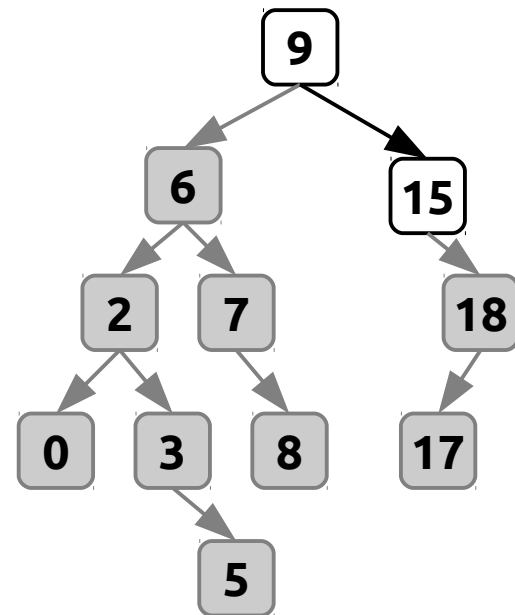
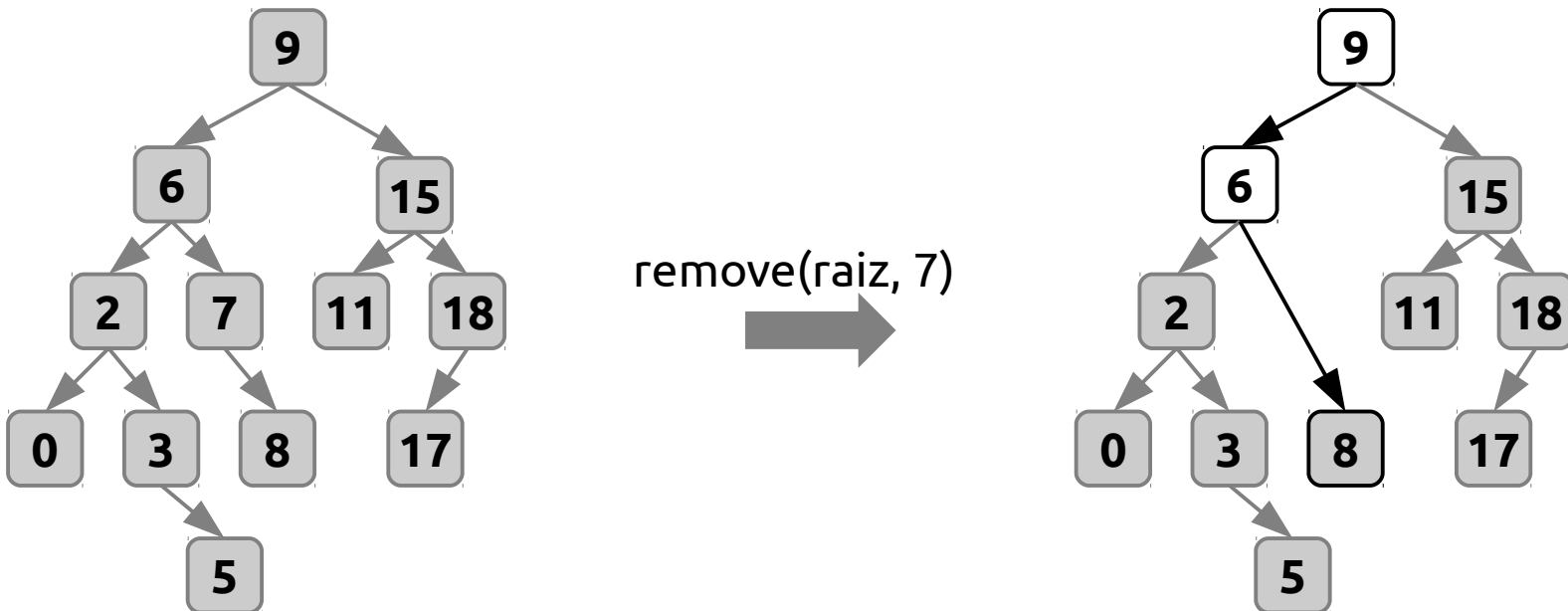


