Melhores momentos

AULA 19

Backtracking

O esquema a seguir descreve o método *backtracking*.

Suponha que a solução de um problema
possa ser vista como uma sequência de decisões

$$\mathtt{x}[1], \mathtt{x}[2], \ldots, \mathtt{x}[\mathtt{n}].$$

Por exemplo, cada `x[k]` pode ser a posição
de uma rainha ou para qual posição mover o cavalo.

A relação invariante chave do método é algo como:

> *No início de cada iteração `x[1..k-1]` é uma "solução parcial" (que pode ou não ser parte de uma solução).*

Backtracking

```
k ← 1
enquanto k ≥ 1 faça
   procure valor para x[k] que ainda
   não foi testado e tal que x[1..k]
   é solução parcial
   se encontrou candidato para x[k] então
      k ← k + 1 (avança)
      se k = n + 1 então
         encontramos uma solução
         imprima x[1..n]
         k ← k − 1 (continua)
      senão k ← k − 1 (volta)
```

AULA 20

Enumeração de subsequências

PF 12

http://www.ime.usp.br/~pf/algoritmos/aulas/enum.html

Enumeração de subsequências

Problema: Enumerar todas as subsequências de `1,2,...,n`, ou seja, fazer uma lista em que cada subsequência aparece uma e uma só vez.

Exemplo: para `n` = 3 as subsequências são

```
1
1   2
1   2   3
1   3
2
2   3
3
```

Enumeração de subsequências

Exemplo: para `n` $= 4$ as subsequências são

```
1
1   2
1   2   3
1   2   3   4
1   2   4
1   3
1   3   4
1   4
2
2   3
2   3   4
2   4
3
3   4
4
```

subseqLex

A função subseqLex recebe n e imprime todas as subsequências não vazias de 1 . . n.

```
void subseqLex (int n) {
   int *s, k;
   s = mallocSafe((n+1) * sizeof(int));
   s[0] = 0;
   k = 0;
```

subseqLex

```
    while(1) {
        if (s[k] < n) {
            s[k+1] = s[k] + 1;
            k += 1;
        } else {
            s[k-1] += 1;
            k -= 1;
        }
        if (k == 0) break;
        imprima(s, k);
    }
    free(s);
}
```

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`.

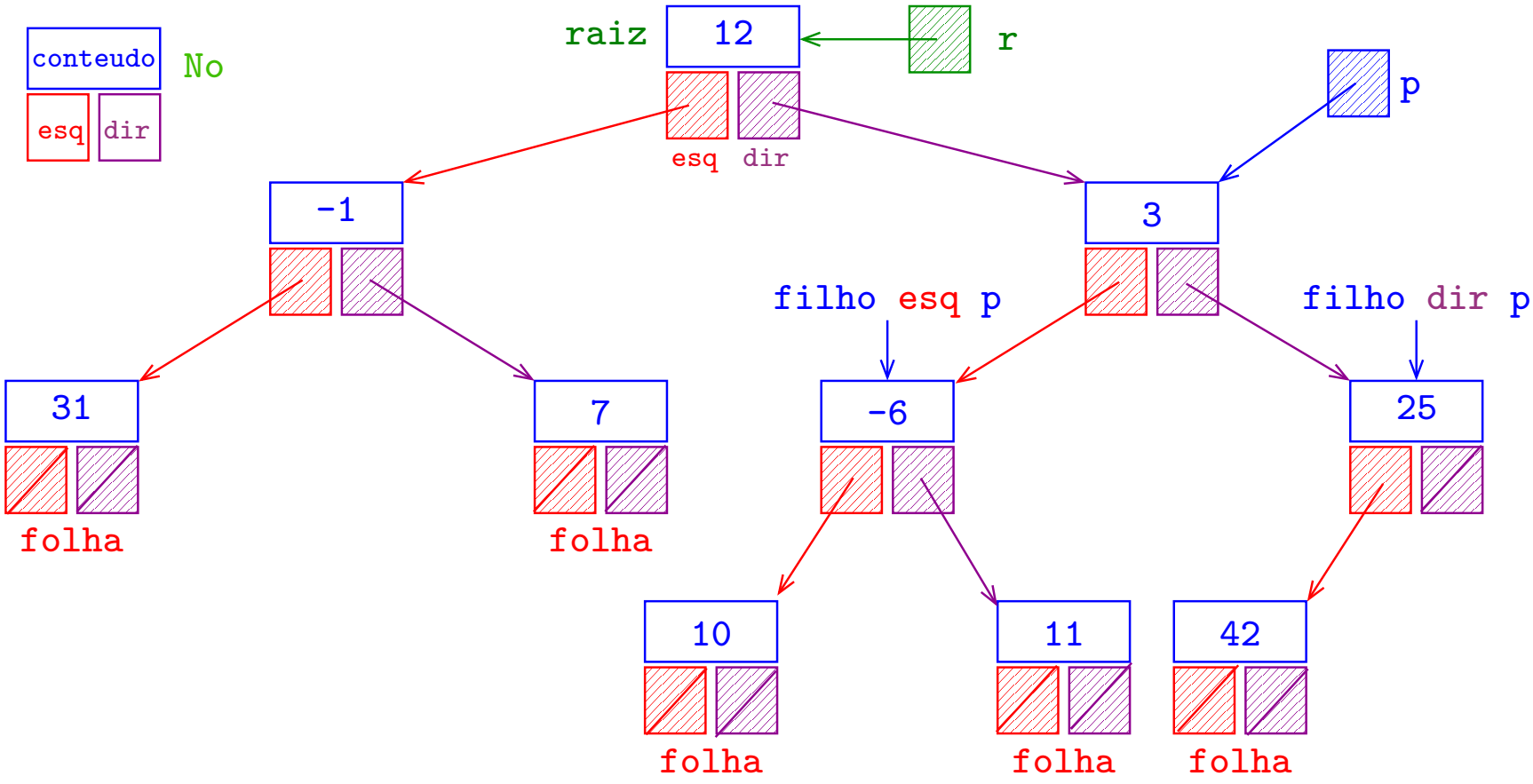
Se `x.dir == y`, então `y` é o **filho direito** de `x`.

