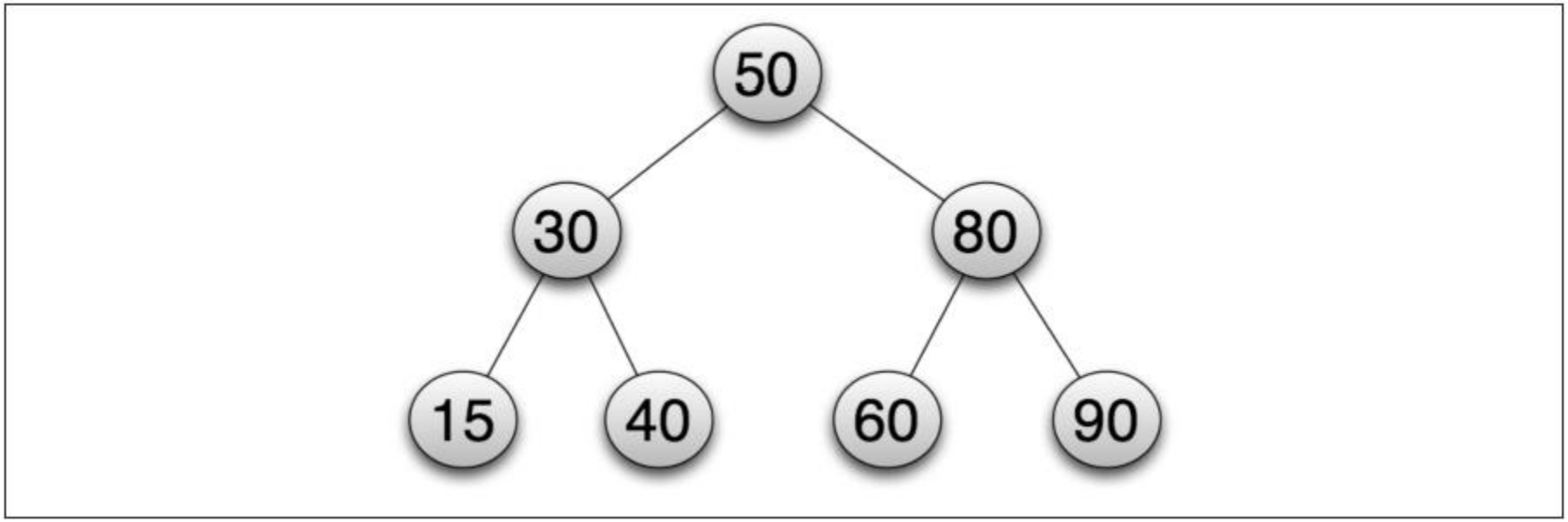
# ESTRUTURAS DE DADOS II – ABP

Prof. Patrícia Noll de Mattos

# Árvores Binárias de Pesquisa (busca)

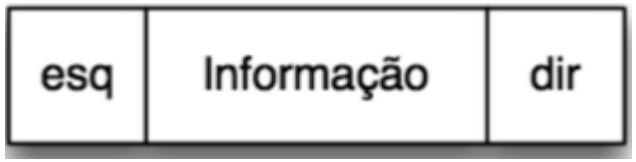
- **Tipo específico de árvores binárias.**
- **Relação de ordem entre os elementos (chave do nodo).**
- **Utilizada em operações que necessitam de busca eficiente.**
- **Relação de ordem:**
- **filho esquerdo do nodo deve possuir uma chave menor que a sua;**
- **Filho direito do nodo deve possuir uma chave maior que a sua.**

# Relação de Ordem



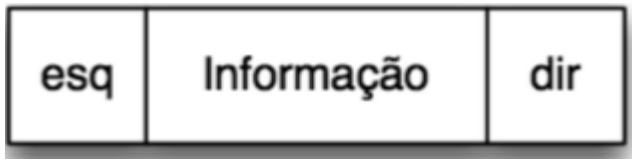

- Subárvore **esquerda**: chave menor
- Subárvore **direita**: chave maior

# Nodo da ABP




```
struct BSTNode
{
        int chave;
        struct BSTNode *esquerda;
        struct BSTNode *direita;
};
struct BSTNode *raiz=NULL;
```