ABB - Remoção

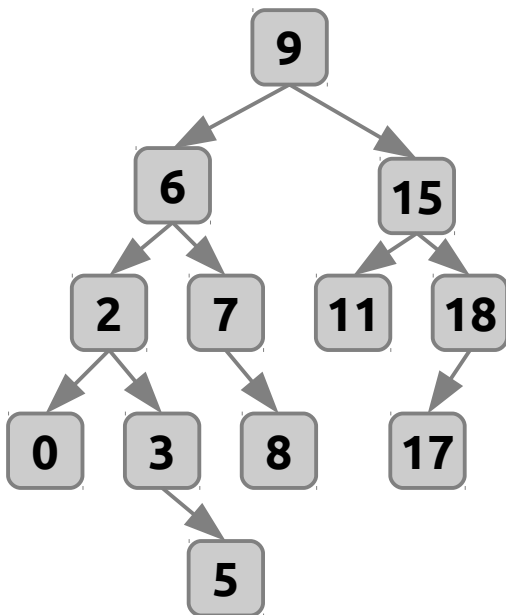
- Caso 2: nó com apenas um filho



Funciona com qualquer filho único? (esquerda ou direita)

ABB - Remoção

- Caso 3: nó com dois filhos



remove(raiz, 6)

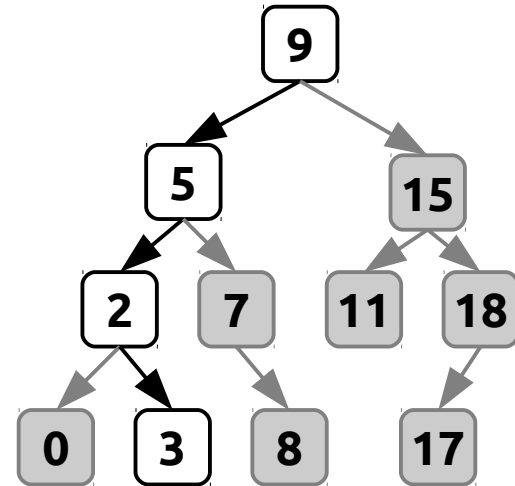



ABB - Máximo

- Encontrando o maior elemento de uma ABB:

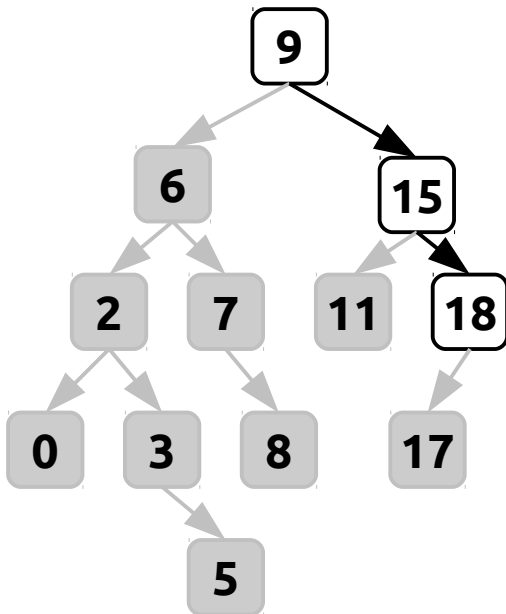
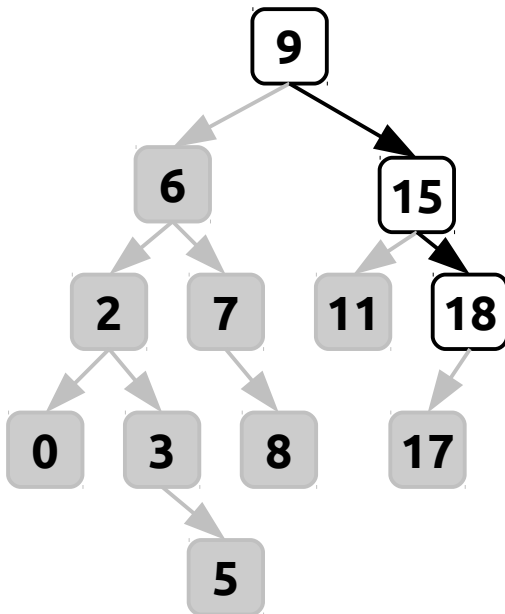


ABB - Máximo

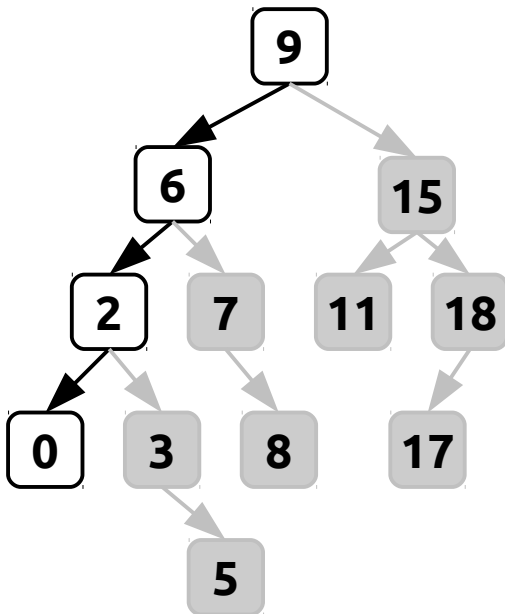
- Encontrando o maior elemento de uma ABB:



```
T max(ABB* arvore) {  
    assert(!vazia(arvore));  
    return max_node(arvore->raiz)->data;  
}  
  
struct node* max_node( struct node* p ) {  
    if( ! p->dir )  
        return p;  
    else  
        return max_node( p->dir );  
}
```