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->esq;        p = p->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->esq` é a raiz da subárvore esquerda de `p` e `p->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

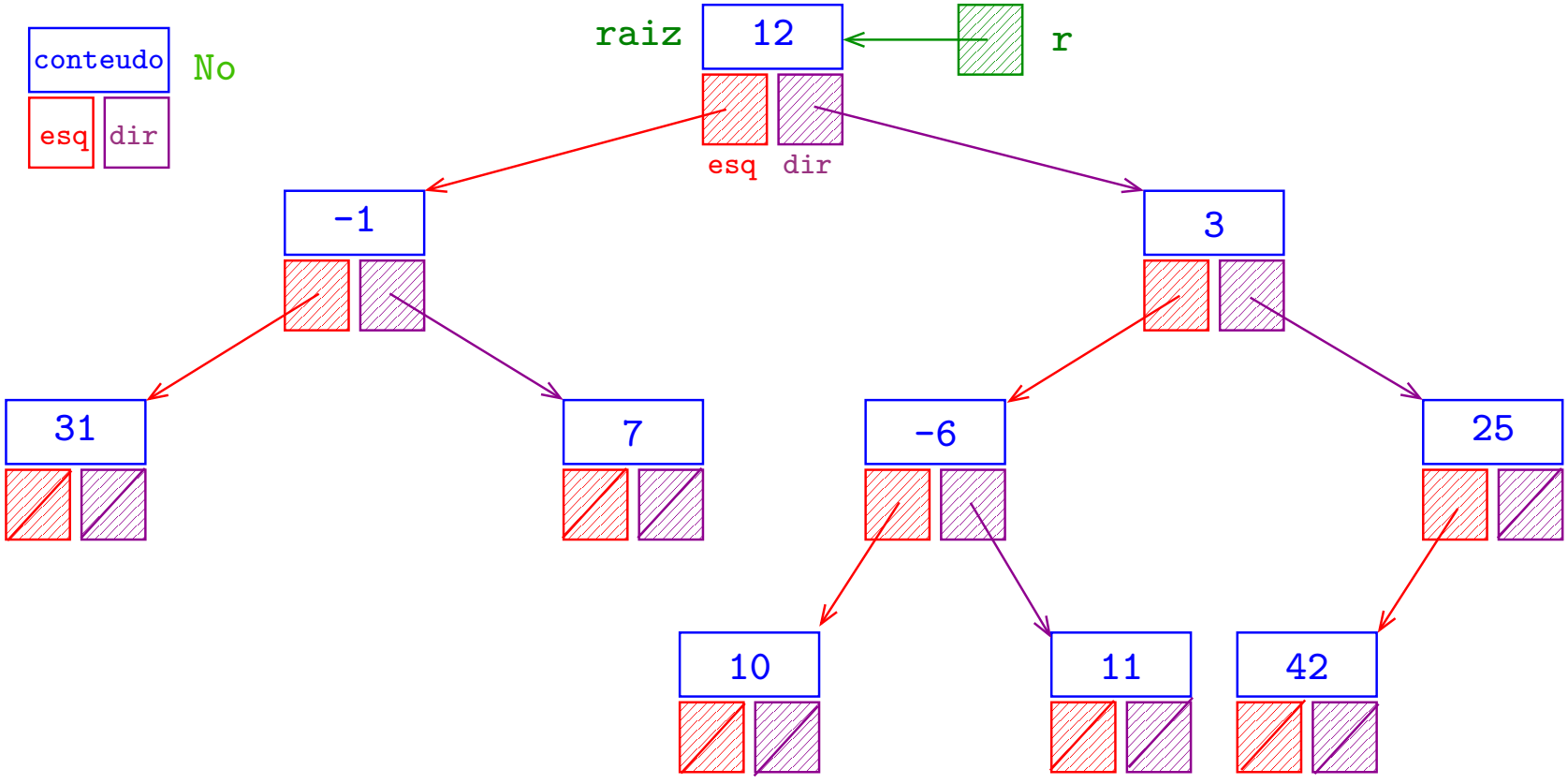
1. a subárvore esquerda da raiz, em ordem e-r-d;
2. depois a raiz;
3. a subárvore direita da raiz, em ordem e-r-d;

```
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

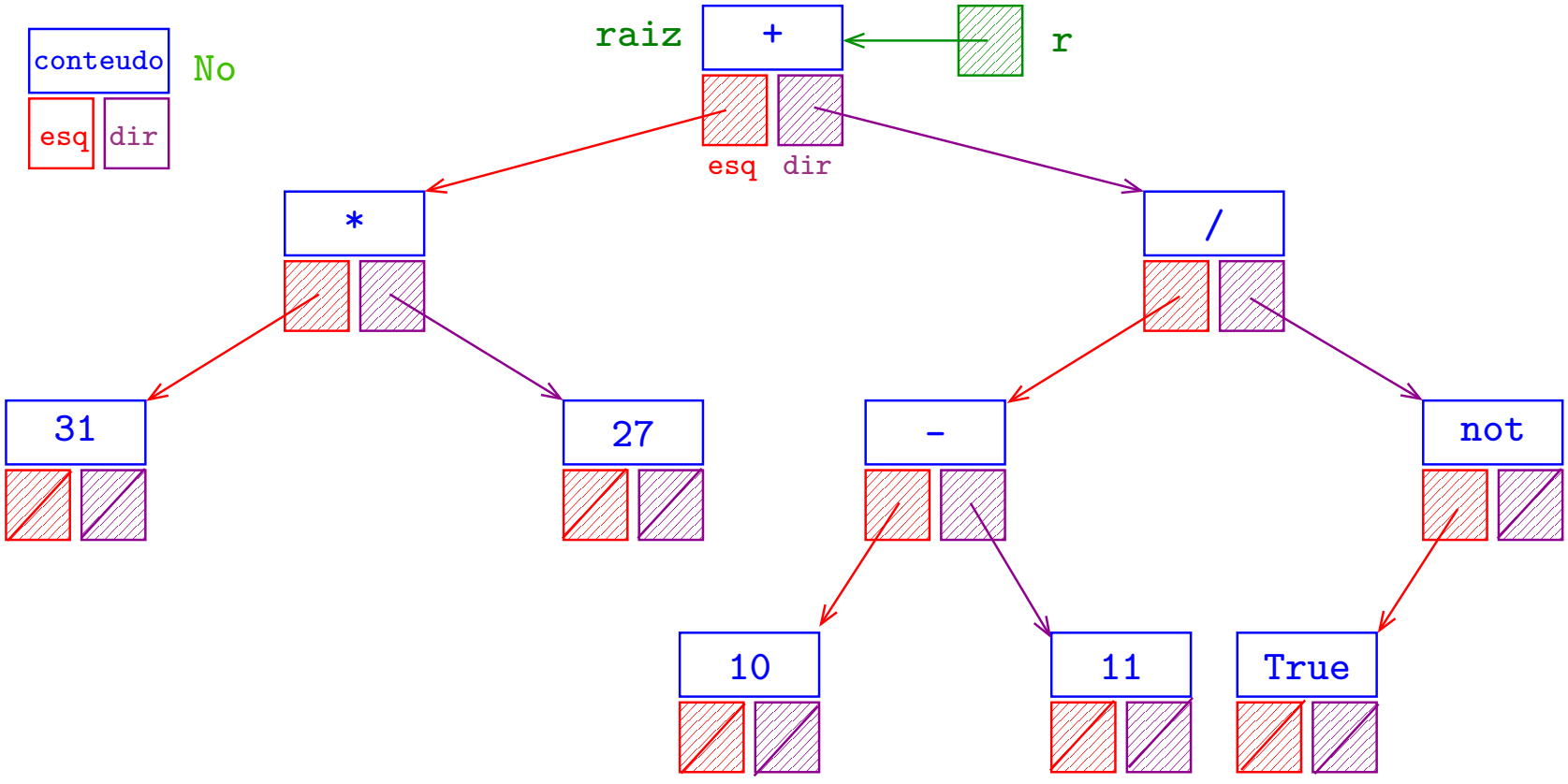


