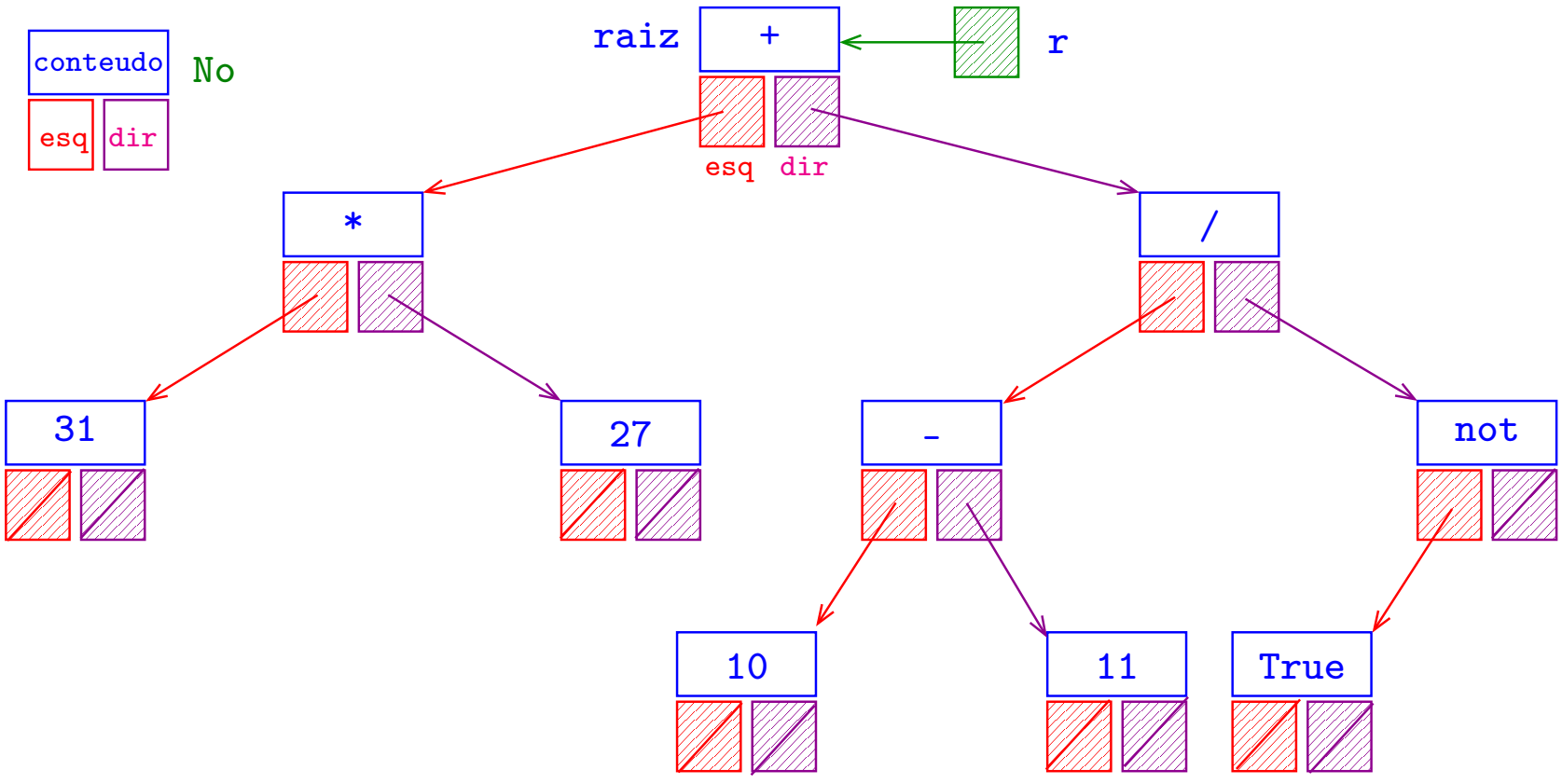
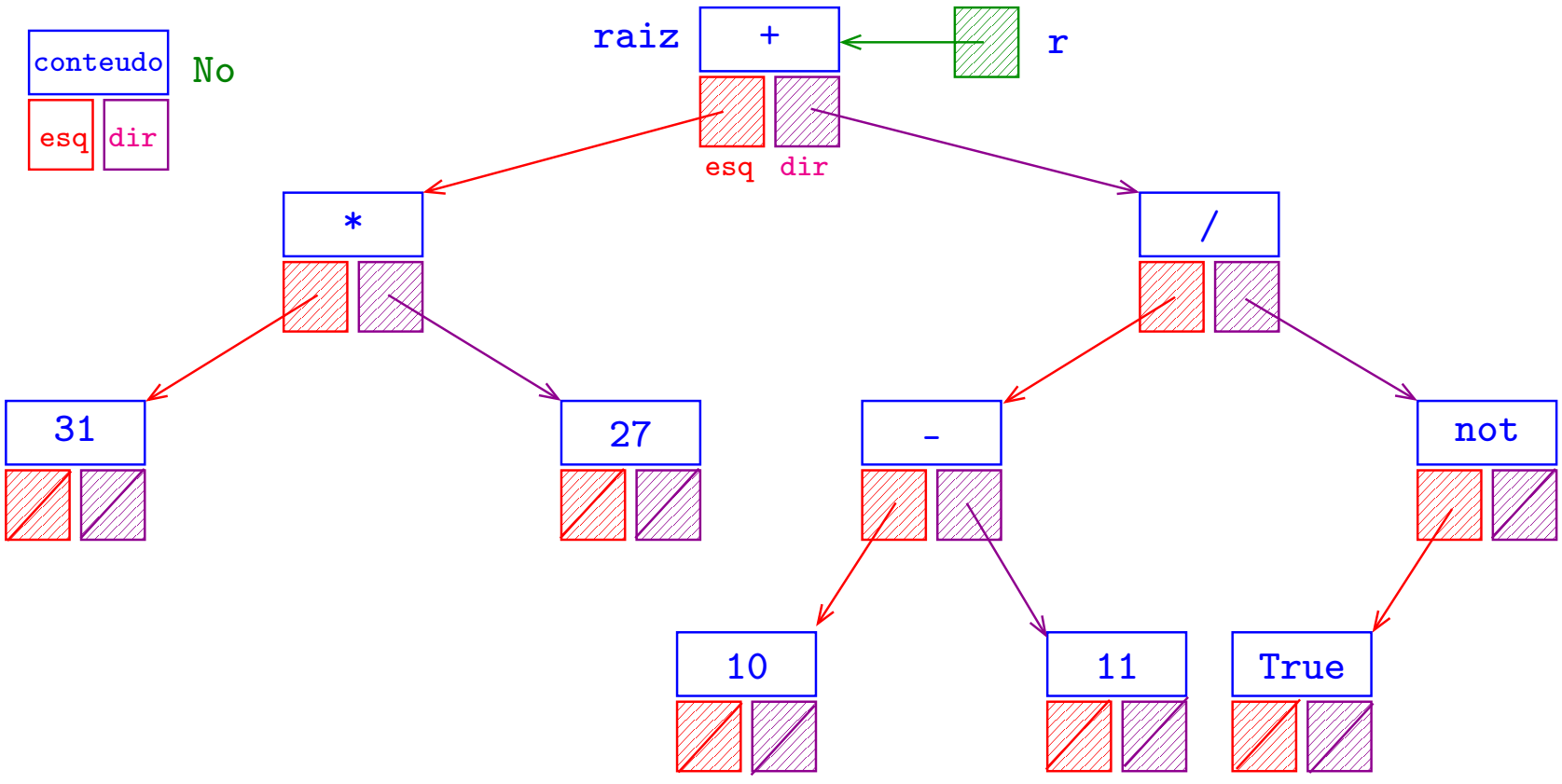
Melhores momentos