ABB - Mínimo

- Encontrando o menor elemento de uma ABB:



```
T min(ABB* arvore) {  
    assert(!vazia(arvore));  
    return min_node(arvore->raiz)->data;  
}  
  
struct node* min_node( struct node* p ) {  
    if( ! p->esq )  
        return p;  
    else  
        return min_node( p->esq );  
}
```


ABB - Remoção

- Remoção completa:
 - Caso 1: nó sem filhos
 - Traquilo!
 - Caso 2: nó com apenas um filho
 - Substituir pela sub-árvore existente
 - Caso 3: nó com dois filhos
 - Substituir pelo max/min das sub-árvores, e removê-lo em seguida

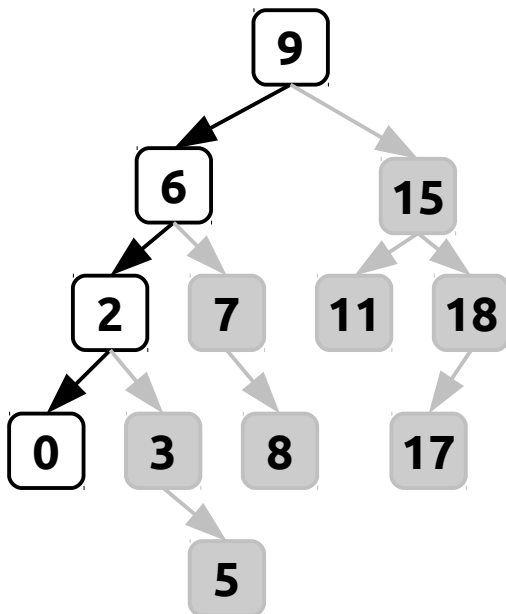
ABB - Remoção

```
void remove( ABB* arvore, K key ) {
    arvore->raiz = remove_no(arvore->raiz, key);
}

struct node* remove_no( struct node *p, K key ) {
    if( !p )
        return 0;
    if( key < p->data->key )
        p->esq = remove_no(p->esq, key);
    else if( key > p->data->key )
        p->dir = remove_no(p->dir, key);
    else {
        if( !p->dir ) {
            struct node *esquerda = p->esq;
            delete p;
            return esquerda;
        }
        if( !p->esq ) {
            struct node *direita = p->dir;
            delete p;
            return direita;
        }
        p->data = max_no(p->esq)->data;
        p->esq = remove_no( p->esq, p->data->key );
    }
    return p;
}
```

ABB – Remover Min/Max

- Remover o menor/maior elemento



```
void deleteMin(ABB* arvore) {
    arvore->raiz = deleteMin_no(arvore->raiz);
}

struct node* deleteMin_no(struct node* p) {
    if (!p->esq) {
        struct node* direita = p->dir;
        delete p;
        return direita;
    } else {
        p->esq = deleteMin_no( p->esq );
        return p;
    }
}
```

ABB – Chão (*floor*)

- Busca pela maior chave menor ou igual a uma chave
 - $\text{floor}(4) = 3$
 - $\text{floor}(10) = 9$
 - $\text{floor}(7) = 7$
 - $\text{floor}(-10) = \text{não tem}$
 - $\text{floor}(70) = 18$

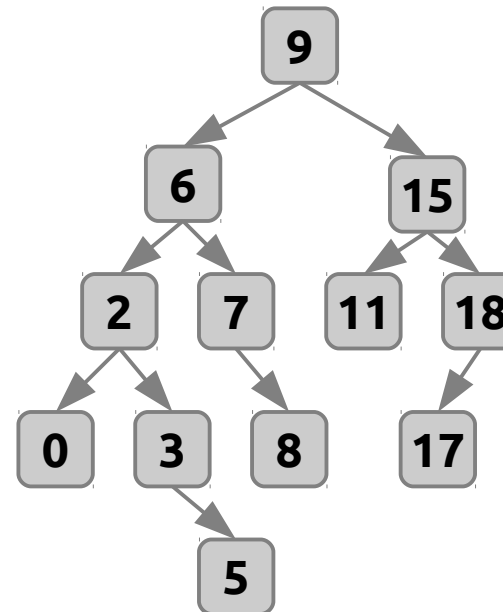


ABB – Chão (*floor*)