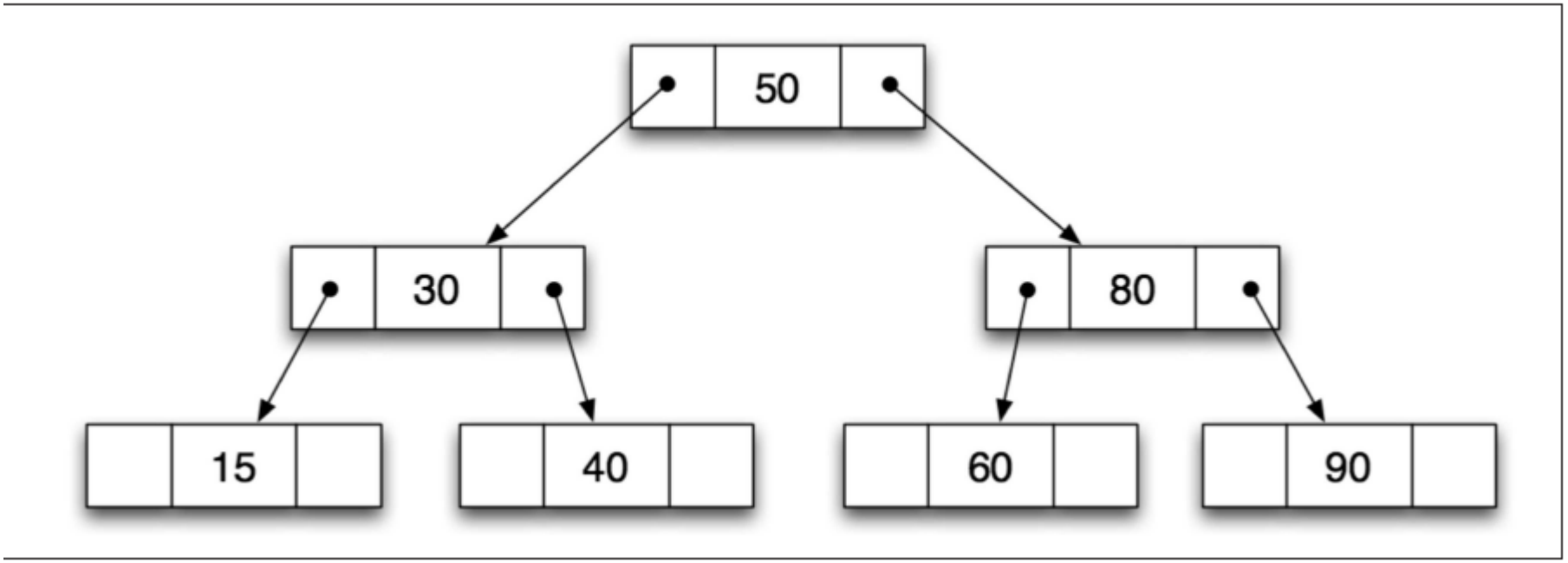
# Aloca Nodo




```
struct BSTNode *reserva(int el){

        struct BSTNode *t=NULL;
        t = (struct BSTNode *) malloc(sizeof(struct BSTNode));
        if(t!=NULL){
                t->chave = el;
                t->direita = NULL;
                t->esquerda = NULL;
        }
        return t;
}
```

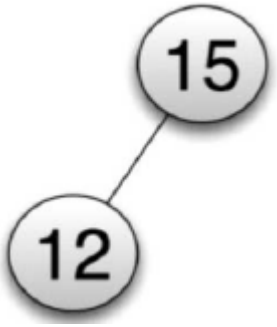
# Nodo = declaração do tipo BSTNode




- Os nodos folha possuem os atributos esquerda e direita nulos.

