

AULA 19

Arvores binárias



Fonte: <https://www.tumblr.com/>

PF 14

<http://www.ime.usp.br/~pf/algoritmos/aulas/bint.html>

Árvore binárias

Uma **árvore binária** (= *binary tree*) é um conjunto de **nós** que satisfaz certas condições.

Cada **nó** tem três campos:

```
typedef struct celula No;

struct celula {
    int conteudo; /* tipo devia ser Item */
    No *esq;
    No *dir;
};

No x, y;
```

Pais e filhos

Os campos `esq` e `dir` dão estrutura à árvore.

Se `x.esq == y`, então `y` é o **filho esquerdo** de `x`.

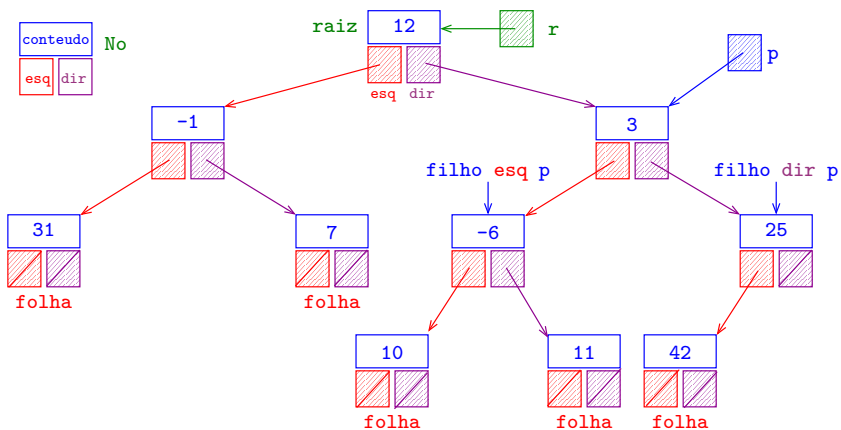
Se `x.dir == y`, então `y` é o **filho direito** de `x`.

Assim, `x` é o **pai** de `y` se
`x.esq == y` ou `x.dir == y`.

Uma **folha** é um nó sem filhos.

Ou seja, se `x.esq == NULL` e `x.dir == NULL`
então `x` é uma **folha**.

Ilustração de uma árvore binária



Árvores e subárvores

Suponha que r e p são (endereços de/ponteiros para) nós.

p é **descendente** de r

se p pode ser alcançado pela iteração dos comandos

$$p = p \rightarrow \text{esq}; \quad p = p \rightarrow \text{dir};$$

em qualquer ordem.

Um nó r juntamente com todos os seus descendentes é uma **árvore binária** e r é dito a **raiz** (= *root*) da árvore.

Para qualquer nó p , $p \rightarrow \text{esq}$ é a raiz da **subárvore esquerda** de p e $p \rightarrow \text{dir}$ é a raiz da **subárvore direita** de p .

Endereço de uma árvore

O endereço de uma árvore binária é o endereço de sua raiz.

```
typedef No *Arvore;  
Arvore r;
```

Endereço de uma árvore

O endereço de uma árvore binária é o endereço de sua raiz.

```
typedef No *Arvore;  
Arvore r;
```

Um objeto `r` é uma árvore binária se

- ▶ `r == NULL` ou
- ▶ `r->esq` e `r->dir` são árvores binárias.