- Busca pela maior chave menor ou igual a uma chave

```
int floor(ABB* arvore, K key, T &f) {  
    struct node* p = floor_no(arvore->raiz, key);  
    if (!p)  
        return 0;  
    else {  
        f = p->data;  
        return 1;  
    }  
}  
  
struct node* floor_no(struct node* p, K key) {  
    if (!p)  
        return p;  
    if( key < p->data->key )  
        return floor_no(p->esq, key);  
    else if( key > p->data->key ) {  
        struct node* t = floor_no(p->dir, key);  
        if(t) return t;  
        else return p;  
    } else  
        return p;  
}
```

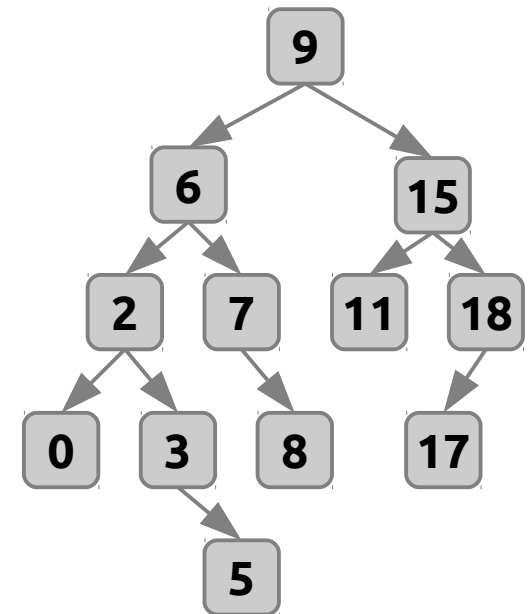


ABB – Teto (*ceiling*)

- Busca pela menor chave maior ou igual a uma chave
 - $\text{ceiling}(4) = 5$
 - $\text{ceiling}(10) = 11$
 - $\text{ceiling}(7) = 7$
 - $\text{ceiling}(-10) = 0$
 - $\text{ceiling}(70) = \text{não tem}$

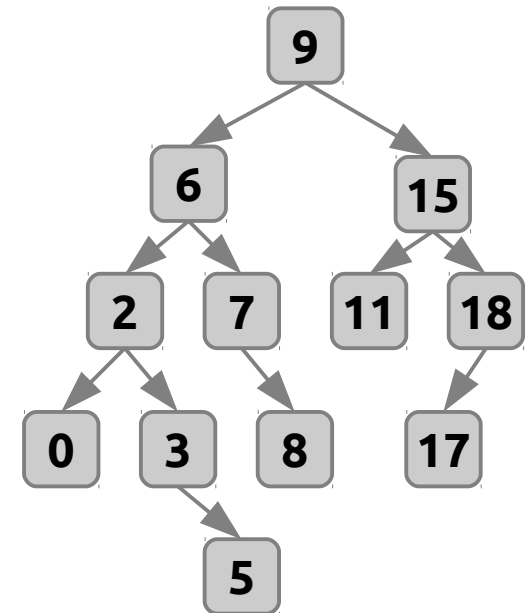
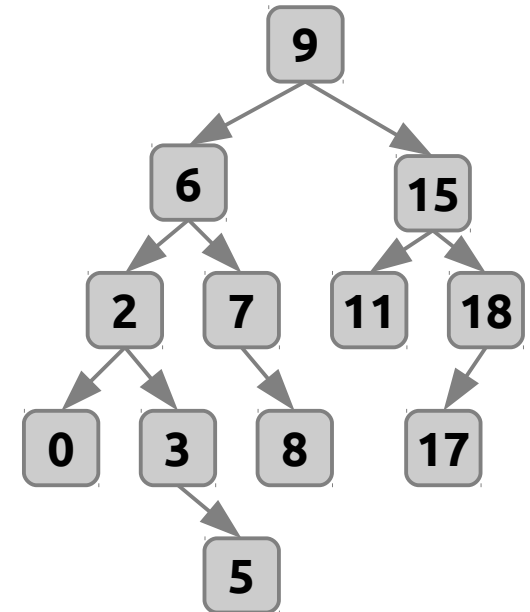


ABB – Teto (*ceiling*)

