MC-202 — Aula 12
Árvore Binárias de Busca

Lehilton Pedrosa

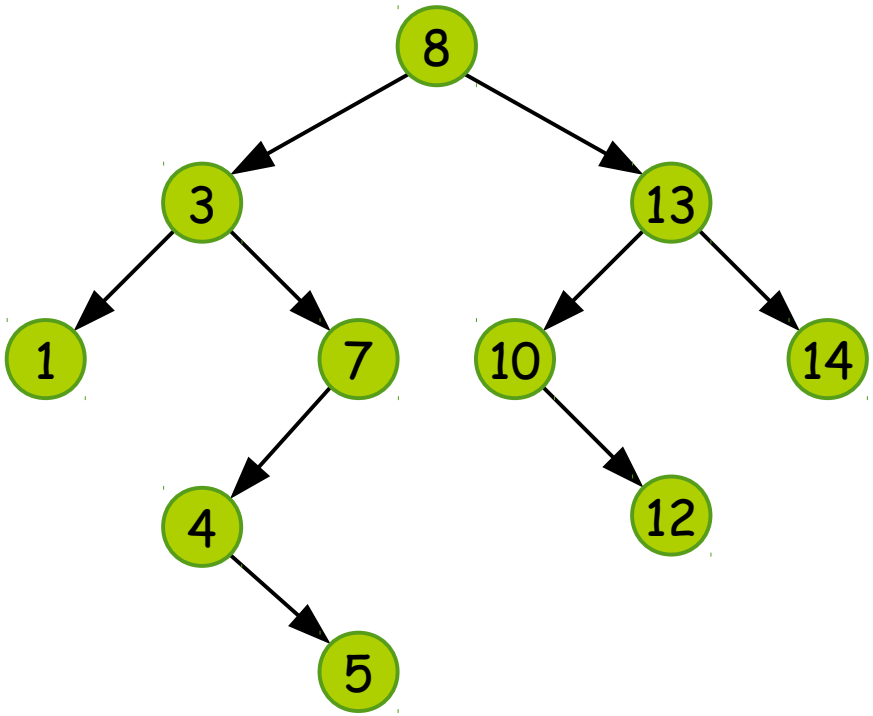
Instituto de Computação – Unicamp

Segundo Semestre de 2015

Roteiro

Introdução

Pergunta: Que propriedade tem essa árvore?

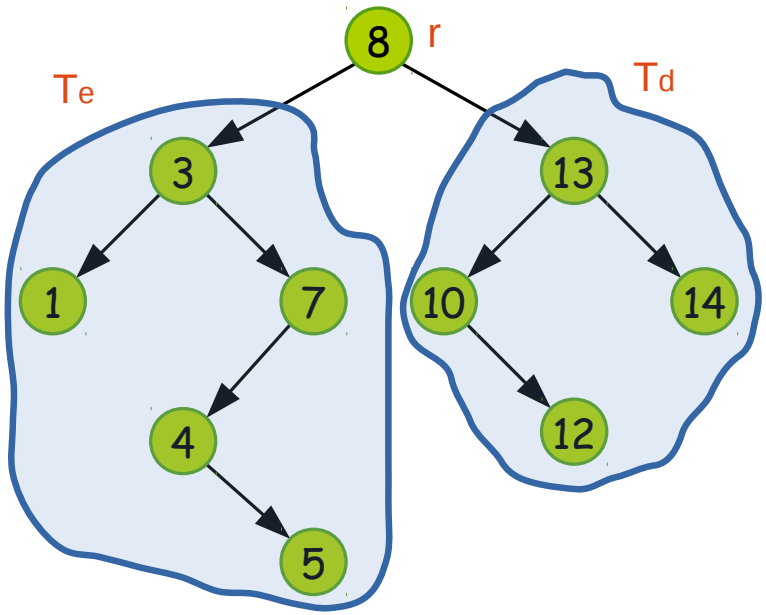


Árvore Binária de Busca

Definição

Uma **Árvore Binária de Busca** (ABB) é uma árvore binária em que cada nó contém um elemento de um conjunto ordenável. Cada nó r, com subárvores esquerda T_e e direita T_d satisfaz a seguinte propriedade:

1. $e \leq r$ para todo elemento $e \in T_e$;
2. $r \leq d$ para todo elemento $d \in T_d$.



Árvore de Busca como Conjunto Dinâmico

Tipo abstrato de dados

```
typedef struct NoArv {
    int chave;
    int dado;
    struct NoArv *esq, *dir;
} *NoArv;
```

Árvore de Busca como Conjunto Dinâmico

Tipo abstrato de dados

```
typedef struct NoArv {
    int chave;
    int dado;
    struct NoArv *esq, *dir;
} *NoArv;

NoArv *iniciar_arv();
void destruir_arv(NoArv **arv);
```

Árvore de Busca como Conjunto Dinâmico

Tipo abstrato de dados

```
typedef struct NoArv {
    int chave;
    int dado;
    struct NoArv *esq, *dir;
} *NoArv;

NoArv *iniciar_arv();
void destruir_arv(NoArv **arv);

void inserir(NoArv **arv, int chave, int dado);
void remover(NoArv **arv, int chave);
```

Árvore de Busca como Conjunto Dinâmico

Tipo abstrato de dados

```
typedef struct NoArv {
    int chave;
    int dado;
    struct NoArv *esq, *dir;
} *NoArv;

NoArv *iniciar_arv();
void destruir_arv(NoArv **arv);

void inserir(NoArv **arv, int chave, int dado);
void remover(NoArv **arv, int chave);

NoArv *buscar(NoArv *arv, int chave);
NoArv *sucessor(NoArv *arv, int chave);
NoArv *antecessor(NoArv *arv, int chave);
```

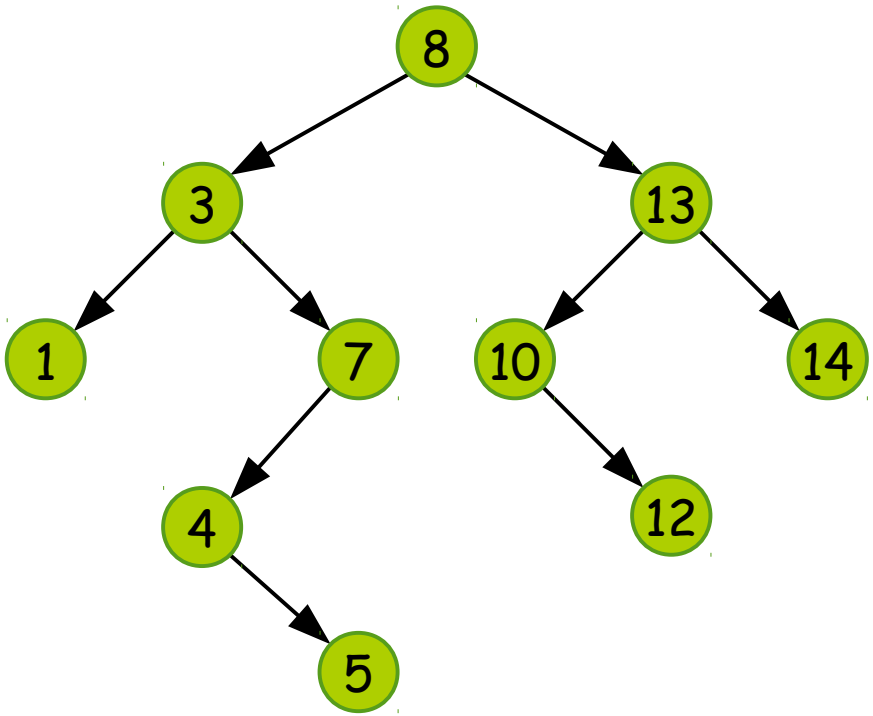
Árvore de Busca como Conjunto Dinâmico

Tipo abstrato de dados

```
typedef struct NoArv {
    int chave;
    int dado;
    struct NoArv *esq, *dir;
} *NoArv;

NoArv *iniciar_arv();
void destruir_arv(NoArv **arv);

void inserir(NoArv **arv, int chave, int dado);
void remover(NoArv **arv, int chave);

NoArv *buscar(NoArv *arv, int chave);
NoArv *sucessor(NoArv *arv, int chave);
NoArv *antecessor(NoArv *arv, int chave);

void imprimir_ordem(NoArv **arv);
```