AULA 21

Árvores com expressões



Árvores com expressões



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

AULA 22

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

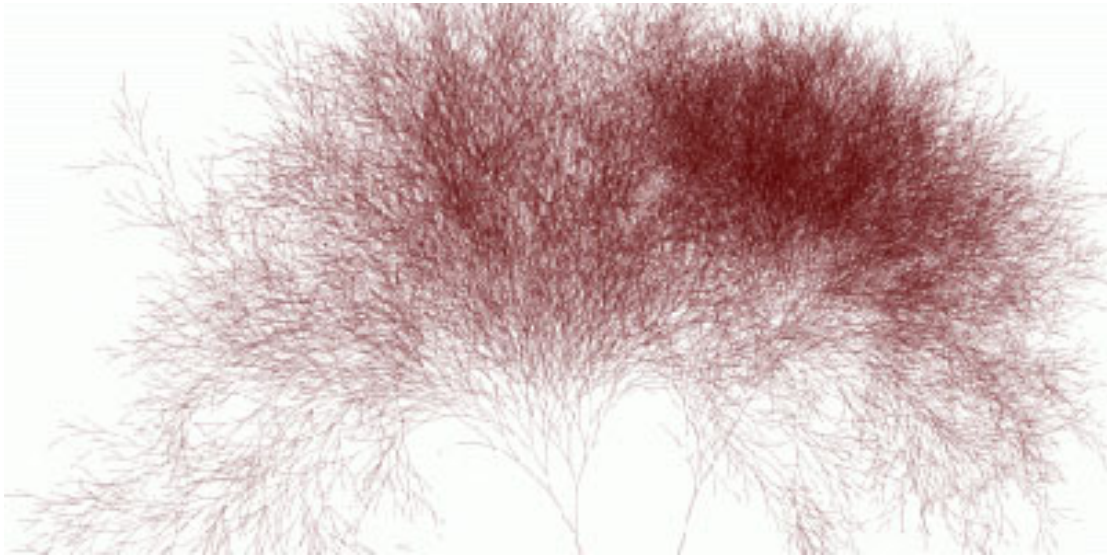
Problema: Organizar uma tabela de símbolos de maneira que as operações de inserção e busca sejam *razoavelmente eficientes*.