- Busca pela menor chave maior ou igual a uma chave

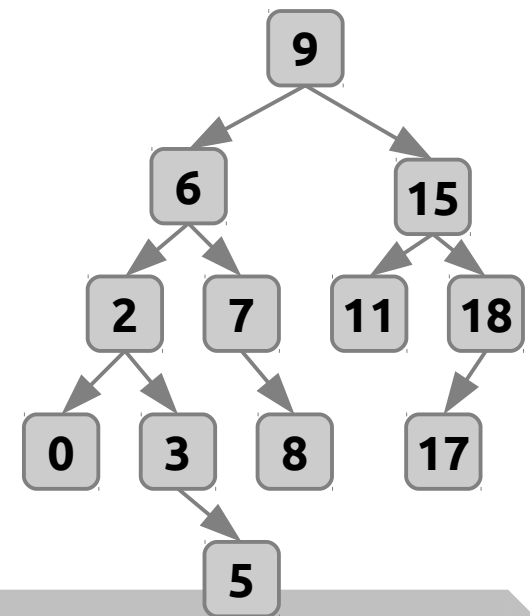
```
int ceiling(ABB* arvore, K key, T &f) {  
    struct node* p = ceiling_no(arvore->raiz, key);  
    if (!p)  
        return 0;  
    else {  
        f = p->data;  
        return 1;  
    }  
}
```

```
struct node* ceiling_no(struct node* p, K key) {  
    if (!p)  
        return p;  
    if( key > p->data->key )  
        return ceiling_no(p->dir, key);  
    else if( key < p->data->key ) {  
        struct node* t = ceiling_no(p->esq, key);  
        if(t) return t;  
        else return p;  
    } else  
        return p;  
}
```



Árvores – Tamanho (*Size*)

- Número de elementos em uma árvore/sub-árvore
 - Calcular (percorrer os nós e contar)
 - Não armazena um campo adicional por nó
 - Computacionalmente caro!
 - Armazenar em cada nó o número de elementos da sub-árvore em que o nó é raiz

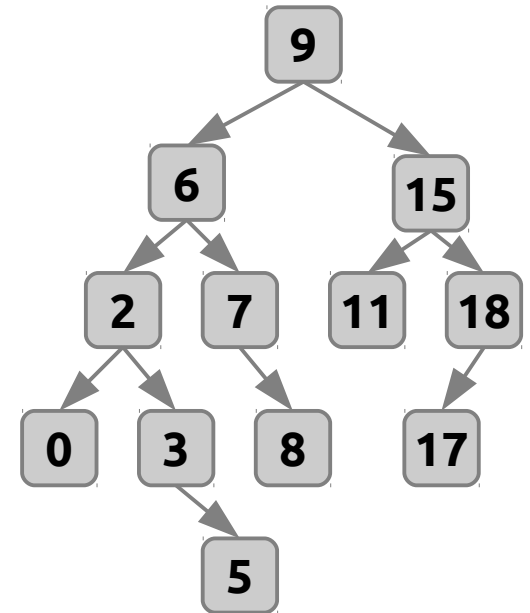


Árvores – Tamanho (*Size*)

- Percorrendo os nós e contando (sem armazenar)

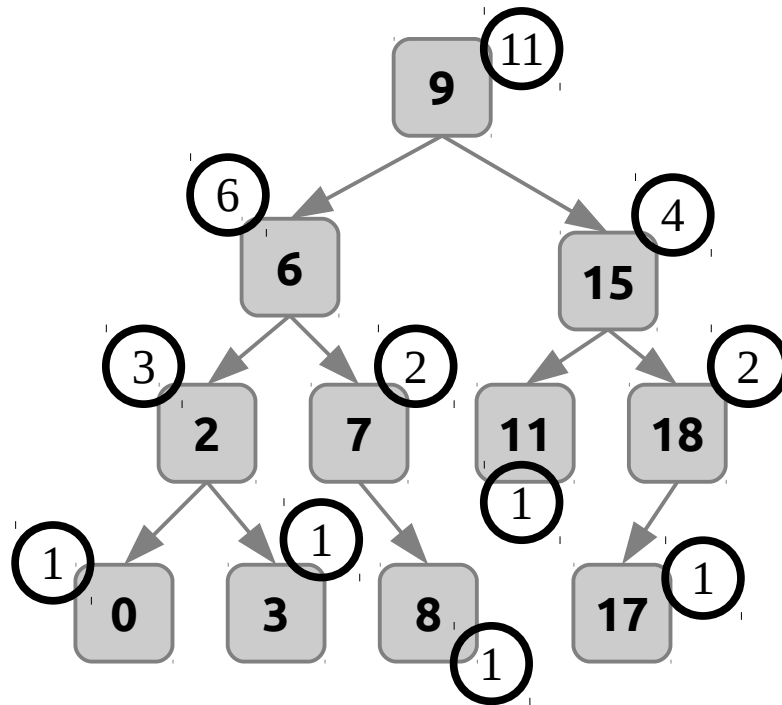
```
int size(ABB* arvore) {  
    return size_privado(arvore->raiz);  
}  
  
int size_privado(struct node* p) {  
    if (!p) return 0;  
    return size(p->esq) + size(p->dir) + 1;  
}
```

Percurso pós-ordem!