Buscando um elemento

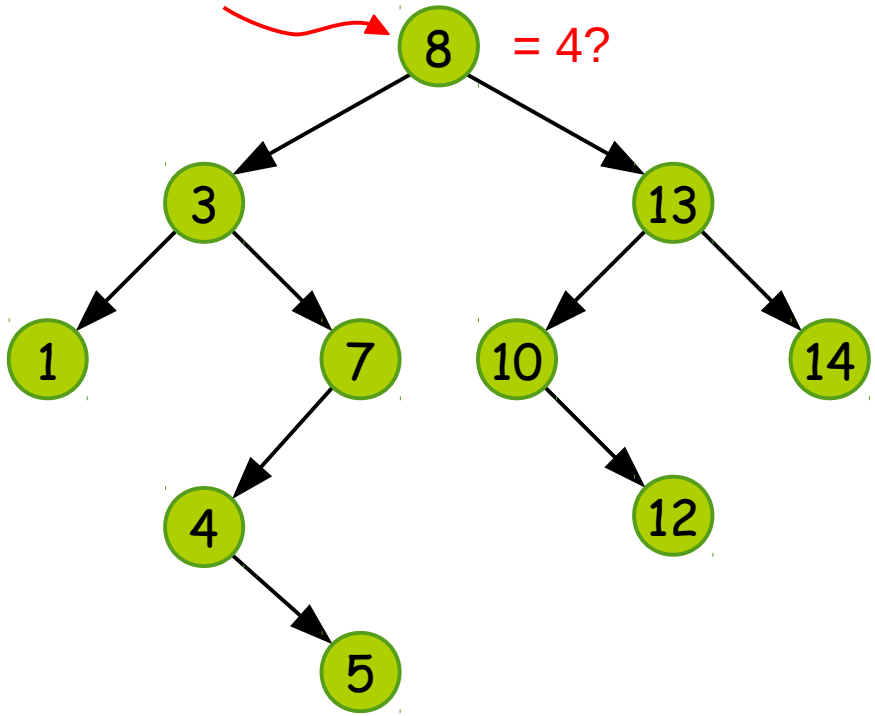
Buscando elemento 4



Buscando um elemento

Buscando elemento 4

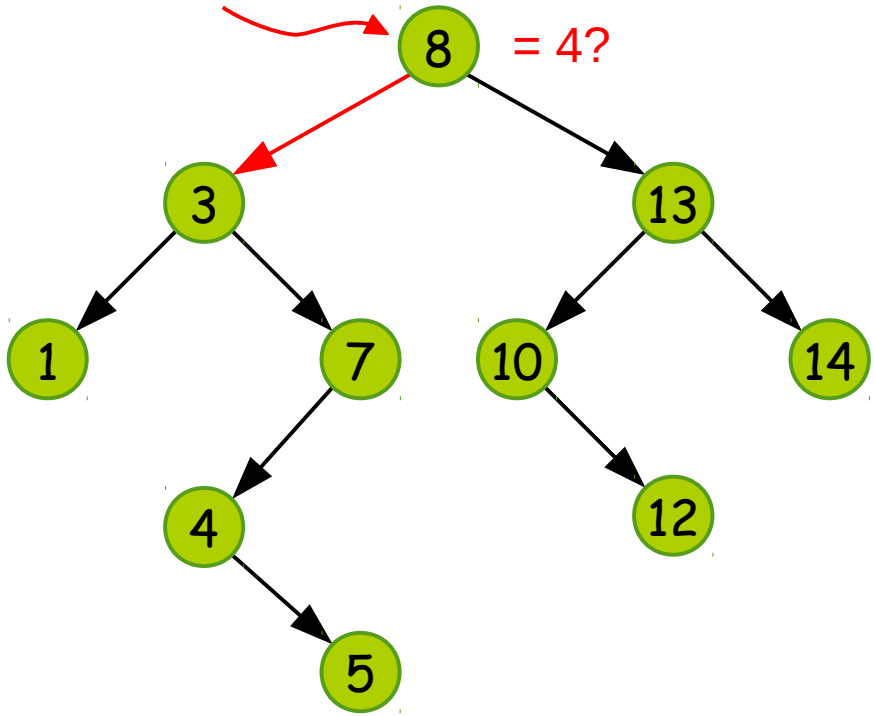
1. começamos procurando pela raiz



Buscando um elemento

Buscando elemento 4

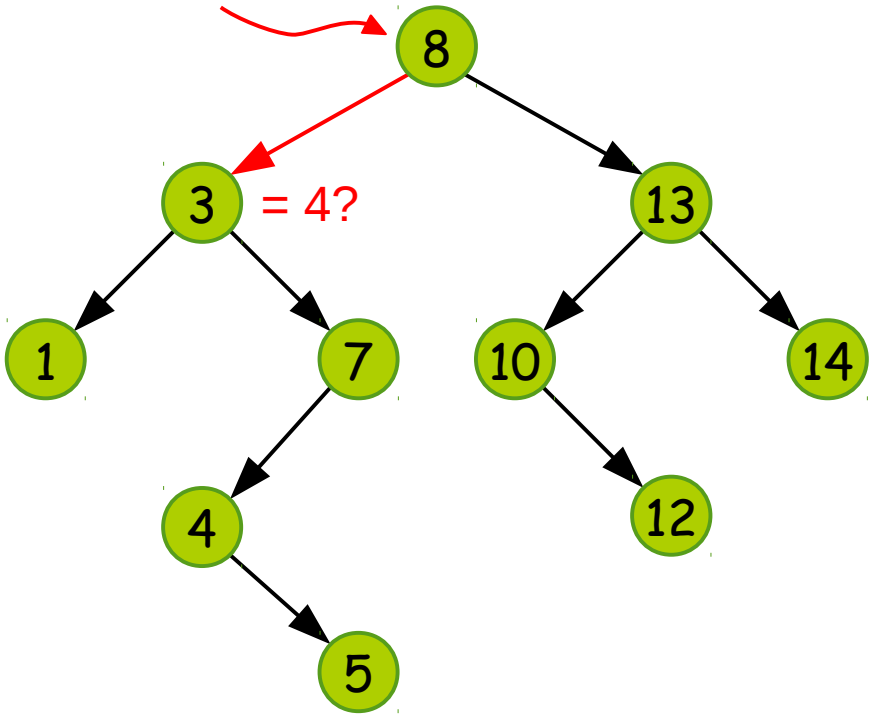
1. começamos procurando pela raiz
2. descemos a árvore recursivamente



Buscando um elemento

Buscando elemento 4

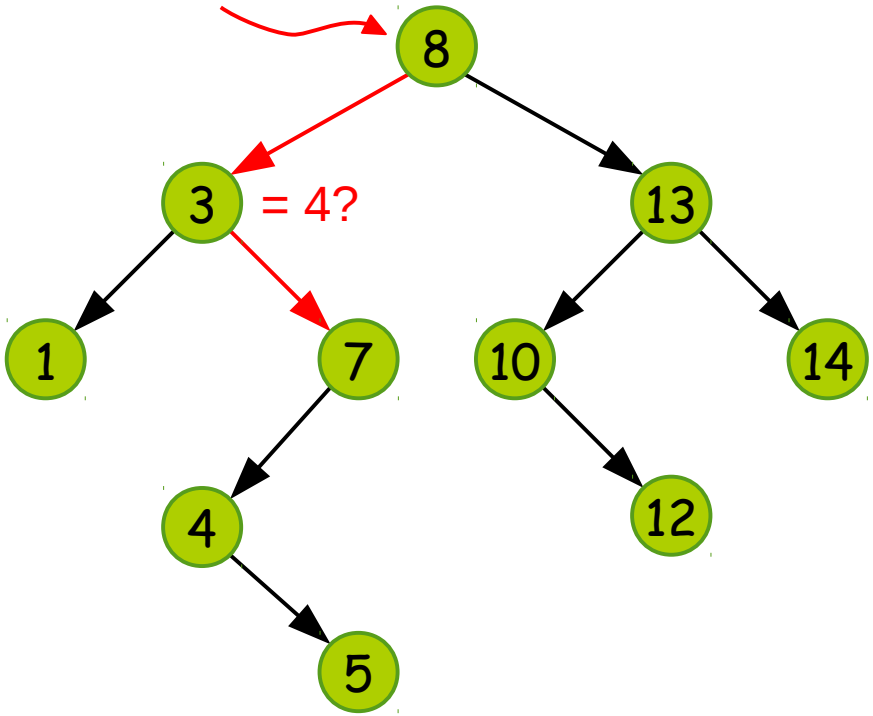
1. começamos procurando pela raiz
2. descemos a árvore recursivamente



Buscando um elemento

Buscando elemento 4

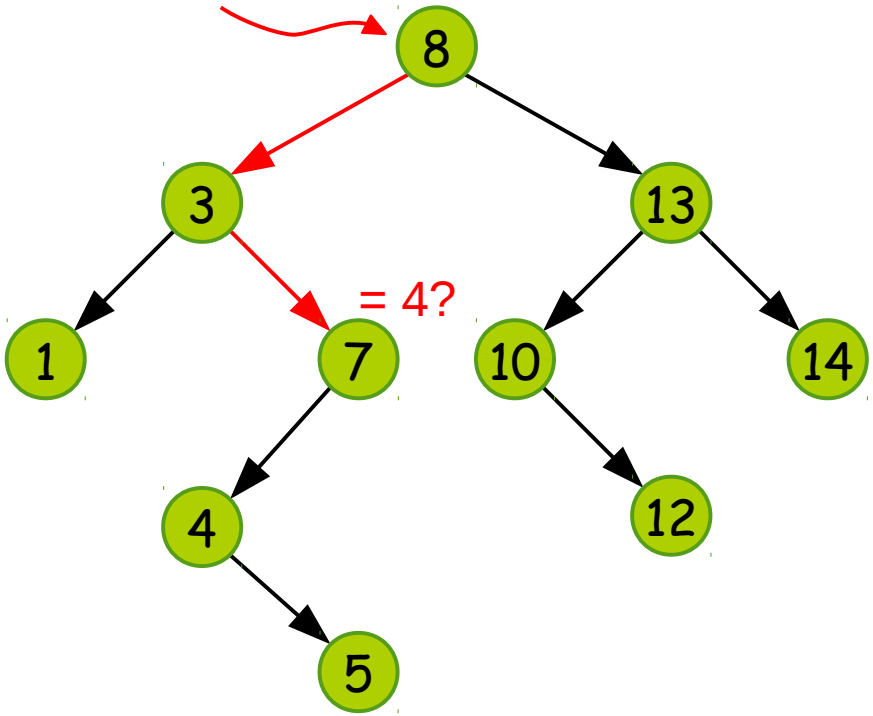
1. começamos procurando pela raiz
2. descemos a árvore recursivamente