Maneiras de varrer uma árvore

Existem várias maneiras de percorrermos uma árvore binária. Talvez as mais tradicionais sejam:

- ▶ *inorder traversal*: esquerda-raiz-direita (e-r-d);
- ▶ *preorder traversal*: raiz-esquerda-direita (r-e-d);
- ▶ *posorder traversal*: esquerda-direita-raiz (e-d-r);

esquerda-raiz-direita

Visitamos

1. a subárvore **esquerda** da **raiz**, em ordem **e-r-d**;
2. depois a **raiz**;
3. a subárvore **direita** da **raiz**, em ordem **e-r-d**;

esquerda-raiz-direita

Visitamos

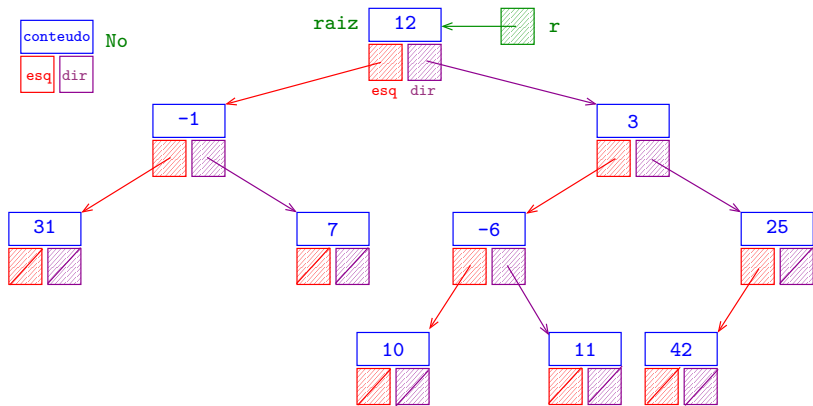
1. a subárvore **esquerda** da **raiz**, em ordem **e-r-d**;
2. depois a **raiz**;
3. a subárvore **direita** da **raiz**, em ordem **e-r-d**;

```
void inOrdem(Arvore r) {  
    if (r != NULL) {  
        inOrdem(r->esq);  
        printf("%d\n", r->conteudo);  
        inOrdem(r->dir);  
    }  
}
```

esquerda-raiz-direita versão iterativa

```
void inOrdem(Arvore r) {
    stackInit();
    while (r != NULL || !stackEmpty()) {
        if (r != NULL) {
            stackPush(r);
            r = r->esq;
        }
        else {
            r = stackPop();
            printf("%d\n", r->conteudo);
            r = r->dir;
        }
    }
}
```

Ilustração de percursos em árvores binárias

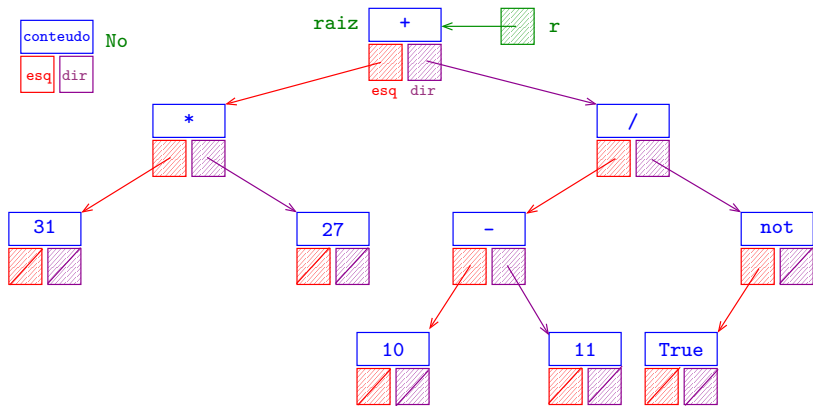


in-ordem (e-r-d): 31 -1 7 12 10 -6 11 3 42 25

pré-ordem (r-e-d): 12 -1 31 7 3 -6 10 11 25 42

pós-ordem (e-d-r): 31 7 -1 10 11 -6 42 25 3 12

Ilustração de percursos em árvores binárias



in-ordem (e-r-d): 31 * 27 + 10 - 11 / True not

pré-ordem (r-e-d): + * 31 27 / - 10 11 not True

pós-ordem (e-d-r): 31 27 * 10 11 - True not / +

Primeiro nó esquerda-raiz-direita

Recebe a raiz r de uma árvore binária não vazia e retorna o primeiro nó na ordem e-r-d.