in-ordem (e-r-d): 31 -1 7 12 10 -6 11 3 42 25

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 $\lg(n)$.

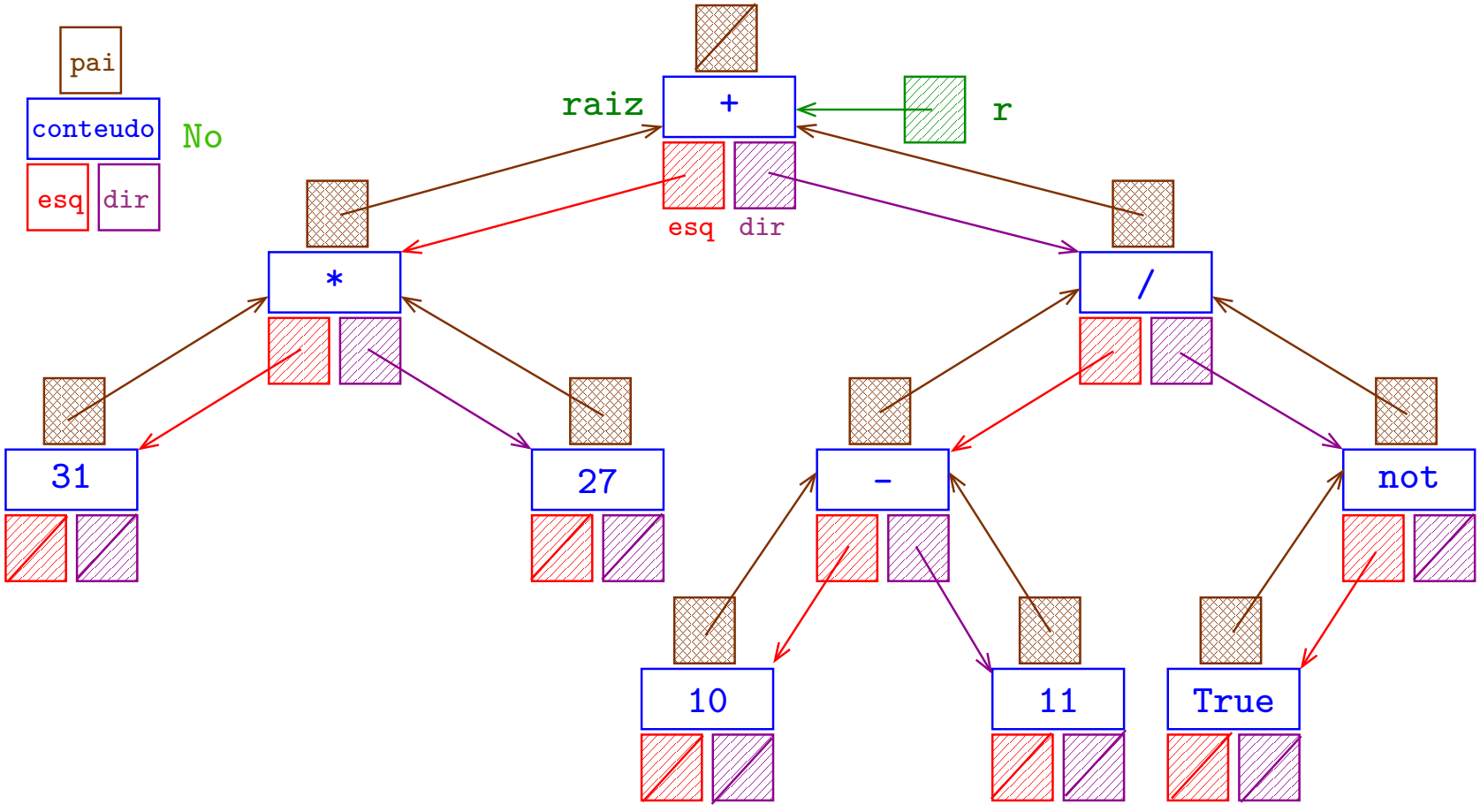
O consumo de tempo dos algoritmos que
manipulam árvores binárias dependem
