

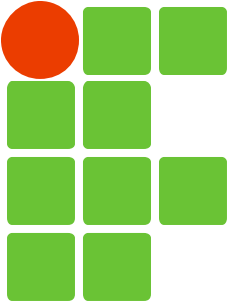
**INSTITUTO FEDERAL**  
**ESPIRITO SANTO**

**CAMPUS COLATINA**

---

# Árvores Binárias - Implementação

Prof. Victorio Albani de Carvalho



INSTITUTO FEDERAL  
ESPIRITO SANTO

CAMPUS COLATINA

# Header (.h)

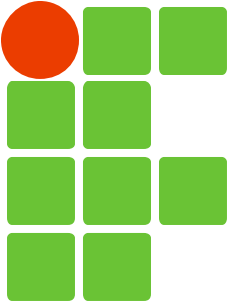
```
#ifndef arvoreBin_h
#define arvoreBin_h

#include <stdio.h>

typedef struct TipoNo{
    int chave;
    struct TipoNo *esq, *dir;
} TNo;

void inicializarArvore (TNo **r);
int inserir(TNo **r, int chave);
TNo* consultar(TNo *r, int chave);
int localizarMaior(TNo *r);
int localizarMenor(TNo *r);
int remover(TNo **r, int chave);
void caminharPreOrdem(TNo *r);
void caminharEmNivel(TNo *r);

#endif /* arvoreBin_h */
```



INSTITUTO FEDERAL  
ESPIRITO SANTO

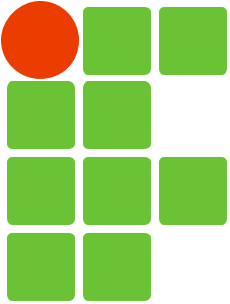
CAMPUS COLATINA

# Inicializar

- Note que queremos mudar o valor do ponteiro para a raiz para nulo, logo precisamos passar ponteiro para ponteiro.

```
void inicializarArvore (TNo **r){  
    *r=NULL;  
}
```

```
int main(int argc, const char * argv[]) {  
    TNo *raiz;  
    inicializarArvore(&raiz);  
}
```

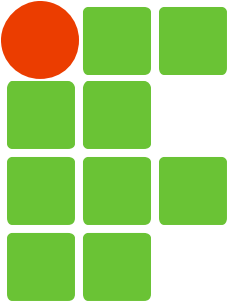


INSTITUTO FEDERAL  
ESPIRITO SANTO

CAMPUS COLATINA

# Consultar

```
TNo* consultar(TNo *r, int chave){
    //Se a arvore/subarvore está vazia, o elemento não existe na árvore
    if(r==NULL)
        return NULL;
    //Se encontrei o elemento na raiz da arvore/subarvore, retorno ele
    else if (r->chave == chave)
        return r;
    //Se a chave que estou buscando é maior que a raiz, vou buscar na subarvore direita
    else if (chave > r->chave)
        return consultar(r->dir, chave);
    //Se a chave que estou buscando é menor que a raiz, vou buscar na subarvore esquerda
    else
        return consultar(r->esq, chave);
}
```