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

Já vimos como organizar tabelas de símbols 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 Celula;
struct celula {
   int conteudo; /* tipo devia ser Item*/
   int chave; /* tipo devia ser Chave*/
   Celula *esq;
   Celula *dir;
};
```

Árvore binárias de busca

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

```
typedef struct celula Celula;
struct celula {
   int conteudo; /* tipo devia ser Item*/
   int chave; /* tipo devia ser Chave*/
   Celula *esq;
   Celula *dir;
};
typedef Celula No;
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->chave <= p->chave e
p->chave <= t->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

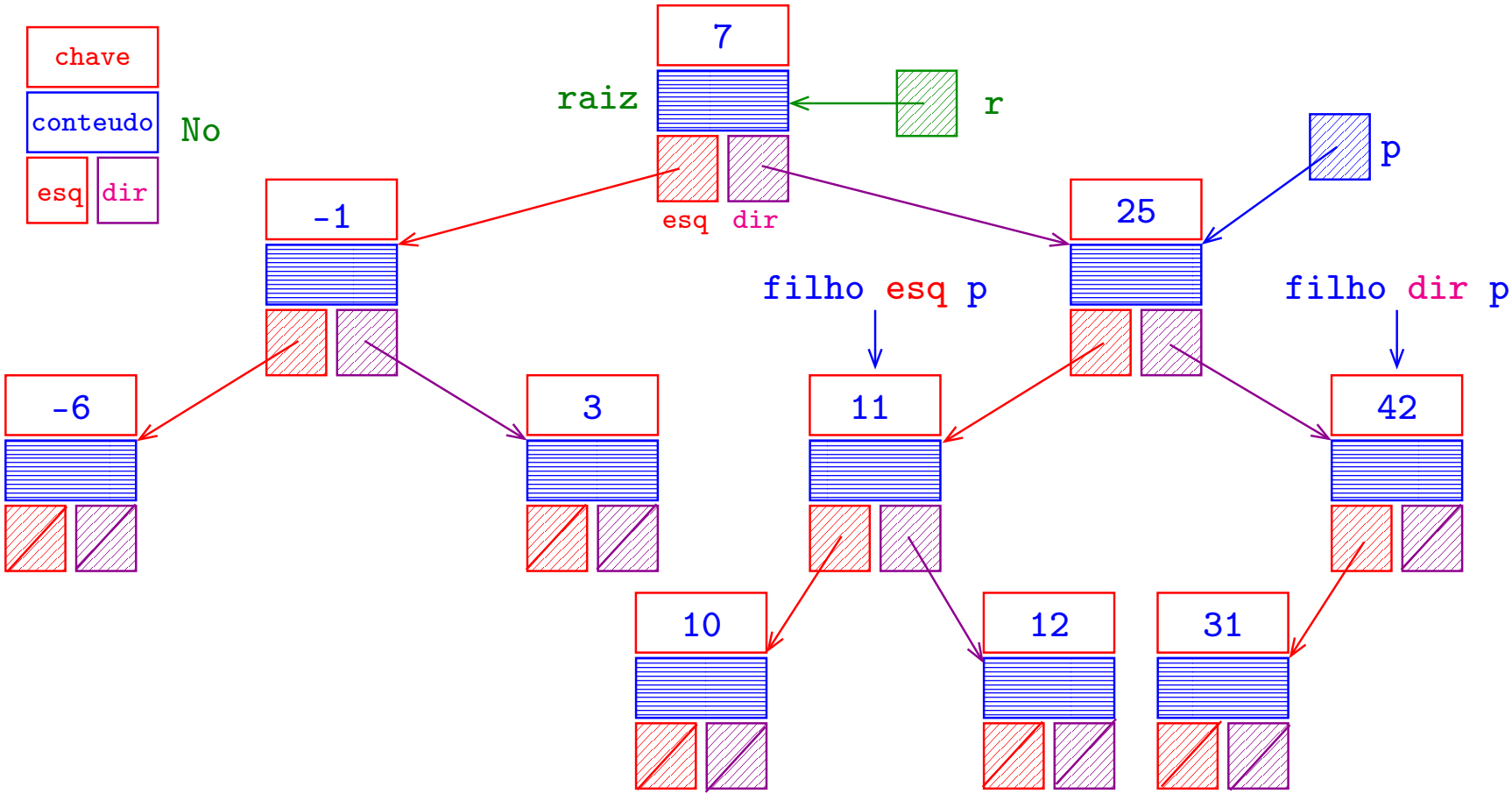
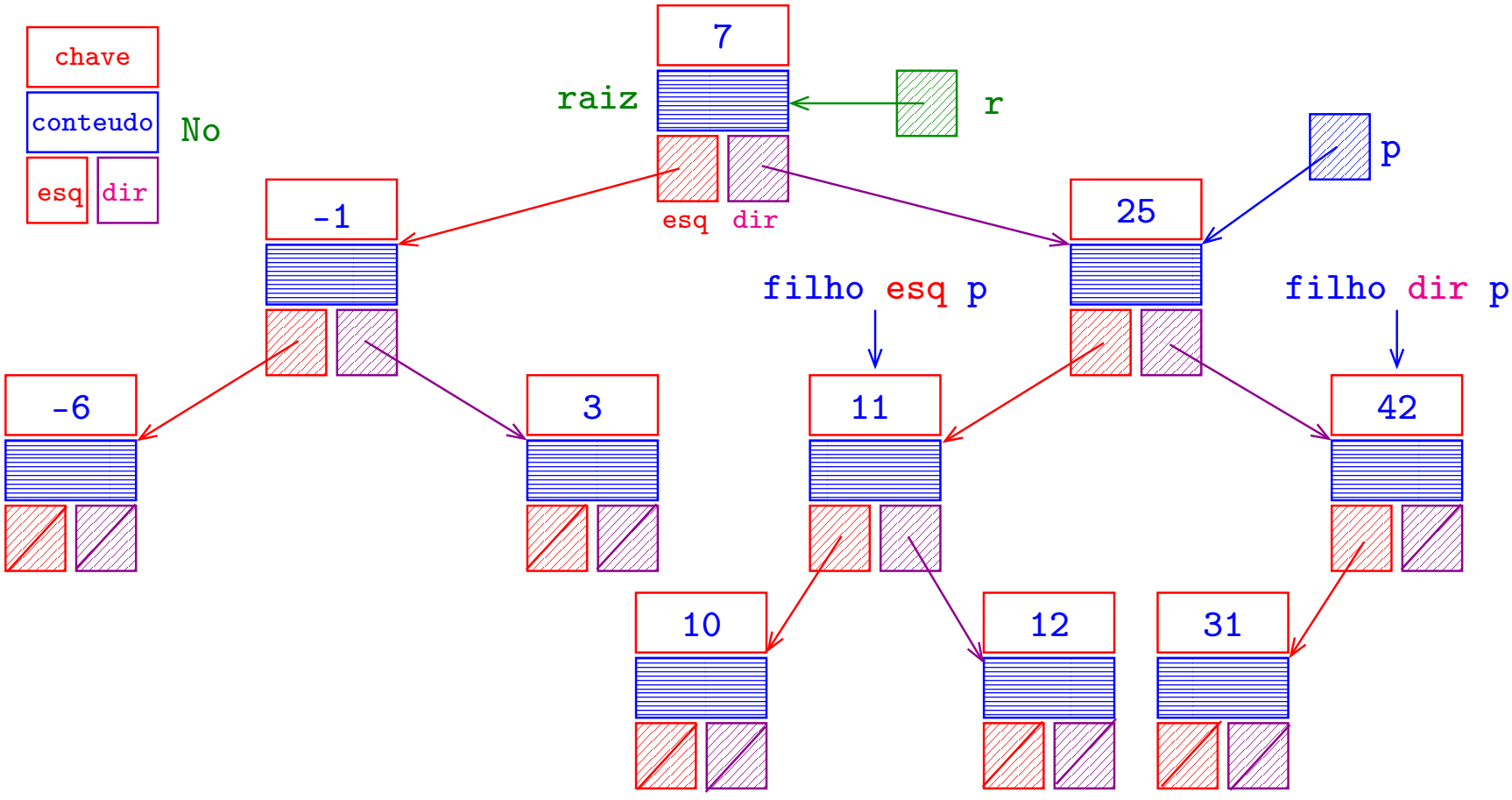


Ilustração de uma árvore binária de busca



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;
   }
   /* 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.

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

...