Primeiro nó esquerda-raiz-direita

Recebe a raiz `r` de uma árvore binária não vazia e retorna o primeiro nó na ordem e-r-d.

```
No *primeiro(Arvore r) {  
    while (r->esq != NULL)  
        r = r->esq;  
    return r;  
}
```


Altura

A **altura** de p é o número de passos do **mais longo caminho** que leva de p até uma folha.

A altura de uma **árvore** é a altura da sua **raiz**.

Altura da árvore **vazia** é -1 .

Altura

A **altura** de **p** é o número de passos do **mais longo caminho** que leva de **p** até uma folha.

A altura de uma **árvore** é a altura da sua **raiz**.

Altura da árvore **vazia** é -1 .

```
#define MAX(a,b) ((a) > (b)? (a): (b))
int altura(Arvore r) {
    if (r == NULL) return -1;
    else {
        int he = altura(r->esq);
        int hd = altura(r->dir);
        return MAX(he,hd) + 1;
    }
}
```

Árvores balanceadas

A altura de uma **árvore** com n nós é um número entre $\lg(n)$ e n .

Uma **árvore binária** é **balanceada** (ou **equilibrada**) se, em cada um de seus nós, as subárvores **esquerda** e **direita** tiverem *aproximadamente* a mesma altura.

Árvores balanceadas têm altura *próxima* de $\lg(n)$.

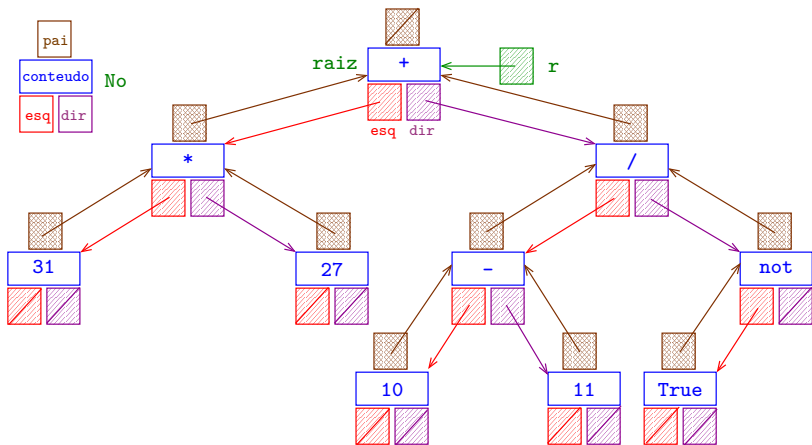
O **consumo de tempo** dos algoritmos que manipulam **árvores binárias** dependem frequentemente da **altura** da **árvore**.

Nós com campo pai

Em algumas aplicações é conveniente ter acesso imediato ao pai de qualquer nó.

```
typedef struct celula Celula;
struct celula {
    int conteudo;           /* tipo devia ser Item */
    Celula *pai;
    Celula *esq;
    Celula *dir;
};
typedef Celula No;
typedef No *Arvore;
```

Ilustração de nós com campo pai



Sucessor e predecessor

Recebe o endereço p de um nó de uma **árvore binária** não vazia e retorna o seu **sucessor** na ordem **e-r-d**.

Sucessor e predecessor

Recebe o endereço `p` de um nó de uma **árvore binária** não vazia e retorna o seu **sucessor** na ordem **e-r-d**.

```
No *sucessor(No *p) {
    if (p->dir != NULL) {
        No *q = p->dir;
        while (q->esq != NULL) q = q->esq;
        return q;
    }
    /* mais a esquerda da subárvore direita */
}
```

Sucessor e predecessor

Recebe o endereço `p` de um nó de uma **árvore binária** não vazia e retorna o seu **sucessor** na ordem **e-r-d**.

```
No *sucessor(No *p) {
    if (p->dir != NULL) {
        No *q = p->dir;
        while (q->esq != NULL) q = q->esq;
        return q;           /* mais a esquerda da subárvore direita */
    }
    while (p->pai != NULL && p->pai->dir == p)
        p = p->pai;
    return p->pai;
}
```

Exercício: escreva a função que retorna o **predecessor**.