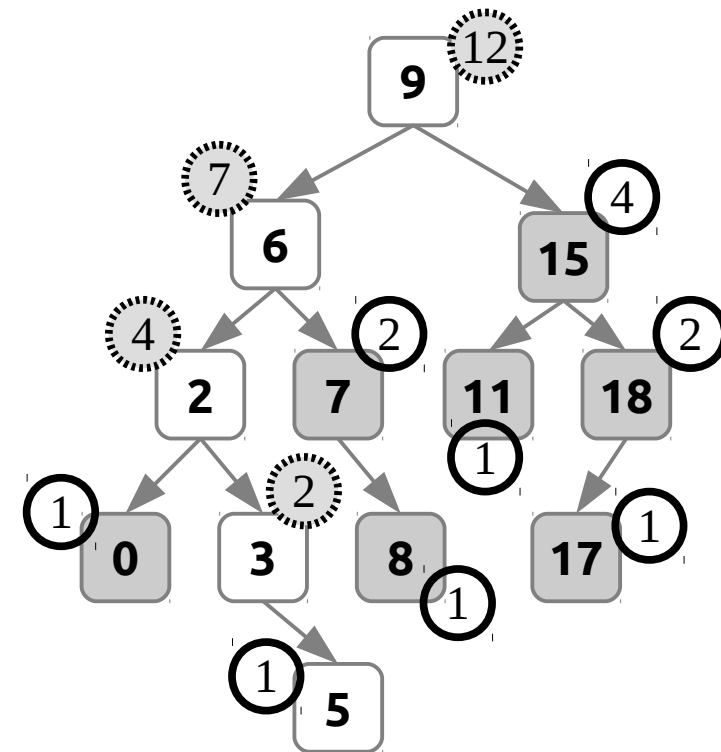


Árvores – Tamanho (*Size*)

- Inserção com campo size

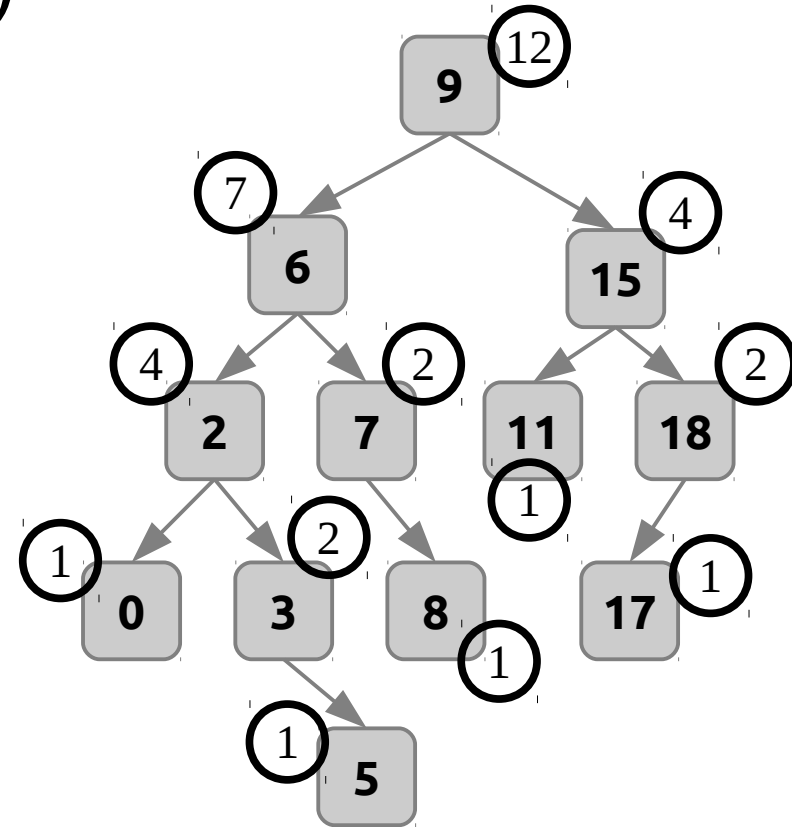


insert(5)



Árvores – Tamanho (*Size*)

- Armazenar em cada nó o número de elementos da sub-árvore:
 - Folhas com o valor 1
 - Em cada operação de inserção incrementamos todo caminho raiz-nó
 - Em cada operação de remoção decrementamos todo caminho raiz-nó