Buscando um elemento

Buscando elemento 4

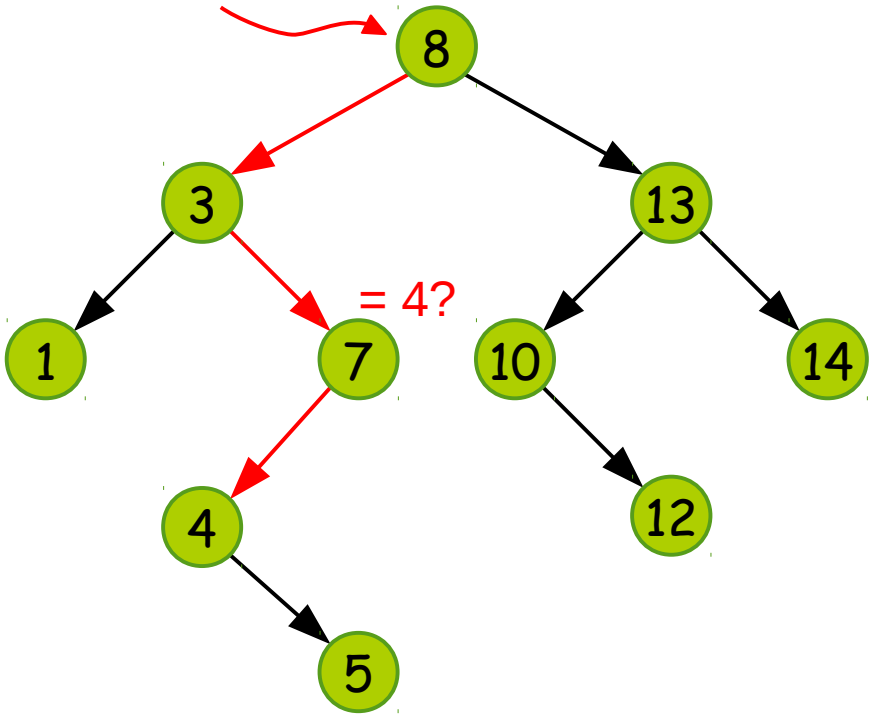
1. começamos procurando pela raiz
2. descemos a árvore recursivamente



Buscando um elemento

Buscando elemento 4

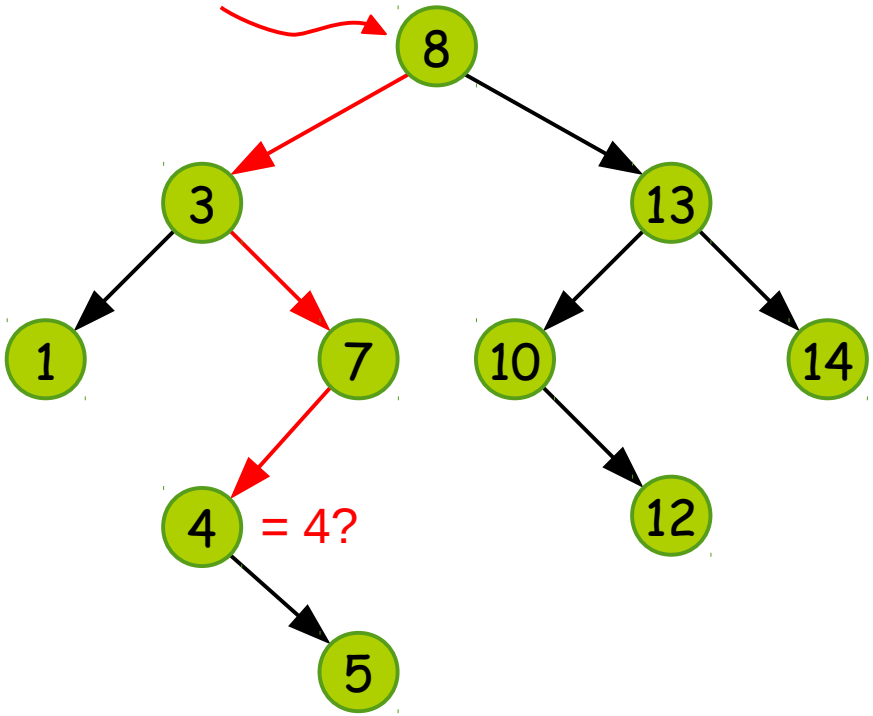
1. começamos procurando pela raiz
2. descemos a árvore recursivamente



Buscando um elemento

Buscando elemento 4

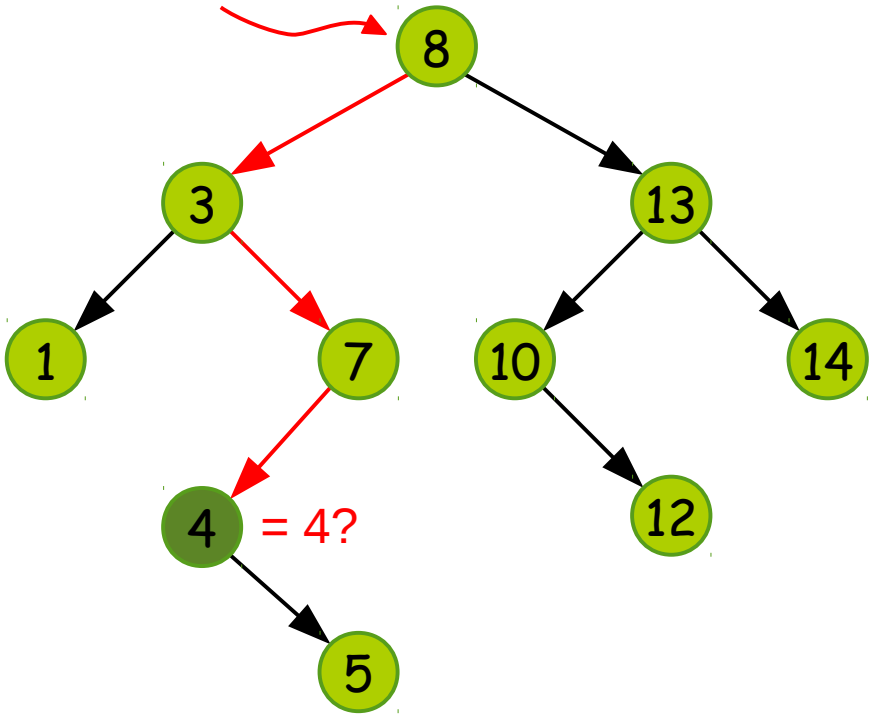
1. começamos procurando pela raiz
2. descemos a árvore recursivamente



Buscando um elemento

Buscando elemento 4

1. começamos procurando pela raiz
2. descemos a árvore recursivamente
3. paramos quando encontrarmos o elemento



Implementando a busca

Busca

```
// retorna um nó ou NULL se não encontrou
NoArv *buscar(NoArv *r, int chave) {
    // enquanto subárvore r não é vazia
    while (r != NULL) {
```

Implementando a busca

Busca

```
// retorna um nó ou NULL se não encontrou
NoArv *buscar(NoArv *r, int chave) {
    // enquanto subárvore r não é vazia
    while (r != NULL) {
        // se encontramos, paramos
        if (r->chave == chave)
            return r;
```

Implementando a busca

Busca

```
// retorna um nó ou NULL se não encontrou
NoArv *buscar(NoArv *r, int chave) {
    // enquanto subárvore r não é vazia
    while (r != NULL) {
        // se encontramos, paramos
        if (r->chave == chave)
            return r;

        // senão, descemos a árvore
        if (r->chave < chave)
            r = r->dir;
        else
            r = r->esq;
    }
```

Implementando a busca

Busca

```
// retorna um nó ou NULL se não encontrou
NoArv *buscar(NoArv *r, int chave) {
    // enquanto subárvore r não é vazia
    while (r != NULL) {
        // se encontramos, paramos
        if (r->chave == chave)
            return r;

        // senão, descemos a árvore
        if (r->chave < chave)
            r = r->dir;
        else
            r = r->esq;
    }
    // se não encontrou, retorna NULL
    return NULL;
}
```

Implementando a busca (cont.)

Busca - versão recursiva

```
// retorna um nó ou NULL se não encontrou
NoArv *buscar(NoArv *r, int chave) {
```

Implementando a busca (cont.)

Busca - versão recursiva

```
// retorna um nó ou NULL se não encontrou
NoArv *buscar(NoArv *r, int chave) {

    // se árvore é vazia, retorna NULL
    if (r == NULL)
        return r;
```

Implementando a busca (cont.)