Mais exercícios

Escreva uma função que recebe o endereço p de um nó de uma **árvore binária** e

1. devolve o número de nós da árvore.
2. devolve o maior número que aparece na árvore ($-\infty$ se a árvore está vazia).
3. espelha a árvore.
4. devolve uma cópia da árvore.
5. libera todos os nós da árvore.

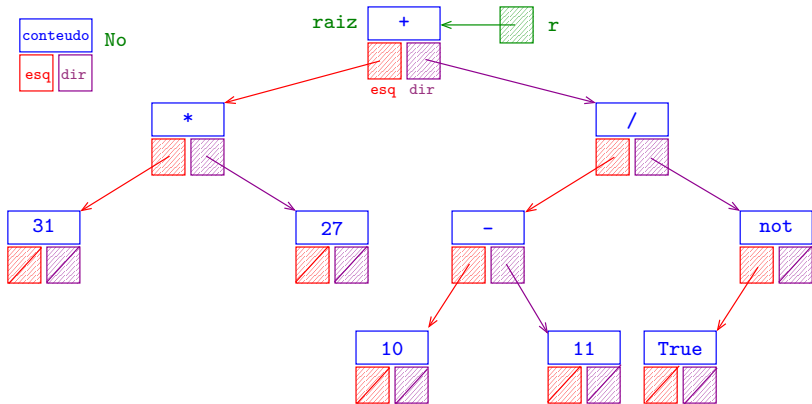
Mais exercícios

Escreva uma função que recebe o endereço p de um nó de uma **árvore binária** e

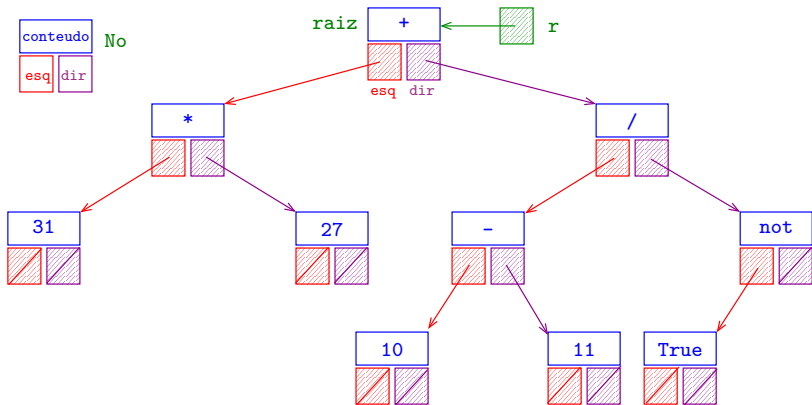
1. devolve o número de nós da árvore.
2. devolve o maior número que aparece na árvore ($-\infty$ se a árvore está vazia).
3. espelha a árvore.
4. devolve uma cópia da árvore.
5. libera todos os nós da árvore.

Vamos fazer um agora? Qual?

Árvores com expressões



Árvores com expressões



Dada uma expressão aritmética, como construir a árvore binária que representa esta expressão?

Árvores com expressões

Converte a expressão na notação posfixa.

Depois processa a expressão, como no EP3, criando um nó novo da árvore para cada símbolo.

Árvores com expressões

Converte a expressão na notação posfixa.

Depois processa a expressão, como no EP3, criando um nó novo da árvore para cada símbolo.

Se o símbolo é uma variável, ele é preenchido e empilhado.