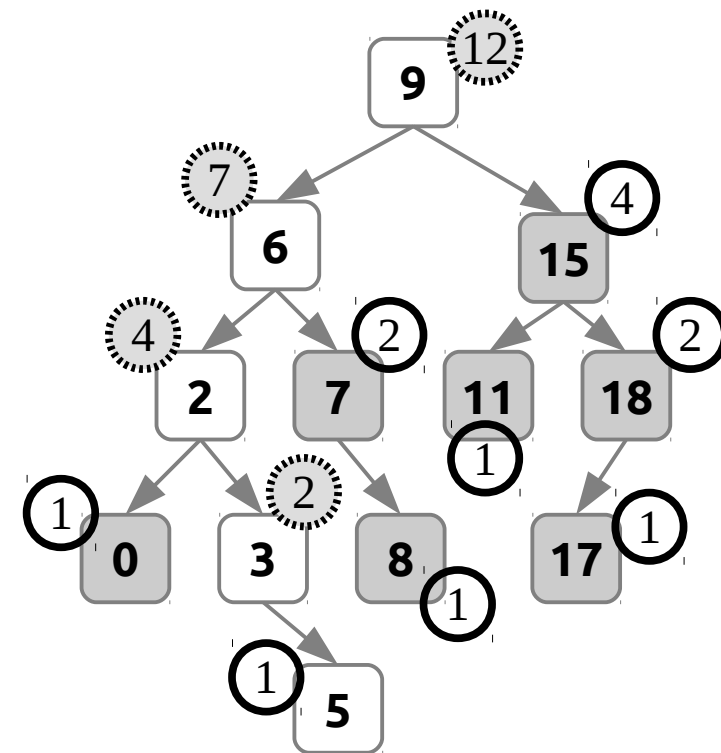


```
struct node {  
    T data;  
    struct node *esq, *dir;  
    struct node *pai; // opcional  
    int size;  
};
```

Árvores – Tamanho (*Size*)

- Inserção com campo size

insert(5)



Árvores – Tamanho (*Size*)

- Inserção com campo size

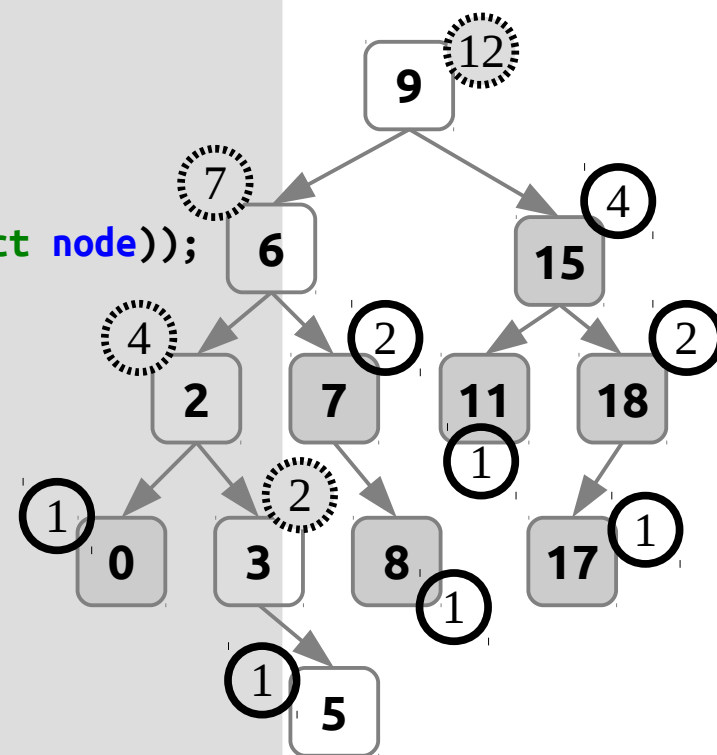
```
int size(struct node* p) {
    if (!p) return 0;
    else return p->size;
}

struct node* inserir_privado( struct node* p, T dados ) {
    if( !p ) {
        struct node* novo = (struct node*)malloc(sizeof(struct node));
        novo->data = dados;
        novo->esq = novo->dir = 0;
        novo->size = 1;
        return novo;
    } else if (dados->key < p->data->key ) {
        p->esq = inserir_privado( p->esq, dados);
    } else if (dados->key > p->data->key ) {
        p->dir = inserir_privado( p->dir, dados);
    } else { //repetido
        p->data = dados; // atualizando outros campos
    }

    p->size = size(p->esq) + size(p->dir) + 1;

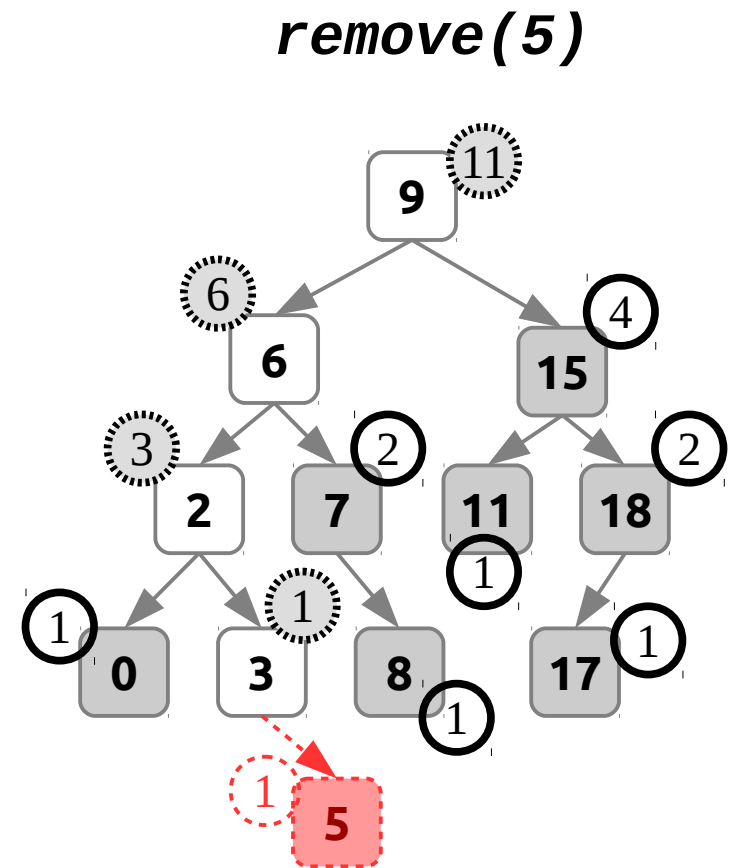
    return p;
}
```