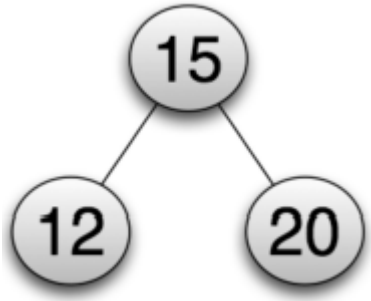
# Construção da ABP

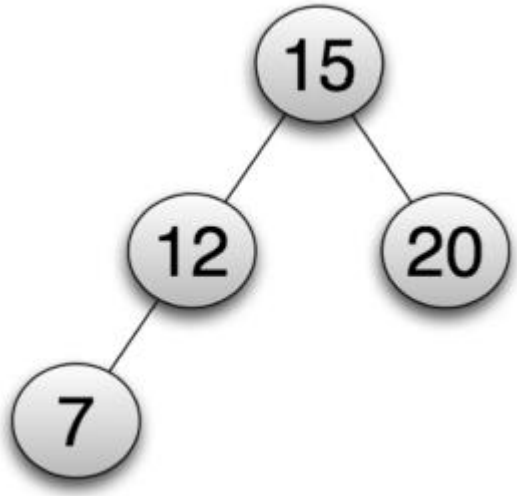
- Valores: 15, 12, 20, 7, 14, 31 e 19.
- **Passo1**: árvore vazia, **nodo raiz;**
- **Passo2:** Compara-se o próximo elemento com a raiz, sendo menor é inserido à esquerda:



# Ordem de inserção de elementos

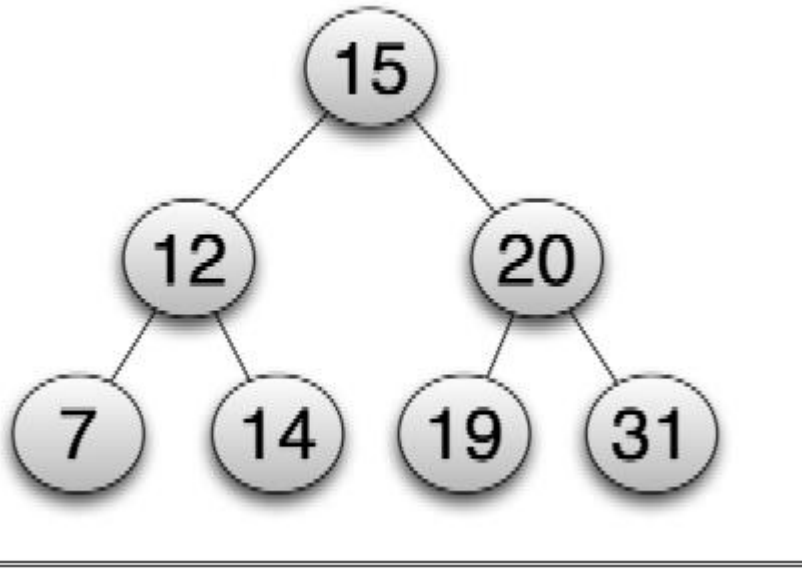
- **Passo 3:** nodo 20 é comparado com a raiz, como é maior e a subárvore direita é nula, é inserido a direira;
- **Passo 4:** o nodo 7 é comparado com a raiz, como é menor, é comparado com o filho esquerdo, nodo 12, como é menor, é inserido à esquerda.



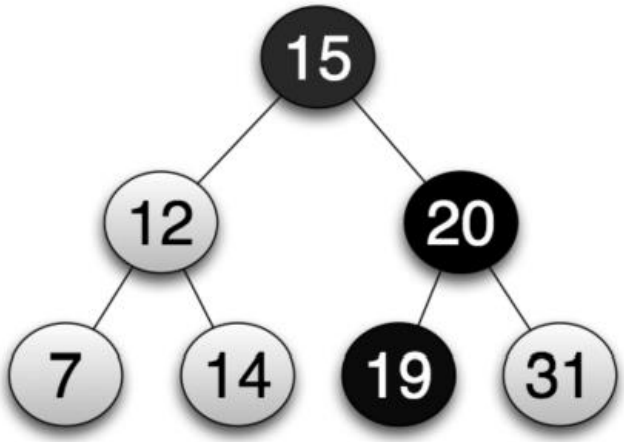
# Ordem de inserção de elementos

- Insere os nodo 14, 31, 19



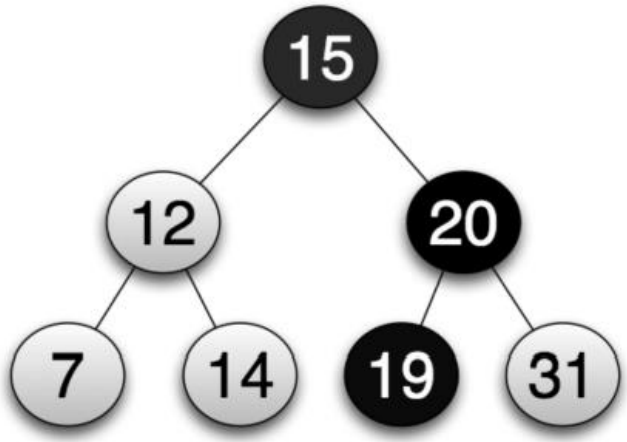
# Busca de elemento

- Inicia pela **raiz:** nodo corrente
- Se a **chave procurada** for menor, o novo **nodo corrente** é o filho esquerdo, repete o procedimento.
- **Para** quando encontrar o elemento ou quando os filhos forem nulos (folha).