Busca - versão recursiva

```
// retorna um nó ou NULL se não encontrou
NoArv *buscar(NoArv *r, int chave) {

    // se árvore é vazia, retorna NULL
    if (r == NULL)
        return r;

    // se encontrou chave, retorna raiz
    if (r->chave == chave)
        return r;
```

Implementando a busca (cont.)

Busca - versão recursiva

```
// retorna um nó ou NULL se não encontrou
NoArv *buscar(NoArv *r, int chave) {

    // se árvore é vazia, retorna NULL
    if (r == NULL)
        return r;

    // se encontrou chave, retorna raiz
    if (r->chave == chave)
        return r;

    // senão, busca recursivamente
    if (r->chave < chave)
        return buscar(r->dir, chave);
    else
        return buscar(r->esq, chave);
}
```

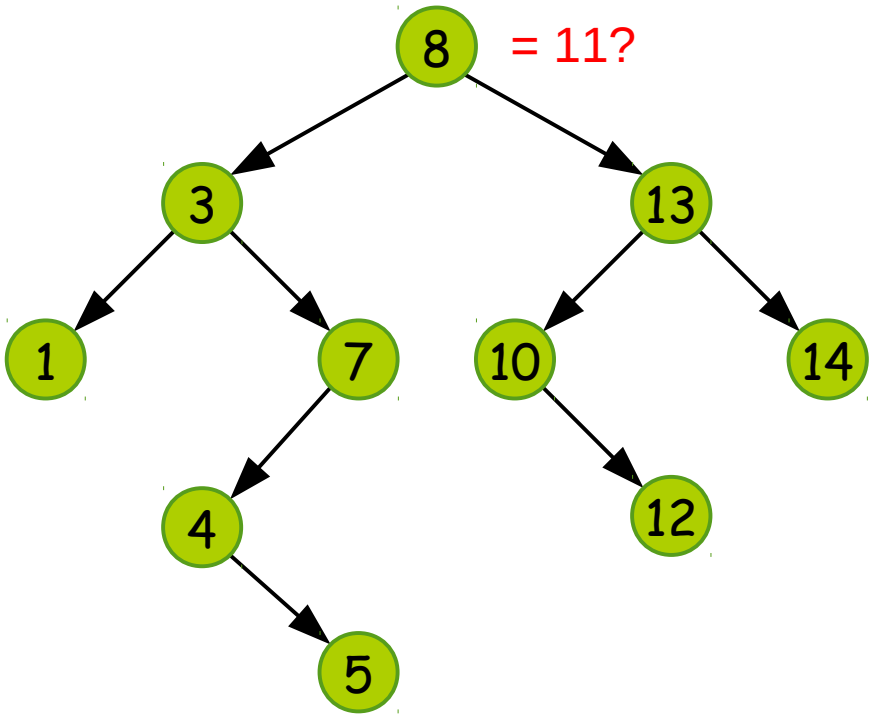
Inserindo um elemento

Inserir o número 11

Inserindo um elemento

Inserir o número 11

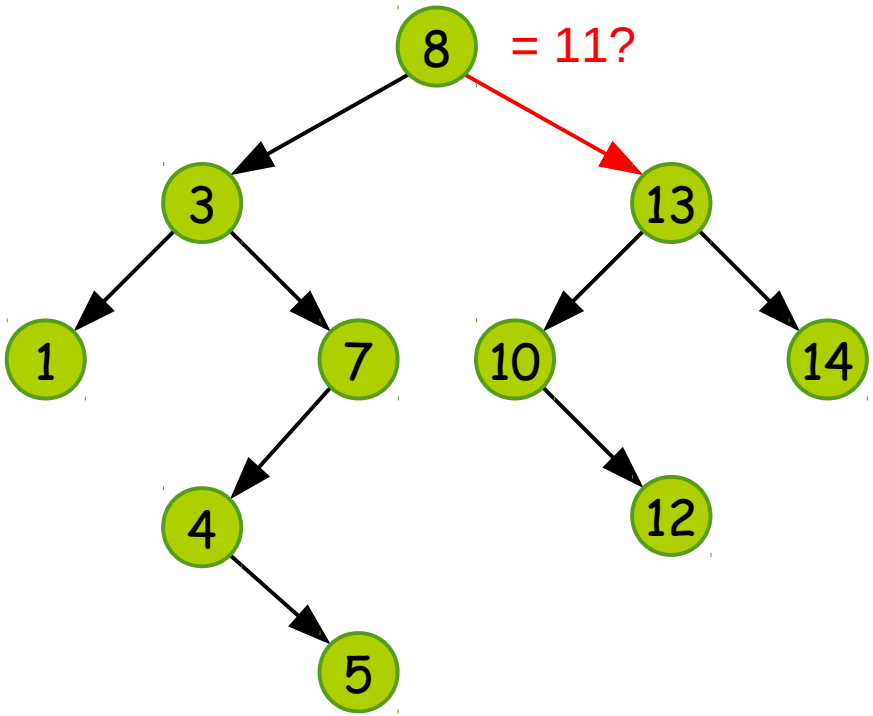
1. procuramos pelo elemento



Inserindo um elemento

Inserir o número 11

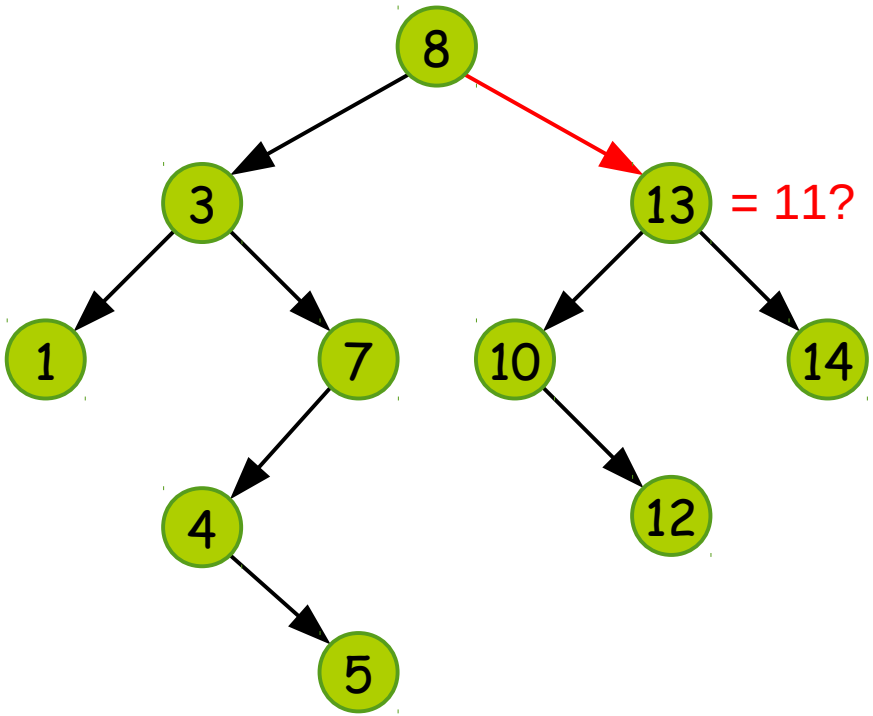
1. procuramos pelo elemento
2. vamos percorrer um caminho da raiz até uma folha



Inserindo um elemento

Inserir o número 11

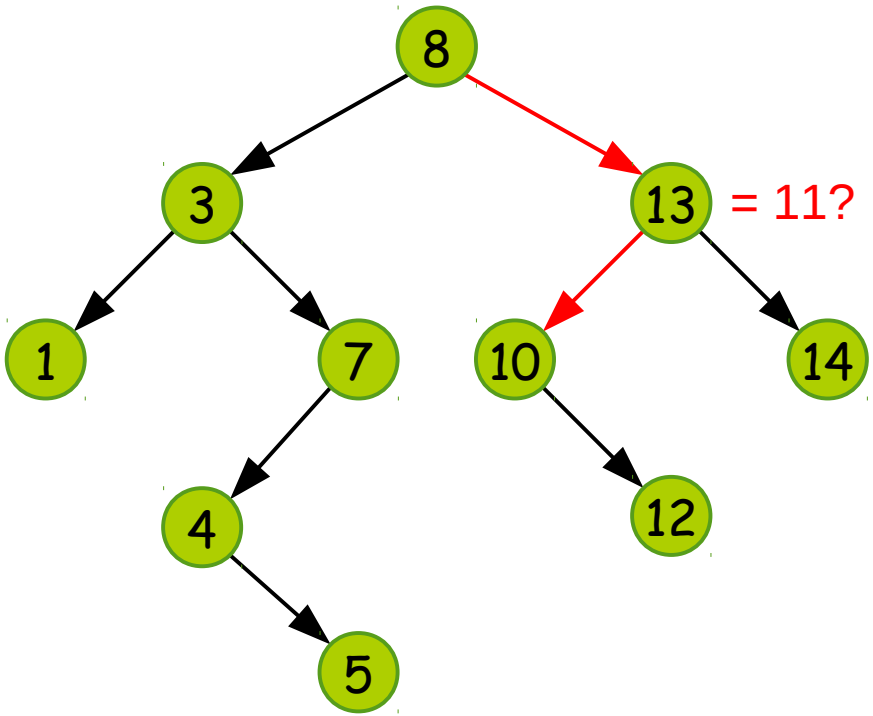
1. procuramos pelo elemento
2. vamos percorrer um caminho da raiz até uma folha