INSTITUTO FEDERAL  
ESPIRITO SANTO

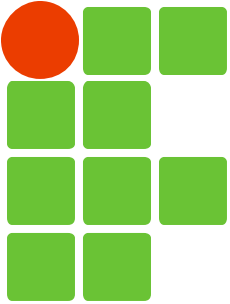
CAMPUS COLATINA

# Inserir

*No inserir também precisaremos alterar valores de ponteiros para TNo. Assim, novamente trabalharemos com ponteiro para ponteiro (\*\*TNo).*

```
int inserir(TNo **r, int chave){  
    //Se a chave não existe crio um novo nó e chamo o método de inserir  
    if (consultar(*r, chave)==NULL){  
        TNo *novo = (TNo *) malloc (sizeof (TNo));  
        novo->chave = chave;  
        novo->dir=NULL;  
        novo->esq=NULL;  
        inserirNo(r, novo);  
        return 1;  
        //Se a chave já existe na árvore retorno zero e não insiro  
    } else {  
        return 0;  
    }  
}
```

```
void inserirNo(TNo **r, TNo *novo){
    //Se a raiz estiver vazia insiro na raiz
    if(*r==NULL){
        *r=novo;
    }
    //Se a raiz não estiver vazia...
    else{
        //Se a chave do novo elemento for maior que a da raiz...
        if(novo->chave > (*r)->chave){
            //Se a raiz não tem filho a direita, o novo elemento será seu filho a direita
            if ((*r)->dir == NULL)
                (*r)->dir = novo;
            //Se a raiz tem filho a direita, chamo o método de inserir recursivamente
            //passando a subarvore direita
            else
                inserirNo(&(*r)->dir, novo);
        }
        //Se a chave do novo elemento for menor que a raiz...
        else{
            //Se a raiz não tem filho a esquerda, o novo elemento será seu filho a esquerda
            if ((*r)->esq == NULL)
                (*r)->esq = novo;
            //Se a raiz tem filho a esquerda, chamo o método de inserir recursivamente
            //passando a subarvore esquerda
            else
                inserirNo(&(*r)->esq, novo);
        }
    }
}
```

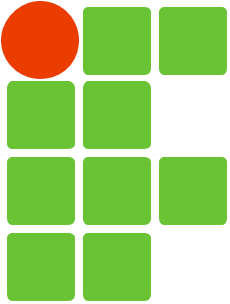


INSTITUTO FEDERAL  
ESPIRITO SANTO

CAMPUS COLATINA

# Encontrar Maior

```
int localizarMaior(TNo *r){  
    if(r==NULL)  
        return -1;  
    else{  
        //Se não tem filho a direita, então ele é o maior  
        if(r->dir == NULL)  
            return r->chave ;  
        //Se tem filho a direita, vou localizar o maior na subarvore direita  
        else  
            return localizarMaior(r->dir);  
    }  
}
```



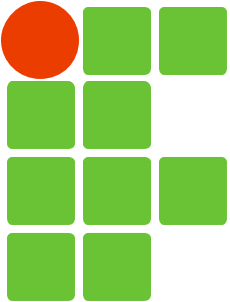
INSTITUTO FEDERAL  
ESPIRITO SANTO

CAMPUS COLATINA

# Caminhamento Pré-ordem

```
void caminharPreOrdem(TNo *r){  
    if(r!=NULL){  
        printf("%d ", r->chave);  
        caminharPreOrdem(r->esq);  
        caminharPreOrdem(r->dir);  
    }  
}
```





INSTITUTO FEDERAL  
ESPIRITO SANTO

CAMPUS COLATINA

# Caminhamento em Nível

Para fazer o caminhamento em nível vamos utilizar uma fila como estrutura auxiliar.

Ao lado temos o .h da biblioteca de filas que será utilizada.

O método de caminhamento em nível está no próximo slide.

```
#ifndef fila_h
#define fila_h

#include <stdio.h>
#include "arvoreBin.h"

typedef struct TipoNoFila{
    TNo *valor;
    struct TipoNoFila *prox;
}TNoFila;

typedef struct Fila{
    TNoFila *prim, *ult;
}TFila;

void inicializarFila(TFila *f);
void inserirFila(TFila *f, TNo *valor);
TNo * removerFila(TFila *f);
int filaVazia(TFila *f);
#endif /* fila_h */
```



# Caminhamento em Nível

INSTITUTO FEDERAL  
ESPIRITO SANTO

CAL void caminharEmNivel(TNo \*r){

TFila f;

TNo \*no;

if (r!=NULL){

*//Para fazer o caminhamento em nivel utilizo uma fila como estrutura auxiliar*

*//Essa fila vai armazenar nós da árvore. Para começar inicio a fila...*

inicializarFila(&f);

*//Insiro a raiz da árvore na fila*

inserirFila(&f, r);

*//Enquanto a fila não estiver vazia, ainda há elemento a imprimir*

while(!filaVazia(&f)){

*//Pego o primeiro elemento da fila*

no=removerFila(&f);