Árvores com expressões

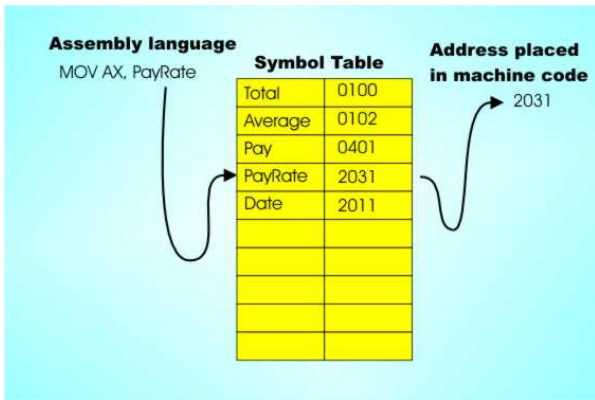
Converte a expressão na notação posfixa.

Depois processa a expressão, como no EP3, criando um nó novo da árvore para cada símbolo.

Se o símbolo é uma variável, ele é preenchido e empilhado.

Se é um operador, desempilhamos as árvores que correspondem às expressões da direita e da esquerda, fazemos o novo nó apontar para estas árvores, e empilhamos o novo nó, que é a raiz da árvore resultante.

Tabelas de Símbolos



Fonte: <http://www.i-programmer.info/>

Tabelas de símbolos (PF)

Tabela de símbolos

Uma **tabela de símbolos** (= *symbol table* = *dictionary*) é um conjunto de **objetos** (*itens*), cada um dotado de uma **chave** (= *key*) e de um **valor** (= *value*).

As chaves podem ser números inteiros ou *strings* ou outro tipo de dados.

Tabela de símbolos

Uma **tabela de símbolos** (= *symbol table* = *dictionary*) é um conjunto de **objetos** (*itens*), cada um dotado de uma **chave** (= *key*) e de um **valor** (= *value*).

As chaves podem ser números inteiros ou *strings* ou outro tipo de dados.

Uma tabela de símbolos está sujeita a **dois tipos de operações**:

- ▶ **inserção**: consiste em introduzir um objeto na tabela;
- ▶ **busca**: consiste em encontrar um elemento que tenha uma dada chave.

Problema

Problema: Organizar uma **tabela de símbolos** de maneira que as operações de **inserção** e **busca** sejam *razoavelmente eficientes*.

Problema

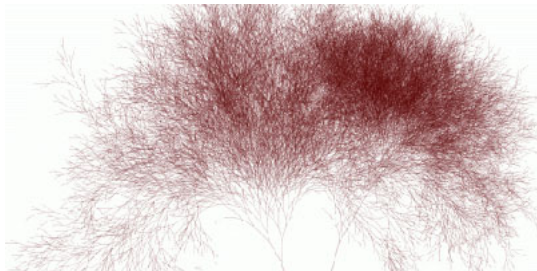
Problema: Organizar uma **tabela de símbolos** de maneira que as operações de **inserção** e **busca** sejam *razoavelmente eficientes*.

Em geral, uma organização que permite **inserções** rápidas impede **buscas** rápidas e vice-versa.

Já vimos como organizar tabelas de símbolos através de **vetores** e **lista encadeadas**.

Hoje:
mais uma maneira de organizar uma tabela de símbolos.

Árvores binárias de busca



Fonte: <http://infosthetics.com/archives/>

PF 15

<http://www.ime.usp.br/~pf/algoritmos/aulas/binst.html>

Árvore binárias de busca

Considere uma **árvore binária** cujos nós têm um campo **chave** (como **int** ou **String**, por exemplo).

```
typedef struct celula No;

struct celula {
    int conteudo;           /* tipo devia ser Item */
    int chave;             /* tipo devia ser Chave */
    No *esq;
    No *dir;
};
```

Árvore binárias de busca

Considere uma **árvore binária** cujos nós têm um campo **chave** (como **int** ou **String**, por exemplo).

```
typedef struct celula No;

struct celula {
    int conteudo;           /* tipo devia ser Item */
    int chave;             /* tipo devia ser Chave */
    No *esq;
    No *dir;
};

typedef No *Arvore;
No x, *p, *q, *r, *t;
```