Inserindo um elemento

Inserir o número 11

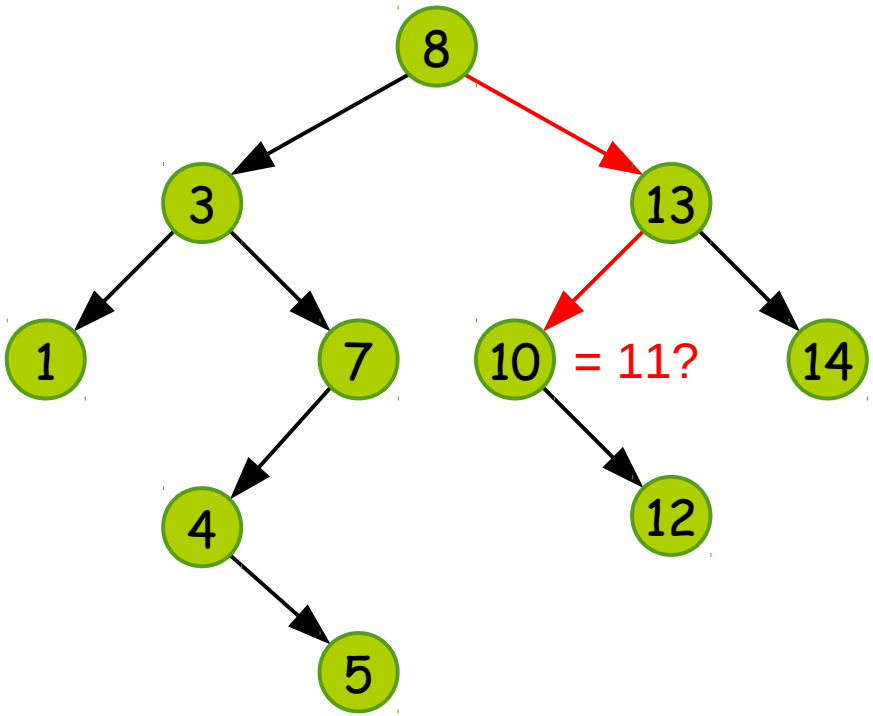
1. procuramos pelo elemento
2. vamos percorrer um caminho da raiz até uma folha



Inserindo um elemento

Inserir o número 11

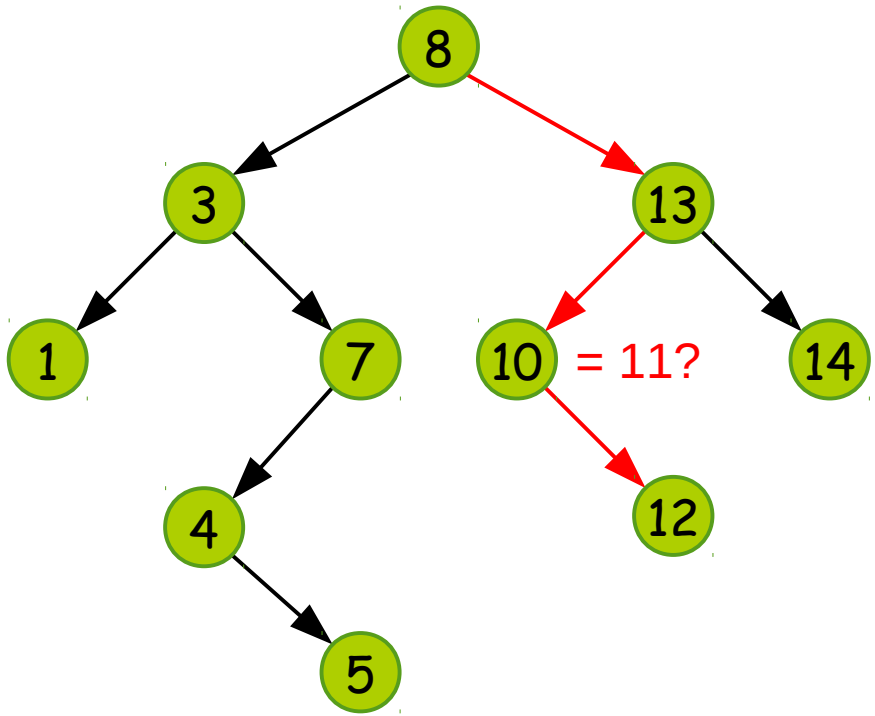
1. procuramos pelo elemento
2. vamos percorrer um caminho da raiz até uma folha



Inserindo um elemento

Inserir o número 11

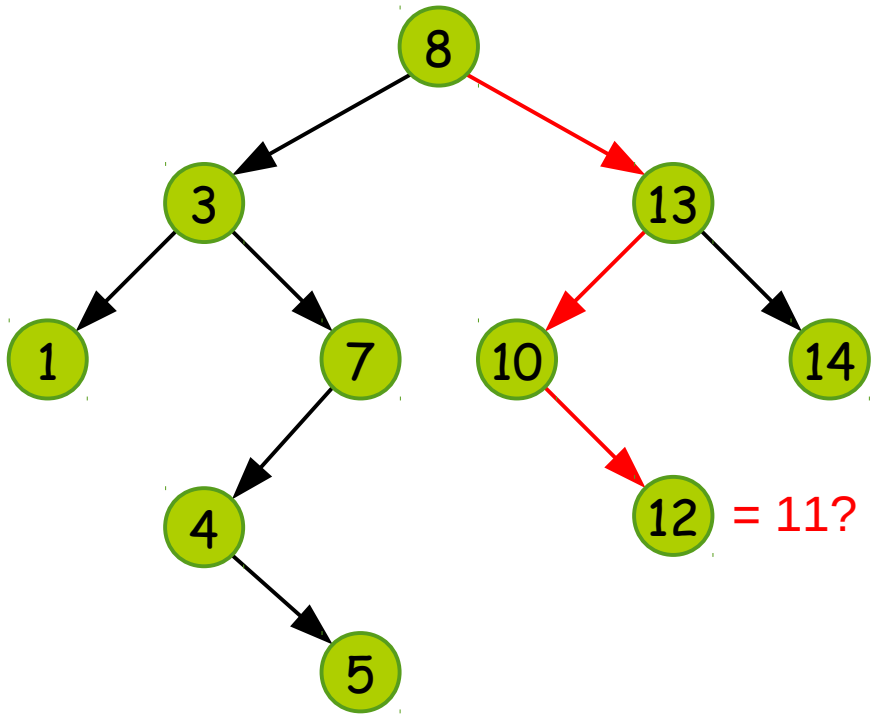
1. procuramos pelo elemento
2. vamos percorrer um caminho da raiz até uma folha



Inserindo um elemento

Inserir o número 11

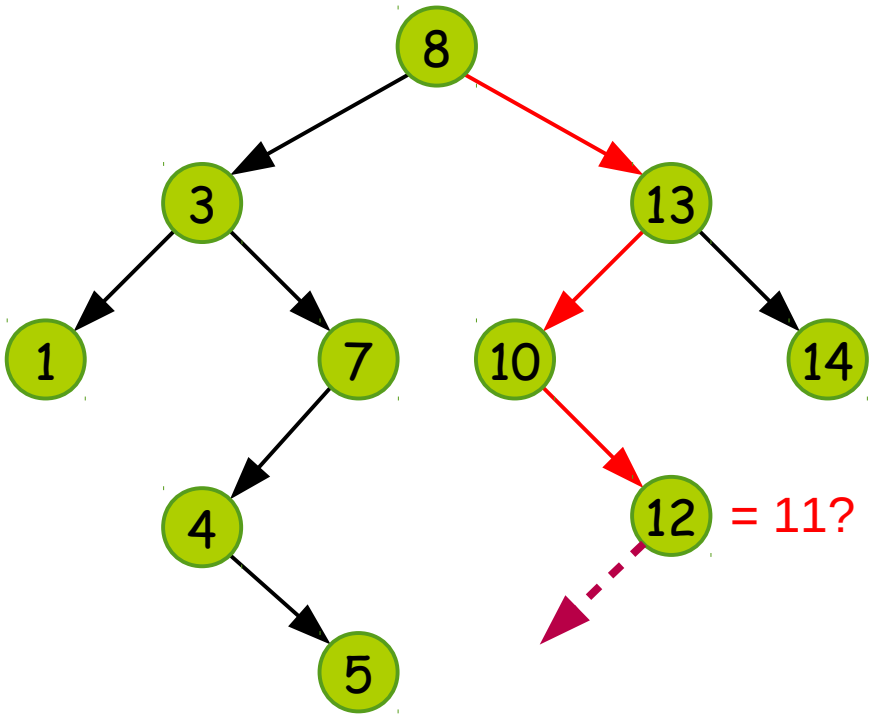
1. procuramos pelo elemento
2. vamos percorrer um caminho da raiz até uma folha