*//Imprimo o elemento que retirei da fila*

printf("%d ", no->chave);

*//Se o elemento que peguei tiver filho a esquerda, insiro esse filho na fila*

if(no->esq != NULL)

inserirFila(&f,no->esq);

*//Se o elemento que peguei tiver filho a direita, insiro esse filho na fila*

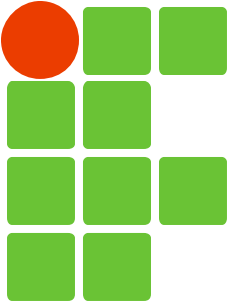
if(no->dir != NULL)

inserirFila(&f,no->dir);

}

}

}



INSTITUTO FEDERAL  
ESPIRITO SANTO

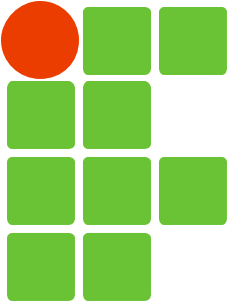
CAMPUS COLATINA

# Remoção de Nó

---

## 3 Possibilidades

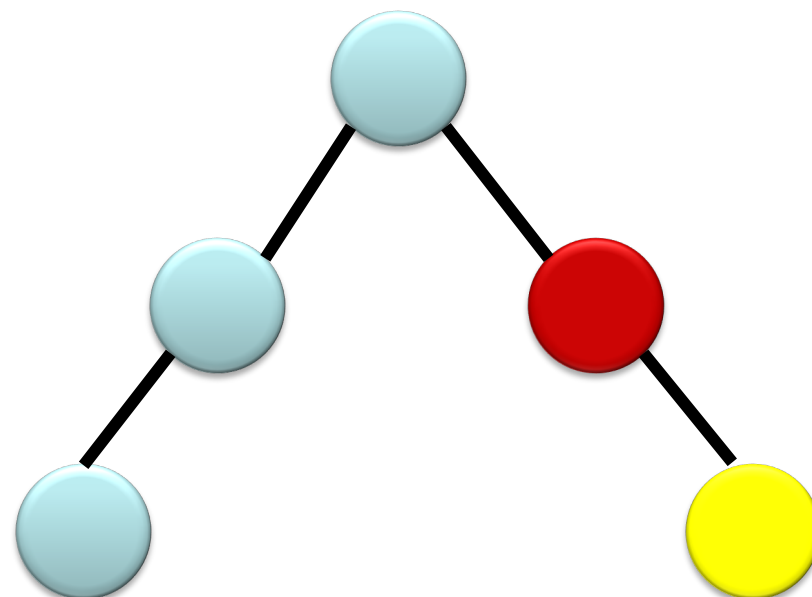
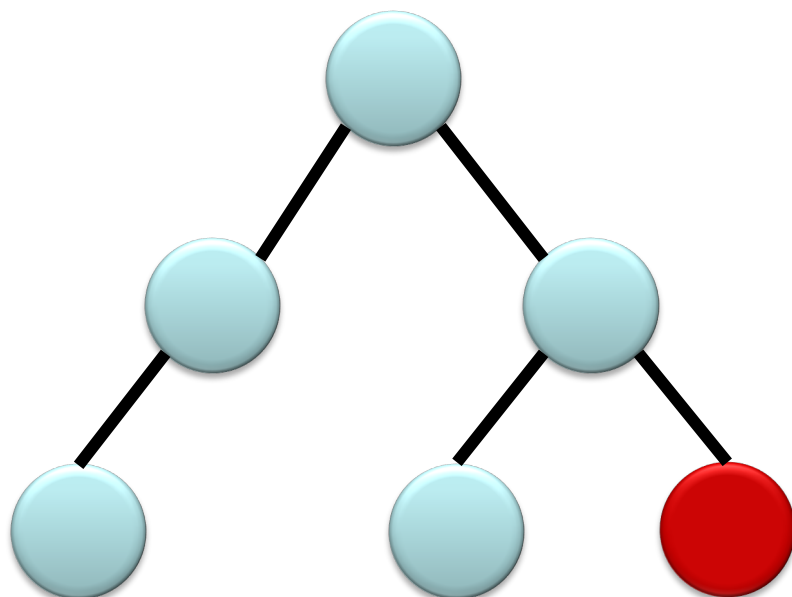
1. Não tem filho
2. Somente 1 Filho
3. 2 Filhos