frequentemente da altura da árvore.

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.

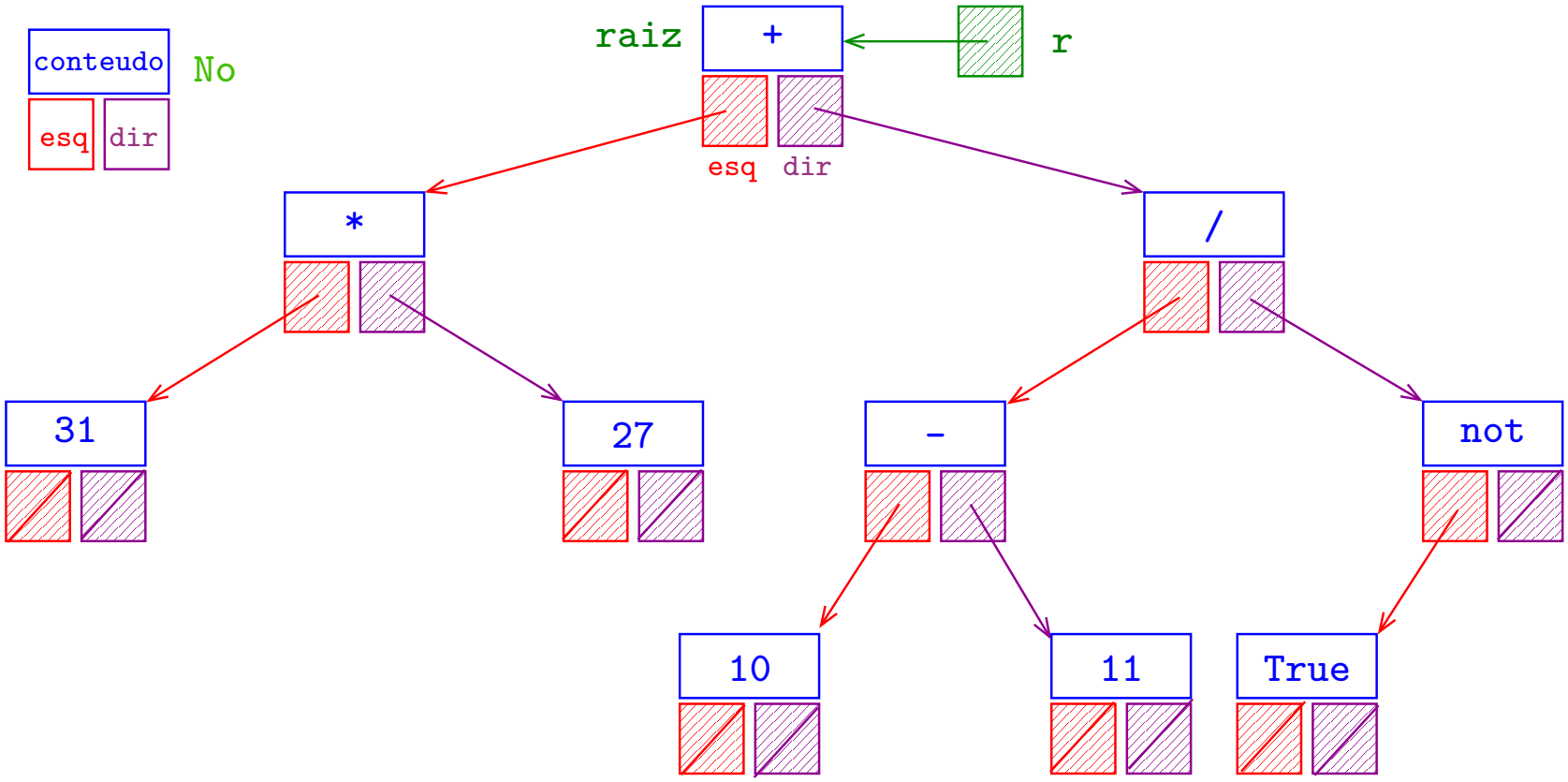
Mais exercícios

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

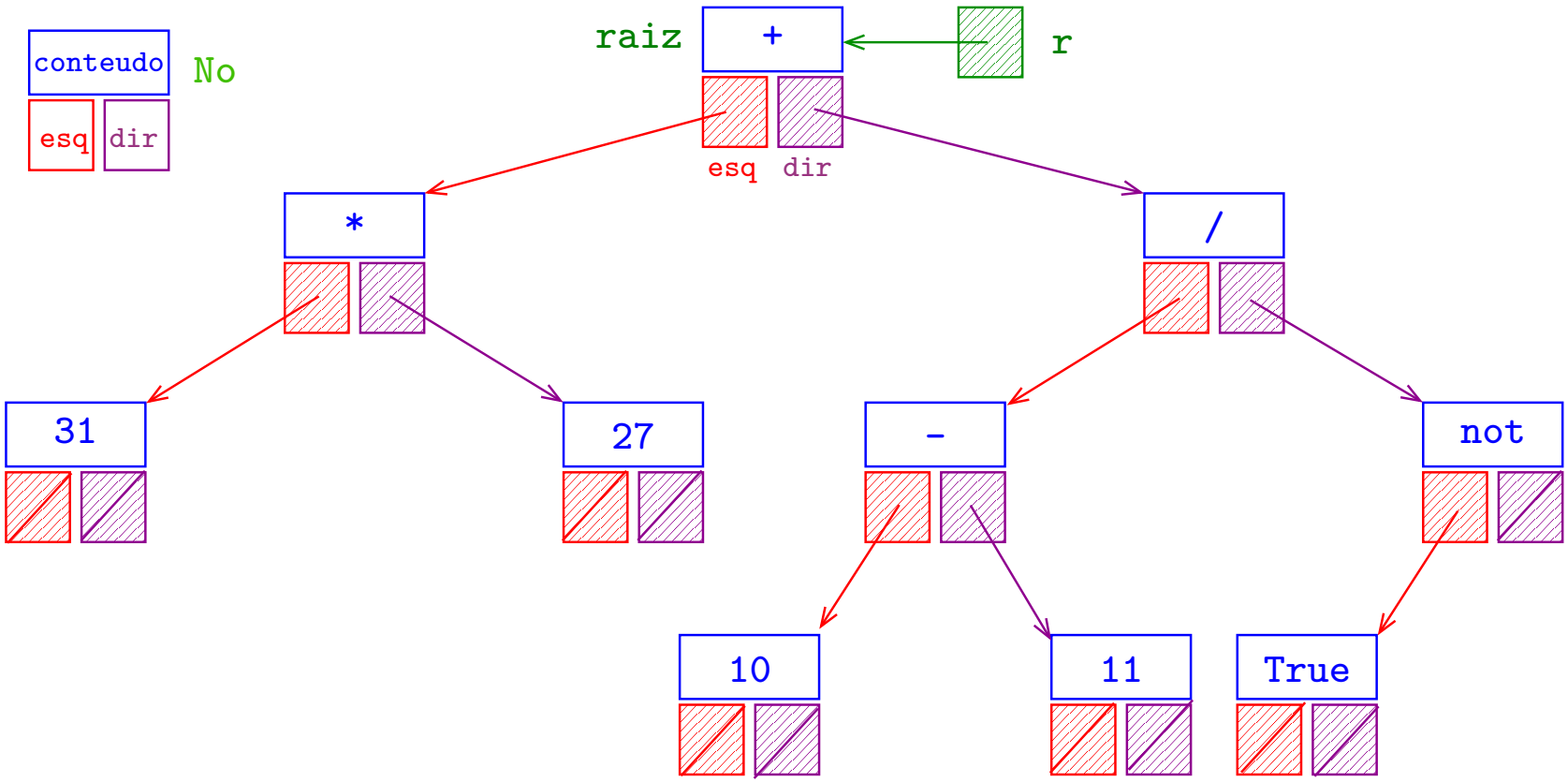