Inserindo um elemento

Inserir o número 11

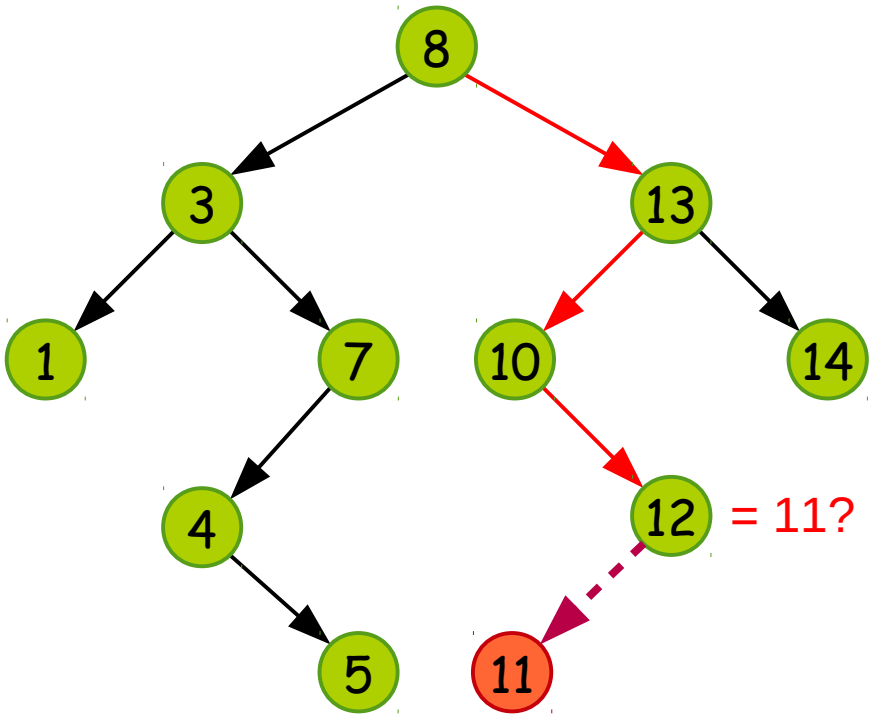
1. procuramos pelo elemento
2. vamos percorrer um caminho da raiz até uma folha
3. como o elemento não existe ainda, vamos chegar a uma referência vazia



Inserindo um elemento

Inserir o número 11

1. procuramos pelo elemento
2. vamos percorrer um caminho da raiz até uma folha
3. como o elemento não existe ainda, vamos chegar a uma referência vazia
4. inserimos o novo elemento



Implementação da inserção

Inserção

```
void inserir(NoArv **r, int chave, int dado) {
    // desce a árvore
    while (*r) {
        if ((*r)->chave < chave)
            r = &((*r)->dir);
        else
            r= &((*r)->esq);
    }
```

Implementação da inserção

Inserção

```
void inserir(NoArv **r, int chave, int dado) {
    // desce a árvore
    while (*r) {
        if ((*r)->chave < chave)
            r = &((*r)->dir);
        else
            r= &((*r)->esq);
    }

    // *r agora é o ponteiro "tracejado"
    *r = malloc(sizeof(NoArv));
    (*r)->esq = (*r)->dir = NULL;
    (*r)->chave = chave;
    (*r)->dado = dado;
}
```

Implementação da inserção

Inserção

```
void inserir(NoArv **r, int chave, int dado) {
    // desce a árvore
    while (*r) {
        if ((*r)->chave < chave)
            r = &((*r)->dir);
        else
            r= &((*r)->esq);
    }

    // *r agora é o ponteiro "tracejado"
    *r = malloc(sizeof(NoArv));
    (*r)->esq = (*r)->dir = NULL;
    (*r)->chave = chave;
    (*r)->dado = dado;
}
```

1) Por que usamos ponteiros duplos?

Implementação da inserção

Inserção

```
void inserir(NoArv **r, int chave, int dado) {
    // desce a árvore
    while (*r) {
        if ((*r)->chave < chave)
            r = &((*r)->dir);
        else
            r= &((*r)->esq);
    }

    // *r agora é o ponteiro "tracejado"
    *r = malloc(sizeof(NoArv));
    (*r)->esq = (*r)->dir = NULL;
    (*r)->chave = chave;
    (*r)->dado = dado;
}
```

1) Por que usamos ponteiros duplos? **2)** E se o elemento existisse?

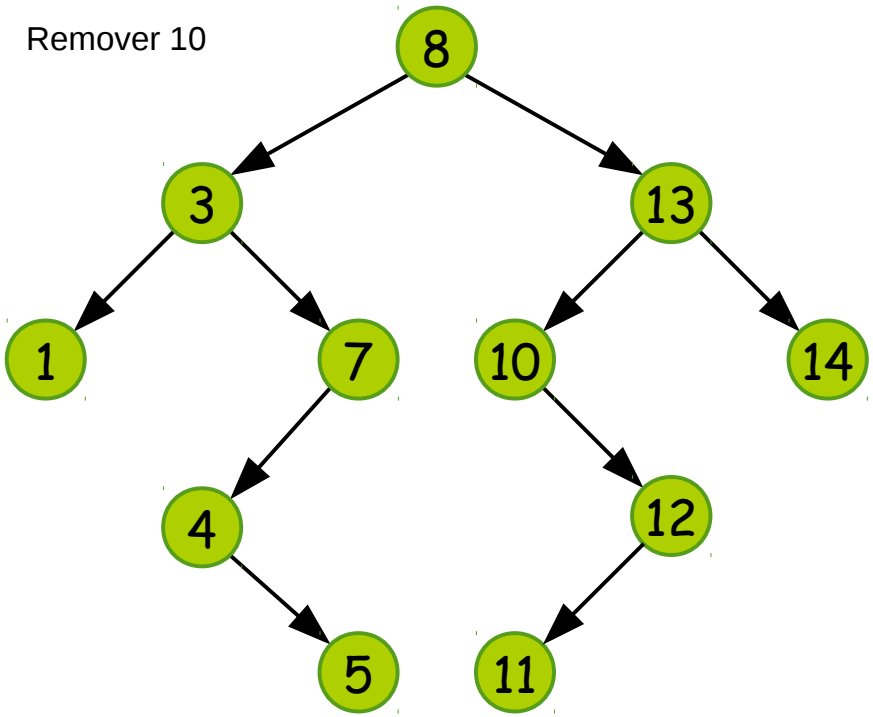
Removendo um elemento

Temos 2 casos!

Removendo um elemento

Temos 2 casos!

Caso 1: pelo menos um filho é vazio

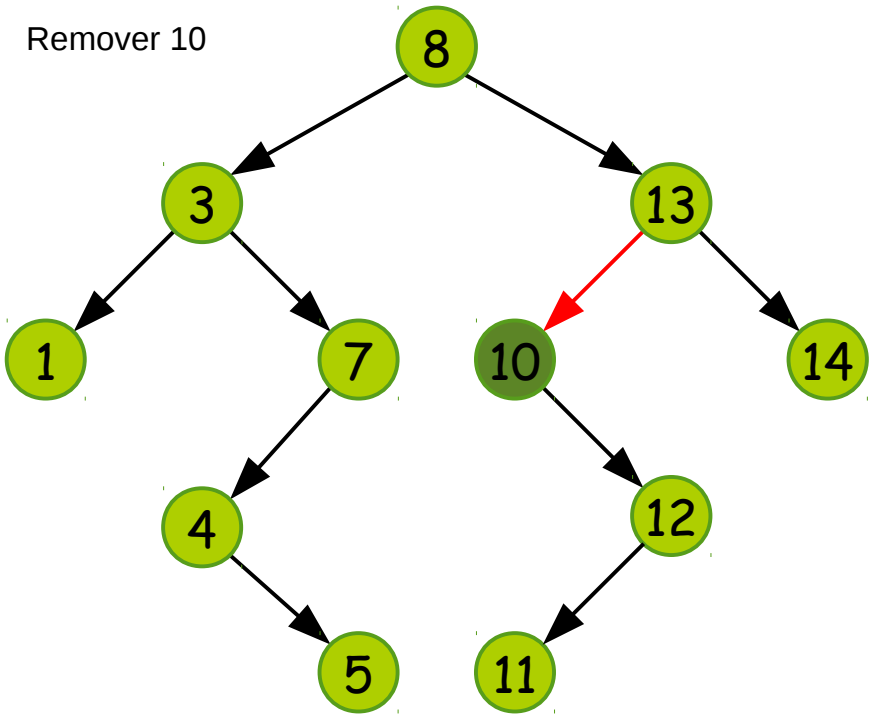


Removendo um elemento

Temos 2 casos!