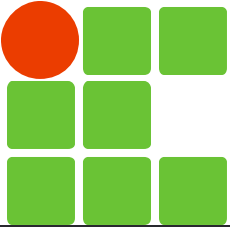


INSTITUTO FEDERAL  
ESPIRITO SANTO

CAMPUS COLATINA

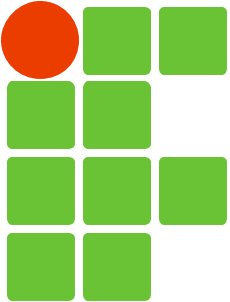
# Sem Filhos ou com um filho





# Remoção com 0 ou 1 filho

```
int remover(TNo **r, int chave){
    TNo *excluir;
    if (*r != NULL){
        //Se a chave a ser excluída está na raiz...
        if ((*r)->chave == chave){
            //guardo uma referência ao elemento a ser excluído
            excluir = *r;
            //Se ele não tem filho a esquerda, trago o filho a direita para o lugar dele
            if ((*r)->esq == NULL)
                *r = (*r)->dir;
            //Se ele não tem filho a direita, trago o filho a esquerda para o lugar dele
            else if ((*r)->dir == NULL)
                *r = (*r)->esq;
            //Se ele tiver dois filhos vou achar o maior elemento da subarvore esquerda, copiar o valor dele
            //para o elemento que seria excluído e guardar uma referência para ele a de de desalocá-lo
            else{
                excluir=localizarERetirarMaior(&(*r)->esq);
                (*r)->chave=excluir->chave;
            }
            free(excluir);
            return 1;
        }
        else if (chave < (*r)->chave)
            return remover(&(*r)->esq, chave);
        else
            return remover(&(*r)->dir, chave);
    }
    else
        return 0;
}
```