# Busca de elemento

```
struct BSTNode *busca(struct BSTNode *raiz, int el)
{
        struct BSTNode *p=raiz;
        while(p!=NULL)
        {
                if(el == p->chave) return p;
                else if(el<p->chave) p = p->esquerda;
                else p = p->direita;
        }
        return NULL;
}
```



# Inserir elementos ABP

- Manter a ordem:
  - Para **cada nodo**, todos os nodos presentes na subárvore esquerda a ele são **menores;**
  - Todos os nodos presentes na subárvore direita são **maiores;**

# Inserir elementos()

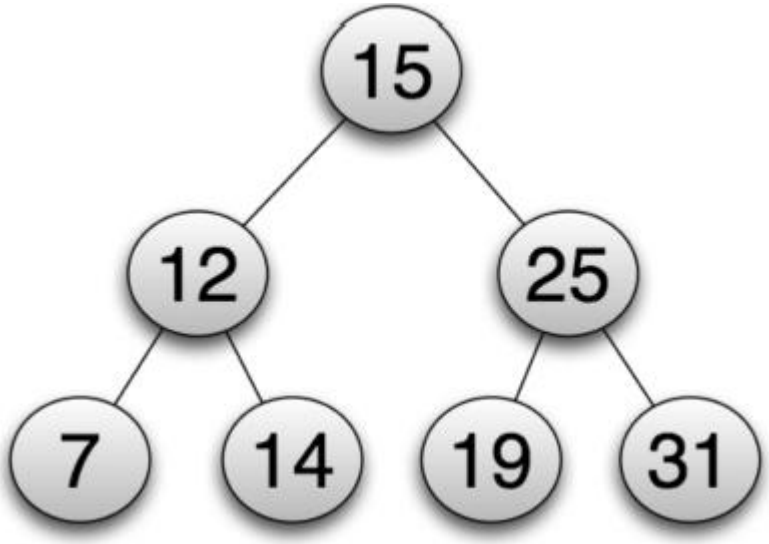
```
void insere(struct BSTNode **raiz, int el)
{
        struct BSTNode *p=*raiz,*ant=NULL;
        if(busca(p,el)==NULL)
        {
                while(p!=NULL)
                {
                   ant=p;
                   if(el<p->chave) p=p->esquerda;
                   else p=p->direita;
                }
                if(*raiz==NULL) *raiz=reserva(el);
                else if(ant->chave<el) ant->direita = reserva(el);
                else ant->esquerda = reserva(el);
        }
}
```

# Caminhamento

- Processo de visitar cada nodo exatamente uma vez;
- **Percurso:** coloca todos os nodos de uma árvore em uma linha;
- **2 tipos de percurso:**
  - Percurso em extensão
  - Percurso em profundidade
- **Extensão:** percorre todos os nodos de um nível da árvore, **nível a nível**, do mais alto ao mais baixo, ou vice versa.

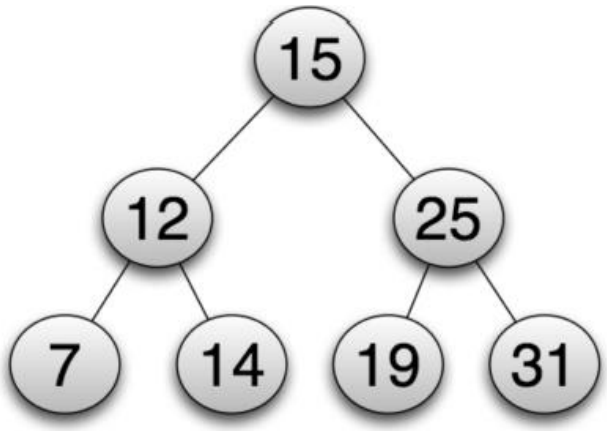
# Caminhamento

- Profundidade: 3 tipos:
  - Caminhamento em ordem
  - Caminhamento pré-ordem
  - Caminhamento pós-ordem