Árvore binárias de busca

Uma **árvore binária** deste tipo é de **busca** (em relação ao campo **chave**) se, para cada nó x , $x.chave$ é

Árvore binárias de busca

Uma **árvore binária** deste tipo é de **busca** (em relação ao campo **chave**) se, para cada nó **x**, **x.chave** é

1. **maior ou igual** à chave de qualquer nó na subárvore **esquerda** de **x** e

Árvore binárias de busca

Uma **árvore binária** deste tipo é de **busca** (em relação ao campo **chave**) se, para cada nó x , $x.chave$ é

1. **maior ou igual** à chave de qualquer nó na subárvore **esquerda** de x e
2. **menor ou igual** à chave de qualquer nó na subárvore **direita** de x .

Árvore binárias de busca

Uma **árvore binária** deste tipo é de **busca** (em relação ao campo **chave**) se, para cada nó x , $x.chave$ é

1. **maior ou igual** à chave de qualquer nó na subárvore **esquerda** de x e
2. **menor ou igual** à chave de qualquer nó na subárvore **direita** de x .

Assim, se p é um nó qualquer então vale que

$$q \rightarrow \text{chave} \leq p \rightarrow \text{chave} \text{ e}$$

$$p \rightarrow \text{chave} \leq t \rightarrow \text{chave}$$

para todo nó q na subárvore **esquerda** de p
e todo nó t na subárvore **direita** de p .

Ilustração de uma árvore binária de busca

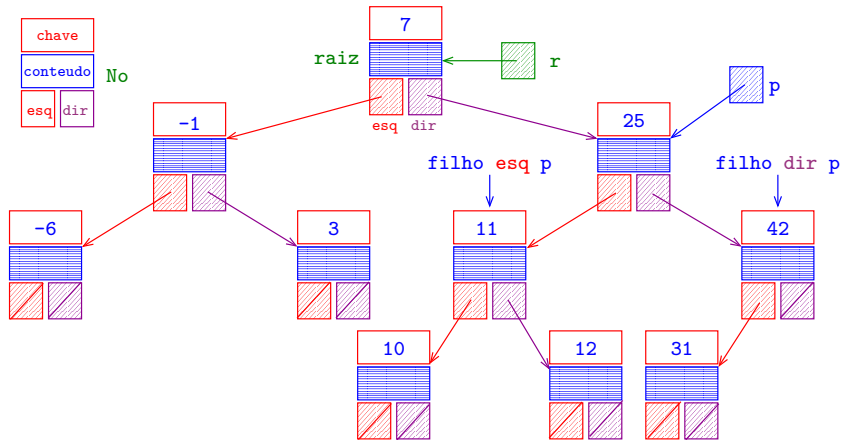
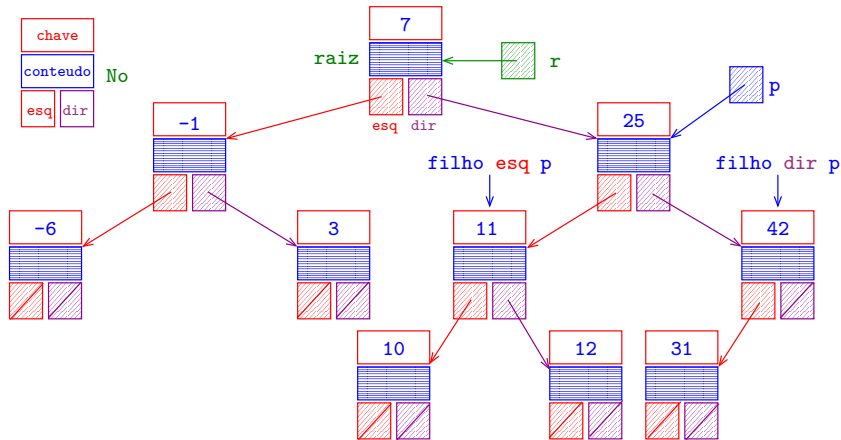


Ilustração de uma árvore binária de busca



in-ordem (e-r-d): -6 -1 3 7 10 11 12 25 31 42

Busca

Recebe um inteiro **k** e uma **árvore de busca r** e retorna um nó cuja chave é **k**; se tal nó não existe, retorna **NULL**.