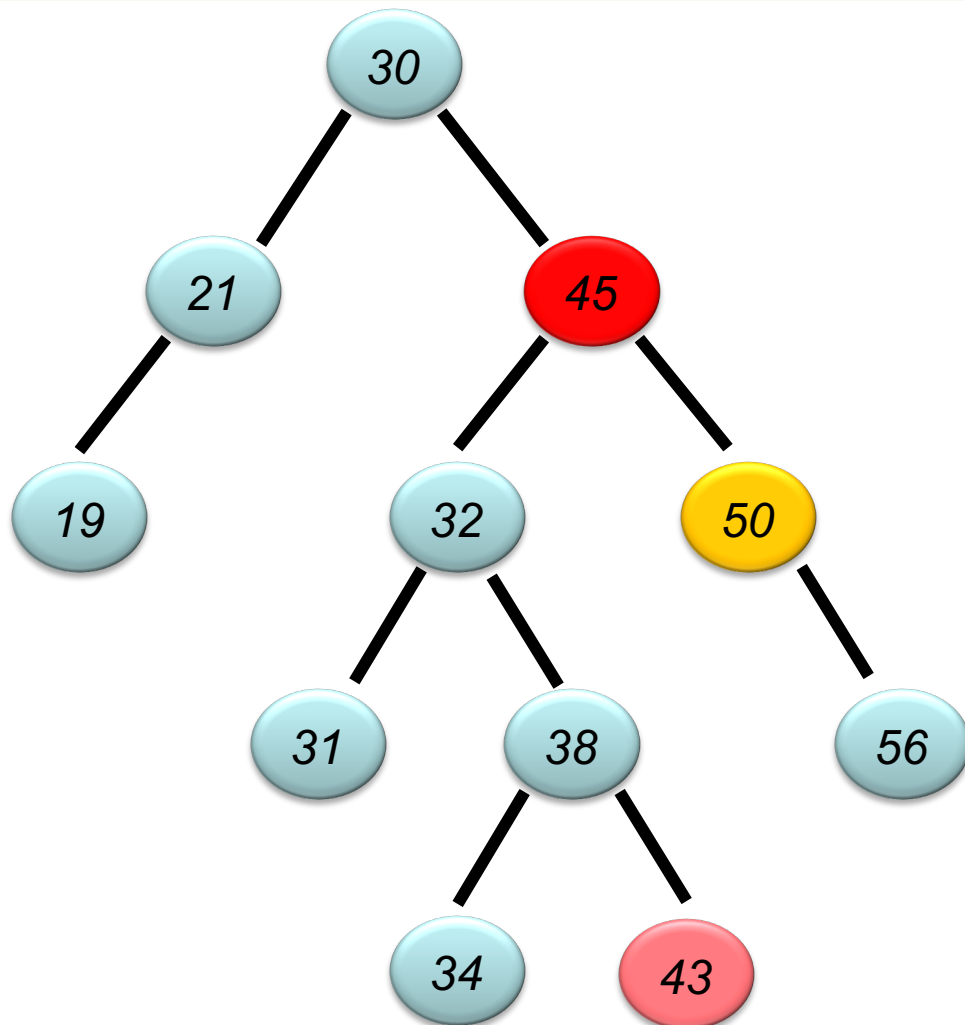


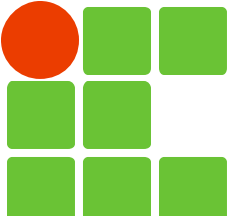
# Remoção - com 2 filhos

2 Opções:

- Máximo da árvore esquerda vai pra raiz;
- Mínimo da árvore direita vai pra raiz;

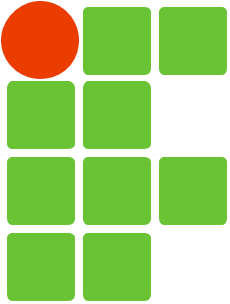
No código a seguir adotamos a primeira opção





# Remoção – com 2 filhos

```
int remover(TNo **r, int chave){
    TNo *excluir;
    if (*r != NULL){
        //Se a chave a ser excluída está na raiz...
        if ((*r)->chave == chave){
            //guardo uma referência ao elemento a ser excluído
            excluir = *r;
            //Se ele não tem filho a esquerda, trago o filho a direita para o lugar dele
            if ((*r)->esq == NULL)
                *r = (*r)->dir;
            //Se ele não tem filho a direita, trago o filho a esquerda para o lugar dele
            else if ((*r)->dir == NULL)
                *r = (*r)->esq;
            //Se ele tiver dois filhos vou achar o maior elemento da subarvore esquerda, copiar o valor dele
            //para o elemento que seria excluído e guardar uma referencia para ele a de de desalocá-lo
            else{
                excluir=localizarERetirarMaior(&(*r)->esq);
                (*r)->chave=excluir->chave;
            }
            free(excluir);
            return 1;
        }
        else if (chave < (*r)->chave)
            return remover(&(*r)->esq, chave);
        else
            return remover(&(*r)->dir, chave);
    }
    else
        return 0;
}
```



INSTITUTO FEDERAL  
ESPIRITO SANTO

CAMPUS COLATINA

# Remoção – Método que Localiza e retira maior

*Abaixo está a função que localiza e retira o maior elemento da subárvore. Essa função é utilizada no método de remoção para o caso em que o elemento a ser excluído tem 2 filhos.*

```
TNo* localizarERetirarMaior(TNo **r){
    if(*r==NULL)
        return NULL;
    else{
        //Se não tem filho a direita, então ele é o maior
        if((*r)->dir == NULL){
            TNo *maior;
            maior = *r;
            *r = (*r)->esq;
            return maior;
        }
        //Se tem filho a direita, vou localizar o maior na subarvore direita
        else
            return localizarERetirarMaior(&(*r)->dir);
    }
}
```