# Caminhamento em-ordem

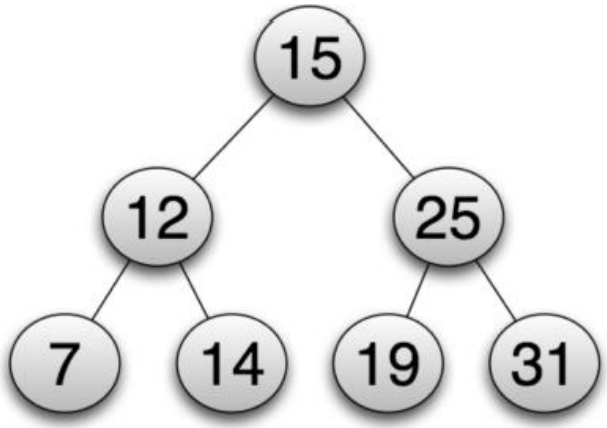
1. Percorre subárvore esquerda
2. Visita o nodo
3. Percorre subárvode direita
4. 7,12,14,15,19,25,31 – maneira crescente




```
void emordem(struct BSTNode *raiz){
        struct BSTNode *p=raiz;
        if(p!=NULL){
                emordem(p->esquerda);
                printf("%i ", p->chave);
                emordem(p->direita);
        }
}
```

# Caminhamento pré-ordem

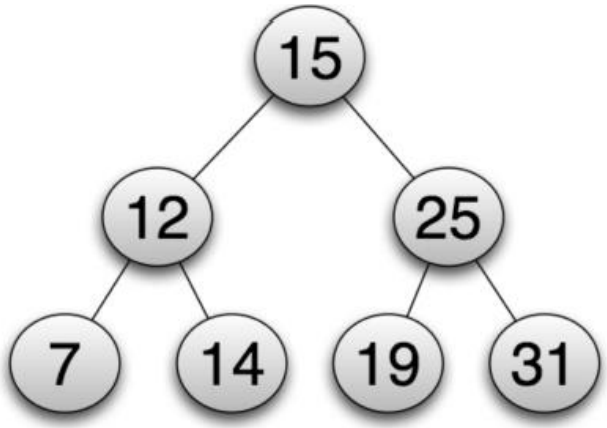
1. Visita o nodo
2. Percorre **a subárvore esquerda**
3. Percorre **a subárvore direira**
4. 15, 12, 7, 14, 25, 19, 31




```
void preordem(struct BSTNode *raiz){
        struct BSTNode *p=raiz;
        if(p!=NULL){
                printf("%i ", p->chave);
                preordem(p->esquerda);
                preordem(p->direita);
        }
}
```

# Caminhamento pós-ordem

1. Percorre **subárvore esquerda**
2. Percorre **subárvore direita**
3. Visita o nodo
4. 7, 14, 12, 19, 31, 25, 15




```
void posordem(struct BSTNode *raiz){
        struct BSTNode *p=raiz;
        if(p!=NULL){
                posordem(p->esquerda);
                posordem(p->direita);
                printf("%i ", p->chave);
        }
}
```

# Função Principal

```
int main(){
    struct BSTNode *raiz=NULL;
    insere(&raiz, 15);
    insere(&raiz, 12);
    insere(&raiz, 25);
    insere(&raiz, 7);
    insere(&raiz, 14);
    insere(&raiz, 31);
    insere(&raiz, 19);
    emordem(raiz); printf("\n");
    preordem(raiz); printf("\n");
    posordem(raiz); printf("\n");
    libera(&raiz);
    getch();
    return 0;
}
```

# Libera memória da árvore

```
void libera(struct BSTNode **p){
    if(*p==NULL) {
            return;
    }else if((*p)->esquerda==NULL && (*p)->direita==NULL){
            printf("\n%i ", (*p)->chave);
            free(*p);
            *p=NULL;
            return;
    }else {
            libera(&(*p)->esquerda);
            libera(&(*p)->direita);
            libera(p);
    }
}
//semelhante caminhamento posordem, primeiro as folhas
```

# Exercícios:

- Crie um procedimento que mostre a maior chave da ABP
- Crie uma função que retorne a soma dos elementos maiores que o enviado por parâmetro.