Busca

Recebe um inteiro **k** e uma **árvore de busca r** e retorna um nó cuja chave é **k**; se tal nó não existe, retorna **NULL**.

```
No *busca(Arvore r, int k) {
    if (r == NULL || r->chave == k)
        return r;

    if (r->chave > k)
        return busca(r->esq, k);
    return busca(r->dir, k);
}
```

Busca versão iterativa

Recebe um inteiro **k** e uma árvore de busca **r** e retorna um nó cuja chave é **k**; se tal nó não existe, retorna **NULL**.

Busca versão iterativa

Recebe um inteiro `k` e uma árvore de busca `r` e retorna um nó cuja chave é `k`; se tal nó não existe, retorna `NULL`.

```
No *busca(Arvore r, int k) {
    while (r != NULL && r->chave != k)
        if (r->chave > k)
            r = r->esq;
        else
            r = r->dir;
    return r;
}
```

Inserção

Recebe uma árvore de busca r e um nó novo.
Insere o nó no lugar correto da árvore de modo que a árvore continue sendo de busca e retorna o endereço da nova árvore.

Inserção

Recebe uma árvore de busca `r` e um nó `novo`.
Insere o nó no lugar correto da árvore de modo que a árvore continue sendo de busca e retorna o endereço da nova árvore.

```
No *new(int chave, int conteudo, No *esq, No *dir) {  
    No *novo = mallocSafe(sizeof *novo);  
    novo->chave = chave;  
    novo->conteudo = conteudo;  
    novo->esq = esq;  
    novo->dir = dir;  
    return novo;  
}
```

Inserção

```
Arvore insere(Arvore r, No *novo) {  
    No *f, /* filho de p */  
    No *p; /* pai de f */  
    if (r == NULL) return novo;  
    f = r;  
    while (f != NULL) {  
        p = f;  
        if (f->chave > novo->chave)  
            f = f->esq;  
        else  
            f = f->dir;  
    }  
}
```

Inserção

```
/* novo sera uma folha
   novo sera filho de p */
if (p->chave > novo->chave)
    p->esq = novo;
else
    p->dir = novo;
return r;
}
```

Remoção

Recebe uma árvore de busca não vazia r . Remove a sua raiz e rearranja a árvore de modo que continue sendo de busca e retorna o endereço da nova árvore.

Remoção

```
Arvore removeRaiz(Arvore r) {  
    No *p, *q;  
    if (r->esq == NULL) {  
        q = r->dir; free(r); return q;  
    }  
}
```

Remoção

```
Arvore removeRaiz(Arvore r) {  
    No *p, *q;  
    if (r->esq == NULL) {  
        q = r->dir; free(r); return q;  
    }  
    /* encontre na subarvore r->esq  
       o nó q com maior valor */  
    p = r; q = r->esq;  
    while (q->dir != NULL) {  
        p = q;  
        q = q->dir;  
    }  
}
```


Remoção

```
/* q é o nó anterior a r na ordem e-r-d
   p é o pai de q */
if (p != r) {
    p->dir = q->esq;
    q->esq = r->esq;
}
q->dir = r->dir; free(r); return q;
}
```

Consumo de tempo

O consumo de tempo das funções `busca`, `insere` e `removeRaiz` é, no pior caso, proporcional à `altura` da `árvore`.

Consumo de tempo

O consumo de tempo das funções `busca`, `insere` e `removeRaiz` é, no pior caso, proporcional à `altura` da `árvore`.

Conclusão: interessa trabalhar com `árvores balanceadas`: `árvores AVL`, `árvores rubro-negras`, `treaps` ...

Aula que vem: `treaps`!