Caso 1: pelo menos um filho é vazio

1. procuramos a referência para o elemento

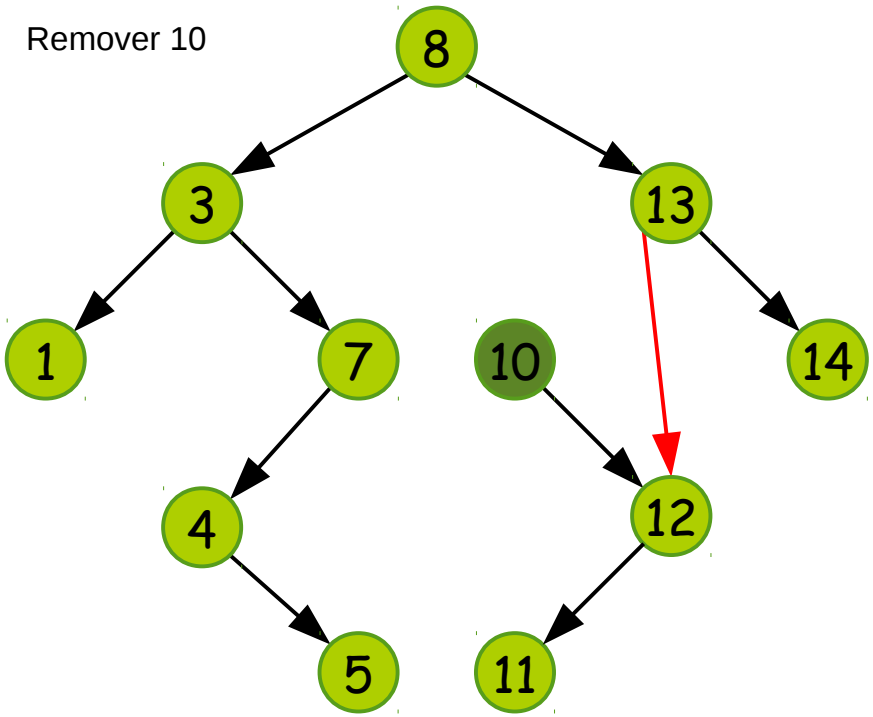


Removendo um elemento

Temos 2 casos!

Caso 1: pelo menos um filho é vazio

1. procuramos a referência para o elemento
2. "saltamos" o ponteiro

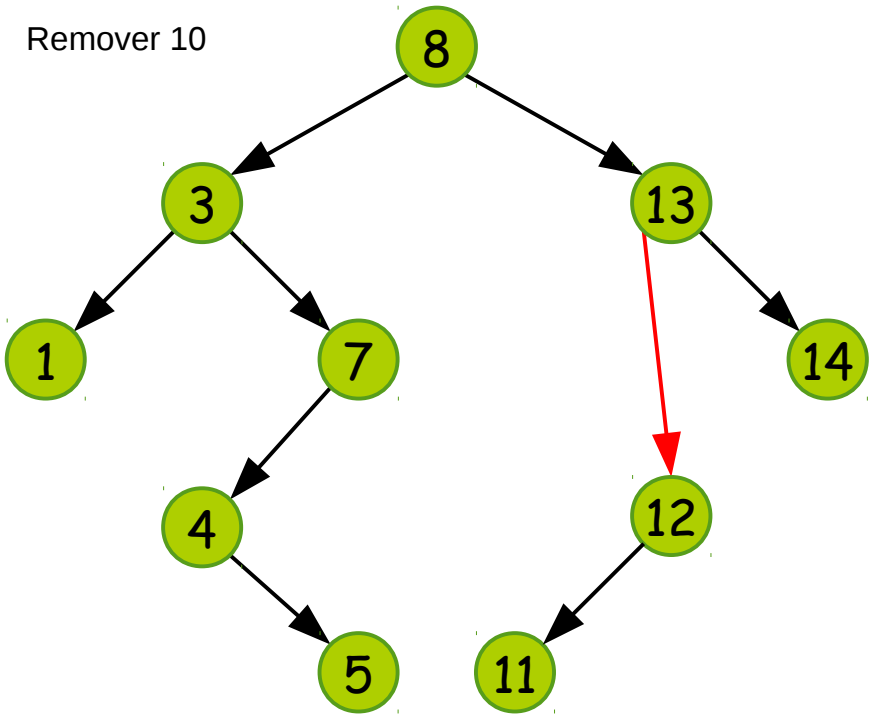


Removendo um elemento

Temos 2 casos!

Caso 1: pelo menos um filho é vazio

1. procuramos a referência para o elemento
2. "saltamos" o ponteiro
3. liberamos a memória



Implementação caso 1

Função auxiliar para caso 1

```
// *r aponta para nó a ser removido com subárvore vazia
void remover_caso1(NoArv **r) {
    NoArv *rem;
    rem = *r; // guarda nó a ser removido
```

Implementação caso 1

Função auxiliar para caso 1

```
// *r aponta para nó a ser removido com subárvore vazia
void remover_caso1(NoArv **r) {
    NoArv *rem;
    rem = *r; // guarda nó a ser removido

    // subárvore esquerda de rem vazia
    if (rem->esq == NULL)
        *r = rem->dir;
```

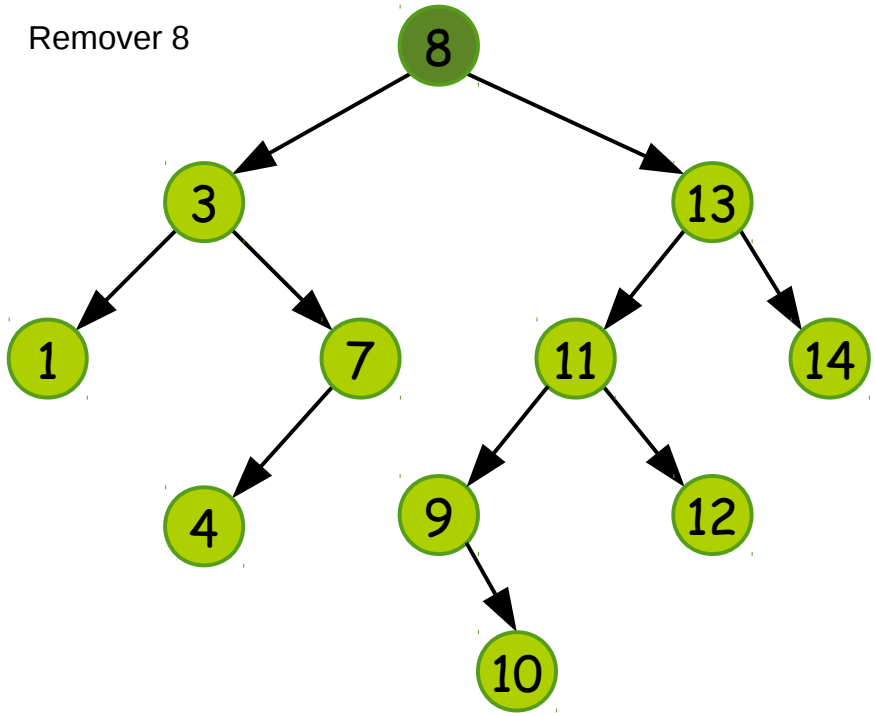
Implementação caso 1

Função auxiliar para caso 1

```
// *r aponta para nó a ser removido com subárvore vazia
void remover_caso1(NoArv **r) {
    NoArv *rem;
    rem = *r; // guarda nó a ser removido

    // subárvore esquerda de rem vazia
    if (rem->esq == NULL)
        *r = rem->dir;
    // subárvore direita de rem vazia
    else
        *r = rem->esq;

    free(rem);
}
```