insert(5)



Árvores – Tamanho (*Size*)

- Remoção com campo size



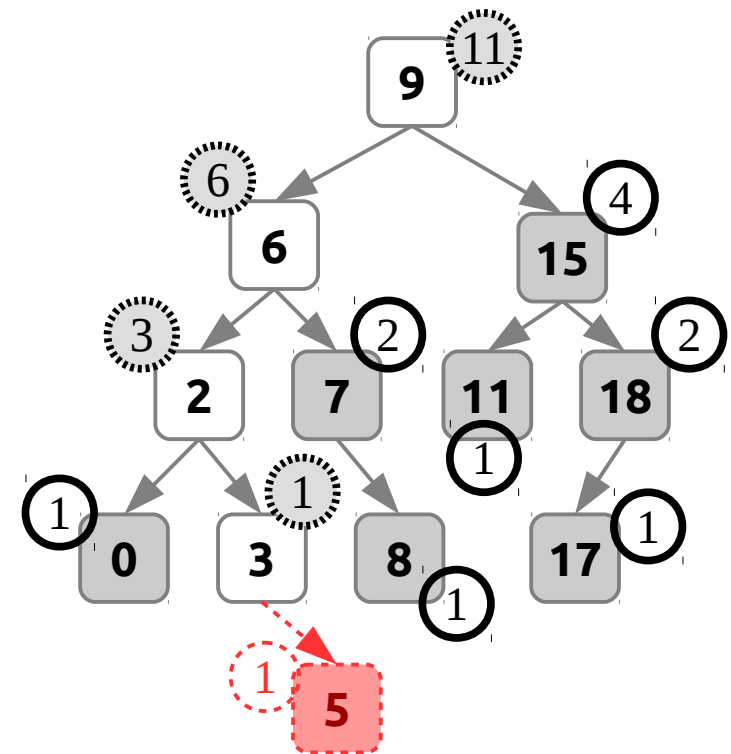
Árvores – Tamanho (*Size*)

- Remoção com campo size

```
void remove( ABB* arvore, K key ) {
    arvore->raiz = remove_no(arvore->raiz, key);
}

struct node* remove_no( struct node *p, K key ) {
    if( !p )
        return 0;
    if( key < p->data->key )
        p->esq = remove_no(p->esq, key);
    else if( key > p->data->key )
        p->dir = remove_no(p->dir, key);
    else {
        if( !p->dir ) {
            struct node *esquerda = p->esq;
            delete p;
            return esquerda;
        }
        if( !p->esq ) {
            struct node *direita = p->dir;
            delete p;
            return direita;
        }
        p->data = max_no(p->esq)->data;
        p->esq = remove_no( p->esq, p->data->key );
    }
    p->size = size(p->esq) + size(p->dir) + 1;
    return p;
}
```

remove(5)



Referências:

- Livro do Cormen
- Livro do Goodrich
- Livro do Robert Sedgewick
 - <https://algs4.cs.princeton.edu>