1. devolve o número de nós da árvore.
2. devolve o maior número que aparece na árvore
   ($-\infty$ se a árvore está vazia).
3. espelha a árvore.
4. devolve uma cópia da árvore.
5. libera todos os nós da árvore.

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

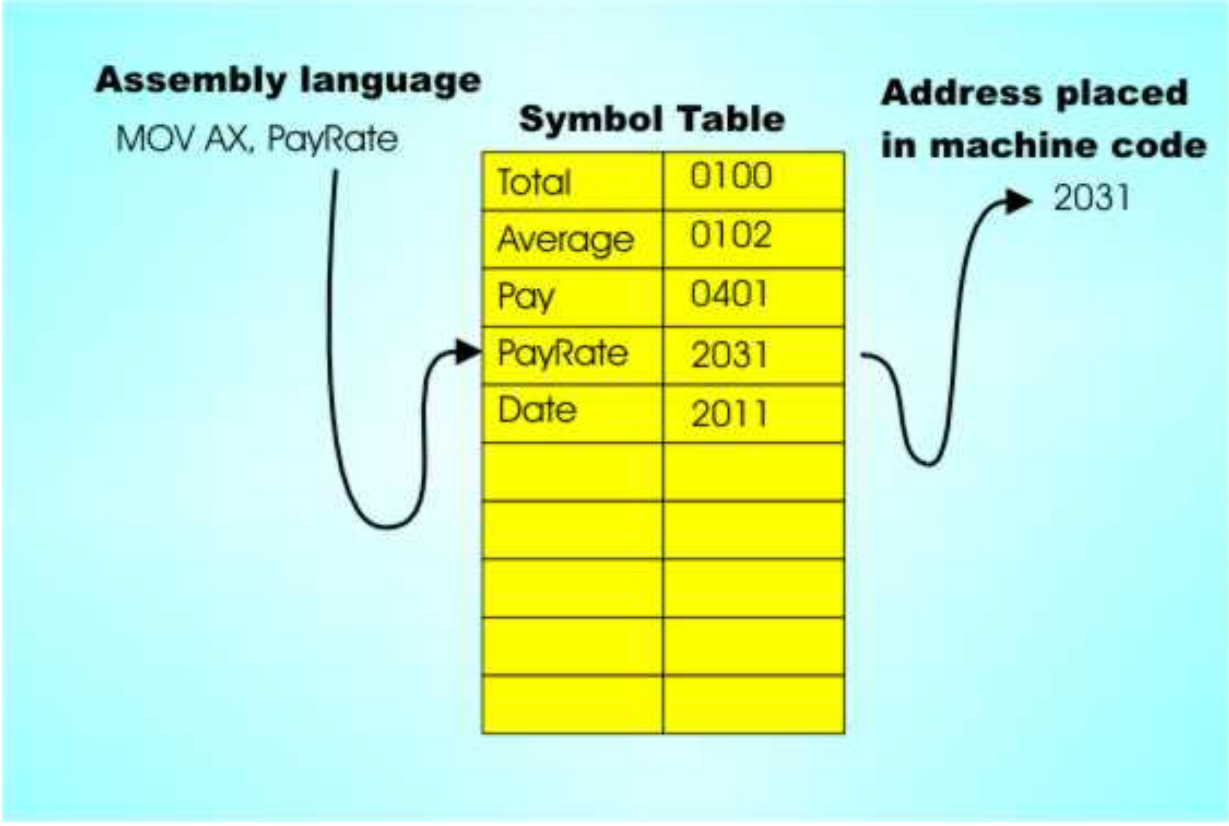
Converte a expressão na notação posfixa.

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.

Próxima aula:
mais uma maneira de organizar uma tabela de símbolos.