Removendo um elemento

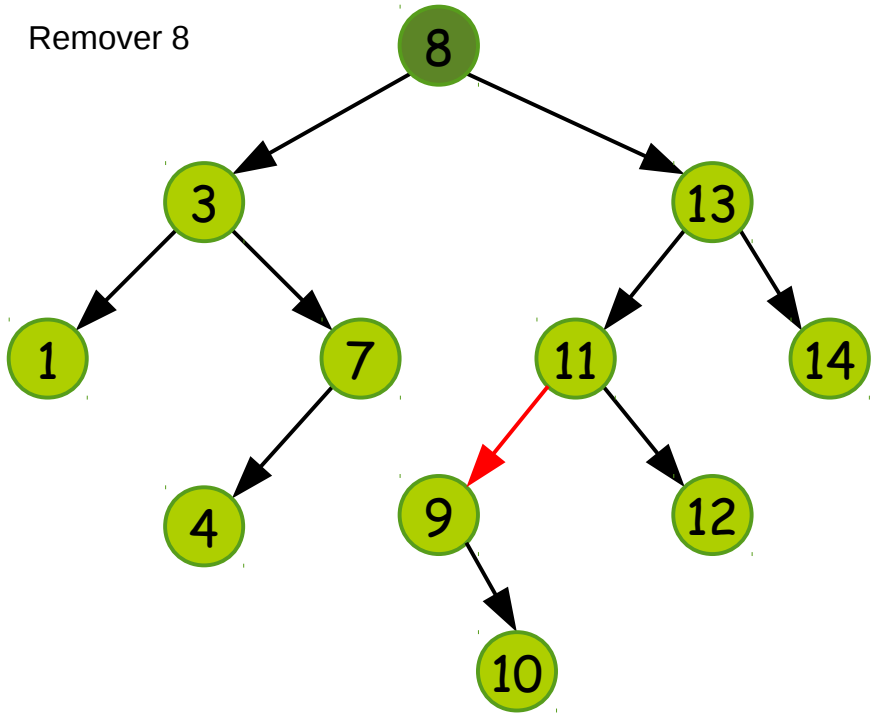
Caso 2: dois filhos não vazios



Removendo um elemento

Caso 2: dois filhos não vazios

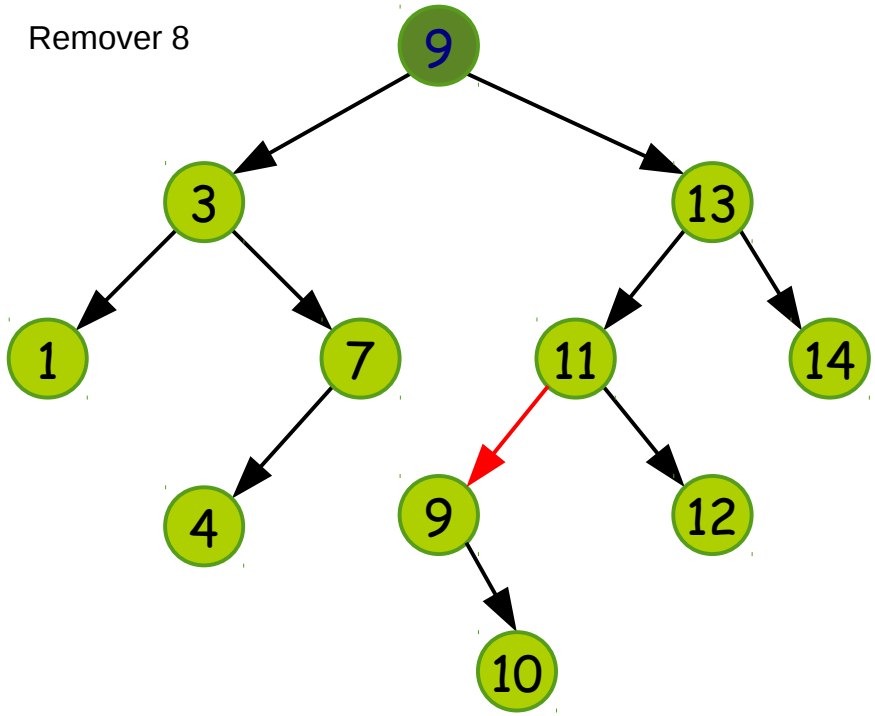
1. procuramos pelo ponteiro para o **sucessor**



Removendo um elemento

Caso 2: dois filhos não vazios

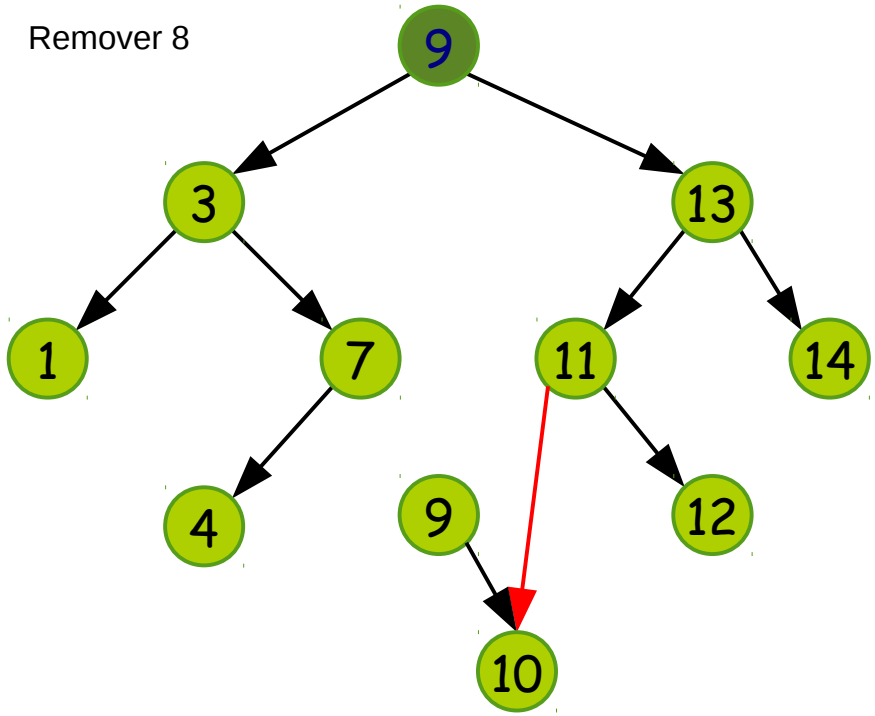
1. procuramos pelo ponteiro para o **sucessor**
2. copiamos os dados do sucessor



Removendo um elemento

Caso 2: dois filhos não vazios

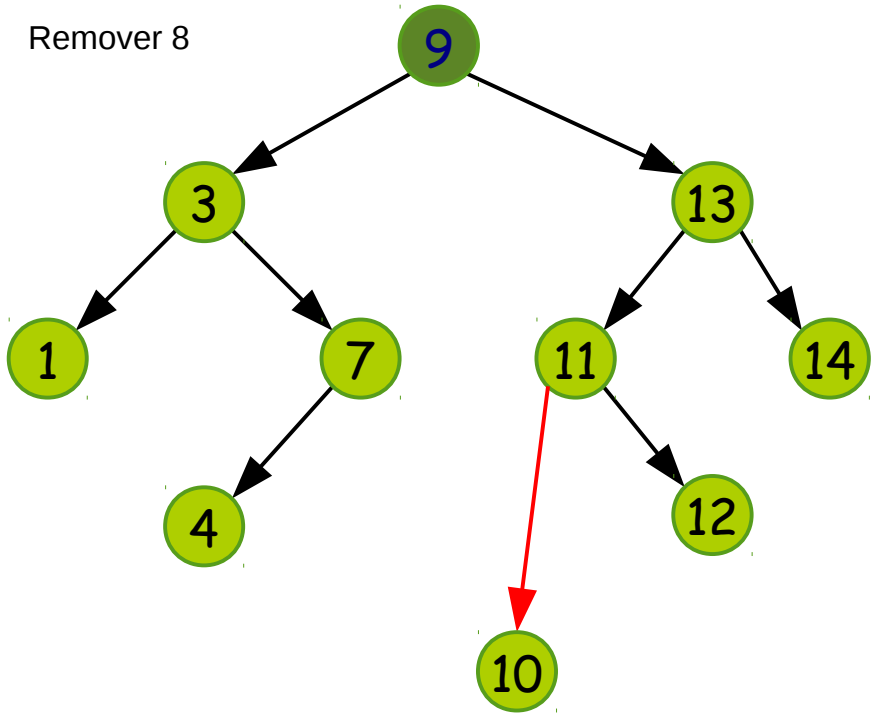
1. procuramos pelo ponteiro para o **sucessor**
2. copiamos os dados do sucessor
3. removemos o antigo nó do sucessor (como no caso 1)



Removendo um elemento

Caso 2: dois filhos não vazios

1. procuramos pelo ponteiro para o **sucessor**
2. copiamos os dados do sucessor
3. removemos o antigo nó do sucessor (como no caso 1)



Implementação caso 2

Função auxiliar para caso 2

```
// rem aponta para nó a ser removido com duas subárvores
void remover_caso2(NoArv *rem) {
    NoArv **suc; // sucessor
```

Implementação caso 2

Função auxiliar para caso 2

```
// rem aponta para nó a ser removido com duas subárvores
void remover_caso2(NoArv *rem) {
    NoArv **suc; // sucessor

    // procura sucessor
    suc = &(rem->dir);
    while((*suc)->esq)
        suc = &((*suc)->esq);
```

Implementação caso 2

Função auxiliar para caso 2

```
// rem aponta para nó a ser removido com duas subárvores
void remover_caso2(NoArv *rem) {
    NoArv **suc; // sucessor

    // procura sucessor
    suc = &(rem->dir);
    while((*suc)->esq)
        suc = &((*suc)->esq);

    // copia e sobrescreve rem
    rem->chave = (*suc)->chave;
    rem->dado = (*suc)->dado;
```

Implementação caso 2

Função auxiliar para caso 2

```
// rem aponta para nó a ser removido com duas subárvores
void remover_caso2(NoArv *rem) {
    NoArv **suc; // sucessor

    // procura sucessor
    suc = &(rem->dir);
    while((*suc)->esq)
        suc = &((*suc)->esq);

    // copia e sobrescreve rem
    rem->chave = (*suc)->chave;
    rem->dado = (*suc)->dado;

    // apaga nó do sucessor
    remover_caso1(suc);
}
```

Implementação da remoção

Combinando casos

```
void remover(NoArv **r, int chave) {
```

Implementação da remoção

Combinando casos

```
void remover(NoArv **r, int chave) {

    // procura nó a ser removido
    while ((*r)->chave != chave) {
        if ((*r)->chave < chave)
            r = &((*r)->dir);
        else
            r = &((*r)->esq);
    }
```

Implementação da remoção

Combinando casos

```
void remover(NoArv **r, int chave) {

    // procura nó a ser removido
    while ((*r)->chave != chave) {
        if ((*r)->chave < chave)
            r = &((*r)->dir);
        else
            r = &((*r)->esq);
    }

    // escolhe caso
    if (!(*r)->dir || !(*r)->esq)
        remover_caso1(r);
    else
        remover_caso2(*r);
}
```

Imprimindo elementos em ordem

Imprimir

```
void imprimir_ordem(NoArv *arv) {

    if (arv) {
        imprimir_ordem(arv->esq);
        printf("%d ", arv->chave);
        imprimir_ordem(arv->dir);
    }

}
```

Exercício

1. Implemente a operação antecessor.
2. Escreva uma função para imprimir as chaves de uma árvore de busca em ordem inversa.
3. Escreva uma função para imprimir todos os elementos em um determinado intervalo.
4. (desafio) Escreva uma função que receba duas árvores de busca e imprima todos os elementos das duas árvores em ordem. Escreva a função mais eficiente que você conseguir. (Dica: seria mais fácil com listas?)